알고리즘 9주차 과제

2022110151 이주연

문제 1. Text String과 Pattern String을 입력 받아 1차원 배열에 저장하고, Brute-Force, KMP, Rabin-Karp 알고리즘을 이용해 Text String에서 Pattern String을 모두 찾는 프로그램을 작성하기

1) 알고리즘 코드 설명

코드는 크게 main함수, 매칭을 진행하는 3가지 알고리즘 함수(bruteForce 함수, rabinKarp함수, computeSP로 kmp용 테이블을 만든 후 kmp로 매칭을 진행하는 함수), 그리고 이외에 문자열의 길이를 계산하는 slen함수와 fgets로 받을 때 생기는 '₩n'을 없애는 trim 함수가 있습니다.

먼저, main 함수에서 텍스트(T)와 패턴(P)를 fgets로 입력 받은 후 trim함수를 통해 '₩n'을 없애주고, 1. Brute-Force 2. Rabin-Karp 3. KMP 함수를 호출해 스트링 매칭을 진행합니다(scanf대신 fgets로 받은 이유는 scanf는 공백을 만나면 읽기를 그 자리에서 멈추기 때문에, 공백을 포함한 줄 전체를 배열에 통째로 저장하기 위해 fgets를 활용했습니다.) 이때, <time.h>헤더 파일에 있는 clock()함수를 활용해 시작 시간과 끝난 시간을 측정하고, 이를 바탕으로 수행시간(exeTime)을 계산해 매칭된 결과와 수행시간을 함께 출력해줬습니다.

```
// [string matching] (텍스트에서 패턴을 찾는) 3가지 방법
// 1.  Brute-Force (직선적 알고리즘)--------------------
void bruteForce(const char T[], const char P[]) {
// 입력 : 텍스트 T[]
//        패턴 P[]  (을 입력받아,)
// 텍스트와 패턴의 길이를 각각 계산해 준 후,
    int n = slen(T); // 텍스트(T) 길이 : n
    int m = slen(P); // 패턴(P) 길이 : m

    int i, j;

    printf("1. Brute-Force 결과\n");
    // 텍스트 첫 글자(0)부터 시작해 (i)
    // 패턴이 텍스트 끝을 넘어가지 않는 범위인 n-m까지 검사하며
    for (i = 0; i <= n - m; i++) {
        //패턴의 각 문자 P[j]와 i에서 j만큼 떨어진 텍스트 문자 T[i+j]를 하나씩씩 비교해,
        for (j = 0; j < m; j++) {
            // 만약 둘의 글자가 다르면 빠져나가고,
            if (T[i + j] != P[j])
                break;
        }
        // 만약 글자가 모두 일치할 경우우
        if (j == m)
            printf("패턴이 텍스트의 %d번째부터 나타납니다.\n", i); //매칭이 시작된 위치를 출력하기기
    }
    printf("\n");
}
```

1) Brute-Force 직선적 알고리즘의 경우, 텍스트 첫 글자부터 시작해서 패턴이 텍스트 끝을 넘어가지 않는 범위인 n-m(패턴이 m길이이므로)까지 검사해, 패턴의 P[j]와 i에서 j만큼 떨어진 택스트 T[i+j] 글자를 하나씩 비교해 둘이 다를 경우 break를 통해 빠져 나가고, 만약 글자가 모두 일치할 경우 매칭이 시작된 위치를 출력했습니다.

