6.6 最終確認問題 26-30 の解答

問題 26

実験開始から n 秒後の物質 A, B, C の量をそれぞれ a_n, b_n, c_n とします。a_n, b_n, c_n は、以下の漸化式によって決まります。

- $a_{n+1} = (1-X)a_n + Yb_n$
- $b_{n+1} = (1-Y)b_n + Zc_n$
- $c_{n+1} = (1-Z)c_n + Xa_n$

この (a_n, b_n, c_n) と $(a_{n+1}, b_{n+1}, c_{n+1})$ の関係は、行列（→**4.7 節**）を使って以下のように表せます。

$$\begin{bmatrix} a_{n+1} \\ b_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} 1-X & Y & 0 \\ 0 & 1-Y & Z \\ X & 0 & 1-Z \end{bmatrix} \begin{bmatrix} a_n \\ b_n \\ c_n \end{bmatrix}$$

この漸化式を繰り返し適用すると、a_T, b_T, c_T は以下の式で表せます。（実験開始時点での物質 A, B, C の量はそれぞれ 1 グラムずつです）

$$\begin{bmatrix} a_T \\ b_T \\ c_T \end{bmatrix} = \begin{bmatrix} 1-X & Y & 0 \\ 0 & 1-Y & Z \\ X & 0 & 1-Z \end{bmatrix}^T \begin{bmatrix} a_0 \\ b_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1-X & Y & 0 \\ 0 & 1-Y & Z \\ X & 0 & 1-Z \end{bmatrix}^T \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

3 × 3 行列の累乗は、繰り返し二乗法（→**4.6.7 項**）を行列に適用することで、計算量 $O(\log T)$ で計算できます。これで、この問題の答えが求まります。

この解法を C++ で実装すると、以下のようになります。

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

struct matrix {
	double x[3][3] = {
		{ 0.0, 0.0, 0.0 },
		{ 0.0, 0.0, 0.0 },
		{ 0.0, 0.0, 0.0 }
	};
};
```

```
// 行列の掛け算
matrix multiplication(matrix A, matrix B) {
	matrix C;
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			for (int k = 0; k < 3; k++) {
				C.x[i][j] += A.x[i][k] * B.x[k][j];
			}
		}
	}
	return C;
}

// 行列の累乗
matrix power(matrix A, int n) {
	matrix P = A, Q;
	bool flag = false;
	for (int i = 0; i < 30; i++) {
		if ((n & (1 << i)) != 0) {
			if (flag == false) { Q = P; flag = true; }
			else Q = multiplication(Q, P);
		}
		P = multiplication(P, P);
	}
	return Q;
}

int main() {
	int Q, T; double X, Y, Z;
	cin >> Q;
	for (int t = 1; t <= Q; t++) {
		// 入力 → 行列 A の構築
		cin >> X >> Y >> Z >> T;
		matrix A;
		A.x[0][0] = 1.0 - X; A.x[2][0] = X;
		A.x[1][1] = 1.0 - Y; A.x[0][1] = Y;
		A.x[2][2] = 1.0 - Z; A.x[1][2] = Z;

		// 行列累乗の計算 → 答えを出力
		matrix B = power(A, T);
		double answerA = B.x[0][0] + B.x[0][1] + B.x[0][2];
		double answerB = B.x[1][0] + B.x[1][1] + B.x[1][2];
		double answerC = B.x[2][0] + B.x[2][1] + B.x[2][2];
		printf("%.12lf %.12lf %.12lf¥n", answerA, answerB, answerC);
	}
	return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

問題 27

書かれている整数の差が k 以上になるボールの選び方の数を求める問題を「問題 k」ということにします。問題 k は、以下のように小問題に分解（→**5.6 節**）できます。

- （差が k 以上になるように）1 個のボールを選ぶ方法は何通り？
- 差が k 以上になるように 2 個のボールを選ぶ方法は何通り？
- 差が k 以上になるように 3 個のボールを選ぶ方法は何通り？
- （中略）
- 差が k 以上になるように $[N/k]$ 個のボールを選ぶ方法は何通り？

$[N/k]$ 個で打ち切ってよい理由は、どう上手く選んでも最大で $[N/k]$ 個のボールしか選べないからです。このように、問題 k は $[N/k]$ 個の「より解きやすそうな小問題」に分解できました。これをもとに問題 $1, 2, 3, \dots, N$ を解くと、全体で

$$\left[\frac{N}{1}\right] + \left[\frac{N}{2}\right] + \left[\frac{N}{3}\right] + \cdots + \left[\frac{N}{N}\right]$$

