Documentation - LCS

1. LCS_Length
2. LCS_Print

1. LCS_Length

```
static int c[100][100]; // 길이가 저장된 2차원 배열
static char b[100][100]; // Up(위), Cross(대각선), Left(왼쪽) 표시를 위해 나타냄
int i, j;
```

지역변수로 LCS 길이를 저장하는 c 2차원 배열과 LCS가 어디서 왔는지 표시를 하기 위한 b 2차원 배열을 선언해 주었다.

```
void LCS_Length(char* X, char* Y, int m, int n){

    // 0으로 설정된 곳, 빈곳 0으로 설정
    for (i = 1; i <= m; i++) {
        c[i][0] = 0;
    }
    for (j = 0; j <= n; j++) {
        c[0][j] = 0;
    }

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (X[i-1] == Y[j-1]) { // 문자열이 같을때
                b[i][j] = 'C';
                c[i][j] = c[i - 1][j - 1] + 1; //  Cross에서 온 값에 문자열이 같으므로 1증가
            }
            else { // 문자열이 다를때
                if (c[i - 1][j] >= c[i][j - 1]) {
                    b[i][j] = 'U'; // i-1이 더 클때(전의 행에서 온 값이 더 클때) Up에서 온 값 저장
                    c[i][j] = c[i - 1][j];
                }
                else {
                    b[i][j] = 'L'; // j-1이 더 클때(전의 열에서 온 값이 더 클때) Left에서 온 값 저장
                    c[i][j] = c[i][j - 1];
                }
            }
        }
    }
}
```

```
char dna1[100]; // 첫번째 DNA
char dna2[100]; // 두번째 DNA

cout << "S1: ";
cin >> dna1;
cout << "S2: ";
cin >> dna2;

int dna1_len = strlen(dna1);
int dna2_len = strlen(dna2);

LCS_Length(dna1, dna2, dna1_len, dna2_len);
```

(in main)

main에서 입력 받은 dna1, dna2와 각각의 길이를 LCS_Length함수로 전달해준다.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } x_i \neq y_j \end{cases}$$

위의 식이 LCS의 Length를 computing 해주는 기본적인 알고리즘이다. 이 식에 맞춰서 문자열이 같을 때는 왼쪽 대각선의 값에 1을 더해주고, 다르면 각각 조건에 맞게 왼쪽과 위에서 온 값 중 큰 값을 c배열에 저장해준다. 그리고 LCS를 출력하기 위해 어디서 왔는지를 표시하기 위한 b배열에 하나의 문자를 저장해준다.

Length 배열:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
0	0	0	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3
0	1	1	1	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	4
0	1	1	1	2	2	2	2	3	4	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5
0	1	2	2	2	3	3	3	4	4	4	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
0	1	2	3	3	3	3	4	4	4	4	4	5	5	6	6	6	6	6	6	6	6	7	7	7	7	7	7
0	1	2	3	4	4	4	4	5	5	5	5	5	6	6	6	7	7	7	7	7	7	7	7	8	8	8	8
0	1	2	3	4	4	4	4	5	5	6	6	6	6	6	6	7	7	7	7	7	7	7	7	8	8	9	9
0	1	2	3	4	4	4	4	5	6	6	6	6	7	7	7	7	7	7	8	8	8	8	8	8	8	9	9
0	1	2	3	4	5	5	5	5	6	6	6	7	7	7	7	7	8	8	8	8	9	9	9	9	9	9	9
0	1	2	3	4	5	5	5	6	6	6	6	7	8	8	8	8	8	8	9	9	9	9	9	10	10	10	10
0	1	2	3	4	5	5	6	6	6	6	6	7	8	9	9	9	9	9	9	10	10	10	10	10	10	10	10
0	1	2	3	4	5	5	6	7	7	7	7	7	8	9	9	10	10	10	10	10	10	10	10	11	11	11	11
0	1	2	3	4	5	5	6	7	7	7	7	7	8	9	10	10	10	10	10	11	11	11	11	11	11	11	11
0	1	2	3	4	5	5	6	7	8	8	8	8	8	9	10	11	11	11	11	11	11	11	11	12	12	12	12
0	1	2	3	4	5	5	6	7	8	8	8	8	9	9	10	11	11	11	12	12	12	12	12	12	12	12	12
0	1	2	3	4	5	5	6	7	8	9	9	9	9	9	10	11	11	11	12	12	12	12	12	12	12	13	13
0	1	2	3	4	5	5	6	7	8	9	10	10	10	10	10	11	11	11	12	12	12	12	12	12	12	13	14
0	1	2	3	4	5	5	6	7	8	9	10	10	11	11	11	11	11	11	12	12	12	12	12	13	13	13	14
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	12	12	12	12	12	13	13	13	13	13	13	13	14
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	13	13	13	13	13	14	14	14	14	14	14	14
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	14	14	14	14	14	15	15	15	15
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	15	15	15	15	15	15	15	15
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	16	16	16	16	16	16	16	16
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	16	16	17	17	17	17	17	17
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	16	16	17	17	18	18	18	18
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	16	16	17	17	18	18	19	19
0	1	2	3	4	5	5	6	7	8	9	10	10	11	12	13	14	14	14	15	16	16	17	17	18	18	19	20

