2018 年度 情報領域演習第三
— 第 14 回 —

注意事項

- 今回はグラフ構造データを扱うアルゴリズムについて実装を通じて学習する．今回もアルゴリズムを理解しないと解くことが難しいため，いきなり問題文や入出力例を読まずに，必ずその前にある説明をよく読むこと．わからないことがあれば TA や教員に質問しよう．
- 演習時間終了時点での提出状況についても評価の対象となるので，時間内にできるだけ多くの問題に挑戦するとよい．また，遅刻や途中退室は記録に残るので注意すること．
- プログラムの形式が指定されている場合には，それに従うこと．従わなくても「成功」と表示されることがあるが，得点にはならない．また，採点の際は `checker` とは異なる入出力例を用いて動作確認するので，「失敗」する反例に対する小手先の対策だけでは得点にならないことがあるので注意せよ．

はじめに

前回同様，CED において以下のコマンド (青字の部分) を実行することで出席を表明できる．ただし，N は所属するクラス名 1，2，3 で置き換え，N の前には空白を入れないこと．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/14/checkerN
提出開始: 1 月 xx 日 14 時 40 分 0 秒
提出締切: 1 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2019-01-xx-14-58
 問題:     結果 | 提出日時            | ハッシュ値                        |
    1:   未提出 |                     |                                  |
    2:   未提出 |                     |                                  |
                          ⋮
```

出席を表明するには，授業の開始から 30 分以内に実行する必要がある．実行しても出席状況が「欠席」である場合は教員に伝えること．なお，上の出力は例であるので，実際の締切りは各自コマンドで確認せよ．

提出も同じコマンドを用いて以下のように実行する．ただし，N はクラス名 (1 から 3)，X は 1 から 5 のいずれかの問題番号であり，`prog.c` は提出する C プログラムのファイル名であり，X の前後には忘れずに空白を入れること．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/14/checkerN X prog.c
```

ここで，ファイル名として指定するのは，問題番号 X の問題を解く C プログラムのソースファイルであり，コンパイル済みの実行ファイルではない．ファイル名は特に指定しないが，「英数字からなる文字列 `.c`」などとすることが望ましい．このコマンドを実行することにより，指定されたファイルがコンパイルされ，実行テストプログラムが自動的に起動される．コンパイルに失敗した場合にはエラーとなり，提出されたことにはならないので注意すること．コンパイルが成功した場合には複数回の実行テストが行われ，実行テストにも成功すると「成功」と表示される．実行テストに失敗すると「失敗」と表示されるとともに反例となる入力と出力が表示されるので，失敗した理由を見つけるための参考にするとよい．なお，入出力が大きい場合やファイルとして入力したい場合には共に表示されるパスにあるファイルを使うと便利である．ただし，入力のない問題の場合は失敗しても反例は出力されない．

正しく提出できたか確認するためには以下のように出席の表明と同じコマンドを用いて確認できる．

```
[p1610999@blue00 ~]> /ced-home/staff/18jr3/14/checkerN
提出開始: 1 月 xx 日 14 時 40 分 0 秒
提出締切: 1 月 yy 日 14 時 39 分 0 秒
ユーザ: p1610999, 出席状況: 2019-01-xx-14-47
```

```
問題:    結果 | 提出日時            | ハッシュ値                          |
  1:     成功 | 2019-01-xx-15-33  | 9b762c81932f3980cf03a768e044e65b |
                      ⋮
```

ハッシュ値は，提出した C プログラムのソースファイルの MD5 値である．CED では `md5sum` コマンドを用いて MD5 値を見ることにより，提出したファイルと手元のファイルが同じものであるかを確認することができる．

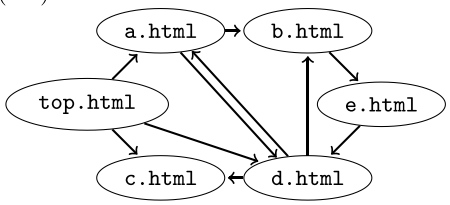
```
[p1610999@blue00 ~]> md5sum prog.c
9b762c81932f3980cf03a768e044e65b
```

なお，一度「成功」となった問題に対し，実行テストに失敗するプログラムを送信してしまうと，結果が「失敗」になってしまうので注意すること．

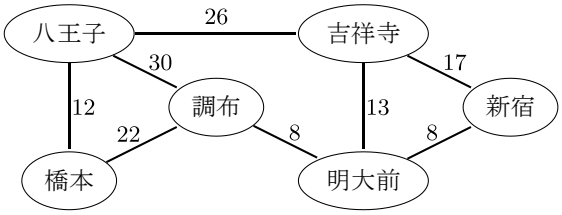
1 グラフ

前回までにリストや木といったデータ構造を扱うプログラムを作成してきたが，今回からグラフ構造を扱うプログラムを作成する．グラフとは頂点とそれらを結ぶ辺の集合からなるもののことで，下のような構造を指す．