個の小問題を解くことになります。これは $O(N \log N)$ 個であり、逆数 $1/x$ の和が $O(\log N)$ になる（→**4.4.4 項**）性質から分かります。

では、各小問題がどうやって解けるかを考えてみましょう。

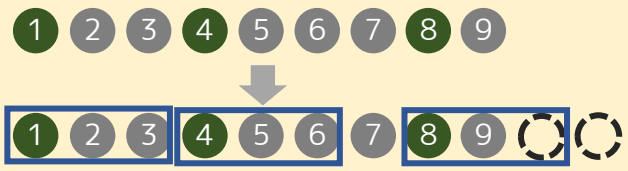
“小問題” はどんな問題か？

$1,2, \dots, N$ が書かれたボールから、どの 2 つの数も m 以上離れるように、k 個のボールを選ぶ方法は何通りか？

小問題の答え

この小問題の答えは $_{N-(k-1)(m-1)}C_m$ 通り です。

理由は、ボール x を選ぶときに以下の図のようにボール $x, x+1, \dots, x+m-1$ を囲むことを考えると、ボール $1, 2, \dots, N+m-1$ から k 個の互いに重ならない「長さ m の囲い」を配置する問題になるからです。これは、単体のボール $N+m-1-km$ 個と囲い k 個を並び替える問題に読み替えられます。



$N = 9, m = 3, k = 3$ の場合の例
2 個の単体のボールと 3 個の囲いを並び替えるので、 $_5C_3 = 10$ 通り

二項係数は、階乗を前計算する方法（→**コード 4.6.6**）を使うと、$M = 10^9 + 7$ として $O(\log M)$ 時間で求められます。先ほど述べたように、$O(N \log N)$ 個の小問題を解くことになるので、この問題は全体計算量 $O(N \log N \log M)$ で解けることになります。

この解法を C++ で実装すると、以下のようになります。

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

const long long mod = 1000000007;
int N; long long fact[100009];

long long modpow(long long a, long long b, long long m) {
	// 繰り返し二乗法（p は a^1, a^2, a^4, a^8, ... といった値をとる）
	long long p = a, Answer = 1;
	for (int i = 0; i < 30; i++) {
		if ((b & (1 << i)) != 0) { Answer *= p; Answer %= m; }
		p *= p; p %= m;
	}
	return Answer;
}
long long Division(long long a, long long b, long long m) {
	return (a * modpow(b, m - 2, m)) % m;
}
long long ncr(int n, int r) {
	// ncr は n! を r! × (n-r)! で割った値
	return Division(fact[n], fact[r] * fact[n - r] % mod, mod);
}

int main() {
	// 配列の初期化（fact[i] は i の階乗を 1000000007 で割った余り）
	fact[0] = 1;
	for (int i = 1; i <= 100000; i++) {
		fact[i] = 1LL * i * fact[i - 1] % mod;
	}

	// 入力 → 答えを計算して出力
	cin >> N;
	for (int i = 1; i <= N; i++) {
		int answer = 0;
		for (int j = 1; j <= (N + i - 1) / i; j++) {
			answer += ncr(N - (i - 1) * (j - 1), j);
			answer %= mod;
		}
		cout << answer << endl;
	}
	return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

問題 28

この問題は、**5.7.5 項**の足し算ピラミッドの問題と似ていますが、色に対して規則が定められているので取り掛かりにくそうに見えます。ここで、色を次のように ID に直して考えると、後々都合がよいです。