그러면 Length 배열(c배열)은 이렇게 구현이 되고, 마지막에 있는 값 c[dna1_len][dna2_len]이 LCS의 길이가 된다.

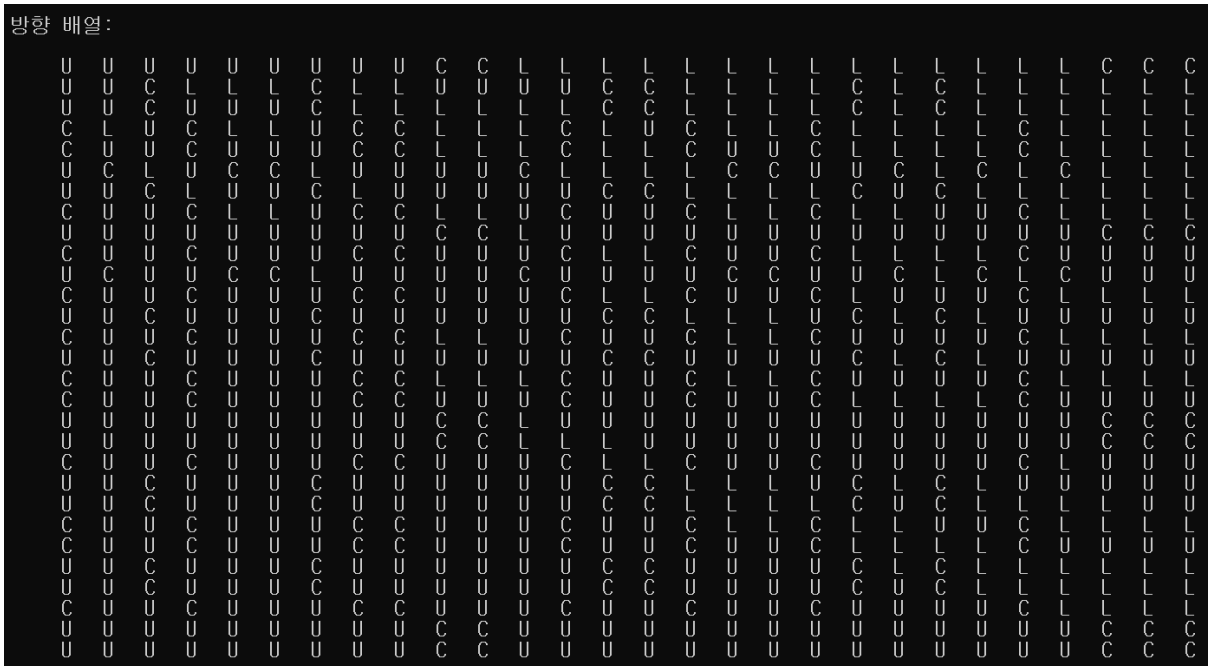
2. LCS_Print

```
cout << "LCS : ";
LCS_Print(dna1,dna1_len,dna2_len);
```

