4.5 節末問題 4.5 の解答

問題 4.5.1

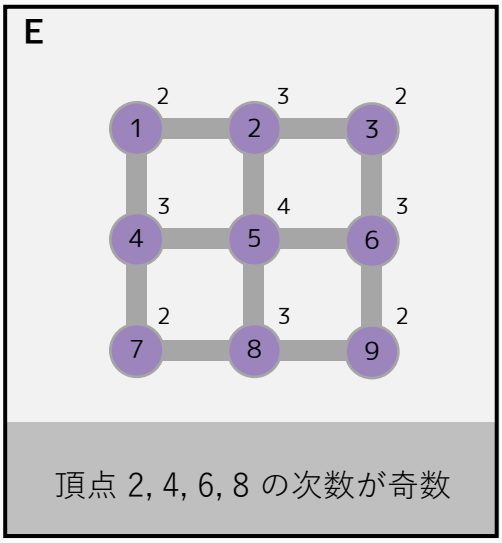
答えは以下の通りです。分からない人は、4.5.2 項・4.5.4 項を確認しましょう。

グラフの番号	グラフの種類	次数最大の頂点
A	重みなし無向グラフ	頂点 1（次数=3）
B	重み付き無向グラフ	頂点 5（次数=4）
C	重みなし有向グラフ	頂点 3（出次数=2）
D	重み付き有向グラフ	頂点 2（出次数=3）

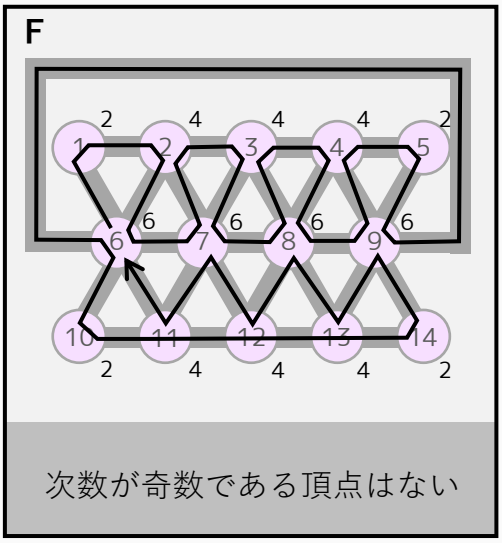
問題 4.5.2

まず、グラフ E については次数が奇数である頂点が存在するため、すべての頂点を一度ずつ通って戻ってくる経路が存在しません。また、グラフ F はすべての頂点の次数が偶数であり、下図のような経路が存在します。

