6.6 最終確認問題 26-30 的解答

問題 26

設實驗開始 n 秒後，物質 A、B、C 的量分別為 a_n、b_n,、c_n。a_n、b_n、c_n由以下的遞迴式決定：

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

(a_n, b_n, c_n) 與 $(a_{n+1}, b_{n+1}, c_{n+1})$ 的關係，可以使用矩陣（參見**4.7 節**）表示如下。。

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

反覆運用此遞迴式，則 a_T、b_T、c_T 可以用以下式子表示。（實驗開始時，物質 A、B、C 的量各為1克）

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

藉由將重複平方法（→ **4.6.7 項**）應用於矩陣，3 × 3 矩陣的乘方可以用計算複雜度 $O(\log T)$ 來計算。如此，可以求出這個問題的答案。

將此解法以C++實作如下。

```
#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 }
	};
};
```

```cpp
// 矩陣的乘法
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;
}

// 矩陣的乘方
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++) {
		// 輸入 → 建構矩陣 A
		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;

		// 矩陣乘方的計算 → 輸出答案
		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 的原始碼請參閱 GitHub的chap6-26_30.md。

問題 27

求出寫下的整數差為 k 以上的球的選取方法數量的問題稱為「問題 k」。問題 k 可以分解為以下小問題（→ **5.6 節**）：

- 選取（使差為 k 以上的）1 顆球的方法有幾種？
- 選取使差為 k 以上的 2 顆球的方法有幾種？
- 選取使差為 k 以上的 2 顆球的方法有幾種？
- （中略）
- 選取使差為 k 以上的 $\lceil N/k \rceil$ 顆球的方法有幾種？

可以在 $\lceil N/k \rceil$ 個中止的理由是因為即使選取再好，最多也只能選取 $\lceil N/k \rceil$ 顆球。這樣，問題 k 可以分解成 $\lceil N/k \rceil$ 個「似乎更容易解決的小問題」。據此，要將問題 1, 2, 3, ⋯ , N 解決的話，整體要解決如下這麼多個小問題。

$$\left\lceil \frac{N}{1} \right\rceil + \left\lceil \frac{N}{2} \right\rceil + \left\lceil \frac{N}{3} \right\rceil + \cdots + \left\lceil \frac{N}{N} \right\rceil$$

從倒數 $1/x$ 的和會為 $O(N \log N)$（→ **4.4.4 項**）的性質可知，此為 $O(\log N)$ 個。

接下來，來考慮每個小問題如何解決吧。

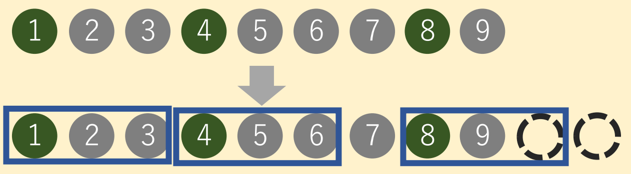
「小問題」是怎樣的問題？

從寫有 $1,2,\ldots,N$ 的球中，選取 k 個球，使得任意兩個的數的差距為 m 以上的方法有幾種？

小問題的解答

這個小問題的解答是 ${}_{N-(k-1)(m-1)}C_m$ 種。

理由是，選取球 x 時，如下圖所示，將球 $x, x+1, \ldots, x+m-1$ 框起來思考的話，會變成從球 $1, 2, \ldots, N+m-1$ 中選取 k 個互不重疊的「長度 m 的框」。



$N=9, m=3, k=3$ 的情況為例

將 2 個單獨的球和 3 個框進行排列，所以是 ${}_5C_3 = 10$ 種

二項係數可以用預先計算階乘的方法（→**程式碼 4.6.6**），以 $M = 10^9 + 7$ 而言在 $O(\log M)$ 時間內計算出來。如前所述，由於需要解決 $O(N \log N)$ 個小問題，故本問題可以整體計算複雜度 $O(N \log N \log M)$ 解決。

將此解法以 C++ 實作如下。

```
#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) {
	// 重複平方法（p 取 a^1, a^2, a^4, a^8, ... 等值）
	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 是將 n! 除以 r! × (n-r)! 的值
	return Division(fact[n], fact[r] * fact[n - r] % mod, mod);
}

int main() {
	// 陣列的初始化（fact[i] 為將 i 的階乘除以 1000000007 的餘數）
	fact[0] = 1;
	for (int i = 1; i <= 100000; i++) {
		fact[i] = 1LL * i * fact[i - 1] % mod;
	}