알고리즘 9주차 과제

2022110151 이주연

```
// 2. Rabin-Karp (해시를 이용한 알고리즘)-----------------
void rabinKarp(const char T[], const char P[]) {
// 입력 : 텍스트 T[]
//        패턴 P[]  (을 입력받아,)
// 텍스트와 패턴의 길이를 각각 계산해 준 후,
    int n = slen(T); // 텍스트(T) 길이 : n
    int m = slen(P); // 패턴(P) 길이 : m
    int h = 1, p = 0, t = 0; // 해시에 사용할 변수들 초기화하기기
    int i, s;

    // 1) D^(m-1) mod Q 를 계산하고
    //   (D=256, Q=101) //여기서 D는 ASCII를 이용해 256으로 설정하였고, Q
    for (i = 0; i < m - 1; i++)
        h = (h * D) % Q;

    // 2) 초기 해시값을 계산하기 (Horner(호너) 방법을 통해 p, t 계산하기)
    for (i = 0; i < m; i++) {
        p = (D * p + P[i]) % Q;
        t = (D * t + T[i]) % Q;
    }

    printf("2. Rabin-Karp 결과\n");
    // 3) 텍스트 각 위치 별로 패턴 매칭시키기
    for (s = 0; s <= n - m; s++) {
        // 만약 hash 값 p와 t가 동일한 경우 세부 패턴 매칭을 진행하기
        if (p == t) {
            int k;
            for (k = 0; k < m; k++) {
                // 만약 다를 경우 중단하고 빠져나가기
                if (T[s + k] != P[k])
                    break;
            }
            // 만약 모두 같을 경우 매칭이 시작된 위치를 출력시키기
            if (k == m)
                printf("패턴이 텍스트의 %d번째부터 나타남\n", s);
        }

        if (s < n - m) {
            // t(s)와 t(s+1)사이의 관계를 이용한 점화식을 통해서 다음 t값 계산
            // e.g 숫자로 된 문자열로 표현했을 때
            // 25436 -> 5436 = 10(25436-2x10^3)+6 여기서 t(s) = 25436, t(s+1) = 5436
            t = (D * (t - T[s] * h) + T[s + m]) % Q;
            if (t < 0)
                t += Q; // 음수 방지용!
        }
    }
    printf("\n");
}
```

2) Rabin-Karp의 경우, 해시를 이용한 알고리즘이므로, 해시에 사용할 변수 h,p,t를 먼저 선언해주었고, 여기서 D와 Q는 문자의 집합 크기 (예를 들어 십진수는 10, 이진수는 2를 의미, 다만 ASCII 기준으로 256이므로 여기서는 256으로 설정했습니다)와 해시 계산용 소수로 101을 설정해주었습니다.

D^(m-1) mod Q 를 계산해 h에 저장하고, 호너 방법으로 초기 해시 값을 계산해 p와 t변수에 저장한 후, s는 0에서 n-범위 내에서 해시 값 p, t를 비교하고, 같을 경우 세부 패턴 매칭을 진행하도록 코드를 작성했습니다. (만약 다를 경우에는 중단하고 빠져나옵니다.) 그래서 모두 같을 경우 매칭이 시작된 위

치가 출력됩니다. 그러나 같지 않을 경우 t(s)와 t(s+1) 사이의 관계를 이용한 점화식 (t = (D * (t - T[s] * h) + T[s + m]) % Q;) 을 통해 다음 t값을 계산하도록 설정했습니다.

```
// 3-1. KMP용 SP 테이블 먼저 계산하기
// 매칭 전 어디서 suffix와 prefix가 일치하는지 알기 위해
// 만든 최대 접두부 테이블
void computeSP(const char P[], int m, int SP[]) {
    int k = -1;  // k: 현재까지 일치한 prefix의 마지막 index를 의미 (어디까지 같은지 나타내는 용도도)
                 // 아직 일치한 문자가 없어 돌아갈 위치도 없기 때문에 -1로 표시
    int j;       // j: 패턴 index
    SP[0] = -1;  // 첫글자에 proper prefix가 없으므로 -1로 초기화
    // 패턴의 1번 인덱스부터 시작해서 패턴 길이전까지
    for (j = 1; j < m; j++) {
        // 만약 불일치 하는 경우, k를 SP[k]값으로 되돌려 더 작은 prefix길이 범위내에서 일치하는게 있는지 다시 확인
        while (k >= 0 && P[k + 1] != P[j])
            k = SP[k];
        // 만약 일치하는 경우, prefix의 길이를 하나 늘리기기
        if (P[k + 1] == P[j])
            k++;
        // 현재 일치하는 길이(k)를 SP 배열에 저장
        SP[j] = k;
    }
}

// 3-2. KMP 매칭 알고리즘 --------------------------------
void kmp(const char T[], const char P[]) {
    int n = slen(T); // n : 텍스트 길이
    int m = slen(P); // m : 패턴 길이
    int *SP = (int *)malloc(sizeof(int) * m); // SP테이블을 먼저 동적 할당해주기기
    int i, k;

    // SP 테이블을 만들어 계산하기
    // 그래야 suffix/prefix match를 해 불필요한 비교 반복을 없앨 수 있음음
    computeSP(P, m, SP);
    printf("3. KMP 결과\n");
    k = -1; // 현재 일치하는 패턴의 최종 index

    // 텍스트 전체 한 글자씩 비교하며,
    for (i = 0; i < n; i++) {
        // 만약 글자가 불일치할 경우 S[k]만큼 다시 되돌아가 비교하고,
        while (k >= 0 && P[k + 1] != T[i])
            k = SP[k];
        // 만약 글자가 일치하는 경우에는 k를 하나 더 늘리며,
        if (P[k + 1] == T[i])
            k++;
        // 전체 패턴이 일치하는 경우에는 위치를 출력하고, k를 재설정하기
        if (k == m - 1) {
            printf("패턴이 텍스트의 %d번째부터 나타남\n", i - m + 1);
            k = SP[k];
        }
    }
    printf("\n");
    free(SP); //malloc을 통해 할당한 동적 메모리를 해제하기기
}
```

