Assignment02

목차

1. 전체 프로그램 개요

이번 k DNA sequences alignment 과제에서는 크게 3가지의 파트로 나뉜다. hw2_ input.txt로부터 염기 서열을 읽어오는 부분, 겹치는 부분을 찾아내는 부분, hw2_output.txt 파일을 생성하여 결과를 텍스트 파일로 저장하는 부분이다. 각 부분의 역할은 다음과 같다.

먼저, 염기 서열을 읽어오는 부분이다. fopen() 함수를 통하여 hw2_ input.txt 파일을 열어낸 후, 맨 첫 줄에 있는 전체 염기 서열의 개수를 입력받는다. 이후 $ 표시를 읽어내어 전체 개수를 나타낸 이후의 다음 파트인 각각의 염기 서열이 나타나는 파트로 넘어감을 확인한 이후, 각각의 염기 서열을 fprintf() 함수를 이용하여 입력받아 배열 안에 저장한다.

다음으로는 염기 서열에서 겹치는 부분을 찾아내는 부분이다. 이 부분에서는 겹치는 부분을 찾기 위해 lcs() 함수를 선언하여, 입력으로 전체 스트링과 스트링의 개수를 주면 그 입력 스트링들의 최대로 겹치는 부분을 문자열로 리턴해 주는 함수를 만든다. 구현을 위한 알고리즘은 수업 교안에 나온 lcs 알고리즘을 사용하였다. 이러한 lcs() 함수를 이용하여, 전체 염기 서열의 최대로 겹치는 부분을 찾아낸다. k의 값이 2부터 5 사이이므로, 입력 염기 서열이 2개부터 5개까지 들어오는 경우의 수를 각각 나눠서 수행하도록 함수를 구성하였다.

마지막으로는, lcs() 함수를 통하여 찾아낸 염기 서열을 가지고 전체 결과 값을 저장하는 부분이다. 이 저장을 하기 위해서는 전처리 작업 부분이 존재한다. 우선적으로, 각각의 염기 서열 별로 lcs의 결과 문자열(이 이후부터 편의상 'lcsResult'라 칭하겠다.) 부분에 해당하는 인덱스를 구한다. 같은 염기가 존재할 경우 기준은 더 먼저 나온 값을 기준으로 한다. 이렇게 구한 인덱스들을 이 이후부터 편의상 'lcs 포인트'라고 칭하겠다.