(in main)

```
void LCS_Print(char* X,int i, int j) { // LCS 프린트하는 과정

    if (i == 0 || j == 0) {
        return;
    }

    if (b[i][j] == 'C') { // Cross 일때 출력
        LCS_Print(X, i - 1, j - 1); // 재귀함수로 구현
        cout << X[i - 1];
    }
    else {
        if (b[i][j] == 'U') {
            LCS_Print(X, i - 1, j); // 행 줄여가면서 재귀함수
        }
        else {
            LCS_Print(X, i, j - 1); // 열 줄여가면서 재귀함수
        }
    }
}
```

main 함수에서 dna1과 dna1, dna2 두개의 길이를 넘겨주어서 LCS_Print함수를 실행한다. 이때는 재귀함수로 구현되어 있는데, C(cross)일 때는 왼쪽 대각선의 값을 넣은 것으로 LCS_Print가 실행되고 나머지 U, L 문자일때는 행과 열을 하나씩 줄여간 값을 LCS_Print에 넣어 실행시킨다. 또한 C(Cross)일 때는 두개의 문자가 같은 것이므로 이때만 dna값이 출력되게 하였다. 그리고 재귀함수 탈출조건인 i==0 또는 j==0을 만들어서 탈출이 되게 하였다.



방향을 나타내는 배열(b배열)은 이렇게 구현이 되어있고, 마지막에 있는 값b[dna1_len][dna2_len] 부터 출발하여 각각 조건을 실행한다.

Documentation – SSSP

1. main
2. check
3. initialization
4. dijkstra
5. bellmanford
6. tracking_dist
7. print_SSSP

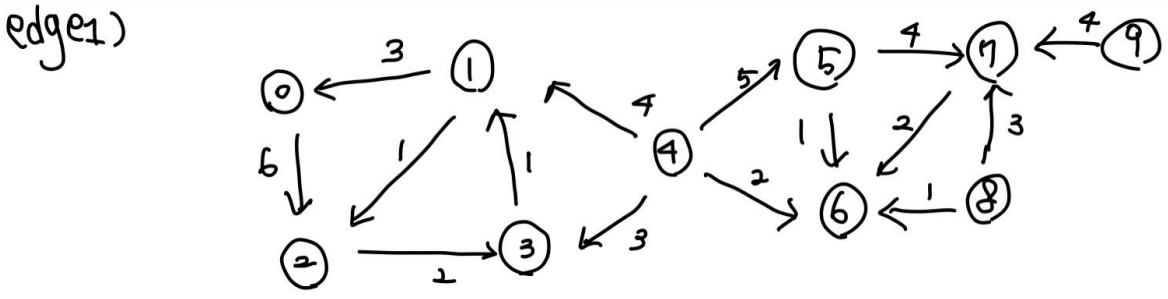
1. main

```
#include <iostream>
#include <queue>
#include <vector>
#pragma warning (disable : 4996)

using namespace std;

#define INF 1000000000
#define SIZE 10

vector <pair<int, int>> graph[SIZE]; // graph[n]번째 노드가 <n번째 노드와연결,cost는 n>
int d[SIZE]; // vertex의 최소비용
int pre[SIZE]; // 어디서왔는지 node저장
int i, j;
int src = 0; // 어디서부터출발할것인지 (전부0번째 node로 설정)
```

INF를 1억으로 만들어주고, SIZE는 node의 개수로 10개가 있는 그래프들을 만들었다. 또한 d 배열은 그 노드의 최소비용을 저장해주고, pre배열은 어디서 왔는지에 대한 노드들을 저장해준다. 또한 src는 어디서부터 출발할 것인지에 대한 값이다. Graph는 vector로 구성되었고 graph[1]<3,2> 이라면 1번째 노드가 3번째 노드와 cost가 2인것으로 연결되어 있다는 뜻이다.