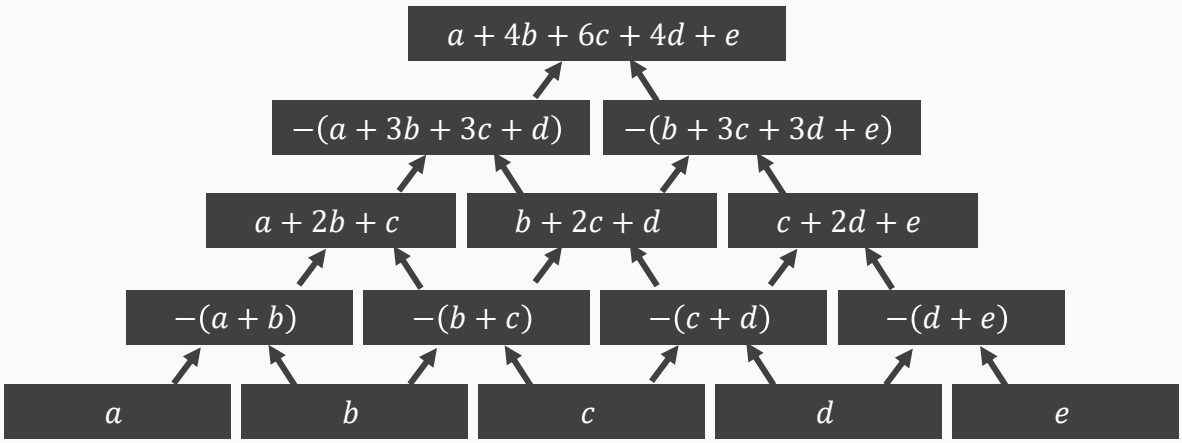
- 「青」→ **0**、「白」→ **1**、「赤」→ **2**

直下にある 2 つのブロックの色の ID を x, y とすると、上のブロックの色は右表のようになります。

x \ y	0	1	2
0	0	2	1
1	2	1	0
2	1	0	2

つまり、直下のブロックの色の ID を x, y とすると、上のブロックの色の ID は $-(x+y) \bmod 3$ と計算できることになります※。

これを $N = 5$ 段のピラミッドの場合でも考えてみましょう。下のブロックの色を a, b, c, d, e とすると、各ブロックの色は下図のようになります（すべて mod 3 で考えています）。



一般の場合も同様に、一番上の色の ID は

$$(-1)^{N-1} \cdot (c_1 \cdot_{N-1} C_0 + c_2 \cdot_{N-1} C_1 + \cdots + c_N \cdot_{N-1} C_{N-1}) \bmod 3$$

で求まることになります。

では、$nCr \bmod 3$ はどうやって求めるのでしょうか？3 は n, r に比べて小さいので、逆元を使った方法（→**4.6.8 項**）は通用しません。

n を 3 進法で表すと $n_{d-1}n_{d-2}\ldots n_1 n_0$、$k$ を 3 進法で表すと $k_{d-1}k_{d-2}\ldots k_1 k_0$ と表されるとします。このとき、

ここで、**リュカの定理**を使いましょう。

> n を 3 進法で表すと $n_{d-1}n_{d-2}\ldots n_1n_0$、$r$ を 3 進法で表すと $r_{d-1}r_{d-2}\ldots r_1r_0$ と表されるとします。このとき、$nCr \bmod 3$ は
>
> $$(n_{d-1}Cr_{d-1} \times n_{d-2}Cr_{d-2} \times \cdots \times n_1Cr_1 \times n_0Cr_0) \bmod 3$$
>
> と計算されます。ただし、式の途中で $n_i < r_i$ となる場合、$nCr \bmod 3 = 0$ となります。一般の $\bmod M$ に対しても同様の定理が成り立ちます。

nCr は計算量 $O(\log n)$ で計算でき、この問題は全体計算量 $O(N \log N)$ で解けました。

この解法を C++ で実装すると、以下のようになります。なお、nCr の計算は、再帰関数を用いて実装されています。（**2.1.9 項**の方法で 3 進法に変換することも可能です）

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

// リュカの定理で ncr mod 3 を計算
int ncr(int x, int y) {
    if (x < 3 && y < 3) {
        if (x < y) return 0;
        if (x == 2 && y == 1) return 2;
        return 1;
    }
    return ncr(x / 3, y / 3) * ncr(x % 3, y % 3) % 3;
}

int main() {
    // 入力
    int N; string C;
    cin >> N >> C;

    // 答えを求める
    int answer = 0;
    for (int i = 0; i < N; i++) {
        int code;
        if (C[i] == 'B') code = 0;
        if (C[i] == 'W') code = 1;
        if (C[i] == 'R') code = 2;
        answer += code * ncr(N - 1, i);
        answer %= 3;
    }

    // 答えを (-1)^(N-1) で掛ける
    if (N % 2 == 0) {
        answer = (3 - answer) % 3;
    }
```

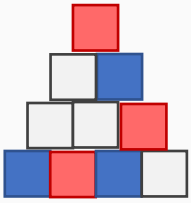
```
    // 答えを出力（"BWR" の answer 文字目）
    cout << "BWR"[answer] << endl;

