Data Structure Project

Project #3

담당교수 : 이기훈 교수님

제출일 : 2018. 12. 12.

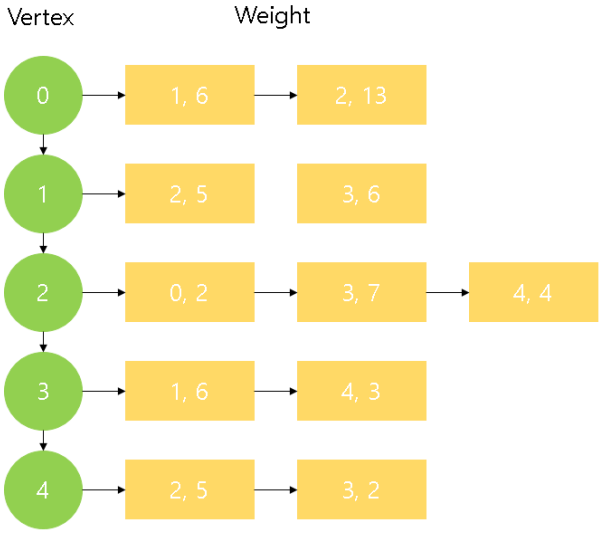
학과 : 컴퓨터정보공학부

학번 : 2015722025

이름 : 정용훈

1. Introduction

Data Structure 수업의 마지막 Project는 그래프의 구현과 각종 그래프 관련 알고리즘, sorting구현에 목적이 있습니다. Project의 내용은 각 Vertex즉, 도시가 지정되어 있다면 각각의 도시끼리 연결된 거리, Weight정보가 text파일로 제공되며, 해당 Matrix data를 기반으로 linked list를 통해 자료를 구현하고 최단거리를 구하는 각각의 여러 알고리즘을 사용하여 결과값을 도출하는 프로그램입니다. 우선 load명령을 통하여 text data를 기반으로 그래프 연산에 사용될 자료를 다음과 같이 구축합니다.



해당 linked list는 Matrix로 제공된 data를 받아 load를 통하여 구축되는 data를 가시적으로 나타낸 것입니다. 해당 linked list를 통하여 각종 그래프 연산을 구현하게 되는데 구현되는 연산은 다음과 같습니다. 각각 연산의 자세한 알고리즘은 알고리즘 항목에서 다룹니다.

-DFS

해당 연산은 Depth first search의 약자로 깊이를 우선으로 탐색을 진행하는 알고리즘입니다. Project내에서 start vertex와 end vertex를 parameter로하여 해당 연산을 실행하면, start vertex부터 end vertex까지의 경로와 거리를 구하는 연산입니다.

-DIJKSTRA

해당 알고리즘은 다익스트라 알고리즘으로써 shortest path를 찾는 알고리즘으로 정의 되어있습니다. Project내에서는 start vertex와 end vertex를 parameter로하여 start vertex부터 end vertex까지의 최단경로와 거리를 구하는 연산입니다.

-DIJKSTRAMIN

해당 알고리즘은 위에서 설명한 다익스트라 알고리즘입니다. 다만 다른 방법으로 구현이 되어 명령이 나뉘게 되었는데, STL set을 사용하여 구현하는 것이 아니라 Min-Heap을 직접 구현하여 최단거리와 경로를 구하는 연산입니다.

-BELLMANGFORD

해당 알고리즘은 벨만포드 알고리즘으로 다익스트라 알고리즘과 동일하게 vertex와 vertex사이의 최단경로를 구할 수 있는 알고리즘입니다. 다만 다익스트라 알고리즘과 다른 점으로는 음수 weight가 포함되어도 계산이 가능합니다. 다만 음수 cycle이 생기면 함수를 실행할 때 마다 값이 음의 무한대로 수렴하기 때문에 cycle이 있는 경우 예외 처리가 필요합니다.

-FLOYD

해당 알고리즘은 플로이드 머셜 알고리즘입니다. 다익스트라와 벨만포드와 다르게 한 정점에서 다른 정점의 최단 거리만을 구하는 것이 아니라 모든 정점에 대하여 모든 최단 경로를 구해주는 알고리즘입니다. 결과화면 또한 Matrix로 표시되며 벨만포드 알고리즘과 마찬가지로 음수 weight가 포함되어 있어도 동작합니다. (음수 cycle은 예외처리가 필요합니다.)

다음으로는 해당 Project에서 사용되는 sort에 관련된 설명입니다. 위 설명한 연산을 통하여 shortest path를 구했다면 해당 path의 경로들을 수업시간에 배운 Insertion sort와 Quick sort를 이용하여 vertex번호에 따라 정렬하여 사용자에게 가시적으로 보여줍니다.

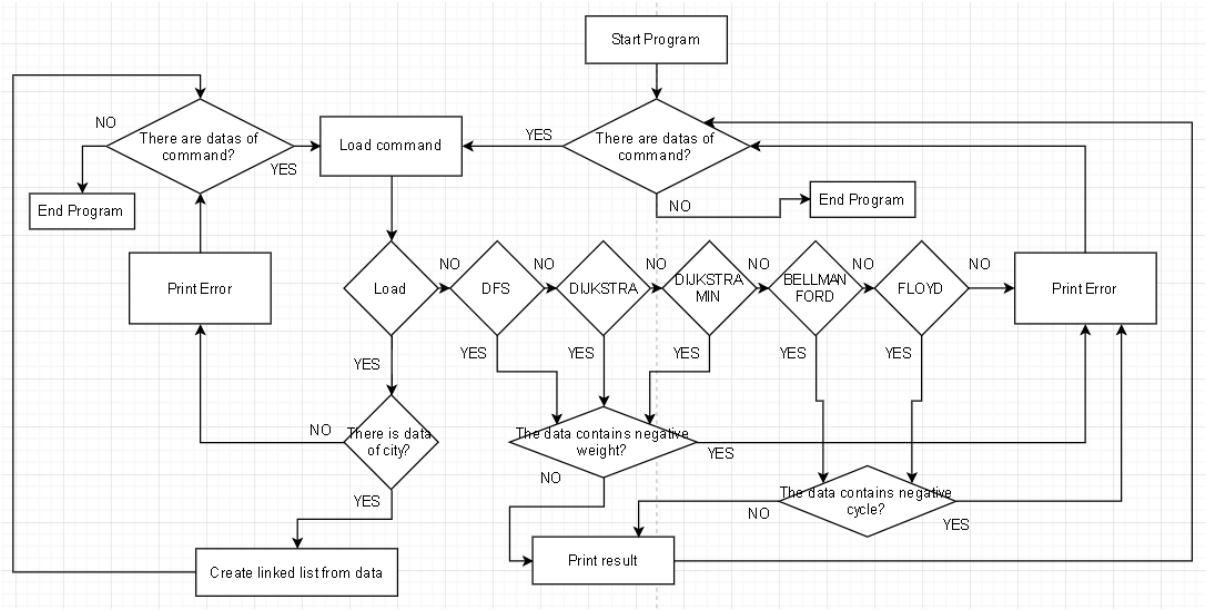
-Insertion sort

삽입 정렬로 버블 정렬과 같이 데이터를 비교하며 이동시키는 것이 아니라 데이터를 비교하되 해당 데이터가 들어가야하는 자리에 바로바로 정렬시켜주는 알고리즘입니다. Project의 조건으로 segment size가 7을 기준으로 작으면 해당 알고리즘을 실행합니다.