分からない人は、オイラーグラフ（→**4.5.2項**）を確認しましょう。なお、下図の頂点に付いている番号は、その頂点の次数です。



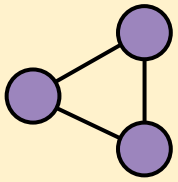
頂点 2, 4, 6, 8 の次数が奇数



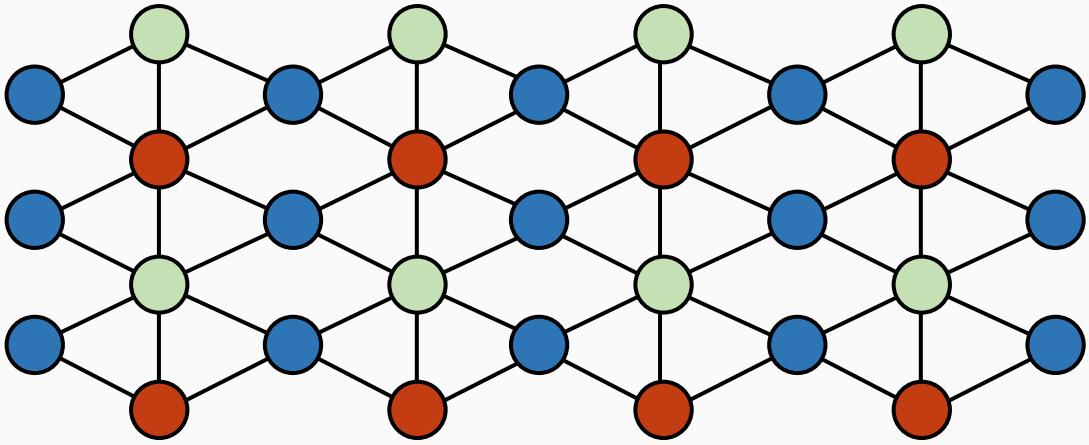
次数が奇数である頂点はない

問題 4.5.3

まず、2 色で塗り分けられない理由は以下の通りです。

> グラフには右図のような三角形が含まれており、この三角形だけを考えても 2 色で塗ることは明らかに不可能だから。
>
> 

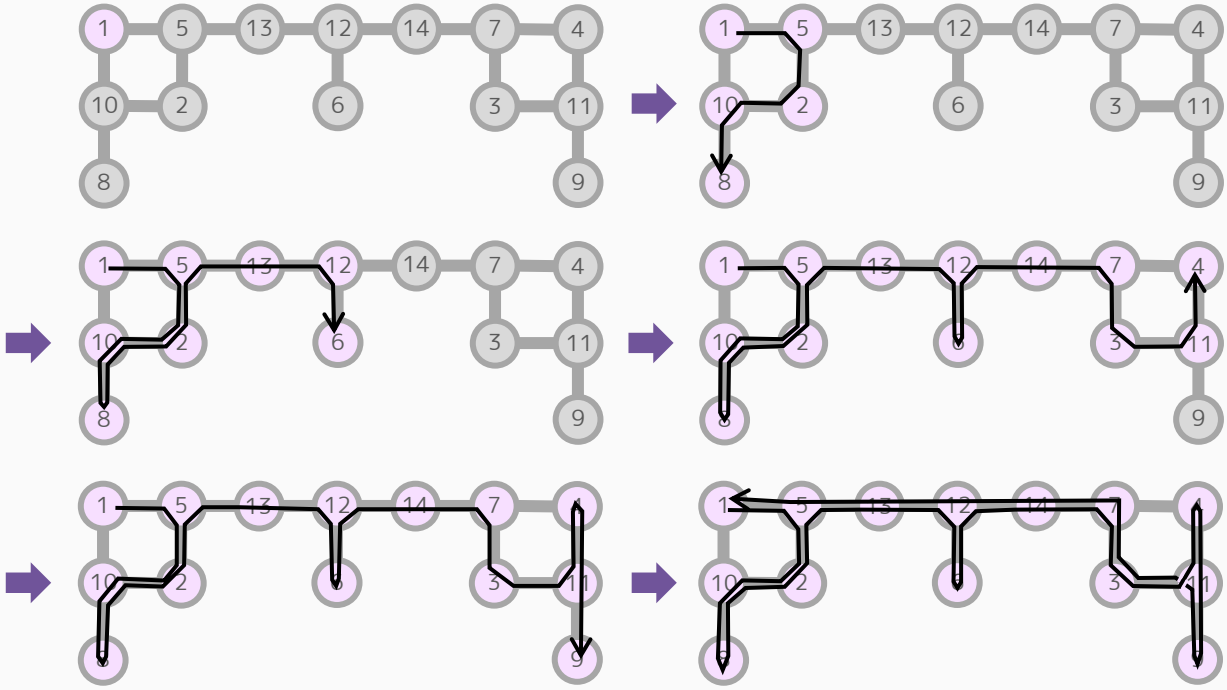
また、以下のように塗ると、3 色で塗り分けることができます。



問題 4.5.4

訪問順序は順に $1, 5, 2, 10, 8, 13, 12, 6, 14, 7, 3, 11, 4, 9$ です。

以下の図は、深さ優先探索の具体的な動きを示しています。



問題 4.5.5

この問題は、隣接リスト形式（→**4.5.5項**）の理解を問う問題です。たとえば以下のように実装すると、正解が得られます。

なお、変数 `cnt` は頂点 i に隣接する頂点のうち、番号が i 未満のものの個数を示しています。また、`G[i].size()` はリスト `G[i]` の要素数です。

```cpp
#include <iostream>
#include <vector>
using namespace std;

int N, M;
int A[100009], B[100009];
vector<int> G[100009];

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 答えを求める
    int Answer = 0;
    for (int i = 1; i <= N; i++) {
        int cnt = 0;
        for (int j = 0; j < G[i].size(); j++) {
            // G[i][j] は頂点 i に隣接している頂点のうち j 個目
            if (G[i][j] < i) cnt += 1;
        }
        // 自分自身より小さい隣接頂点が 1 つであれば Answer に 1 を加算する
        if (cnt == 1) Answer += 1;
    }

    // 出力
    cout << Answer << endl;
    return 0;
}
```

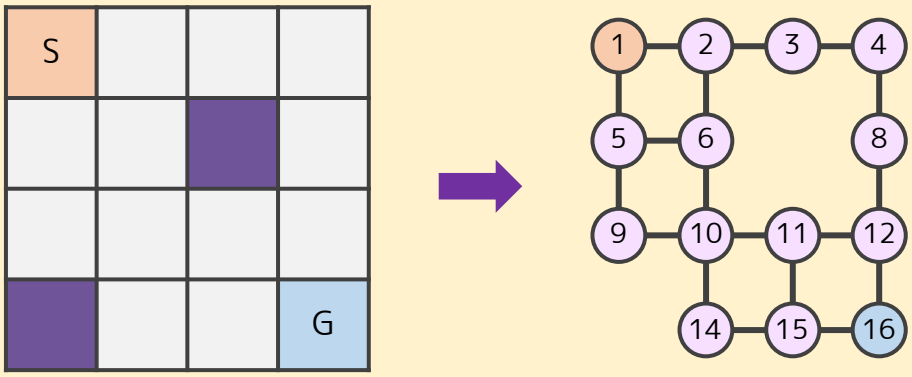
※ Python などのソースコードは chap4-5.md をご覧ください。

問題 4.5.6

以下のように頂点番号を設定すると、最短経路問題（→**4.5.7項**）に帰着することができます。頂点数は HW、辺の数は $4HW$ 以下なので、1 秒以内で実行が終わります。

上から i 行目、左から j マス目の頂点番号を $(i-1)\times W+j$ に設定する。

