자료구조 Project Report

**2012210112** 유현우

**목차**

내용

[**1.** **개발 환경 및 개발 도구** 3](#_Toc454604536)

[**2.** **개발 산출물** 3](#_Toc454604537)

[**3.** **샘플 데이터 생성** 5](#_Toc454604538)

[**3.1.** **샘플 User 데이터 생성** 5](#_Toc454604539)

[**3.2.** **샘플 Friend 데이터 생성** 7](#_Toc454604540)

[**3.3.** **샘플 Tweet 데이터 생성** 9](#_Toc454604541)

[**4.** **기능 구현** 13](#_Toc454604542)

[**4.1.** **Read data files** 14](#_Toc454604543)

[**4.2.** **Statistics** 16](#_Toc454604544)

[**4.3.** **Top 5 most tweeted words** 18](#_Toc454604545)

[**4.4.** **Top 5 most tweeted users** 20](#_Toc454604546)

[**4.5.** **Find all users who mentioned a word** 21](#_Toc454604547)

[**4.6.** **Find all users who are friend of the above user** 25](#_Toc454604548)

[**4.7.** **Delete all users who mentioned a word** 27](#_Toc454604549)

[**4.8.** **Top 5 strongly connected components** 29](#_Toc454604550)

[**4.8.1.** **Graph 구현** 29](#_Toc454604551)

[**4.8.2.** **Koraraju의 알고리즘** 30](#_Toc454604552)

[**4.8.3.** **Strongly Connected Component 추출 기능 구현** 36](#_Toc454604553)

[**4.9.** **Find shortest path from a user (id)** 40](#_Toc454604554)

[**4.9.1.** **Graph 구현** 40](#_Toc454604555)

[**4.9.2.** **Dijkstra 알고리즘** 41](#_Toc454604556)

[**4.9.3.** **Shortest Path 기능 구현** 44](#_Toc454604557)

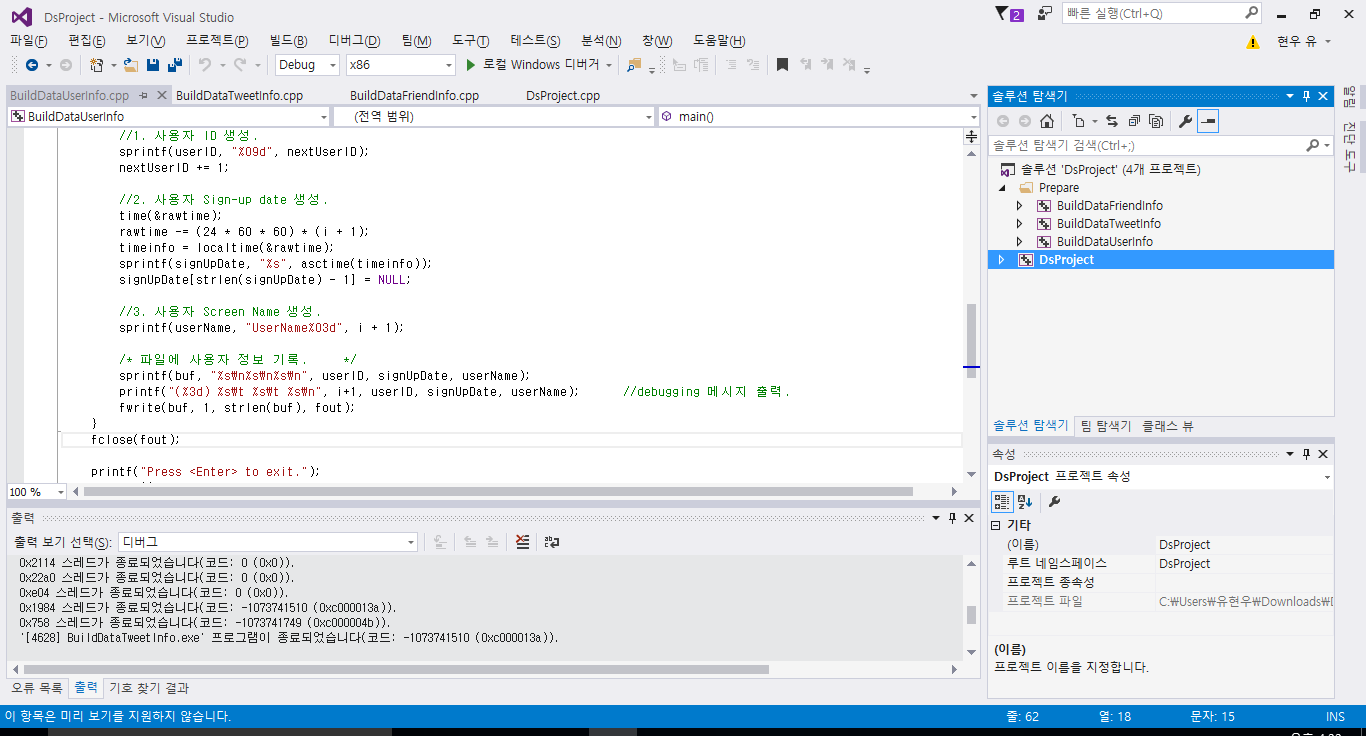
[**5.** **개선이 필요한 내용** 46](#_Toc454604558)

# **개발 환경 및 개발 도구**

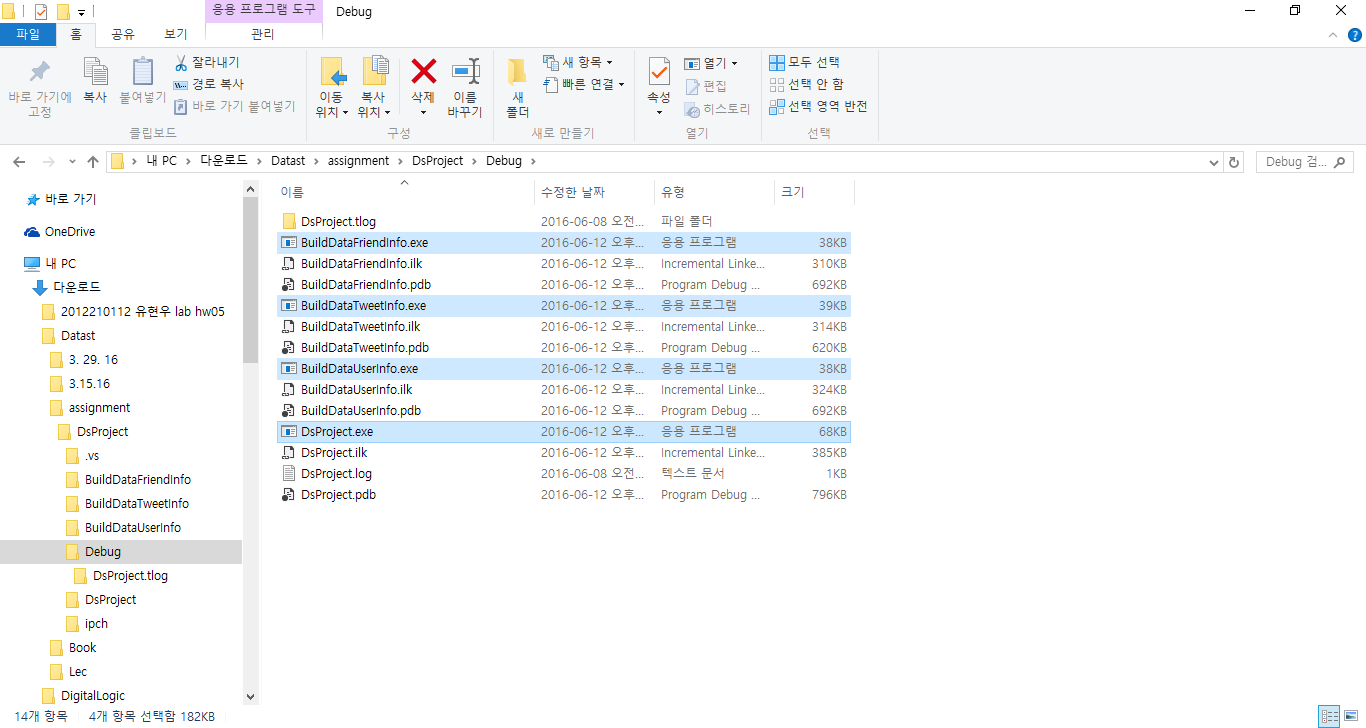
* Microsoft Visual Studio Community 2015를 사용하여 작성하였음.
* 개발언어는 C를 기본으로 하고, 일부 영역에서는 Visual C++의 기능을 이용하였음.

# **개발 산출물**

* 1. 1개의 Solution으로 구성되어 있으며, Solution 이름은 **DsProject.sin** 이다.



* 1. Solution DsProject는 다음과 같은 4개의 Project로 구성되어 있다.
* **BuildDataUserInfo.vcxproj**: Project에서 사용하는 샘플 User 정보를 생성하는 프로그램.
* **BuildDataFriendInfo.vcxproj**: Project에서 사용하는 User들의 샘플 Friend 정보를 생성하는 프로그램.
* **BuildDataTweetInfo.vcxproj**: Project에서 사용하는 User들의 샘플 Tweet 정보를 생성하는 프로그램.
* **DsProject.vcxproj**: 위의 3 프로그램을 이용해 생성한 샘플 데이터를 이용해, Project 내용을 수행하는 프로그램.
  1. Solution을 모두 Build에 성공하면 Debug 폴더에 “**BuildDataUserInfo.exe**”, “**BuildDataFriendInfo.exe**”, “**BuildDataTweetInfo.exe**”, 그리고 “**DsProject.exe**”라는 4개의 실행 프로그램 파일이 생성된다.



# **샘플 데이터 생성**

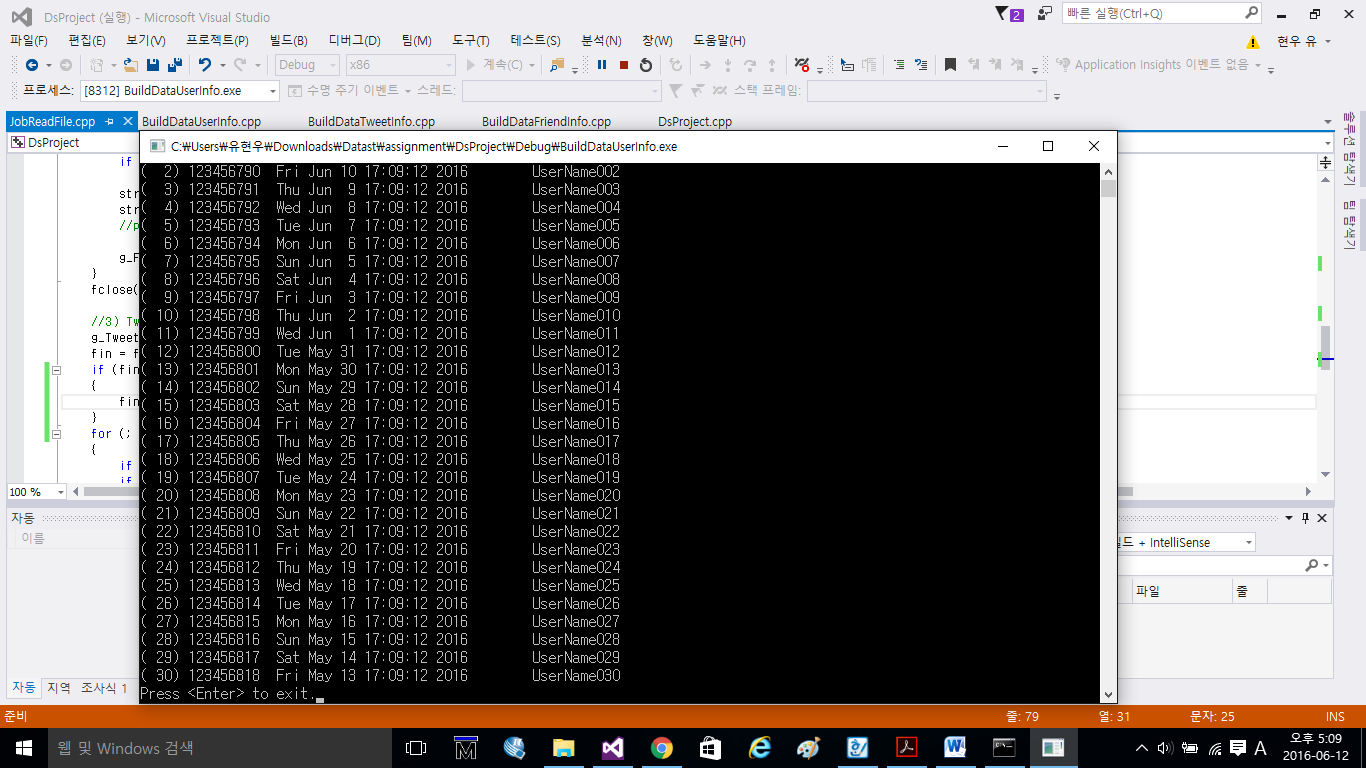
다양한 데이터에 대해 테스트를 수행하기 위해, 필요에 따라 손 쉽게 샘플 데이터를 구성할 필요가 있다. 따라서 효율적으로 샘플 데이터를 생성할 수 있는 프로그램을 제작해서 사용하게 되었다. 이를 통해, 다양하게 데이터를 바꿔가면서 테스트를 수행할 수 있게 되었다.

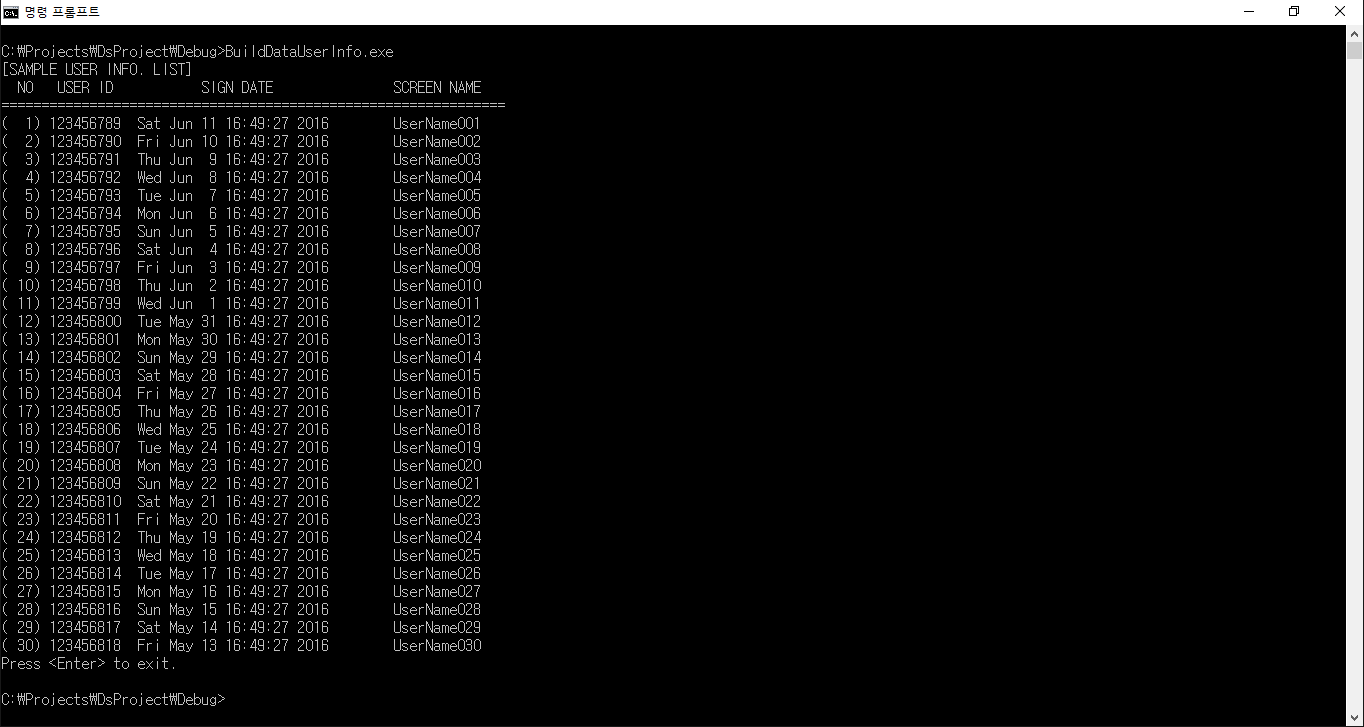
샘플 데이터는 각각의 전용 프로그램 3개를 이용해서 생성하는데, 프로그램에서 생성하는 샘플 데이터는 각각 다음과 같은 파일에 저장된다. 따라서 Project에서 이 파일을 읽어와 이용할 수 있다.

