데이터구조설계
Project #3

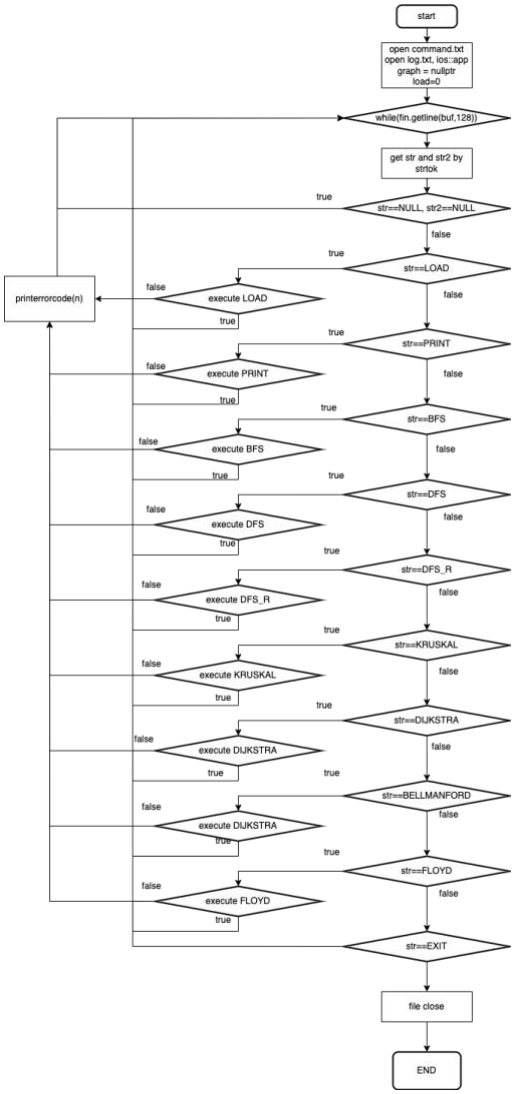
학부: 컴퓨터정보공학부
담당교수: 이기훈 교수님
학번: 2019202021
성명: 정 성 엽
제출일: 2022.12.15

1. Introduction

이번 프로젝트에서는 Matrix 또는 List 형식으로 주어진 Graph를 통해 Graph 출력, Graph 탐색과 그래프 최단 경로 탐색 알고리즘을 이용하여 그래프의 vertex와 weight 또는 합계 또는 이동 경로를 출력하도록 하였다. BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMANFORD, FLOYD 알고리즘을 이용하여 탐색을 진행하였습니다. 탐색에 사용되는 그래프는 graph_L.txt 또는 graph_M.txt에 데이터를 넣어 구성하였고, vertex 값이 없거나 초과되거나 또는 음수 사이클이 발생한 경우를 고려하여 에러 메세지를 출력한다.

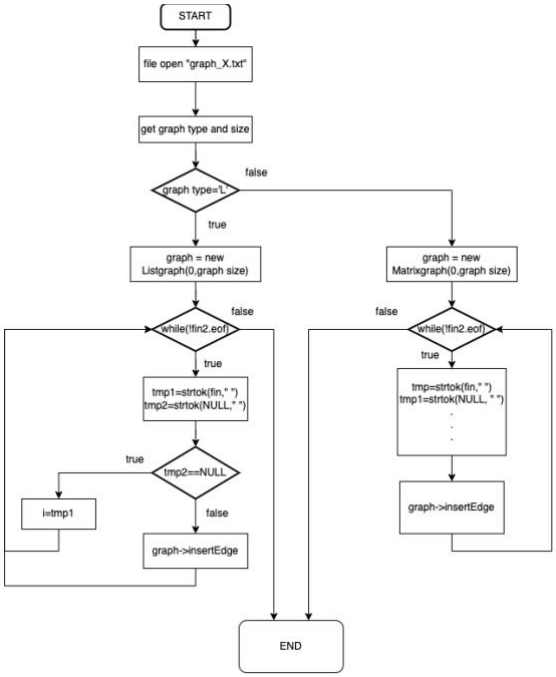
2. Flowchart

1) Manager.cpp function Run



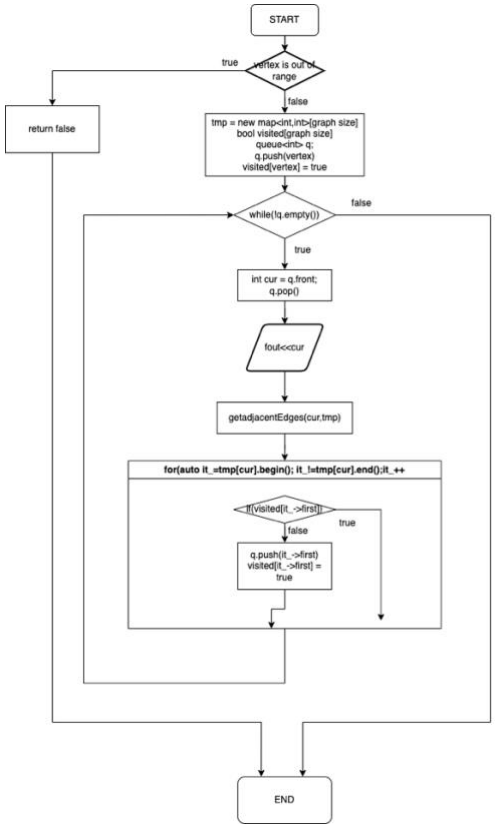
manager.cpp의 Run 함수의 flowchart로 사용자가 command.txt에 무엇을 line 별로 입력했는지에 따라 다른 함수를 실행한다. 함수는 LOAD, PRINT, BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMANFORD, FLOYD가 있고 해당 함수 에러 발생 시 에러 코드를 출력하도록 한다.

2) Manager.cpp function LOAD



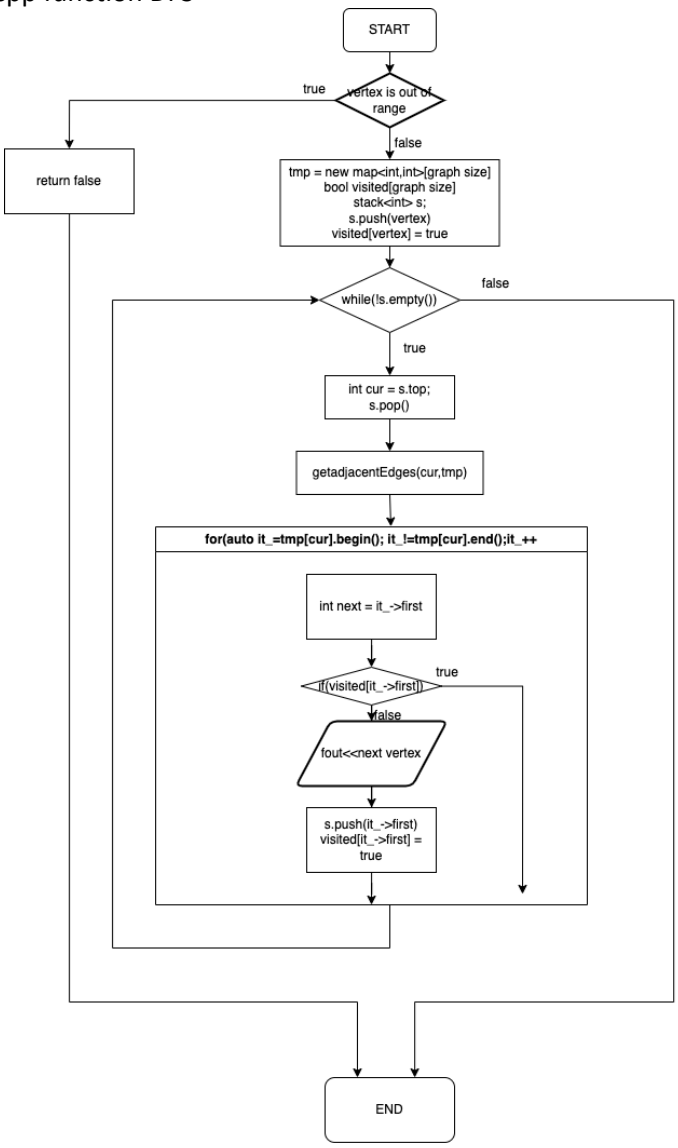
어떤 그래프 파일이 들어오던 맨 처음 줄에는 그래프 타입, 다음 줄에는 그래프 사이즈가 존재하므로 먼저 읽어서 가져온다. 처음 읽은 그래프 타입 별로 프로그램을 진행하며 L인 경우 각 줄에 1개 또는 2개의 숫자를 읽어와서 InsertEgde 함수를 실행한다. M인 경우 그래프 사이즈 만큼 반복하여 각 라인의 수를 모두 가져오고 각 insertEdge 함수를 실행한다.