$H=4, W=4$ の場合の具体例は以下の図の通り。



したがって、以下のような実装で正解を出すことができます。なお、各マスに頂点番号を振るように、データを数値で表現して識別することをハッシュといいます。資格試験などでも頻出なので、興味のある人はインターネットなどで調べてみてください。

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 入力
int H, W;
int sx, sy, start; // スタートの座標 (sx, sy) と頂点番号 sx*W+sy
int gx, gy, goal;  // ゴールの座標 (gx, gy) と頂点番号 gx*W+gy
char c[59][59];

// グラフ・最短経路
int dist[2509];
vector<int> G[2509];

int main() {
    // 入力
    cin >> H >> W;
    cin >> sx >> sy; start = sx * W + sy;
    cin >> gx >> gy; goal = gx * W + gy;
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) cin >> c[i][j];
    }
```

```
    // 横方向の辺 [(i, j) - (i, j+1)] をグラフに追加
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W - 1; j++) {
            int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
            int idx2 = i * W + (j+1); // 頂点 (i, j+1) の頂点番号
            if (c[i][j] == '.' && c[i][j+1] == '.'){
                G[idx1].push_back(idx2);
                G[idx2].push_back(idx1);
            }
        }
    }

    // 縦方向の辺 [(i, j) - (i+1, j)] をグラフに追加
    for (int i = 1; i <= H - 1; i++) {
        for (int j = 1; j <= W; j++) {
            int idx1 = i * W + j; // 頂点 (i, j) の頂点番号
            int idx2 = (i+1) * W + j; // 頂点 (i+1, j) の頂点番号
            if (c[i][j] == '.' && c[i+1][j] == '.'){
                G[idx1].push_back(idx2);
                G[idx2].push_back(idx1);
            }
        }
    }