* 파일 **UserInfo.txt**: 샘플 User 목록.
* 파일 **FriendInfo.txt**: 샘플 Friend 목록.
* 파일 **TweetInfo.txt**: 샘플 Tweet 목록.

# **샘플 User 데이터 생성**

Visual Studio에서 Project BuildDataUserInfo를 시작 프로젝트로 설정하고 프로그램을 실행하거나, Command 창에서 BuildDataUserInfo.exe를 실행해서 샘플 User 데이터를 생성한다.





프로그램을 다음과 같은 정보를 이용해 30명(MAX\_USER\_COUNT)의 샘플 User 정보를 생성한다. 사용자의 User ID는 123456789(USER\_ID\_START)부터 1씩 증가하는 값을 사용한다. 따라서 User 1의 ID는 123456789, User 2의 ID는 123456790, User 30의 ID는 123456818이 된다. 그리고 각 User의 Screen Name은 특별한 의미를 갖지 않기 때문에 단순히 “UserName###”이라는 형식을 사용했다.

#define MAX\_USER\_COUNT 30 //사용자 개수.

#define USER\_ID\_START 123456789 //첫번째 사용자 ID.

//3. 사용자 Screen Name 생성.

sprintf(userName, "UserName%03d", i + 1);

User 정보는 문자열 형태의 User ID, Sign Date, Screen Name으로 구성되기 때문에, 다음과 같은 파일 형식을 사용하고 있다.

123456789

Tue Jun 7 11:12:42 2016

UserName001

123456790

Mon Jun 6 11:12:42 2016

UserName002

123456791

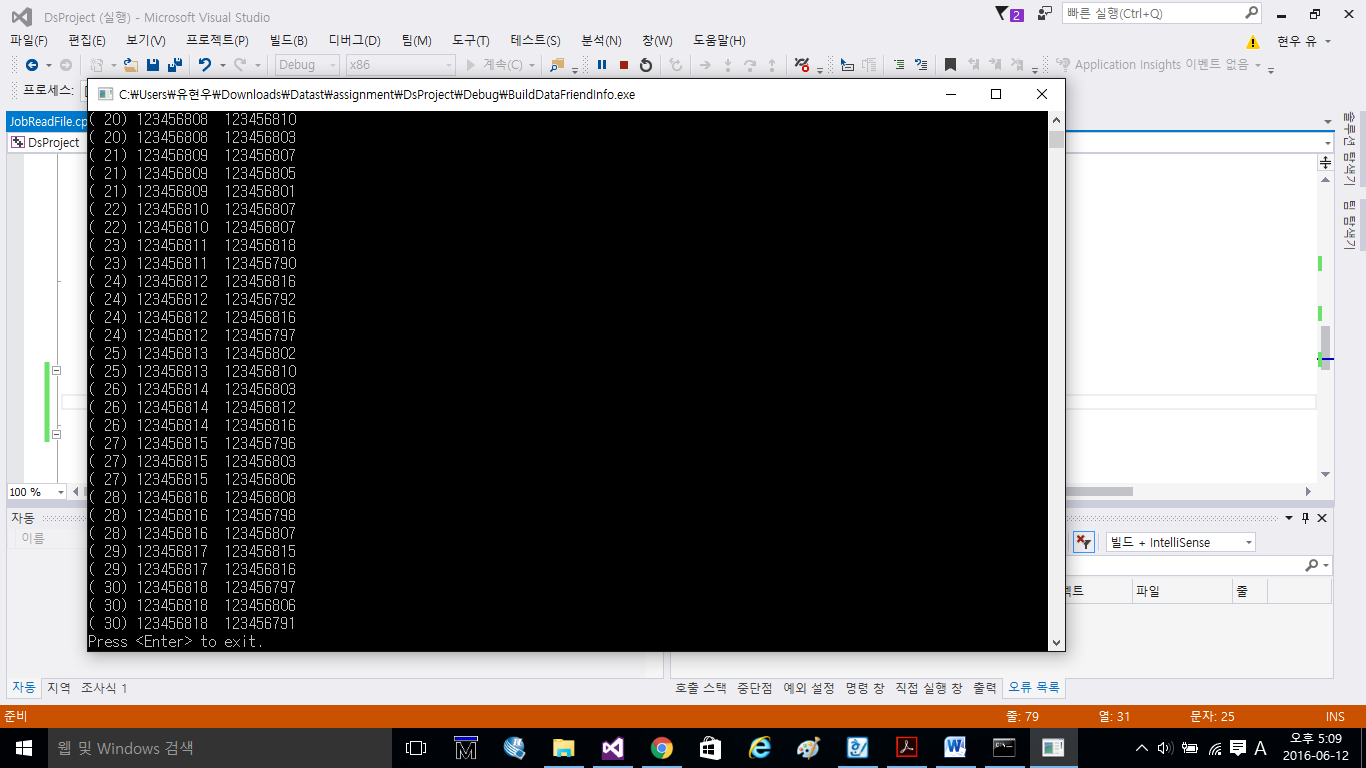
Sun Jun 5 11:12:42 2016

UserName003

…

# **샘플 Friend 데이터 생성**

Visual Studio에서 Project BuildDataFriendInfo를 시작 프로젝트로 설정하고 프로그램을 실행하거나, Command 창에서 BuildDataFriendInfo.exe를 실행해서 샘플 Friend 데이터를 생성한다.





프로그램을 Random 함수를 사용해 샘플 Friend 정보를 생성한다. 먼저 Random 함수를 사용해 각각의 User에 대한 Friend 수를 2명에서 4명 사이로 정의한다. **Friend 수는 이후 Strongly Connected Component를 구하는 것과 관련이 깊은데, Friend 수를 너무 많게 하면 Graph의 모든 Node들이 1개의 Strongly Connected Component에 모두 포함되는 문제가 있고, Friend 수를 너무 적게 하면 Graph에서 Strongly Connected Component에 속하는 Node가 너무 적게 되는 문제점이 있기 때문에, 반복 테스트를 통해 2~4명 사이가 가장 적당하다고 판단되어 사용하게 되었다**.

다음에는 역시 Random 함수를 이용해, 전체 User 목록에서 자신에게 배정된 개수만큼 Friend가 되는 User ID를 지정하도록 구현했다.

//2. i번째 사용자의 Friend 개수 생성(Random 함수를 사용해 2~4명 사이를 결정한다.

countFriends = 2 + (rand() % 3);

//3. i번째 사용자의 Friend 목록 구성.

for (j = 0; j < countFriends; j++)

{

//i번째 사용자의 Friend ID 생성.

sprintf(toUserID, "%09d", nextFriendID);

}

Friend 정보는 문자열 형태의 User ID 순서쌍(from user, to user)으로 구성되기 때문에, 다음과 같은 파일 형식을 사용하고 있다.

123456789 123456807

123456789 123456816

123456789 123456797

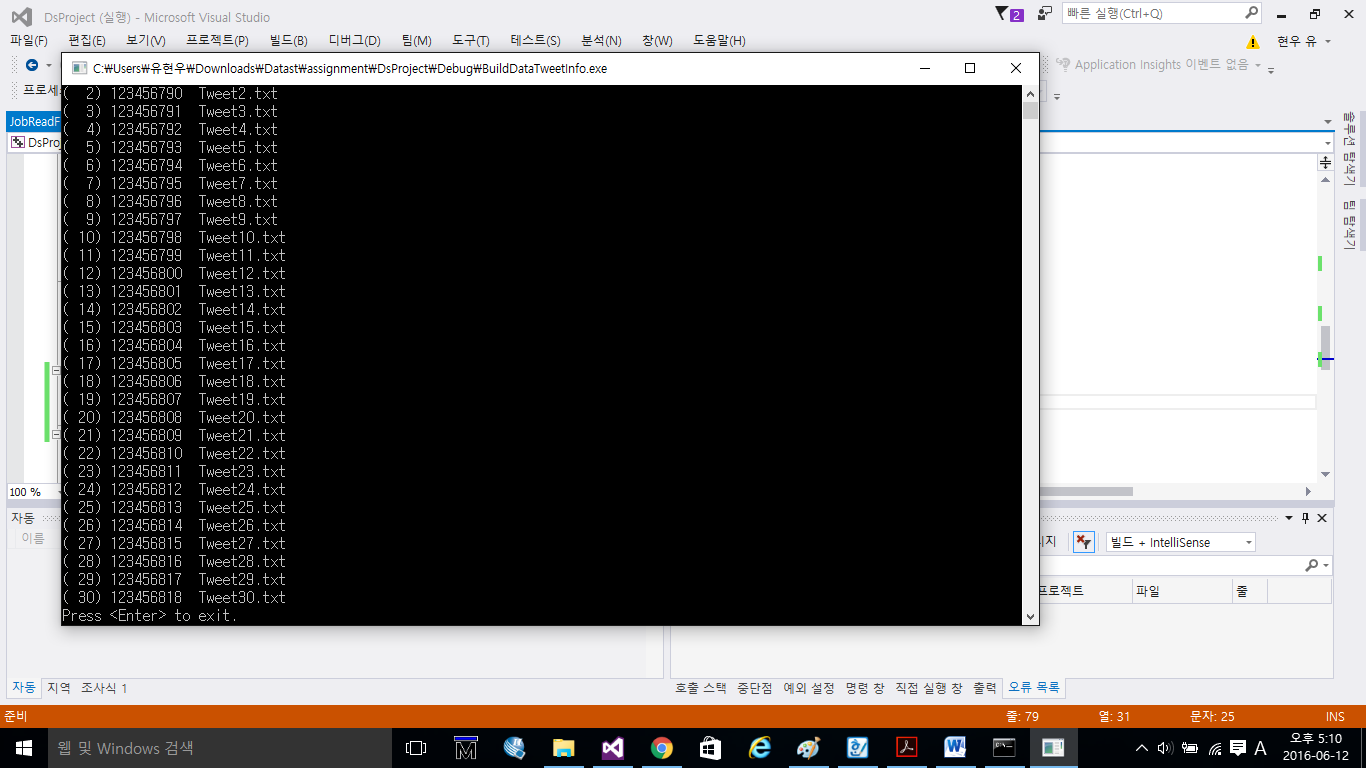
123456789 123456812

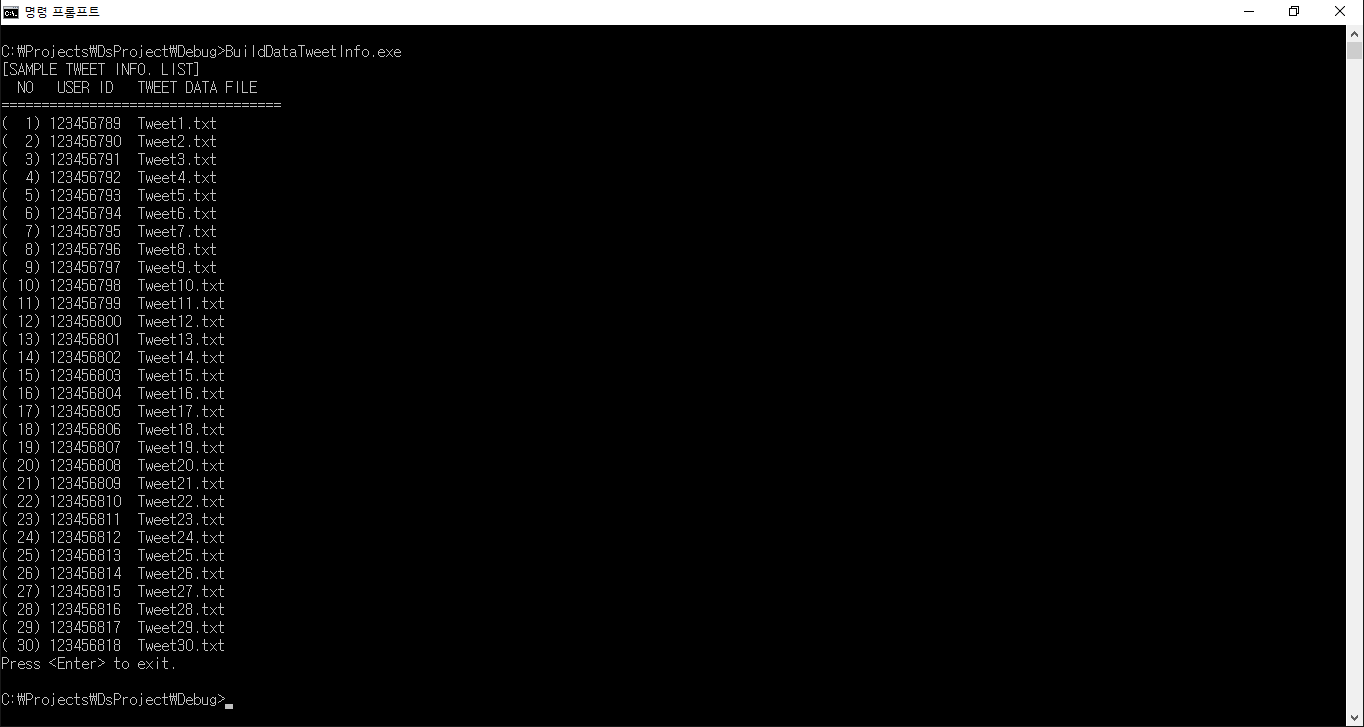
123456790 123456817

…

# **샘플 Tweet 데이터 생성**

Visual Studio에서 Project BuildDataTweetInfo를 시작 프로젝트로 설정하고 프로그램을 실행하거나, Command 창에서 BuildDataTweetInfo.exe를 실행해서 샘플 Tweet 데이터를 생성한다.





각 User가 Tweet 하는 정보에 대한 샘플을 구성하는 것도 쉬운 문제는 아니었다. **데이터가 적으면 충분한 테스트를 할 수가 없고, 데이터를 많이 구성하려면 너무나 많은 시간과 노력이 필요하기 때문이다**. 그래서 각 User별로 1개의 텍스트 파일을 지정하고, 그곳에 해당 사용자가 Tweet 했다고 가정하는 문장을 저장하는 방법을 채택했다. 프로그램에서 파일을 내용을 읽어, 1개씩 Word로 분리한 다음, 해당 Word를 Tweet 한 것으로 샘플 데이터를 구성했다. **이렇게 함으로써 대량의 Tweet 정보도 비교적 손 쉽게 샘플로 구축하고, 단지 User용 텍스트 파일의 내용을 변경해서 프로그램을 실행함으로써 필요에 따라 샘플을 쉽게 변경할 수 있게 되었다**.

