

演算法×數學」圖解學習全指南

解答・解說

1

解說的目次

第 2 章 用於演算法的數學基本知識	3
節末問題 2.1 的解答 4	節末問題 2.4 的解答 13
節末問題 2.2 的解答 6	節末問題 2.5 的解答 16
節末問題 2.3 的解答 9	
第 3 章 基本的演算法	20
節末問題 3.1 的解答 21	節末問題 3.5 的解答 36
節末問題 3.2 的解答 23	節末問題 3.6 的解答 39
節末問題 3.3 的解答 26	節末問題 3.7 的解答 42
節末問題 3.4 的解答 31	
第 4 章 進階的演算法	47
節末問題 4.1 的解答 48	節末問題 4.5 的解答 70
節末問題 4.2 的解答 54	節末問題 4.6 的解答 81
節末問題 4.3 的解答 59	節末問題 4.7 的解答 85
節末問題 4.4 的解答 63	
第 5 章 用以解決問題的數學思考	95
節末問題 5.2 的解答 96	節末問題 5.7 的解答 114
節末問題 5.3 的解答 100	節末問題 5.8 的解答 118
節末問題 5.4 的解答 103	節末問題 5.9 的解答 123
節末問題 5.5 的解答 108	節末問題 5.10 的解答 129
節末問題 5.6 的解答 111	
第 6 章 最終確認問題	140
問題 01~15	141
問題 16~30	153

第 2 章

用於演算法的
數學基本知識

2.1

節末問題 2.1 的解答

問題 2.1.1

這是測試對於數的分類（→[2.1.1項](#)）的理解的問題。解答是：

- 整數： $-100, -20, 0, 1, 70$
- 正整數： $1, 70$

整數是指不含一數點的數。正整數是指在這之中大於 0 的數。

問題 2.1.2

這是測試對於代數式及其寫法（→[2.1.2項](#)）的理解的問題。解答是：

- $A + B + C = 25 + 4 + 12 = 41$
- $ABC = 25 \times 4 \times 12 = 1200$

在此，注意 ABC 是表示 $A \times B \times C$ 的意思。

問題 2.1.3

這個問題是「輸入3個整數，並輸出其乘積」的意思。因此，編寫如下程式即可得到正確答案。對於不熟悉像數列一樣附有編號的文字式的人，請回到 [2.1.4 項](#) 和 [2.1.5 項](#) 複習。

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

int main() {
    int A[4];
    cin >> A[1] >> A[2] >> A[3];
    cout << A[1] * A[2] * A[3] << endl;
    return 0;
}
```

※ Python等原始碼請參閱 GitHub 的 codes 資料夾。

問題 2.1.4 (1)

這是測試如何將二進制轉換為十進制（→**2.1.7項**）的問題。進行如下計算，可以知道答案是 9。

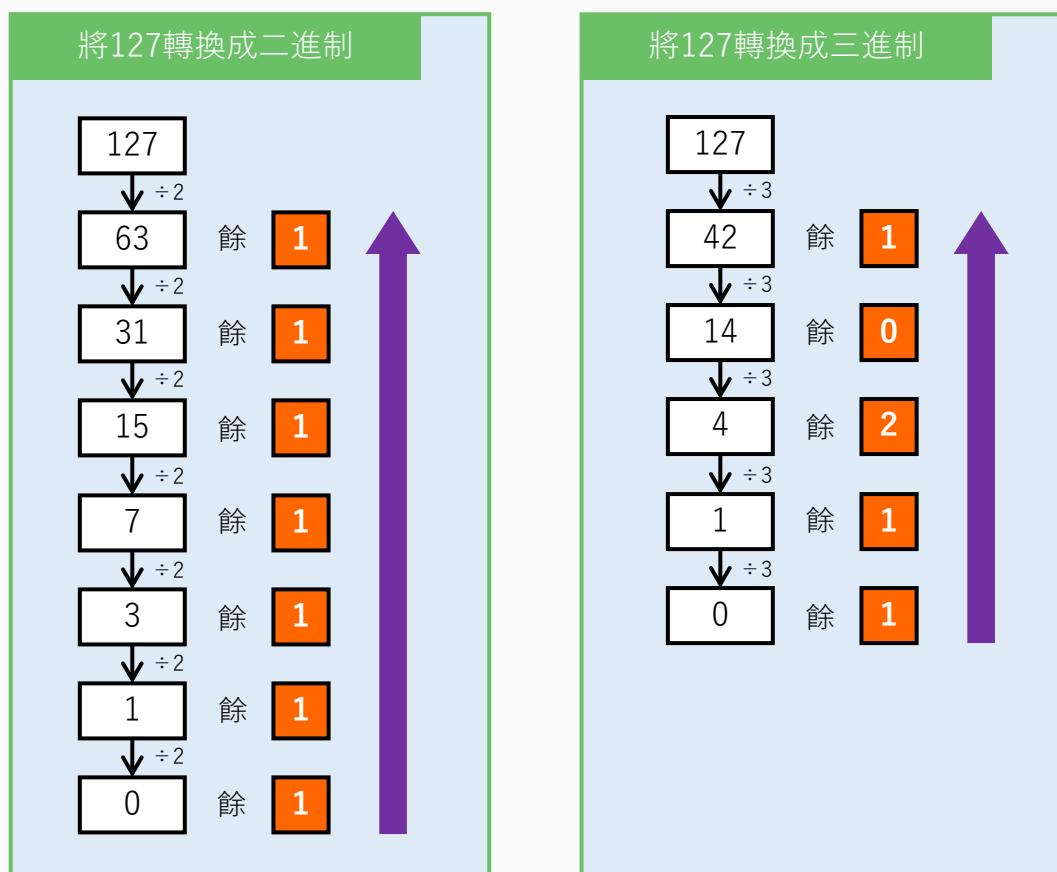
- 8 位的值是 1
- 4 位的值是 0
- 2 位的值是 0
- 1 位的值是 1
- 位和該位的數字合起來算是 $(8 \times 1) + (4 \times 0) + (2 \times 0) + (1 \times 1) = 9$

問題 2.1.4 (2)

這是測試如何將十進制轉換為二進制等（→**2.1.9項**）的問題。一般來說，將十進制轉換為 K 進制時，在值變為 0 為止前不斷除以 K ，並將餘數倒過來讀即可。因此，通過以下計算可以知道：

- 127 二進制表示是 1111111
- 127 三進制表示是 11201

示意圖如下所示。



2.2

節末問題 2.2 的解答

問題 2.2.1

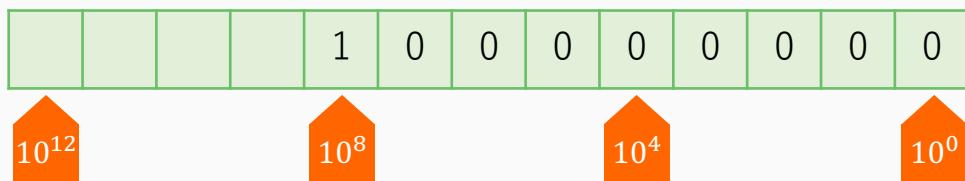
這個問題是測試是否理解乘方（→2.2.3項）的問題。答案如下：

- 1 萬是 10 的 4 次方 (10^4)
- 1 億是 10 的 8 次方 (10^8)
- 1 兆是 10 的 12 次方 (10^{12})

如下思考「計算 10 的 n 次方時， n 增加 1，0 的數量就增加 1」的話，即可簡單知道答案。

- $10^1 = 10$ (1 個 0)
- $10^2 = 10 \times 10 = 100$ (2 個 0)
- $10^3 = 10 \times 10 \times 10 = 1000$ (3 個 0)
- $10^4 = 10 \times 10 \times 10 \times 10 = 10000$ (4 個 0)

因此，1萬是 10000 (4 個 0)，1億是 100000000 (8 個 0)，1兆是 1000000000000 (12 個 0)，因此分別是 $10^4, 10^8, 10^{12}$ 。



問題 2.2.2

這是測試是否理解乘方（→2.2.3項）和方根（→2.2.4項）的問題。答案如下：

- $29^2 = 841$ ，且 $\sqrt{841} = 29$
- $4^5 = 1024$ ，且 $\sqrt[5]{1024} = 4$

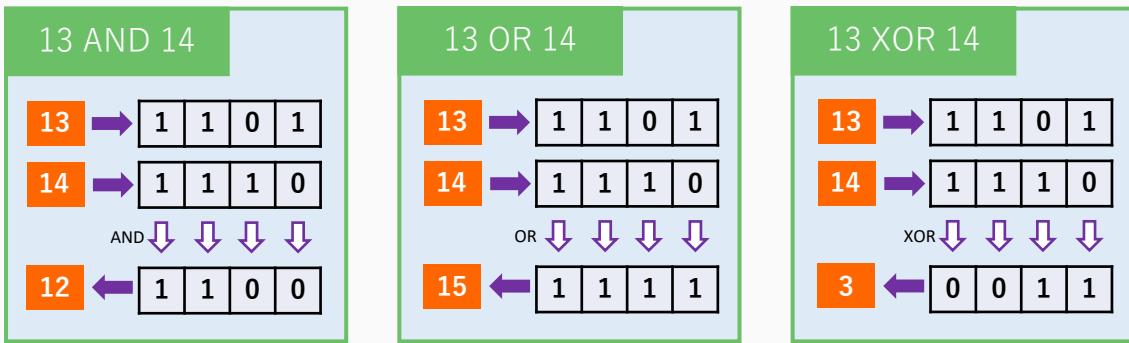
另外，當 $a^b = x$ 時， $\sqrt[b]{x} = a$ 的重要性質成立。

問題 2.2.3 (1)

這是測試是否理解2個數的位元運算（→2.2.7項～2.2.9項）的問題。藉由以下計算，可以得到：

- $13 \text{ AND } 14 = 12$
- $13 \text{ OR } 14 = 15$
- $13 \text{ XOR } 14 = 3$

不懂的話，請回想一下邏輯運算中的 AND 是「兩者都為 1 則為 1」，OR 是「其中一個為 1 則為 1」，XOR 是「只有一個為 1 則為 1」。位元運算是將每個數轉換為二進制，並對每個位元進行邏輯運算。。



問題 2.2.3 (2)

這是測試是否理解3個以上數字的位元運算（→2.2.11項）的問題。藉由以下計算，可以得到答案是 15。

$$\begin{aligned} & ((8 \text{ OR } 4) \text{ OR } 2) \text{ OR } 1 \\ &= (12 \text{ OR } 2) \text{ OR } 1 \\ &= 14 \text{ OR } 1 \\ &= 15 \end{aligned}$$

另外，將 8、4、2、1 轉換為二進制，分別為 1000、0100、0010、0001。由於這 4 個數的每一位都存在有 1 個以上的「1」，因此可以計算出答案以二進制表示為 1111（十進制為 15）。

問題 2.2.4

這是測試是否理解餘數 (mod) 的實作方法（→2.2.1項）的問題。在 C++ 中，藉由撰寫如下程式可以得到正確答案。此外，在 C++ 和 Python 等程式語言中，變數 a 除以 b 的餘數可以用 $a \% b$ 計算。

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

int N, A[109];
int Answer = 0;

int main() {
    // 輸入    cin >> N;
    for (int i = 1; i <= N; i++) {
        cin >> A[i];
    }

    // 計算答案
    for (int i = 1; i <= N; i++) {
        Answer += A[i];
    }

    // 輸出
    cout << Answer % 100 << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap2-2.md 。

2.3

節末問題 2.3 的解答

問題 2.3.1

這是測試是否理解 $f(x)$ 這樣的函數標記（→2.3.1項）、多項式函數（→2.3.7項）的問題。答案如下。

- $f(1) = 1^3 = 1 \times 1 \times 1 = 1$
- $f(5) = 5^3 = 5 \times 5 \times 5 = 125$
- $f(10) = 10^3 = 10 \times 10 \times 10 = 1000$

另外， $f(x) = ax^3 + bx^2 + cx + d$ 形式的函數稱為 **三次函數**。這個問題的 $f(x) = x^3$ 也是三次函數的一種。

問題 2.3.2 (1)

這是測試對對數函數（→2.3.10項）的理解的問題。

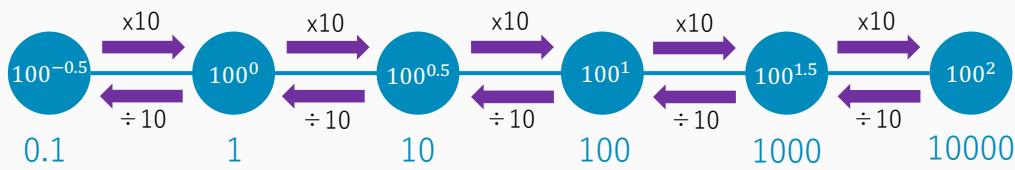
因為 $2^3 = 8$ ，所以答案是 $\log_2 8 = 3$ 。（參考第36頁的例子）

問題 2.3.2 (2)

這是測試對乘冪擴展（→2.3.8項）的理解的問題。

一般來說，因為 $a^{\frac{n}{m}} = \sqrt[m]{a^n}$ 成立，所以將 $a = 100$ 、 $n = 3$ 、 $m = 2$ 代入，可以得知

答案為 $100^{1.5} = \sqrt{100^3} = \sqrt{1000000} = 1000$ 。



問題 2.3.2 (3)

這是測試對取底函數、取頂函數（→2.3.11項）的理解的問題。

$[20.21]$ 是 20.21 以下最大的整數 20。

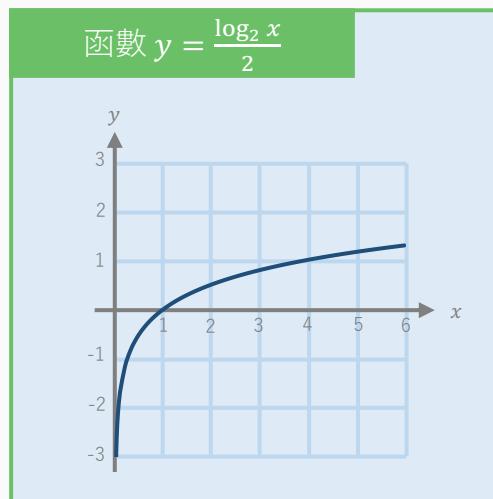
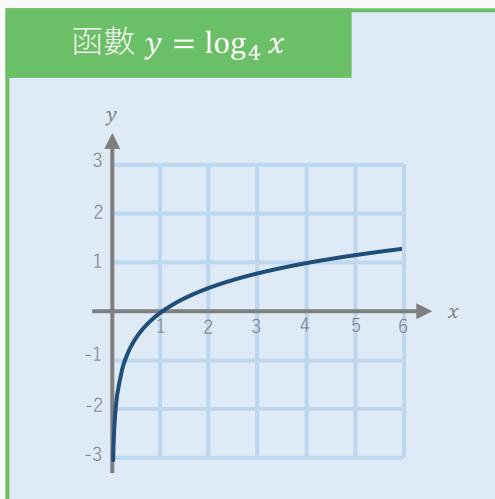
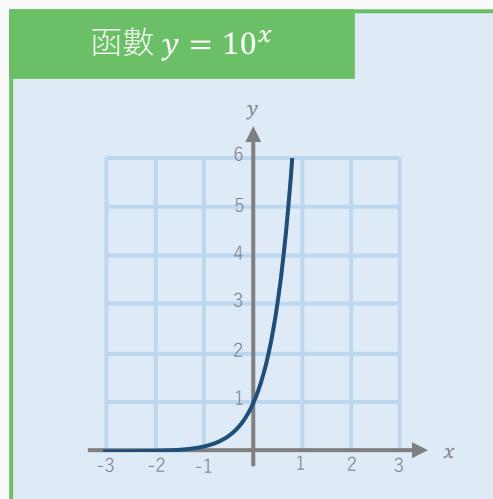