3) KMP 알고리즘의 경우, computeSP(최대 접두부 테이블을 계산하는 함수)와 kmp를 진행하는 함수로 이루어져 있습니다. kmp함수가 호출되면 텍스트 T와 패턴 P가 입력으로 들어와 동적 할당된 sp와 함께 computeSP(테이블 생성 함수)를 먼저 수행합니다. 여기서 k는 지금까지 일치한 접두부 길이를 나타내는 변수고, SP[]는 prefix와 suffix가 일치하는 길이를 저장하는 배열입니다. 아직 일치한 글자가 없기 때문에 k를 -1로 설정하고 SP[0]도 -1로 설정했고(첫 글자부분엔 proper prefix가 없으므로), j 1부터 m-1까지 P[k + 1]와 P[j]를 비교해서 만약 다를 경우 k를 SP[k]값으로 되돌려서 더 작은 prefix 길이 범위 내에서 다시 비교하고, 만약 P[k + 1]와 P[j]이 비교해서 같을 경우에는 k를 하나 늘려 같은 글자가 하나 더 늘어났음을 알려주고, 현재 일치하는 길이를

2022110151 이주연

SP배열에 저장해줍니다. 이를 바탕으로 SP테이블을 만든 후, kmp 함수 내에서 텍스트를 한 글자씩 비교하며 만약 글자가 불일치할 경우에는 SP[k]만큼 다시 돌아가서 비교하도록, 만약 글자가 같을 경우에는 k를 하나 더 늘리도록, 따라서 전체 패턴이 같을 경우에는 나타난 시작 위치를 출력하고 k를 재설정하도록 코드를 작성했습니다.

2) 알고리즘 결과와 수행시간 출력 화면

```
2022110151@linuxserver1:~$ gcc stringMatchFin.c -o stringMatching
2022110151@linuxserver1:~$ ./stringMatching
Text String: A STRING SEARCHING EXAMPLE CONSISTING OF A GIVEN PATTERN STRING
Pattern String: STRING
1. Brute-Force 결과
패턴이 텍스트의 2번째부터 나타납니다.
패턴이 텍스트의 57번째부터 나타납니다.

-> Brute-Force 수행시간: 0.000117초

2. Rabin-Karp 결과
패턴이 텍스트의 2번째부터 나타남
패턴이 텍스트의 57번째부터 나타남

-> Rabin-Karp 수행시간: 0.000075초

3. KMP 결과
패턴이 텍스트의 2번째부터 나타남
패턴이 텍스트의 57번째부터 나타남

-> KMP 수행시간: 0.000051초
2022110151@linuxserver1:~$
```

텍스트와 패턴 스트링을 입력받아 3가지 알고리즘으로 매칭을 진행했을 때, 세 알고리즘 모두 index 2번과 index 57번에서 패턴 스트링을 정확히 찾아내는 모습을 알 수 있었습니다. 이때, 세 알고리즘 수행시간에서 차이가 발생했는데, Brute-Force의 경우 직선적으로 일일이 문자비교를 수행하기 때문에 가장 느리게 시간이 출력되었고, Rabin-Karp의 경우 해시 값으로 빠르게 후보만 걸러내 문자를 비교했기 때문에 BruteForce보다 빠르게 수행되었습니다. SP 테이블을 활용한 KMP의 경우 prefix/suffix 매칭을 통해 불필요한 비교를 최소화했기 때문에 실행했을 때 가장 빠르게 수행된 것을 알 수 있었습니다.