TG앤컴퍼니(대표 이홍선)가 루나폰 OS `마시멜로` 업그레이드를 진행한다고 밝혔다.

루나폰

<루나폰>

업그레이드에는 미국 보안업체 짐페리엄이 개발한 스마트폰 보안 솔루션 `zIPS`를 탑재해 보안 성능을 강화했다. zIPS는 내장된 학습 시스템을 통해 스마트폰이 정상 작동하는 상태를 파악한 후 변화가 발생할 때마다 사용자에게 알림을 제공하는 솔루션이다.

호스트와 네트워크 기반 공격을 실시간으로 탐지해 알려지지 않은 위협에서도 기기를 보호하는 것이 특징이다.

TG앤컴퍼니 관계자는 “업그레이드 핵심은 보안 기능”이라며 “루나폰은 행동 분석형 침입 방지 시스템을 제공해 위협 요소를 사전에 차단할 수 있게 됐다”고 설명했다.

이번 업그레이드를 통해 앱이 스마트폰 특정 정보에 접근할 수 있는 허락 범위를 사용자가 지정하는 `앱 권한 설정 기능`, 사용자 움직임을 감지해 장기간 스마트폰을 사용하지 않으면 비활성화시켜 배터리 시간을 늘려주는 `도즈(Doze)` 기능, 메모리 관리 기능 등 사용자의 편의를 위한 기능을 만나볼 수 있다.

함지현기자 goham@etnews.com

Tweet 정보는 문자열 형태의 User ID, Register Date, Tweet Word로 구성되기 때문에, 다음과 같은 파일 형식을 사용하고 있다.

123456789

Sat May 28 15:18:41 2016

TG앤컴퍼니(대표

123456789

Wed Jun 1 23:18:41 2016

이홍선)가

123456789

Sat Jun 4 15:18:41 2016

루나폰

123456789

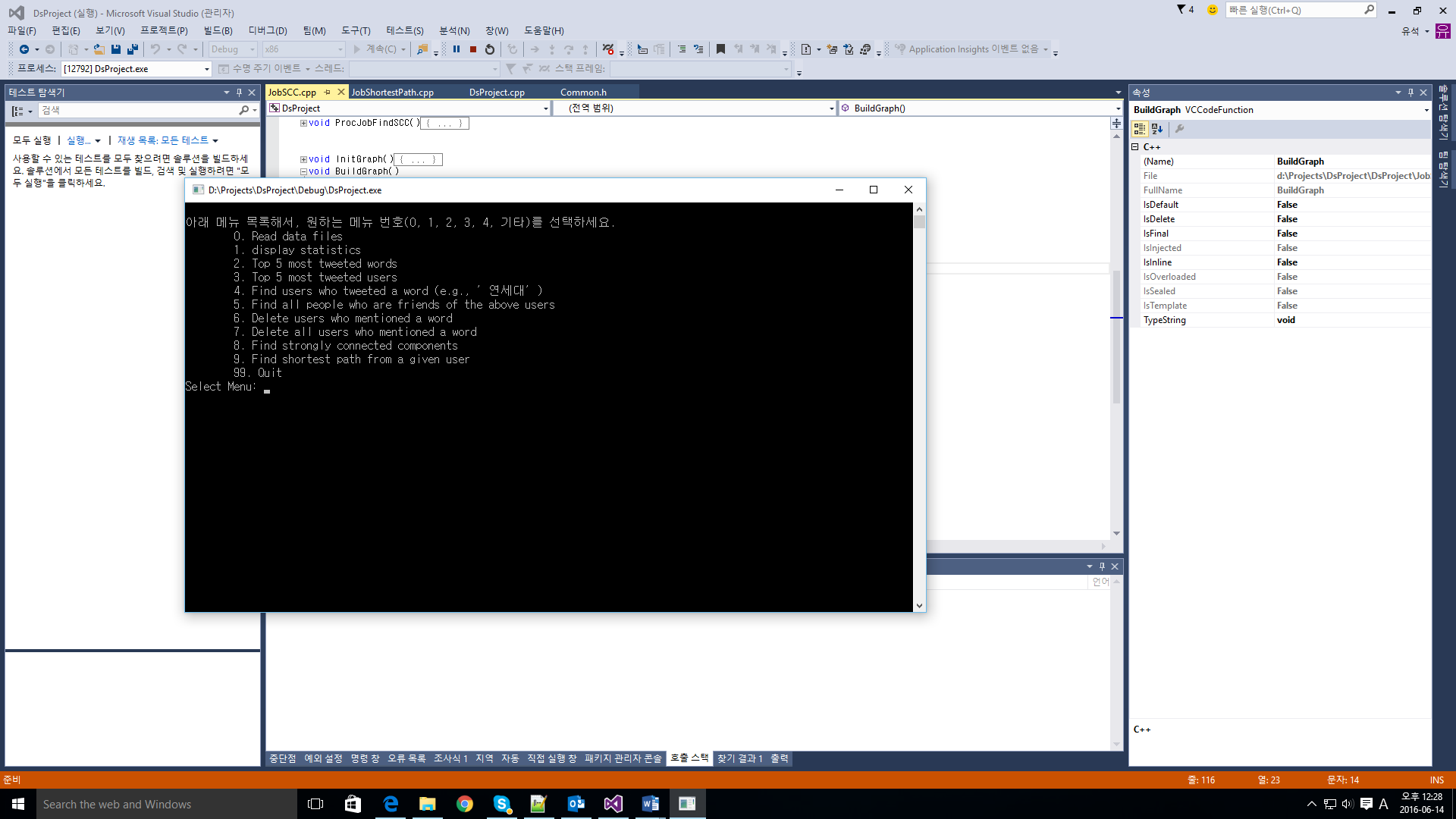
Wed Jun 8 09:18:41 2016

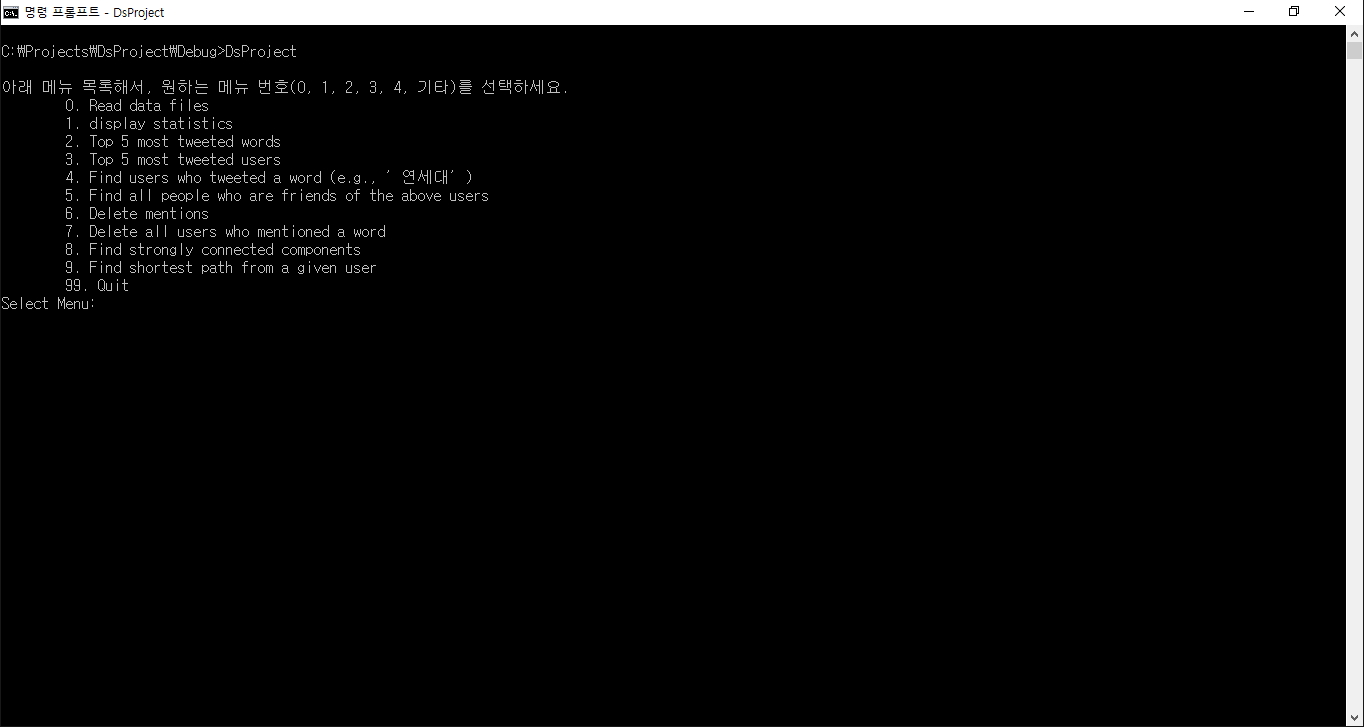
OS

…

# **기능 구현**

Visual Studio에서 Project **DsProject**를 시작 프로젝트로 설정하고 프로그램을 실행하거나, 아래 화면처럼 Command 창에서 명령어 **DsProject**를 입력해서 프로젝트 프로그램을 실행한다.





# **Read data files**

메뉴에서 0을 선택하면, 앞 단계에서 생성한 3개의 샘플 데이터 파일(User, Friend, Tweet 데이터 파일)의 내용을 읽어와서, 프로그램에서 사용하는 자료구조를 구성한다. 자료구조는 C 구조체(struct)에 대한 배열을 사용했다.

//User 정보

//1. 전체 User Info 목록 - 0. Read data files의 작업결과.

int g\_UserCount;

struct userInfo\_t

{

char userID[MAX\_USER\_ID\_LEN];

char signUpDate[MAX\_DATE\_TIME\_LEN];

char userName[MAX\_USER\_NAME\_LEN];

};

struct userInfo\_t g\_UserInfoList[];

//Friend 정보

//1. 전체 Friend Info 목록 - 0. Read data files의 작업결과.

int g\_FriendCount;

struct friendInfo\_t

{

char fromUserID[MAX\_USER\_ID\_LEN];

char toUserID[MAX\_USER\_ID\_LEN];

};

struct friendInfo\_t g\_FriendInfoList[];

//Tweet 정보

//1. 전체 Tweet Info 목록 - 0. Read data files의 작업결과.

int g\_TweetCount;

struct tweetInfo\_t

{

char fromUserID[MAX\_USER\_ID\_LEN];

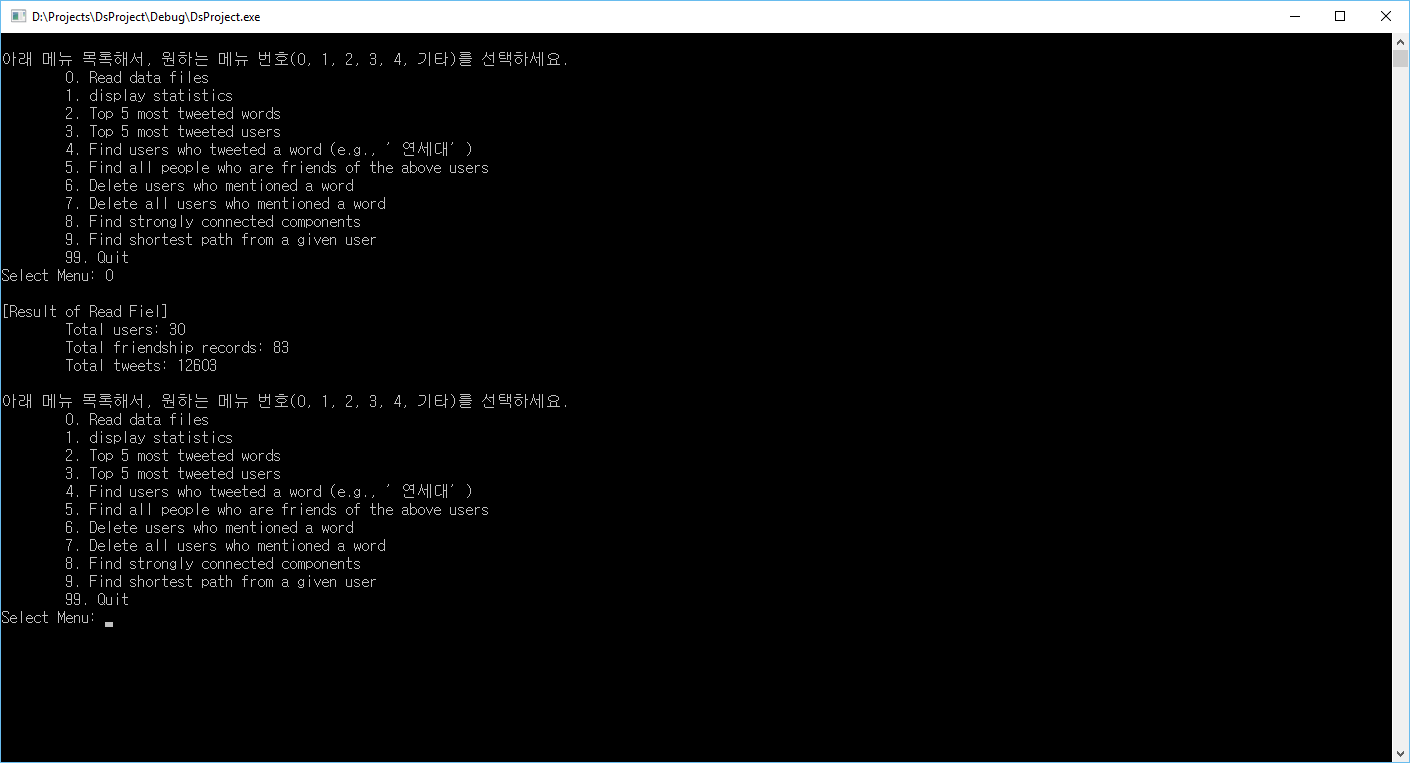
char signUpDate[MAX\_DATE\_TIME\_LEN];

char tweetWord[MAX\_WORD\_LEN];

};