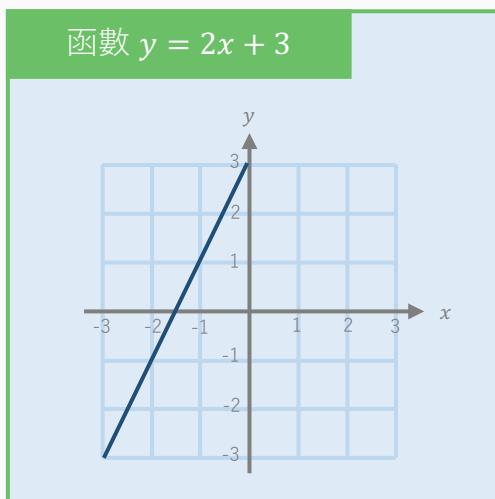
$[20.21]$ 是 20.21 以上最小的整數 21。

問題 2.3.3

這是測試對函數圖形（→2.3.4項）的理解的問題。答案如下圖所示。注意以下幾點，會更容易繪製圖形。

- 一次函數的圖形是直線
- 指數函數是單調遞增，且遞增速度非常快
- 對數函數是單調遞增，但遞增速度較慢

此外，請注意第 3 個圖形和第 4 個圖形完全相同。根據底數轉換公式， $\log_4 x = \log_2 x \div \log_2 4 = (\log_2 x)/2$ 會成立。



問題 2.3.4

這是測試對指數法則（→2.3.9項）的理解的問題。答案如下：

1. 當 $f(x) = 2^x$ 時， $f(20) = 2^{20} = 1048576$ 。
2. 根據指數法則， $2^{20} = 2^{10} \times 2^{10}$ 。因為 2^{10} 大約等於 1000，所以 2^{20} 大約是 $1000 \times 1000 = 1000000 (= 10^6)$ である。

問題 2.3.5 (1)

這是測試對對數函數 (\rightarrow 2.3.10項) 的理解的問題。

由於 $10^6 = 1000000$ ，因此 $g(1000000) = \log_{10} 1000000 = 6$ 。

問題 2.3.5 (2)

這是測試對對數函數公式 (\rightarrow 2.3.10項) 的理解的問題。

$$\log_2 16N - \log_2 N$$

$$= \log_2 \left(\frac{16N}{N} \right)$$

$$= \log_2 16 = 4$$

因此，答案是 4。另外，對數函數 $\log_a b$ 具有「當真數 b 以常數倍增（如 2 倍等）時，對數值會增加固定的量」這一性質。

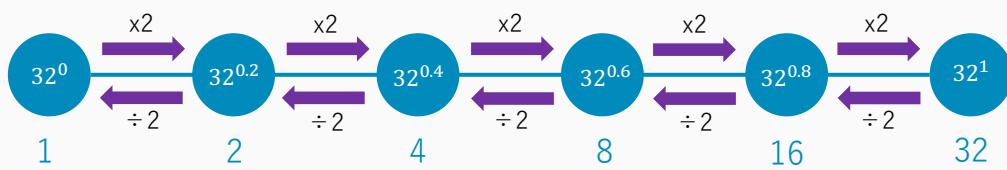
問題 2.3.6

這是測試是否可以熟練使用指數為小數時的乘幕公式 (\rightarrow 2.3.8項)、指數法則

(\rightarrow 2.3.9項) 的問題。每個問題的答案如下所示。

編號	規模	差為幾倍？	答案
1.	6.0 vs 5.0	$32^{6.0-5.0} = 32^{1.0}$	= 32 倍
2.	7.3 vs 5.3	$32^{7.3-5.3} = 32^{2.0}$	= 1024 倍
3.	9.0 vs 7.2	$32^{9.0-7.2} = 32^{1.8}$	= 512 倍

此外， $32^{1.8}$ 的值可以透過 $32^{1.0} \times 32^{0.8} = 32 \times 16 = 512$ 得到。另外，像 $32^{0.8}$ 這樣的值參考 2.3.8 項的圖即可理解。



問題 2.3.7

這個問題的答案是 $y = \lfloor \log_2 x \rfloor + 1$ 。依如下過程可以導出。

步驟 1

某個整數 x 要在二進制中表示成 n 位數的條件是滿足 $2^{n-1} \leq x < 2^n$ 。具體例子如下：

- 成為 3 位數的條件是、 $2^2 = 4$ 以上且小於 $2^3 = 8$
- 成為 4 位數的條件是、 $2^3 = 8$ 以上且小於 $2^4 = 16$
- 成為 5 位數的條件是、 $2^4 = 16$ 以上且小於 $2^5 = 32$

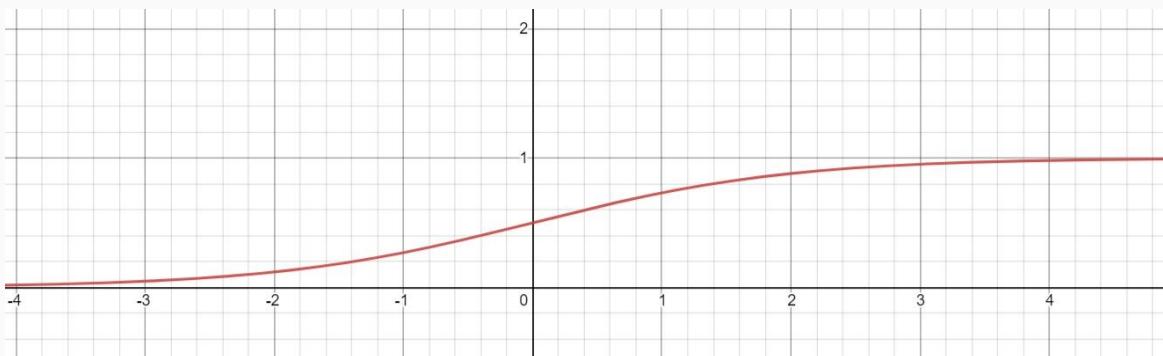
步驟 2

將步驟 1 換句話說，當 $n - 1 \leq \log_2 x < n$ 時，在二進制為 n 位數。

即，因為 $\lfloor \log_2 x \rfloor = n - 1$ ，所以在二進制的位數 n 是 $\lfloor \log_2 x \rfloor + 1$ 位。

問題 2.3.8

可以思考例如 $f(x) = 1/(1 + 2^{-x})$ 作為答案的例子。另外，有一個類似的函數是**S型函數**，在機器學習中經常使用，有興趣的人可以調查看看。



2.4

節末問題 2.4 的解答

問題 2.4.1

答案如下。另外，用大 O 記法表示的值，是透過「刪除最重要的項後，再去掉常數倍（如 $7N^2$ 中 7 的部分）」這樣的操作而求得的（→2.4.8項）。。。

1. $T_1(N) = O(N^3)$
2. $T_2(N) = O(N)$
3. $T_3(N) = O(2^N)$
4. $T_4(N) = O(N!)$

問題 2.4.2

這個程式執行二重迴圈，每個變數的取值如下：

- 變數 $i : 1, 2, 3, \dots, N$ 的 N 種值
- 變數 $j : 1, 2, 3, \dots, 100N$ 的 $100N$ 種值

因此，總迴圈次數為 $N \times 100N = 100N^2$ 次，也就是說計算複雜度是 $O(N^2)$ 。另外，迴圈次數用乘法表示的原因，將變數 i 和 j 的取值方式排列成一個矩形（→2.4.5項）。的話即可容易理解。



問題 2.4.3

為了確認 $\log_2 N$ 和 $\log_{10} N$ 之間只有常數倍的差異，我們可以將 $\log_2 N$ 除以 $\log_{10} N$ 。根據底數轉換公式（→**2.3.10項**），下式會成立。

$$\frac{\log_2 N}{\log_{10} N} = \frac{\log_2 N}{\log_2 N \div \log_2 10} = \log_2 10 \doteq 3.32$$

因此，可知 $\log_2 N$ 是 $\log_{10} N$ 的約 3.32 倍。這也是在用大 O 記法表示對數時，使用如 $O(\log N)$ 而省略底數表示的一個原因。

問題 2.4.4

答えは以下のようになります。なお、はと同じ意味です。答案如下。另外， $N \log N$ 的意思與 $N \times \log N$ 相同。

計算次數	$N \log N$	N^2	2^N
10^6 次以内	$N \leq 60000$	$N \leq 1000$	$N \leq 20$
10^7 次以内	$N \leq 500000$	$N \leq 3000$	$N \leq 23$
10^8 次以内	$N \leq 4000000$	$N \leq 10000$	$M \leq 26$
10^9 次以内	$N \leq 40000000$	$N \leq 30000$	$N \leq 30$

問題 2.4.5

由於 N 增加 2 時，執行時間大約增加 9 倍，因此計算複雜度應該是 $O(3^N)$ 。另外，由於 $O(N \times 3^N)$ 或 $O(10^{N/2})$ 也不會不自然，因此可以作為另一個解答。

N	14	16	18	20
実行時間	0.049 秒	0.447 秒	4.025 秒	36.189 秒

9.12 倍 9.00 倍 8.99 倍

問題 2.4.6

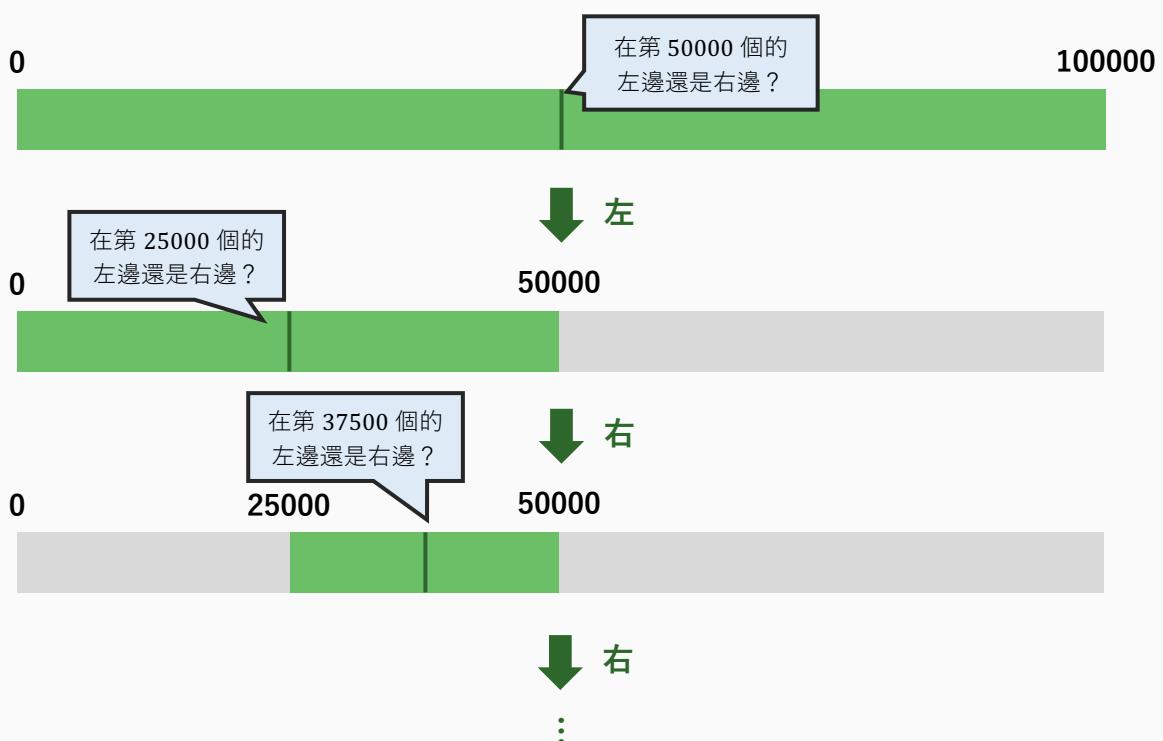
直覺的做法是以 “a” → “aardvark” → “aback” → “abalone” → “abandon” → 這樣的方式，從前面記載的單字開始一個一個搜尋。然而，當單字數為 N 的時候步驟數最差情況是 N 次，若 $N = 100000$ ，對人類來說幾乎不可能。因此，如果使用例如以下的方法，可以更有效率（→**2.4.7項**）。

重複進行「查看目前考慮範圍中間的單字，然後檢查在它之前還是之後」。

下圖顯示了當單字數為 100000 個時的步驟示意圖。

這是很類似二元搜尋法的方法，只需 $\lceil \log_2 N \rceil$ 個步驟即可找到目標單字。

另外，實作中，尋找例如「第 50000 個單字在哪裡」這樣的問題也很麻煩，因此，在一開始的問題中，與大致中央頁面的單字進行比較即可。大家在使用字典查單字時，不妨嘗試使用二元搜尋法。



2.5

節末問題 2.5 的解答

問題 2.5.1

這是測試對求和符號（→**2.5.9項**）的理解的問題。

答案如下：

$$\sum_{i=1}^{100} i = (1 + 2 + 3 + \dots + 100) = 5050$$
$$\sum_{i=1}^3 \sum_{j=1}^3 ij = (1 + 2 + 3 + 2 + 4 + 6 + 3 + 6 + 9) = 36$$

另外，從 1 到 100 的總和在第 1.1 節中也有提到，可以用和的公式（→**2.5.10項**）來計算出 $100 \times 101 \div 2 = 5050$ 。

問題 2.5.2

這是測試集合基本概念（→**2.5.5項**）理解的問題。答案如下：

1. $|S| = 3, |T| = 4$
2. $S \cup T = \{2, 3, 4, 7, 8, 9\}$ (任意一方包含的部分)
3. $S \cap T = \{2\}$ (兩者都包含的部分)
4. 非空的子集合有 $\{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\}, \{2, 4, 7\}$ 共 7 個。不懂的話，請回到第 58 頁確認。

問題 2.5.3

由於是階乘 $N! = 1 \times 2 \times 3 \times \dots \times N$ ，撰寫使用 for 敘述進行乘法運算的程式即可。此外，當 $N = 20$ 時， $N! = 2.4 \times 10^{18}$ ，注意如int型態等 32 位元的整數會發生溢出。（在以下的原始碼中，使用了 long long 型態來取代）

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