(G1)



(G2)



(G1) は Web サイトのページのリンク関係を表していて，1 つの頂点は 1 つの Web ページに対応し，Web ページ v_1 から Web ページ v_2 へリンクが張られているとき，v_1 から v_2 への辺が矢印で描かれている．(G2) は調布を中心とした路線図を簡略化したもので，1 つの頂点は 1 つの駅に対応し，駅 v_1 から駅 v_2 の間に敷かれた線路が辺に対応する．(G2) の各辺に書かれた数値は所要時間 (単位は分) を表す．今回は (G2) のような路線図をデータとしてプログラムを作成してもらうが，まず，グラフの概念から復習しておこう．

隣接 グラフの頂点 A と頂点 B の間に辺があるとき，A と B は**隣接**しているという．

有向・無向 グラフ (G1) のように 2 つの頂点を結ぶ辺に向きがあるグラフを**有向グラフ**という．グラフ (G2) のように辺に向きがないグラフを**無向グラフ**という．無向グラフの各辺 $A - B$ を $A \rightleftarrows B$ のように向きのある 2 つの辺であると見なして，有向グラフと同様に扱うこともできる．

辺の重み グラフ (G2) のように各辺についている数値を**重み**といい，このグラフを**重み付きグラフ**という．最短経路問題や最小全域木問題などはこのような重み付きグラフを対象としている．

道 (経路) グラフ内のある頂点からいくつかの辺をたどったときに通る頂点の列を**道** (または**経路**) という．有向グラフの場合には向きの通りに辺をたどらなければならない．たとえば，(G1) における `a.html-d.html-b.html-e.html` や，(G2) における調布-八王子-吉祥寺は道である．同じ頂点を通るものも道と呼んでもよいが，同じ頂点を通らない道を**単純な道** (**単純な経路**) という．

閉路 ある頂点から始めて再びその頂点に戻ってくる道を**閉路**といい，同じ頂点を通らずに戻ってくる閉路を**単純な閉路**という．

連結 ある頂点 v_1 から別の頂点 v_2 への道が存在するとき，v_2 は v_1 から**到達可能**であるといい，グラフ内のどの頂点も他のすべての頂点から到達可能であるとき，そのグラフを**連結グラフ**という．

2 グラフの表現

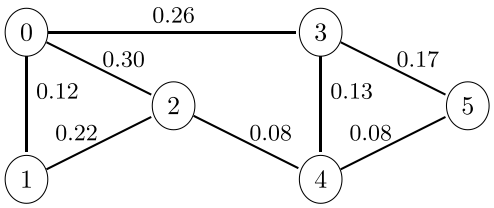
形式上はグラフは「頂点の集合 V と辺の集合 E の組」によって表し，各辺は頂点と頂点の 2 つ組 (またはそれらに重みも併記した 3 つ組) で表現する．たとえば，(G2) のグラフであれば，

$$V = \{\ \text{八王子}, \text{吉祥寺}, \text{調布}, \text{新宿}, \text{橋本}, \text{明大前}\ \}$$
$$E = \{\ (\text{八王子}, 26, \text{吉祥寺}), (\text{八王子}, 30, \text{調布}), (\text{八王子}, 12, \text{橋本}), (\text{吉祥寺}, 17, \text{新宿}),$$
$$(\text{吉祥寺}, 13, \text{明大前}), (\text{調布}, 8, \text{明大前}), (\text{調布}, 22, \text{橋本}), (\text{新宿}, 8, \text{明大前})\ \}$$

と表される．しかし，計算機上でこの表現をそのまま使うと効率が悪くなることが多い．たとえば，調布と隣接する頂点はどれかを知りたいときや，調布と橋本が連結されているかを知りたいときに，集合 E に含まれているすべての辺を見て「調布」を探すのはあまり賢い方法とは言えない．そこで今回は，計算機上でグラフを処理するための 2 つの方法として**隣接リスト表現**と**隣接行列表現**を C 言語で実装してみよう．

今回扱うグラフは鉄道の路線図を簡略化したもので，入力としては以下のような形式で与えられる．

```
6
0:1:1011:0.120
0:2:1012:0.300
0:3:1013:0.260
1:2:1014:0.220
2:4:1012:0.080
3:4:1015:0.130
3:5:1013:0.170
4:5:1012:0.080
```