    // 以降は（頂点数などを除き）コード 4.5.3 と同じ
    // 幅優先探索の初期化（dist[i]=-1 のとき、未到達の白色頂点である）
    for (int i = 1; i <= H * W; i++) dist[i] = -1;
    queue<int> Q; // キュー Q を定義する
    Q.push(start); dist[start] = 0; // Q に 1 を追加（操作 1）

    // 幅優先探索
    while (!Q.empty()) {
        int pos = Q.front(); // Q の先頭を調べる（操作 2）
        Q.pop(); // Q の先頭を取り出す（操作 3）
        for (int i = 0; i < (int)G[pos].size(); i++) {
            int nex = G[pos][i];
            if (dist[nex] == -1) {
                dist[nex] = dist[pos] + 1;
                Q.push(nex); // Q に nex を追加（操作 1）
            }
        }
    }

    // 答えを出力
    cout << dist[goal] << endl;
    return 0;
}
```

※ Python などのソースコードは chap4-5.md をご覧ください。

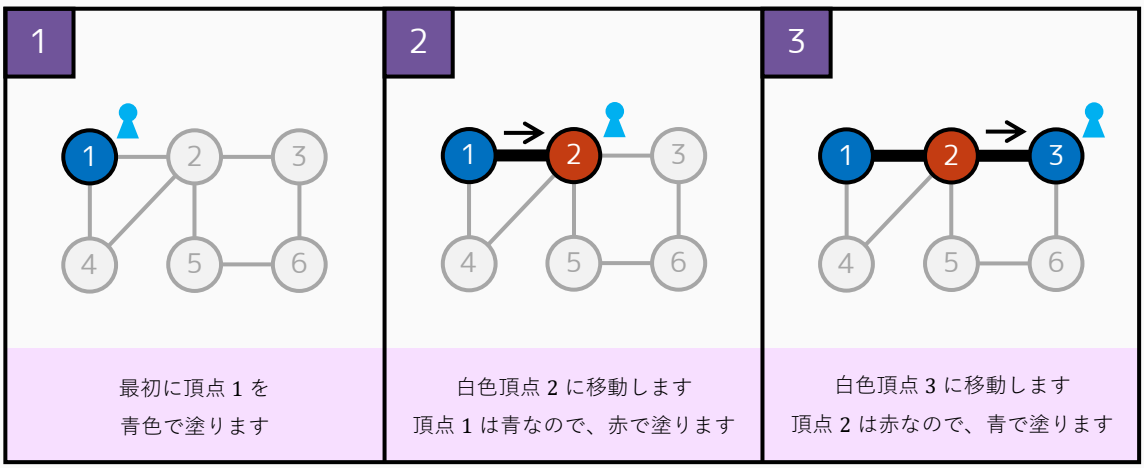
問題 4.5.7

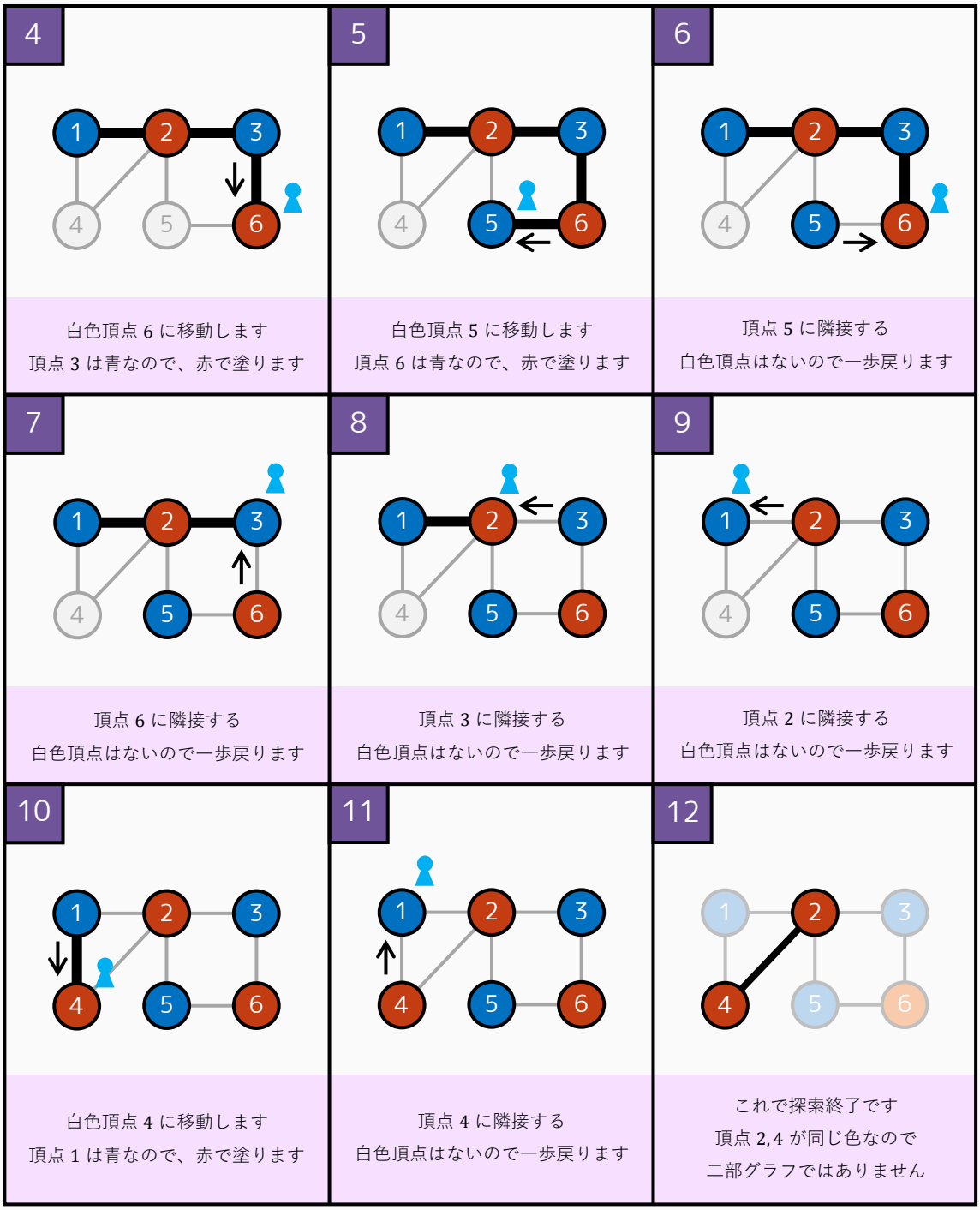
まず、グラフが連結である場合、1 つの頂点の色さえ決まれば、グラフを青と赤の 2 色で塗る方法は高々 1 つに定まります（下図参照）。なぜなら、青の隣に赤、赤の隣に青、… と塗っていく必要があるからです。

このため、以下のように深さ優先探索をすると、グラフを青と赤で塗る方法を 1 つ構成することができます。なお、赤文字で示した部分は、4.5.6 項で述べた連結判定のアルゴリズムと異なる部分です。

1. すべての頂点を白色で塗る。
2. 一番最初に頂点 1 を訪問し、頂点 1 を**青色**で塗る。
3. その後、以下の操作を繰り返す。
   A) 隣接する白色頂点がない：一歩戻る
   B) 隣接する白色頂点がある：これらの中で番号が最小の頂点を訪問する。新たに頂点を訪問する際には、**前の頂点と異なる色**で塗る。
4. 最終的に、**どの隣り合う 2 頂点も異なる色で塗られていたら、グラフは二部グラフである**。

このアルゴリズムを具体的なグラフに適用させると、以下のようになります。太線は移動経路の跡を示しています。





このアルゴリズムを実装すると、次ページのようになります。なお、プログラミングでは実際に頂点を白・青・赤で塗ることはできないので、

- `color[i]=0` のとき：頂点 i が白色
- `color[i]=1` のとき：頂点 i が青色
- `color[i]=2` のとき：頂点 i が赤色