```

int main() {
    long long N;
    long long Answer = 1;
    cin >> N;
    for (int i = 2; i <= N; i++) Answer *= i; // 將 Answer 乘以 i
    cout << Answer << endl;
    return 0;
}

```

※ Python 等原始碼請參閱 chap2-5.md。

問題 2.5.4

撰寫如下的程式即可獲得正確答案。函數 `isprime(x)` 是用來判斷 2 以上的整數 x 是否為質數的函數，如果 x 是質數為 `true`，否則為 `false`。此外，如以下步驟逐個檢查，藉此判斷 x 是否為質數。

- x 能被 2 整除嗎？
- x 能被 3 整除嗎？
- :
- x 能被 $N - 1$ 整除嗎？

```

#include <iostream>
using namespace std;

bool isprime(int x) {
    for (int i = 2; i <= x - 1; i++) {
        // 將 x 除以 i 的餘數為 0 時、x 可以被 i 整除
        if (x % i == 0) return false;
    }
    return true;
}

int main() {
    int N, Answer = 0;
    cin >> N;
    for (int i = 2; i <= N; i++) {
        if (isprime(i) == true) cout << i << endl;
    }
    return 0;
}

```

※ Python 等原始碼請參閱 chap2-5.md。

問題 2.5.5

這個問題的答案是 **1000**。

最簡單的方法是計算所有包含在 $1 \leq a \leq 4, 1 \leq b \leq 4, 1 \leq c \leq 4$ 內的整數組合 (a, b, c) ，但這樣做會很繁瑣。

a = 1 時 合計 100			
1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

a = 2 時 合計 200			
2	4	6	8
4	8	12	16
6	12	18	24
8	16	24	32

a = 3 時 合計 300			
3	6	9	12
6	12	18	24
9	18	27	36
12	24	36	48

a = 4 時 合計 400			
4	8	12	16
8	16	24	32
12	24	36	48
16	32	48	64

所以，思考以下的雙重求和的值。總和為 **100**。

$$\sum_{b=1}^4 \sum_{c=1}^4 bc = 100$$

合計各 a 的 abc 的總和如下：

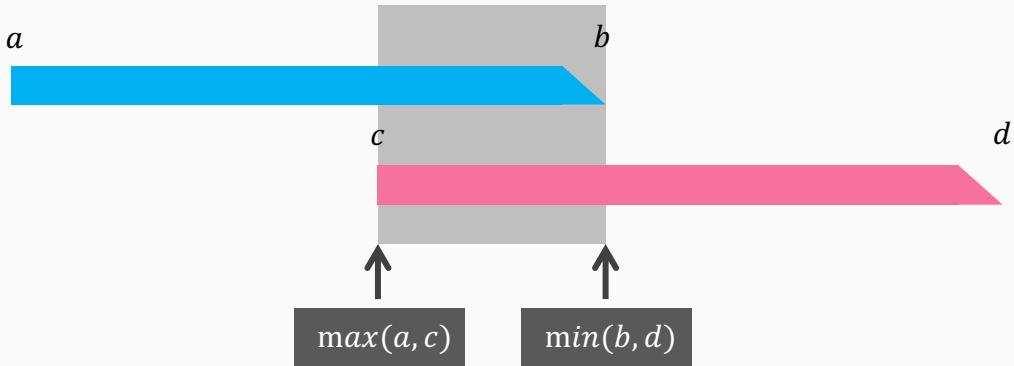
- 當 $a = 1$ 時， $abc (= 1 \times bc)$ 的總和： $1 \times 100 = 100$
- 當 $a = 2$ 時， $abc (= 2 \times bc)$ 的總和： $2 \times 100 = 200$
- 當 $a = 3$ 時， $abc (= 3 \times bc)$ 的總和： $3 \times 100 = 300$
- 當 $a = 4$ 時， $abc (= 4 \times bc)$ 的總和： $4 \times 100 = 400$

所求的三重求和為上述 4 項全部相加的值 **1000**。

問題 2.5.6

擁有共通部分的充要條件是 $\max(a, c) < \min(b, d)$ 。

不懂 \max 函數和 \min 函數的人可以回到第 2.3.2 項確認。



問題 2.5.7

每個 i 的值的「 cnt 增加次數」如下：。

- $i=1$ 時 : $2 \leq j \leq N$ ， 故增加次數為 $N - 1$ 次
- $i=2$ 時 : $3 \leq j \leq N$ ， 故增加次數為 $N - 2$ 次
- $i=3$ 時 : $4 \leq j \leq N$ ， 故增加次數為 $N - 3$ 次
- ⋮
- $i=N-1$ 時 : 1 次
- $i=N$ 時 : 0 次

因此，依和的公式（→2.5.10項），執行結束時的 cnt 值為：

$$(N - 1) + (N - 2) + \dots + 1 + 0 = \frac{(N - 1) \times N}{2} \left(= \frac{1}{2}N^2 - \frac{1}{2}N \right)$$

cnt 的值中最重要的項為 $(1/2) \times N^2$ ，所以，這個程式的計算複雜度是 $O(N^2)$ 。

（→2.4.8項）

第 3 章

基本的演算法

3.1

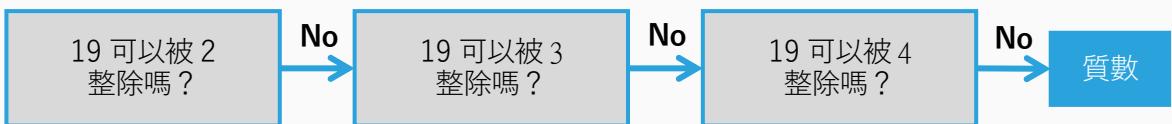
節末問題 3.1 的解答

問題 3.1.1

對於自己的年齡 N 為 2 歲以上的情況，可以進行如下判斷：

- 若無法被 2 以上 \sqrt{N} 以下的整數整除時：質數
- 若無法被整除：合數

例如 19 歲的情況，根據 $\sqrt{19} = 4.358 \dots$ ，若可以被 2, 3, 4 整除即為質數，但因為都無法被整除，所以 19 為質數。



問題 3.1.2

首先，當將自然數 N 進行質因數分解時，超過 \sqrt{N} 的質因數最多只有一個。這可以用反證法（→3.1.3項）如下證明。

假設將自然數 N 進行質因數分解時，超過 \sqrt{N} 的質因數有 2 個以上。即，假設用超過 N 的整數 A 和 B 來進行質因數分解，如下所示：

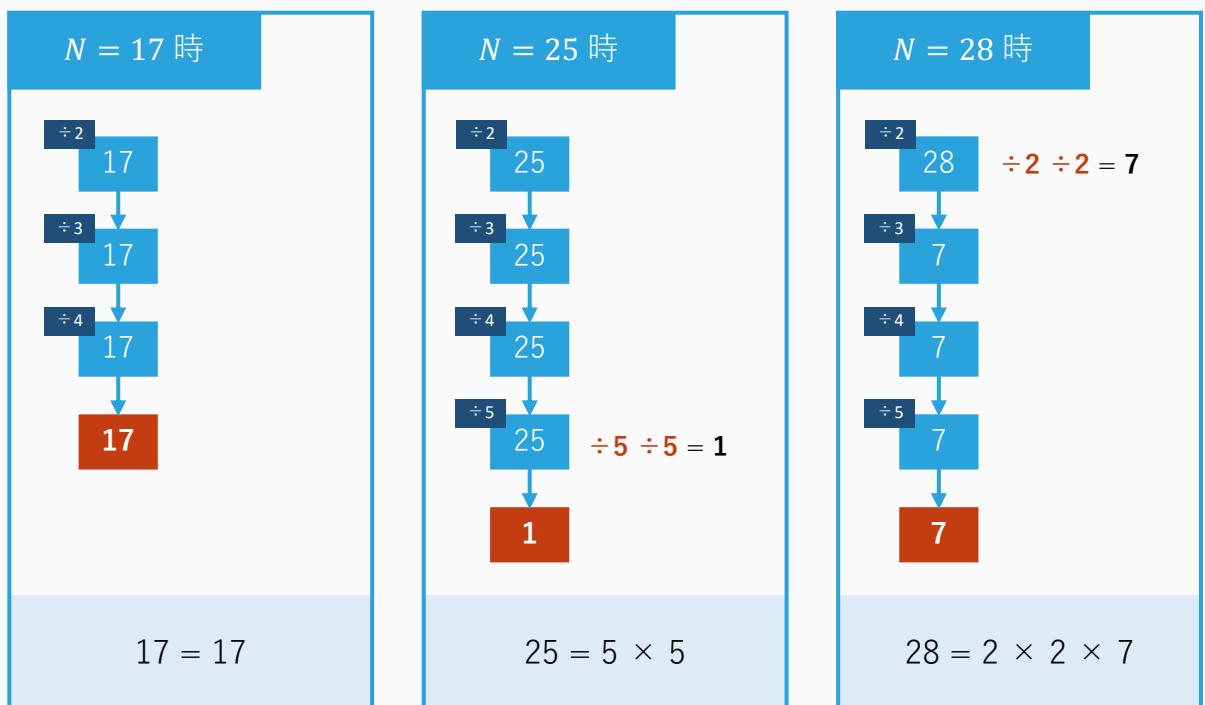
$$N = \bigcirc \times \bigcirc \times \cdots \times \bigcirc \times A \times B$$

然而，因為 $A \times B > N$ 而產生矛盾。因此，超過 \sqrt{N} 的質因數最多只有一個。

因此，可以用以下的演算法來進行質因數分解：

- 將 N 盡可能多次地除以 2。
- 將 N 盡可能多次地除以 3。
- 對 $4, 5, \dots, [\sqrt{N}]$ 進行同樣的操作。。
- 最後，若剩下的 N 不為 1 時，則將其加入質因數。

例如，於 $N = 17, 25, 28$ 的情況，演算法將如下圖所示運作。注意可能會有同一數字被多次除的情況。



C++ 的實作例如下。

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

int main() {
    // 輸入
    long long N;
    cin >> N;

    // 質因數分解、輸出
    for (long long i = 2; i * i <= N; i++) {
        while (N % i == 0) {
            N /= i;
            cout << i << endl;
        }
    }
    if (N >= 2) cout << N << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap3-1.md 。

3.2

節末問題 3.2 的解答

問題 3.2.1

答案如下所示。不懂的人可以回到 **3.2.2項** 確認。。

步驟數	0	1	2	3	4	5	6
A 的值	372	372	104	104	14	14	0
B 的值	506	134	134	30	30	2	2

問題 3.2.2

整數 A_1, A_2, \dots, A_N 的最大公因數可以如下計算 (\rightarrow **3.2.5項**)

- 首先計算 A_1 和 A_2 的最大公因數。
- 接著，計算上一步的計算結果與 A_3 的最大公因數。。
- ：
- 接著，計算上一步的計算結果與 A_N 的最大公因數。

將這些步驟進行實作如下。函式 $\text{GCD}(A, B)$ 用於計算 A 和 B 的最大公因數。另外，變數 r 表示前一步的計算結果。

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

long long GCD(long long A, long long B) {
    while (A >= 1 && B >= 1) {
        if (A < B) B = B % A; // A < B 時，改寫較大的數 B
        else A = A % B; // A >= B 時，改寫較大的數 A
    }
    if (A >= 1) return A;
    return B;
}

long long N;
long long A[100009];

int main() {
```

下一頁

```

// 輸入
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i];

// 求出答案
long long R = GCD(A[1], A[2]);
for (int i = 3; i <= N; i++) {
    R = GCD(R, A[i]);
}

// 輸出
cout << R << endl;
return 0;
}

```

※ Python等原始碼請參閱 chap3-2.md。

問題 3.2.3

整數 A_1, A_2, \dots, A_N 的最小公倍數可以如下計算。

- 首先計算 A_1 和 A_2 的最小公倍數。
- 接著，計算上一步的計算結果與 A_3 的最小公倍數。。
- ：
- 接著，計算上一步的計算結果與 A_N 的最小公倍數。。

此外，2 個數 A 和 B 具有以下性質（→2.5.2 項）

$$A \times B = (\text{A 和 B 的最大公因數}) \times (\text{A 和 B 的最小公倍數})$$

$$\text{即 } (\text{A 和 B 的最小公倍數}) = \frac{A \times B}{(\text{A 和 B 的最大公因數})}$$

因此，撰寫如下程式可以得出正確答案。又，函式 $\text{LCM}(A, B)$ 用於計算 A 和 B 的最小公倍數。

```

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

long long GCD(long long A, long long B) {
    while (A >= 1 && B >= 1) {
        if (A < B) B = B % A; // A < B 時，改寫較大的數 B
        else A = A % B; // A >= B 時，改寫較大的數 A
    }
    return A;
}

long long LCM(long long A, long long B) {
    return (A * B) / GCD(A, B);
}

```

下一頁

```
        }
        if (A >= 1) return A;
        return B;
    }

long long LCM(long long A, long long B) {
    return (A / GCD(A, B)) * B;
}

long long N;
long long A[100009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 求出答案
    long long R = LCM(A[1], A[2]);
    for (int i = 3; i <= N; i++) {
        R = LCM(R, A[i]);
    }

    // 輸出
    cout << R << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap3-2.md。

3.3

節末問題 3.3 的解答

問題 3.3.1

這是測試對情況數的公式（→3.3.4項、3.3.5項）的理解的問題。答案如下：

$${}_2C_1 = \frac{2}{1} = 2$$

$${}_8C_5 = \frac{8 \times 7 \times 6 \times 5 \times 4}{5 \times 4 \times 3 \times 2 \times 1} = 56$$

$${}_7P_2 = 7 \times 6 = 42$$

$${}_{10}P_3 = 10 \times 9 \times 8 = 720$$

此外，二項係數 nCr 是 nPr 的 $1/r!$ 倍，因此可以計算下式。

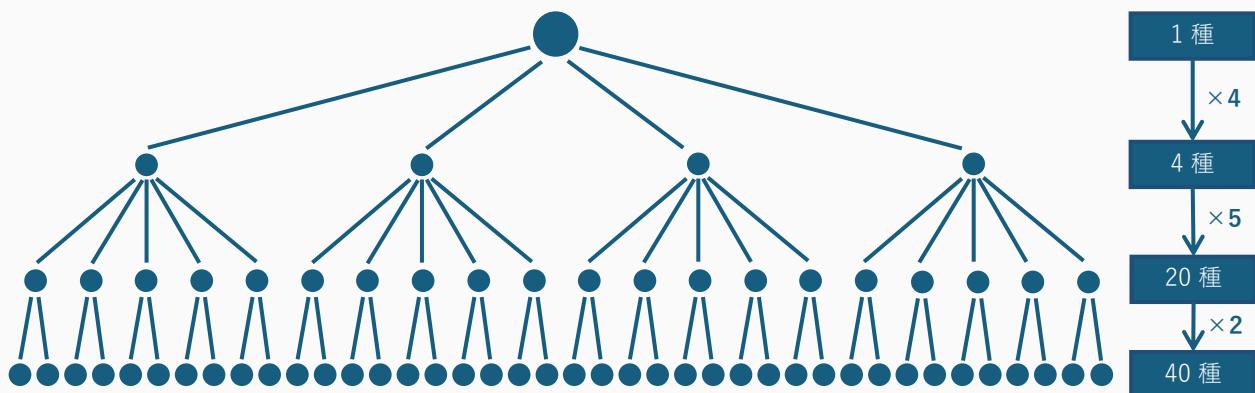
$$nCr = \frac{nPr}{r!} = \frac{n \times (n-1) \times (n-2) \times (n-r+1)}{r \times (r-1) \times (r-2) \times \cdots \times 1}$$

問題 3.3.2

這是測試對乘法原理（→3.3.2項）的理解的問題。

- 大小的選擇方式：4 種
- 配料的選擇方式：5 種
- 名牌的選擇方式：2 種

因此，答案是 $4 \times 5 \times 2 = 40$ 種。



問題 3.3.3

首先，計算以下的值。（計算 $n!$ 的方法：→[節末問題 2.5.3](#)）

- Fact_n : $n!$ 的值
- Fact_r : $r!$ 的值
- Fact_nr : $(n - r)!$ 的值

此時，所求的 nCr 的是 $\text{Fact_n} / (\text{Fact_r} * \text{Fact_nr})$ ，因此，撰寫輸出該值的程式即為正解。C++ 的實作例如下。

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

long long n, r;
long long Fact_n = 1;
long long Fact_r = 1;
long long Fact_nr = 1;

int main() {
    // 輸入
    cin >> n >> r;

    // 階乘的計算
    for (int i = 1; i <= n; i++) Fact_n *= i;
    for (int i = 1; i <= r; i++) Fact_r *= i;
    for (int i = 1; i <= n - r; i++) Fact_nr *= i;

    // 輸出
    cout << Fact_n / (Fact_r * Fact_nr) << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap3-3.md。

問題 3.3.4

如書中所解釋的實作即可。以下是 C++ 的解答範例。本問題的限制條件為 $N \leq 200000$ 之大，答案有可能會超過 10^{10} 種。注意如 int 型態等的 32 位元整數會溢出。（在 Python 不會有這個問題）

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

long long N;
```

下一頁

```

long long A[200009];
long long a = 0, b = 0, c = 0, d = 0; // 為了避免溢出，使用 64 位元的整數

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 計數 a、b、c、d 的個數
    for (int i = 1; i <= N; i++) {
        if (A[i] == 100) a += 1;
        if (A[i] == 200) b += 1;
        if (A[i] == 300) c += 1;
        if (A[i] == 400) d += 1;
    }

    // 輸出（答案為 a * d + b * c）
    cout << a * d + b * c << endl;
    return 0;
}

```

※ Python等原始碼請參閱 chap3-3.md。

問題 3.3.5

如書中所解釋的實作即可。以下是 C++ 的解答範例。本問題的限制條件為 $N \leq 500000$ 之大，答案有可能會超過 10^{11} 種。

注意如 `int` 型態等的32位元整數會溢出，因此建議使用 `long long` 型態等64位元整數。（在Python不會有這個問題）

```

#include <iostream>
using namespace std;

long long N;
long long A[500009];
long long x = 0, y = 0, z = 0;

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 計數 a、b、c、d 的個數
    for (int i = 1; i <= N; i++) {
        if (A[i] == 1) x += 1;
        if (A[i] == 2) y += 1;
        if (A[i] == 3) z += 1;
    }
}

```

下一頁

```

    if (A[i] == 3) z += 1;
}

// 輸出
cout << x * (x - 1) / 2 + y * (y - 1) / 2 + z * (z - 1) / 2 << endl;
return 0;
}

```

※ Python 等原始碼請參閱 chap3-3.md。

問題 3.3.6

首先，令 A_1, A_2, \dots, A_N 中 i 的個數為 $\text{cnt}[i]$ ，則 $\text{cnt}[1], \text{cnt}[2], \dots, \text{cnt}[99999]$ 可以如下計數。。

```

// 將陣列 cnt[i] 初始化
for (int i = 1; i <= N; i++) cnt[i] = 0;

// 出現 A[i] 的話，於 cnt[A[i]] 加上1
for (int i = 1; i <= N; i++) cnt[A[i]] += 1;

```

因此，和為 100000 的 2 張卡片的選擇方法可以列舉如下：

- 選擇 1 和 99999 的卡片（有 $\text{cnt}[1]*\text{cnt}[99999]$ 種方法）
- 選擇 2 和 99998 的卡片（有 $\text{cnt}[2]*\text{cnt}[99998]$ 種方法）
- 選擇 3 和 99997 的卡片（有 $\text{cnt}[3]*\text{cnt}[99997]$ 種方法）
- :
- 選擇 49999 和 50001 的卡片（有 $\text{cnt}[49999]*\text{cnt}[50001]$ 種方法）
- 選擇兩張 50000 的卡片（有 $\text{cnt}[50000]*(\text{cnt}[50000]-1)/2$ 種方法）

注意當選擇兩張 50000 的卡片時，不是 $\text{cnt}[50000]*\text{cnt}[50000]$ 種方法。

因此，撰寫將紅色標記值的總和輸出的程式，即可得到正確答案。以下是 C++ 的解答範例。。

```

#include <iostream>
using namespace std;

long long N, A[200009];
long long cnt[100009];
long long Answer = 0;

int main() {

```

下一頁

```

// 輸入
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i];

// 計數 cnt[1], cnt[2], ..., cnt[99999]
for (int i = 1; i <= 99999; i++) cnt[i] = 0;
for (int i = 1; i <= N; i++) cnt[A[i]] += 1;

// 求出答案
for (int i = 1; i <= 49999; i++) {
    Answer += cnt[i] * cnt[100000 - i];
}
Answer += cnt[50000] * (cnt[50000] - 1) / 2;

// 輸出
cout << Answer << endl;
return 0;
}

```

※ Python 等原始碼請參閱 chap3-3.md。

問題 3.3.7

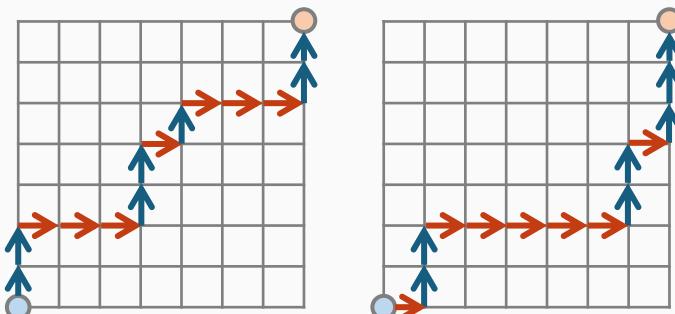
首先，從起點到終點以最短距離（14 步）前進的充要條件（→**2.5.6項**）如下：

「向上移動 1 步」和「向右移動 1 步」各進行 7 次。除此之外，不進行其他移動。

因此，答案是從 14 步中選擇 7 步向上移動的情況數如下。

$${}_{14}C_7 = \frac{14!}{7! \times 7!} = 3432 \text{ 種}$$

直接手算比較麻煩，但將節末問題 3.3.3 的程式中 $n = 14$ 和 $r = 7$ 代入，便可簡單得到答案。



向上的移動 7 次

向右的移動 7 次

3.4

節末問題 3.4 的解答

問題 3.4.1

21 個模式不一定會等機率發生。事實上是：

- 出現 (1, 1) 的機率是 $1/36$
- 出現 (1, 2) 的機率是 $1/18$

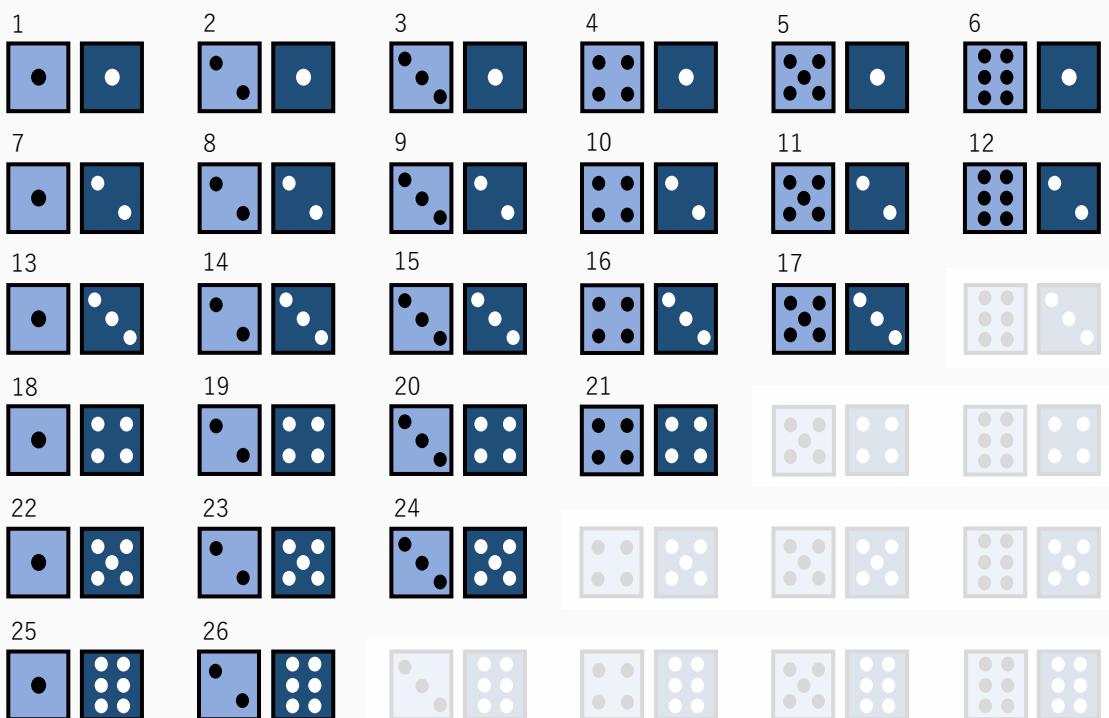
由於這些機率不同，不能單純地認為「21 個模式中有 15 個模式的點數和為 8 以下」，所以所求的機率是 $15/21$ 」。