簡単のため，この路線図ファイルでは駅名は番号 (以降，駅番号という) で表示されている．また，各線路にも路線ごとに割り振られた番号 (以下，路線番号という) がついているものとする [1]．上の形式の最初の行は頂点数 (駅の数) を表し，このファイルが表現するグラフの頂点は 6 個である．この場合，駅番号は 0 から 5 までの整数となる．続く各行は辺 (各駅間の線路) の情報を表し，

駅番号 1:駅番号 2:路線番号:距離

の形式で与えられている．たとえば，2 行めの `0:1:1011:0.120` は「駅番号 0 の駅と駅番号 1 の駅が路線番号 1011 の線路で繋がれていて，その距離が 0.120 km である」ということを表している．左の入力形式全体に対応するグラフは右のように図示できる (路線番号は省略している)．今回は，この形式で与えられるグラフに対して問題を解くことになるが，入力の方法は各問題で雛形として与えられるのでそれを利用すればよい．

路線図データについて　演習の Web ページに日本全国の駅を網羅したデータファイルを載せているので，必要に応じてダウンロードするとよい．

`tonari.txt` 先述の入力と同様の形式で記述したファイル．1 つの辺における 2 つの駅番号は必ず異なるが，距離が 0.000 であることもある．また，元データ [2] の駅の経度と緯度から距離を計算しているため，実際の距離とは大きく異なることもある．

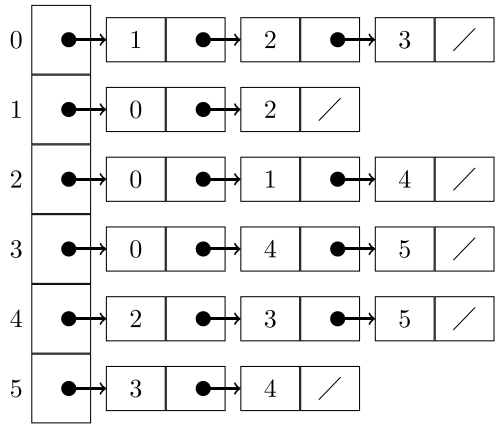
`eki.txt` 駅番号と駅名の対応を表すファイル．同じ駅でも路線が異なれば駅番号は異なる．たとえば，京王線の調布駅と京王相模原線の調布駅は異なる駅番号をもつ．

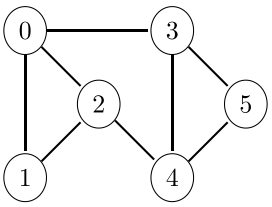
`rosen.txt` 路線番号と路線名の対応を表すファイル．徒歩での移動も想定されている 2 つの駅を辺でつなぐために,「徒歩」という特別な路線も用意している．

[1] たとえば，京王線なら `1012`，京王井の頭線なら `1015` といった具合である．
[2] このデータは `http://www.ekidata.jp/dl/` のデータを加工したもである．

2.1 隣接リスト表現

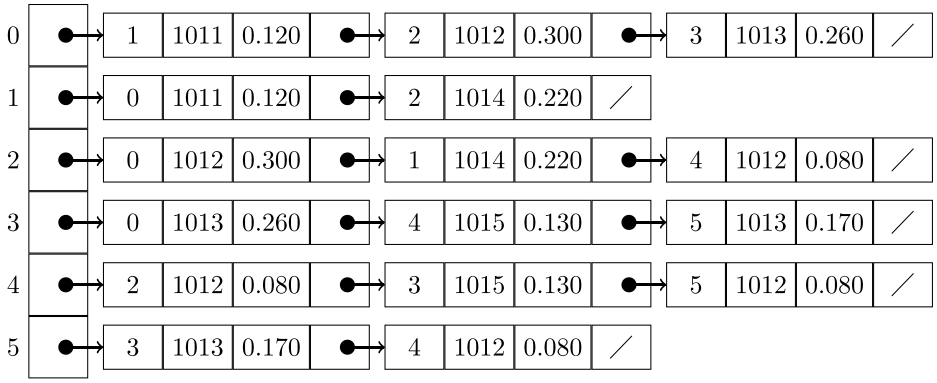
今回扱うグラフは辺の集合として表現することができるが，グラフを扱うアルゴリズムの多くは「ある頂点に対して隣接する頂点を見つける」といった操作を頻繁に行うため，辺の集合をあらかじめ整理しておく方が効率的であると考えられる．そこで考え出されたグラフの表現が**隣接リスト表現**で，頂点ごとに隣接する頂点 (1 つの辺で繋がっている頂点) のリストを用意することでグラフを表現する．たとえば，上のグラフなら頂点が 6 つなので長さ 6 の配列を用意して，各要素には，その添字に対応する頂点に隣接する頂点を繋いだ連結リスト (の先頭のアドレス) を格納すればよい．辺についた情報 (路線番号や距離) を無視すれば次のような形になる (下左図).