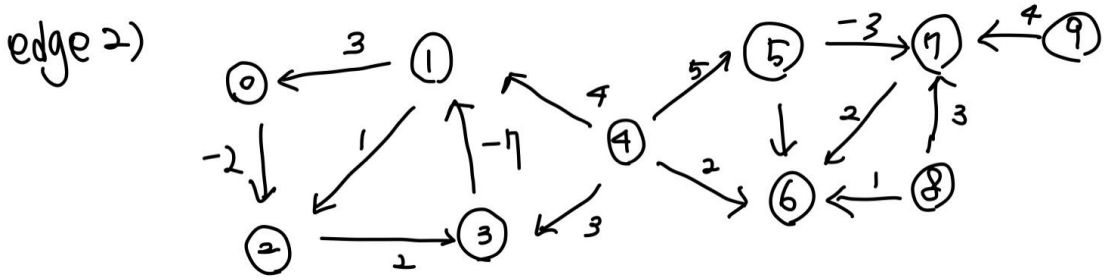
총 5개의 그래프를 검증해야 하고, 5개의 input은 아래와 같다.

```
int edge1[SIZE][SIZE] = {
    {0,INF,6,INF,INF,INF,INF,INF,INF,INF},
    {3,0,1,INF,INF,INF,INF,INF,INF,INF},
    {INF,INF,0,2,INF,INF,INF,INF,INF,INF},
    {INF,1,INF,0,INF,INF,INF,INF,INF,INF},
    {INF,4,INF,3,0,5,2,INF,INF,INF},
    {INF,INF,INF,INF,INF,0,1,4,INF,INF},
    {INF,INF,INF,INF,INF,INF,0,INF,INF,INF},
    {INF,INF,INF,INF,INF,INF,2,0,INF,INF},
    {INF,INF,INF,INF,INF,INF,1,3,0,INF},
    {INF,INF,INF,INF,INF,INF,INF,4,INF,0},
}; // 방향 , NEGATIVE x
```



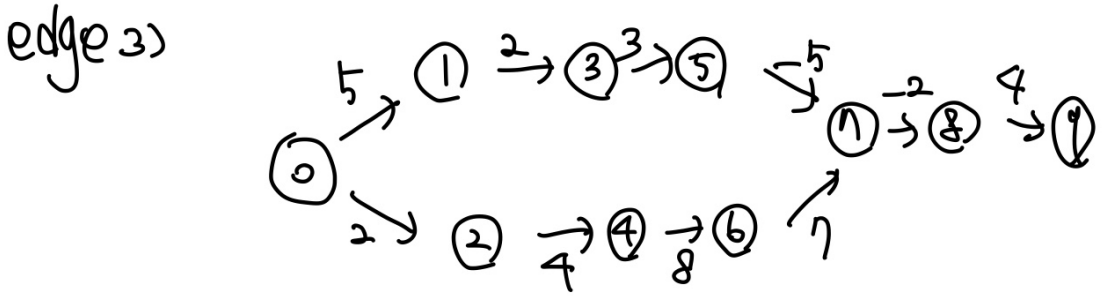
첫번째 input은 그림과 같은 그래프로 adjacency matrix로 구현하였다. 방향이 있고 negative weight edge가 없는 그래프이다. 그래서 negative cycle도 없다. 이 그래프는 다익스트라 알고리즘을 이용하여 해결될 것으로 기대된다.

```
int edge2[SIZE][SIZE] = {
    {0,INF,-2,INF,INF,INF,INF,INF,INF,INF},
    {3,0,1,INF,INF,INF,INF,INF,INF,INF},
    {INF,INF,0,2,INF,INF,INF,INF,INF,INF},
    {INF,-7,INF,0,INF,INF,INF,INF,INF,INF},
    {INF,4,INF,3,0,5,2,INF,INF,INF},
    {INF,INF,INF,INF,INF,0,1,-3,INF,INF},
    {INF,INF,INF,INF,INF,INF,0,INF,INF,INF},
    {INF,INF,INF,INF,INF,INF,2,0,INF,INF},
    {INF,INF,INF,INF,INF,INF,1,3,0,INF},
    {INF,INF,INF,INF,INF,INF,INF,4,INF,0},
}; //방향, NEGATIVE O CYCLE O
```