另外，用 \star 表示的部分可以理解如下。

。 擲兩個骰子時，出現 (1, 2) 點數的模式有

- 第一次擲出的是 1，第二次擲出的是 2
- 第二次擲出的是 2，第一次擲出的是 1

這 2 種情況。由於在 36 種情況中佔 2 種，機率是 $2/36 = 1/18$ 。



第一次擲的骰子



第二次擲的骰子

問題 3.4.2

將期望值公式（→**3.4.2項**）代入，答案如下：

$$\begin{aligned} & \left(1000000 \times \frac{1}{10000}\right) + \left(100000 \times \frac{9}{10000}\right) + \left(10000 \times \frac{9}{1000}\right) + \left(1000 \times \frac{9}{100}\right) + \left(0 \times \frac{9}{10}\right) \\ &= 100 + 90 + 90 + 90 \\ &= 370 \end{aligned}$$

由於獲得獎金的期望值是 370 日圓，當參加費是 500 日圓的情況下是虧損的。

問題 3.4.3

首先，由於期望值的線性關係（→**3.4.3項**），以下關係成立。

(學習時間的期望值總和)

$$= (\text{第 } 1 \text{ 天學習時間的期望值}) + \cdots + (\text{第 } N \text{ 天學習時間的期望值})$$

因此，每天的學習時間的期望值如下：

- 第 1 天： $(A_1 \times 1/3) + (B_1 \times 2/3)$
- 第 2 天： $(A_2 \times 1/3) + (B_2 \times 2/3)$
- 第 3 天： $(A_3 \times 1/3) + (B_3 \times 2/3)$
- \vdots
- 第 N 天： $(A_N \times 1/3) + (B_N \times 2/3)$

因此，所求的總學習時間的期望值如下：

$$\left(A_1 \times \frac{1}{3} + B_1 \times \frac{2}{3}\right) + \left(A_2 \times \frac{1}{3} + B_2 \times \frac{2}{3}\right) + \cdots + \left(A_N \times \frac{1}{3} + B_N \times \frac{2}{3}\right)$$

撰寫將此值輸出的程式即為正解。以下是 C++ 的解答範例。

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

int N;
double A[109], B[109];
double Answer = 0.0;

int main() {
```

下一頁

```

// 輸入
cin >> N;
for (int i = 1; i <= N; i++) cin >> A[i] >> B[i];

// 求出期望值
for (int i = 1; i <= N; i++) {
    double eval = A[i] * (1.0 / 3.0) + B[i] * (2.0 / 3.0);
    Answer += eval;
}

// 輸出
printf("%.12lf\n", Answer);
return 0;
}

```

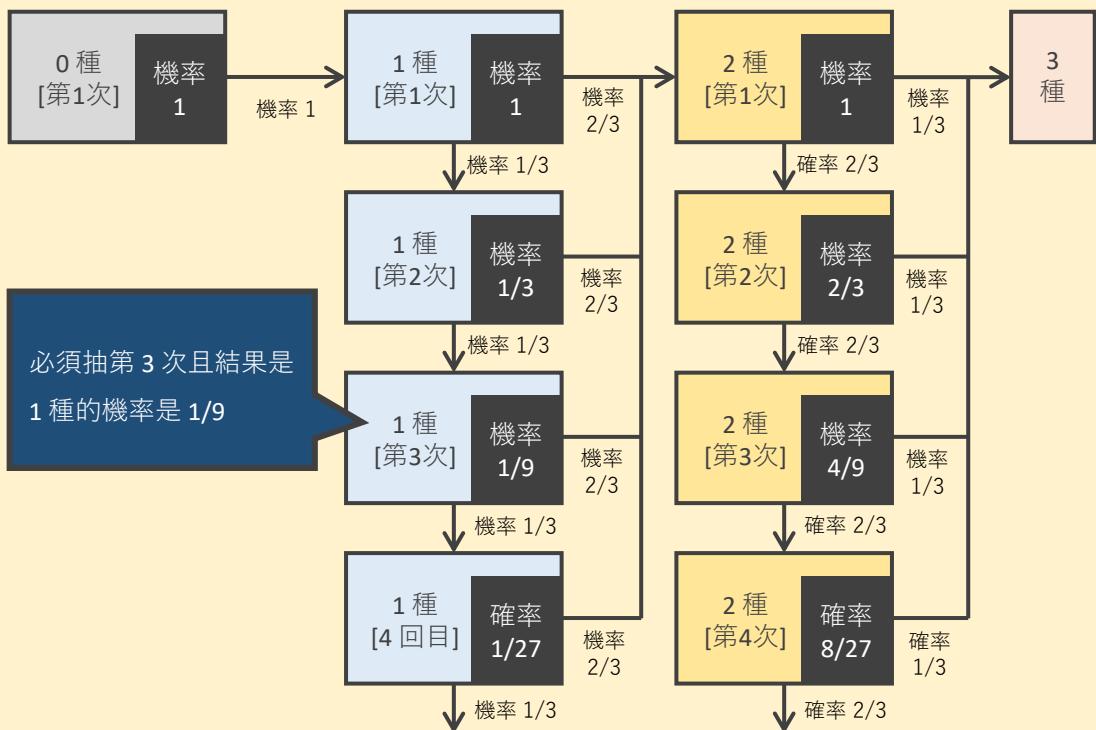
※ Python 等原始碼請參閱 chap3-4.md。

問題 3.4.4

此問題的設定很複雜且難度較高，因此先思考 $N = 3$ 的情況。

首先，表示收集到所有硬幣的過程的圖如下。

- 實線表示獲得了尚未收集過的硬幣種類
- 虛線表示抽到了已經收集過的硬幣種類



因此，由於期望值的線性關係，收集所有種類所需次數的期望值是以下總和。

- 從 0 種變成 1 種所需次數的期望值 … (1)
- 從 1 種變成 2 種所需次數的期望值 … (2)
- 從 2 種變成 3 種所需次數的期望值 … (3)

根據前一頁的圖及和的公式 (→**2.5.10 項の最後**) :

- (1) 的期望值是 1 次
- (2) 的期望值是 $1 + (1/3) + (1/9) + \dots = 3/2$ 次
- (3) 的期望值是 $1 + (2/3) + (4/9) + \dots = 3$ 次

因此，當 $N = 3$ 時，答案是 $1 + 3/2 + 3 = \mathbf{11/2}$ 回 次。。

接下來，思考一般 N 的情況。由於期望值的線性關係，所求的答案是以下值的總和。

- 從 0 種變成 1 種所需次數的期望值
- 從 1 種變成 2 種所需次數的期望值
- :
- 從 $N - 1$ 種變成 N 種所需次數的期望值

因此，從已經收集到 r 種硬幣的狀態到收集到第 $r + 1$ 種硬幣時，拿到已經收集過的硬幣的機率是 r/N ，所以：

- 需要 1 次以上的機率：1
- 需要 2 次以上的機率： $(r/N)^1$
- 需要 3 次以上的機率： $(r/N)^2$
- 需要 4 次以上的機率： $(r/N)^3$ [以下略]

因此，次數的期望值如下：

$$1 + \left(\frac{r}{N}\right)^1 + \left(\frac{r}{N}\right)^2 + \left(\frac{r}{N}\right)^3 + \dots = \frac{N}{N-r}$$

最後，總次數的期望值是對 $r = 0, 1, 2, \dots, N - 1$ 將紅色部分相加如下。

$$\frac{N}{N} + \frac{N}{N-1} + \frac{N}{N-2} + \dots + \frac{N}{2} + \frac{N}{1}$$

因此，撰寫如下將此值輸出的程式即為正解。

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

int N;
double Answer = 0;

int main() {
    // 輸入
    cin >> N;

    // 求出期望值
    for (int i = N; i >= 1; i--) {
        Answer += 1.0 * N / i;
    }

    // 輸出
    printf("%.12lf\n", Answer);
    return 0;
}
```

※ Python等原始碼請參閱 chap3-4.md。

3.5

節末問題 3.5 の解答

問題 3.5.1 (1), (2)

正面出現的機率為 $p = 0.5$ ，試驗次數為 $n = 10000$ ，因此代入3.5.6項提到的公式後，
10000次中正面出現的比例分布近似於如下的常態分布。

- 平均： $p = 0.5$
- 標準差： $\sqrt{p(1-p)/n} = \sqrt{0.5 \times (1 - 0.5) \div 10000} = 0.005$

若換算成次數，平均 $\mu = 5000$ 次，標準差 $\sigma = 50$ 次。

因此， $\mu - 2\sigma = 4900$, $\mu + 2\sigma = 5100$ ，這代表次數為 4900 次以上 5100 次以下的機率約 95%（68 – 95 – 99.7 法則：→ 3.5.5項）。此外，也可以使用以下公式直接計算平均值和標準差。

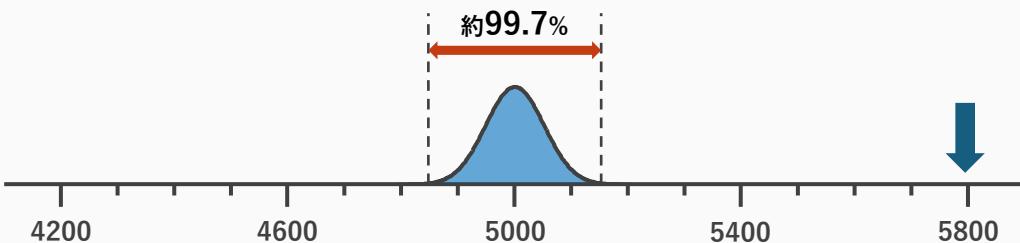
以試驗成功的機率 p 進行 n 次試驗時，成功的次數的分布可以近似為平均值 $\mu = np$ ，標準差 $\sigma = \sqrt{np(1-p)}$ 的常態分布。

問題 3.5.1 (3)

根據(1)和(2)的結果，正面出現的次數約99.7%的機率在以下範圍內：

- $\mu - 3\sigma = 5000 - 150 = 4850$ 次以上
- $\mu + 3\sigma = 5000 + 150 = 5150$ 次以上

5800 次大幅超出了這個範圍，因此可以認為這枚硬幣出現機率**不是 50% 的可能性很高**。



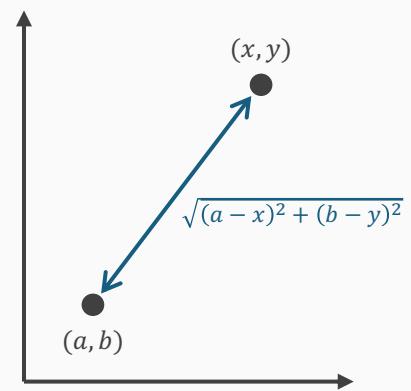
問題 3.5.2 (1)

撰寫以下程式，可判斷隨機放置的 100 萬個點中有多少點落在兩個圓中的至少一個內。

例如，在作者的環境中，這個程式輸出為 719653[※]。

另外，座標 (a, b) 和座標 (x, y) 之間的距離為

$\sqrt{(a - x)^2 + (b - y)^2}$ ，在 4.1 節也會詳細解釋。



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

int main() {
    int N = 1000000;
    int M = 0;

    for (int i = 1; i <= N; i++) {
        double px = 6.0 * rand() / (double)RAND_MAX;
        double py = 9.0 * rand() / (double)RAND_MAX;

        // 與點 (3, 3) 之間的距離。若此值為 3 以下，則會被包含在半徑 3 的圓內。。
        double dist_33 = sqrt((px - 3.0) * (px - 3.0) + (py - 3.0) * (py - 3.0));

        // 與點 (3, 7) 之間的距離。若此值為 2 以下，則會被包含在半徑 2 的圓內。
        double dist_37 = sqrt((px - 3.0) * (px - 3.0) + (py - 7.0) * (py - 7.0));