	// 輸入 → 計算答案並輸出
	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的原始碼請參閱GitHub的chap6-26_30.md。

問題 28

本問題與 **5.7.5 項** 的加法金字塔的問題類似，但因為有對顏色定下規則，所以看似難以著手。在此，為了方便，將顏色如下改成 ID。

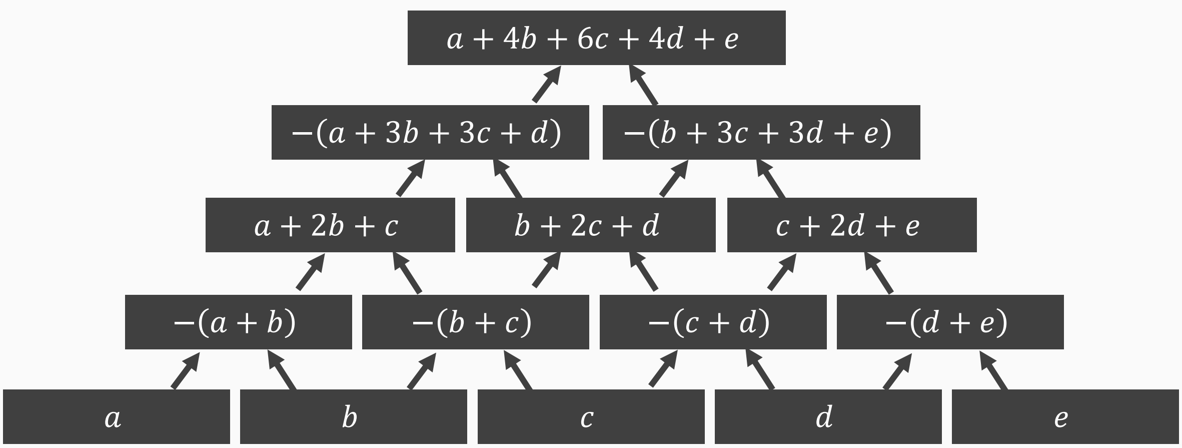
- 「藍」→ **0**、「白」→ **1**、「紅」→ **2**

假設正下方兩個方塊的顏色 ID 為 x、y，則上面的方塊顏色如右表所示。

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

也就是說，假設正下方的兩個方塊顏色 ID 為 x、y，上面的方塊顏色 ID 可以由 $-(x+y) \bmod 3$ 計算※。

思考 $N=5$ 層的金字塔的狀況吧。令底部的方塊顏色分別為 a、b、c、d、e，則各方塊的顏色如下圖所示（全部都以 mod 3 來思考）。



同樣地，對於一般情況，頂端的顏色ID以下式求出。

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

那麼，如何求出 $nCr \bmod 3$ 呢？因為 3 與 n、r 相比較小，因此無法適用使用反元素的方法（→ **4.6.8 項**）。

假設將 n 以三進制表示為 $n_{d-1}n_{d-2}\dots n_1 n_0$，$k$ 以三進制表示為 $k_{d-1}k_{d-2}\dots k_1 k_0$。此時，

使用**盧卡斯定理**吧。

> 將 n 以三進制表示為 $n_{d-1}n_{d-2}\ldots n_1 n_0$ ， r 以三進制表示為 $r_{d-1}r_{d-2}\ldots r_1 r_0$ 。此時， $nCr \bmod 3$ 如下計算：
>
> $$(n_{d-1}Cr_{d-1} \times n_{d-2}Cr_{d-2} \times \cdots \times n_1 Cr_1 \times n_0 Cr_0) \bmod 3$$
>
> 其中，在式子過程中變為 $n_i < r_i$ 狀況下，則 $nCr \bmod 3 = 0$ 。對於一般的 $\bmod M$ ，同樣的定理也成立。

nCr 可以用計算複雜度 $O(\log n)$ 來計算，此問題可以用整體的計算複雜度 $O(N \log N)$ 來解決。

此解法以 C++ 實作如下。此外，nCr 的計算是用遞迴函數進行實作。（亦可以用 **2.1.9 項** 的方法轉換為三進制）

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

// 用盧卡斯定理計算 ncr mod 3
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() {
    // 輸入
    int N; string C;
    cin >> N >> C;