-Quick sort

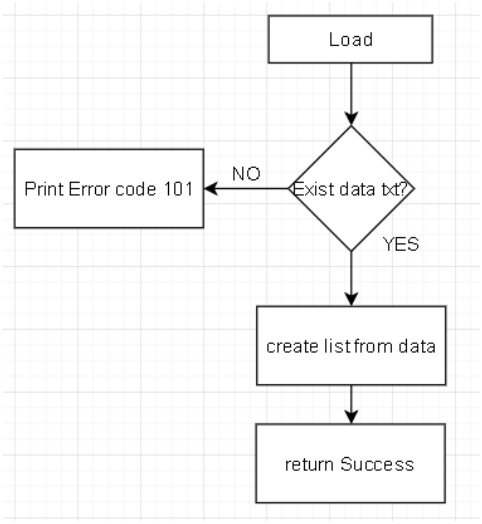
정렬 알고리즘 중에 가장 빠른 정렬이 가능한 Quick sort입니다. 자세한 알고리즘은 알고리즘 항목에서 설명하며, 해당 알고리즘을 사용하기 위해서는 pivot에 관련된 개념이 필요합니다. Project에서는 segment size가 7을 기준으로 보다 클 때 실행합니다.

2. Flow Chart



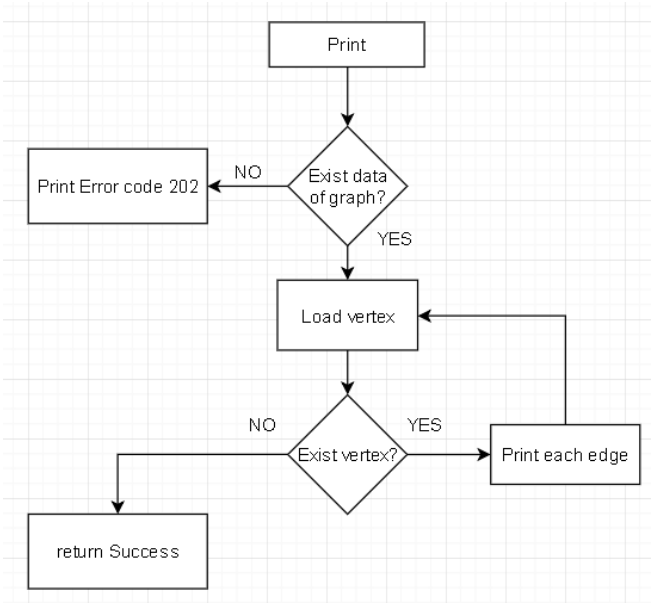
다음 Flow chart는 해당 프로그램의 전체적인 동작을 간단하게 나타낸 그림입니다. Program을 시작하고 해당 프로그램에서 EXIT명령이 따로 정의되어 있지 않기 때문에 command file의 끝을 End로 판단하게 됩니다. 그래서 불러올 command가 있는지 먼저 판단 후 command를 불러옵니다. 그 후 해당 커맨드에 맞는 함수를 동작 시킵니다. 우선 Load 명령은 data가 있는지 없는지 판단하며, Error code를 출력할지 말지 판단하게 됩니다. Data가 있다면 해당 data를 기반으로 linked list를 구현합니다. DFS와 다익스트라 함수는 구현된 List를 기반으로 음수 weight가 포함되어 있으면 Error code를 출력하게 됩니다. 마지막으로 벨만포드와 플로이드 머셜 함수는 음수 cycle이 있는지 없는지 판단하게 되며, cycle이 존재하면 Error code를 출력하도록 합니다.

LOAD



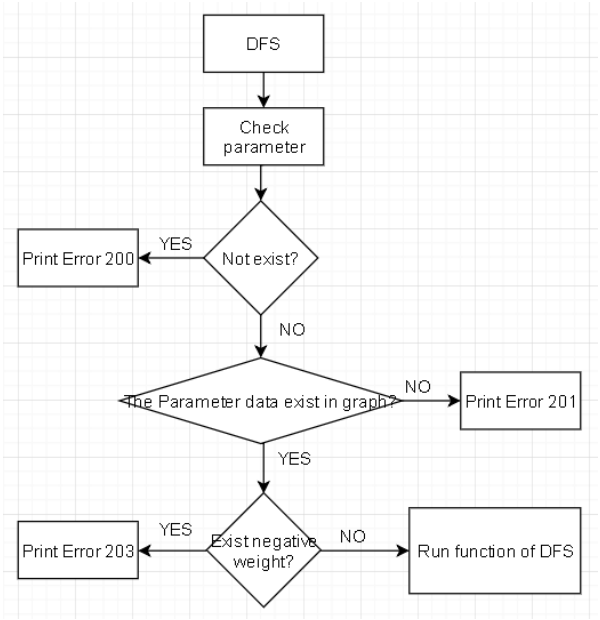
다음 그림은 LOAD에 대한 flow chart입니다. Load 명령이 실행된 후 data를 불러올 text파일을 판단하여 해당 text파일이 없으면 Error code를 출력 후 함수를 종료하게 됩니다. 정상적인 동작이 모두 실행되면 Success code를 return합니다.

PRINT



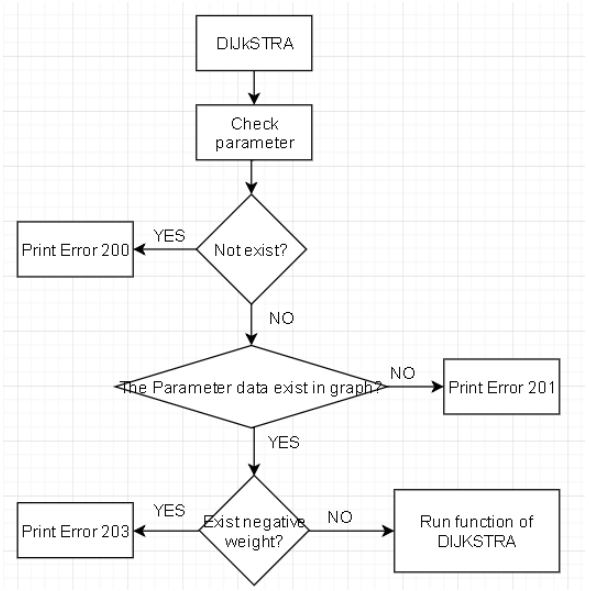
다음은 Print 함수에 대한 Flow chart입니다. List존재 여부를 판단한 후 없으면 Error code 200을 출력하고 존재한다면 각각의 Vertex를 순차적으로 접근하여 Edge를 모두 출력하는 방식입니다.