        // 條件分岐
        if (dist_33 <= 3.0 || dist_37 <= 2.0) M += 1;
    }

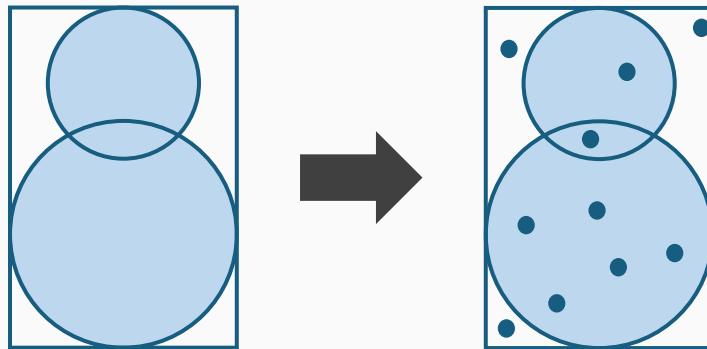
    // 輸出 N 次中有多少次進入表中
    cout << M << endl;
    return 0;
}
```

※ 雖然在作者環境中是 719653 次，但落在 718000~721000 次的範圍內即可。

※ Python 等原始碼請參閱 chap3-5.md。

問題 3.5.2 (2)

隨機放置點的區域 ($0 \leq x \leq 6, 0 \leq y \leq 9$) 的面積為 $6 \times 9 = 54$ ，因此，當令落在圖中藍色區域的點的比例為 p 時，藍色區域的面積可以近似為 $54 \times p$ 。



利用這個方法來計算面積吧。例如，使用作者環境中的結果，計算出 $54 \times 719653 \div 1000000 = 38.861262$ 。實際的值約為 **38.850912677 ...**，因此誤差為 0.02 以下。

3.6

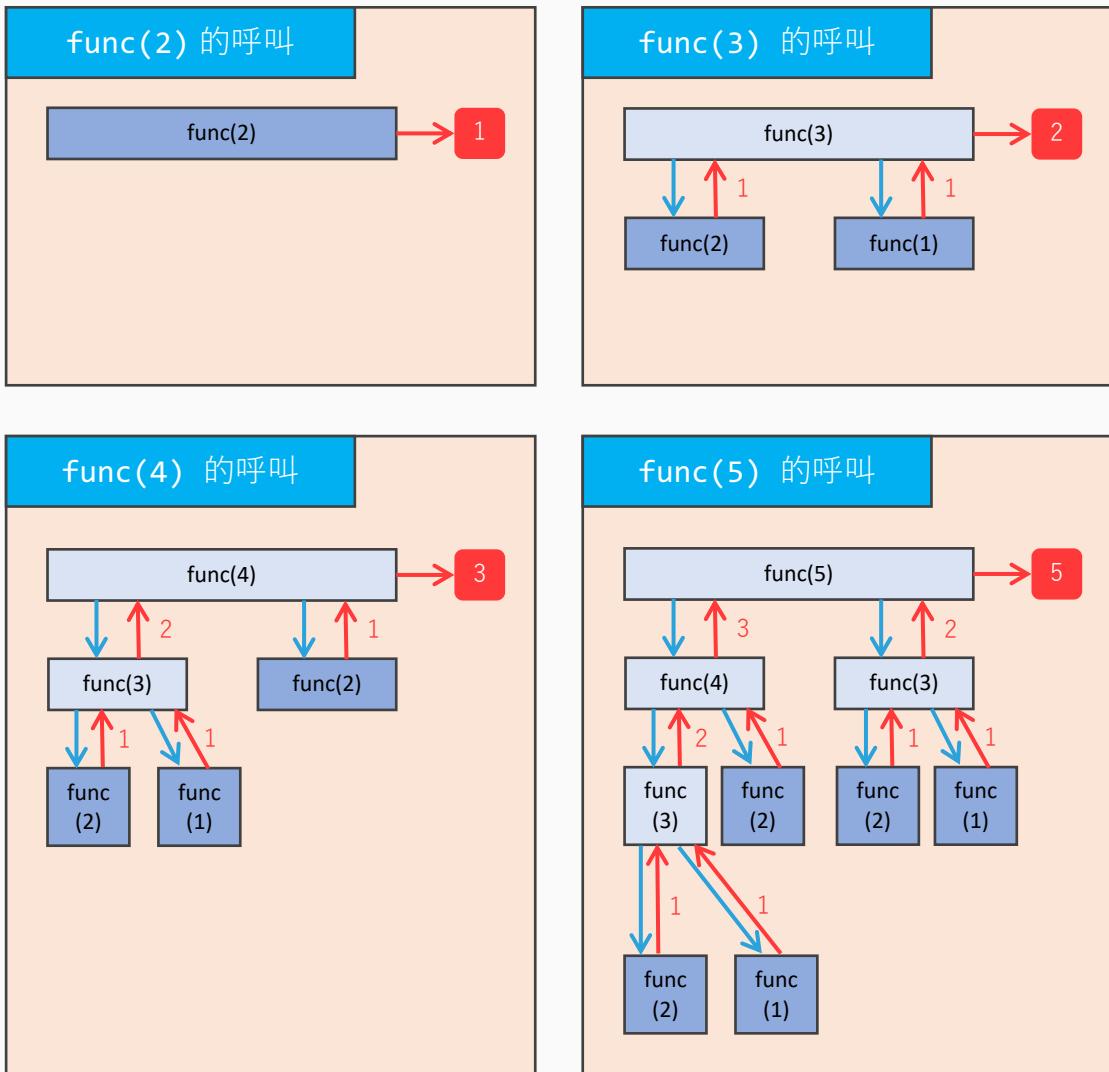
節末問題 3.6 的解答

問題 3.6.1

答案如下。特別注意在 `func(2)` 中，會在最開始的條件分歧 ($N \leq 2$) 就回傳值。

函數的呼叫	<code>func(2)</code>	<code>func(3)</code>	<code>func(4)</code>	<code>func(5)</code>
答案	1	2	3	5

遞迴呼叫的示意圖如下所示。函數 `func(N)` 會回傳費波那契數（→3.7.2項）的第 N 項。

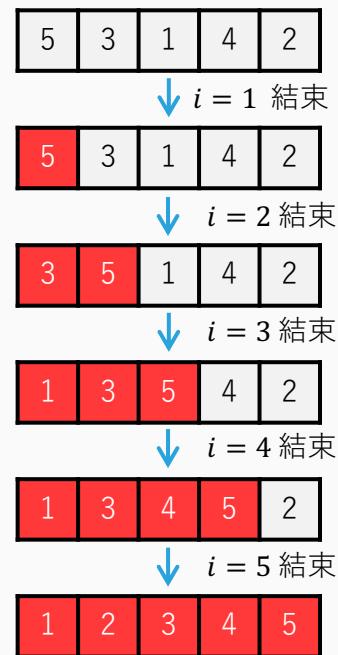


問題 3.6.2

在這個排序演算法中，以下事實 A 成立。右圖顯示 $A = [5, 3, 1, 4, 2]$ 的例子。

當 $i = I$ 的迴圈結束時，

$A[1] \leq A[2] \leq A[3] \leq \dots \leq A[I]$ 成立。



這可以證明如下。

另外，「假設在 $i = I - 1$ 時滿足條件，證明在 $i = I$ 時也滿足條件」的證明方法稱為**數學歸納法**。

欲證明的事實（事實 B）

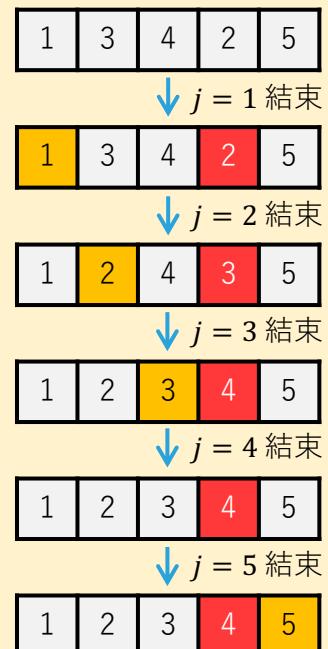
在 $i = I - 1$ 的迴圈結束時 $A[1] \leq A[2] \leq \dots \leq A[I - 1]$ ，則進行 $i = I$ 復圈時會滿足 $A[1] \leq A[2] \leq \dots \leq A[I]$ 。

事實 B 的證明

在 $i = I - 1$ 時，由於當 $A[t - 1] \leq A[I] \leq A[t]$ 時：

- 對 $j = 1, \dots, t - 1$ ，不會進行交換。
- 當 $j = t$ 時首次進行交換。
- 對 $j = t + 1, \dots, I - 1$ ，也會進行交換。
- 在這之後： $A[I]$ 的值可能會增加，但不會減少。

因此，最終 $A[1] \leq A[2] \leq \dots \leq A[I]$ 會成立。右圖為一個例子 ($I = 4$ 的例子)。



當事實 B 可以證明之後？

顯然，當 $i = 1$ 時，事實 A 成立。另外，根據事實 B，當 $i = 2$ 時，事實 A 也成立。重複運用事實 B，可知當 $i = N$ 時，事實 A 也成立（操作結束時排序完成）。

問題 3.6.3

首先，由於當列 B' 為空時，會將列 A' 最左邊的元素 $A[c1]$ 取出，因此撰寫如下程式即可。注意在取出元素後，列 A' 最左邊的位置 $c1$ 會增加 1。

```
else if (c2 == r) {  
    // 列  $B'$  為空時  
    C[cnt] = A[c1]; c1++;  
}
```

其次，當列 A' 和列 B' 都不為空時，可以如下分類：

- 列 A' 的左端 $A[c1]$ 小於列 B' 的左端 $A[c2]$ ：取出 $A[c1]$
- 列 B' 的左端 $A[c2]$ 小於列 A' 的左端 $A[c1]$ ：取出 $A[c2]$

將其編寫成程式如下。

```
else if {  
    // 列  $B'$  為空時  
    if (A[c1] <= A[c2]) {  
        C[cnt] = A[c1]; c1++;  
    }  
    else {  
        C[cnt] = A[c2]; c2++;  
    }  
}
```

關於完整的 C++ 程式、Python、JAVA、C 的解答例，請參閱 chap3-6.md。

3.7

節末問題 3.7 的解答

問題 3.7.1

答案如下。

如果不理解，可以回到3.7.1項～3.7.3項進行確認。

元素	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
值	1	1	1	3	5	9	17	31	57	105

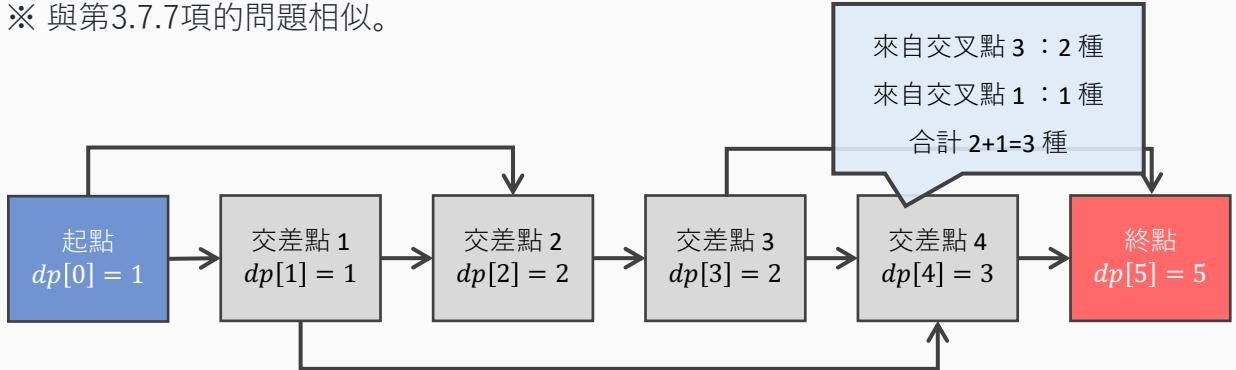
問題 3.7.2

答案是 5 種。

令 $dp[i]$ = (行進到交叉點 i 為止的方法數) 來進行動態規劃法，可以得到答案。

其中，將起點設為交叉點 0，終點設為交叉點5。

※ 與第3.7.7項的問題相似。

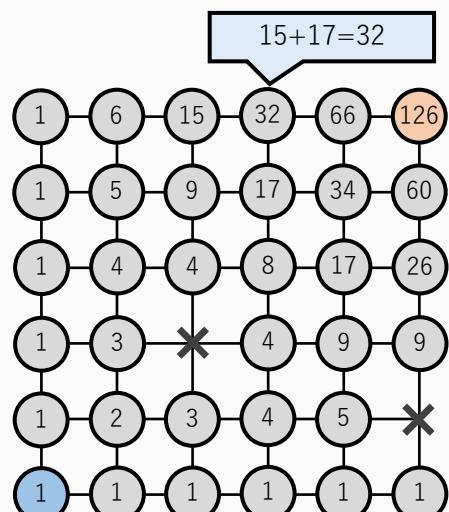


問題 3.7.3

答案是 126 種。與問題3.7.2用相同的方針進行動態規
劃法即可。

另外，注意從起點到終點以最短路徑（10步）移動時，
只能向上和向右移動。

朝從左開始的第 i 列、從下開始第 j 行的格子 (i, j) 移
動時，前一個格子為 $(i - 1, j)$ 或 $(i, j - 1)$ 。



問題 3.7.4

部分和問題可以用類似於背包問題（→3.7.8項）的以下方法來解決。

準備的陣列（二維陣列）

$dp[i][j]$ ：從左開始到第 i 個卡片（以下稱為卡 i ）之中，如果存在總和為 j 的組合，則為 `true`，否則為 `false`。

動態規劃法的轉換 ($i = 0$)

顯然只有「什麼都不選」這種方法，因此：

- $dp[0][j] = \text{true} (j = 0)$
- $dp[0][j] = \text{false} (j \neq 0)$

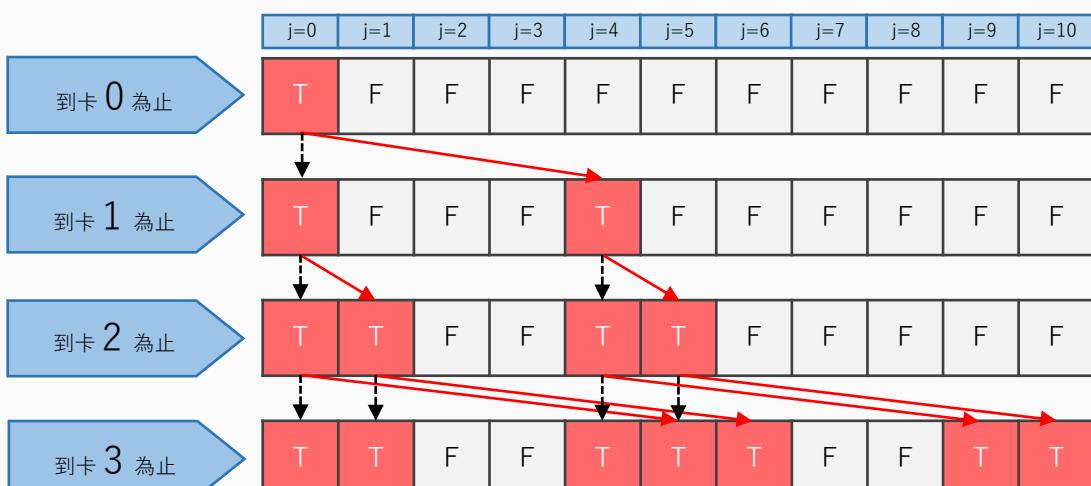
動態規劃的轉換（以 $i = 1, 2, \dots, N$ 的順序計算）

到卡 i 為止，從中選擇以使總和為 j 的方法有以下兩種。（以最後的行動 [是否選擇卡 i] 來區分）

- 卡 $i - 1$ 為止的總和為 $j - A_i$ 選擇卡 i
- 卡 $i - 1$ 為止的總和為 j 不選擇卡 i

因此， $dp[i - 1][j - A_i], dp[i - 1][j]$ 之中至少一個為 `true` 的時候， $dp[i][j] = \text{true}$ ，否則為 `false`。

例如， $N = 3, (A_1, A_2, A_3) = (4, 1, 5)$ 時，陣列 dp 如下所示。在此，當 $dp[N][S] = \text{true}$ 時，存在總和為 S 的選擇方法。



這個解法用C++實作如下。注意與背包問題的程式碼3.7.3不同，陣列 dp 為bool型態。

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

int N, S, A[69];
bool dp[69][10009];

int main() {
    // 輸入
    cin >> N >> S;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 陣列的初始化
    dp[0][0] = true;
    for (int i = 1; i <= S; i++) dp[0][i] = false;

    // 動態規劃法
    for (int i = 1; i <= N; i++) {
        for (int j = 0; j <= S; j++) {
            // j < A[i] 時，無法選擇卡 i
            if (j < A[i]) dp[i][j] = dp[i-1][j];
            // j >= A[i] 時，有選擇 / 不選擇 兩種選項
            if (j >= A[i]) {
                if (dp[i-1][j] == true || dp[i-1][j-A[i]] == true) dp[i][j] = true;
                else dp[i][j] = false;
            }
        }
    }

    // 輸出答案
    if (dp[N][S] == true) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap3-7.md。

問題 3.7.5

如下述，可以將第1.1.4項的問題歸納為背包問題。

- 重量：物品的價格
- 價值：物品的卡路里
- 重量上限：500日元

問題 3.7.6

這個問題可以用以下方法解決。準備 2 個一維陣列，從第 1 天開始按順序進行動態規劃法處理。

準備的陣列（二維陣列）

$dp1[i]$ ：當第 i 天學習時，到目前為止實力提升的最大值

$dp2[i]$ ：最大當第 i 天不學習時，到目前為止實力提升的最大值

動態規劃法的轉換 ($i = 0$)

由於從第 1 天開始才可以學習，設置 $dp1[0] = 0, dp2[0] = 0$ 等適當的值即可。

動動態規劃的轉換（以 $i = 1, 2, \dots, N$ 的順序計算）

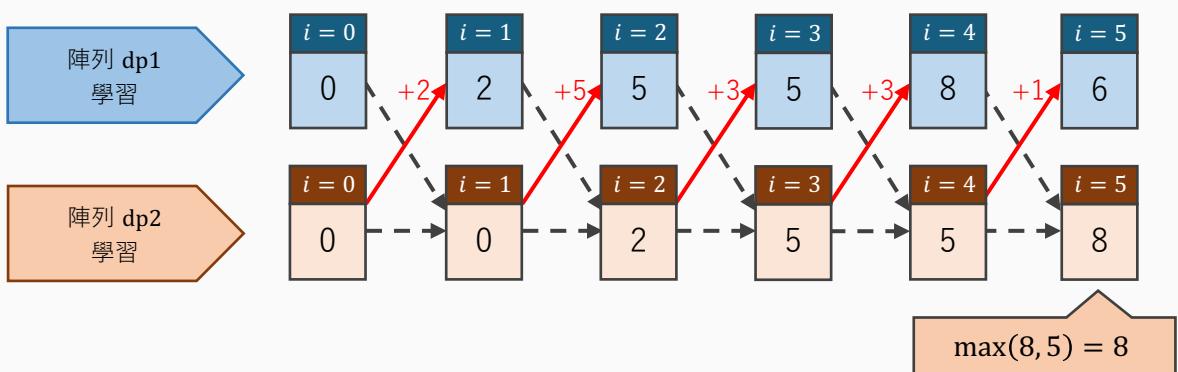
首先，第 i 天學習的方法只有一種，由於第 i 天學習的話，實力會提升 A_i ，因此 $dp1[i] = dp2[i - 1] + A_i$ 。

- 第 $i - 1$ 天不學習（對應於 $dp2[i - 1]$ ）

另一方面，第 i 天不學習的方法有以下兩種，所以 $dp2[i] = \max(dp1[i - 1], dp2[i - 1])$ 。

- 第 $i - 1$ 天學習（對應於 $dp1[i - 1]$ ）
- 第 $i - 1$ 天不學習（對應於 $dp2[i - 1]$ ）

例如，當 $N = 5, (A_1, A_2, A_3, A_4, A_5) = (2, 5, 3, 3, 1)$ 時，陣列 $dp1$ 、 $dp2$ 的轉換如下所示。在此，由於所求的答案（第 N 天結束後實力提升的最大值）是 $\max(dp1[N], dp2[N])$ ，在此例中答案為 8。



這個解法用 C++ 實作如下。注意限制條件為 $N \leq 500000$, $A_i \leq 10^9$ 之大，答案可能超過 10^{14} 。

由於 `int` 型態等32位元整數會發生溢出，因此建議使用 `long long` 型態等 64 位元整數。

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

long long N, A[500009];
long long dp1[500009], dp2[500009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 陣列的初始化
    dp1[0] = 0;
    dp2[0] = 0;

    // 動態規劃法
    for (int i = 1; i <= N; i++) {
        dp1[i] = dp2[i - 1] + A[i];
        dp2[i] = max(dp1[i - 1], dp2[i - 1]);
    }

    // 輸出答案
    cout << max(dp1[N], dp2[N]) << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap3-7.md。。

第 4 章

進階的演算法

4.1

節末問題 4.1 的解答

問題 4.1.1

(1) 由於 $\vec{A} + \vec{B} = (2 + 3, 4 - 9) = (5, -5)$ 答案如下。

- $|\vec{A}| = \sqrt{2^2 + 4^2} = \sqrt{20} = 2\sqrt{5}$ ($\sqrt{5}$ 的 2 倍)
- $|\vec{B}| = \sqrt{3^2 + (-9)^2} = \sqrt{90} = 3\sqrt{10}$ ($\sqrt{10}$ 的 3 倍)
- $|\vec{A} + \vec{B}| = \sqrt{5^2 + (-5)^2} = \sqrt{50} = 5\sqrt{2}$ ($\sqrt{5}$ 的 2 倍)

(2) 根據內積公式 (\rightarrow 4.1.4 項) 計算如下。

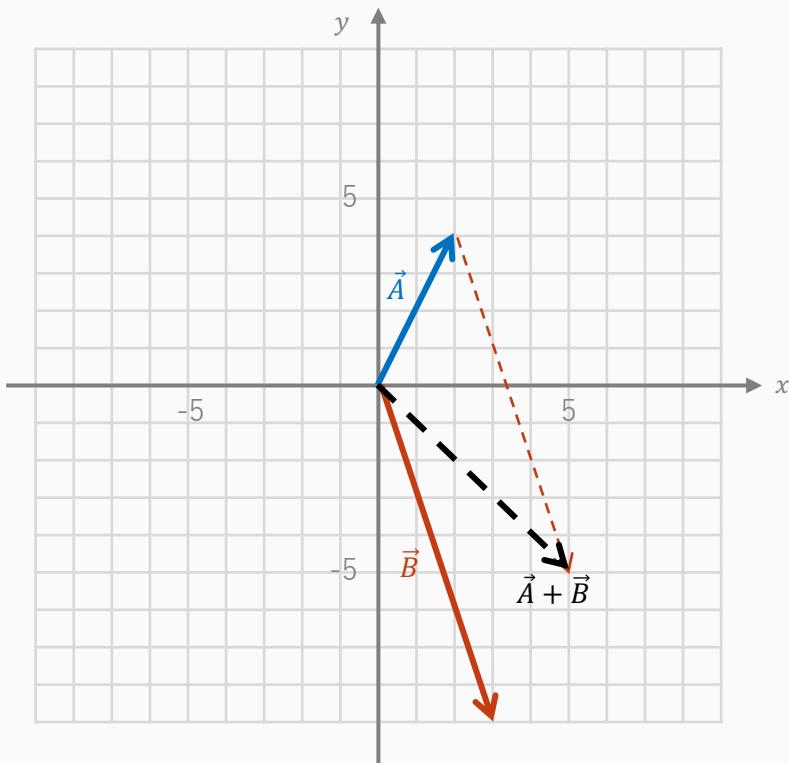
$$\vec{A} \cdot \vec{B} = 2 \times 3 + 4 \times (-9) = -30$$

(3) 雖然見下圖即可立刻理解，但還是特意利用內積來求解吧。

根據 (2) 的答案，由於內積為負，因此夾角 **超過 90 度**。

(4) 根據外積公式 (\rightarrow 4.1.5 項) 計算如下。

$$|A \times B| = |2 \times (-9) - 3 \times 4| = 30.$$



問題 4.1.2

點 (x_i, y_i) 與點 (x_j, y_j) 之間的距離可以用下式表示：

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

因此，如下撰寫一個程式將所有點的組合 (i, j) 進行全搜尋，可以得到正解。此外，可以使用 `sqrt` 函式來計算方根。

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

int N;
double x[2009], y[2009];
double Answer = 1000000000.0; // 初始化成非常大的值

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> x[i] >> y[i];

    // 全搜尋
    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            // dist 為第 i 個點與第 j 個點之間的距離
            double dist = sqrt((x[i]-x[j]) * (x[i]-x[j]) + (y[i]-y[j]) * (y[i]-y[j]));
            Answer = min(Answer, dist);
        }
    }

    // 輸出答案
    printf("%.12lf\n", Answer);
    return 0;
}
```