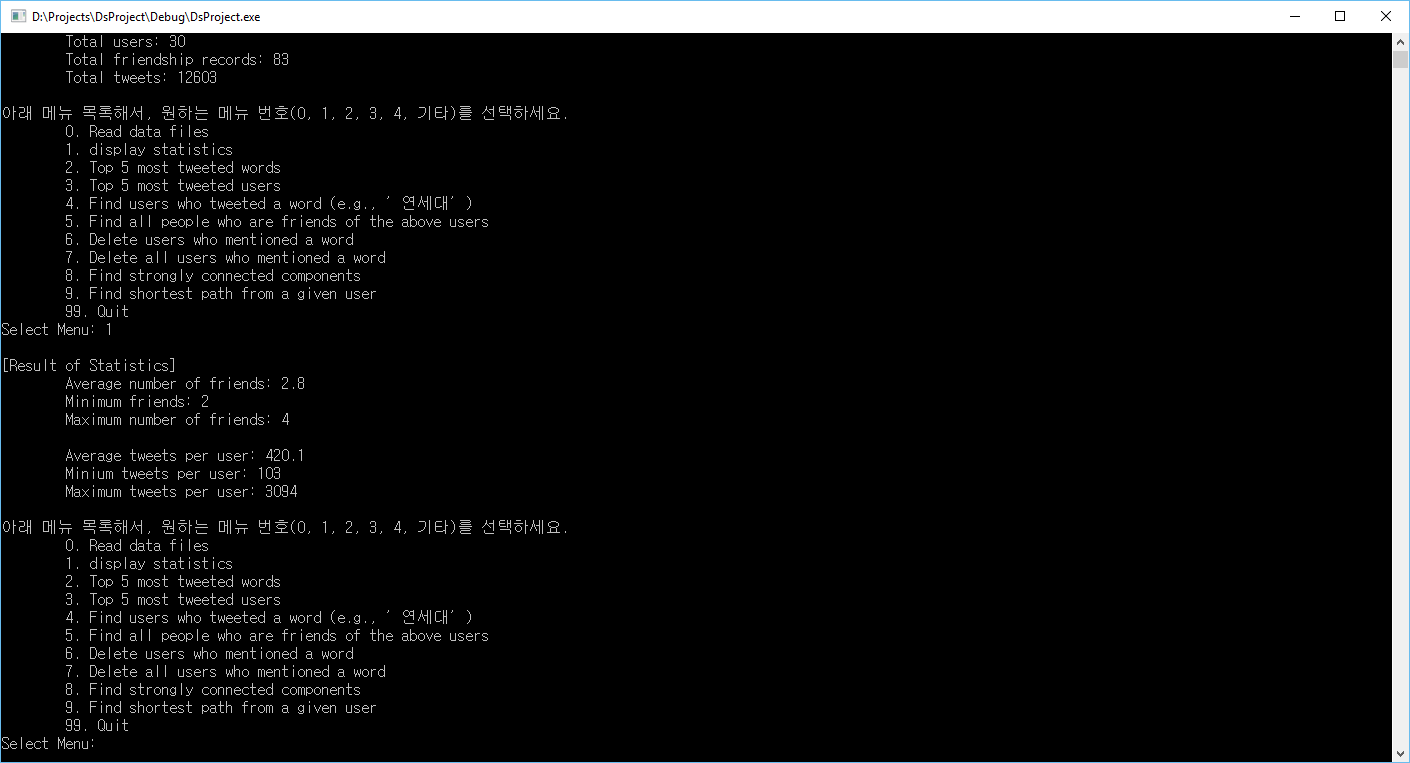
struct tweetInfo\_t g\_TweetInfoList[];

작업을 성공적으로 완료하면, 아래 화면처럼 샘플 데이터 파일에서 읽어 들인 항목의 개수를 출력하고 있다. 예제 화면에서는 30명의 User에 대해 83개의 Friend 항목과 12,603개의 Tweet 항목으로 파일에서 읽어왔다고 표시하고 있다. 이 정도면 테스트에 충분한 데이터를 확보 했다고 할 수 있다.



# **Statistics**

메뉴 1번에 대한 작업은 메뉴 0번 작업에서 파일에서 읽어 들인 Friend, Tweet 파일 내용을 분석해서, 통계자료를 추출해 그 결과를 화면에 출력한다.



작업을 성공적으로 완료하면, 위의 화면처럼 샘플 데이터 파일에서 읽어 들인 항목의 개수를 분석한 결과를 출력하고 있다. 예제 화면에서는 30명의 User들이 평균 2.8명의 Friend를 등록하고 있으면, 최소 2명에서 최대 4명의 Friend를 등록하고 있다고 표시하고 있다. 그리고 Tweet 정보에 대해서는 30명의 User들이 평균 420.1개의 Word들을 Tweet하고 있으면, 최소 103개에서 최대 3,094개의 Word들을 Tweet하고 있다고 표시하고 있다.

작업 처리는 파일에서 읽어 들였던 데이터에 대해 User별 Friend 개수 목록과 Tweet 개수 목록을 구축하고, 단순히 Linear Search를 사용해 O(n)의 Time Complexity로 평균, 최소, 최대 값을 구하는 작업을 처리했다.

//2. User별 Friend Info 목록 - 1. display statistics의 작업결과.

struct userFriendInfo\_t

{

char fromUserID[MAX\_USER\_ID\_LEN];

int count;

};

struct userFriendInfo\_t \*g\_UserFriendInfoList;

//1) Friends 정보 분석하기.

//1-1) User별 Friend 목록 구성.

s\_userFriendCount = 0;

if (g\_UserFriendInfoList != NULL)

free(g\_UserFriendInfoList);

g\_UserFriendInfoList = (struct userFriendInfo\_t\*) malloc(sizeof(struct userFriendInfo\_t) \* g\_UserCount);

for (i = 0; i < g\_FriendCount; i++)

{

statisticisFriend(&g\_FriendInfoList[i]);

}

//1-2) User별 Friend statistics 정보 구성.

s\_FriendAve = 0.0;

s\_FriendMin = s\_FriendMax = 0;

sum = 0;

for (i = 0; i < s\_userFriendCount; i++)

{

sum += g\_UserFriendInfoList[i].count;

if (i == 0)

{

s\_FriendMin = s\_FriendMax = g\_UserFriendInfoList[i].count;

}

else

{

if (s\_FriendMin > g\_UserFriendInfoList[i].count)

s\_FriendMin = g\_UserFriendInfoList[i].count;

if (s\_FriendMax < g\_UserFriendInfoList[i].count)

s\_FriendMax = g\_UserFriendInfoList[i].count;

}

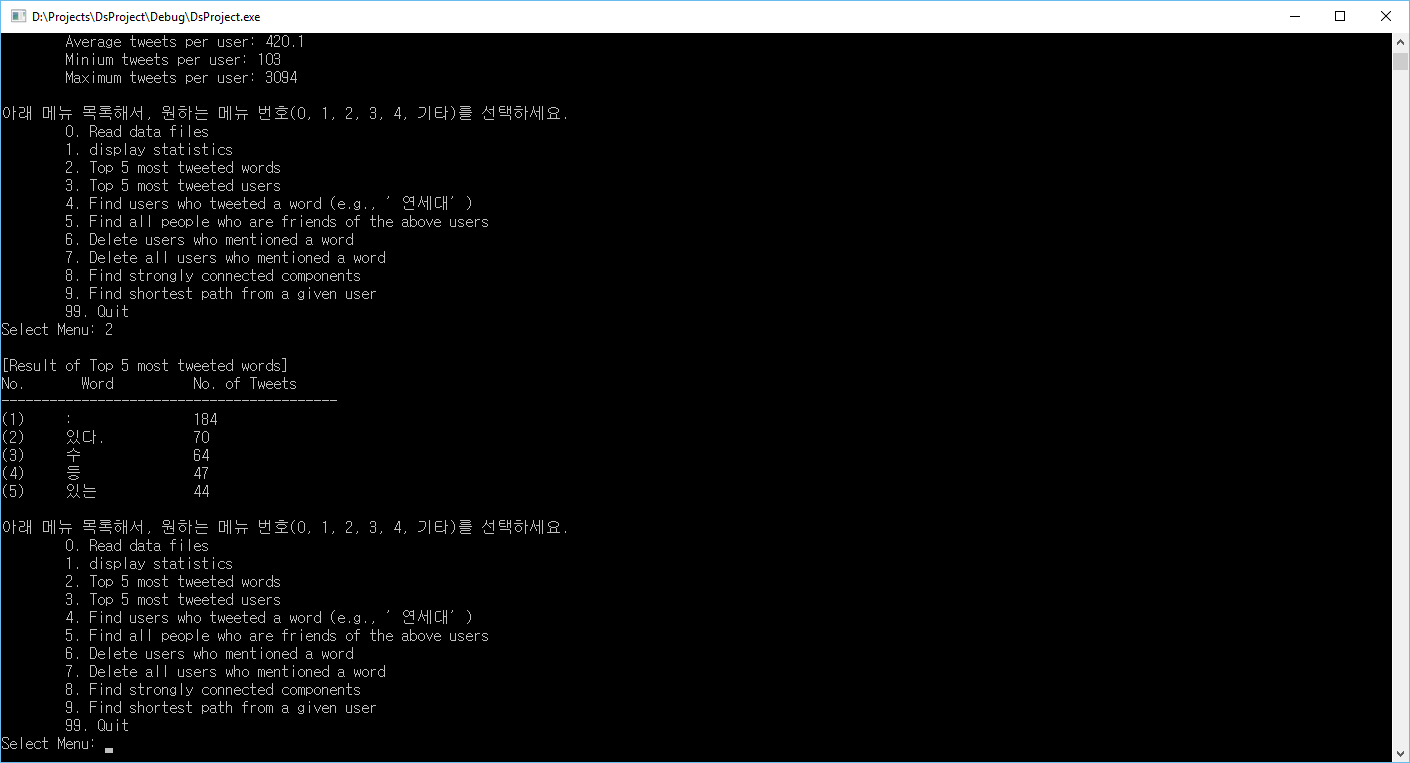
}

s\_FriendAve = sum / s\_userFriendCount;

# **Top 5 most tweeted words**

Tweet 데이터에 대해 Linear Search를 적용해서, 각 Word별 발생 빈도에 대한 정보를 구성하고, Quick Sort를 이용해 발생 빈도를 기준으로 Tweet Word들을 정렬해서, 가장 발생 빈도가 높은 최상위 5 Word를 판별하고 있다. 따라서 각각 O(n)과 O(n log n)의 Complexity로 문제를 해결했다.

결과 화면에서는 최상위 순으로 가장 많은 발생 빈도를 나타내는 Word와 발생 빈도수를 출력한다. 아래 예제 화면은 가장 발생빈도가 높은 5 Word(:, 있다., 수, 등, 있는)에 대한 발생 빈도(184, 70, 64, 47, 44)를 출력하고 있다.



기능을 구현하기 위해, Tweet 정보 목록에서 Word별 발생빈도를 추출해야 한다. Tweet 목록을 scanning 하는 작업은 O(n), Word별 발생빈도 데이터 목록을 구성하는 작업은 O(n2), 발생 빈도수에 따라 내림차순으로 정렬하는 작업은 O(n log n)의 Complexity를 가지는 알고리즘으로 구현했다.

//3. Word별 Tweet Info 목록 - 3. Top 5 most tweeted users의 작업결과.

struct wordTweetInfo\_t

{

char tweetWord[MAX\_WORD\_LEN];

int count;

};

struct wordTweetInfo\_t \*g\_WordTweetInfoList;

//1. User별 Friend 목록 구성.

s\_WordCount = 0;

if (g\_WordTweetInfoList != NULL)

free(g\_WordTweetInfoList);

g\_WordTweetInfoList = (struct wordTweetInfo\_t \*) malloc(sizeof(struct wordTweetInfo\_t) \* g\_TweetCount);

for (i = 0; i < g\_TweetCount; i++)

{

statisticisWord(&g\_TweetInfoList[i]);

}

//User별 Tweet Info 목록을 내림차순으로 정렬.

quick\_sort(g\_WordTweetInfoList, 0, s\_WordCount - 1);

void statisticisWord(struct tweetInfo\_t \*tweet)

{

int i;

//User별 Friend 항목이 존재하면, count를 증가시킨다.

for (i = 0; i < s\_WordCount; i++)

{

if (strcmp(g\_WordTweetInfoList[i].tweetWord, tweet->tweetWord) == 0)

{

g\_WordTweetInfoList[i].count++;

return;

}

}

//User별 Friend 항목이 존재하지 않는 경우, 추가한다.

strcpy(g\_WordTweetInfoList[s\_WordCount].tweetWord, tweet->tweetWord);

g\_WordTweetInfoList[s\_WordCount].count = 1;

s\_WordCount++;

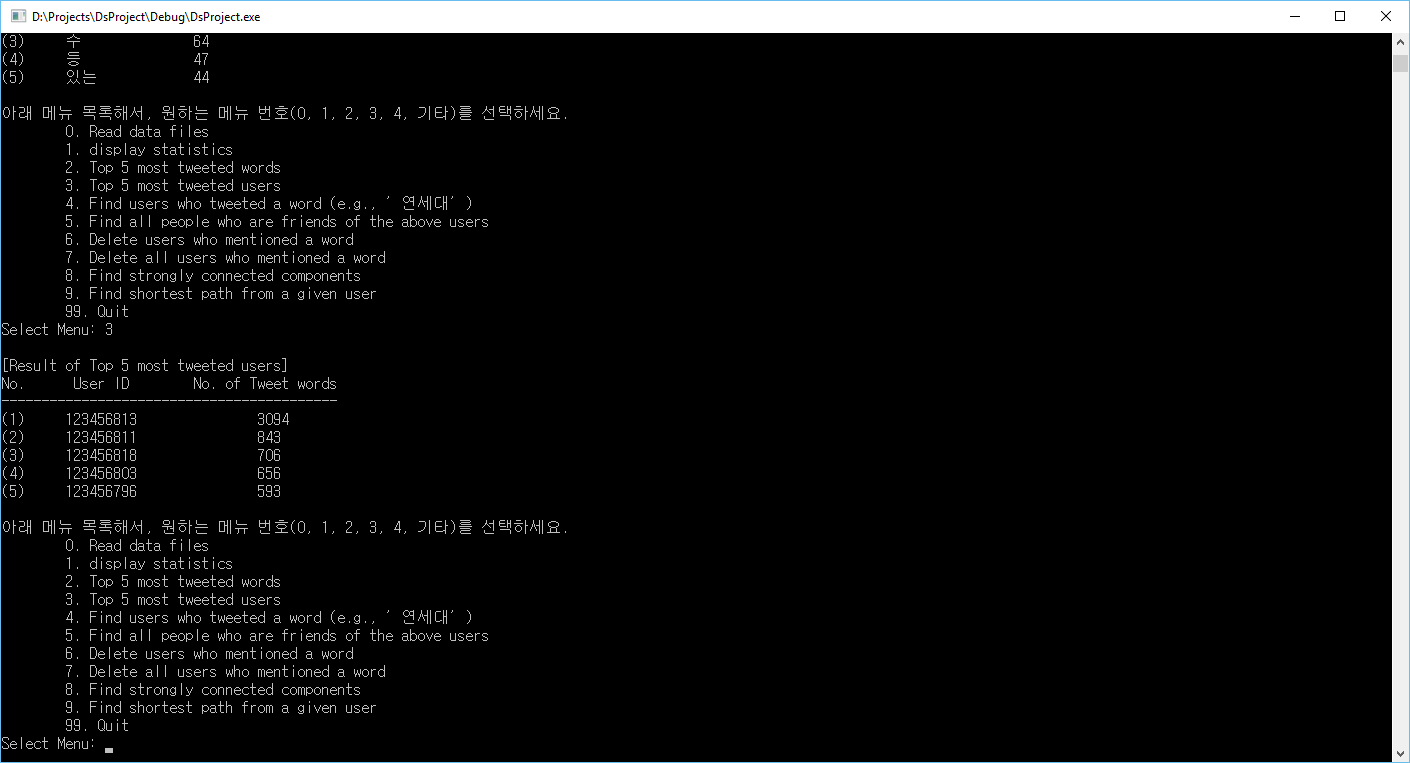
}

# **Top 5 most tweeted users**

메뉴 3번의 작업은 Tweet 데이터에 대해 Linear Search를 적용해서, 각 User별 Tweet Word 개수에 대한 정보를 구성하고, Quick Sort를 이용해 Tweet Word 개수를 기준으로 User별 Tweet 정보들을 정렬해서, 가장 많은 Word를 Tweet한 최상위 5 User를 판별하고 있다. 따라서 각각 O(n)과 O(n log n)의 Time Complexity로 문제를 해결했다.