左の隣接リスト表現は右のグラフに対応する．頂点 0 は頂点 1 と頂点 2 と頂点 3 につながっているので，配列の 0 番めには頂点 1 と頂点 2 と頂点 3 が列んだ連結リストの先頭のアドレスが格納されている．頂点 1 は頂点 0 と頂点 2 につながっているので，配列の 1 番めには頂点 0 と頂点 2 が列んだ連結リストの先頭のアドレスが格納されている．グラフの表現としては連結リストの中の順序は気にしなくてよいが，わかりやすいようにここでは昇順に列べている．

今回扱う路線図の例では，各辺に路線番号や距離などの情報が付加されているため，次のような隣接リスト表現になる．



駅 0 からは 3 つの駅に繋がっていて，

- 駅 1 に繋がっている線路の路線番号が 1011，距離が 0.120,
- 駅 2 に繋がっている線路の路線番号が 1012，距離が 0.300,
- 駅 3 に繋がっている線路の路線番号が 1013，距離が 0.260

といった具合に表現すればよい．なお，このグラフは無向グラフであるため，駅 0 に対する隣接リストの中に駅 1 が含まれていれば，駅 1 に対する隣接リストの中には駅 0 が必ず含まれる．

これらを踏まえ，路線図を表す隣接リストを C 言語で実装するために，次で宣言される構造体 `node` を用意しよう．

```
struct node { int eki, rosen; float kyori; struct node *next; };
```

メンバ `eki` は駅番号，メンバ `rosen` は路線番号，メンバ `kyori` は距離，メンバ `next` は次の節点のアドレスを表す．これまで通り，連結リストの終わりであれば `NULL` を入れる．隣接リスト表現は，構造体 `node` のアドレスを要素とする配列で表される．

問題 1

この問題の目的は，辺の集合 (羅列) として与えられたグラフに対する隣接リスト表現を求めることである．具体的には，次の関数 `add_edge` を実装すればよい．

```
void add_edge(struct node *adjlist[], int eki1, int eki2, int rosen, float kyori);
```

関数 `add_edge` は,「隣接リスト表現 `adjlist` に，駅番号 `eki1` から駅番号 `eki2` への辺 (路線番号 `rosen`，距離 `kyori`) を追加する関数」である．すなわち，隣接リスト表現の配列のうち，添字 `eki1` にある連結リストの先頭に `eki2` を含む構造体 `node` を追加し，添字 `eki2` にある連結リストの先頭に `eki1` を含む構造体 `node` を追加すればよい．結果として，最初の方に追加した辺の情報ほど連結リストの後ろの方に現れることになる．

作成すべきプログラムの形式は後に示す `main` 関数を含むものとし，関数 `add_edge` の定義を適切に埋めることによりプログラムを完成させよ．標準入力から与えられる入力は先述の路線図データを表す文字列である．標準出力として出力される各行は，隣接リスト表現の配列の各要素の内容を添字の小さい方から順に，

駅番号: (駅番号, 路線番号, 距離) (駅番号, 路線番号, 距離) ... (駅番号, 路線番号, 距離)

という形式で出力するものとする．最初の駅番号は添字であり，コロン (:) の後ろに隣接する頂点の情報を隣接リストの先頭から出力する (後に示すプログラムの通りに記述すればよい)．なお，同じ駅から同じ駅への 1 つの辺だけでたどり着くことはないと仮定してよい．与えられる駅の数は 11000 以下であるとする．

入力例

```
3
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
```

出力例

```
0: (2,1023,4.000) (1,1012,2.000)
1: (2,1012,3.000) (0,1012,2.000)
2: (1,1012,3.000) (0,1023,4.000)
```

入力例

```
4
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
1:3:1008:2.500
2:3:1023:1.500
```

出力例

```
0: (2,1023,4.000) (1,1012,2.000)
1: (3,1008,2.500) (2,1012,3.000) (0,1012,2.000)
2: (3,1023,1.500) (1,1012,3.000) (0,1023,4.000)
3: (2,1023,1.500) (1,1008,2.500)
```

入力例

```
6
0:1:1011:0.120
0:2:1012:0.300
0:3:1013:0.260
1:2:1014:0.220
2:4:1012:0.080
3:4:1015:0.130
3:5:1013:0.170
4:5:1012:0.080
```

出力例

```
0: (3,1013,0.260) (2,1012,0.300) (1,1011,0.120)
1: (2,1014,0.220) (0,1011,0.120)
2: (4,1012,0.080) (1,1014,0.220) (0,1012,0.300)
3: (5,1013,0.170) (4,1015,0.130) (0,1013,0.260)
4: (5,1012,0.080) (3,1015,0.130) (2,1012,0.080)
5: (4,1012,0.080) (3,1013,0.170)
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えればよいが，必要に応じて補助関数を定義してもよい．