    return 0;
}
```

Python・JAVA・C のソースコードは、GitHub の chap6-26_30.md をご覧ください。

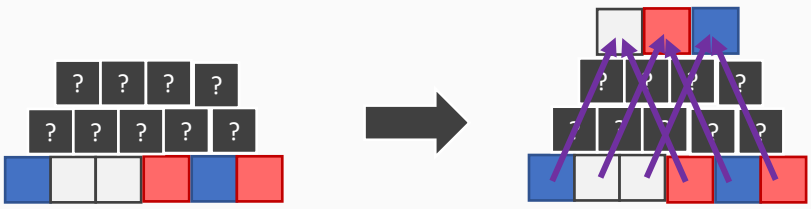
問題 28 ― 別解

この問題には別解があります。「規則性を考える」（→**5.2 節**）を使った解法です。

$N = 4$ の場合を考えてみましょう。実は、どのような場合でも、一番上のブロックの色は、一番左のブロックと一番右のブロックだけで決まります。2, 3 番目のブロックは関係ありません。右図の例では、一番左のブロックは青、一番右のブロックは白なので、一番上のブロックは赤になります。



$N = 10$ の場合も同様で、一番左のブロックと一番右のブロックだけで決まり、2, 3, 4, 5, 6, 7, 8, 9 番目のブロックは関係ありません。$N = 28, 82, 244, 730, 2188, 6562, \ldots$ の場合も同様です。一般に、影響範囲の一番左のブロックと一番右のブロックを見るだけで、3^k 段上のブロックの配置が求まることになります。



これを巧みに利用すれば、$O(\log N)$ 段のブロックしか計算する必要がなくなります。例えば $N = 23$ の場合、3 進法を利用すると $22 = 9 + 9 + 3 + 1$ なので、1 段目のブロックを入力してから、10 段目を求め、19 段目を求め、22 段目を求め、23 段目を求める、といった流れになります。各ステップには計算量 $O(N)$ しかかからず、全体で $O(\log N)$ ステップあるので、全体計算量 $O(N \log N)$ でこの問題が解けます。

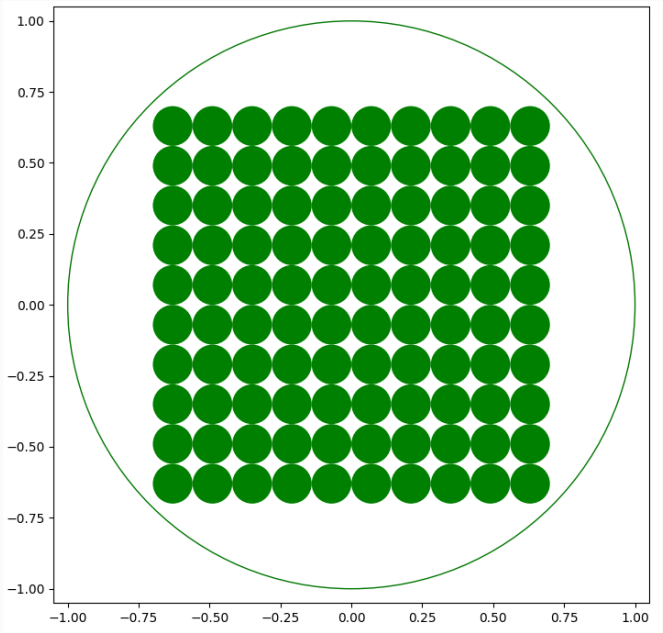
実装コードは、紙面の都合上省略します。

問題 29

この問題は、半径 1 の円にできるだけ半径の大きな 100 個の小さな円を敷き詰める問題です。半径が大きい配置ほど高い得点が得られる形式なので、この問題に対しては様々な解法が考えられます。

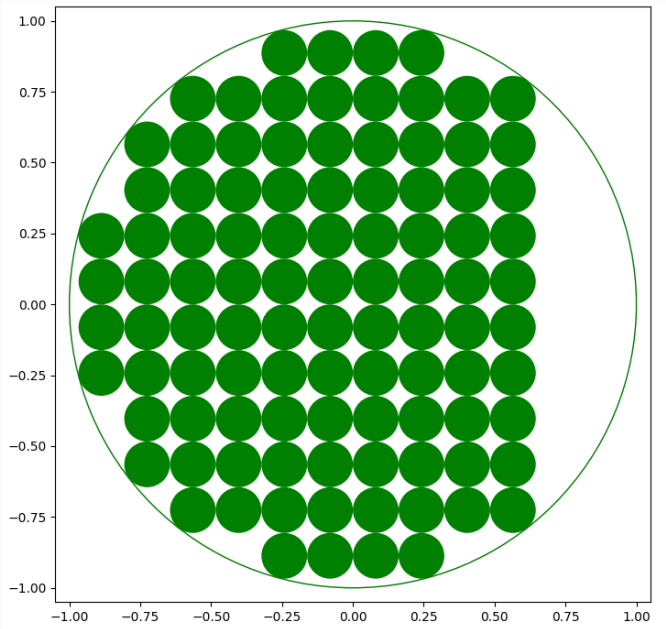