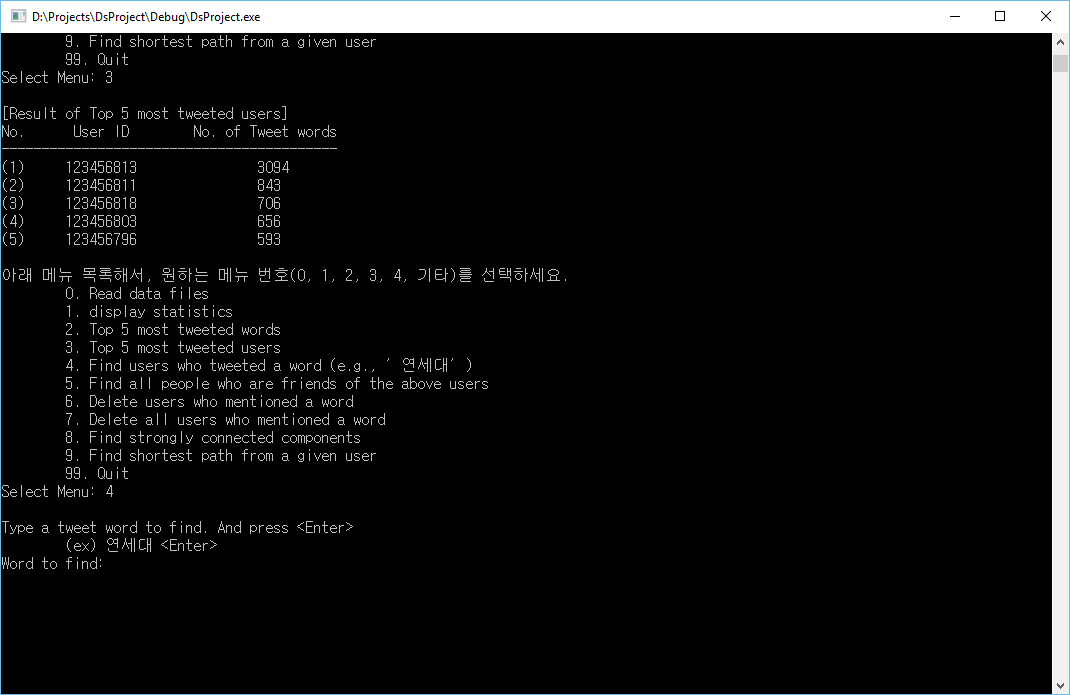
각 User별 Tweet Word 개수에 대한 정보를 구성하는 작업은 메뉴 1번에서 수행하고 있기 때문에, 메뉴 3번 작업은 메뉴 1번 작업결과를 이용하고 있다. 따라서 메뉴 3번을 실행하려면, 먼저 메뉴 1번이 실행되어야 한다. 만일 메뉴 1번이 실행되지 않은 상태에서 메뉴 3번을 실행하려 하면, 메뉴 1번을 먼저 실행하라고 오류 메시지를 출력한다.

결과 화면에서는 가장 많은 Word를 Tweet한 User 순서로, User의 User ID와 Tweet한 Word 개수를 출력하고 있다. 아래 예제 화면은 User 123456813이 3,094개의 Word를 Tweet 해서 가장 많은 Tweet를 한 User에 해당하고, 그 뒤를 이어 123456811, 123456818, 123456803, 123456796 순으로 많은 Word를 Tweet 했다고 알려주고 있다.

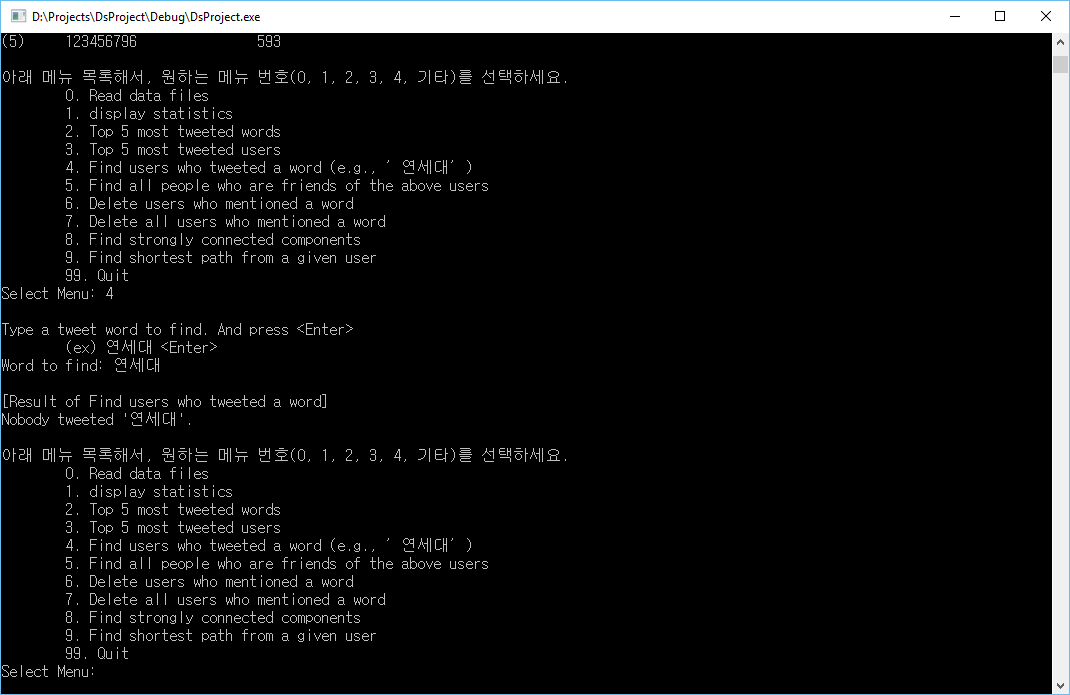


# **Find all users who mentioned a word**

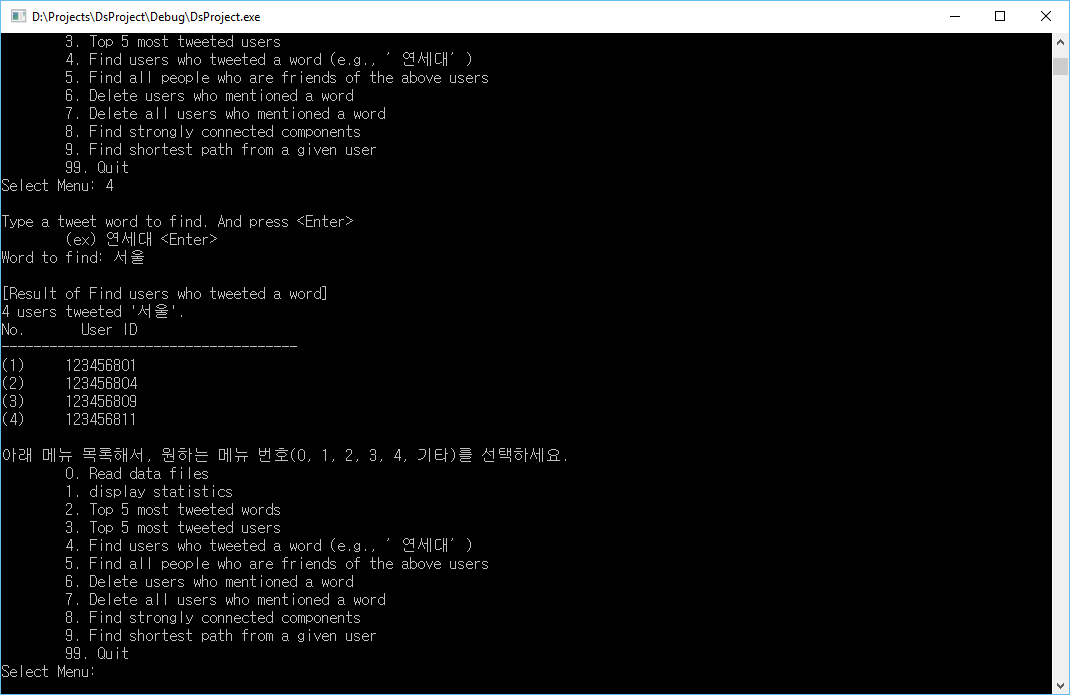
메뉴 4번은 사용자로부터 1개의 Keyword를 입력 받아, 사용자가 입력한 Keyword를 Tweet한 User 목록을 검색한다. 따라서 사용자가 메뉴 4번을 선택하면, Tweet 목록에서 검색하려는 Keyword를 입력 받는 메뉴를 출력한다.



사용자가 Keyword를 입력하면, Keyword를 Tweet 했던 User 목록을 검색한다. 만일 사용자가 입력한 Keyword를 Tweet한 User를 발견하지 못한 경우에는 메시지를 출력해서 이 사실을 알려주고 있다. 아래 결과 화면은 “연세대”라는 Word를 Tweet한 User들을 발견할 수 없다고 알려주고 있다.



그리고 사용자가 입력한 Keyword를 Tweet한 User를 발견한 경우에는, 화면에 해당 User들의 User ID 목록을 출력하고 있다. 아래 결과 화면은 “서울”이라는 Word를 Tweet한 4명의 User 목록을 보여주고 있다. 아래 결과화면에는 4명의 User “123456801”, “123456804”, “123456809”, “123456811”이 “서울”이라는 Word를 Tweet 했다고 알려주고 있다.



Linear Search를 이용해 Tweet 데이터 목록을 검사해서 사용자가 입력한 Keyword를 Tweet한 User들을 검색했다. 따라서 O(n) 의 Complexity로 문제를 해결했다.

//Tweet 사용자 정보 - 4. Find users who tweeted a word의 작업결과.

int s\_TweetUserCount = 0;

char s\_TweetUserList[MAX\_USER\_COUNT][MAX\_USER\_ID\_LEN];

//Tweet 목록에서, 주어진 Word를 포함하는 User 목록을 추출한다.

s\_TweetUserCount = 0;

for (i = 0; i < g\_TweetCount; i++)

{

if (strcmp(keyword, g\_TweetInfoList[i].tweetWord) == 0)

{

InsertUserId(g\_TweetInfoList[i].fromUserID);

}

}

//Keyword를 Tweet한 User 목록 구성.

void InsertUserId(char \*fromUserID)

{

int i;

//이미 사용자 목록에 추가되어 있는지를 검사한다.

for (i = 0; i < s\_TweetUserCount; i++)

{

if (strcmp(s\_TweetUserList[i], fromUserID) == 0)

{

//이미 User ID가 등록되어 있기 때문에 무시한다.

return;

}

}

//사용자 목록에 User ID를 추가한다.

strcpy(s\_TweetUserList[s\_TweetUserCount], fromUserID);

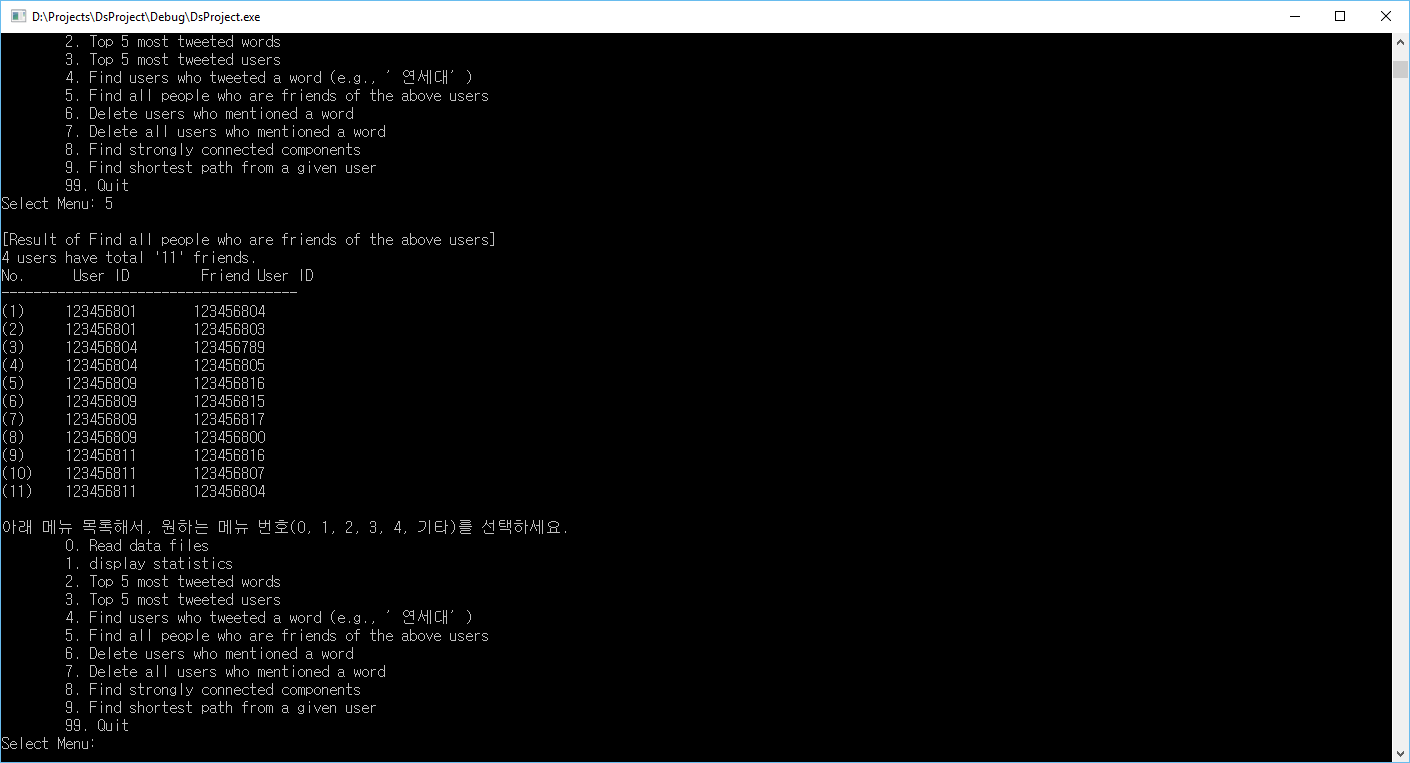
//printf("+ (%d) %s\n", s\_TweetUserCount, s\_TweetUserList[s\_TweetUserCount]);

s\_TweetUserCount++;

}

# **Find all users who are friend of the above user**

Friend 데이터 목록에 대해 Linear Search를 이용해, 메뉴 4번 작업에서 검색한 User 목록에 대한 Friend 목록을 검색한다. 결과화면은 메뉴 4번 작업에서 검색한 User 목록에 대해, 해당 User가 Friend 관계를 가지고 있는 User들의 User ID 목록을 출력하고 있다. 아래 화면은 4명의 User “123456801”, “123456804”, “123456809”, “123456811”에 대한 11명의 Friend 목록을 출력하고 있다.



메뉴 5번 작업을 배열을 자료구조로 사용하는 Friend 목록을 검색하기 때문에, O(n)의 Complexity로 문제를 해결했다. 하지만 Friend 목록을 구성하기 위해 다시 O(m)의 Complexity가 필요하기 때문에, 전체적으로는 O(n\*m)의 Complexity로 문제를 해결했다.