```c
#include<stdio.h>
#include<stdlib.h>
char buf[256];

struct node { int eki, rosen; float kyori; struct node *next; };

void add_edge(struct node *adjlist[], int eki1, int eki2, int rosen, float kyori) {
  /* ※ここを適切なプログラムで埋める */
}

/* 頂点数 n の隣接リスト表現を表示する関数 */
void print_adjlist(struct node *adjlist[], int n) {
  int i;
  struct node *p;
  for(i=0;i<n;++i) {
    printf("%d:", i);
    p = adjlist[i];
    while(p!=NULL) {
      printf("␣(%d,%d,%.3f)", p->eki, p->rosen, p->kyori);
      p = p->next;
    }
    printf("\n");
  }
  return;
}

int main() {
  int eki1, eki2, rosen, ekisu, i;
  float kyori;
  /* 頂点数 (駅の数) を ekisu に格納 */
  scanf("%d␣", &ekisu);
  struct node *adjlist[ekisu];
  /* 隣接リスト表現を初期化．すべての頂点に対する隣接リストを空にする */
  for(i=0;i<ekisu;++i) adjlist[i] = NULL;
  while(fgets(buf,sizeof(buf),stdin)!=NULL) {
    /* 隣り合う駅の情報を読取り */
    sscanf(buf, "%d:%d:%d:%f␣", &eki1, &eki2, &rosen, &kyori);
    /* そのデータを隣接リスト表現のグラフに追加 */
    add_edge(adjlist, eki1, eki2, rosen, kyori);
  }
  print_adjlist(adjlist, ekisu);
  return 0;
}
```

グラフの隣接リスト表現の扱いに慣れるために，次の問題も解いてみよう．

問題 2

この問題の目的は，2 駅離れた駅のうち最も遠い駅までの距離を計算する関数 two_hop_kyori を定義することである．

```
float two_hop_kyori(struct node *adjlist[], int eki);
```

この関数は,「隣接リスト adjlist で表される路線図データにおいて，駅番号 eki の駅から 2 駅離れた駅のうち，最も遠い駅までの距離を出力する関数」である．1 駅離れた駅は隣接リストを見ればわかるが，それらの駅に対する隣接リストにある駅のうち，元の駅を除いたものが 2 駅離れた駅である．なお，2 駅離れた駅が存在しないときは，1 駅離れた駅のうち遠い方の駅までの距離を返すものとする．1 駅離れた駅も存在しないとき (1 本も線路が繋がっていない駅の場合 [3]) には 0.000 を返す．

作成すべきプログラムの形式は後に示す main 関数を含むものとし，関数 two_hop_kyori の定義を適切に埋めることによりプログラムを完成させよ．標準入力から与えられる入力は問題 1 と同様に路線図データを表す無向グラフである．標準出力の各行には，駅番号ごとに「駅番号: 2 駅離れた駅までの距離の最大値 (小数点以下 3 桁まで表示)」の形式で出力する．

入力例

```
3
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
```

出力例

```
0: 7.000
1: 7.000
2: 6.000
```

入力例

```
4
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
1:3:1008:2.500
2:3:1023:1.500
```

出力例

```
0: 7.000
1: 7.000
2: 6.000
3: 5.500
```

入力例

```
6
0:1:1011:0.120
0:2:1012:0.300
0:3:1013:0.260
1:2:1014:0.220
2:4:1012:0.080
3:4:1015:0.130
3:5:1013:0.170
4:5:1012:0.080
```

出力例

```
0: 0.520
1: 0.520
2: 0.560
3: 0.560
4: 0.390
5: 0.430
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えればよいが，必要に応じて補助関数を定義してもよい．

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 char buf[256];
```

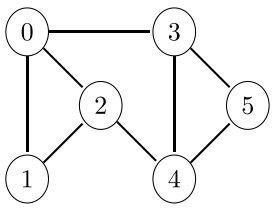
[3] 実際の路線図データにはそのような駅は存在しない．