3) GraphMethod.cpp function BFS



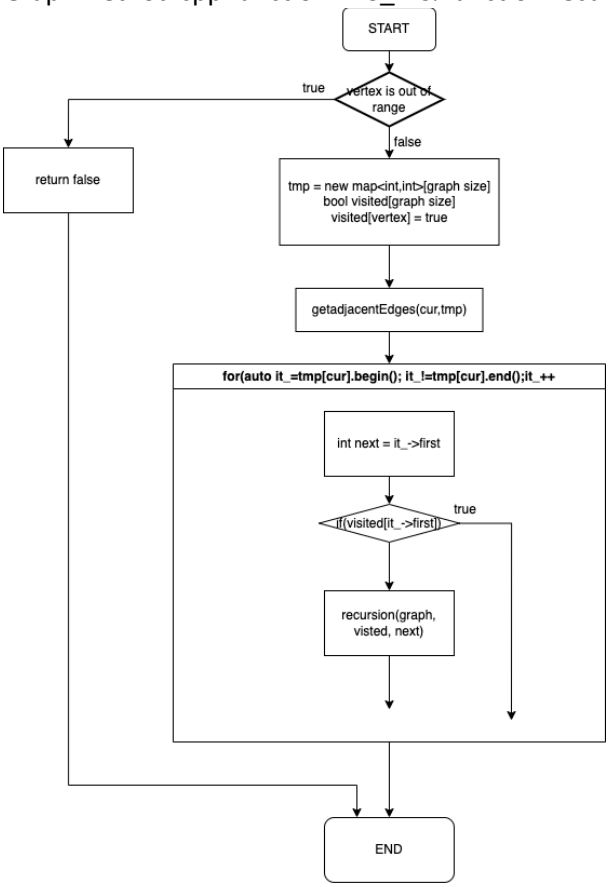
BFS는 queue에 처음 방문할 vertex를 넣고 FIFO 규칙에 따라 해당 숫자의 인접한 vertex를 queue에 넣고 뺀다. queue에서 pop된 vertex 숫자가 이미 방문한 vertex인 경우 다음 for문으로 넘어간다. 중요한 것은 while문 이후 for문 사이에 fout으로 출력하는 점이다.

4) GraphMethod.cpp function DFS



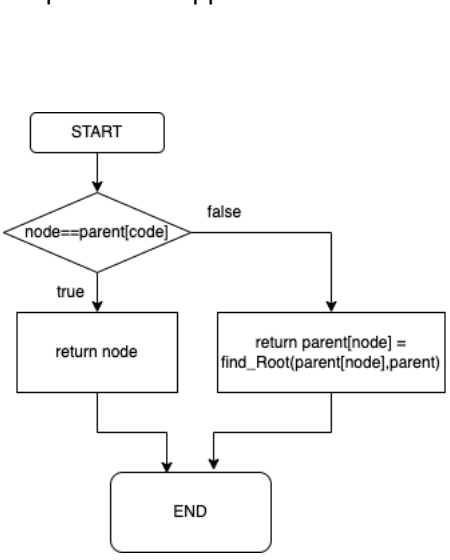
DFS 또한 BFS와 크게 다르지 않으나 사용하는 STL이 queue가 아닌 stack을 사용해서 지나갈 vertex를 LIFO 규칙에 따라 넣고 뺀다. stack에서 pop된 숫자가 이미 방문한 vertex인 경우 다음 for문으로 넘어가고 또 BFS와 다른 점은 for문 안에서 출력이 진행된다.

5) GraphMethod.cpp function DFS_R & function recursion

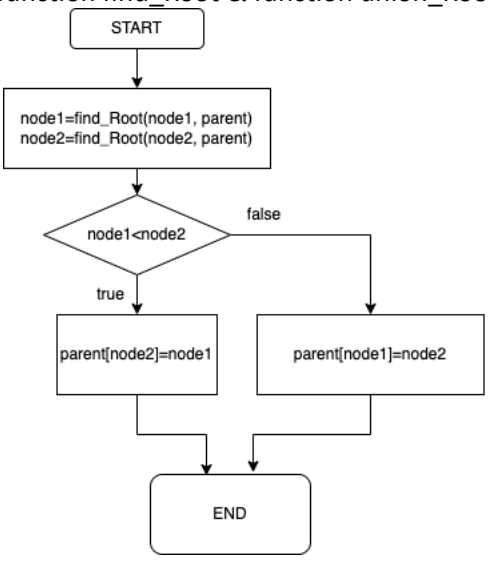


DFS_R은 DFS와 다르게 stack을 사용해서 출력한다. 함수는 2개를 만들어 사용했으나 Flowchart가 1개인 이유는 안에서 돌아가는 동작은 같으나 fout << "=========="와 같은 동작은 DFS_R에서 진행되고 나머지 출력은 recursion 함수에서 진행되도록 만들었기 때문이다.

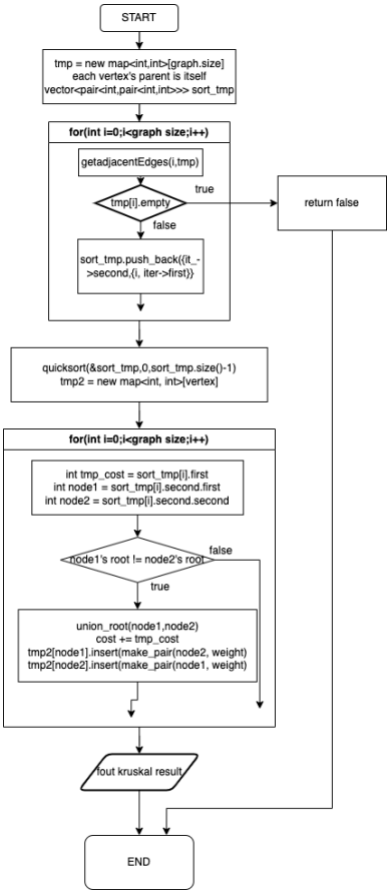
6) GraphMethod.cpp function Kruskal & function find_Root & function union_Root