解法 0 — AtCoder 上の出力例（$R = 0.07$）

これは AtCoder の問題文の出力例にも書かれた、半径 $R = 0.07$ の敷き詰め方です。



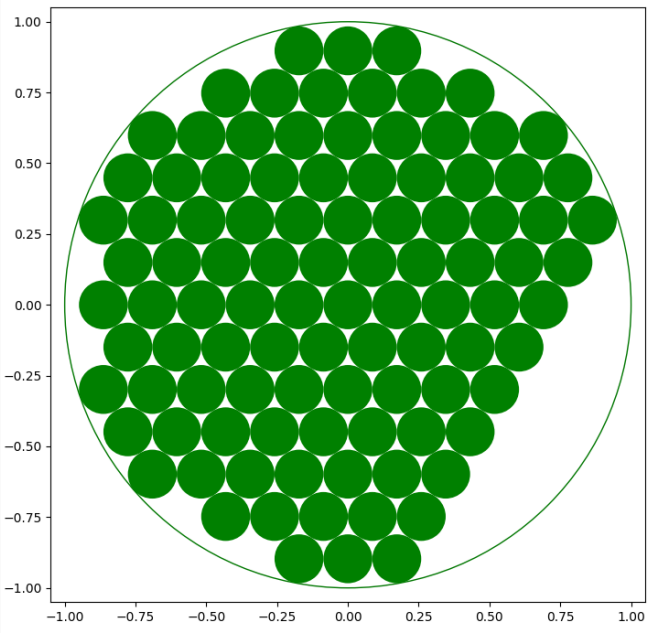
解法 1 — 正方形状に敷き詰め（$R = 0.0806$）

解法 0 の敷き詰め方だと、上下左右にまだスペースがあります。ここも敷き詰めると、$R = 0.0806$ が達成できます。



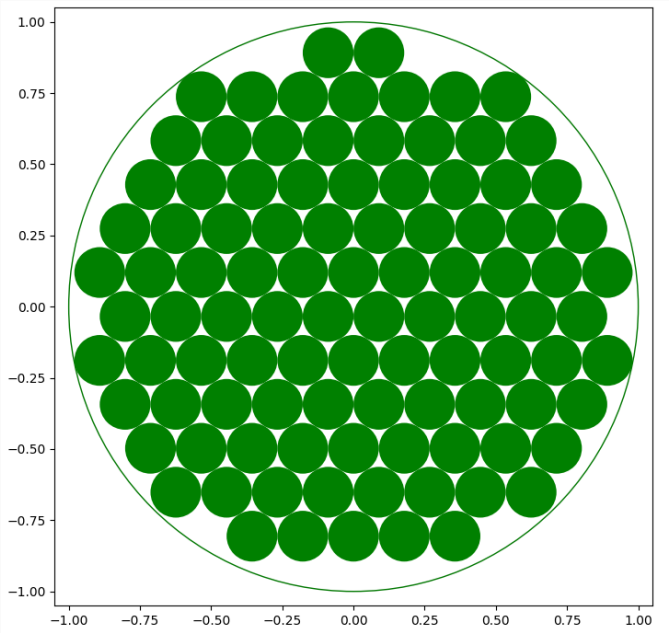
解法 2 — 六角形状に敷き詰め（$R = 0.0863$）

六角形の形に敷き詰めると、さらに効率的に敷き詰められます。敷き詰め可能な半径を二分探索を使って求めると、$R = 0.0863$ まで作れることが分かります。



解法 3 — 敷き詰めの中心位置をずらす（$R = 0.0891$）

解法 2 では、最も真ん中の円の中心座標を $(0, 0)$ に固定していました。これをずらすと、さらに効率的な敷き詰めができる場合があります。中心座標を何万パターンも試してみると、$R = 0.0891$ の敷き詰め方が見つかりました。



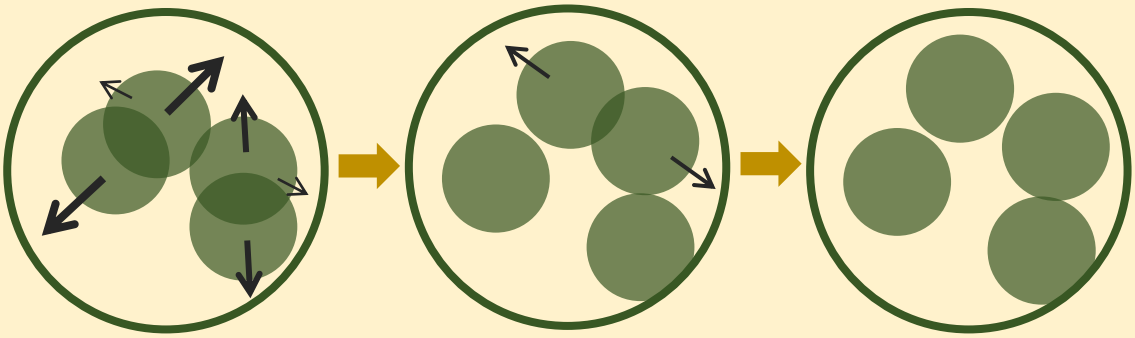
解法 4 — 勾配降下法を使う（$R = 0.0899$）

実は、勾配降下法（→**コラム 5**）を使うことで、もっと良い解を出すことができます。

解法のアイデア

最初、目標とする半径 R を固定します。（例えば $R = 0.0895$ など）

その後、ランダムに中心座標を決めて、$N = 100$ 個の円を描きます。いくつかの円は重なりますが、「円を少しずらす」ことによって円の重なりをどんどん減らしていき、最終的にどの円も重ならないようにするのが目標です。



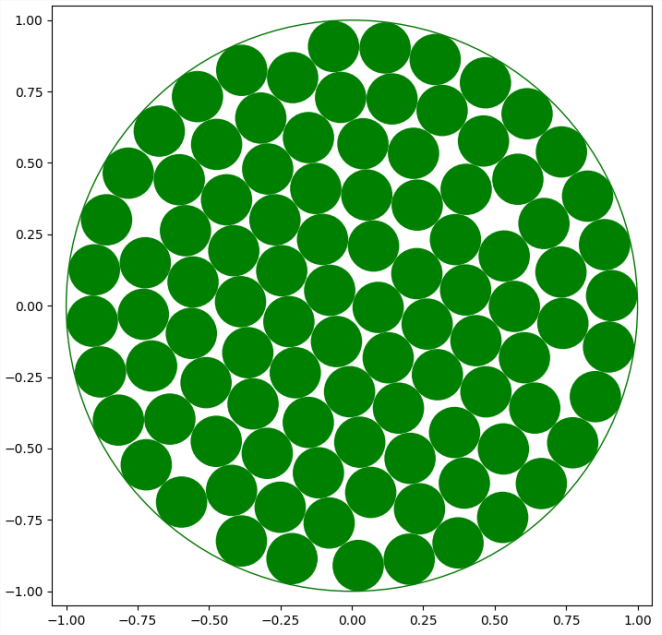
これは勾配降下法を用いて実装できます。例えば、**ペナルティー**を

$$P = \sum_{i=1}^{n} \max\left(\left((1-R) - \sqrt{x_i^2 + y_i^2}\right)^2 - R^2, 0\right) + \sum_{1 \le i < j \le n} \max\left((x_i - x_j)^2 + (y_i - y_j)^2 - R^2, 0\right)$$