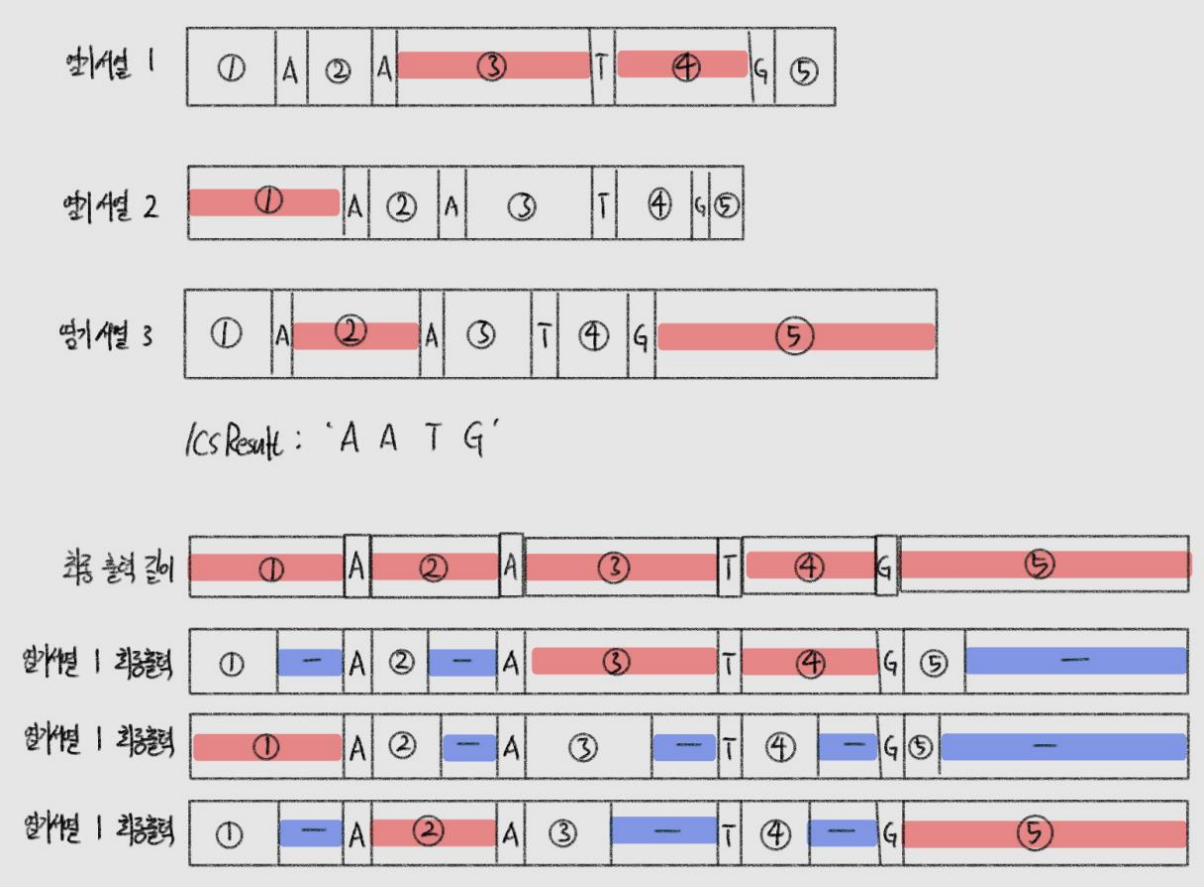
예) 첫 번째 염기 서열 – ATTGCCATT

lcsResult – ATCCAT

lcs 포인트 – 0, 1, 4, 5, 6, 7

이렇게 주어진 입력 염기 서열 별로(k의 값에 따라 변화, 최소 2개 ~ 최대 5개) lcs 포인트들을 구하여 배열을 정한다. 이 경우 이를 저장한 배열의 크기는 (총 입력 염기 서열의 개수) * (lcsResult의 길이) 가 나오게 된다. 이 lcs 포인트를 이용하여 최종 출력의 기반을 마련한다.

출력 폼에서는 모든 염기 서열의 길이가 통일되어 있어야 한다. 각 줄 별로 같은 길이를 가지고 있어야 하며, 각 lcsResult가 나타나는 부분이 동일한 위치에 있어야 하며 길이를 맞추기 위하여 빈 공백이 생길 경우 -문자로 채워주어야 한다. 줄 별로 통일된 최종적인 길이를 구하기 위하여 앞에서의 lcsResult를 구한 것이다. 최종적인 길이를 구하기 위한 방법은 다음과 같다.



각 lcs 포인트를 기준으로 각각의 염기서열을 분할한다. 이 때 염기 서열 별로 각각 분할된 방식이 전부 lcsResult의 길이와 lcs 포인트를 기준으로 분할되어 있기 때문에, 각각의 염기 서열의 분할된 개수는 같고, 각각의 대응되는 분할 파트들이 존재한다. 위에서의 예시를 통해 설명을 하면 다음과 같다. lcsResult가 'AATG'이기 때문에 이 4개의 포인트를 기준으로 각각의 염기 서열을 분할하여 염기 서열 당 5개의 서브 파트가 나오게 된다. 이후 전부 동일하게 대응되는 파트들이 염기 서열별로 대응되게 한다. 위의서의 예시로는 1번 ~ 5번까지의 각 서브파트가 존재하게 되는 것이다. 이후 같은 번호로 대응되는 서브 파트끼리 길이를 비교한다. 나중의 최종 출력의 경우 이 각 서브 파트들 중 최대 길이를 갖는 서브 파트들의 합과 여기에 lcsResult의 합을 더한 결과가 곧 최종 길이가 된다. 이 최대 서브 파트는 위의 그림에서는 빨간 색으로 칠해진 부분과 같다. 이렇게 될 경우 나중에 최종 출력에서는 각 대응되는 파트 부분 출력 시 우선적으로 본인의 서브 파트 내용을 출력하고 서브 파트의 길이가 최대 서브 파트의 길이가 될 때까지 -를 출력하여 길이를 맞춰주면 된다. 이후 각 lcsResult의 값을 각 서브 파트의 사이사이마다 출력하게 되면 모든 겹치는 부분의 출력 위치가 동일해지고, 최종 길이 또한 동일해진다. 이후 마지막 줄을 새로 만들어 각 lcsResult에 포함되는 값들 또한 그 통일된 위치에 *을 출력하고, 이 결과를 hw2_output.txt 파일을 새로 만들어 저장함으로써 전체 프로그램이 종료된다.