    // 求出答案
    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;
    }

    // 將答案乘以 (-1)^(N-1)
    if (N % 2 == 0) {
        answer = (3 - answer) % 3;
    }
```

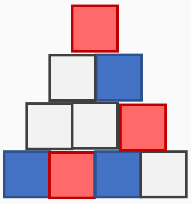
```
    // 輸出答案（"BWR" 的第 answer 個字母）
    cout << "BWR"[answer] << endl;

    return 0;
}
```

Python、JAVA、C 的原始碼請參閱 GitHub的chap6-26_30.md。

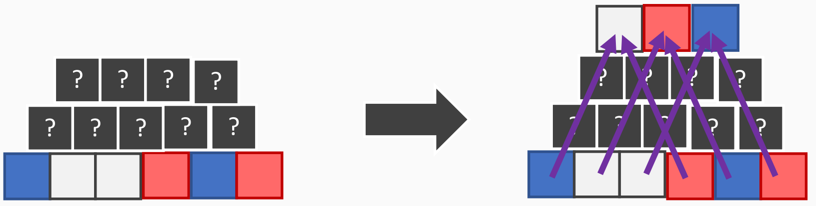
問題 28 — 別解

這個問題有其他解法。即為使用「考慮規律性」（→ **5.2 節**）的解法。

考慮 $N = 4$ 的情況。實際上不論情況如何，頂部方塊的顏色只取決於最左和最右的方塊。第2、3個方塊的顏色不影響結果。在右圖的例子中，最左的方塊為藍色，最右的方塊為白色，因此頂部的方塊為紅色。



在 $N = 10$ 的情況也同樣地只取決於最左和最右的方塊，與第 2, 3, 4, 5, 6, 7, 8, 9 個方塊無關。對於 $N = 28, 82, 244, 730, 2188, 6562, \ldots$ 的情況也一樣。一般而言，只需要看最左和最右的方塊，就能求出 $3k$ 層上的方塊配置。。