DFS



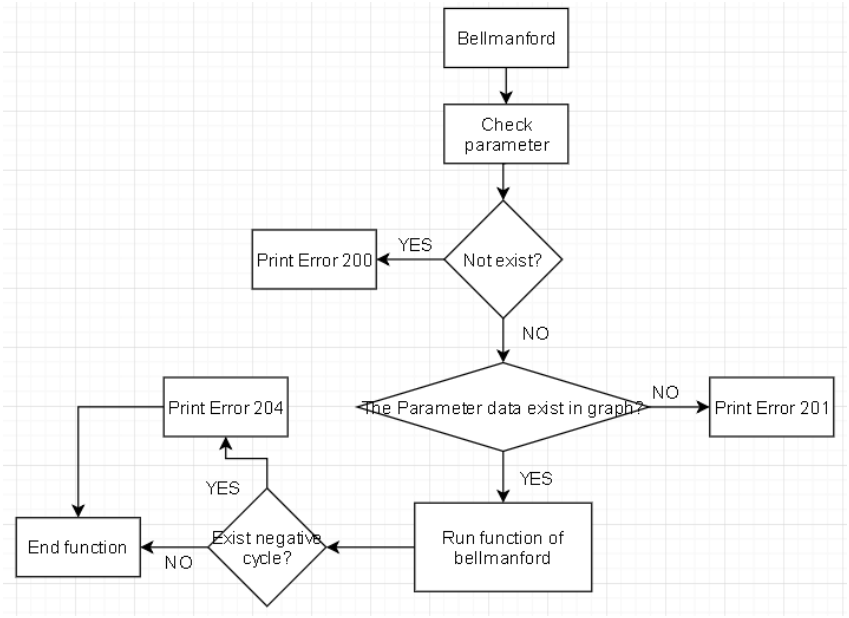
다음은 DFS의 Flow chart입니다. 명령어의 Parameter와 Graph에 포함되어 있는 정보를 Error를 판단하게 됩니다. 200은 parameter가 존재하지 않는 경우이며, 201은 parameter는 존재하지만 그래프가 포함하지 않는 Vertex이며, 203은 DFS의 특성상 음수 weight가 포함되면 안되므로 출력되는 Error입니다. 모든 검증을 통과하였다면 연산을 실행합니다.

DIJKSTRA & DIJKSTRAMIN



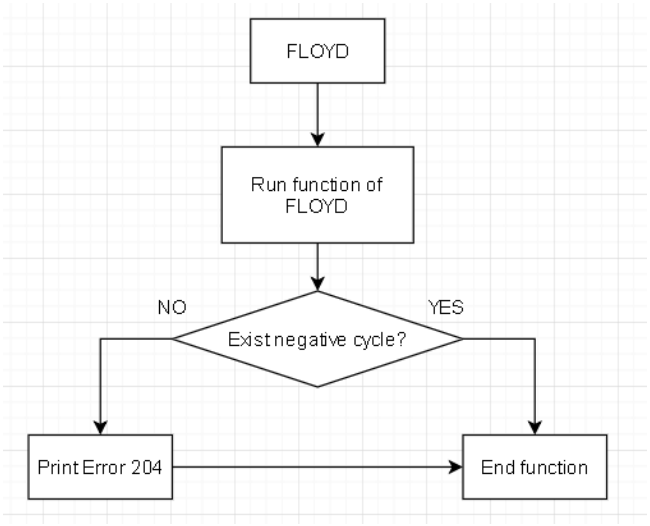
다익스트라의 set과 heap을쓴 명령어도 DFS와 동일한 Flow chart를 따르게 됩니다.

BELLMANFORD



다음은 벨만포드 명령어의 Flow chart입니다. 위 명령어들과는 다르게 음수 weight가 있어도 함수가 실행되야 하고 음수 cycle이 있으면 Error를 판단하게 됩니다. 나머지 Error에 대한 판단은 동일합니다.

FLOYD



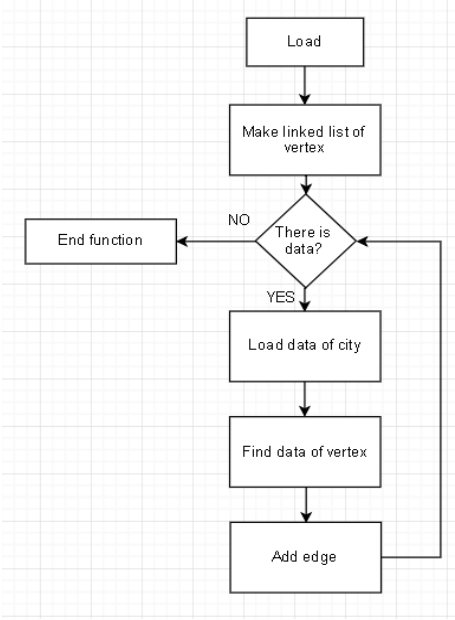
FLOYD명령에 대한 flow chart는 굉장히 단순합니다. parameter가 없기 때문에 그래프의 존재 유무만 판단하면 되고 음수 cycle에 대한 예외처리만 해주면 됩니다.

위 flow chart들의 가정은 그래프가 존재한다는 것입니다. 존재하지 않는다면 Error를 Print합니다.

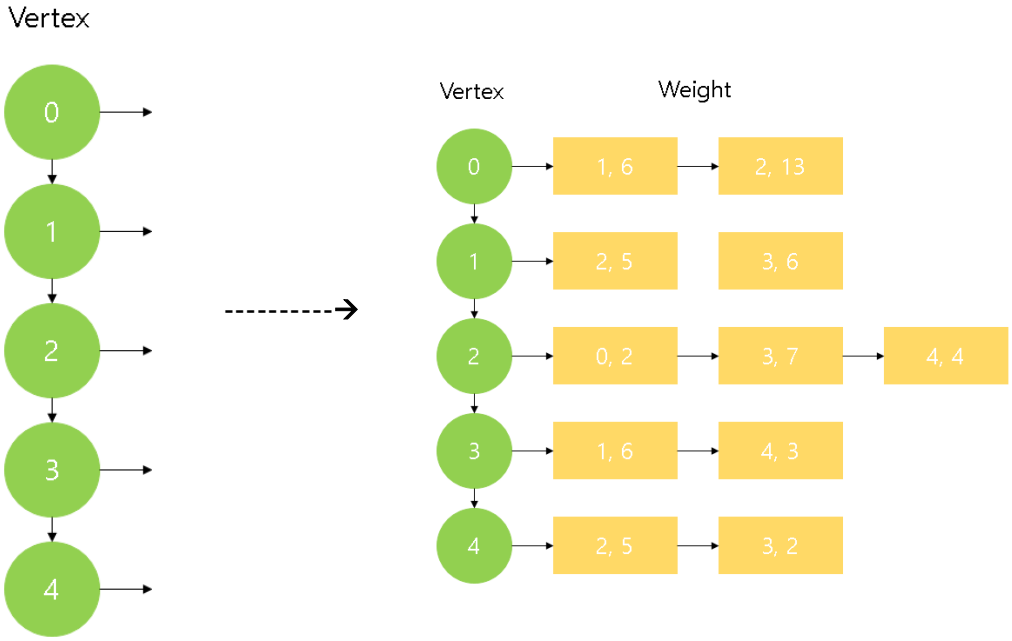
3. Algorithm

-Load

Load동작의 알고리즘은 기존 Linked list를 구현하는것과 같습니다. 아래 그림은 Load를 실행하는 flow chart입니다.