알고리즘 9주차 과제

2022110151 이주연

문제 2. 스트링 매칭 응용 문제

- input.txt에서 랜덤한 A/C/G/T로 이루어진 길이 m 패턴을 모두 찾아 각 인덱스(인덱스는 0부터 시작)를 ouput.txt 파일에 출력으로 저장하는 프로그램을 직선적 방법(A) 과 그 이외의 알고리즘을 사용하여 구현(B)하기
- n을 1000부터 10배씩 증가해가면서, 각각에 대해 m을 5에서 30까지 적당히 (예를 들어 m=5,10,15,20,40) 늘려가면서 수행시간(시간단위는 비교가 보여질 수 있는 정도로 적절하게 각자 정할 것)을 측정하기

1) 프로그램 설명

```c
int main(int argc, char *argv[]) {
    int i, j;
    // 프로그램이 실행될 때, 1. 입력파일, 2. 패턴, 3. 알고리즘 3개의 인자 필요
    // 만약, 인자 개수가 맞지 않으면(인자가 부족한 경우) 오류 출력하도록 에러 처리
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input.txt PATTERN ALGORITHM\n", argv[0]);
        return 1;
    }

    // 다양한 크기 n을 가지고 실행할 것 ( 직선적 알고리즘과 다른 알고리즘을 비교하기 위해)
    //  n_values: 파일크기를 나타내는 배열 (1000, 10000, 100000, 1000000, 10000000)
    //  두 알고리즘의 수행시간이 도저히 유의미하게 출력되지 않아 이렇게 표현해봤습니다...
    //  m_values 배열은 패턴 길이를 나타냄 (5, 10, 15, 20, 30)
    int n_values[] = {1000, 10000, 100000, 1000000, 10000000};
    int m_values[] = {5, 10, 15, 20, 30};

    // 출력 파일을 위한 포인터 변수
    FILE *output_file;

    // CSV 파일 열기 (결과를 저장하기 위해)
    // -> 이 파일에 각 파일 크기와 알고리즘 실행 시간 정보를 기록
    FILE *csv_file = fopen("C:\\Users\\juyeo\\Desktop\\ACCGTAT\\time_results.csv", "w");  // CSV

    if (!csv_file) {
        perror("(Error) CSV 파일 열기 실패(w모드로)");
        return 1;
    }

    // CSV 파일 헤더 작성하는 부분
    fprintf(csv_file, "n, m, brute_force_time, kmp_time\n");

    // (n_values 배열에 있는 각 n 값에 대해 반복하는데, )
    for (i = 0; i < sizeof(n_values) / sizeof(n_values[0]); i++) {
        // 현재 배열 값을 n에 넣어주고,
        int n = n_values[i];

        // (m_values 배열에 있는 각 m 값에 대해 )
        for (j = 0; j < sizeof(m_values) / sizeof(m_values[0]); j++) {
            // 현재 m_values[j] 값을 m 변수에 저장한 후,
            int m = m_values[j];

            // 1) DNA 문자열을 랜덤으로 생성해서 (by 함수 호출)
            // ( n 크기의 랜덤 DNA 문자열을 생성해 -> input.txt 파일에 저장하기 )
            generate_random_dna("C:\\Users\\juyeo\\Desktop\\ACCGTAT\\input.txt", n);  // 랜덤 DN

            // 2) input.txt 파일을 열어서 읽고,
            FILE *fin = fopen(argv[1], "r"); // 입력 파일을 읽기 모드로 열기:
            if (!fin) { perror("fopen"); return 1; } //파일 열기 실패할 경우 에러처리

            // 3) 파일의 끝까지 이동하여 크기를 측정하고
            fseek(fin, 0, SEEK_END);
            int file_size = ftell(fin);  // 파일 크기 구하기
            rewind(fin);  // 파일 포인터는 처음으로 되돌리기
```

