알고리즘 실습(11주차)

201000287 일어일문학과 유다훈

1. 과제 설명 및 해결 방법

1. Dynamic Programming

Dynamic Programming은 문제를 부분적으로 분할하여 각 부분에 대한 답을 구하고, 그것을 이용하여 전체 문제의 답을 구하는 알고리즘 방법이다.

① Matrix-Chain Multiplication

- 어떠한 행렬들을 곱할 때, 행렬의 결합법칙에 의해 행렬을 어떻게 묶어주던지간에 곱하는 각각의 행렬의 가로와 세로의 길이가 같다면 결과값은 동일하다.
- 그러나 결합을 하는데 필요한 연산의 횟수가 달라진다.
- Matrix-Chain Multiplication은 이러한 행렬의 곱에서 최적의 연산을 하는 행렬들의 결합을 찾아내는 알고리즘이다.
- 최소 곱셈횟수 m[I,j]를 찾아내기 위해선 i와 j사이의 임의의 값 k를 이용하여 m[I,k] 와 m[k+1,j]를 더하고, 여기에 곱셈연산횟수 를 더해주는 방식이다.
- 즉, m[I,j]를 구하기 위해선 미리 이전 값들을 구해놓아야 하기 때문에 bottom-up방식으로 해야한다.
- 구현이나 시간은 n^3의 시간이 걸린다.

② Longest Common Subsequence

- 어떠한 문자열 A와 B의 공통된 순서의 연결되지 않는 부분 문자열을 구하는 문제이다.
- 시간복잡도는 Θ(m*n)의 시간이 걸린다.

2. 주요 부분 코드 설명(알고리즘 부분)

1. Matrix-Chain Multiplication

```
void matrix_chain_order(int *p, int n, int maxValue) {
    int ** m;
    int ** s;
    int length = n+1; //길이

    /*

     행렬의 가로세로를 정하는 수는 0~6까지 총 7개이다.
     행렬의 개수는 총 6개이다.
     인덱스를 1에서부터 시작하기 위해 길이를 7로 지정하고 시작한다.

     */

    /* 배열의 초기화 */

    m = (int **)malloc(sizeof(int*) * length);
    s= (int **)malloc(sizeof(int*) * length);
    for (int i = 1; i < length; i++) {
        m[i] = (int *)malloc(sizeof(int *) * length);
    }
    for (int i = 1; i < length; i++) {
        s[i] = (int *)malloc(sizeof(int *) * length);
    }
    for(int i=1; i<length; i++) {
        m[i][i] = 0;
    }

    /* 배열의 초기화 */

    /*Matrix Chain Order*/
    for(int l = 2 ; l <length; l++) { //대각선으로 이동하기 위한 index
        for (int i = 1 ; i< length-l+1; i++) { //대각선에 계산해야할 값들의 개수만큼 진행
            int j = i+l-1; //열 index지정. 대각선으로 진행해야하므로 결과적으로는 대각선 아래방향으로 내려간다.
            m[i][j] = maxValue;
            for(int k=i; k<= j-1; k++) { //어떠한 값 k에 대하여 계산
                 int q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if( m[i][j]>q) { //최소값을 계산하여 삽입.
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    /*Matrix Chain Order*/
```

- 행렬계산의 최소값을 계산해넣을 배열m과 결합순서를 표시할 배열 s를 선언한다.
- 배열 m[i.j]를 구하기 위해선 m[I,k]와 m[k+1,j]가 필요한데, 여기서 k값은 i보다 커야한다. 따라서 i=k가 같은 부분은 필요가 없으므로 0으로 채워넣고 시작한다. 즉 배열m의 초기상태는 전체 배열중 대각선 값은 0으로부터 채워진 상태에서 시작한다.
- 또한 bottom-up방식으로 진행하는 이 방법은 어떠한 값 m[I,j]를 구하기 위해선 바로 직전 열의 값 m[I,k]와 다음 행의 값 m[k+1, j]가 필요하기 때

문에, 이 방법을 만족시키기 위해서 대각선으로 진행해 나가며 값을 구한다.