巧妙利用這一點的話，就只需要計算 $O(\log N)$ 層的方塊。例如， $N = 23$ 時，利用三進制而得 22 = 9 + 9 + 3 + 1，因此會成為輸入第1層方塊後，求出第10層、第19層、第22層、第23層這樣的流程。每個步驟需要計算複雜度 $O(N)$， 總共 有 $O(\log N)$ 步，因此本問題以整體計算複雜度量 $O(N \log N)$ 來解決。。

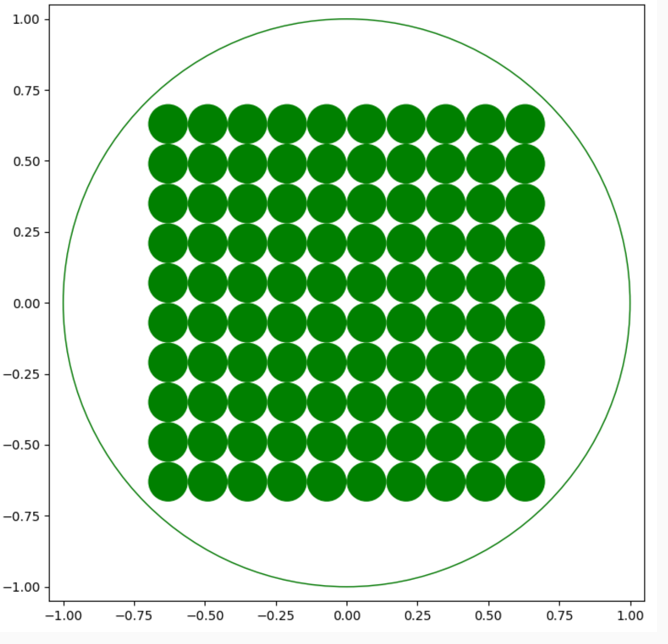
實作程式碼因篇幅問題省略。

問題 29

本問題是在半徑為1的圓內填滿半徑盡可能大的100個小圓。因為配置的半徑越大，得分越高的形式，對於這個問題可以考慮各種解法。。

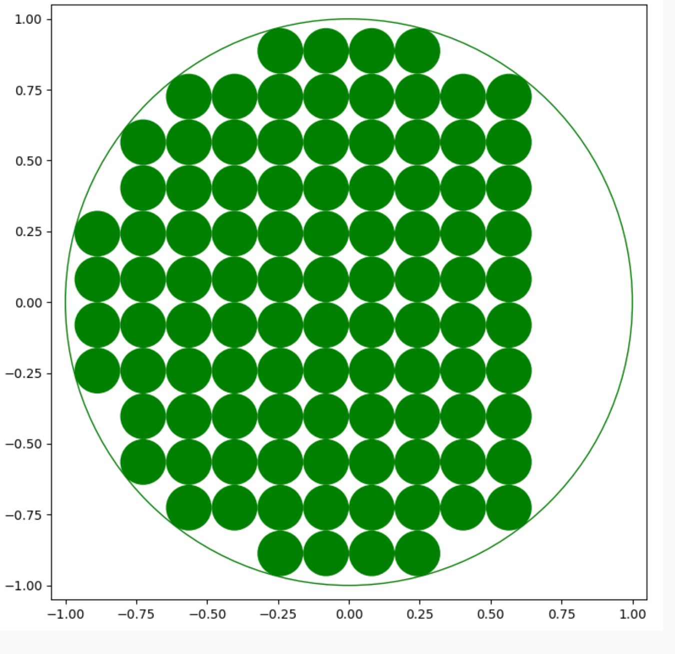
解法 0 — AtCoder上的輸出例($R = 0.07$)

這也寫在 AtCoder 的問題描述的輸出例，半徑 $R = 0.07$ 的填滿方法。



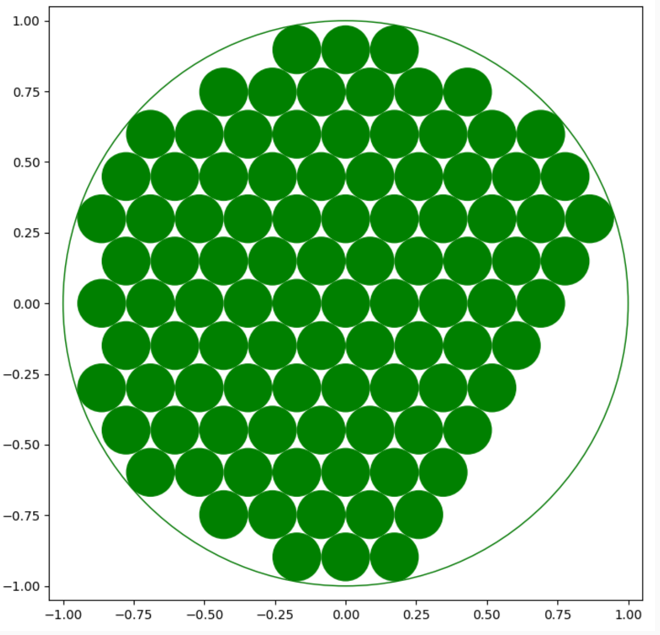
解法 1 —填滿成正方形（R=0.0806）

在解法 0 的填滿方法中，上下左右還有空間。如果這些空間也被填滿，可以達到 $R = 0.0806$ 。



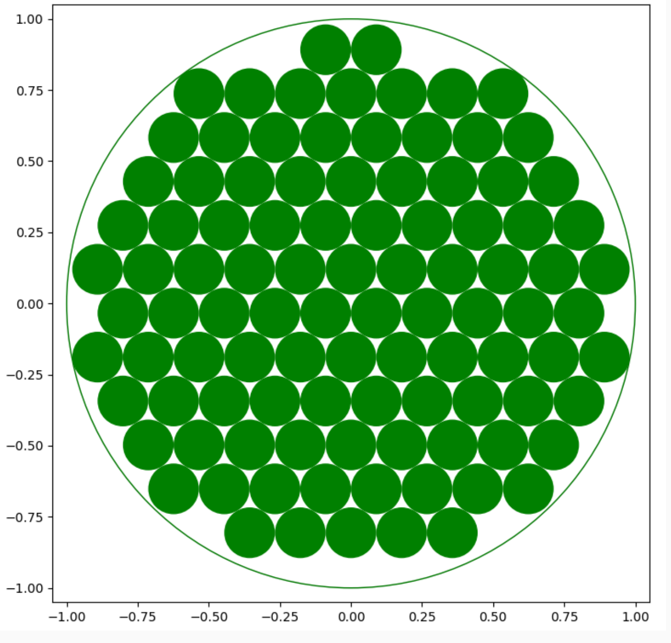
解法 2 — 填滿成六角形($R = 0.0863$)

以六角形的形狀進行，可以更有效地填滿。使用二元搜尋來求出可填滿的最大半徑，則可以達到 $R = 0.0863$ 。



解法 3 —偏移填滿的中心位置 ($R = 0.0891$)

在解法2中，最中間的圓的中心固定在 (0, 0)。如果將此中心偏移，有時可以找到更有效進行填滿的情況。經過嘗試數萬種中心位置，找到了 $R = 0.0891$ 的填滿方法。



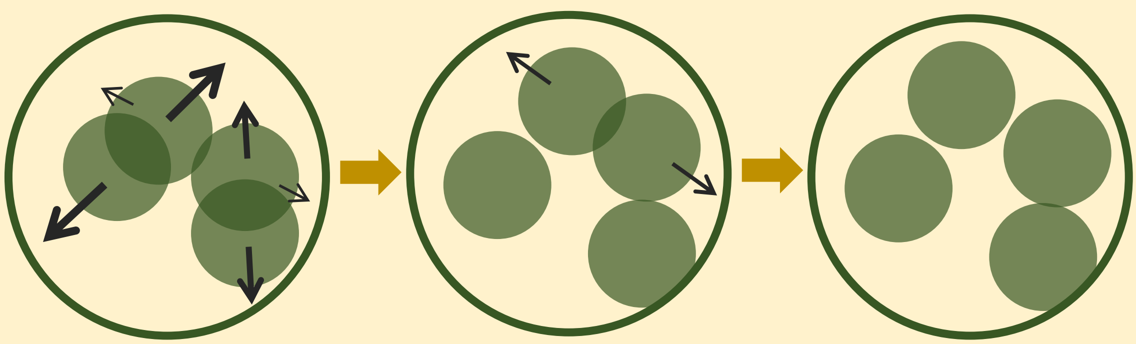
解法4—使用梯度下降法 ($R = 0.0899$)