알고리즘 9주차 과제

2022110151 이주연

```
        // 4) 파일에서 텍스트(T)dlfrrl
        // 텍스트를 메모리로 읽어들이고,
        // 끝에 \0을 추가해서 문자열을 종료하기
        char *T = malloc(file_size + 1);  // 메모리 할당
        fread(T, 1, file_size, fin);  // 파일 읽기
        T[file_size] = '\0';  // 문자열 끝에 \0 추가
        fclose(fin); //파일 닫기

        // 5) 패턴 P를 명령줄 인자에서 가져와서
        char *P = argv[2];

        // 6) output.txt 파일 열고 (여기에 매칭된 결과를 저장할 것)
        output_file = fopen("C:\\Users\\juyeo\\Desktop\\ACCGTAT\\output.txt", "w");
        if (!output_file) {
            perror("(Error) output.txt를 w모드로 열기 실패");
            return 1;
        }

        // 7-1) Brute-Force 알고리즘 수행시간 측정하기
        double brute_force_time = measure_time(brute_force, T, P, file_size, m, output_file)

        // 7-2) KMP 알고리즘 수행시간 측정 측정하기
        double kmp_time = measure_time(kmp_search, T, P, file_size, m, output_file);

        printf("Execution time for n=%d, m=%d (Brute-Force): %f seconds\n", n, m, brute_forc
        printf("Execution time for n=%d, m=%d (KMP): %f seconds\n", n, m, kmp_time);  // 실?

        // 8) CSV 파일에 수행시간 기록하기
        fprintf(csv_file, "%d,%d,%f,%f\n", n, m, brute_force_time, kmp_time);  // n, m, 두 ?

        fclose(output_file);  // output.txt 파일 닫기
        free(T);  // 동적 할당된 메모리 해제하기
    }
}

fclose(csv_file);  // CSV 파일 닫기
return 0; //프로그램 종료
```

스트랭 매칭에 있어서 직선적인 방법(brute-force)과 이외의 방법을 비교하기 위해 kmp(sp 테이블을 활용해 불필요한 반복을 줄이는 방식)을 활용했습니다. Brute-force와 kmp알고리즘은 1번에서 사용한 알고리즘과 동일하기 때문에 대신 main함수 진행 방식에 대해 설명드리겠습니다.

먼저 input.txt와 pattern과 algorithm(bf혹은 kmp) 세개의 인자를 체크한 후, 테스트할 파일 크기와 패턴 길이(n,m)을 설정했고 (출력시간이 가시적으로 보이기 하기 위해 여러 크기와 길이를 설정했습니다.), time_results.csv 파일을 열고 헤더를 작성해 기록할 준비를 한 후, 파일 크기 n에 대해 반복하는 범위 내에서, 각 파일 크기에 대한 패턴 길이 m을 반복했습니다. 여기서 DNA 문자열을 랜덤으로 생성해서 input.txt에 저장한 후, input.txt 파일을 열어 텍스트 파일을 읽고 DNA 문자열을 T에 저장했습니다. 그 후 Brute-force와 kmp알고리즘을 실행해서 수행시간을 출력하고 이를 csv파일에 기록하는 방식으로 코드를 완성했습니다.

알고리즘 9주차 과제

2022110151 이주연

2) 결과 출력 (그래프) 및 성능 및 장단점 분석

```
C:\Users\juyeo\Desktop\ACCGTAT>comparing.exe C:\Users\juyeo\Desktop\ACCGTAT\input.txt ACCGT brute_force
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=10 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=15 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=20 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=20 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=30 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=10 (KMP): 0.003000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=15 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=20 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=20 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=30 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=10 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=15 (KMP): 0.001000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=20 (Brute-Force): 0.008000 seconds
Execution time for n=100000, m=20 (KMP): 0.001000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=30 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=5 (Brute-Force): 0.008000 seconds
Execution time for n=1000000, m=5 (KMP): 0.012000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=10 (Brute-Force): 0.002000 seconds
Execution time for n=1000000, m=10 (KMP): 0.008000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=15 (Brute-Force): 0.008000 seconds
Execution time for n=1000000, m=15 (KMP): 0.005000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=20 (Brute-Force): 0.012000 seconds
Execution time for n=1000000, m=20 (KMP): 0.005000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=30 (Brute-Force): 0.007000 seconds
Execution time for n=1000000, m=30 (KMP): 0.008000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=5 (Brute-Force): 0.061000 seconds
Execution time for n=10000000, m=5 (KMP): 0.048000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=10 (Brute-Force): 0.056000 seconds
Execution time for n=10000000, m=10 (KMP): 0.049000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=15 (Brute-Force): 0.055000 seconds
Execution time for n=10000000, m=15 (KMP): 0.050000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=20 (Brute-Force): 0.058000 seconds
Execution time for n=10000000, m=20 (KMP): 0.049000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=30 (Brute-Force): 0.148000 seconds
```