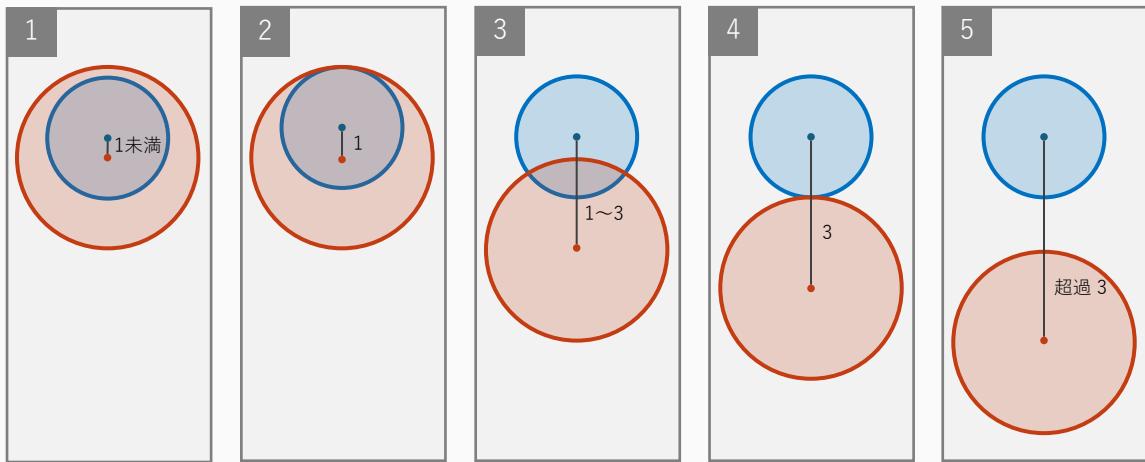
※ Python等原始碼請參閱 chap4-1.md。。

問題 4.1.3

當兩個圓的圓心距離為 d 時，圓的重疊情況如右表所示。（模式的編號請參考書籍的問題敘述）

圓心距離 d	重疊情形
$d < r_1 - r_2 $	模式 [1]
$d = r_1 - r_2 $	模式 [2]
$ r_1 - r_2 < d < r_1 + r_2$	模式 [3]
$d = r_1 + r_2$	模式 [4]
$r_1 + r_2 < d$	模式 [5]

例如，當半徑 2 的圓和半徑 3 的圓逐漸分開時，會如下圖所示。根據前頁的表可知，距離為 1 時在內側相切（**內切**），當距離為 5 時在外側相切（**外切**）。



因此，如下撰寫一個程式來求出圓心距離 d ，可以得到正解。計算兩點間距離的方法如節末問題4.1.2所探討。。

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

double X1, Y1, R1;
double X2, Y2, R2;

int main() {
    // 輸入
    cin >> X1 >> Y1 >> R1;
    cin >> X2 >> Y2 >> R2;

    // 求出圓心之間的距離
    double d = sqrt((X1 - X2) * (X1 - X2) + (Y1 - Y2) * (Y1 - Y2));

    // 輸出答案
    if (d < abs(R1 - R2)) cout << "1" << endl;
    else if (d == abs(R1 - R2)) cout << "2" << endl;
    else if (d < R1 + R2) cout << "3" << endl;
    else if (d == R1 + R2) cout << "4" << endl;
    else cout << "5" << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap4-1.md。

問題 4.1.4

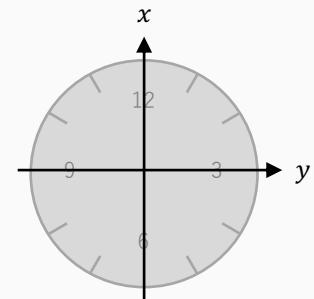
此問題可以使用三角函數（→專欄 4）來解決。首先，當12點方向為 0° 時， H 時 M 分的角度如下：

- 時針的角度： $30H + 0.5M^\circ$
- 分針的角度： $6M^\circ$

因此，當時鐘的圓心為座標 $(0, 0)$ 時，各針的座標如下。注意12點方向為 x 軸。

- 時針的前端： $(A \cos(30H + 0.5)^\circ, A \sin(30H + 0.5)^\circ)$
- 分針的前端： $(B \cos 6H^\circ, B \sin 6H^\circ)$

製作計算這兩點間距離的程式即可得到正解。另外，也可以使用餘弦定理（不在本書範圍內）來解決。



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

const double PI = 3.14159265358979;

int main() {
    // 輸入
    double A, B, H, M;
    cin >> A >> B >> H >> M;

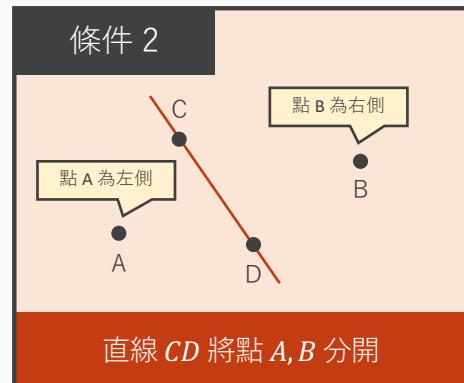
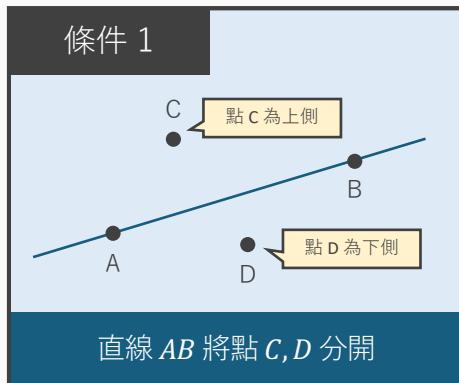
    // 求出座標
    double AngleH = 30.0 * H + 0.5 * M;
    double AngleM = 6.0 * M;
    double Hx = A * cos(AngleH * PI / 180.0), Hy = A * sin(AngleH * PI / 180.0);
    double Mx = B * cos(AngleM * PI / 180.0), My = B * sin(AngleM * PI / 180.0);

    // 求出距離 → 輸出
    double d = sqrt((Hx - Mx) * (Hx - Mx) + (Hy - My) * (Hy - My));
    printf("%.12lf\n", d);
    return 0;
}
```

※ Python等原始碼請參閱 chap4-1.md。

問題 4.1.5

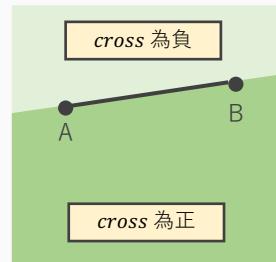
令第 1 條線段的端點為 A, B , 第 2 條線段的端點為 C, D , 兩條線段重相交（具有共同的點）的充要條件基本為滿足以下兩個條件。



因此，線段 AB 是否分隔點 C, D ，可以用以下兩個值是否相反來判斷。

- $\text{cross}(\overrightarrow{AB}, \overrightarrow{AC})$ 的符號（正負）
- $\text{cross}(\overrightarrow{AB}, \overrightarrow{AD})$ 的符號（正負）

cross 函數可以回到 4.1.5 項確認。



因此，如下實作可求解此問題。注意當點 A, B, C, D 排列在一直線上時， $\text{cross}(\overrightarrow{AB}, \overrightarrow{AC}) = 0$ 時等特殊狀況（稱為邊角案例），需要另外區分。

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

long long cross(long long ax, long long ay, long long bx, long long by) {
    // 向量 (ax, ay) 與 (bx, by) 的外積大小
    return ax * by - ay * bx;
}

int main() {
    // 輸入
    long long X1, Y1, X2, Y2, X3, Y3, X4, Y4;
    cin >> X1 >> Y1; // 輸入點 A 的座標
    cin >> X2 >> Y2; // 輸入點 B 的座標
    cin >> X3 >> Y3; // 輸入點 C 的座標
    cin >> X4 >> Y4; // 輸入點 D 的座標
```

```

// 計算 cross(AB, AC)
long long ans1 = cross(X2-X1, Y2-Y1, X3-X1, Y3-Y1);
long long ans2 = cross(X2-X1, Y2-Y1, X4-X1, Y4-Y1);
long long ans3 = cross(X4-X3, Y4-Y3, X1-X3, Y1-Y3);
long long ans4 = cross(X4-X3, Y4-Y3, X2-X3, Y2-Y3);

// 全部排在一直線上時 (邊角案例)
if (ans1 == 0 && ans2 == 0 && ans3 == 0 && ans4 == 0) {
    // 將 A, B, C, D 視為數值 (正確來說是 pair 型態)
    // 藉由適當的進行 swap , 可以假設 A<B, C<D
    // 如此, 可以歸結成判斷區間是否重疊的問題 (節末問題 2.5.6)
    pair<long long, long long> A = make_pair(X1, Y1);
    pair<long long, long long> B = make_pair(X2, Y2);
    pair<long long, long long> C = make_pair(X3, Y3);
    pair<long long, long long> D = make_pair(X4, Y4);
    if (A > B) swap(A, B);
    if (C > D) swap(C, D);
    if (max(A, C) <= min(B, D)) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}

// 並非如此時
// IsAB: 線段 AB 是否將點 C, D 分開?
// IsCD: 線段 CD 是否將點 A, B 分開?
bool IsAB = false, IsCD = false;
if (ans1 >= 0 && ans2 <= 0) IsAB = true;
if (ans1 <= 0 && ans2 >= 0) IsAB = true;
if (ans3 >= 0 && ans4 <= 0) IsCD = true;
if (ans3 <= 0 && ans4 >= 0) IsCD = true;

// 輸出答案
if (IsAB == true && IsCD == true) {
    cout << "Yes" << endl;
}
else {
    cout << "No" << endl;
}
return 0;
}

```

※ Python等原始碼請參閱 chap4-1.md。

4.2

節末問題 4.2 的解答

問題 4.2.1

本問題中，如果用簡單的方法計算每個經過地點之間的距離，計算複雜度為 $O(N)$ ，整體計算複雜度為 $O(NM)$ ，無法滿足本問題的執行時間限制，但若使用累積和（→4.2.1 項）的概念，可以將演算法進行改進。。

首先，令 $X < Y$ 時，以下性質成立。

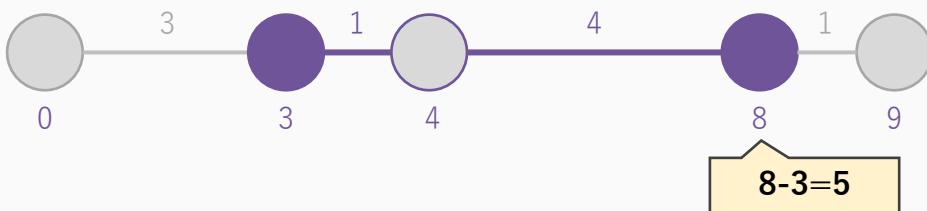
(從站 X 到站 Y 的距離)

$$= (\text{從站 1 到站 } Y \text{ 的距離 } S_Y) - (\text{從站 1 到站 } X \text{ 的距離 } S_X)$$

下圖是當 $N = 5, (A_1, A_2, A_3, A_4) = (3, 1, 4, 1)$ 時的具體例。從站 2 到站 4 的距離可以計算為 $1 + 4 = 5$ ，但也可以使用以下數值，求出 $8 - 3 = 5$ 來取代。

- 從站 1 到站 4 的距離：8
- 從站 1 到站 2 的距離：3

當 $X > Y$ 時，將 X 和 Y 轉倒過來即可。



因此，從站 1 到站 i 的距離為 $S_i = A_1 + \dots + A_{i-1}$ ，對數列 $[A_1, A_2, \dots, A_N]$ 取累積和，可以得到數列 $[S_1, S_2, \dots, S_N]$ 。因此，如下實作時，可以得到正確答案。計算複雜度為 $O(N + M)$ 。

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

int N;
int A[200009], B[200009]; // 站間距離、累積和
```

```

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N - 1; i++) cin >> A[i];
    cin >> M;
    for (int i = 1; i <= M; i++) cin >> B[i];

    // 取累積和
    S[1] = 0;
    for (int i = 2; i <= N; i++) S[i] = S[i - 1] + A[i - 1];

    // 求出答案
    long long Answer = 0;
    for (int i = 1; i <= M - 1; i++) {
        if (B[i] < B[i + 1]) {
            Answer += (S[B[i + 1]] - S[B[i]]);
        }
        else {
            Answer += (S[B[i]] - S[B[i + 1]]);
        }
    }