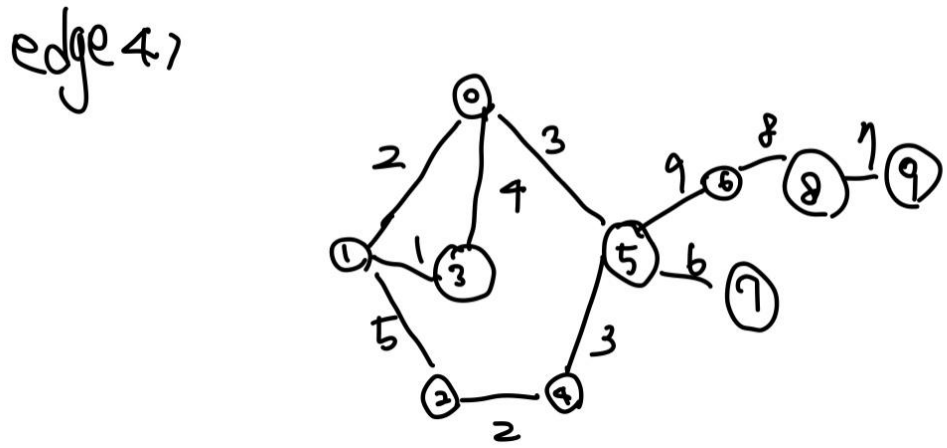
두번째 input은 그림과 같은 그래프로 adjacency matrix로 구현하였다. 방향이 있고 negative weight edge가 있는 그래프이다. 그리고 negative cycle도 있다. 이 그래프는 에러메세지가 뜨고 종료될 것으로 기대된다.

```
int edge3[SIZE][SIZE] = {
    {0,5,2,INF,INF,INF,INF,INF,INF,INF},
    {INF,0,INF,2,INF,INF,INF,INF,INF,INF},
    {INF,INF,0,INF,4,INF,INF,INF,INF,INF},
    {INF,INF,INF,0,INF,3,INF,INF,INF,INF},
    {INF,INF,INF,INF,0,INF,8,INF,INF,INF},
    {INF,INF,INF,INF,INF,0,INF,-5,INF,INF},
    {INF,INF,INF,INF,INF,INF,0,7,INF,INF},
    {INF,INF,INF,INF,INF,INF,INF,0,-2,INF},
    {INF,INF,INF,INF,INF,INF,INF,INF,0,4},
    {INF,INF,INF,INF,INF,INF,INF,INF,INF,0},
}; //방향, NEGATIVE O CYCLE X
```



세번째 input은 그림과 같은 그래프로 adjacency matrix로 구현하였다. 방향이 있고 negative weight edge가 있는 그래프이다. 그리고 negative cycle은 없다. 이 그래프는 벨만포드 알고리즘을 이용하여 해결될 것으로 기대된다.

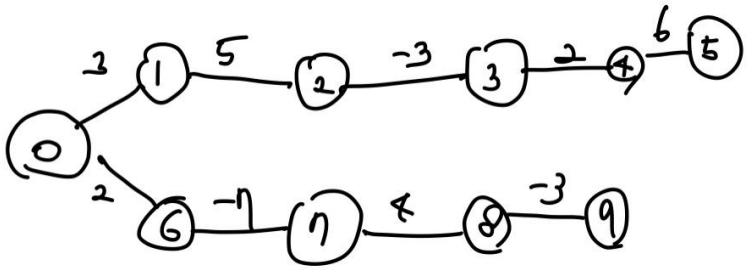
```
int edge4[SIZE][SIZE] = {
    {0,2,INF,4,INF,3,INF,INF,INF,INF},
    {2,0,5,1,INF,INF,INF,INF,INF,INF},
    {INF,5,0,INF,2,INF,INF,INF,INF,INF},
    {4,1,INF,0,INF,INF,INF,INF,INF,INF},
    {INF,INF,2,INF,0,3,INF,INF,INF,INF},
    {3,INF,INF,INF,3,0,9,6,INF,INF},
    {INF,INF,INF,INF,INF,9,0,INF,8,INF},
    {INF,INF,INF,INF,INF,6,INF,0,INF,INF},
    {INF,INF,INF,INF,INF,INF,8,INF,0,7},
    {INF,INF,INF,INF,INF,INF,INF,INF,7,0},
}; //무방향, NEGATIVE X
```



네번째 input은 그림과 같은 그래프로 adjacency matrix로 구현하였다. 방향이 없고 negative weight edge가 없는 그래프이다. 그래서 negative cycle도 없다. 이 그래프는 다익스트라 알고리즘을 이용하여 해결될 것으로 기대된다.

```
int edge5[SIZE][SIZE] = {
    {0,3,INF,INF,INF,INF,2,INF,INF,INF},
    {3,0,5,INF,INF,INF,INF,INF,INF,INF},
    {INF,5,0,-3,INF,INF,INF,INF,INF,INF},
    {INF,INF,-3,0,2,INF,INF,INF,INF,INF},
    {INF,INF,INF,2,0,6,INF,INF,INF,INF},
    {INF,INF,INF,INF,6,0,INF,INF,INF,INF},
    {2,INF,INF,INF,INF,INF,0,-7,INF,INF},
    {INF,INF,INF,INF,INF,INF,-7,0,4,INF},
    {INF,INF,INF,INF,INF,INF,INF,4,0,-3},
    {INF,INF,INF,INF,INF,INF,INF,INF,-3,0},
}; //무방향, NEGATIVE O, CYCLE O
```

edge5)