- m[I,j]의 값과, m[I,k]+m[k+1,j]+곱셈연산횟수를 더한 값을 비교하여, 전자가 더 크다면, 이 알고리즘은 최소의 계산으로 행렬을 합치는 알고리즘이기때문에 값을 바꾸어준다.

2. Longest Common Subsequence

```
void LCS_Length(char * firstSequence, char * secondSequence, int firstM, int secondN, int maxValue) {

    int ** b;
    int ** c;

    b = (int **)malloc(sizeof(int*) * firstM); //첫번째 문자열
    c = (int **)malloc(sizeof(int*) * firstM); //두번째 문자열
    for (int i = 0; i < firstM; i++) {
        b[i] = (int *)malloc(sizeof(int *) * secondN);
        c[i] = (int *)malloc(sizeof(int *) * secondN);
    }

    for(int i=0; i<firstM; i++) {
        for(int j=0; j<secondN; j++) {
            b[i][j] =maxValue; //값을 제대로 검사하기 위해서 모든 값들을 최대값으로 채워넣고 시작.
        }
    }
    /* 배열의 초기화 */
    for(int i = 0; i<firstM; i++) {
        c[i][0] =0;
    }
    for (int j=0; j<secondN; j++) {
        c[0][j] = 0;
    }
    /* 배열의 초기화 */

    /* Longest Common Subsequence */
    for(int i = 1; i<firstM; i++) {
        for (int j = 1; j<secondN ; j++ ) {
            if(firstSequence[i-1] == secondSequence[j-1]) { //첫번째 문자열을 기준으로 두번째 문자열을 비교해나가기 시작
                c[i][j] = c[i-1][j-1] + 1; // 만일 같은 값이라면 대각선 이전 위치의 값에 +1을 더함
                b[i][j] = 1; // 순서를 저장할 배열 b에 1 저장
            } else if (c[i-1][j] >= c[i][j-1]) { //그렇지 않다면, 이전행의 값이 이전열의 값보다 크다면
                c[i][j] = c[i-1][j]; //이전 행의 값을 옮겨 저장
                b[i][j] = 0; //순서를 저장할 배열 b에 0 저장
            } else {
                c[i][j]= c[i][j-1]; //그렇지 않다면, 이전열의 값 저장
                b[i][j] = -1; //순서를 저장할 배열 b에 -1저장
            }
        }
    }
    /* Longest Common Subsequence */
```

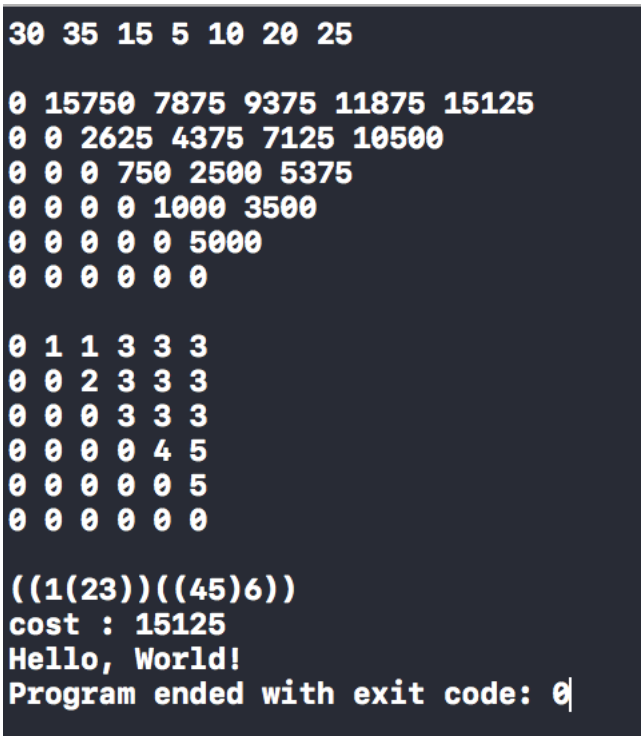
- 문자비교값을 저장할 배열 c와 같은 문자의 순서를 저장할 배열 b를 생성하고 초기화한다.
- 첫번째 문자열을 기준으로 두번째 문자열과 비교하여 같은 부분 문자열의 순서와 어떤 문자인지를 저장해 나간다.

```c
void print_LCS(int ** b, char * firstSequence, int firstM, int secondN) { //부분문자열의 순서를 출력하는 함수
    if ( firstM == 0 || secondN == 0) {
        return;
    }
    if(b[firstM][secondN] == 1) {
        print_LCS(b, firstSequence, firstM-1, secondN-1);
        printf("%c", firstSequence[firstM-1]);
    } else if (b[firstM][secondN] == 0) {
        print_LCS(b, firstSequence, firstM-1, secondN);
    } else {
        print_LCS(b, firstSequence, firstM, secondN-1);
    }
}
```