としています。また、グラフが連結でないケースが入力される可能性があるので、手順 2. に相当する操作を各連結成分に対して行わなければならないことに注意してください。（プログラムの「深さ優先探索」部分参照）

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int N, M, A[200009], B[200009]; // N, M ≦ 200000 なので配列の大きさは 200009 にしておく
vector<int> G[200009];
int color[200009];

void dfs(int pos) {
    for (int i : G[pos]) { // 範囲 for 文
        if (color[i] == 0) {
            // color[pos]=1 のとき 2、color[pos]=2 のとき 1
            color[i] = 3 - color[pos];
            dfs(i);
        }
    }
}

int main() {
    // 入力
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i];
        G[A[i]].push_back(B[i]);
        G[B[i]].push_back(A[i]);
    }

    // 深さ優先探索
    for (int i = 1; i <= N; i++) color[i] = 0;
    for (int i = 1; i <= N; i++) {
        if (color[i] == 0) {
            // 頂点 i は白である（まだ探索されていない連結成分である）
            color[i] = 1;
            dfs(i);
        }
    }

    // 二部グラフかどうかの判定
    bool Answer = true;
    for (int i = 1; i <= M; i++) {
        if (color[A[i]] == color[B[i]]) Answer = false;
    }
    if (Answer == true) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

※ Python などのソースコードは chap4-5.md をご覧ください。

問題 4.5.8

注意：この問題は、4.5.8 項「その他の代表的なグラフアルゴリズム」で紹介されたダイクストラ法を使います。初学者は解けなくて当然ですので、ご安心ください。

一般に、整数は「10 倍して 1 ～ 9 の値を足す」という操作の繰り返しによって作ることができます。たとえば整数 8691 は、

- 整数 0 から始める
- 10 倍して 8 を足す（$0 \times 10 + 8 = 8$ になる）
- 10 倍して 6 を足す（$8 \times 10 + 6 = 86$ になる）
- 10 倍して 9 を足す（$86 \times 10 + 9 = 869$ になる）
- 10 倍して 1 を足す（$869 \times 10 + 1 = 8691$ になる）