다섯 번째 input은 그림과 같은 그래프로 adjacency matrix로 구현하였다. 방향이 없고 negative weight edge가 있는 그래프이다. 그리고 negative cycle이 존재한다. 모든 노드가 하나로 연결되어 일자 형태를 갖고 있기 때문이다. 이 그래프는 에러메세지가 뜨고 종료될 것으로 기대된다.

```
int main() {

    int edge1[SIZE][SIZE] = {
        {0,INF,6,INF,INF,INF,INF,INF,INF,INF},
        {3,0,1,INF,INF,INF,INF,INF,INF,INF},
        {INF,INF,0,2,INF,INF,INF,INF,INF,INF},
        {INF,1,INF,0,INF,INF,INF,INF,INF,INF},
        {INF,4,INF,3,0,5,2,INF,INF,INF},
        {INF,INF,INF,INF,INF,0,1,4,INF,INF},
        {INF,INF,INF,INF,INF,INF,0,INF,INF,INF},
        {INF,INF,INF,INF,INF,INF,2,0,INF,INF},
        {INF,INF,INF,INF,INF,INF,1,3,0,INF},
        {INF,INF,INF,INF,INF,INF,INF,4,INF,0},
    }; // 방향 , NEGATIVE x

    cout << "------------case1--------------\n";
    if (check(edge1)) {
        for (i = 0; i < SIZE; i++) {
            for (j = 0; j < SIZE; j++) {
                if (edge1[i][j] != INF && edge1[i][j] != 0) graph[i].push_back(make_pair(j, edge1[i][j]));
            }
        }
        dijkstra(src);  printSSSP();
    }
    else {
        if (bellmanford(edge1)) printSSSP();
    }
```

그래서 main함수를 살펴보면, adjacency matrix로 구현한 그래프를 edge의 가중치들을 설정하여 그래프를 만들었다. 그리고 if문안의 check함수를 통해 negative edge가 있는지 확인을 하고 없다면 check는 1, 있으면 check는 0이다. Negative edge가 없다는 것은 다익스트라 알고리즘을 이용할 것 기대되고, negative edge가 있으면 벨만포드 알고리즘을 이용할 것으로 기대된다. 그리고 또한 다익스트라 알고리즘을 이용할 때 min priority queue를 이용해야 해서 기존 adjacency matrix의 값들을 vector에 넣어주는 것을 수행해야 한다.

2. check

```
bool check(int edge[][SIZE]) {

    bool checkpoint = 1; // negative edge있는지 확인

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            if (edge[i][j] < 0) {
                checkpoint = 0; // 있으면 checkpoint 0으로 설정
            }
        }
    }
    return checkpoint;
}
```

check함수는 2차원배열들인 edge를 받아 하나라도 음수가 있다면 0으로 설정하여 return해준다.

3. initialization

```
void initialization() {

    for (i = 0; i < sizeof(d) / sizeof(int); i++) {
        d[i] = INF;
        pre[i] = NULL;
    }
    d[src] = 0; // initialiazation 과정, d[]는 모두 무한대로, pre는 모두 null값으로 설정
}
```

벨만포드, 다익스트라 모두 initialization해주는 과정이 포함된다. 그래서 모든 노드의 값들을 INF로 설정하고, pre배열은 전에서 온 것들이 없으므로 NULL로 설정해준다. 또한 어디서부터 시작할 것인지를 나타내는 src의 node값은 0으로 설정해준다.