2. main()

메인 함수의 경우 앞에서 말한 전체 프로그램 흐름에 의해 실행되는 구조를 가지고 있다. 먼저, 입력을 받는 부분을 코드로 구현한 부분은 다음과 같다.

```
// variables
int num, lcslen, len, ATGClen, total, temp, temp2, temp3;
int* maxPoint;
int** points;
char dollar[5];
char* lcsResult;
char** ATGC;

// open file
FILE* inputFile = fopen("hw2_input.txt", "r");
FILE* outputFile = fopen("hw2_output.txt", "w");

// read file
fscanf(inputFile, "%d", &num);
fscanf(inputFile, "%s", dollar);

ATGC = (char**)malloc(sizeof(char*) * num);
for (int i = 0; i < num; i++) {
    ATGC[i] = (char*)malloc(sizeof(char) * 121);
    fscanf(inputFile, "%s", ATGC[i]);
    //printf("%s\n", ATGC[i]);
}
```

입력 파일을 fopen() 함수를 통해 열어낸 다음 파일 디스크립터를 받고, 이를 이용하여 ATGC라는 2차원 배열에 전체 염기서열의 개수만큼 입력받는다. 이 때 조건에 2<= k <= 5, 1 <= n <= 120이라는 조건이 붙어있기 때문에 최대 길이 + '₩0'을 고려한 121의 길이의 문자열을 k(코드 상에서는 num)개만큼 동적 할당받을 수 있도록 malloc() 함수를 이용하여 메모리를 할당받는다. 이 결과가 담길 ATGC는 총 num의 길이를 가지는 배열이고, 각각의 element들은 121의 길이를 같는 문자열들이다. 이 경우 전체 시간 복잡도는 전체 입력의 크기에 비례하는 O(k * n)이다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이를 의미한다.)

다음으로 전체 흐름 부분에서 설명한 lcsResult를 구하는 부분이다. 코드로 구현한 형태는 다음과 같다.

```
// find lcs
lcsResult = (char*)malloc(sizeof(char) * 121);
lcsResult = lcs(ATGC, num);
lcslen = strlen(lcsResult);
//printf("LCS: %s\n", lcsResult);
//printf("%d\n", lcslen);
```

입력받은 염기서열을 저장하고 있는 ATGC와 각각의 염기 서열의 개수를 저장하고 있는 num을 함수에 인자로 넣어주어 최종 lcsResult를 구하게 된다. 이 부분의 시간 복잡도는 뒤의 3번 lcs 함수 설명 부분에서 설명할 내용에 의해, k개의 입력에 대해 길이 최대 n인 스트링을 비교하는 과정이므로 $O(n^k)$이다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이를 의미한다.)

이후의 작업들은 이 lcsResult의 길이가 0이 아닐 경우에만 실행하게 된다. 먼저, 앞에서 설명한 lcs 포인트들을 구하는 부분은 코드로 구현하면 다음과 같다.

```
if (lcslen) {
    // find lcs points
    points = (int**)malloc(sizeof(int*) * num);
    for (int i = 0; i < num; i++) {
        points[i] = (int*)malloc(sizeof(int) * lcslen);
        ATGClen = strlen(ATGC[i]);
        for (int j = 0, temp = 0; (j < ATGClen) && (temp < lcslen); j++) {
            if (ATGC[i][j] == lcsResult[temp]) {
                points[i][temp++] = j;
                //printf("%d ", points[i][temp - 1]);
            }
        }
        //printf("\n");
    }
```