//검색한 Friend 목록

static int s\_FriendCount = 0;

static struct friendInfo\_t s\_FriendList[MAX\_FRIEND\_COUNT];

//Friend 목록에서, 주어진 User들의 Friends 목록을 추출한다.

s\_FriendCount = 0;

for (i = 0; i < g\_TweetCount; i++)

{

for (j = 0; j < s\_TweetUserCount; j++)

{

if (strcmp(s\_TweetUserList[j], g\_FriendInfoList[i].fromUserID) == 0)

{

InsertFriends(&g\_FriendInfoList[i]);

}

}

}

void InsertFriends(struct friendInfo\_t \*friendUserID)

{

int i;

//이미 Friend 목록에 추가되어 있는지를 검사한다.

for (i = 0; i < s\_FriendCount; i++)

{

if (strcmp(s\_FriendList[i].fromUserID, friendUserID->fromUserID) == 0

&& strcmp(s\_FriendList[i].toUserID, friendUserID->toUserID) == 0)

{

//이미 Friend가 등록되어 있기 때문에 무시한다.

return;

}

}

//Friend 목록에 항목을 추가한다.

strcpy(s\_FriendList[s\_FriendCount].fromUserID, friendUserID->fromUserID);

strcpy(s\_FriendList[s\_FriendCount].toUserID, friendUserID->toUserID);

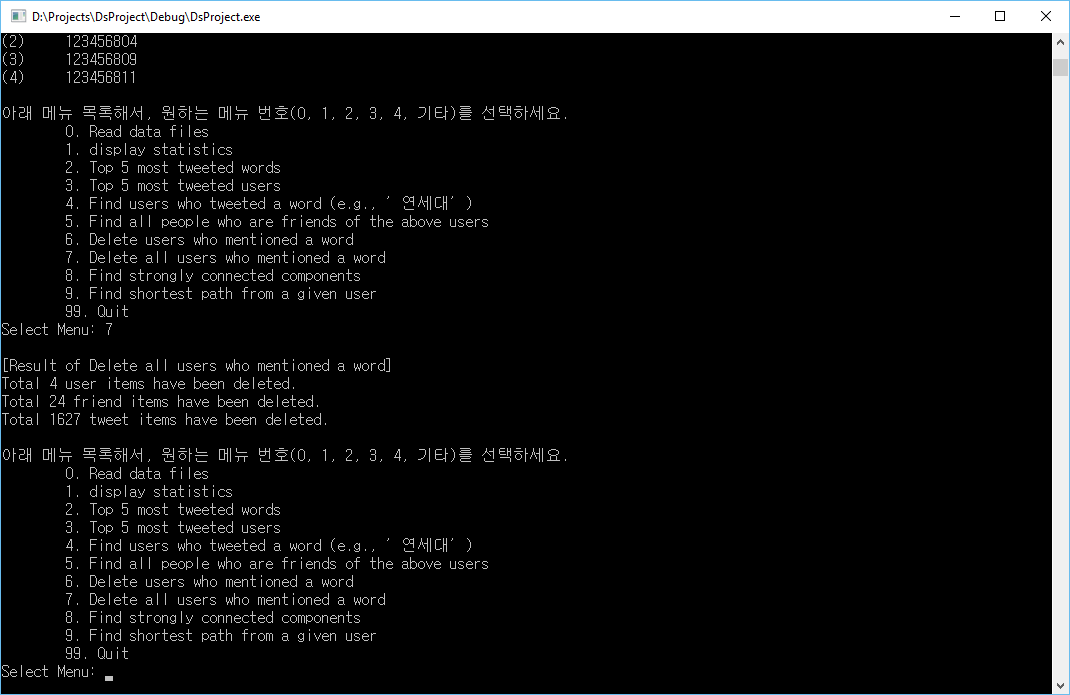
s\_FriendCount++;

}

# **Delete all users who mentioned a word**

메뉴 7번은 메뉴 4번에서 검색한 User들을 모두 삭제한다. 하지만 단순히 User 목록만 삭제하는 것이 아니라, User 정보 자체를 삭제해야 하기 때문에, 해당 User가 연계된 Friend 정보와 해당 User가 Tweet한 Tweet 정보들도 함께 모두 삭제해야 한다.

아래 화면은 메뉴 4번에서 검색한 4명의 User “123456801”, “123456804”, “123456809”, “123456811”를 삭제하고 있다. 뿐만 아니라 이들 4명의 User가 연계된 Friend 항목 24개, 이들 User들이 Tweet한 Tweet 항목 1,627개를 모두 삭제했다고 알려주고 있다.



작업의 핵심은 User 목록에서 User 정보를 삭제하는 것이 아니라, User 자체를 삭제하는 것이기 때문에, User 정보는 물론 Friend 정보와 Tweet 정보를 모두 함께 삭제해야 한다는 것이다. User, Friend, Tweet 정보는 배열을 자료구조로 사용해 구현되었기 때문에, 1개의 항목을 삭제하려면 O(n) Complexity가 필요하다. 따라서 m개의 항목을 삭제하기 때문에, 전체적으로 O(n\*m) Complexity를 사용해서 문제를 해결했다.

//User 목록에서 항목 삭제.

i = 0;

while (i < g\_UserCount)

{

for (j = 0; j < s\_TweetUserCount; j++)

{

if (strcmp(g\_UserInfoList[i].userID, s\_TweetUserList[j]) == 0)

{

//User 정보 삭제.

for (k = i + 1; k < g\_UserCount; k++)

{

g\_UserInfoList[k - 1] = g\_UserInfoList[k];

}

g\_UserCount--;

//삭제된 User 정보개수 갱신.

s\_DeleteUserCount++;

break;

}

}

if (j >= s\_TweetUserCount) i++;

}

# **Top 5 strongly connected components**

메뉴 8번 Graph에서 Strongly Connected Component를 추출하는 문제는 Kosaraju의 알고리즘을 채용해서 구현했다.

# **Graph 구현**

* 자료구조는 Matrix(2차원 배열) 사용.
* 자료구조 Graph에서 Node는 User에 해당하고, User들 사이에 Friend 관계가 있는 경우에 Directed Edge가 존재한다.
* Friend 관계가 있는 User들 사이에 Directed Edge 표현. 즉, Graph[i][j]는 i라는 User가 j라는 User를 Friend로 하고 있는 경우에 1이라는 값을 가지고, Friend가 아닌 경우에는 0이라는 정보를 가진다.

//Directed Graph를 위한 자료구조.

long s\_nNodeCount = 0; //Graph의 Node 개수.

long s\_nEdgeCount = 0; //Grapho의 Edge 개수.

//Directed Graph. s\_Graph[i][j] = Edge i -> j의 weight.

long s\_Graph[MAX\_USER\_COUNT + 1][MAX\_USER\_COUNT + 1];

//Directed Graph 구성.

void BuildGraph()

{

long fromNode, toNode;

long i;

//Graph의 Directed Edge 목록 구성(순방향 Edge, 역방향 Edge)

for (i = 0; i < s\_nEdgeCount; i++)

{

fromNode = atol(g\_FriendInfoList[i].fromUserID) - USER\_ID\_START;

toNode = atol(g\_FriendInfoList[i].toUserID) - USER\_ID\_START;

//weight가 없는 directed graph의 Edge 구성.

s\_Graph[fromNode][toNode] = 1; //Graph의 Directed Edge 구성.

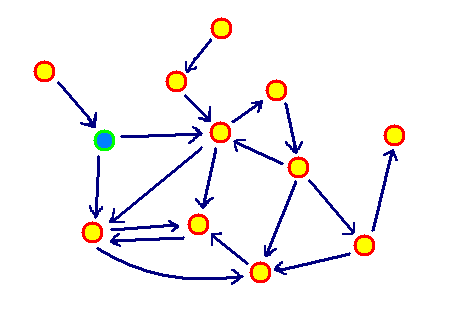
}

}

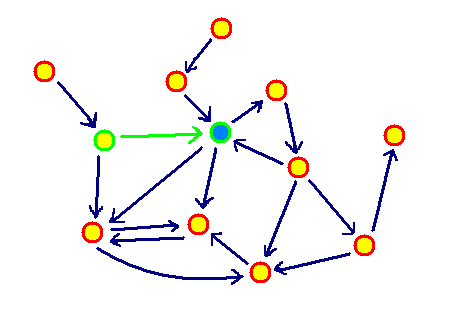
# **Koraraju의 알고리즘**

Graph에서 Strongly Connected Component를 추출하는 Kosaraju의 알고리즘 개요는 다음과 같다.

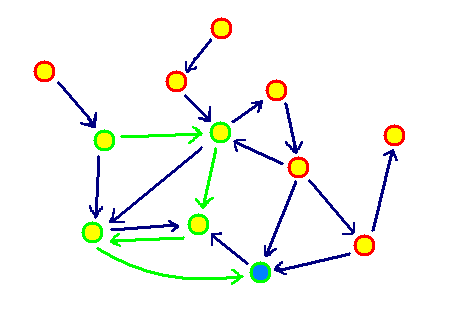
* 일단 임의의 정점을 선택한다.



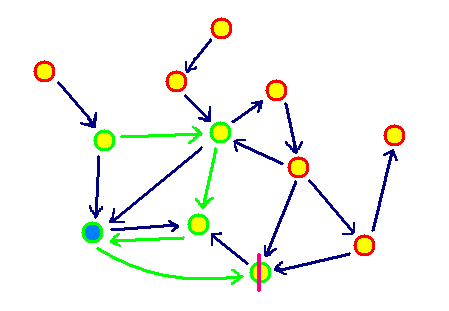
* 그리고 이 정점에서 시작하여 갈수 있는 인접한 정점으로 이동한다. (파란색으로 칠해 논 점은 현재 위치이고, 연두색 테두리는 이미 방문한 정점을 나타낸다)

[](http://pds24.egloos.com/pds/201208/02/23/d0091423_501a7fef39102.png)

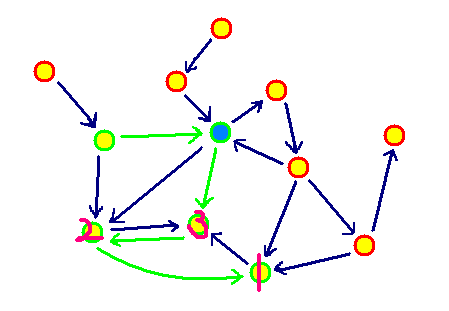
* 이동했으면, 이동한 정점에서 똑같이 갈 수 있는 인접한 정점을 찾아서 간다. 이걸 계속 반복한다. 단, 이미 방문했던 정점(연두색 테두리)은 다시 갈 수가 없다!

[](http://pds25.egloos.com/pds/201208/02/23/d0091423_501a8028d4d13.png)

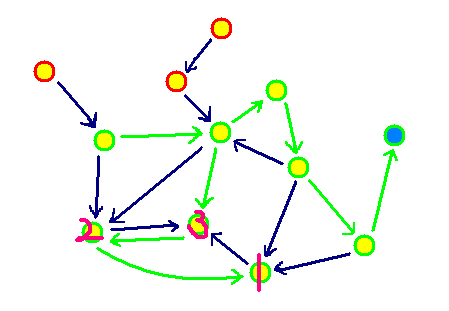
* 가다 보면, 위의 경우와 같이 더 이상 어떤 정점으로도 이동할 수 없는 경우가 생긴다. 그러면 그 정점에 1번 번호를 매긴다.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a80a7aa006.png)

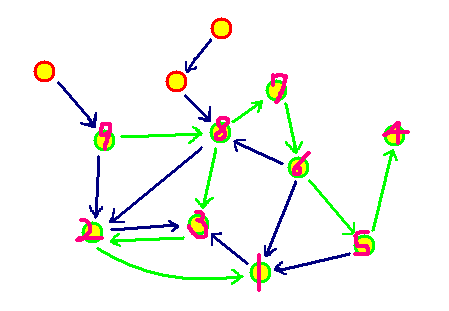
* 번호를 매겼으면, 그 점으로 온 경로를 거슬러가 바로 이전 점으로 돌아간다. (위 그림의 파란색 점). 이 점에서도 더 이상 갈 수 있는 경로가 없으므로 번호를 2번으로 매겨준다. 2번점 바로 이전 점에서도 갈 수 있는 경로가 없으므로 3번을 매겨준다.

[](http://pds22.egloos.com/pds/201208/02/23/d0091423_501a80fcdb5ca.png)

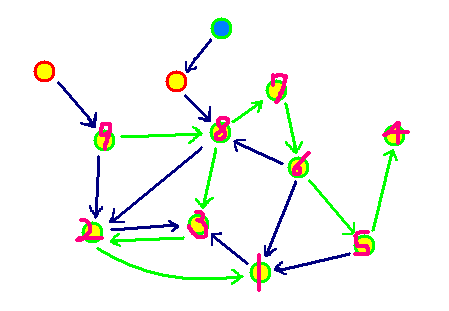
* 3번의 이전 점으로 돌아왔다. 여기서는 아직 갈 수 있는 경로가 남아있다!! 길이 보이면 무작정 계속 가자.

[](http://pds24.egloos.com/pds/201208/02/23/d0091423_501a8147525b8.png)

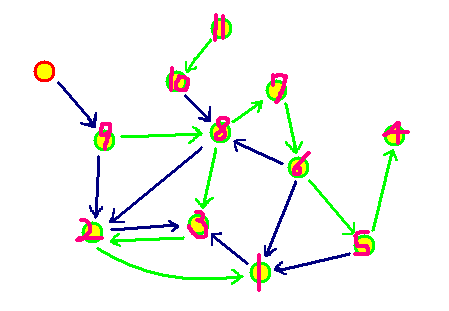
* 위와 같이 쭉쭉 가다 보면 또 막힌다. 그러면 아까와 같이 막히는 점에 대해 순서대로 번호를 매겨준다. 이제 4번부터 매길 차례.

[](http://pds25.egloos.com/pds/201208/02/23/d0091423_501a816f149e8.png)

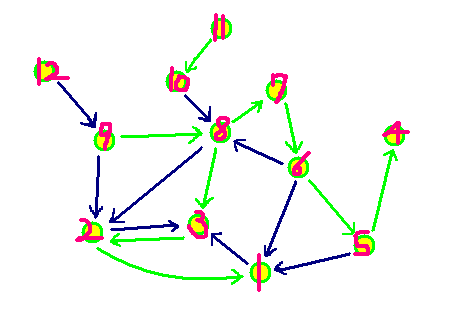
* 이렇게 해서 9번까지 번호를 매겼다. 처음 시작했던 점까지 번호가 매겨졌다. 그럼 다음 과정은?? 다시 맨 처음으로 돌아와서, 임의의 점 하나를 선택하고 아까와 같은 짓을 반복하면 된다.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a81cb8db5c.png)

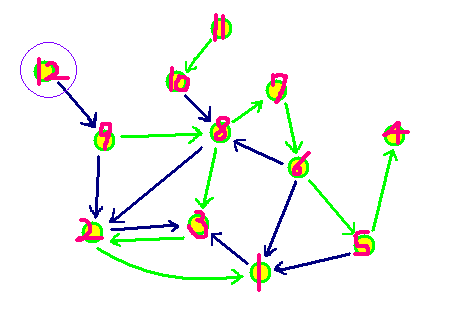
* 임의의 점을 골라서,

[](http://pds24.egloos.com/pds/201208/02/23/d0091423_501a81e04fd33.png)

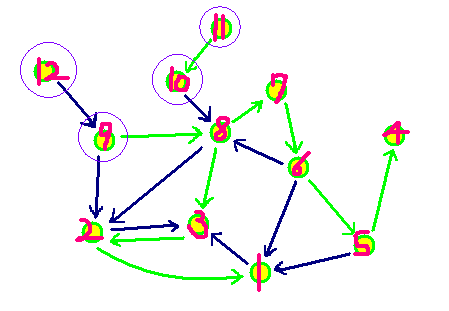
* 11번까지 번호를 매기고...