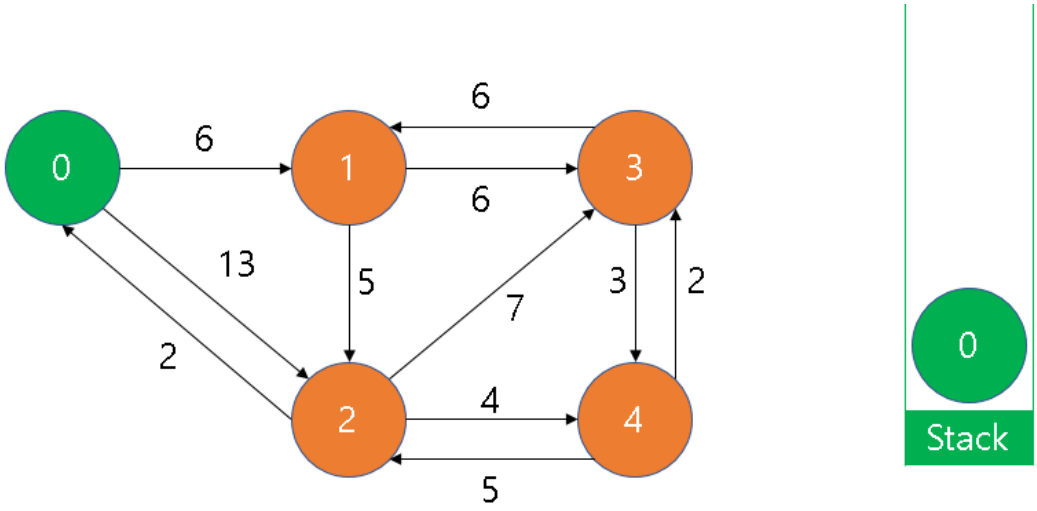


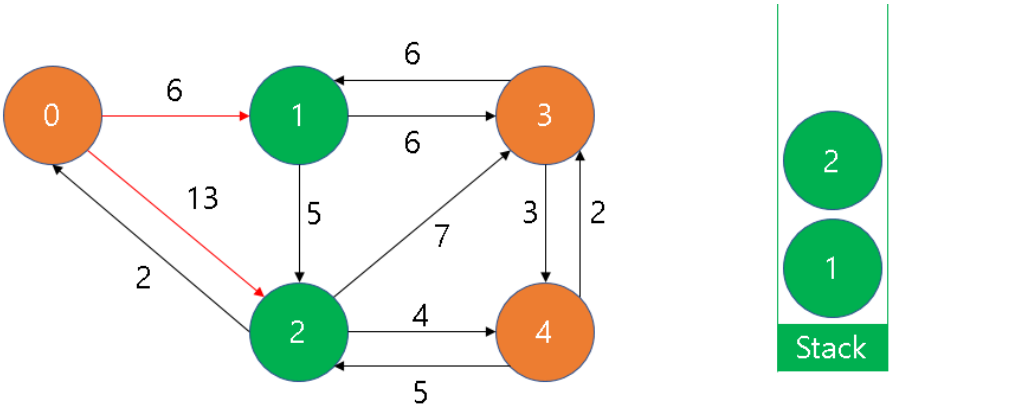
기본적으로 Vertex개수의 정보를 받아 0부터 순서대로 Vertex의 정보를 담은 Node를 List로 연결해주며 data를 통해 해당 Vertex에 맞는 edge를 생성하여 옆으로 list를 생성합니다. 구현되는 순서는 다음 그림과 같습니다.



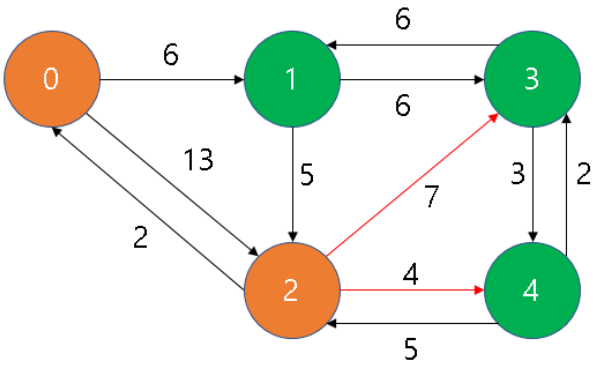
-DFS

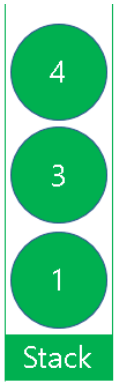
DFS는 Depth first search의 약자로써 깊이를 먼저 탐색하는 방법이다. 수업 중 배운 DFS와 조금 다른 알고리즘으로 동작이 진행되며, 해당 알고리즘은 후에 추가적으로 QnA 페이지를 이용하여 배운 알고리즘이다. DFS의 Project내에서의 동작은 다음과 같다.