などに設定して、P ができるだけ小さくなる方向に $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ を勾配降下法で移動させていきます。すると、最終的には P が局所的最適解になります。$P = 0$ になれば、目標達成です。

もちろん、1 回の試行では $P = 0$ の解が見つからない可能性があるので、勾配降下法のステップをある程度繰り返したところで打ち切り、リセットして $N = 100$ 個のランダムな円の配置からやり直します。目標とする半径 R の値によっては、この試行を何十回・何百回と繰り返して、やっと $P = 0$ の解にたどり着く場合もあります。

実際にやってみたところ、$R = 0.0895$ 程度なら数回の試行で $P = 0$ の解が見つかりました。一方、R が大きくなると解は見つかりにくくなり、$R = 0.0899$ では C++ のプログラムを 40 分程度回し、勾配降下法の試行を 3000 回ほど行ったところで、解が見つかりました。見つかった解は、次ページに載せています。



さらに効率的な敷き詰め方

この問題はよく研究されており、2021 年 12 月現在、$R = 0.0902352$ の解が見つかっています。詳しく知りたい方は、以下の論文をお読みください。

- Grosso, A., Jamali, A. R. M. J. U., Locatelli, M., & Schoen, F. (2010). Solving the problem of packing equal and unequal circles in a circular container. *Journal of Global Optimization, 47*(1), 63-81.

しかしながら、現時点で最も良い解は 2008 年に発見されたものです。この問題は、最適解がまだ知られていない未解決問題であり、将来この解がさらに改善される可能性も十分あります。