- 같은 문자값을 출력하는 함수.
- 저장된 값이 1이라면 대각선으로 상승한다. 값이 1일때 두 문자열에서 같은 문자이다.
- 값이 0이라면 한 행 위로 상승한다.
- 값이 -1이라면 한 열 옆으로 이동한다.
- 마지막값까지 재귀적으로 호출하면 값이 1인 값의 문자값을 호출해나가며 다시 복귀한다.

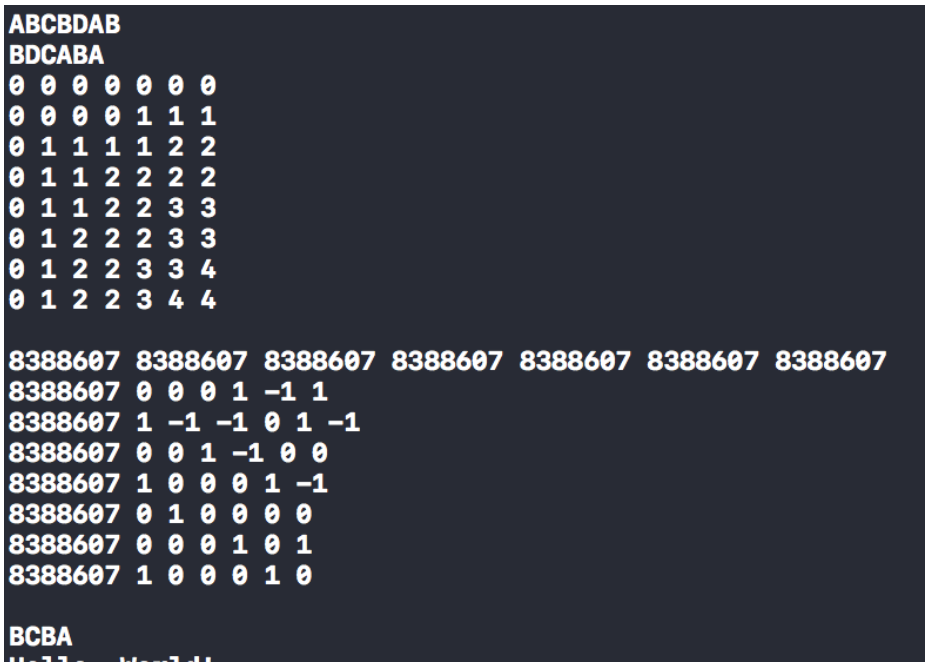
3. 결과

1. Matrix-Chain Multiplication



- Matrix-Chain Multiplication의 최적의 곱셈횟수 값과 각 행렬의 결합순서를 표시한다. 또한
- 본 과제는 수업자료의 input과 output이 동일하다.
- (1*(2*3)) * ((4 * 5)*6) 의 행렬순서대로 곱셈을 한다면 15125번이라는 곱셈횟수가 나온다.

2. Longest Common Subsequence



- 문자열의 순서를 정하고 있는 배열 b의 초기값을 출력해보면, 각 문자열의 알파벳을 나타내는 첫번째 행과 첫번째 열을 제외한 나머지 값에는 값이 제대로 들어가 있다.
- 첫번째 문자열과 두번째 문자열을 비교한 결과 순서대로 중복이 되는 BCBA가 출력되는 것을 확인할 수 있다.