[](http://pds25.egloos.com/pds/201208/02/23/d0091423_501a81f01b79c.png)

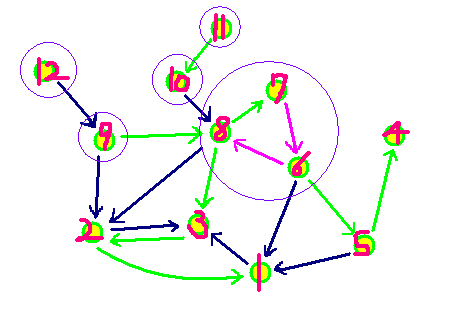
* 마무리!!
* 그 다음 과정은, 이제 가장 숫자가 큰 번호부터 탐색을 다시 하는데, 이번에는 화살표가 나가는 방향 말고, 화살표가 자신으로 들어오는 방향으로 이동을 하는 것이다. 일단 12번 점을 선택하고, 자신으로 들어오는 화살표가 없으므로.. 12번 점만 묶어준다.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a82f83d136.png)

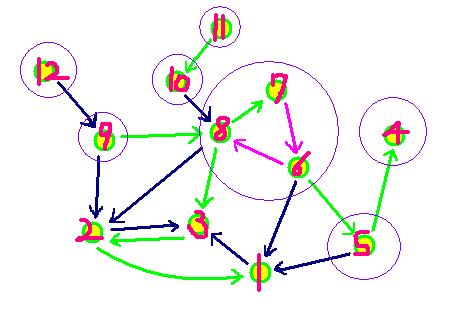
* 11번, 10번, 9번도 마찬가지.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a831f76543.png)

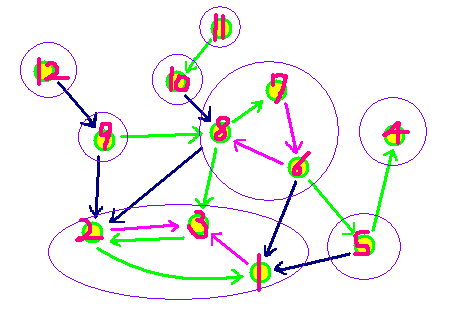
* 8번점은.. 6번점으로 갈 수 있는 역방향 화살표가 있다. 그러면 6번으로.. (10번 점은 이미 동그라미 친 점이므로 못 감). 6번에서도 똑같이 역방향 화살표를 찾는다. 7번으로 갈 수 있으니 간다. 이제 더 이상 역방향 화살표로 갈 수 있는 곳이 없으므로, 지금까지 갔던 모든 점들을 묶어준다.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a8460d5fed.png)

* 그 다음, 안묶인 점들 중에서 번호가 제일 높은 5번 점부터 다시 시작한다. 5, 4번 둘다 똑같이 아무런 경로가 존재하지 않는다.

[](http://pds25.egloos.com/pds/201208/02/23/d0091423_501a847f0e6c2.png)

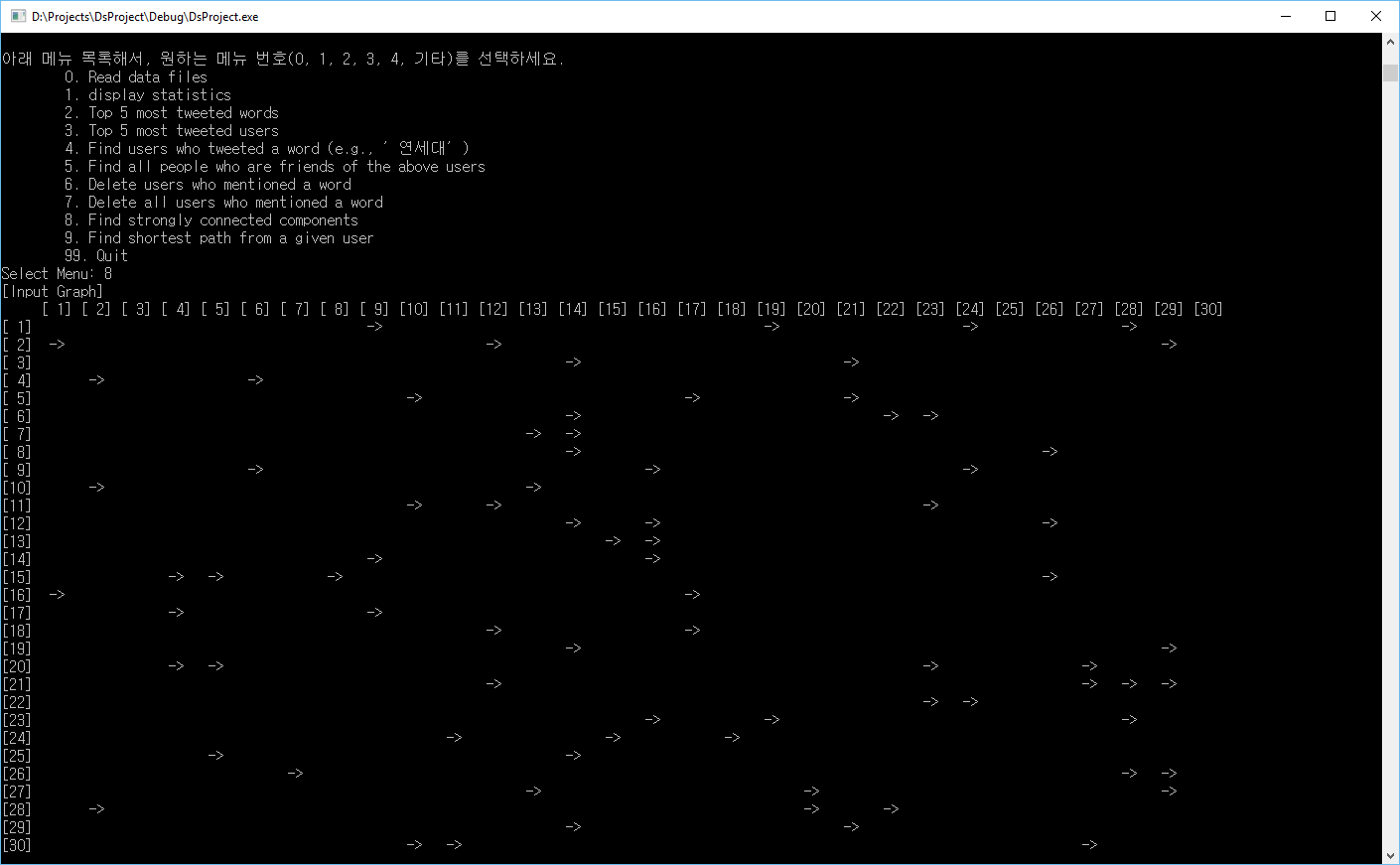
* 3번 점은 역방향 경로가 2개나 있다. 모두 방문해준다.

[](http://pds23.egloos.com/pds/201208/02/23/d0091423_501a84b913421.png)

* 묶어주면, 이제 모든 점에 대한 방문이 끝이 났다. 여기서 같은 동그라미 내에 있는 것들은 같은 SCC 내에 있다고 보면 된다. 즉, 이 그래프는 8개의 SCC로 이루어져 있다.

# **Strongly Connected Component 추출 기능 구현**

사용자가 메뉴 8번을 선택하면, 일단 현재 사용하는 데이터에 대한 Friend 관계를 화면에 출력하고 있다. 화면에서 y축인 j를 의미하고, x축은 i를 의미한다. 따라서 아래 화면의 경우 User 1은 User 9, 19, 24, 28을 Friend로 하고 있다는 것을 알 수 있다.



2차례의 DFS(Depth First Search)를 사용해서 Strongly Connected Component를 추출하고 있다. 첫 번째 DFS에서는 Spanning Tree를 구성해서 각 Node에 Sort No.(이후에 Strongly Connected Component를 찾기 위해 방문하는 순서의 역순)를 할당한다.

//주어진 nodeID를 root로 사용해서 순방향 Edge를 따라가며, 방문 가능한 모든 Node들을 방문한다.

//Recurrsing을 사용하기 때문에, 방문 가능한 모든 Node들을 방문한다.

void DfsSpanningTree(long nodeID)

{

long nodeTo;

long i;

//Node에 대한 방문을 표시.

s\_FlagVisited[nodeID] = true;

//Node가 가지고 있는 모든 순방향 Edge에 대해 DFS 방식으로 Spanning Tree를 만든다.

for (i = 0; i < s\_nNodeCount; i++)

{

//Edge의 Target이 아직 방문하지 않은 것이라면, Target에 대해 DFS Spanning Tree를 구성한다.

if (s\_Graph[nodeID][i] < 1) continue; //Invalid Edge는 무시.

nodeTo = i; //next Outgoing Edge의 target Node -> next Adjacent Node.

printf(" %ld", nodeTo + 1);

if (!s\_FlagVisited[nodeTo])

{

DfsSpanningTree(nodeTo);

}

}

//Node에 대한 방문번호를 등록한다.

for (i = 0; i < s\_nNodeCount; i++)

{

if (s\_SortedNode[i] >= 0) continue;

s\_SortedNode[i] = nodeID;

break;

}

}

그리고 Sort No. 역순으로 DFS를 사용해, Strongly Connected Component들을 판별한다.

//주어진 nodeID를 root로 사용해서 역방향 Edge를 따라가며, 방문 가능한 모든 Node들을 방문한다.

//Recurrsing을 사용하기 때문에, 방문 가능한 모든 Node들을 방문해서, SCC(Strongly Connected Component)를 구성한다.

void DfsFindSCC(long nodeID, long nSccComponentId)

{

long nodeFrom;

long i;

//방문 기록을 지워서, SCC 확인이 끝났음을 표시한다.

s\_FlagVisited[nodeID] = false;

//해당 Node의 SCC ID를 배정한다.

s\_SccResult[nodeID] = nSccComponentId;

//자신으로 들어오는 역방향 Edge가 있으면, 모두 조사한다.

for (i = 0; i < s\_nNodeCount; i++)

{

//자신으로 들어오는 역방향 Edge를 가진 Node의 SCC가 아직 결정되지 않았다면,

// 역시 DFS 방법으로 방문 가능한 모든 역방향 Edge를 조사한다.

if (s\_Graph[i][nodeID] < 1) continue;

nodeFrom = i;

if (s\_FlagVisited[nodeFrom])

{

printf(" %ld", nodeFrom + 1);

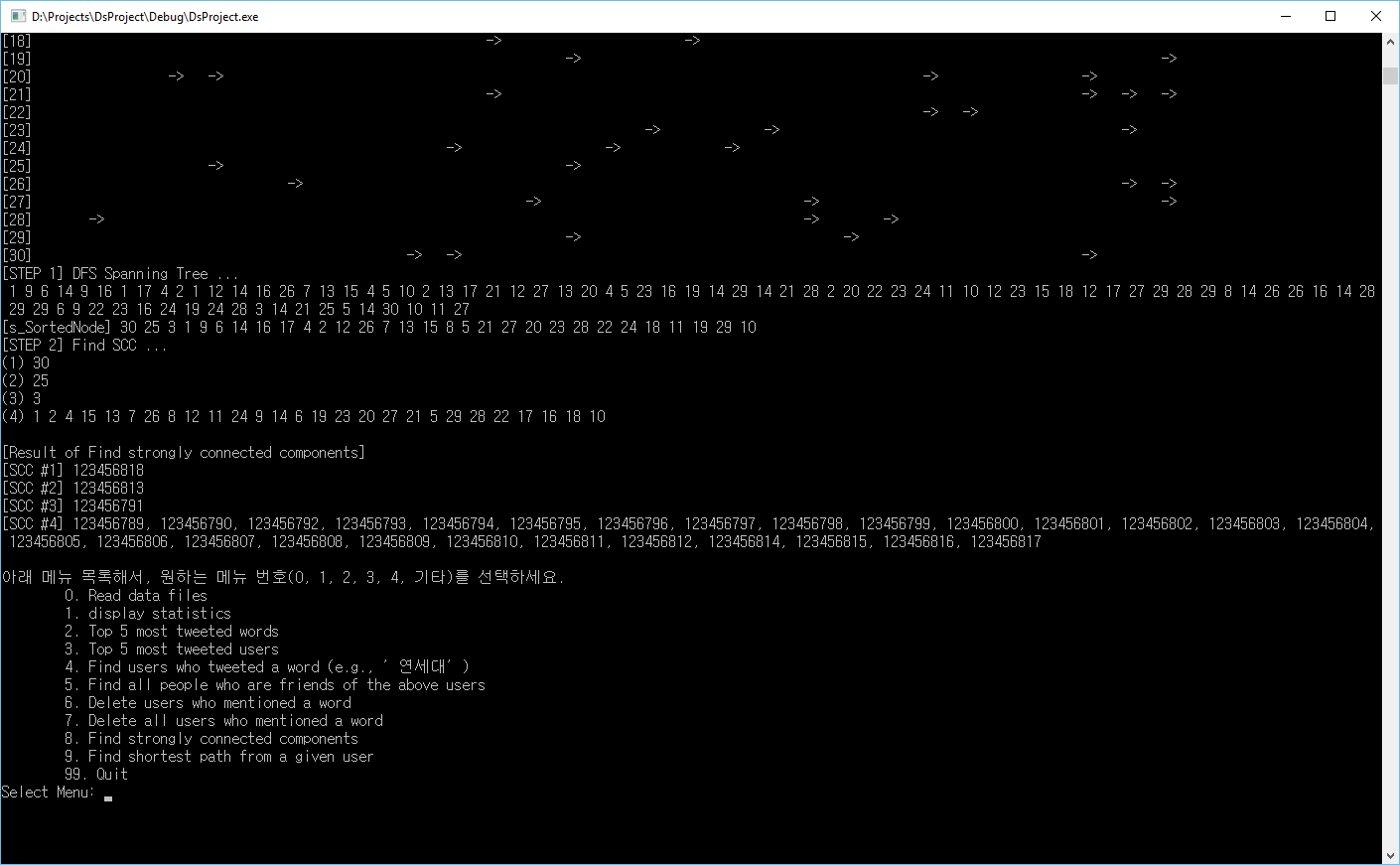
DfsFindSCC(nodeFrom, nSccComponentId);

}

}

}

아래 결과화면은 모두 4개의 Strongly Connected Component가 존재하고 있음을 알려주고 있다. 3명의 User들은 자신만이 고립되어 있고, 이들을 제외한 모든 User들은 동일한 Strongly Connected Component에 포함되어 있음을 보여주고 있다.



# **Find shortest path from a user (id)**

메뉴 9번 Graph에서 Shortest Path 추출 문제는 Dijkstra 알고리즘을 채용해서 구현했다.

# **Graph 구현**

Graph의 자료구조는 Strongly Connected Component 추출을 위해 사용했던 것과 동일한 것을 사용. 즉, 자료구조 Graph에서 Node는 User에 해당하고, User들 사이에 Friend 관계가 있는 경우에 Directed Edge가 존재한다. Graph는 2차원 배열 Matrix를 자료구조로 사용했다.

1가지 차이점은 Edge에 weight를 부여하는 방법이다. Shortest Path 추출을 위한 Graph에서Directed Edge의 weight는 Friend 수를 사용했다. 따라서 Graph[i][j]는 i라는 User가 j라는 User를 Friend로 하고 있는 경우에 j라는 User에 존재하는 Friend 수를 값을 가지고, Friend가 아닌 경우에는 0이라는 정보를 가진다.

long s\_nNodeCount = 0; //Graph의 Node 개수.

long s\_nEdgeCount = 0; //Grapho의 Edge 개수.