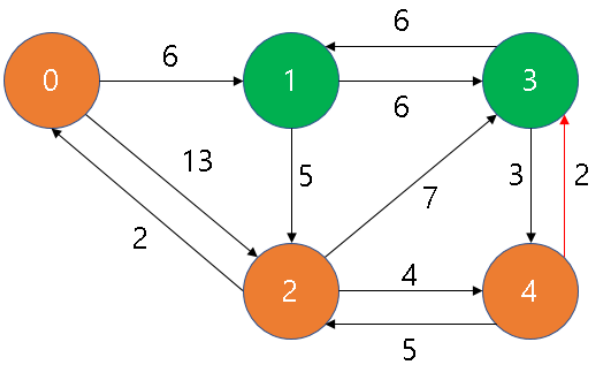


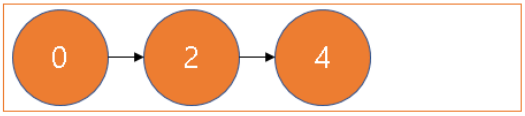
Path
Cost: 0



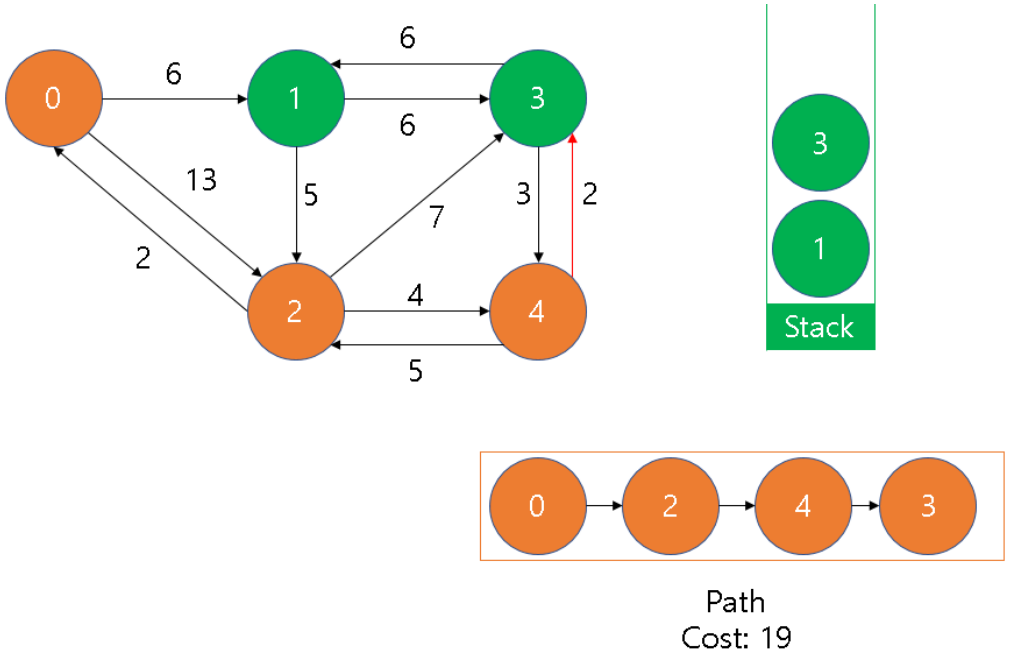


Path
Cost: 13





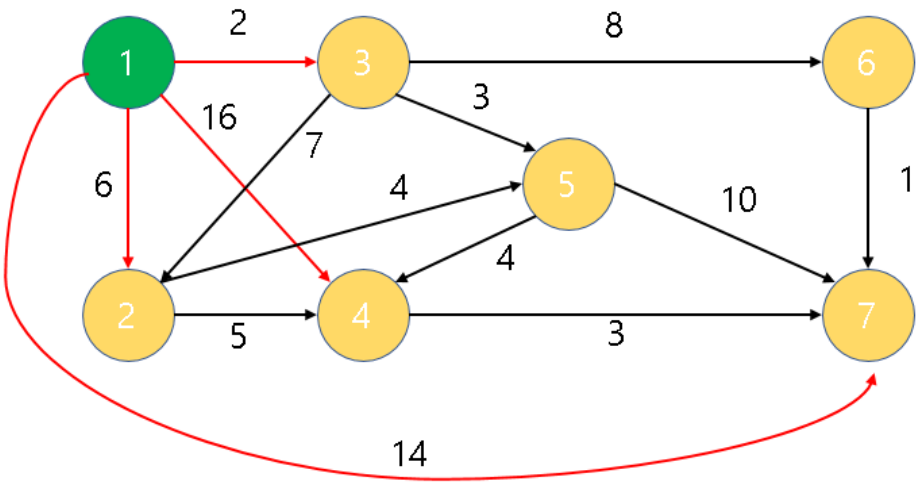
Path
Cost: 17



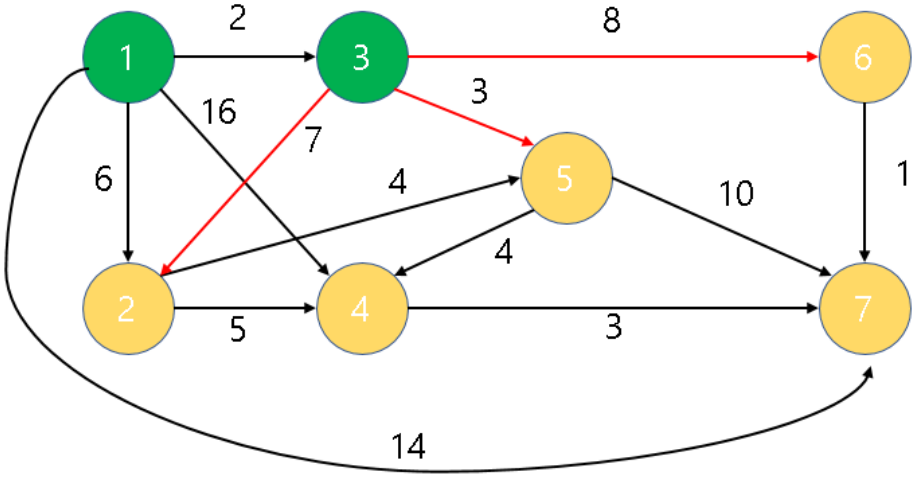
해당 결과화면은 이번 프로젝트에서 쓰인 data를 기반으로 작성하게 된 DFS의 예시입니다. BFS알고리즘과 거의 흡사하며 BFS에서 쓰인 Queue대신 Stack을 쓴 것이 특징입니다. 솔직히 개인적으로는 정확히 이해가 되지 않은 알고리즘으로써 고찰에서 한번 더 언급할 예정입니다.

-DIJKSTRA & DIJKSTRAMIN

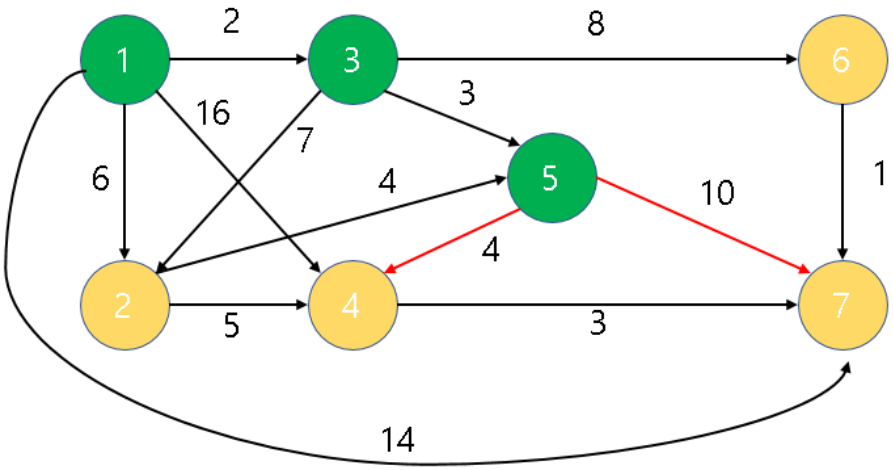
다익스트라 알고리즘은 parameter로 받은 start vertex와 end vertex의 정보를 기반으로 shortest path를 찾는 알고리즘입니다. 해당 연산의 알고리즘은 다음 그림과 같은 순서데로 알고리즘이 실행됩니다.



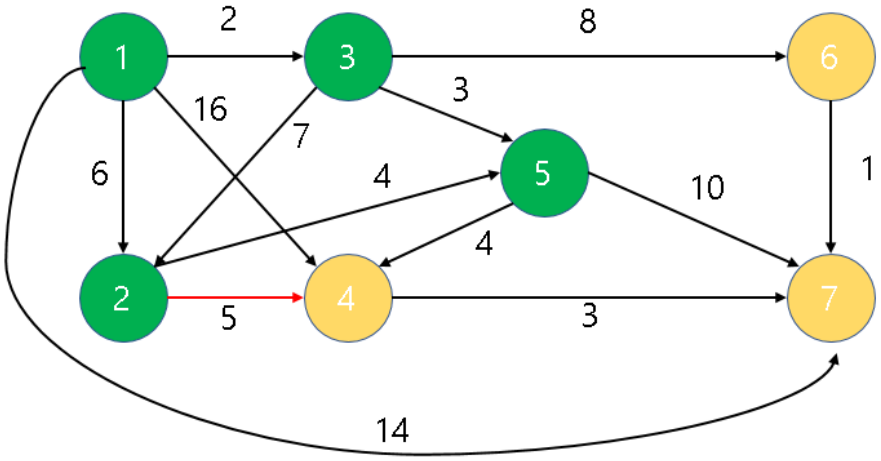
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	16	-	-	14
prev	-	1	1	1	-	-	1