實際上，藉由使用梯度下降法（→**專欄 5**）可以找到更好的解。

解法的想法

首先，固定作為目標的半徑 R。（例如 $R = 0.0895$等））

然後，隨機決定中心座標，描繪出 $N = 100$ 個圓。雖然有些圓會重疊，但藉由「將圓稍微移動」來逐漸減少圓的重疊，目標是最終讓所有圓都不重疊。



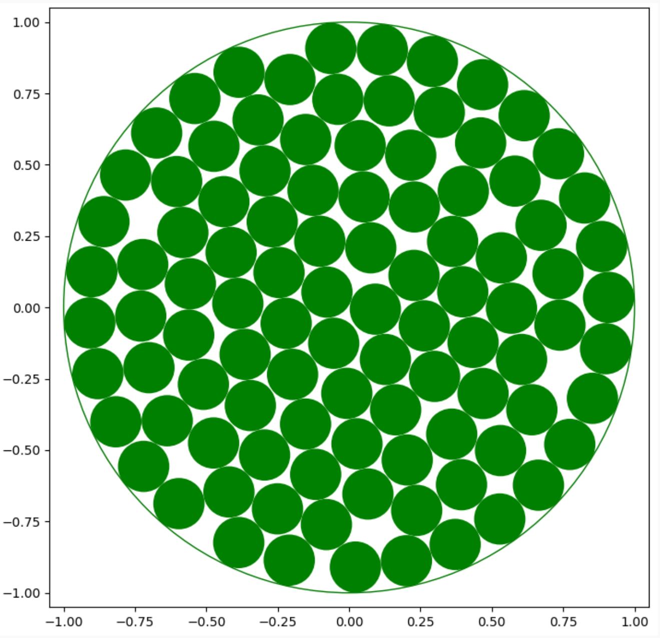
這可以使用梯度下降法來實作。例如，設定**懲罰**函數等如下：

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

朝使 P 盡可能減小的方向，將 $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ 透過梯度下降法來移動。如此，最終 P 會成為局部最佳解。如果 $P = 0$，則達成目標。

當然，一次的嘗試可能無法找到 $P = 0$ 的解，因此將梯度下降法的步驟執行到一定程度後中止，重新設置並從 $N = 100$ 個隨機配置的圓再次開始。根據作為目標的半徑 R 的值，也可能需要重複嘗試數十次或數百次才能找到 $P = 0$ 的解。

實際操作中，若是 $R = 0.0895$ 程度的話，嘗試數次就能找到 $P = 0$ 的解。另一方面，隨著 R 的增加，找到解的難度會增加，$R = 0.0899$ 時，在C++的程式執行約40分鐘，進行了3000次左右梯度下降法的嘗試才找到解。找到的解刊載如下。



更有效率的填滿方法

這個問題已被廣泛研究，截至2021年12月，已找到 $R = 0.0902352$ 的解。想了解更多的讀者可以參考以下論文：。

- 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.