4. dijkstra

```cpp
void dijkstra(int start) {

    initialization();
    d[SIZE] = 0;
    pre[SIZE] = 0;

    // priority queue로 구성
    priority_queue <pair<int, int>> pq;
    pq.push(make_pair(start, 0));

    while (!pq.empty()) {

        // extract - min(Q)
        int current = pq.top().first; // 현재 node
        int dist = -pq.top().second; // 현재 node의 비용
        pq.pop();

        if (d[current] < dist) continue; // 최단거리가 아닌 경우 스킵

        for (i = 0; i < graph[current].size(); i++) {

            // 선택된 노드의 인접 노드에 대해 relax
            int next = graph[current][i].first; // next node
            int nextdist = dist + graph[current][i].second; // next node의 비용

            // 기존의 최소 비용보다 더 적으면 교체
            if (nextdist < d[next]) {
                d[next] = nextdist;
                pq.push(make_pair(next, -nextdist)); //min priority queue 구성 위해 비용을 -로저장, 꺼낼때 -붙여서 꺼냄
                pre[next] = current; //pre에 어디서왔는지 저장
            }
        }
    }
    cout << "dijkstra로 구현함\n\n";
}
```

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = V
4  while Q ≠ ∅ do
5  |    u = EXTRACT-MIN(Q)
6  |    S = S ∪ {u}
7  |    for each v ∈ Adj[u] do
8  |    |    RELAX(u, v, w)
9  |    end
10 end
```

위의 수도코드를 참고하여 구현하였다. start 로 어디서부터 시작할 것인지 노드를 함수의 매개변수로 받고, initialization을 실행한다. 그리고 d, pre 배열을 0으로 초기화 시켜주고, priority queue로 pq를 구성한 후에 start의 node정보를 저장한다. 그리고 pq의 queue가 빌 때까지 extract-min(q) 과정과 현재 pq에서 꺼낸 node와 인접한 node에 대해 relax과정을 진행한다. 그리고 최소비용으로 바꾸는 과정에서 pre배열에 어디서 온 node인지 저장해준다.

5. bellmanford

```
bool bellmanford(int edge[][SIZE]) {

    initialization();
    d[SIZE] = 0;
    pre[SIZE] = 0;

    // relax과정
    for (int count = 0; count < SIZE - 1; count++) {
        for (i = 0; i < sizeof(d) / sizeof(int); i++) {
            for (j = 0; j < sizeof(d) / sizeof(int); j++) {
                if (edge[i][j] != INF && d[j] > d[i] + edge[i][j]) { // 더 작은 비용으로 교체
                    d[j] = d[i] + edge[i][j];
                    pre[j] = i;
                }
            }
        }
    }

    // negative cycle 찾는 과정
    for (i = 0; i < sizeof(d) / sizeof(int); i++) {
        for (j = 0; j < sizeof(d) / sizeof(int); j++) {
            if (d[j] > d[i] + edge[i][j]) {
                cout << "negative cycle이 발견되어서 종료합니다.\n\n";
                return false;
            }
        }
    }

    cout << "bellmanford로 구현함\n\n";
    return true;
}
```

```
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |V| − 1 do
3      for each edge (u, v) ∈ E do
4          RELAX(u, v, w)
5      end
6  end
7  for each edge (u, v) ∈ E do
8      if d[v] > d[u] + w(u, v) then
9          return FALSE // G has a negative-wt cycle
10     
11 end
12 return TRUE // G has no neg-wt cycle reachable frm s
```

위의 수도코드를 참고하여 구현하였다. edge를 매개변수로 받고, initialization을 실행한다. 그리고 d, pre 배열을 0으로 초기화 시켜주고, 노드의 개수-1만큼 모든 edge에 대해 더 작은 비용으로 교체하는 relax를 시켜준다. 최소비용으로 바꾸는 과정에서 pre배열에 어디서 온 node인지 저장해준다. 그 다음 한 번 더 relax를 시켜 만약 relax한 값이 더 작다면 그것은 negative cycle이 있는 것으로 판별을 해준다.

6. tracking_dist

```
void tracking_dist(int k) {

    // 각 node에 대해서 어디서 온 최소비용인지 재귀함수로 구현
    if (pre[k] != src) {
        tracking_dist(pre[k]);
        cout << pre[k] << "->";
    }
    else {
        cout <<  src << "->";
    }
    return;
}
```

k 로 어떤 node에 대한 path를 찾을 것인지 입력 받고, node가 src가 될 때까지 tracking_dist 재귀함수로 구현하고 출력하는 과정을 구현하였다.

7. printSSSP