lcs 포인트를 담을 이차원 배열 points를 생성하게 된다. 이 배열의 크기는 num * (lcsResult의 길이)에 해당한다. 전체 염기 서열을 순회하면서, 동시에 lcsResult를 순회하며 두 값이 일치하는 부분의 인덱스를 저장하게 된다. 이 경우 총 순회 횟수는 (총 염기 서열 개수) * (각 염기 서열의 길이) * (lcsResult의 길이)의 횟수에 해당하게 된다. 이 경우 lcsResult의 경우 입력 염기 서열의 길이인 n에 선형으로 비례하게 되므로(n보다 작거나 같기 때문) 이 부분의 시간 복잡도는 O(k * n * n)에 해당하게 된다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이를 의미한다.)

다음의 부분은 이렇게 구한 lcs 포인트들을 가지고 출력을 위한 각 줄에서 LcsResult의 각 요소들이 출력될 위치를 구하는 과정이다. 각 포인트를 기준으로 서브 파트를 계산한 후, 그 중에서의 max 값을 취하게 된다. 이러한 값을 담는 lcsResult의 크기를 갖는 배열 maxPoint를 구하기 위해서는 다음과 같은 식이 성립하게 된다.

maxPoint[i] = MAX(maxPoint[i], points[j][i]) -> i = 0일 때

MAX(maxPoint[i], (points[j][i] - points[j][i - 1] + maxPoint[i - 1])) -> 그 외의 경우

이를 코드로 구현한 방식은 다음과 같다.

```
// find maxPoint
maxPoint = (int*)malloc(sizeof(int) * lcslen);
for (int i = 0; i < lcslen; i++) maxPoint[i] = 0;
for (int i = 0; i < lcslen; i++) {
    for (int j = 0; j < num; j++) {
        if (i == 0) maxPoint[i] = MAX(maxPoint[i], points[j][i]);
        else maxPoint[i] = MAX(maxPoint[i], (points[j][i] - points[j][i - 1] + maxPoint[i - 1]));
    }
    //printf("%d ", maxPoint[i]);
}
//printf("\n");

// find total length for each line
total = maxPoint[lcslen - 1];
for (int i = 0; i < num; i++) {
    total = MAX(total, (maxPoint[lcslen - 1] + strlen(ATGC[i]) - points[i][lcslen - 1]));
}
```

이 경우 시간 복잡도는 (points의 크기) * (lcsResult의 길이)에 비례하게 되므로 O(k * n * n)에 해당하게 된다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이를 의미한다.)

이후 결과를 결과 파일 안에 출력하는 부분이다. 코드는 다음과 같다.

```
// write result
for (int i = 0; i < num; i++) {
    len = strlen(ATGC[i]);
    for (int j = 0, temp2 = 0, temp3 = 0; j < len; ) {
        if (temp2 == 0) {
            for (int k = 0; k < points[i][temp2]; k++, j++, temp3++) fprintf(outputFile, "%c", ATGC[i][k]);
            for ( ; temp3 < maxPoint[temp2]; temp3++) fprintf(outputFile, "-");
            temp2++;
        }
        else if (temp2 == lcslen) {
            for (int k = points[i][temp2 - 1]; j < len; k++, j++, temp3++) fprintf(outputFile, "%c", ATGC[i][k]);
            for ( ; temp3 < total; temp3++) fprintf(outputFile, "-");
        }
        else {
            for (int k = points[i][temp2 - 1]; k < points[i][temp2]; k++, j++, temp3++) fprintf(outputFile, "%c", ATGC[i][k]);
            for ( ; temp3 < maxPoint[temp2]; temp3++) fprintf(outputFile, "-");
            temp2++;
        }
    }
    fprintf(outputFile, "\n");
}
for (int i = 0, temp2 = 0; i <= maxPoint[lcslen - 1]; i++) {
    if (i == maxPoint[temp2]) {
        fprintf(outputFile, "*");
        temp2++;
    }
    else fprintf(outputFile, " ");
}
```

전체 염기를 순회하면서 출력을 진행하고, 각 lcs 포인트를 기준으로 출력 -> 비는 부분에 '-' 출력 -> lcsResult의 element 출력을 반복한다. 이후 마지막에 lcs 포인트마다 *을 출력하고 출력을 종료한다. 이 경우 시간 복잡도는 전체 데이터를 순회하며 출력하는 과정이므로 O(k * n + c)에 속하게 된다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이, c는 -를 출력하기 위한 추가적인 overhead 상수를 의미한다.)