然而，目前最好的解是在2008年發現的。本問題的最佳解仍然未知，是一個未解決的問題，將來十分有可能更進一步改善此解。

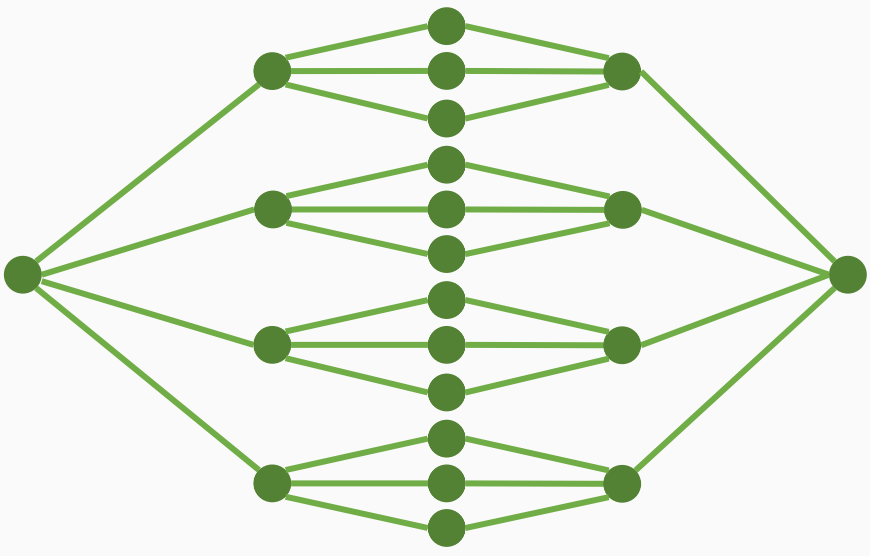
有興趣的話，請務必嘗試挑戰這個未解決的問題吧。

問題 30

作出使度 d 以下，直徑 k 以下的頂點數盡可能多的圖形問題稱為「度-直徑問題」，經常被研究。本問題是 $d = 4, k = 4$ 的情況。盡可能使頂點數越多則得分越高的形式，可以思考各種圖的作成方法。

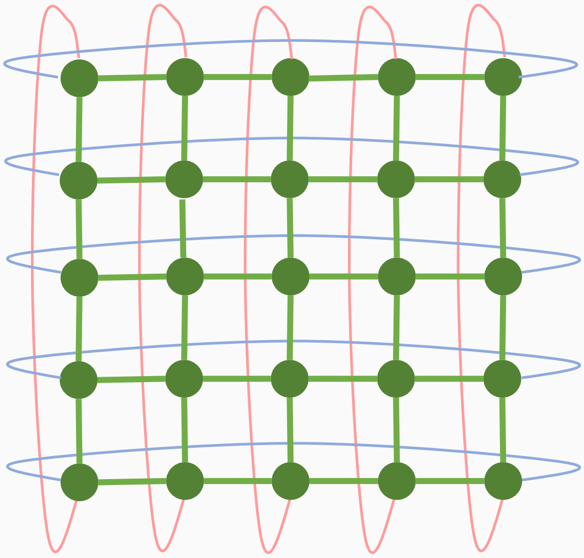
解法 0 —問題描述中的例（22頂點）

這是在書中的問題描述亦有示範的包含22個頂點的圖。



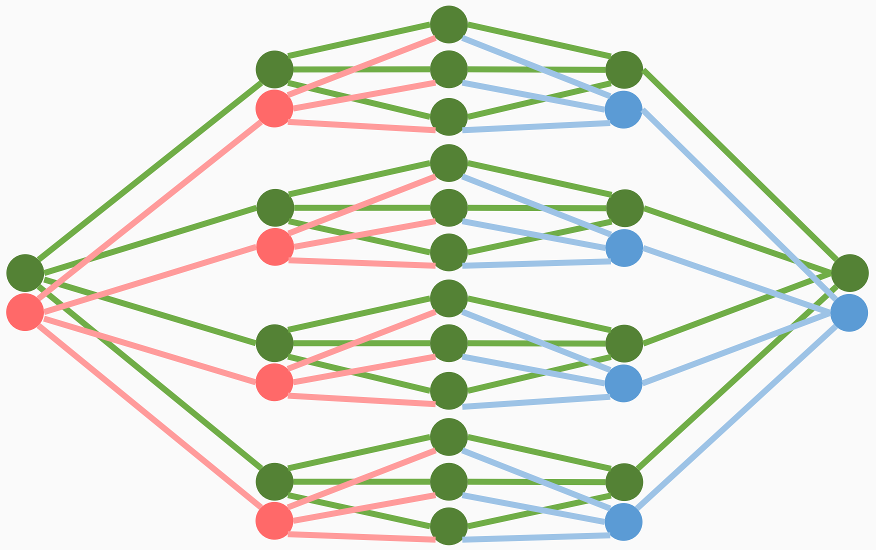
解法 1 — 5×5 的格子狀圖（25頂點）

作出一個5×5的格子狀的圖，從左上到右下的最短路徑長度為8，但藉由將上和下的頂點，左和右的頂點連接起來，可以使任何兩頂點之間的最短路徑為4 以下。



解法 2 — 將解法 0 [問題描述中的例] 加工（32頂點）

先前的解法0具有正中心的12個頂點的度為2的缺點。以這12個頂點為基礎，可以將圖進行如下擴展。



除此之外，如解法 0、1、2 所舉出的「規律性的」圖的作成方法還有許多。但是，手動作出規律性的圖亦有極限，尤其是難以找到 40 個以上頂點的圖。然而，藉由演算法的力量，可以作出 70〜80 個頂點左右的圖。。

解法 3 — 使用登山法的演算法（73頂點）

登山法在 **專欄 5** 也有提及，是藉由一點點改進解而得到盡可能好的解的演算法。登山法是簡單同時也很有效的演算法，也可以運用於此問題。

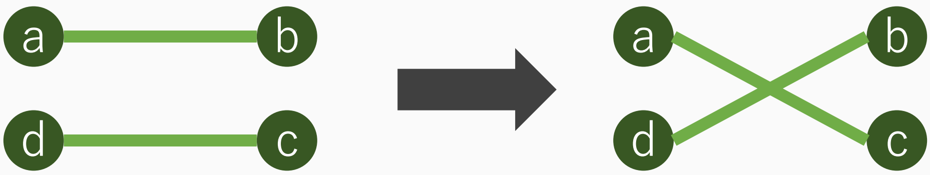