    // 輸出
    cout << Answer << endl;
    return 0;
}

```

※ Python等原始碼請參閱 chap4-2.md。

問題 4.2.2

這個問題可以照以下步驟解決：

- 計算時刻 $t - 0.5$ 和時刻 $t + 0.5$ 之間的員工數差 B_t 。
- 對列 $[B_1, B_2, \dots, B_T]$ 取累積和，得到列 $[A_1, A_2, \dots, A_T]$ 。

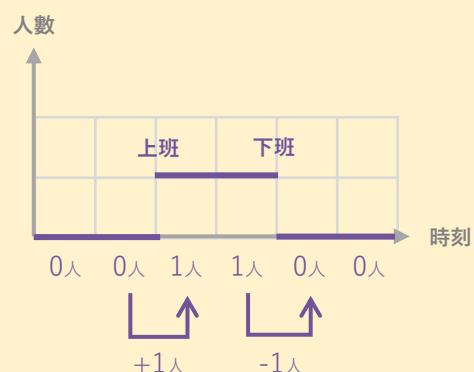
關於差 B_i 可以如下計算。

對於在時刻 L 上班、時刻 R 下班的員工，

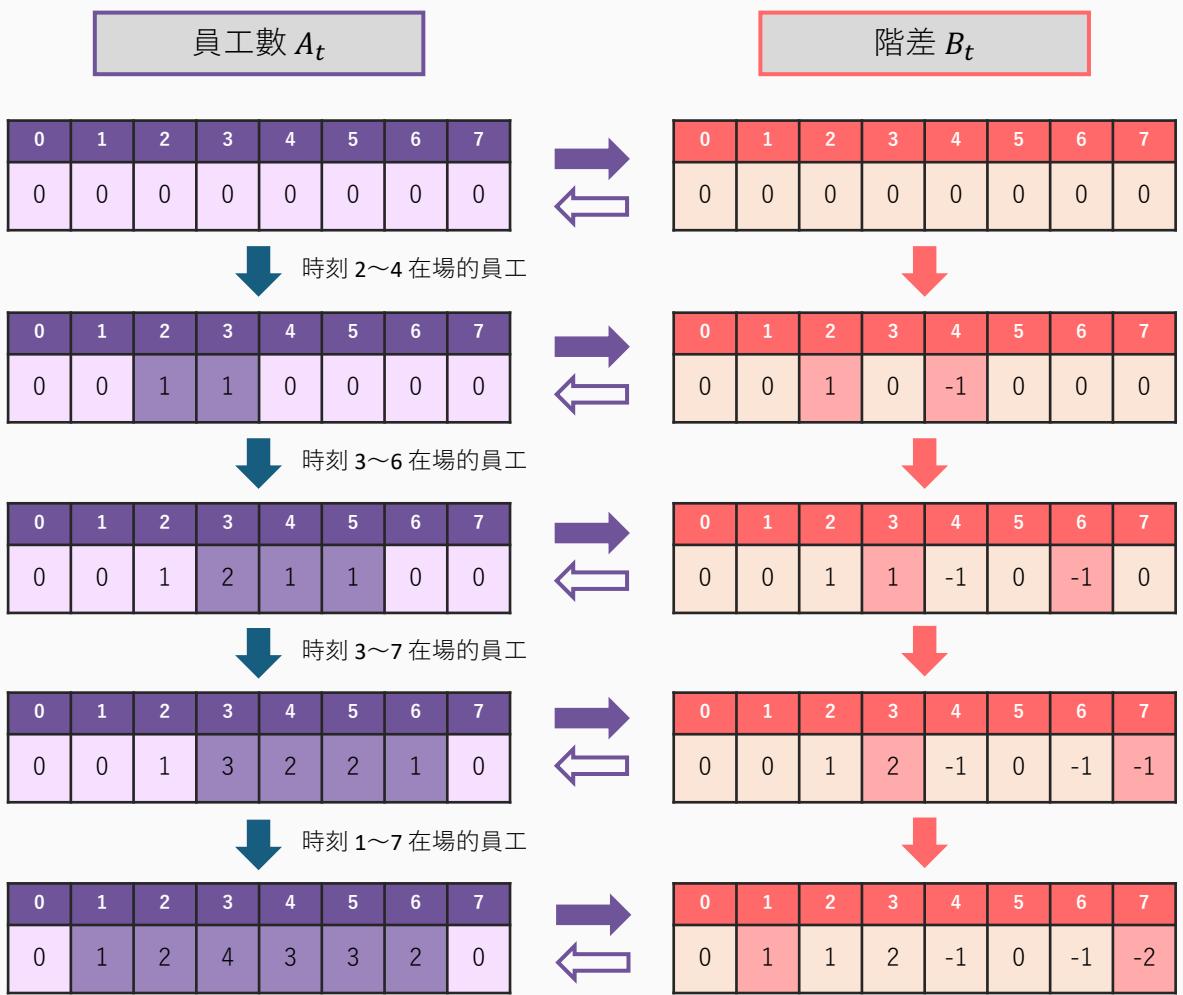
進行以下操作：

- 對 B_L 加 1。
- 對 B_{R+1} 減 1。

在進行操作前，初始化為 $B_i = 0$ 。



例如，當 $T = 8$, $(L_i, R_i) = (2, 4), (3, 6), (3, 7), (1, 7)$ 時，差 B_t 和員工數 A_t 的變化如下所示。



因此，撰寫程式對各個 i ($1 \leq i \leq N$)，於 $B[L_i] + 1$ 且於 $B[R_i] - 1$ 之後，輸出陣列 B 的累積和，即可得到正解。計算複雜度為 $O(N + T)$ 。

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

int N, T;
int L[500009], R[500009];
int A[500009], B[500009];

int main() {
    // 輸入
    cin >> T >> N;
    for (int i = 1; i <= N; i++) cin >> L[i] >> R[i];

    // 計算階差 B[i]
    for (int i = 0; i <= T; i++) B[i] = 0;
    for (int i = 1; i <= N; i++) {
        B[L[i]]++;
        B[R[i]]--;
    }
    for (int i = 1; i <= T; i++) B[i] += B[i-1];
}
```

```

    for (int i = 1; i <= N; i++) {
        B[L[i]] += 1;
        B[R[i]] -= 1;
    }

    // 計算累積和 A[i]
    A[0] = B[0];
    for (int i = 1; i <= T; i++) {
        A[i] = A[i - 1] + B[i];
    }

    // 輸出答案
    for (int i = 0; i < T; i++) cout << A[i] << endl;
    return 0;
}

```

※ Python等原始碼請參閱 chap4-2.md。

問題 4.2.3

首先，如果可以證明以下兩點，就能知道只有以 $f(x) = ax^2 + bx + c$ 的形式表示時，才會回傳 `true`。

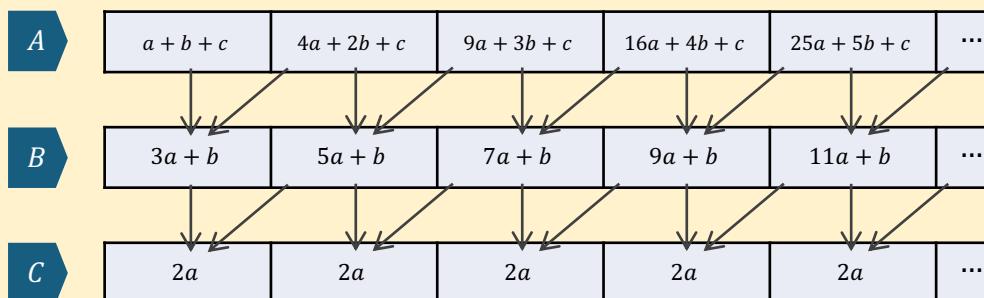
1. 如果為 $f(x) = ax^2 + bx + c$ 的形式，則回傳 `true`。
2. 當回傳 `true` 時，是以 $f(x) = ax^2 + bx + c$ 的形式表示。

對以上兩點分別進行證明吧。

1. 的證明

當 $f(x) = ax^2 + bx + c$ 時， $A[1] = a + b + c$ 、 $A[2] = 4a + 2b + c$ 、 $A[3] = 9a + 3b + c$ 、…以此類推。。

因此，由於 $B[1] = A[2] - A[1]$ ，故 $B[1] = 3a + b$ 。其他情況也進行計算，如下表所示。

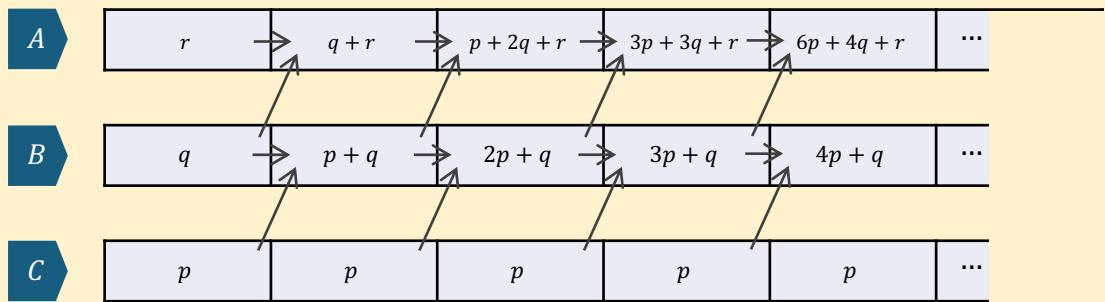


因為 $C = [2a, 2a, \dots, 2a]$ ，回傳 `true`。

2. 的證明

思考函數 `func` 回傳 `true` 的狀況，即 $C[1] = C[2] = \dots = C[N] = p$ 。

階差是累積和的相反操作，所以 B 是 C 的累積和， A 是 B 的累積和。因此，若 $A[1] = r, B[1] = q$ ，則例如 $B[2] = p + q$ 。其他情況也進行計算，如下表所示。



因此，當 $[A[1], A[2], \dots, A[N]] = [r, q + r, p + 2q + r, 3p + 3q + r, \dots]$ 時，可以表示成：

$$f(x) = A[x] = \left(\frac{1}{2}x^2 - \frac{3}{2}x + 1\right)p + (x-1)q + r$$

此為二次函數 ($f(x) = ax^2 + bx + c$ 的形式)，因此可證明 2.。

證明如上。此外，由於已知當 $f(x)$ 為 K 次函數時，取一次階差變成 $K-1$ 次函數，因此取 K 次階差後所有元素變為相同。（本問題是 $K=2$ 的情況）

4.3

節末問題 4.3 的解答

問題 4.3.1

這是測試對多項式函數的微分（→4.3.3項）的理解的問題。答案如下所示。。

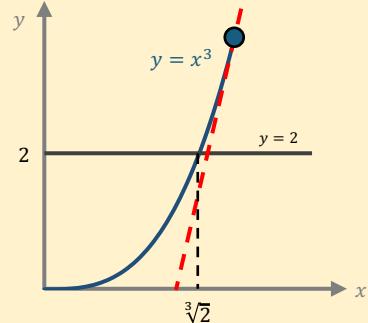
1. $f'(x) = 7$
2. $f'(x) = 2x + 4$
3. $f'(x) = 5x^4 + 4x^3 + 3x^2 + 2x + 1$

問題 4.3.2

$\sqrt[3]{2}$ 的值可以根據以下方法求得（→4.3.6項）。

- 令 $f(x) = x^3$ 。此時 $f'(x) = 3x^2$ 。
- 首先，設定一個任意的初始值 a 。
- 然後，持續將 a 的值更新如下。

通過點 $(a, f(a))$ 之切線與直線
 $y = 2$ 的交點的 x 座標



因此，撰寫如下程式即可。

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

int main() {
    double r = 2.0; // 因為想求  $\sqrt{2}$ 
    double a = 2.0; // 將初始值隨意的設為 2.0

    for (int i = 1; i <= 5; i++) {
        // 求點  $(a, f(a))$  的 x 座標與 y 座標
        double zahyou_x = a;
        double zahyou_y = a * a * a; ← 從程式碼 4.3.1 變更的部分

        // 求切線的斜率 [令  $y = (sessen_a)x + sessen_b$ ]
```

```
double sessen_a = 3.0 * zahyou_x * zahyou_x; ← 從程式碼 4.3.1 變更的部分
```

```
double sessen_b = zahyou_y - sessen_a * zahyou_x;
```

```
// 求下一個 a 的值 next_a
double next_a = (r - sessen_b) / sessen_a;
printf("Step #%d: a = %.12lf -> %.12lf\n", i, a, next_a);
a = next_a;
}
return 0;
}
```

此時，輸出會如下所示。急遽地逼近 $\sqrt[3]{2} = 1.259921049894 \dots$ ，僅用5次而達到小數點後 12 位數的一致性。

```
Step #1: a = 2.000000000000 -> 1.500000000000
Step #2: a = 1.500000000000 -> 1.296296296296
Step #3: a = 1.296296296296 -> 1.260932224742
Step #4: a = 1.260932224742 -> 1.259921860566
Step #5: a = 1.259921860566 -> 1.259921049895
```

另外，有關Python、Java、C的原始碼，請參閱 GitHub 上的 chap4-3.md。

問題 4.3.3

使用二元搜尋法，手動計算 $\sqrt{2}$ 的過程如下表所示。。

操作次數	l	r	m	$m^2 < 2$ 嗎？	範圍示意圖
第 1 次	1.00000	2.00000	1.50000	No	
第 2 次	1.00000	1.50000	1.25000	Yes	
第 3 次	1.25000	1.50000	1.37500	Yes	
第 4 次	1.37500	1.50000	1.43750	No	
第 5 次	1.37500	1.43750	1.40625	Yes	
第 6 次	1.40625	1.43750	1.42188	No	
第 7 次	1.40625	1.42188	1.41406	Yes	
第 8 次	1.41406	1.42188	1.41797	No	
第 9 次	1.41406	1.41797	1.41602	No	

雖然進行了 9 次操作，但 $\sqrt{2} = 1.41421 \dots$ 的小數點後 6 位仍未能一致。

因此，製作如下程式，來檢查需要多少次操作才能達到小數點後6位一致吧。（Python、Java、C的程式請參閱chap4-3.md）

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

int main() {
    double l = 1.0;
    double r = 2.0;

    for (int i = 1; i <= 20; i++) {
        double m = (l + r) / 2.0;
        if (m * m < 2.0) l = m;
        else r = m;
        printf("Step #%-d: m = %.12lf\n", i, m);
    }
    return 0;
}
```

輸出如下所示，可知在 第15次 操作時，終於達到小數點後 6 位一致。由於牛頓法只需 3 次操作，與之相比較慢。

```
Step #1: m = 1.500000000000
Step #2: m = 1.250000000000
Step #3: m = 1.375000000000
Step #4: m = 1.437500000000
Step #5: m = 1.406250000000
Step #6: m = 1.421875000000
Step #7: m = 1.414062500000
Step #8: m = 1.417968750000
Step #9: m = 1.416015625000
Step #10: m = 1.415039062500
Step #11: m = 1.414550781250
Step #12: m = 1.414306640625
Step #13: m = 1.414184570312
Step #14: m = 1.414245605469
Step #15: m = 1.414215087891
Step #16: m = 1.414199829102
Step #17: m = 1.414207458496
Step #18: m = 1.414211273193
Step #19: m = 1.414213180542
Step #20: m = 1.414214134216
```

這樣的二元搜尋法，藉由 1 次操作將精準度變成 2 倍，因此要將精準度提高 P 倍，大約需要 $\log_2 P$ 次操作。本次操作中 $P = 10^5$ ，因此操作次數為 $\log_2 P \approx 16$ 次，與實際次數幾乎一致。

問題 4.3.4

根據指數法則（→**2.3.9項**） $10^{0.3} = 1000^{0.1} = \sqrt[10]{1000}$ 。因此，可以考慮例如以下的方法。

注意，如 x^5 的乘方（整數次方），即使不使用 `pow` 函數，也可以只用四則運算如 `x * x * x * x * x` 來計算。

方法1

令 $f(x) = x^{10}, r = 2$ ，適用一般化的牛頓法（→**4.3.6項**）。在此， $f'(x) = 10x^9$ 。

方法2

透過二元搜尋法（→**節末問題4.3.3**）求出使 $x^{10} = 1000$ 的 x 值。顯然 $1 < x < 2$ ，因此可以設初始值為 $l = 1, r = 2$ 。

還有許多其他方法，請務必思考看看。

4.4

節末問題 4.4 的解答

問題 4.4.1 (1)

這是測試對於將多項式函數積分的方法（→4.4.3項）理解的問題。

$$F(x) = \frac{1}{4}x^4 + x^3 + \frac{3}{2}x^2 + x$$

令 $F(x)$ 為上式，則所求答案如下：

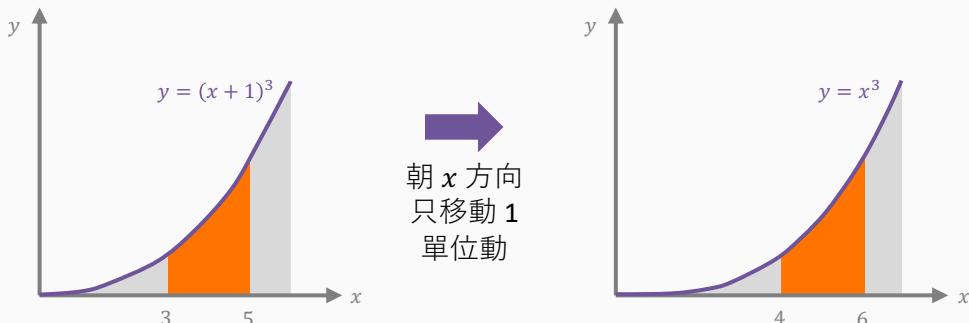
$$\int_3^5 (x^3 + 3x^2 + 3x + 1) dx = F(5) - F(3)$$

因為 $F(5) = 323.75, F(3) = 63.75$ ，所以答案是 $323.75 - 63.75 = \boxed{260}$

另外，使用 $(x^3 + 3x^2 + 3x + 1) = (x + 1)^3$ 的話可以輕鬆計算。

$$\int_3^5 (x + 1)^3 dx = \int_4^6 x^3 dx = \frac{1}{4}(6^4 - 4^4) = 260$$

若使函數的圖形向右平行移動 1 單位，會更容易理解。



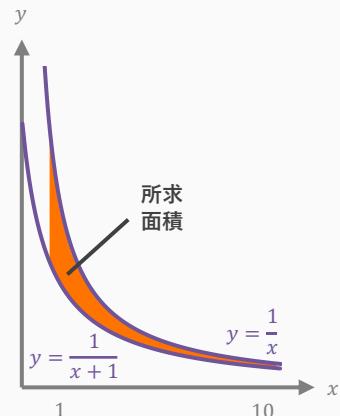
問題 4.4.1 (2)

這是測試對將 $1/x$ 積分的方法（→4.4.3項）理解的問題。

由於積分對應於求附帶符號地面積的操作，因此下式成立。

$$F(x) = \int_1^{10} \frac{1}{x} - \frac{1}{x+1} dx = \int_1^{10} \frac{1}{x} dx - \int_1^{10} \frac{1}{x+1} dx$$

示意圖如右圖。



分別求出紅色部分和藍色部分如下。如果不瞭解 $1/(x+1)$ 的積分，請回到4.4.5項確認。

$$\int_1^{10} \frac{1}{x} dx = \log_e 10 - \log_e 1 = \log_e 10$$

$$\int_1^{10} \frac{1}{x+1} dx = \int_2^{11} \frac{1}{x} dx = \log_e 11 - \log_e 2 = \log_e 11/2$$

因此，根據對數函數的公式（→2.3.10項），求得的答案如下。

$$\log_e 10 - \log_e \frac{11}{2} = \log_e \left(10 \div \frac{11}{2} \right) = \boxed{\log_e \frac{20}{11}}$$

約為 0.5978。

問題 4.4.1 (3)

實際上，下式會成立。

$$\frac{1}{x^2 + x} = \frac{1}{x} - \frac{1}{x+1}$$

因此，所求答案與(2)相同。

$$\int_1^{10} \frac{1}{x^2 + x} dx = \int_1^{10} \frac{1}{x} - \frac{1}{x+1} dx = \boxed{\log_e \frac{20}{11}}$$

問題 4.4.2

已知定積分的值為約 1.2882263643059391197。

那麼，如何求得這個值呢？雖然多項式函數等的積分也可以手動計算出正確的值，但在 $f(x) = 2^{x^2}$ 的情況下，由於函數 $f(x)$ 很複雜，縝密的計算答案非常困難。。