마지막으로, 메모리 해제 및 파일 디스크립터를 닫으며 전체 프로세스를 종료하는 부분이다. 코드는 다음과 같다.

```
// memory deallocation
free(lcsResult);
free(maxPoint);
for (int i = 0; i < num; i++) free(points[i]);
free(points);
for (int i = 0; i < num; i++) free(ATGC[i]);
free(ATGC);

// close file
fclose(inputFile);
fclose(outputFile);
```

다음과 같이 전체 프로그램이 main 함수에 의해 진행이 되며, 시간 복잡도는 이 중에서 가장 지배적으로 이루어지는 lcsResult를 구하는 부분에 의해 결정되고, 따라서 $O(n^k)$이다. (여기서의 k는 염기 서열의 개수를 의미하고, n은 각각 염기 서열의 최대 길이를 의미한다.)

3. char* lcs(char** str, int num)

lcs 함수의 파라미터는 다음과 같다. 비교를 위한 입력 스트링(즉, 염기 서열)들을 가지고 있는 2채원 char 배열 str과 그 해당 스트링들의 총 개수 num이다. 또한 이 함수의 최종 파라미터 값은 char* 타입의 문자열로, 겹치는 부분의 문자들을 나열한 문자열이다. 이 함수는 최대로 겹치는 부분의 길이를 구하는 파트와, 실제로 그 문자열을 구하는 파트의 두 파트로 나뉜다.

먼저, 최대 길이를 구하는 방법이다. num차원 행렬(테이블)을 만들어, Dynamic Programming의 방식을 통해 구하는 방법을 택하였다. 여기서 입력의 개수가 2개일 경우, 최종 문제를 '최대 길이'라고 정하고 이를 구하기 위한 테이블 값 c[i][j]를 '입력 스트링 x의 i번째까지의 부분 스트링과 입력 스트링 y의 j번째까지의 부분 스트링을 가지고 계산한 두 스트링의 겹치는 부분의 최대 길이'라고 정의하면, 이를 구하기 위한 하위 문제는 다음과 같이 정할 수 있다.

c[i][j] = c[i – 1][j – 1] + 1 -> x[i] == y[j]일 때

max(c[i - 1][j], c[i][j - 1]) -> 그 외의 경우

num이 3, 4, 5일 경우에도 점화식은 같은 방식으로 확장된다. 단순히 차원이 하나 늘어난 경우에 해당하게 되고, 고려해야 할 요소만 하나씩 증가하는 형태가 된다. 따라서 입력 스트링의 개수가 증가함에 따라 점화식은 다음과 같이 확장될 수 있다.

num = 3일 때

c[i][j][k] = c[i – 1][j – 1][k – 1] + 1 -> x[i] == y[j] == z[k]일 때

max(c[i - 1][j][k], c[i][j - 1][k], c[i][j][k – 1]) -> 그 외의 경우

num = 4일 때

c[i][j][k][l] = c[i – 1][j – 1][k – 1][l - 1] + 1 -> x[i] == y[j] == z[k] = w[l]일 때

max(c[i - 1][j][k][l], c[i][j - 1][k][l], c[i][j][k – 1][l], c[i][j][k][l – 1]) -> 그 외의 경우

num = 5일 때

c[i][j][k][l][m] = c[i – 1][j – 1][k – 1][l - 1][m - 1] + 1 -> x[i] == y[j] == z[k] = w[l] == v[m]일 때