<function find_Root>



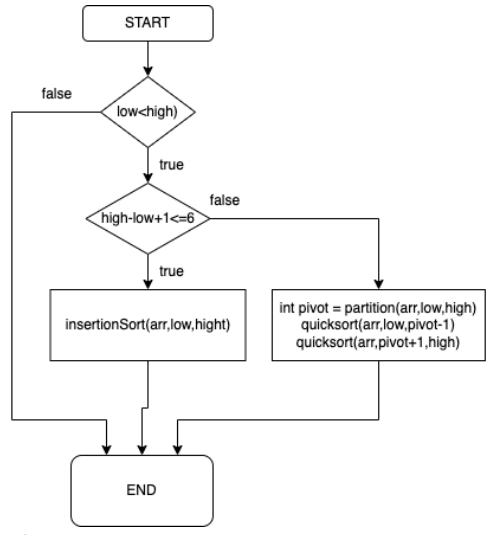
<function union_Root>



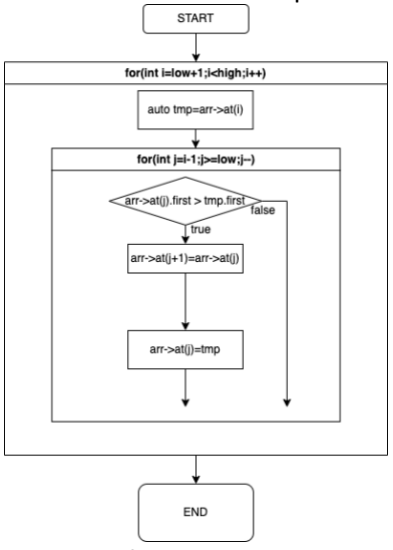
<function Kruskal>

Kruskal은 MST를 찾는 함수로 모든 vertex에서 인접한 vertex와 그에 대한 가중치를 가져와서 sort_tmp라는 vector에 입력한다. 후에 quicksort 또는 insertionsort를 진행하여 오름차순으로 정렬 후에 MST를 제작한다. MST는 find_Root를 통해 root가 같을 경우 cycle이므로 생성되지 않도록 하였고 다른 경우 union_Root를 통해 연결시키고 tmp2에 연결된 edges를 저장하여 후에 출력할 때 사용하였다.

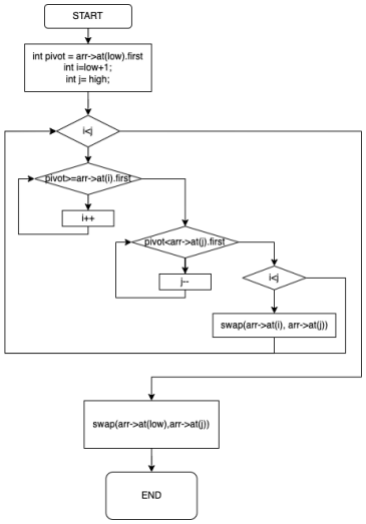
7) GraphMethod.cpp function quicksort & function insertsionSort & function partition



<function quicksort>



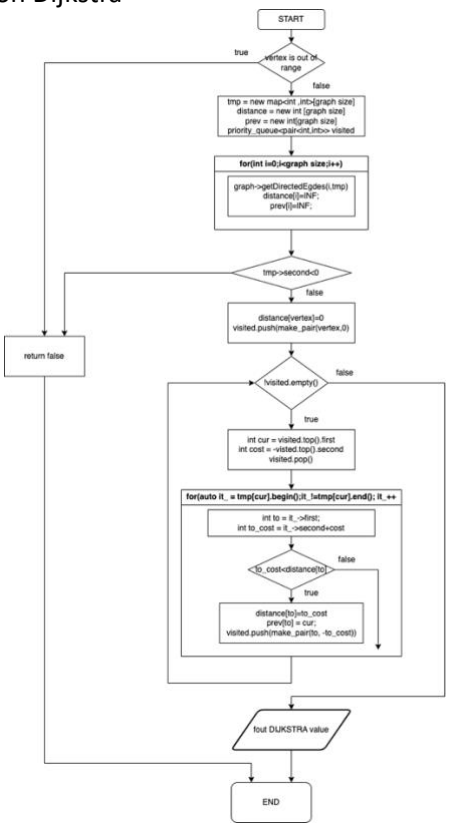
<function insertionSort>



<function Partition>

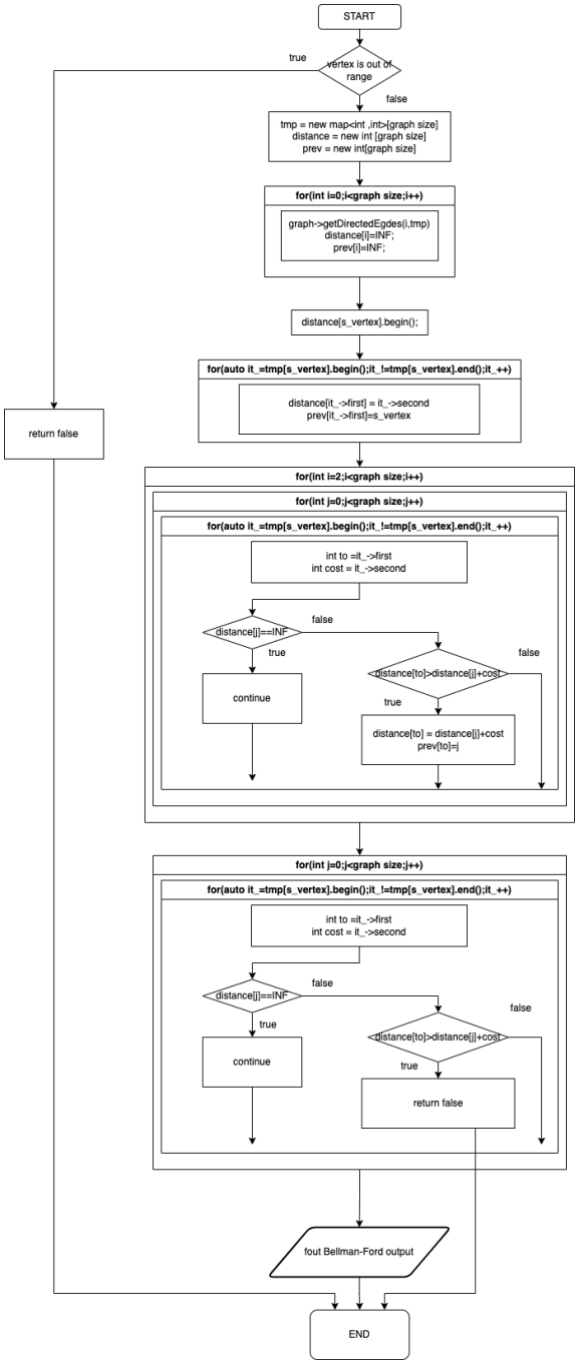
Kruskal 알고리즘에서 정렬을 사용할 때 사용하는 정렬 알고리즘으로 quicksort 함수를 호출해서 불러온 정렬 범위가 6이하인 경우 insertion sort를 진행하고 아니라면 partition을 나눠 정렬한 후에 quicksort를 재귀 호출한다. insertionSort 함수 같은 경우 low+1의 값을 임의로 뽑고 해당 위치 전부터 순차적으로 내려가며 비교하고 임의로 뽑은 값이 더 크면 그 자리에 넣는다. Partition에서는 주어진 범위에서 제일 처음 자리의 값을 pivot으로 두고 pivot을 기준으로 작은 경우 i++ pivot을 기준으로 큰 경우 j--를 각각 진행하고 i와 j가 구해진 경우 각 자리를 swap 해준다. 반복문이 종료되면 pivot 자리와 가장 큰 자리의 값과 자리를 swap을 해준다.