//Directed Graph. s\_Graph[i][j] = Edge i -> j의 weight.

long s\_Graph[MAX\_USER\_COUNT + 1][MAX\_USER\_COUNT + 1];

//Friend 정보를 이용해서 자료구조 Graph 구성.

void BuildGraph()

{

long weight;

long fromNode, toNode;

long i;

//모든 Friend 정보를 이용해서 각 User와 Friend들에 대한 Directed Graph 구성.

for (i = 0; i < s\_nEdgeCount; i++)

{

fromNode = atol(g\_FriendInfoList[i].fromUserID) - USER\_ID\_START;

toNode = atol(g\_FriendInfoList[i].toUserID) - USER\_ID\_START;

//weight를 가지는 directed graph의 Edge 구성.

weight = g\_UserFriendInfoList[toNode].count; //weigt는 toNode Node의 Friend 수.

s\_Graph[fromNode][toNode] = weight; //Graph의 Directed Edge 구성. Weight 값 배정.

}

}

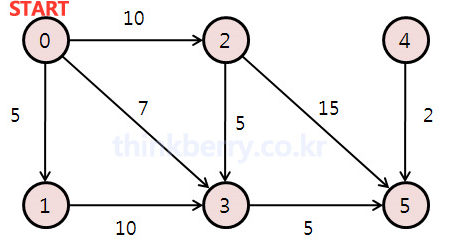
# **Dijkstra 알고리즘**

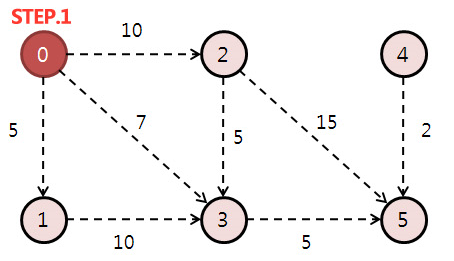
Shortest Path 알고리즘의 기본 아이디어는 다음과 같다.

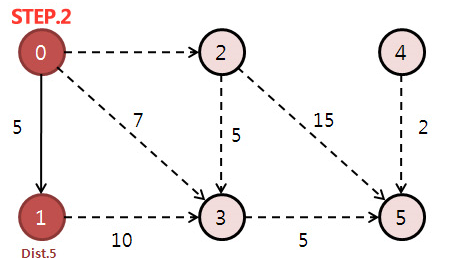
* 시작점을 '최단경로를 찾은 정점 리스트'에 담아둔다.
* '최단경로를 찾은 정점 리스트'에 들어있는 정점으로부터 인접한 정점들 중 가장 가중치가 적은 정점을 하나를 탐색한다. 이는 곧 해당 정점까지의 최단경로이다.
* 리스트 내에서의 정점들을 기준으로 가장 가중치가 적은 정점을 다시 탐색한다. 이때 발견되는 정점까지의 가중치는 직전 정점까지의 가중치 + 발견된 정정까지의 가중치로 계산된다. (엄밀히 말해 1과 2는 같다)
* 가능한 모든 정점을 연결할 때까지 반복한다.

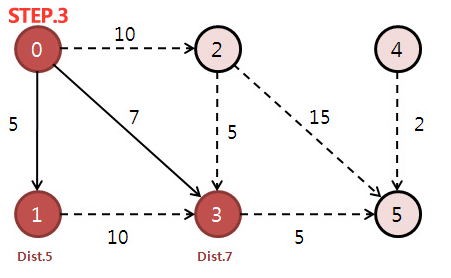
구체적인 예를 통해 Shortest Path를 구하는 방법은 다음과 같다.

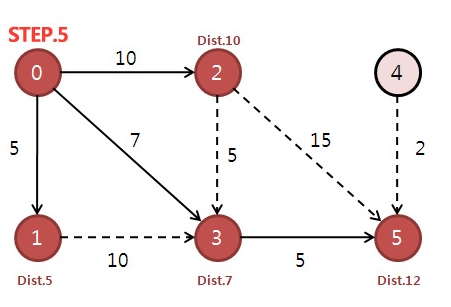
* 시작 정점부터 가장 가까운 정점을 선택하고 이를 리스트에 담아 기억하고 있다가 이들 정점들로부터 시작점으로부터의 가중치가 적은 간선 하나만 선택한다. 계속 반복해서 모든 경로를 잇게 되면 시작점부터 모든 점까지의 최단경로는 모두 구해지는 것이다.

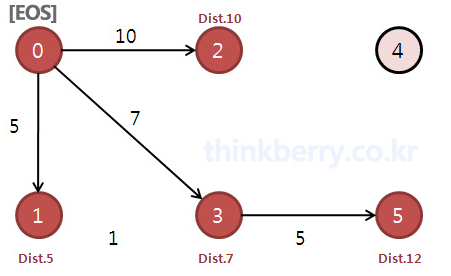








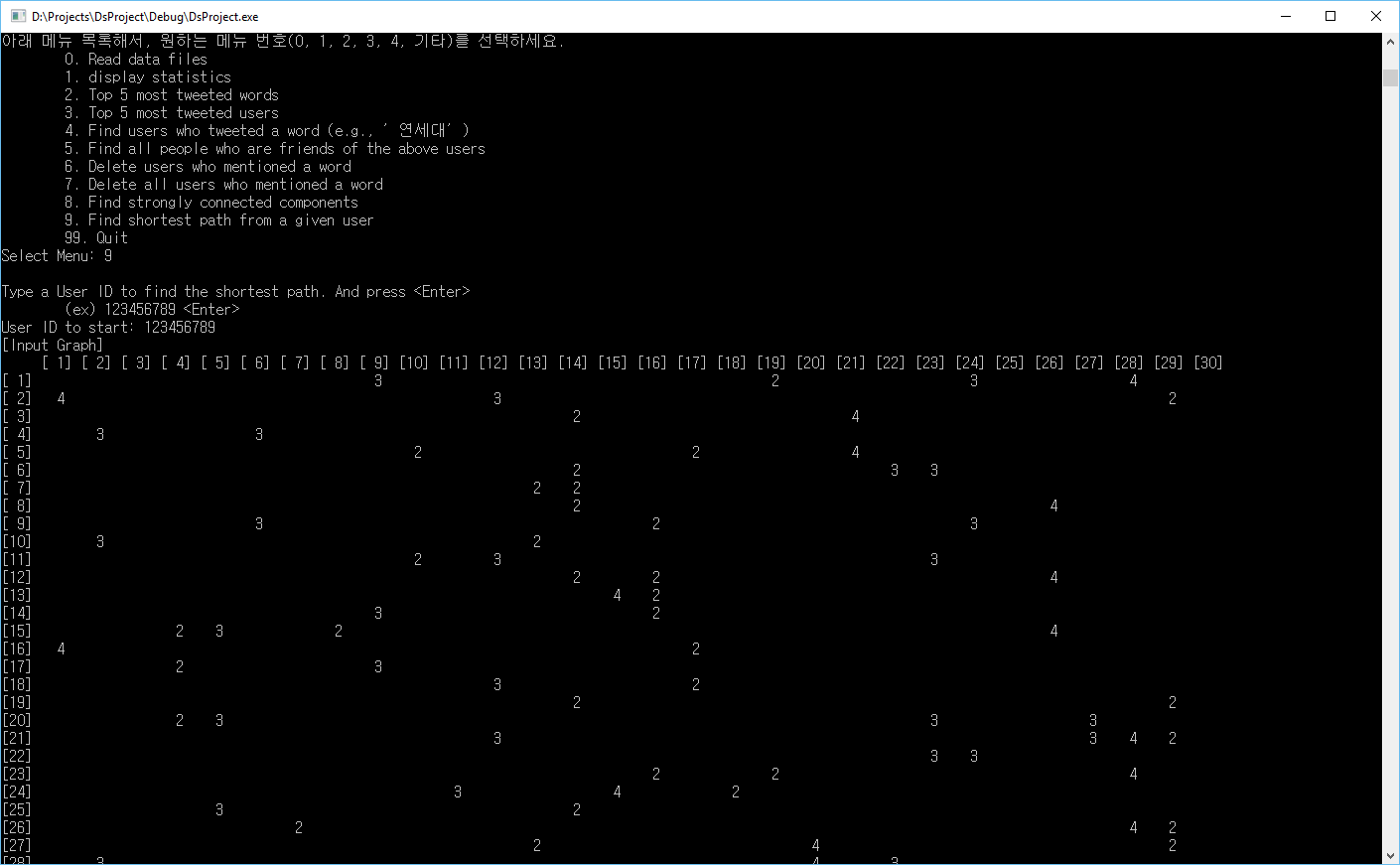




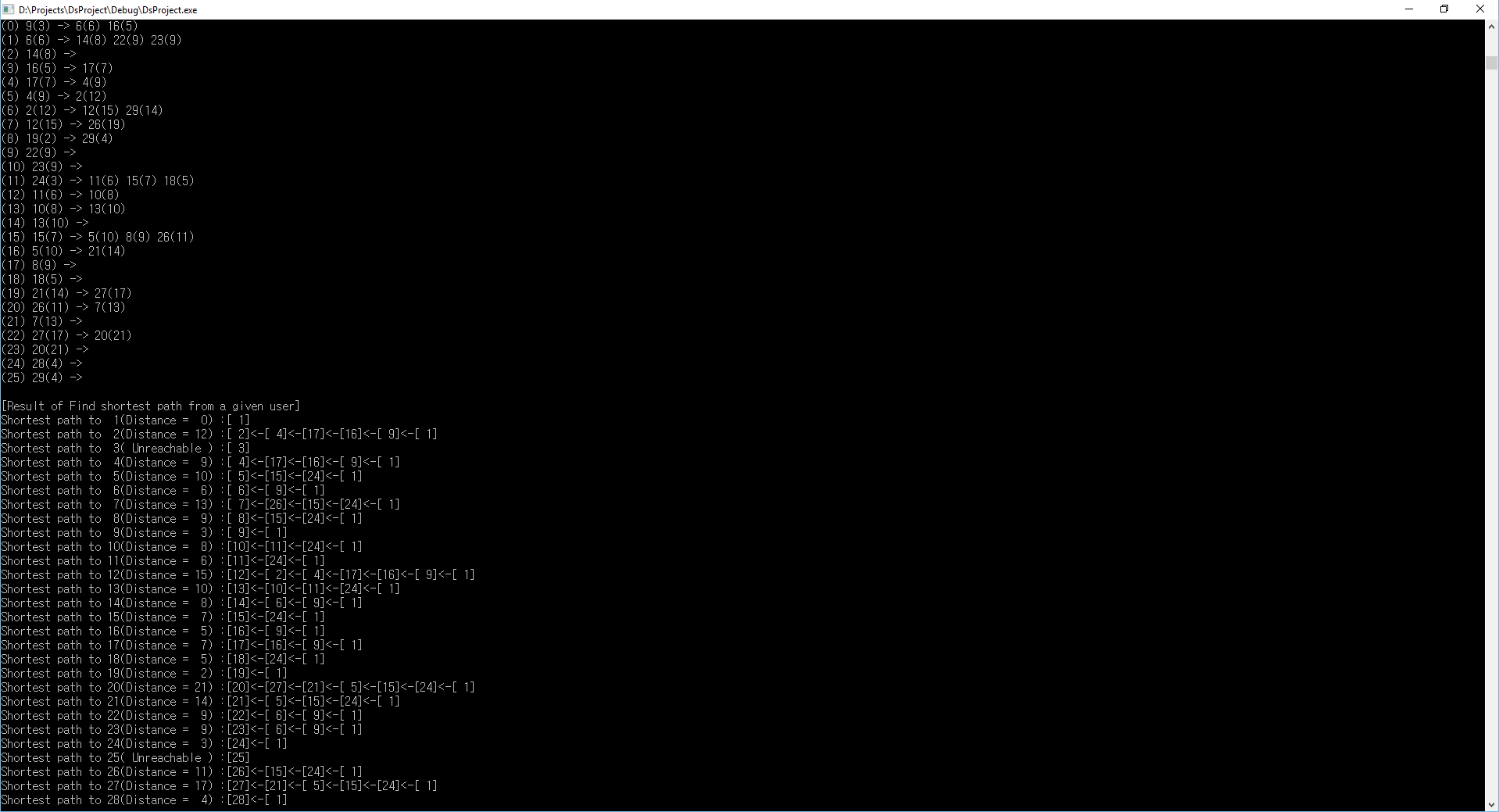
* Shortest Path 알고리즘은 시작점으로부터 모든 정점까지의 최단거리를 계산할 수 있다. 시작점으로부터 가장 가까운 거리의 정점을 우선 찾고 시작점부터 발견된 정점까지의 거리(가중치)를 별도로 기억한다(distance). 그리고 발견된 모든 정점들로부터 가장 가중치가 적은 정점을 계속해서 찾아 나감으로 써 최적의 경로를 모두 탐색하는 것이다. 핵심은 가중치가 누적되고, 가장 적은 가중치의 정점을 계속해서 찾아가는 이른바 갈망법(greedy method)이다.

# **Shortest Path 기능 구현**

먼저 사용자로부터 Shortest Path를 추출하는 시작 위치에 해당하는 User ID를 입력 받는다. 다음에는 Friend 정보를 조사해서 이차원 배열을 자료구조로 사용하는 Graph를 구성해서 화면에 출력한다. 화면에서 y축인 j를 의미하고, x축은 i를 의미한다. 따라서 아래 화면의 경우, User 1은 User 9, 19, 24, 28을 Friend로 하고 있다는 것을 알 수 있다. 그리고 User 9, 19, 24, 28는 각각 3, 2, 3, 4명의 Friend를 가지고 있다는 것을 의미한다.



사용자가 입력한 User에 해당하는 Node로부터 시작해서, Shortest Path를 추출한다. 아래 결과 화면은 Shortest Path 추출을 위해, 검사를 수행하는 Node들의 작업순서가 화면에 출력되고 있다. 즉, 아래 화면에서는 시작 위치에 해당하는 User 1번부터 시작해서 9 -> 6 -> 14 -> 16 -> … 순서로 작업을 진행하고 있음을 알 수 있다.



이처럼 작업을 진행해서 완료한 다음에는, 각 User까지 도달하기 위한 Shortest Path의 경로와 Distance를 출력하고 있다. 예를 들어 위의 결과 화면에는 User 1에서 User 2에 도달하기 위해 “1 -> 9 -> 16 -> 17 -> 4 -> 2”라는 Path를 Shortest Path로 가지고, Distance는 12라는 것을 보여주고 있다. 그리고 User 3의 경우에는 User 1로부터 도달하는 Path가 존재하지 않는다고 보여주고 있다.

# **개선이 필요한 내용**

* 샘플 데이터 변경을 위한 프로그램 수정

프로그램의 Symbol을 통해 생성하는 샘플 데이터를 정의하고 있다. 프로그램에 인자를 전달해서 샘플 데이터를 설정할 수 있도록 한다면, 샘플 데이터를 변경하기 위해 프로그램을 수정할 필요가 없을 것 같다.

* User별 또는 Word별 데이터 구축의 효율성 개선.

메뉴 작업 1번, 3번에서 작업을 처리하기 위해 User별 Friend 및 Tweet 정보를 구축해야 했다. 구현은 O(n2)의 Complexity 알고리즘을 사용했는데, 좀 더 효율적인 알고리즘으로 개선할 수 있을 것 같다.

* Delete 작업의 효율성 개선.

배열을 기본 자료구조로 사용했기 때문에, 데이터 목록에서 1개의 항목을 삭제하려면 O(n)의 Complexity가 필요했다. 만일 Linked List로 자료구조를 사용했다면 O(1)의 Complexity로 문제를 해결할 수도 있기 때문이다.