そこで、以下のような重み付き有向グラフを考えてみましょう。頂点 i $(0 \leq i < K)$ は「K で割って i 余る整数」を指します。

頂点について

- K 個の頂点を用意する。
- 頂点番号はそれぞれ $0, 1, 2, 3, \ldots, K-1$ とする。

辺について

各頂点 i $(0 \leq i < K)$ について、以下の 10 個の辺を追加する。

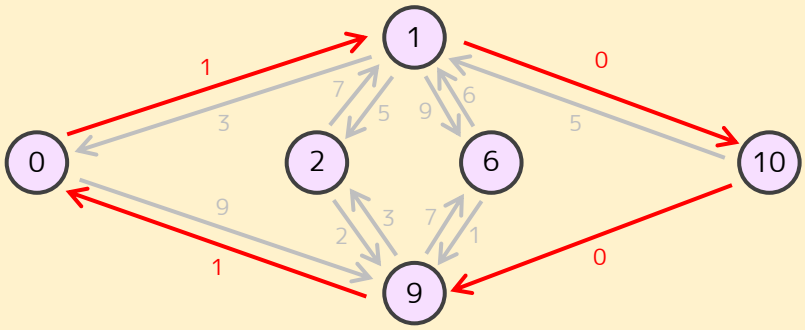
- 頂点 i から頂点 $(10i + 0) \bmod K$ に向かう、重み 0 の辺※
- 頂点 i から頂点 $(10i + 1) \bmod K$ に向かう、重み 1 の辺
- 頂点 i から頂点 $(10i + 2) \bmod K$ に向かう、重み 2 の辺
- 頂点 i から頂点 $(10i + 3) \bmod K$ に向かう、重み 3 の辺
- 頂点 i から頂点 $(10i + 4) \bmod K$ に向かう、重み 4 の辺
- 頂点 i から頂点 $(10i + 5) \bmod K$ に向かう、重み 5 の辺
- 頂点 i から頂点 $(10i + 6) \bmod K$ に向かう、重み 6 の辺
- 頂点 i から頂点 $(10i + 7) \bmod K$ に向かう、重み 7 の辺
- 頂点 i から頂点 $(10i + 8) \bmod K$ に向かう、重み 8 の辺
- 頂点 i から頂点 $(10i + 9) \bmod K$ に向かう、重み 9 の辺

※ 実装の都合上、頂点 0 → 0 の辺は例外的に追加しません。

ここで、一つの辺を通ることが 1 回の操作に対応します。たとえば、$K = 13$ で 86 を 869 にする操作は、頂点 $8 \to 11$ に向かう重み 9 の辺を通ることに対応します。（$86 \bmod 13 = 8,\ 869 \bmod 13 = 11$）

したがって、頂点 0 から頂点 $i\ (\geq 1)$ までの最短経路長が、K で割って i 余る数における各桁の和の最小値となります。

同様に、頂点 0 から頂点 0 へ戻る最短経路の長さが、K の倍数における各桁の和の最小値になります。具体例は以下の通りです。

> たとえば $K = 13$ の場合、各桁の和の最小値は 2（1001 という数）ですが、これは $0 \to 1 \to 10 \to 9 \to 0$ という経路に対応します。（補足： $1 \bmod 13 = 1$、$10 \bmod 13 = 10$、$100 \bmod 13 = 9$、$1001 \bmod 13 = 0$）
>
> この経路は、前ページの方法で構成したグラフにおける最短経路になっています。（見やすさの都合上、一部の頂点と辺を省略しています）
>
> 
>

よって、ダイクストラ法（→**4.5.8項**）を使って重み付きグラフの最短経路長を求めれば、正しい答えを出すことができます。

ただし、自然に実装すると頂点 0 から 0 までの最短経路長は 0 になってしまうので、少し工夫を行う必要があります。詳しくは、次ページの実装例を参考にしてください。

※ より丁寧な公式解説もご覧ください。（リンクは chap4-5.md にあります）