8) GraphMethod.cpp function Dijkstra



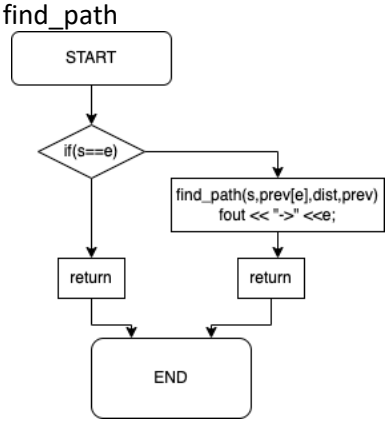
Dijkstra 알고리즘에서는 시작점을 입력 받고 해당 시작점부터 각 vertex까지의 도달 여부 또는 도달 경로를 출력한다. 각 거리는 distance 배열에 업데이트하고 이전 노드는 prev 배열에 업데이트 한다. 처음의 distance와 prev는 모두 무한의 값으로 두어서 연결되지 않았음을 뜻하게 한다. 이때 priority queue로 방문한 곳을 push하고 pop하게 한다. Dijkstra는 음수의 가중치를 처리하지 못하기 때문에 가중치가 음수인 경우 return false한다. distance 업데이트 할 때는 이전 vertex+다음 vertex의 가중치가 직선의 거리보다 짧을 경우 한다. 후에 경로를 역순으로 출력해야 하는 데 이는 뒤에 find_Path 함수를 선언하여 사용한다.

9) GraphMethod.cpp function Bellmanford



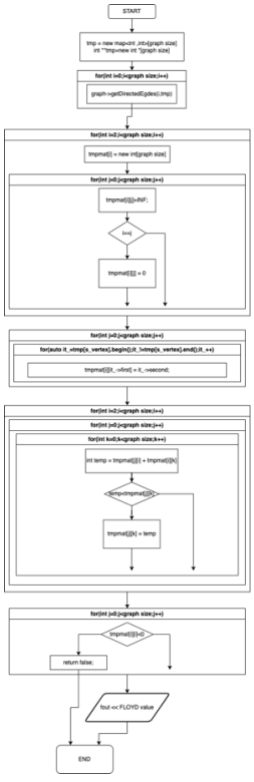
Bellman-Ford 알고리즘은 Dijkstra처럼 모든 distance 배열과 prev 배열을 무한의 값으로 초기화한다. 후에 distance[s_vertex]값은 0으로 둔다. 이 과정은 강의자료 내용 중 k=0 번째 과정과 같다. 다음에는 k=1 번째 과정으로 처음 vertex와 연결된 다음 vertex를 getdirectededges를 통해 가져와서 넣는다. 그 이후에는 k=2의 과정부터 k=max 까지의 과정을 반복하여 진행하는데 만약 distance[j]가 무한이라면 넘어가고 distance[to]>distance[j]+cost보다 큰 경우 distance 값을 업데이트 하고 이전 vertex의 위치도 업데이트 한다. Bellman ford 는 음수의 가중치도 사용하기 때문에 음수 사이클이 발생할 경우가 생기는 데 이때는 false를 반환하는데 이를 판단하기 위해서는 위 과정이 끝난 후 다시 bellman ford를 진행했을 때 값이 업데이트 되면 문제가 생긴 것이다.

10) GraphMethod.cpp function find_path



find_path 함수로 start vertex와 end vertex를 입력받아서 같은 경우 그냥 return 하고 아닌 경우 find_path를 recursion한 후에 e의 값을 출력한다.

11) GraphMethod.cpp function FLOYD



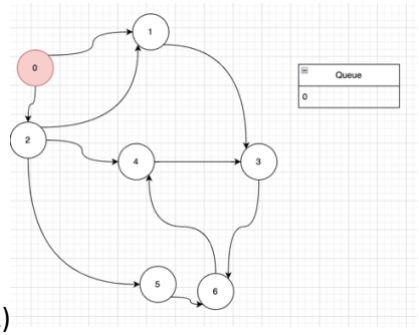
Floyd 알고리즘은 임의의 matrix에 무한의 값을 넣은 후에 인접한 vertex의 가중치를 넣는다. 이때 자기 자신으로의 경로는 0으로 초기화 한다. Bellman-Ford와 마찬가지로 경로상의 거리가 이전 vertex의 가중치 + 다음 vertex의 가중치보다 큰 경우 새로운 값으로 업데이트를 한다. 이것도 음수 사이클이 발생할 수 있는데 자기 자신의 값이 0이 아닌 경우 음수 사이클이 발생한 것이므로 예외처리를 해준다.

3. Algorithm

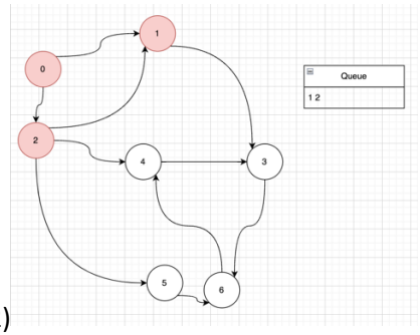
A. BFS

BFS 구현에는 queue를 사용했고 queue q에 vertex값을 push하고 while문을 실행하였다, int 형 변수 cur에 front를 저장하고 q를 pop하였다. cur의 값을 출력하고 cur에 대해 getAdjacentEdges를 사용하여 tmp의 map stl에 인접 vertex를 저장해 가져왔다. tmp의 first vertex에 방문하지 않았다면 q에 tmp의 first 값을 push하고 visited 배열의 해당 위치를 true로 업데이트 한다. 이를 반복하여 BFS의 규칙에 따라 출력한다. 아래는 과제 예시를 BFS로 진행했을 때의 예시이다.

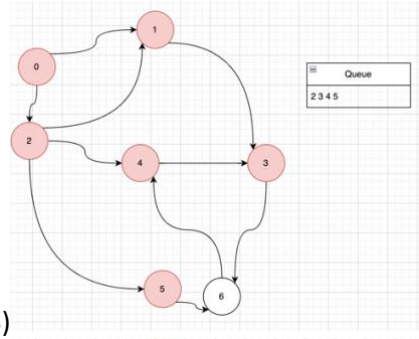
1)



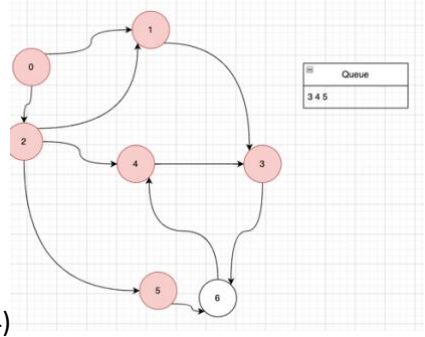
2)



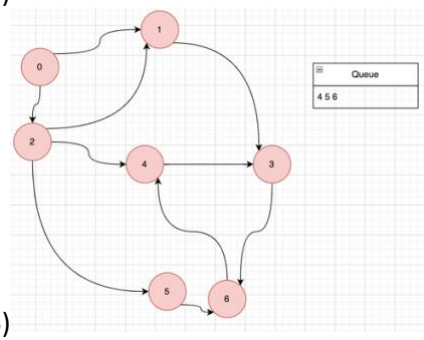
3)



4)



5)