max(c[i - 1][j][k][l][m], c[i][j - 1][k][l][m], c[i][j][k – 1][l][m],

c[i][j][k][l – 1][m], c[i][j][k][l][m – 1]) -> 그 외의 경우

이러한 방식을 사용하여 구성한 최대 길이를 구하는 부분의 코드는 다음과 같다. 차원만 확장한 것이기에 모든 케이스의 경우가 동일하고, 따라서 같은 내용의 중복 설명을 방지하기 위해 여기서는 num = 2의 경우만 가지고 설명을 진행하겠다. num = 2의 경우에서의 최대 길이를 구하는 부분을 코드로 구현하면 다음과 같다.

```c
char* lcs(char** str, int num) {
    int num1, num2, num3, num4, num5;
    int maxLen, stackSize;
    char* LCS;
    char* stack;

    if (num == 2) {
        num1 = strlen(str[0]);
        num2 = strlen(str[1]);

        // memory allocation & initialization
        int** arr = (int**)malloc(sizeof(int*) * (num1 + 1));
        for (int i = 0; i <= num1; i++) {
            arr[i] = (int*)malloc(sizeof(int) * (num2 + 1));
            for (int j = 0; j <= num2; j++) {
                arr[i][j] = 0;
            }
        }

        // finding maximal length of LCS
        for (int i = 1; i <= num1; i++) {
            for (int j = 1; j <= num2; j++) {
                if (str[0][i - 1] == str[1][j - 1]) arr[i][j] = arr[i - 1][j - 1] + 1;
                else arr[i][j] = MAX(arr[i - 1][j], arr[i][j - 1]);
            }
        }
        maxLen = arr[num1][num2];
```

이를 통해 최대 길이를 구한 이후에, 다음 파트인 그 실제 값을 구하는 부분으로 넘어가게 된다. LCS Problem 수업에서 교수님께서 설명하신 stack을 이용하는 backtracking 방식을 사용하여 해당 부분을 구하게 되었다. 앞의 num = 2의 케이스에서의 c[i][j]를 구하는 점화식을 보면, x[i] ==

y[j]일 때 c[i – 1][j – 1] + 1로 넘어가게 되는 구조를 가졌기에, 이를 이용하여 c[i][j] – 1 = c[i – 1][j – 1]이며 x[i] == y[j]일 때 해당 값을 stack에 넣어준 후, 되추적이 끝난 이후에 stack의 모든 내용을 pop하여 역순으로 정렬해준 후 최종 값을 리턴한다.

이러한 알고리즘을 사용하여 되추적 후 실제 값을 구하는 부분은 다음과 같다.

```
// find LCS
LCS = (char*)malloc(sizeof(char) * (maxLen + 1));
stack = (char*)malloc(sizeof(char) * maxLen);
stackSize = 0;

// backtracking
int var1, var2, idx;
var1 = num1;
var2 = num2;
while((var1 != 0) && (var2 != 0)) {
    if ((arr[var1][var2] == (arr[var1 - 1][var2 - 1] + 1)) && (str[0][var1 - 1] == str[1][var2 - 1])) {
        stack[stackSize++] = str[0][var1 - 1];
        var1--;
        var2--;
    }
    else if (MAX(arr[var1 - 1][var2], arr[var1][var2 - 1]) == arr[var1 - 1][var2]) {
        var1--;
    }
    else {
        var2--;
    }
}

// make result
for (int i = 0; i < maxLen; i++) {
    LCS[i] = stack[--stackSize];
}
LCS[maxLen] = '\0';

// memory deallocation
for (int i = 0; i <= num1; i++) {
    free(arr[i]);
}
free(arr);
free(stack);
```

이 LCS 알고리즘의 시간 복잡도를 계산해 보면, 입력 변수의 크기와 개수에 의해 시간 복잡도가 결정된다. 전체 테이블의 크기가 num차원(과제 설명 pdf 기준으로는 k)의 크기이므로 입력 변수 스트링의 개수에 따라 이 값이 비례하게 되고, 따라서 전체 테이블을 1회 순회하며 최대 길이를 구하는 부분의 시간 복잡도는 $O(n^k)$에 속하게 된다. (여기서 n은 각 입력 스트링의 최대 길이이고, k는 입력 스트링의 개수를 의미한다.) 또한 Backtrack하는 과정 또한 전체 테이블을 역순으로 1회 순회하는 방식으로 구해지기 때문에, 동일하게 시간 복잡도는 $O(n^k)$에 속하게 된다. (여기서 n은 각 입력 스트링의 최대 길이이고, k는 입력 스트링의 개수를 의미한다.) 따라서 이 함수의 전체 시간 복잡도는 $O(n^k) + O(n^k) = O(n^k)$에 속하게 된다.