```
4
5  struct node { int eki, rosen; float kyori; struct node *next; };
6
7  void add_edge(struct node *adjlist[], int eki1, int eki2, int rosen, float kyori) {
8    /* ※ここは問題 1 と同じ */
9  }
10
11 float two_hop_kyori(struct node *adjlist[], int eki) {
12   /* ※ここを適切なプログラムで埋める */
13 }
14
15 int main() {
16   int eki1, eki2, rosen, ekisu, i;
17   float kyori;
18   scanf("%d␣", &ekisu);
19   struct node *adjlist[ekisu];
20   /* 隣接リスト表現を初期化．すべての頂点に対する隣接リストを空にする */
21   for(i=0;i<ekisu;++i) adjlist[i] = NULL;
22   while(fgets(buf,sizeof(buf),stdin)!=NULL) {
23     /* 隣り合う駅の情報を読取り */
24     sscanf(buf, "%d:%d:%d:%f␣", &eki1, &eki2, &rosen, &kyori);
25     /* そのデータを隣接リスト表現のグラフに追加 */
26     add_edge(adjlist, eki1, eki2, rosen, kyori);
27   }
28   for(i=0;i<ekisu;++i)
29     printf("%d:␣%.3f\n", i, two_hop_kyori(adjlist, i));
30   return 0;
31 }
```

2.2 隣接行列表現

グラフ構造を処理する際に,「与えられた 2 つの頂点が隣接しているか」という判定を頻繁に必要とする場合がある．辺の集合をそのまま使うとすべての辺を見ないとわからないし，隣接リスト表現を使ったとしても 1 つの頂点から多数の頂点に辺がある場合には時間がかかってしまう．そこで考え出されたグラフの表現が**隣接行列表現**で，頂点数が N であれば $N \times N$ の行列を用い，i 番めの頂点と j 番めの頂点が隣接していれば行列の (i, j) 成分が 1，隣接していなければ 0 としてグラフの構造を表現する．たとえば，次の行列はその右に示したグラフに対応する [4]．

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$



頂点 0 から頂点 1 に辺があるので $(0, 1)$ 成分は 1，頂点 0 から頂点 4 に辺がないので $(0, 4)$ 成分は 0 となっている．このグラフは無向グラフであり，頂点 i から頂点 j に辺があれば頂点 j から頂点 i にも辺があるので，隣接行列は対称行列 [5] となる．重みのついた辺 (路線図データにおける路線番号や距離) を扱う場合に

[4] 数学では $N \times N$ 行列といえば $(1, 1)$ 成分から (N, N) 成分までであるが，ここでは C 言語に合わせて $(0, 0)$ 成分から $(N-1, N-1)$ 成分までとしている．

[5] 習っているはずであるが念のため補足しておくと，対称行列とは (i, j) 成分と (j, i) 成分が必ず等しい行列のことである．

は，各成分を 0 か 1 にする代わりにその重みで表現することがあるが，今回は 0 と 1 を用いた単純な隣接行列を扱う．

問題 3

この問題の目的は，辺の集合 (羅列) として与えられたグラフに対する隣接行列表現を求めることである．隣接行列表現は `int` 型 (0 か 1) を要素とする 2 次元配列で表現できるので，(i, j) 成分を `a[`i`][`j`]` に保持するような 2 次元配列 `a` で表せばよい．

作成すべきプログラムの形式は後に示す `main` 関数を含むものとし，※印のコメントで指定された箇所を適切に埋めることによりプログラムを完成させよ (入力の読み込みは問題 1 を参考にするとよい)．標準入力から与えられる入力は問題 1 と同様に路線図データを表す無向グラフである．プログラムは入力に対応する隣接行列を `adjmat` に格納し，標準出力にはその行列の各行を 0 と 1 の文字列として 1 行ずつ出力する (具体的な形式は出力例を参考にするとよい)．与えられる駅の数は 128 以下であるとする．

入力例

```
3
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
```

出力例

```
011
101
110
```

入力例

```
4
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
1:3:1008:2.500
2:3:1023:1.500
```

出力例

```
0110
1011
1101
0110
```

入力例

```
6
0:1:1011:0.120
0:2:1012:0.300
0:3:1013:0.260
1:2:1014:0.220
2:4:1012:0.080
3:4:1015:0.130
3:5:1013:0.170
4:5:1012:0.080
```

出力例

```
011100
101000
110010
100011
001101
000110
```

作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えればよいが，必要に応じて補助関数を定義してもよい．

```c
#include<stdio.h>
#include<stdlib.h>
char buf[256];

int main() {
  int ekisu;
  scanf("%d␣", &ekisu);
  int adjmat[ekisu][ekisu]; /* ← 隣接行列を表す 2次元配列 */
  /* ※ここを適切なプログラムで埋める */
  return 0;
}
```

注意 8 行めの宣言では，ekisu の値が大きい場合にメモリを確保しきれなくてセグメントエラーになることがあるので，tonari.txt のような大きな入力を扱いたければ，