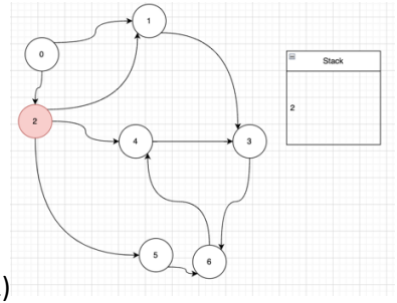


6)queue의 나머지 순서대로 출력

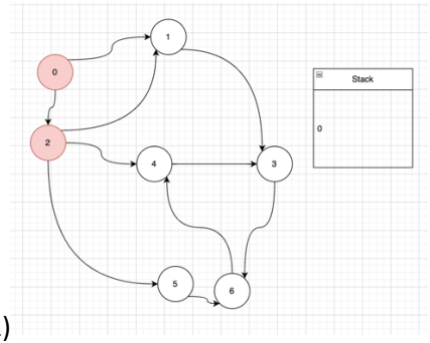
B. DFS

DFS 알고리즘은 BFS와 다르게 stack을 사용하고 stack s에 vertex값을 push하고 while문을 실행한다. int 형 변수 cur에 s.top()을 저장하고 cur에 대해 getadjacentEdges를 진행하여서 tmp에 인접 vertex를 저장한다. tmp의 first를 방문하지 않은 경우 다음 vertex값을 출력하고 방문한 vertex는 visited 배열에서 true로 저장하고 반복문을 탈출한다. 아니라면 다음 tmp의 first를 가져와 비교한다. 아래는 과제의 예시를 DFS로 진행하였을 때의 예시이다.

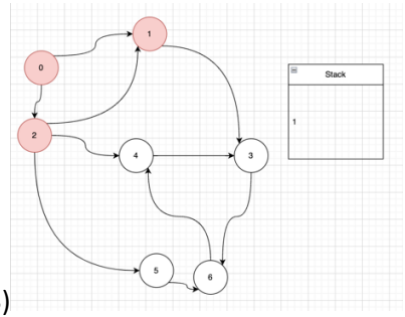
1)



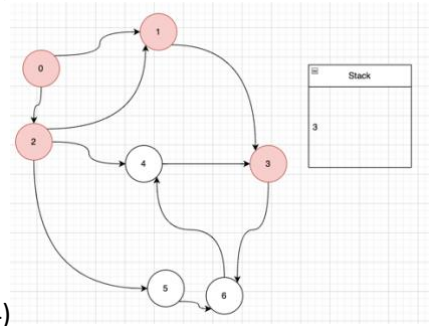
2)



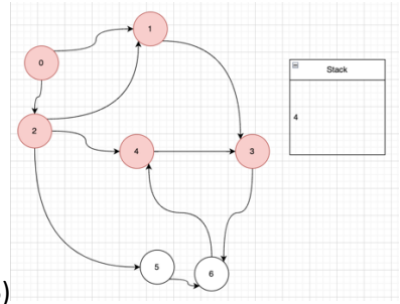
3)



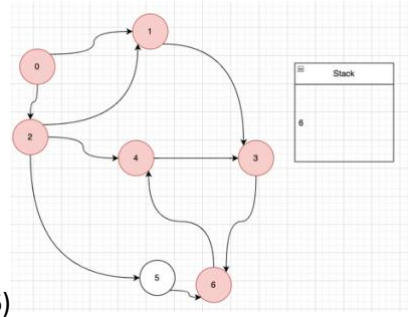
4)



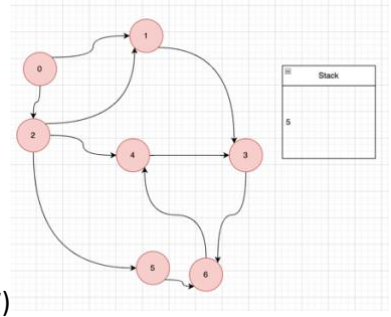
5)



6)



7)



8) 남은 스택 출력

C. DFS_R

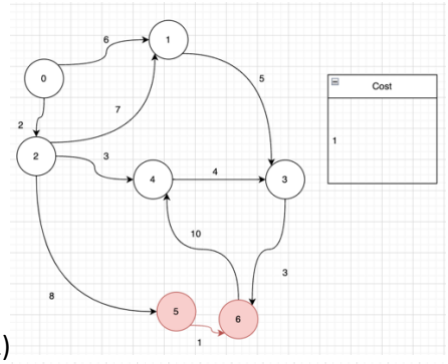
DFS_R은 DFS와 같은 출력을 나타내지만 stack을 사용하지 않고 recursion을 사용하는데 이는 stack 역할을 recursion으로 만들어서 visited[next]에 따라 recursion을 진행할지 안 할지 정하고 이미 방문한 경우 다음으로 넘어가서

이전의 다른 다음 vertex를 확인하며 다시 recursion을 한다.
알고리즘의 진행 순서는 DFS의 그림과 같다.

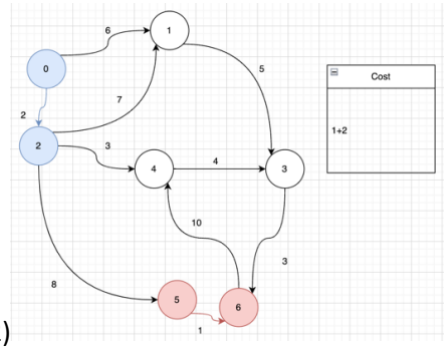
D. KRUSKAL

Kruskal 알고리즘은 모든 연결선들 중에서 가장 작은 값을 가진 연결선부터 사용하여 MST를 만드는 알고리즘으로 만약 연결선의 시작과 끝의 vertex의 조상 vertex 즉 root가 같은 경우 사이클이 발생하므로 해당 연결선은 넘어간다. 연결이 될 때마다 각 vertex의 부모 vertex를 업데이트하고 cost에 값을 더한다. 아래는 과제의 예시를 Kruskal 알고리즘을 사용했을 때의 예시이다.

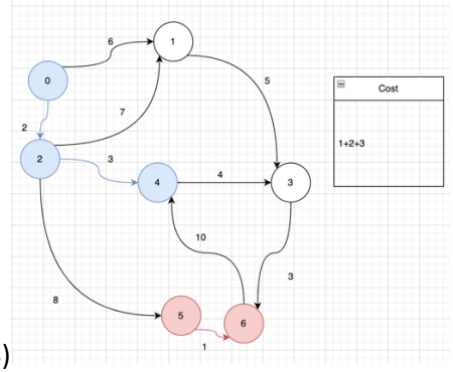
1)



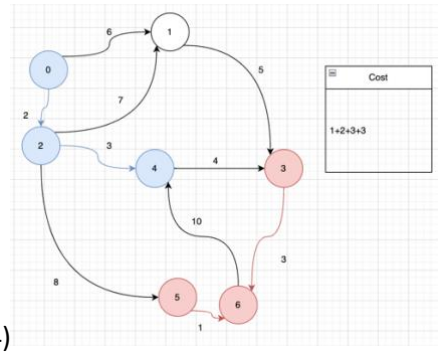
2)



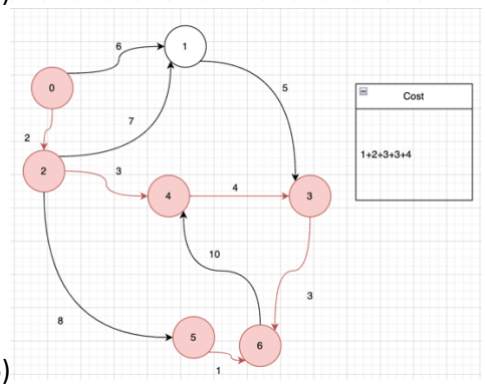
3)