登山法的方針如下：

1. 隨機生成N 個頂點所有度都為4的圖G。。
2. 大量重複以下操作：
    - 隨機選擇圖的兩條邊，進行改變連接的操作
    - 只有在改變連接後「最短距離為5以上的頂點組合的數量」增加的情況下，將改變連接後的圖還原

3. 最終，如果「最短距離為5以上的頂點組合的數量」為0，則目標的圖完成。

其中，邊 $a-b$ 和邊 $c-d$ 改變連接操作如下：

- 刪除邊 $a-b$ 和邊$c-d$，新增邊 $a-c$ 和邊 $b-d$ 。
- 但只有在圖中不存在邊 $a-c$ 和邊 $b-d$ 時才能進行改變連接的操作。



進行這種操作仍可使各頂點的度維持4不變，可以將圖「一點點改變」，因此對登山法而言，會是很方便的變化方法。

實際去做後，即使設定 $N=73$，也能以登山法漸漸改善圖，在幾秒內求出滿足條件的圖。由於此圖不具規律性且很複雜，所以這裡不會刊載。

此外，使用將登山法改良後的「模擬退火法」的程式，在運行約 10 分鐘後，也可找到 $N=79$ 個頂點的圖。

頂點數更多的圖

截至2021年12月，度為 4 以下且直徑（兩頂點間最短路徑長度的最大值）為4以下的圖而言，有98個頂點的圖已經被發現，但還未知道這是否為最佳解，尚有可能會發現更多頂點的圖。想了解更多的讀者請參考以下網站：
COMBINATORICS WIKI, The Degree Diameter Problem for General Graphs

不只是（度d, 直徑k）=(4, 4) 的情況，對於其他多種(d, k)，是否也還有改善空間呢？目前已知最佳解的情況，除了不言而喻的以外，只有$(d,k)=$ $(3,2),(4,2),(5,2),(6,2),(7,2),(3,3),(4,3)$ 這 7 種。

有興趣的話，請務必嘗試挑戰這個未解決的問題吧。