```
void printSSSP() {// print 함수

    for (i = 1; i < SIZE; i++) {
        if (d[i] != INF) {
            cout << src << "부터 " << i << "까지의 최단 경로 : ";
            tracking_dist(i);

            cout << i << " cost : " << d[i] << "\n\n\n";
        }
    }
}
```

d[i]가 INF가 아니면, 즉 방문해서 최소비용으로 맞춰진 노드라면 경로를 tracking_dist 함수를 이용해서 print되도록 하였고 그리고 그 최소비용을 print 하게 구현하였다.

Lessons

코드를 실행시켜 나온 output file을 보면, 우선 LCS는 가장 마지막에 있는c[dna1_len][dna2_len]이 20이 되어 LCS 길이가 20이 되는 것을 확인할 수 있고, 또한 b[dna1_len][dna2_len]에서 C의 값을 따라가서 LCS문자가 출력되는 것을 확인해볼 수 있다. LCS 알고리즘을 brute-force로 문제를 풀 때는, X, Y 두개의 문자열이 있을 때 모든 X의 subsequence에 대해 Y의 subsequence를 체크해야 하고 그때의 시간복잡도는 Θ(n*2^m)으로 나타낼 수 있다. m과 n은 입력 문자열의 길이이고 X의 subsequences는 2^m개이고 그 subsequences를 각각 check하는 시간은 Θ(n)이 소요된다. 이때는 중복되는 서브문제들이 나올 수도 있고 이를 모두 재귀적으로 계산하는 것은 비효율적이다. 따라서 각문자열의 마지막 문자의 값의 동일여부를 이용해 LCS값을 2차원 배열에 저장하는 Dynamic Programming을 이용하여 알고리즘을 구성하였고 이때의 시간복잡도는 Θ(m*n), 즉 Θ(n^2)으로 나타낼 수 있다. 또한 이 2차원배열로 문자열의 길이를 알 수 있을 뿐만 아니라 실질적으로 문자열이 어디서 왔는지 추적할 수 있게 해준다. 효율성을 중시하는 알고리즘에서, Brute-force로 문제를 구현했을 때 보다 중복되는 계산을 피하기 때문에 시간이 훨씬 적게 걸리는 것을 확인해 볼 수 있다. 이를 통해 알고리즘은 효율성을 가장 중시해야 해야 한다는 것을 깨닫고, DP의 중요성과 memoization을 어떻게 해야 하는지 알게 되었다.

SSSP는 벨만포드, 다익스트라 두개로 구현할 수 있는데 negative cycle이 존재하는 경우는 계산이 불가능하고, negative edge가 존재하는 경우는 벨만포드로, 아닌 경우는 벨만포드, 다익스트라 두가지 다 계산이 가능하다는 것을 확인해 볼 수 있다. Input의 그래프들을 모두 실행시켜본 결과 기대했던 결과와 일치하는 것을 확인하였다. 벨만포드의 시간복잡도는 initialization하는 시간 Θ(V)와 relax하는 시간 O(VE), negative cycle을 검사하는 시간 O(E)가 합쳐져서 O(V+VE+E), 즉 O(V*E)=O(V^2)이다. 하지만 다익스트라의 경우는 다르다. min-priority queue를 이용하여 가장 최단 거리를 가진 node를 찾는 시간은 logV이다. 따라서 min-priority queue를 사용하지 않는 다익스트라 알고리즘은 똑같이 O(V^2)이지만, min-priority queue를 사용하는 다익스트라 알고리즘은 initialization하는 시간 Θ(V)와 priority queue에서 min node의 정보를 찾고 삭제하는 시간 O(VlogV), queue를 사용해서 relax하는 시간 O(ElogV)가 합쳐져서 O(VlogV+ElogV)=O(ElogV)가 된다. 이를 통해 하나의 데이터가 어떤 방식으로 저장되어 있는지에 따라 알고리즘의 효율성이 달라지는 것을 알게 되었다. 그래서 각 데이터 특성에 맞게 queue, stack, heap등의 자료구조를 사용해야 하고, 최단거리를 찾는 알고리즘은 항상 최소인 거리를 반환해야 하므로 min heap구조를 이용하는 것이 적절하다는 것을 과제를 통해 깨닫게 되었다.