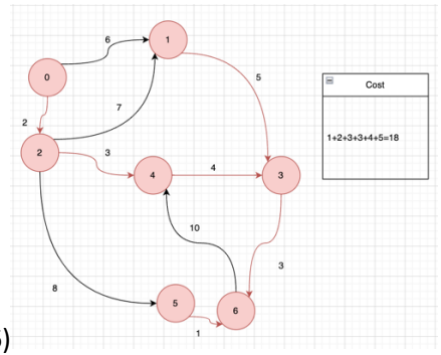
4)



5)



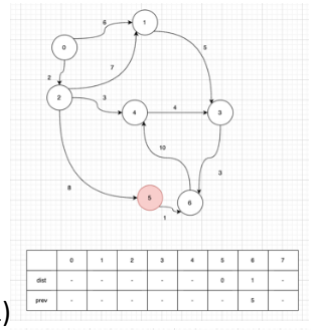
6)

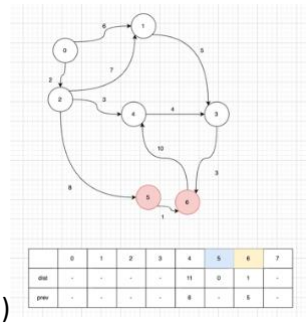


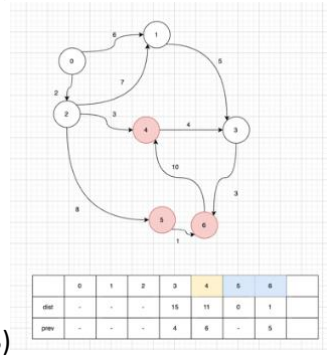
E. DIJKSTRA

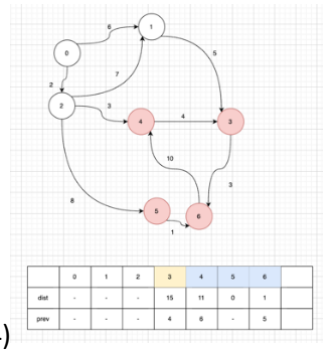
Dijkstra 알고리즘은 시작 vertex에서 방향을 가진 연결된 다음 vertex를 distance에 업데이트하고 previous에는 해당 다음 vertex 자리에 시작 vertex를 가리키도록 한다. 이제 distance 값 중에서 방문하지 않은 vertex를 기준으로 하여 그 vertex에서 다음 vertex를 가리키는 거리보다 distance에서 저장된 거리 + distance에서 저장된 이전의 거리보다 작은 경우 새롭게 distance 값을 업

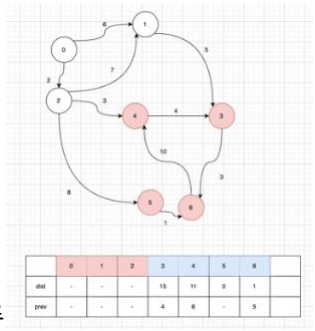
데이트 하고 previous에는 해당 다음 vertex자리에 해당 vertex 값을 가리키도록 하여 반복한다. 해당 작업이 완료되면 visited에 다음 vertex와 cost를 push 한다.

1) 

2) 

3) 

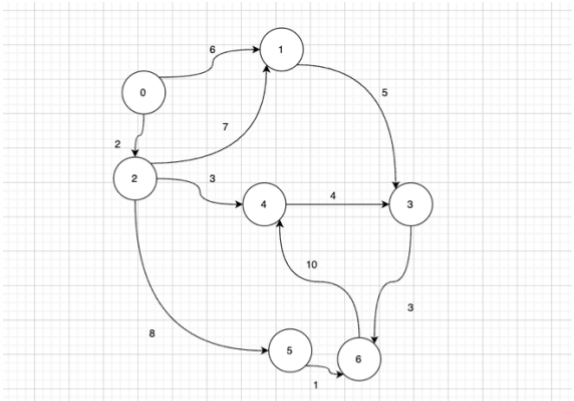
4) 

5) 최종 

위의 예시를 보면 5에서 출발했을 때 0,1,2 vertex는 도달하지 못하고 3, 4, 6 vertex는 도달하여 distance와 이전 노드를 잘 저장하였다.

F. BELLMAN-FORD

Bellman-Ford 알고리즘은 시작 vertex부터 시작하여 인접한 vertex의 distance 값을 업데이트하여 저장하고 이전의 vertex를 저장한다. 이후에 업데이트된 vertex들을 기준으로 하여 새로이 값을 업데이트 하는데 만약 distance가 무한인 경우 다음 vertex를 비교하러 가고 아니고 만약 distance[다음 vertex]가 distance[현재 vertex]+현재를 기준으로 다음 vertex의 distance 보다 큰 경우 distance[다음 vertex]에 값을 새로 업데이트 하고 이전 노드도 저장한다. 중요한 것은 위 과정을 진행하고 한번 더 값의 업데이트를 시도할 때 값이 변경된다면 음의 사이클이 발생 중이므로 return false 한다. 아래는 과제 예시를 Bellman-ford 했을 때 예시이다.



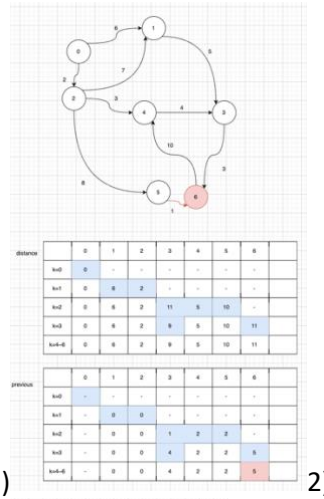
distance

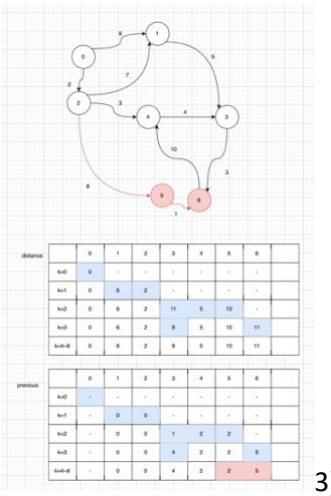
	0	1	2	3	4	5	6	
k=0	0	-	-	-	-	-	-	
k=1	0	6	2	-	-	-	-	
k=2	0	6	2	11	5	10	-	
k=3	0	6	2	9	5	10	11	
k=4~6	0	6	2	9	5	10	11	

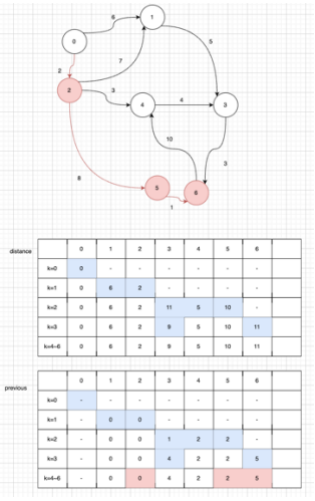
previous

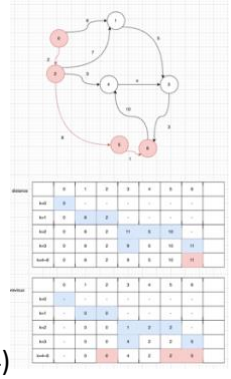
	0	1	2	3	4	5	6	
k=0	-	-	-	-	-	-	-	
k=1	-	0	0	-	-	-	-	
k=2	-	0	0	1	2	2	-	
k=3	-	0	0	4	2	2	5	
k=4~6	-	0	0	4	2	2	5	

이를 바탕으로 0부터 6까지의 가장 짧은 경로를 구하면

1) 

2) 

3) 

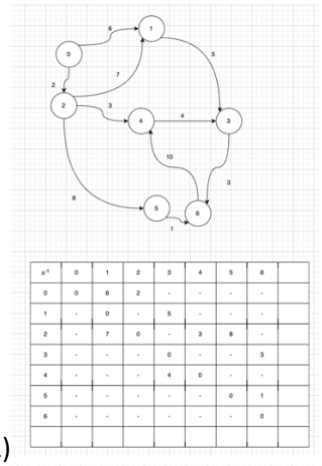
4) 

최단 경로는 0->2->5->6이고 cost는 11이다.