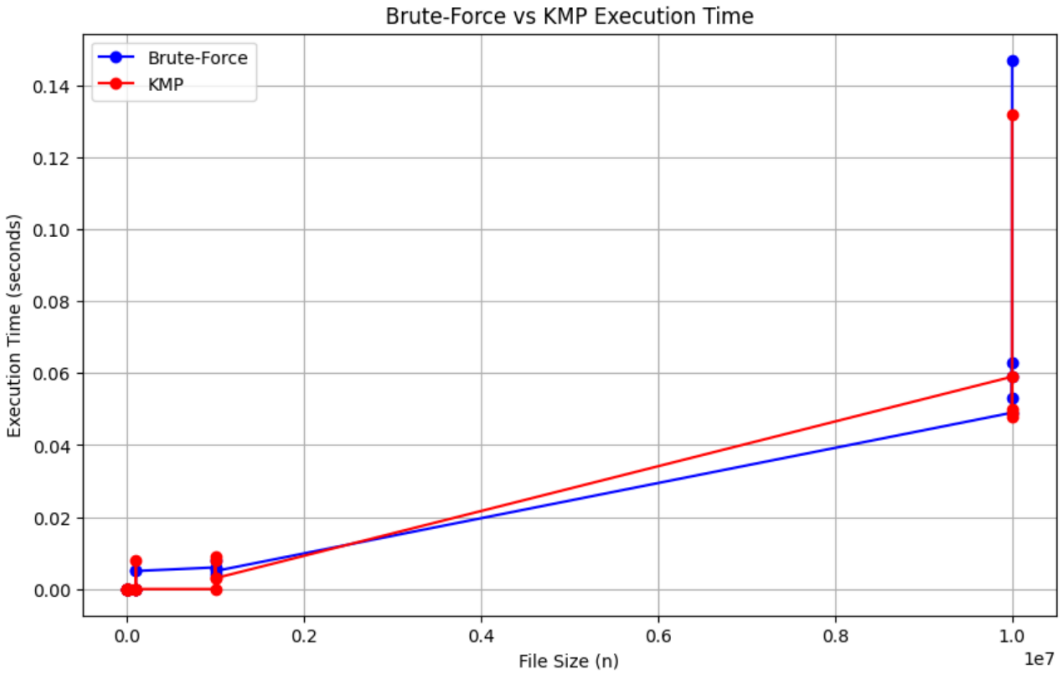
알고리즘 9주차 과제

2022110151 이주연

```
C:\Users\juyeo\Desktop\ACCGTAT>comparing.exe C:\Users\juyeo\Desktop\ACCGTAT\input.txt ACCGT kmp
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=10 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=15 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=20 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=20 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000.
Execution time for n=1000, m=30 (Brute-Force): 0.000000 seconds
Execution time for n=1000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=10 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=15 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=20 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=20 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000.
Execution time for n=10000, m=30 (Brute-Force): 0.000000 seconds
Execution time for n=10000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=5 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=10 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=10 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=15 (Brute-Force): 0.000000 seconds
Execution time for n=100000, m=15 (KMP): 0.008000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=20 (Brute-Force): 0.005000 seconds
Execution time for n=100000, m=20 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 100000.
Execution time for n=100000, m=30 (Brute-Force): 0.005000 seconds
Execution time for n=100000, m=30 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=5 (Brute-Force): 0.006000 seconds
Execution time for n=1000000, m=5 (KMP): 0.000000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=10 (Brute-Force): 0.008000 seconds
Execution time for n=1000000, m=10 (KMP): 0.009000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=15 (Brute-Force): 0.006000 seconds
Execution time for n=1000000, m=15 (KMP): 0.003000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=20 (Brute-Force): 0.004000 seconds
Execution time for n=1000000, m=20 (KMP): 0.008000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 1000000.
Execution time for n=1000000, m=30 (Brute-Force): 0.005000 seconds
Execution time for n=1000000, m=30 (KMP): 0.003000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=5 (Brute-Force): 0.049000 seconds
Execution time for n=10000000, m=5 (KMP): 0.059000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=10 (Brute-Force): 0.059000 seconds
Execution time for n=10000000, m=10 (KMP): 0.049000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=15 (Brute-Force): 0.063000 seconds
Execution time for n=10000000, m=15 (KMP): 0.048000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=20 (Brute-Force): 0.053000 seconds
Execution time for n=10000000, m=20 (KMP): 0.050000 seconds
File 'C:\Users\juyeo\Desktop\ACCGTAT\input.txt' has been created with random DNA string of size 10000000.
Execution time for n=10000000, m=30 (Brute-Force): 0.147000 seconds
```