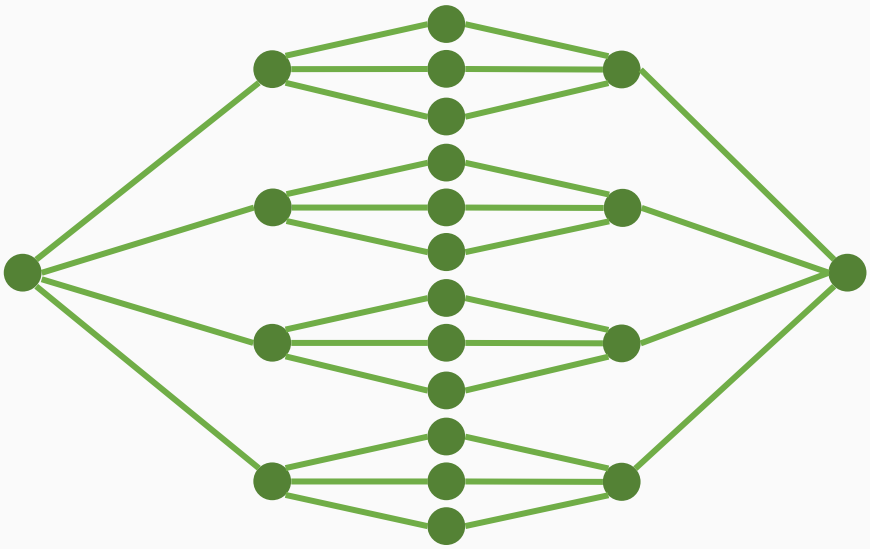
興味を持たれた方は、この未解決問題にぜひ挑戦してみましょう。

問題 30

次数 d 以下、直径 k 以下のできるだけ頂点数が多いグラフを作る問題は「次数直径問題」といい、よく研究されています。本問題は、$d = 4, k = 4$ の場合です。できるだけ頂点数が多くなるほど得点が上がる形式であり、様々なグラフの作り方が考えられます。

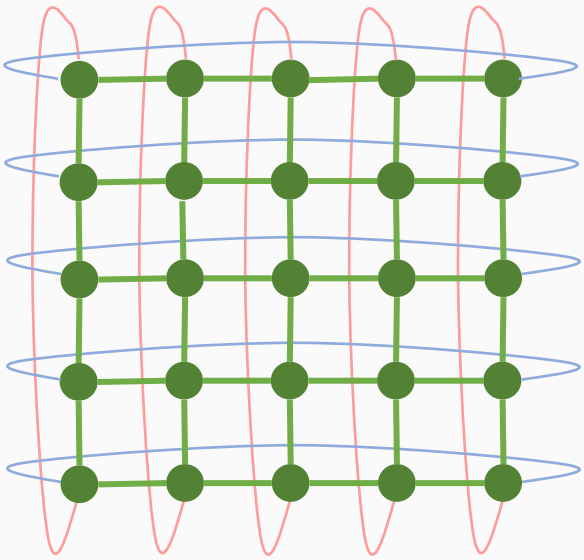
解法 0 — 問題文中の例（22 頂点）

これは、本の問題文にも例示されている、22 頂点のグラフです。



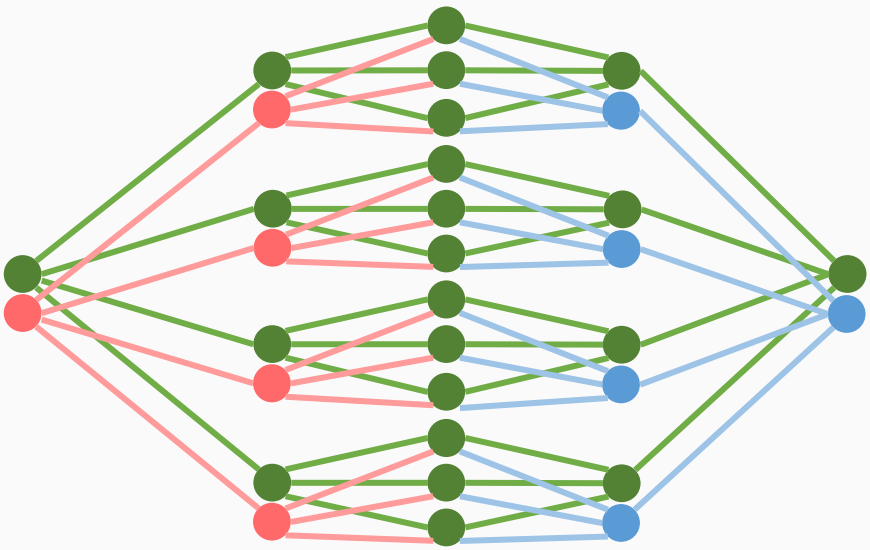
解法 1 — 5×5 のマス目のようなグラフ（25 頂点）

5×5 のマス目のようなグラフを作ると、左上から右下までの最短経路長が 8 になりますが、上と下の頂点、左と右の頂点をつなぐことによって、どの 2 頂点間の最短経路も 4 以下にできます。



解法 2 — 解法 0［問題文中の例］を工夫する（32 頂点）

先ほどの解法 0 は、真ん中の 12 頂点の次数が 2 であるという欠点があります。この 12 頂点をもとにグラフを以下のように拡張することができます。



それ以外にも、解法 0・1・2 で挙げたような「規則的な」グラフの作り方はたくさんあります。しかし、規則的なグラフを手で作るのにも限界があり、特に 40 頂点以上のグラフを見つけるのは難しいです。しかし、アルゴリズムの力で、70〜80 頂点程度のグラフを作ることもできてしまいます。

解法 3 — 山登り法を使ったアルゴリズム（73 頂点）

山登り法は、**コラム 5** でも言及されていますが、解を少しずつ改善していくことでできるだけ良い解を得るアルゴリズムです。山登り法はシンプルながらも非常に効果的なアルゴリズムであり、この問題にも適用することができます。

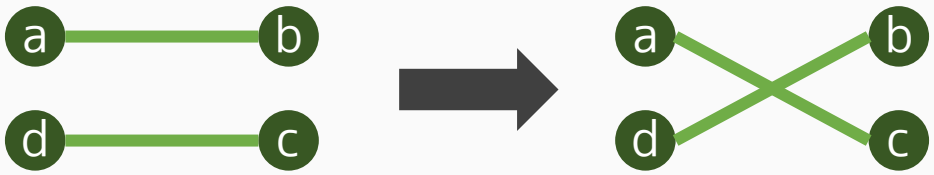
山登り法の方針は、以下の通りです。

1. 頂点数 N を決め、N 頂点の全ての次数が 4 のグラフ G をランダムに生成する。