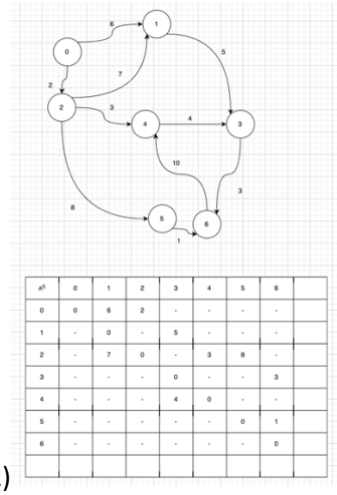
G. FLOYD

FLOYD도 Dijkstra나 Bellman-Ford와 마찬가지로 경로를 따라가서 최소값을 찾아내는 알고리즘으로 해당 그래프 사이즈 만큼 반복하되 각 루프마다 해당 vertex를 지나서 최소값을 만들 수 있는 수를 찾아서 업데이트한다. 경로가 연결되지 않으면 무한으로 저장하여 연결되지 않았음을 뜻하게 하고 후에 출력할 때는 x로 출력하도록 한다. 아래는 FLOYD 알고리즘을 실행했을 때의 결과이다.

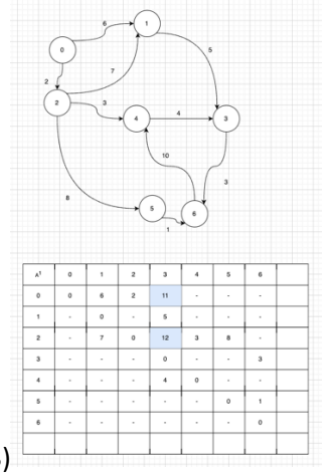
1)



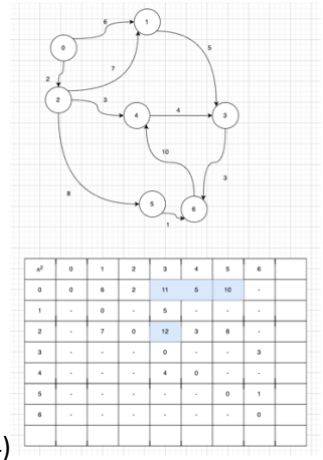
2)



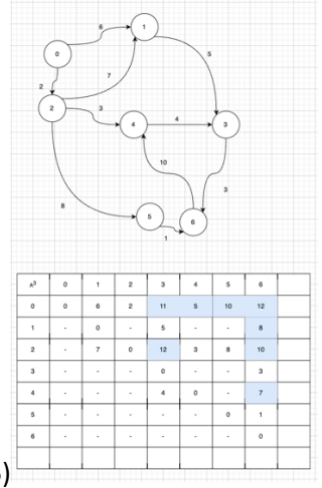
3)



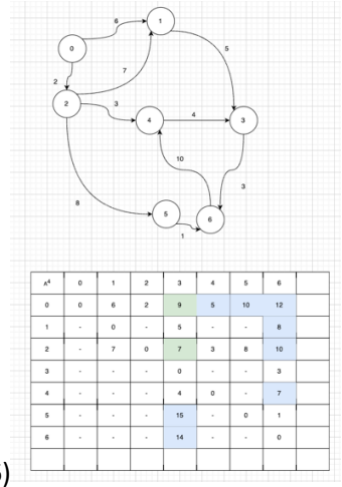
4)

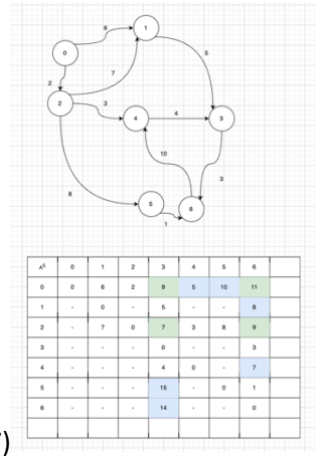


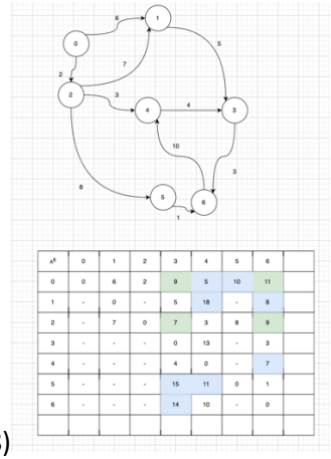
5)



6)



7) 

8) 

H. Quick Sort

Sorting할 개수가 7이상인 경우 Quick sort를 진행하는데 이 때 제일 왼쪽 값을 pivot으로 두고 왼쪽부터 pivot보다 작은 수와 오른쪽부터 pivot보다 큰 수를 계속해서 자리를 바꾸고 각 위치가 교차 되는 경우 반복문을 종료한다. 반복문이 종료된 이후에는 pivot과 피벗보다 작은 것 중 가장 오른쪽의 것과 자리를 바꾼다.

I. Insertion Sort

Sorting할 개수가 6 이하인 경우 Insertion Sort를 진행하는데 이 때 가장 왼쪽에서 두 번째부터 추출하여 값을 왼쪽부터 차례로 비교한다. 만약 추출한 값이 비교 대상보다 큰 경우 해당 위치의 자리로 두고 아닌 경우 비교 대상을 오른쪽으로 옮기고 추출한 값을 기존의 비교대상 자리로 위치하게 한다. 이 과정을 반복하여 다음 순서의 값도 추출하여 왼쪽과 비교하여 오름차순으로 정렬한다.

4. Result Screen

A. CASE1

1) Command.txt

```
LOAD graph_M.txt
PRINT
BFS 0
DFS 2
DFS_R 2
KRUSKAL
DIJKSTRA 5
BELLMANFORD 0 6
FLOYD
EXIT
```

2) graph_M.txt

```
M
7
0 6 2 0 0 0 0
0 0 0 5 0 0 0
0 7 0 0 3 8 0
0 0 0 0 0 0 3
0 0 0 4 0 0 0
0 0 0 0 0 0 1
0 0 0 0 10 0 0
```