```
adjmat = (int**)malloc(sizeof(int*)*ekisu);
for(int i=0;i<ekisu;++i) adjmat[i] = (int*)malloc(sizeof(int)*ekisu);
```

のように malloc 関数を用いる必要がある．ただし，checker ではそこまで大きな入力は与えられないので，8 行めの通りに宣言してもよい．

与えられたグラフ G に対して,「頂点 u と頂点 v が隣接していて，頂点 v と頂点 w が隣接しているとき，頂点 u と頂点 w が隣接していなければ辺でつなぐ」という処理を可能な限り繰り返して得られるグラフを G の**推移的閉包**という．G において，頂点 p から頂点 q に到達可能 (p と q の間に道が存在する) かどうかは，G の推移的閉包において p と q が隣接しているかを確かめることによって判定できる．上述の隣接行列表現を用いれば，推移的閉包を簡単に求めることができることを確かめてみよう．

問題 4

この問題の目的は，隣接行列表現で与えられたグラフに対し，そのグラフの推移的閉包の隣接行列表現を求めるプログラムを作成することである．具体的には，次の関数 warshall を実装すればよい．

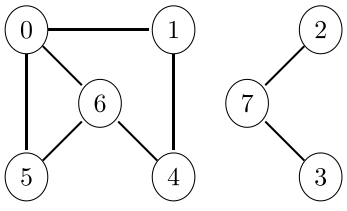
```
void warshall(int n, int adjmat[n][n], int result[n][n]);
```

この関数は「頂点数 n のグラフを表す隣接行列 adjmat に対して，そのグラフの推移的閉包の隣接行列 result を作成する関数」である．入力のグラフにおいて頂点 i から頂点 j に到達可能であれば，result[i][j] に 1 が格納されていて，到達可能でなければ 0 が格納されていなければならない．入力のグラフが連結であれば，すべての要素が 1 になる．

この判定はワーシャル (Warshall) のアルゴリズムによって実現できるが，このアルゴリズムの主な手順は，次の性質を満たすような int 型の 3 次元配列 reach を完成させることである (ただし，頂点数 n に対して，$0 \leq k \leq n$，$0 \leq i < n$，$0 \leq j < n$)[6].

$$\texttt{reach}[k][i][j] = \begin{cases} 1 & \cdots\text{頂点番号が } k \text{ 未満の頂点だけを通って，頂点 } i \text{ から頂点 } j \text{ に到達できるとき} \\ 0 & \cdots\text{それ以外の場合} \end{cases}$$

「頂点番号が k 未満の頂点だけを通って，頂点 i から頂点 j に到達できるとき」というのは，途中で通る点がすべて k 未満でなければならないだけで，i や j が k 未満である必要はない．たとえば，右のグラフにおいて頂点 4 から頂点 5 へたどる道はいくつかあるが，4，1，0，5 のように頂点番号が 2 未満の点だけを通ってたどり着く道があるので，reach[2][4][5] は 1 である．一方，頂点番号が 1 未満の頂点だけを通ってたどり着く道はないので，reach[1][4][5] は 0 である．また，頂点 2 から頂点 6 へはそもそもたどり着く道がないので，どんな k に対しても reach[k][2][6] は 0 である．このような配列 reach を作ることさえできれば，あとは result[i][j] に reach[8][i][j] の値を入れるだけである (頂点番号の最大値が 7 なので,「頂点番号が 8 未満の頂点を通って頂点 i から頂点 j にたどり着く」というのは,「このグラフにおいて頂点 i から頂点 j にたどり着く」ことと同義である).



では，reach はどのように計算していけばいいだろうか.「頂点番号が k 未満の頂点だけを通って，頂点 i から頂点 j にたどり着く」というのは次のどちらかであることに注目しよう．

- **頂点 $k-1$ を通らない場合**
  頂点番号が $k-1$ 未満の頂点だけを通って，頂点 i から頂点 j にたどり着く
- **頂点 $k-1$ を通る場合**
  頂点番号が $k-1$ 未満の頂点だけを通って，頂点 i から頂点 $k-1$ にたどり着き，
  頂点番号が $k-1$ 未満の頂点だけを通って，頂点 $k-1$ から頂点 j にたどり着く．

[6] i や j は頂点番号なので n 未満であるが，k については n 以下であることに注意！

つまり，reach[$k-1$][x][y] の要素が任意の x，y についてわかっていれば，reach[k][i][j] の要素がわかるということになる．したがって，k が小さい方から順番に，すべての i，j について reach[k][i][j] を確定させていき，$k=n$ (n は頂点数) まで計算すれば目的の答えが得られる．なお，$k=0$ のときは，頂点番号が 0 未満の頂点 (そんな頂点は存在しない！) を通って i から j にたどり着くとき，すなわち，頂点 i と頂点 j が元のグラフにおいて隣接しているか，あるいは $i=j$ (つまり自分自身) のときに，reach[k][i][j] の値が 1 になる．

作成すべきプログラムの形式は後に示す main 関数を含むものとし，関数 warshall の定義を適切に埋めることによりプログラムを完成させよ．標準入力から与えられる入力は問題 1 と同様に路線図データを表す無向グラフである．標準出力には，到達可能性を表す行列 result を問題 3 と同じ形式で出力する．

入力例