```cpp
#include <bits/stdc++.h>
using namespace std;

int K, dist[100009];
bool used[100009];
vector<pair<int, int>> G[100009];
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> Q;

// ダイクストラ法
void dijkstra() {
    // 配列の初期化など
    for (int i = 0; i < K; i++) dist[i] = (1 << 30);
    for (int i = 0; i < K; i++) used[i] = false;
    Q.push(make_pair(0, 0)); // ここで dist[0] = 0 にはしないことに注意！

    // キューの更新
    while (!Q.empty()) {
        int pos = Q.top().second; Q.pop();
        if (used[pos] == true) continue;
        used[pos] = true;
        for (pair<int, int> i : G[pos]) {
            int to = i.first, cost = dist[pos] + i.second;
            if (pos == 0) cost = i.second; // 頂点 0 の場合は例外
            if (dist[to] > cost) {
                dist[to] = cost;
                Q.push(make_pair(dist[to], to));
            }
        }
    }
}

int main() {
    // 入力
    cin >> K;

    // グラフの辺を追加
    for (int i = 0; i < K; i++) {
        for (int j = 0; j < 10; j++) {
            if (i == 0 && j == 0) continue;
            G[i].push_back(make_pair((i * 10 + j) % K, j));
        }
    }

    // ダイクストラ法・出力
    dijkstra();
    cout << dist[0] << endl;
    return 0;
}
```

※ Python などのソースコードは chap4-5.md をご覧ください。