알고리즘 9주차 과제

2022110151 이주연



처음에는 학교 linux 서버를 활용해 brute-force.c 함수와 kmp.c함수를 각각 따로 작성해 output을 2개를 출력해 이를 바탕으로 그래프를 그려보려 시도했지만, 파일을 로컬 환경에 저장할 수 없다보니, 제출하기 어려워 로컬 환경에서 다시 작업했습니다.

따라서 ACCGTAT폴더를 만들었고, 여기서 comparing.c를 작성해 input.txt이 작성되고 알고리즘(직선적 방법/km p방법)을 활용해 패턴이 분석되고, csv파일로도 저장해 그래프를 그릴 수 있도록 하였습니다.

그래프를 그리는 방식에 대해 고민하였는데, c언어로 그래프를 구현하기에 까다로워 손으로 그리거나 excel로도 표현하는 방식을 고민했지만, 학교 교육과정 교과목 내에서 배운 것을 활용하는 방식이 최선이라 생각해 jupyter notebook을 열어 그래프를 그려보았습니다.

따라서, 기존 brute-force 알고리즘과 kmp 알고리즘을 비교 분석해보면 다음과 같습니다. x축은 파일 크기(n)이고 y축은 수행 시간입니다. 파일 크기가 작을 때는 brute-force 알고리즘 수행시간이 매우 짧지만, 파일크기가 커지면 수행시간이 급격히 증가하는 모습을 볼 수 있었습니다. 일일히 비교하기 때문에 직선적 알고리즘이 O(n * m)의 시간복잡도를 가지게 돼, 패턴길이에 따라 수행시간이 비례하며 증가했고, 특히 n이 1,000,000이상일 때 수행시간이 급격히 늘어나 0.14초로 kmp보다 더 걸리는 것을 알 수 있었습니다. Kmp 의 경우에는 O(n + m)도의 시간복잡도를 가지는데, 그래프로 표현했을 때 특히 n이 1,000,000이상일 경우에 brute-force보다 근소하게 수행시간이 비교적 덜 걸리는 걸 볼 수 있었습니다.

알고리즘 9주차 과제

2022110151 이주연

다만, 최대한 차이를 보이기 위해 clock을 통해 시간을 분석하려 했지만, 주어진 범위내에서 brute-force 알고리즘과 kmp알고리즘 시간이 소수점 단위로 나타나다 보니 차이를 분명하게 발견하기 어려웠고, 로컬 환경의 한계로 최대한 성능차이를 표현하기 어려웠습니다.

최종적으로 정리하자면 Brute-force 알고리즘은 간단하고 직관적이고 구현하기 쉽지만, 시간 복잡도가 O(n * m)으로 텍스트 크기(n)와 패턴크기(m)가 커질수록 성능이 떨어진다는 것을 공부할 수 있었습니다. 큰 입력에 있어서는 효율적이지 않다고 생각합니다. 이와 반대로 다른 알고리즘, 수업시간에 활용한 KMP 알고리즘은 sp테이블을 만들어prefix/suffix 매칭을 통해 불필요한 비교를 줄이기 때문에 O(n + m)의 시간 복잡도를 갖고, 직선적인 방법인 Brute-force 알고리즘에 비해 텍스트의 크기가 클 경우에는 효율적으로 작동해 수행시간이 상대적으로 짧게 나타난다는 것을 공부할 수 있었습니다. 다만, kmp알고리즘을 사용하기 위해서는 sp테이블 계산을 위한 추가 작업이 필요하고, 구현이 brute-force 보다는 까다롭기 때문에 이런 부분에 있어서는 단점이라고 말할 수 있습니다. 더욱이 작은 텍스트에 있어서는 brute-force가 더 빠를 수 있기 때문에 텍스트 크기가 큰지 작은지 상황에 따라 그에 맞는 알고리즘 선택이 중요함을 알 수 있는 시간이었습니다.