```
3
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
```

出力例

```
111
111
111
```

入力例

```
4
0:1:1012:2.000 ← 頂点番号 2 を含む辺がないので
0:3:1012:3.000 ← 頂点 2 は孤立した点となる．
1:3:1008:2.500 ←
```

出力例

```
1101
1101
0010
1101
```

入力例

```
8
0:2:1011:0.120
0:5:1012:0.300
1:3:1013:0.260
2:4:1011:0.080
2:6:1014:0.080
3:7:1013:0.130
4:6:1011:0.170
5:6:1012:0.080
```

出力例

```
10101110
01010001
10101110
01010001
10101110
10101110
10101110
01010001
```

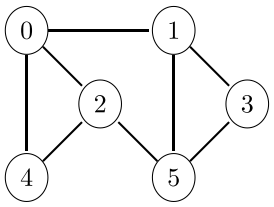
作成すべきプログラムは以下の形式とする．※印を含むコメント部分を適切に書き換えればよいが，必要に応じて補助関数を定義してもよい．

```c
#include<stdio.h>
#include<stdlib.h>
char buf[256];

void warshall(int n, int adjmat[n][n], int result[n][n]) {
  /* ※ここを適切なプログラムで埋める */
}

int main() {
  int ekisu;
  scanf("%d␣", &ekisu);
  int adjmat[ekisu][ekisu], result[ekisu][ekisu];
  /* ※ここは問題 3 と同じ (ただし，出力部分は除く) */
  warshall(ekisu, adjmat, result);
  /* ※ここで result を適切に出力 */
  return 0;
}
```

メモ `tonari.txt` のように大きなデータを入力とする場合には，0 か 1 を要素とする 2 次元配列を使って隣接行列を表現するとメモリを無駄に消費してしまう．`tonari.txt` には 1 万以上の駅 (頂点) が含まれているので，1 万 × 1 万 で 1 億個の要素を含む 2 次元配列を作る必要がある (ワーシャルのアルゴリズムで 3 次元配列を用いるなら要素は 1 兆個！[7])．これはあまり現実的ではないので，実際には圧縮した形で隣接行列を表すことがある．なお，`tonari.txt` のように辺の数が頂点の数に比べてそれほど多くない場合[8] には，隣接リストのような表現の方が望ましい．

2.3 発展問題

右の連結グラフでは，どの 2 つの頂点を選んでも 3 つの辺を使って繋ぐことができるが，2 つの辺だけでは繋ぐことのできない頂点がある (頂点 3 と頂点 4)．このとき，このグラフの「直径」は 3 であるという．一般に，連結グラフ G について，2 つの頂点を結ぶ最短の道の長さ (通る辺の数) を考えるとき，この長さの最大値をグラフ G の**直径**という．最後の問題は，与えられたグラフの直径を求める問題である．



問題 5

この問題の目的は，路線図データを表すグラフの直径を求める関数 `diameter` を定義することである．この問題においては，グラフを表現する方法として隣接リストと隣接行列のどちらを選んでもよい．隣接リスト表現を用いる場合は，関数 `diameter` の型は

```
int diameter(int n, struct node *adjlist[n]);
```

として定義し，隣接行列表現を用いる場合は，

```
int diameter(int n, int adjmat[n][n]);
```

として定義する．この関数は，「`n` 個の駅 (頂点) を含む路線図データを表すグラフ (`adjlist` または `adjmat`) を引数に取り，そのグラフの直径を返す関数」である．与えられるグラフは連結グラフであると仮定してよい．

作成すべきプログラムの形式は指定しないが，指定された型 (上のいずれか) の関数 `diameter` の定義を含むものとし，この関数を用いて直径を計算することが求められる．標準入力から与えられる入力は問題 1 と同様に路線図データを表す連結無向グラフであり，標準出力にはその直径を出力する．プログラムの記述については，隣接リスト表現を用いる場合は問題 1 を，隣接行列表現を用いる場合は問題 3 を参考にするとよい．

入力例

```
3
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
```

出力例

```
1
```

入力例

```
4
0:1:1012:2.000
0:2:1023:4.000
1:2:1012:3.000
1:3:1008:2.500
2:3:1023:1.500
```

出力例

```
2
```

[7] ただし，少しアルゴリズムを工夫すれば `reach` は 2 次元で済む．
[8] 頂点の数が 10555 で辺の数が 12839.

入力例

```
6
0:1:1011:0.120
0:2:1012:0.300
0:3:1013:0.260
1:2:1014:0.220
2:4:1012:0.080
3:4:1015:0.130
3:5:1013:0.170
4:5:1012:0.080
```

出力例

```
3
```