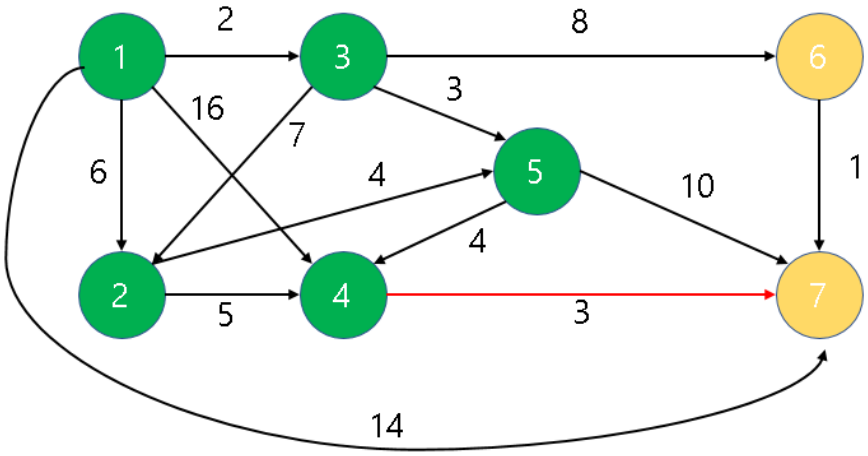
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	16	5	10	14
prev	-	1	1	1	3	3	1



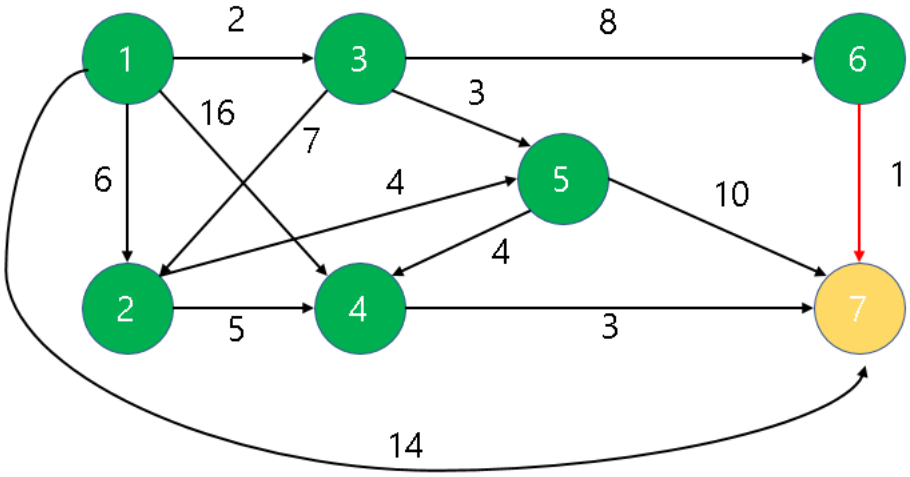
	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	14
prev	-	1	1	5	3	3	1



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	14
prev	-	1	1	5	3	3	1



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	12
prev	-	1	1	5	3	3	4

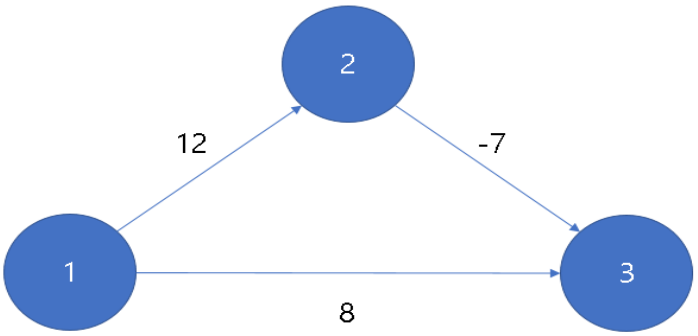


	[1]	[2]	[3]	[4]	[5]	[6]	[7]
dist	0	6	2	9	5	10	11
prev	-	1	1	5	3	3	6

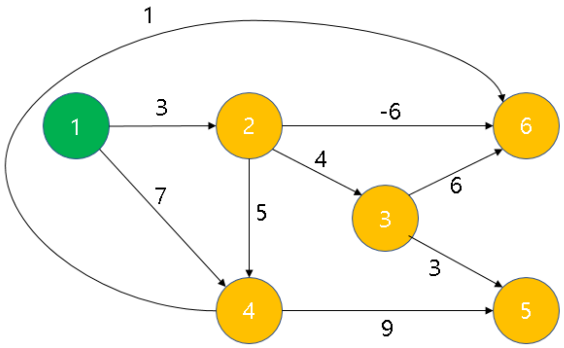
해당 알고리즘의 동작이 모두 끝난 상태이다. 그림에서는 Start Vertex가 1로 지정되어 있으며, 결과는 Vertex 1부터 각각의 정점까지의 최단거리가 표시되어 있다. Project내에서는 End Vertex정보를 Parameter로 받게 되며, 해당 정점까지의 경로와 거리만 구하게 되어있다.

-BELLMANFORD

다음으로는 벨만포드 알고리즘이다. 벨만포드 알고리즘은 다익스트라와 다르게 음수의 weight까지 포함하여 최단거리를 계산할 수 있다. 그 이유는 다음과 같다. N개의 vertex가 있을 때 Vertex부터 Vertex까지의 간선은 최대 N-1개를 가질 수 있게 된다. 음수 weight가 있을 경우 다익스트라는 다음 경우 때문에 계산이 불가하다.



다음의 경우 실제 최단 경로는 1->2->3 이지만, 다익스트라 알고리즘을 통하여 경로를 구하게 되면 weight에 따라 1->3을 먼저 실행하여 3을 방문해 버리기 때문에 Vertex 2를 방문한다고 하여도 3으로 가는 path를 최신화 할 수 없는 문제점이 생긴다. 이에 반해 벨만포드는 위에서 정의한 N-1개의 간선을 최대로 가질 수 있으므로 모든 경로에 따른 경우를 구하여 비교하여 최단경로를 구해주기 때문에 프로그램의 속도는 저하될 수 있어도 음수의 경우가 있어도 계산이 가능하다. 주의할 점으로는 음수 cycle이 생기는 경우는 예외 시킨다. 이유는 경로가 음의 무한대로 수렴하는 이유가 있다. 다음은 벨만포드 알고리즘의 동작 결과다.

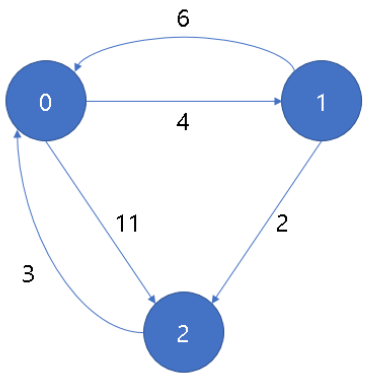


	1	2	3	4	5	6
0	0	-	-	-	-	-
1	0	3	-	7	-	-
2	0	3	7	7	16	8
3	0	2	7	7	10	8
4	0	2	6	7	10	8
5	0	2	6	7	9	8

d(V, K)

-FLOYD

FLOYD알고리즘은 앞선 두 알고리즘과 다르게 모든 정점에서 다른 모든 정점의 거리를 구할 수 있는 알고리즘이다. 음수 weight가 있어도 연산 가능하며 벨만포드와 마찬가지로 음수 Cycle이 존재하면 음의 무한대로 값들이 계속 수렴하기 때문에 존재해서는 안된다. 다음은 FLOY알고리즘의 동작 예시다.