2. 以下の操作をたくさん繰り返す
    - グラフの 2 辺をランダムに選び、これをつなぎ変える操作を行う
    - つなぎ変えたことで「最短距離が 5 以上の頂点の組の個数」が増加した場合のみ、つなぎ変えたグラフを元に戻す

3. 最終的に「最短距離が 5 以上の頂点の組の個数」が 0 になれば、目的のグラフが作れた

ただし、辺 $a-b$ と辺 $c-d$ をつなぎ変える操作とは、以下の操作のことを指します。

- 辺 $a-b$ と辺 $c-d$ を削除し、新たに辺 $a-c$ と辺 $b-d$ を追加する。
- ただし、現在のグラフに辺 $a-c$ または辺 $b-d$ が存在しない場合のみ、つなぎ変える操作ができる。



この操作を行っても、各頂点の次数は 4 で変わらず、グラフを「少しずつ改変」できるため、山登り法にとって都合のよい変化の方法になります。

実際にやってみると、$N=73$ にセットしても、山登り法でどんどんグラフが改善されていき、数秒で条件を満たすグラフが求まります。このグラフは規則性がなく複雑であるため、ここには掲載しないことにします。

また、山登り法を改良した「焼きなまし法」を使ったプログラムを 10 分ほど実行したところ、$N=79$ 頂点のグラフも求まりました。

さらに頂点数の多いグラフ

2021 年 12 月現在、次数が 4 以下で直径（2 頂点間の最短経路長の最大値）が 4 以下のグラフとして、98 頂点のものが発見されていますが、これが最適であるかは分かっておらず、さらに頂点数の多いグラフが発見される可能性もあります。詳しく知りたい方は、以下のウェブサイトをご覧ください。

- COMBINATORICS WIKI, The Degree Diameter Problem for General Graphs

（次数 d，直径 k）$=(4,4)$ の場合だけでなく、他の多くの (d,k) に対しても、改善の余地がまだ残されているどころか、現時点で最適性が知られているのは、自明なものを除き $(d,k)=(3,2),(4,2),(5,2),(6,2),(7,2),(3,3),(4,3)$ の 7 種類しかありません。

興味を持たれた方は、この未解決問題にぜひ挑戦してみましょう。