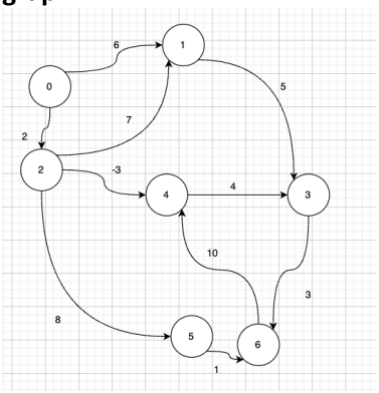
3) log.txt

```
==========LOAD==========
Success
========================
==========PRINT==========
      [0]   [1]   [2]   [3]   [4]   [5]   [6]
[0]   0     6     2     0     0     0     0
[1]   0     0     0     5     0     0     0
[2]   0     7     0     0     3     8     0
[3]   0     0     0     0     0     0     3
[4]   0     0     0     4     0     0     0
[5]   0     0     0     0     0     0     1
[6]   0     0     0     0     10    0     0
========================
==========BFS==========
startvertex: 0
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
======================
==========DFS==========
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
======================
==========DFS_R==========
startvertex: 2
2 -> 0 -> 1 -> 3 -> 4 -> 6 -> 5
======================
==========Kruskal==========
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)

cost: 18
========================
```

```
======Dijkstra======
startvertex: 5
[0] x
[1] x
[2] x
[3] 5 -> 6 -> 4 -> 3 (15)
[4] 5 -> 6 -> 4 (11)
[6] 5 -> 6 (1)
======Bellman-Ford======
0 -> 2 -> 5 -> 6
cost: 11
======================
==========FLOYD==========
      [0]   [1]   [2]   [3]   [4]   [5]   [6]
[0]   0     6     2     9     5     10    11
[1]   x     0     x     5     18    x     8
[2]   x     7     0     7     3     8     9
[3]   x     x     x     0     13    x     3
[4]   x     x     x     4     0     x     7
[5]   x     x     x     15    11    0     1
[6]   x     x     x     14    10    x     0
========================
```

4) graph



case1은 제안서에서 주어진 예시를 바탕으로 LOAD, PRINT, BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMANFORD, FLOYD를 진행하였을 때의 결과이다. LOAD가 정상적으로 작동하여 PRINT도 Matrix처럼 출력되었으며 BFS가 0부터 시작할 때 0->1->2->3->4->5->6처럼 진행됨을 볼 수 있다. DFS와 DFS_R 모두 2부터 시작할 때 2->0->1->3->4->6->5의 경로를 가졌다. KRUSKAL 또한 최소 값을 잘 출력하였으며, 해당 값일 때 연결을 확인했다. DIJKSTRA에서는 5에서는 0,1,2 vertex와 연결이 되지 않고, 3,4,6 vertex와는 연결이 되었다. 본인의 vertex는 출력하지 않도록 했다. BELLMANFORD는 미리 배열에 해당 경로의 최단 거리를 정리하고 이를 이용해 경로를 출력하도록 하여 0부터 6의 경로를 출력하고 비용을 계산하였다. 마지막으로 FLOYD는 연결된 것은 가중치 합으로 안 된 것은 x로 정상적으로 출력되었다.

B. CASE2

1) Command.txt

```
LOAD graph_L.txt
PRINT
BFS
DFS
DFS_R
KRUSKAL
DIJKSTRA
BELLFORDMAN
FLOYD
EXIT
```

2) graph_L.txt

```
L
7
0
1 6
2 2
1
3 5
2
1 7
4 3
5 8
3
6 3
4
3 4
5
6 1
6
4 10
```

3) **log.txt**

```
==========LOAD==========
Success
========================
==========PRINT==========
[0] -> (1,6) -> (2,2)
[1] -> (3,5)
[2] -> (1,7) -> (4,3) -> (5,8)
[3] -> (6,3)
[4] -> (3,4)
[5] -> (6,1)
[6] -> (4,10)

========================
======== ERROR ========
300
======================
======== ERROR ========
400
======================
======== ERROR ========
500
======================
==========Kruskal==========
[0] 2(2)
[1] 3(5)
[2] 0(2) 4(3)
[3] 1(5) 4(4) 6(3)
[4] 2(3) 3(4)
[5] 6(1)
[6] 3(3) 5(1)

cost: 18
==========================
======== ERROR ========
700
======================
======== ERROR ========
800
======================
==========FLOYD==========
       [0]    [1]    [2]    [3]    [4]    [5]    [6]
[0]    0      6      2      9      5      10     11
[1]    x      0      x      5      18     x      8
[2]    x      7      0      7      3      8      9
[3]    x      x      x      0      13     x      3
[4]    x      x      x      4      0      x      7
[5]    x      x      x      15     11     0      1
[6]    x      x      x      14     10     x      0
```

case 2는 graph_L.txt를 잘 읽고 PRINT 했을 때 정상적으로 출력됨을 확인했고 숫자를 입력하지 않음으로써의 오류를 확인하였다.

C. case3

1) command.txt

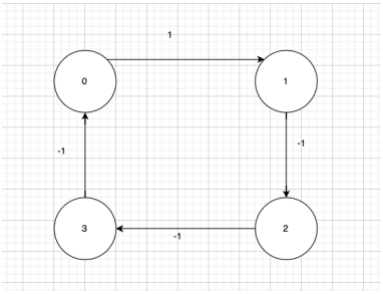
```
LOAD graph_M.txt
PRINT
DIJKSTRA 1
BELLMANFORD 0 3
FLOYD
EXIT
```

2) graph_M.txt

```
M
4
0 1 0 0
0 0 -1 0
0 0 0 -1
-1 0 0 0
```

3) log.txt

```
==========LOAD==========
Success
========================
==========PRINT==========
        [0]     [1]     [2]     [3]
[0]     0       1       0       0
[1]     0       0       -1      0
[2]     0       0       0       -1
[3]     -1      0       0       0
========================
======== ERROR ========
700
======================
======== ERROR ========
800
======================
======== ERROR ========
900
======================
```

4) 

case3는 음의 사이클이 발생했을 때 오류코드 발생을 확인하였다.

KRUSKAL은 DIJKSTRA는 음수가 입력되면 MST를 만들지 못하므로 에러이고 BELLMANFORD와 FLOYD는 음수 사이클이 발생하여 오류코드가 발생하였다.

5. Consideration

이번 프로젝트에서는 graph를 불러와 BFS, DFS, DFS_R, KRUSKAL, DIJKSTRA, BELLMANFORD, FLOYD 알고리즘을 이용해 탐색 및 최단 경로를 구하여 출력하였다. 프로젝트를 하기 전에는 최단 경로를 찾는 방법을 이론상으로만 알고 직접 구현한 적은 없었기 때문에 구현의 어려움을 알지 못했으나 따져야 할 것이 많았다. 특히 BELLMANFORD나 FLOYD의 음수 사이클을 허용하지 않는 것을 어떤 기준으로 판별해서 오류 코드를 출력해야 할지 많은 생각을 하였다. 다만 BFS, DFS에서 queue나 stack을 이용해서 구현하는 것은 오히려 쉽게 데이터를 처리할 수 있게 되어서 빠른 시간 안에 구현할 수 있었다.

우분투로 코드를 옮겨 파일을 실행시켰을 때 처음에는 잘 작동하였으나 graph_M.txt나 graph_L.txt를 수정하여서 실행하면 “std::logic_error”가 발생하였고 이를 확인하기 위해 valgrind로 run을 실행하자 stoi 즉 string을 integer로 바꾸는 것에서 문제가 발생하였다. 알고 보니 txt를 읽는 과정에서 NULL을 읽고 stoi를 진행하자 문제가 발생한 것이었다. 어디서 잘못된 NULL을 읽는지는 파악하지 못했으나 NULL 일 경우 continue를 진행하여 해결하였다.

저번 과제에서는 하나의 파일에 모든 class와 function을 제작하여 나중에 파일

로 나누는 형식으로 진행했으나 컴파일 에러가 발생하였기에 다시 파일들을 skeleton코드에 맞춰 진행하여 진행하였고, 컴파일 에러에 유의하여 제작하여서 다행히 컴파일 에러를 발생시키지 않고 프로젝트를 마무리할 수 있었다.