$A^0[2][1]=\min(A^{-1}[2][1],\ A^{-1}[2][0]+A^{-1}[0][1])$

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	INF	0

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

$A^1[0][2]=\min(A^0[0][2],\ A^0[0][1]+A^0[1][2])$

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

$A^2[1][0]=\min(A^1[1][0],\ A^1[1][2]+A^0[2][0])$

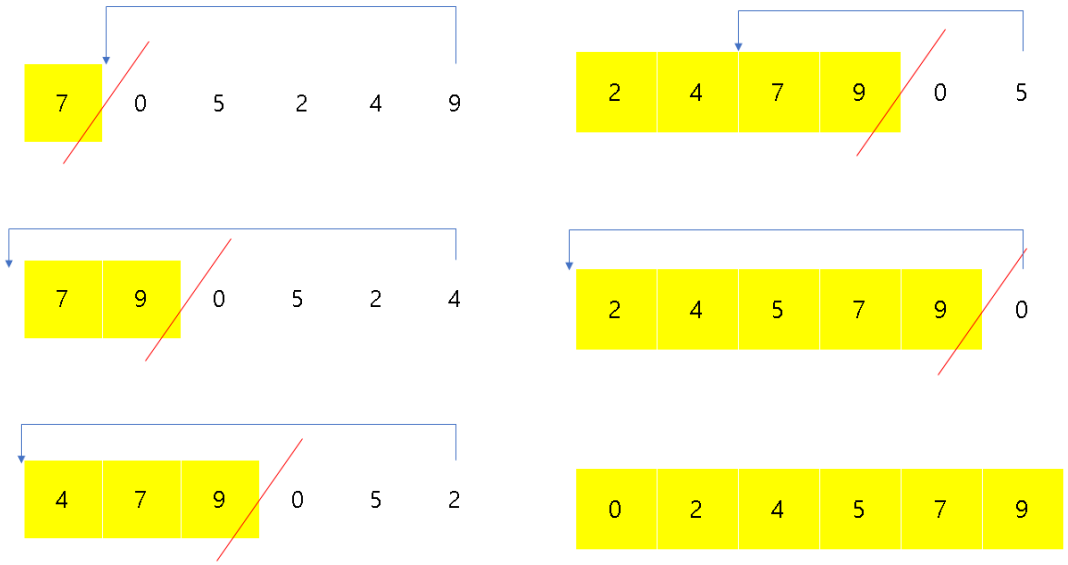
다음 동작의 예시는 아래와 같은 식으로 정리하여 나타낼 수 있다.

```
for(int k = 1; k <= n; k++)
  for(int i = 1; i <= n; i++)
    for(int j = 1; j <= j; j++)
      A[i][j] = min{A[i][j], A[i][k] + A[k][j]};
```

Program내에서 2차원 행렬로 table을 구성하면 다음과 같은 코드로 쉽게 구현할 수 있다. FLOY의 음수 cycle포함 판별 방법으로는 해당 알고리즘을 한번 더 실행하여 값이 update 되는 순간 음수 cycle이 있다고 판별하면 예외처리 또한 어렵지 않게 할 수 있다.

-Insertion sort

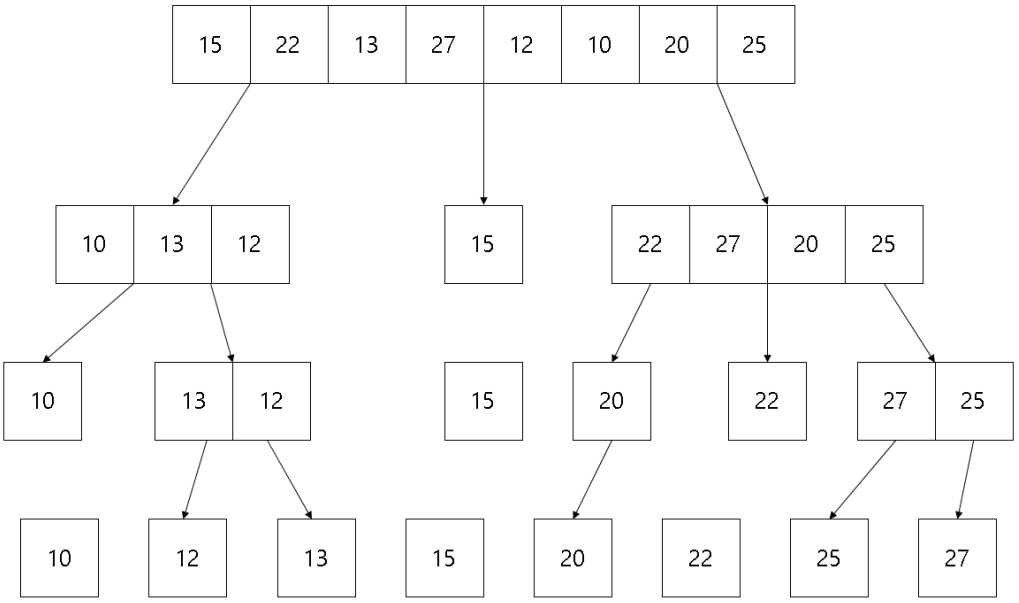
정렬의 한 종류로써 인접한 데이터와 비교하여 자리를 계속 바꾸는 것이 아니라 들어갈 위치를 바로 탐색하여 정렬이 완료된 부분은 계속 정렬 상태를 유지할 수 있도록 하는 알고리즘이다. 아래 그림은 삽입 정렬의 순서를 나타낸 것이다.



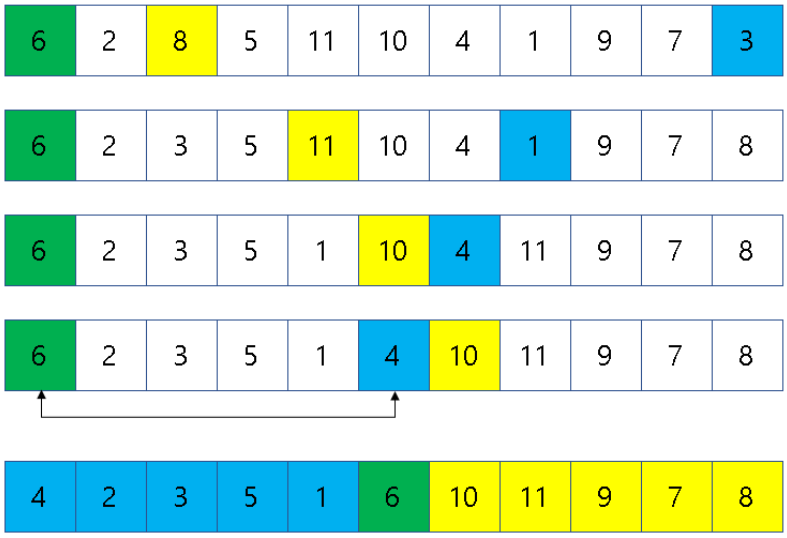
다음 그림과 같이 정렬된 부분은 계속 정렬된 상태를 유지하는 것을 확인할 수 있다. Segment size에 따라 정렬알고리즘들의 속도가 다르기 때문에 해당 프로젝트에서는 segment size가 7기준으로 작을 때 프로그램의 성능을 높이기 위해 삽입 정렬을 사용한다.

-Quick sort

Quick sort는 정렬알고리즘들 중에서 속도가 성능이 가장 좋은 것으로 알려져 있다. Quick sort의 동작은 아래 그림과 같이 이루어진다.



Pivot을 기준으로 데이터들의 swap이 일어나며, 함수를 재귀적으로 호출하여 정렬을 반복적으로 수행한다. 아래는 pivot의 값이 선택되어 위치하는 순서다.



4. Result Screen

```
LOAD mapdata.txt
PRINT
DFS 0 3
DIJKSTRA 0 3
DFS 1 4
BELLMANFORD 1 4
DIJKSTRA -1 10
BELLMANFORD
ASTAR 1 4
```

Command.txt

```
5
0 6 13 0 0
0 0 5 6 0
2 0 0 7 4
0 6 0 0 3
0 0 5 2 0
```

mapdata.txt

-Load

```
======LOAD======
Success
================
=================
Error code: 0
=================
```

해당 명령어를 통해 Error없이 Project에서 사용될 Linked list가 구현되었음을 Success를 확인하여 알 수 있습니다.

-PRINT

```
======PRINT======
0 6 13 0 0
0 0 5 6 0
2 0 0 7 4
0 6 0 0 3
0 0 5 2 0
=================
=================
Error code: 0
=================
```

Print명령을 통해 Error없이 현재 구현된 Linked list의 정보를 사용자에게 2차원 행렬로 보여줍니다.

-DFS

```
=========DFS=========
shortest path: 0 2 4 3
sorted nodes: 0 2 3 4
path length: 19
====================
=================
Error code: 0
=================
```

DFS알고리즘을 통하여 도출된 shortest path의 경로와 Vertex의 값에 따라 sorting된 값을 확인할 수 있으면 최종적으로 start경로에서 end경로까지의 거리가 있습니다.

-DIJKSTRA

```
=======DIJKSTRA=======
shortest path: 0 1 3
sorted nodes: 0 1 3
path length: 12
====================
=================
Error code: 0
=================
```

Start경로부터 End경로까지의 최단경로와 weight를 구해주는 다익스트라 알고리즘입니다. Error없이 실행된 것을 확인할 수 있습니다.

-DFS

```
=========DFS=========
shortest path: 1 3 4
sorted nodes: 1 3 4
path length: 9
====================
=================
Error code: 0
=================
```

위 DFS와 같은 설명입니다.

-BELLMANFORD

```
=====BELLMANFORD=====
shortest path: 1 2 4
sorted nodes: 1 2 4
path length: 9
====================
==================
Error code: 0
==================
```

다익스트라와 마찬가지로 최단 경로를 구하는 벨만포드 알고리즘입니다. 결과 같은 다익스트라와 동일하다고 생각하면 됩니다.

-DIJKSTRA (ERROR)

```
=======DIJKSTRA=======
InvalidVertexKey
====================
==================
Error code: 201
==================
```

Vertex의 잘못된 정보가 parameter로 들어간 경우의 ERROR입니다.

-BELLMANFORD(ERROR)

```
=====BELLMANFORD=====
VertexKeyNotExist
====================
==================
Error code: 200
==================
```

Parameter가 부족한 경우 출력되는 ERROR입니다.

-Not exist command

```
========ASTAR========
NonDefinedCommand
================
==================
Error code: 300
==================
```

해당 명령어가 존재하지 않는 경우 출력하는 ERROR입니다. 잘못 입력된 command를 출력하며 사용자에게 알려줍니다.

5. Consideration

마지막 프로젝트는 그래프와 관련된 개념과 정렬에 관련된 개념을 기반으로 각종 알고리즘을 구현하는 Project였습니다. Project의 난이도는 두번째 Project에 비해 높지 않은 편이었지만 각종 Project에 의해 시간이 많이 부족했습니다. 해당 Project를 진행하면서 의문이 있던 알고리즘은 DFS였습니다. 수업에서 배운 내용으로는 깊이 우선 탐색으로 하여 Vertex가 다른 Vertex로 이동할 수 없는 경우 돌아와 다른 경로를 탐색하는 것으로 배우게 되었는데 해당 Project에서 쓰인 DFS는 흡사 BFS와 비슷하였습니다. 다른 점은 BFS는 Queue를 사용하였지만 Stack을 이용하여 구현했다는 것이 가장 큰 특징이었습니다. 프로젝트에서 사용한 DFS는 Project제출 후 조금 더 생각하여 이해할 필요가 있다고 느껴졌습니다. 다음으로는 다익스트라 알고리즘과 벨만포드 알고리즘을 구현하는 것이었습니다. 해당 세개의 알고리즘은 구현하는데 크게 어려운 점이 없었습니다. 세개의 알고리즘 모두 비슷한 방법으로 구현하였으며, 특히 다익스트라는 set STL과 heap을 사용하는 두가지 방법으로 구현하였으며, 벨만포드 알고리즘 같은 경우 개념을 익혀 각 vertex는 최대 n-1개의 간선을 가질 수 있다는 점과 음수 cycle에 대한 예외처리가 특징입니다. 다음으로 구현하게 된 알고리즘은 Floy로 해당 알고리즘 또한 강의자료를 참고하여 쉽게 구현하였습니다. 벨만포드 알고리즘과 마찬가지로 음수 weight가 존재하여도 동작해야 하며, 음수 cycle이 있는 경우는 예외처리가 필요합니다. 또한 앞서 구현한 알고리즘과 다르게 2차원 matrix를 따로 구현하여 수식을 통한 함수 동작을 할 수 있도록 하였으며, Print되는 data 또한 2차원 matrix로 출력될 수 있도록 하였습니다. 음수 cycle예외 처리 같은 경우 해당 함수를 최종적으로 update된 matrix를 기반으로 돌려 값이 다시 새롭게 update되면 음수 cycle을 포함하고 있다는 것으로 간주합니다. 마지막으로 각 shortest path의 Vertex 정보를 기반으로 sorting함수를 구현하게 되었습니다. Sorting함수도 마찬가지로 제공된 data가 있기에 어렵지 않게 구현하였으며, sorting의 성능을 최대한 끌어올리기 위하여 path의 segment size를 판단하여 7을 기준으로 적으면 Insertion sorting 크면 Quick sorting을 작동할 수 있도록 구현하게 되었습니다. Sorting을 배우면서 아무리 Quick sort여도 data의 크기에 따라 다른 sort에 비해 속도가 느릴 수 있다는 것을 알게 되었고 Quick sort또한 처음 구현해보는 것이기 때문에 기존에 알던 Bubble sort나 선택 정렬, 삽입 정렬과는 다른 정렬에 익숙해질 수 있던 계기가 되었습니다. 다음 학습 목표로는 이번 자료구조에서는 제대로 구현하지 않은 다른 graph관련 알고리즘이나 sorting 파트에서 배운 merge sort를 구현해봐도 좋을 것 같습니다. 시험공부 병행하면서 프로젝트를 구현하였기에 부담이 컸지만 프로젝트를 이해하고 구현하는데 오히려 큰 도움이 될 수 있었습니다.