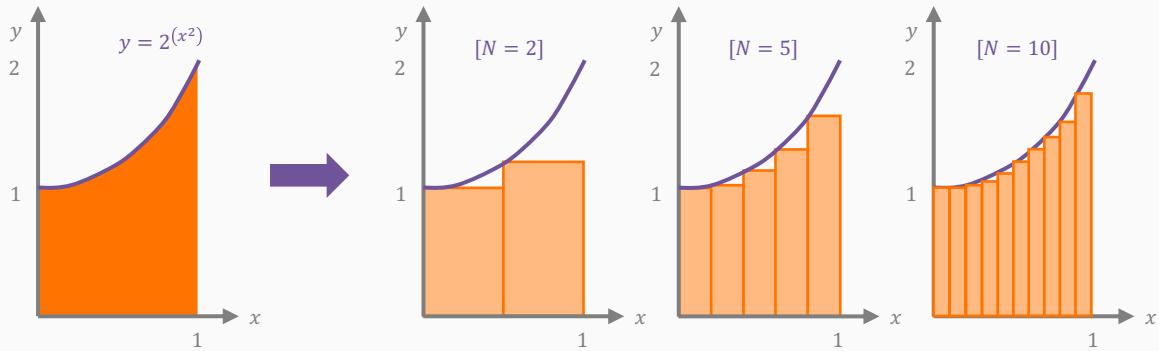
在這種時候，經常使用稱為數值計算（→4.3.7項）的方法，計算答案的近似值來取代。這裡介紹兩種代表性的方法。

方法 1：簡單的區間求積法

因為積分對應於求面積的操作，當 $f(x) = 2^{(x^2)}$ 時，可以近似如下。

$$\int_0^1 f(x)dx = \frac{f(0) + f\left(\frac{1}{N}\right) + f\left(\frac{2}{N}\right) + \cdots + f\left(\frac{N-1}{N}\right)}{N}$$

$N = 2, 5, 10$ 時的示意圖如下。



以這種方法求解定積分值的程式範例如下。在此，越增加 N 的值越可以提高近似精準度。

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

int main() {
    int N = 1000000;
    double Answer = 0.0;

    for (int i = 0; i < N; i++) {
        double x = 1.0 * i / N;
        double value = pow(2.0, x * x); // f(i/N) 的值
        Answer += value;
    }
    printf("%.14lf\n", Answer / N);
    return 0;
}
```

然而，使用這種方法難以將絕對誤差控制在 10^{-12} 以下。實際上：

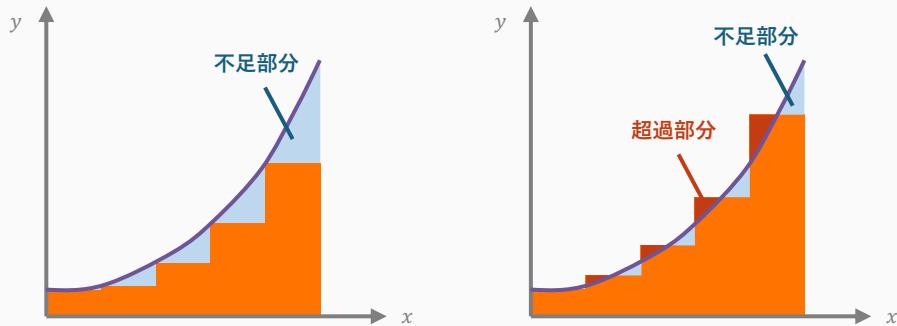
- $N = 1000$ 時，輸出為 1.28772659535497
- $N = 1000000$ 時，輸出為 1.28822586430618

僅能達到 3~6 位的一致性。

方法 2：使用中央值

方法 1 中是使用區間的左端，這裡則使用中央值 $f(1/2N), f(3/2N), \dots$ 來求面積。

下圖下圖顯示了 $N = 5$ 的例子，紅色代表超過的部分，藍色代表不足的部分。使用中央值時，紅色和藍色的面積幾乎相等，可以被正確地計數。



若以數式表示，定積分可近似如下：

$$\int_0^1 f(x) dx = \frac{f\left(\frac{1}{2N}\right) + f\left(\frac{3}{2N}\right) + \cdots + f\left(\frac{2N-1}{2N}\right)}{N}$$

實作後會如下所示。（Python、Java、C的程式請參閱 chap4-4.md）

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

int main() {
    int N = 1000000;
    double Answer = 0.0;

    for (int i = 0; i < N; i++) {
        double x = 1.0 * (2 * i + 1) / (2 * N);
        double value = pow(2.0, x * x); // f((2i+1)/2N) 的值
        Answer += value;
    }
    printf("%.14lf\n", Answer / N);
    return 0;
}
```

與方法 1 相比，精準度大幅提高， $N = 1000000$ 時可實現絕對誤差 10^{-12} 。

- $N = 1000$ 時，輸出為 1.28822624878143
- $N = 1000000$ 時，輸出為 1.28822636430577

此外，還有如辛普森公式等已知的更有效率求定積分近似值的方法。對此感興趣的讀者可以在網路等進行調查。

問題 4.4.3

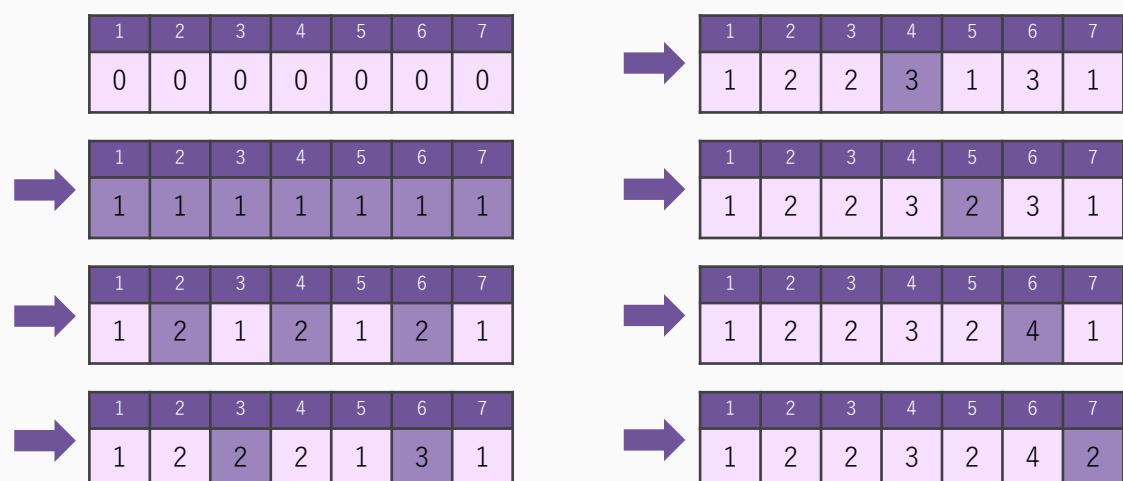
簡單的解法如下。

按照 $i = 1, 2, \dots, N$ 的順序，列舉出所有因數，藉此來計算 $f(i)$ 。這樣就能得知答案。因為列舉因數的計算複雜度為 $O(\sqrt{N})$ ，所以整體處理的計算複雜度 $O(N^{1.5})$ 。

然而，在本問題的限制下會超過執行時間限制（TLE）。更快速地計算 $f(1), f(2), \dots, f(N)$ 的方法如下。

1. 最初，對所有 i ($1 \leq i \leq N$) 設 $f(i) = 0$ 。
2. 1 的倍數：於 $f(1), f(2), f(3), f(4), \dots$ 加 1。
3. 2 的倍數：於 $f(2), f(4), f(6), f(8), \dots$ 加 1。
4. 3 的倍數：於 $f(3), f(6), f(9), f(12), \dots$ 加 1。
5. 對 $4, 5, 6, 7, \dots, N$ 的倍數也進行同樣操作。

$N = 7$ 時，求 $f(1), f(2), \dots, f(N)$ 的過程如下。



接著，來估算此演算法的計算複雜度吧。 x 的倍數全部有 N/x 個，故對所有 x 的倍數加 1 的操作的計算複雜度需要 $O(N/x)$ 。因此，整體的計算次數如下。

$$\frac{N}{1} + \frac{N}{2} + \dots + \frac{N}{N} = O(N \log N)$$

即使 $N = 10^7$ 也能夠高速運行（但 Python 等慢速的程式語言可能會 TLE）。實作範例如下。

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

long long N;
long long F[10000009];
long long Answer = 0;

int main() {
    // 輸入 → 陣列的初始化
    cin >> N;
    for (int i = 1; i <= N; i++) F[i] = 0;

    // 計算 F[1], F[2], ..., F[N]
    for (int i = 1; i <= N; i++) {
        // 對 F[i], F[2i], F[3i], ... 加算 1
        for (int j = i; j <= N; j += i) F[j] += 1;
    }

    // 求出答案 → 輸出
    for (int i = 1; i <= N; i++) {
        Answer += 1LL * i * F[i];
    }
    cout << Answer << endl;
    return 0;
}
```

此外，利用數學考察篇中的「考慮相加次數的技巧（→5.7節）」可將此問題的計算複雜度降為 $O(N)$ 。

問題 4.4.4

已知答案為 6,000,022,499,693。

- 出典：<https://oeis.org/A004080/b004080.txt>

(解說在下頁繼續)

作為簡單的方法，考慮如下程式將 $1/1 + 1/2 + 1/3 + \dots$ 依序相加的方法。但現實的時間上，此方法只能到 $N = 23$ 的程度，否則無法完成執行。

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

int main() {
    // 參數的設定、初始化
    long long cnt = 0;
    double LIMIT = 23; // 將此設為 30 的話，答案即為所求
    double Current = 0;

    // 1 個 1 個相加
    while (Current < LIMIT) {
        cnt += 1;
        Current += 1.0 / cnt;
    }

    // 輸出答案
    cout << cnt << endl;
    return 0;
}
```

因此，舉以下兩種方法作為代表性的快速求解方法。其他還有多種方法，請思考看看。

方法 1：使用近似

如注腳所述，歐拉常數 $\gamma = 0.57721566490153286 \dots$ ，令從 $1/1$ 到 $1/n$ 的和為 H_n ，則 H_n 值會非常接近 $\log_e n - \gamma$ 。因此，使 $\log_e n - \gamma \geq 30$ 的最小的 n 為：

$$\lfloor e^{30+\gamma} \rfloor = \lfloor 6000022499693.369 \dots \rfloor = 6000022499693$$

答案是一致的。

方法 2：使用並行計算

計算次數超過 10^{12} 時，通常難以在現實的時間中完成。然而，若使用 CUDA 等程式語言進行並行計算，可將計算時間縮短 100 倍以上。著名的「超級電腦富岳」也是利用並行計算運行。有興趣的讀者請在網路上調查看看。

4.5

節末問題 4.5 的解答

問題 4.5.1

答案如下。不理解的人請確認 4.5.2 項、4.5.4 項。

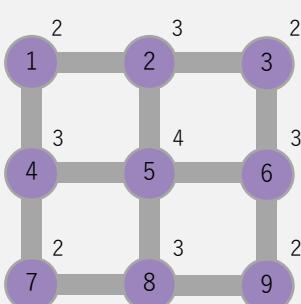
圖的編號	圖的種類	度最大的頂點
A	無權重無向圖形	頂點 1 (度=3)
B	加權無向圖形	頂點 5 (度=1)
C	無權重有向圖形	頂點 3 (出度=2)
D	加權有向圖形	頂點 2 (出度=3)

問題 4.5.2

首先，因為圖 E 存在有奇數度的頂點，所以不存在能夠經過一次所有頂點且回到原頂點的路徑。此外，圖 F 所有頂點的度都是偶數，存在如下圖所示的路徑。

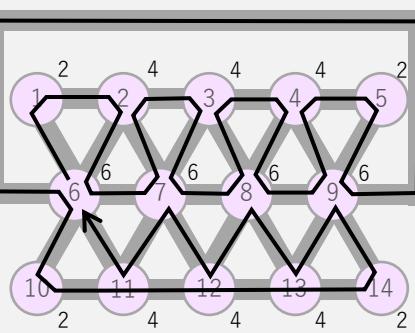
不瞭解的人請確認尤拉圖（→4.5.2項）。下圖中頂點所附的數字是該頂點的度。

E



頂點 2, 4, 6, 8 為奇數度

F

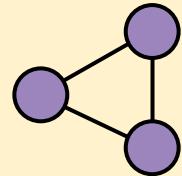


沒有奇數度的頂點

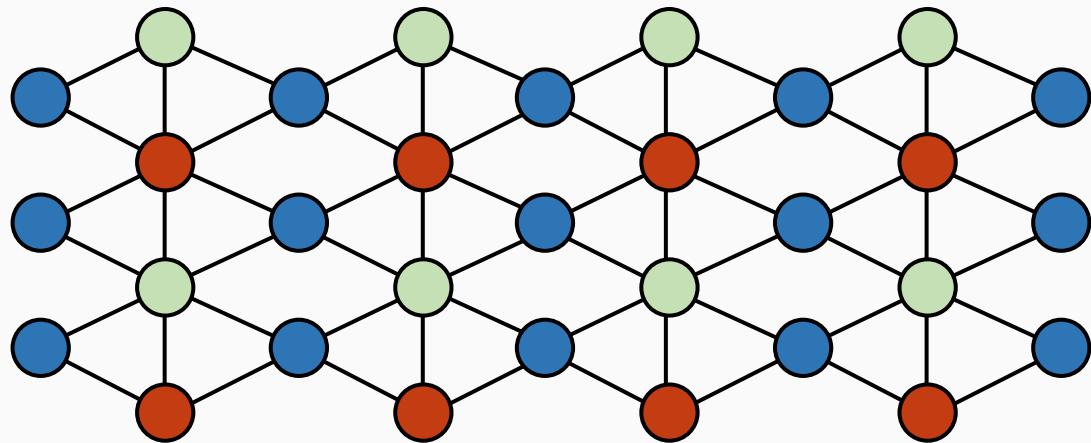
問題 4.5.3

無法用兩種顏色著色的理由如下。

因為圖中包含了如右圖所示的三角形，僅考慮此三角形即可知用兩種顏色著色顯然是不可能的。



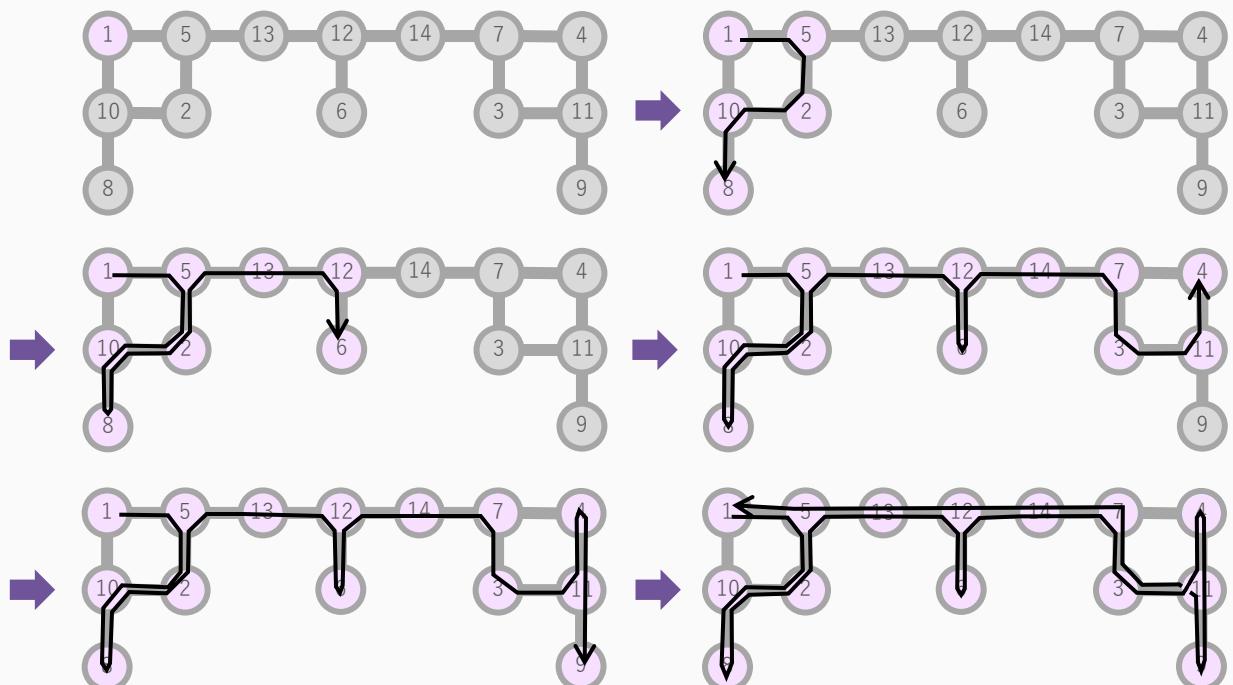
此外，如下圖所示，可以用三種顏色著色。



問題 4.5.4

訪問順序為 **1, 5, 2, 10, 8, 13, 12, 6, 14, 7, 3, 11, 4, 9**。

下圖顯示了深度優先搜尋的具體動作。



問題 4.5.5

這是測試對鄰接表形式（→**4.5.5項**）理解的問題。例如進行以下實作即可得到正解。

此外，變數 `cnt` 表示與頂點 i 鄰接的頂點中，編號小於 i 的頂點個數。`G[i].size()` 是列表 `G[i]` 的元素數量。

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

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

int main() {
    // 輸入
    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]);
    }

    // 求出答案
    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] 是與頂點 i 鄰接的頂點中第 j+1 個
            if (G[i][j] < i) cnt += 1;
        }
        // 如果比自己小的鄰接頂點為 1 個的話，將 Answer 加 1
        if (cnt == 1) Answer += 1;
    }

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

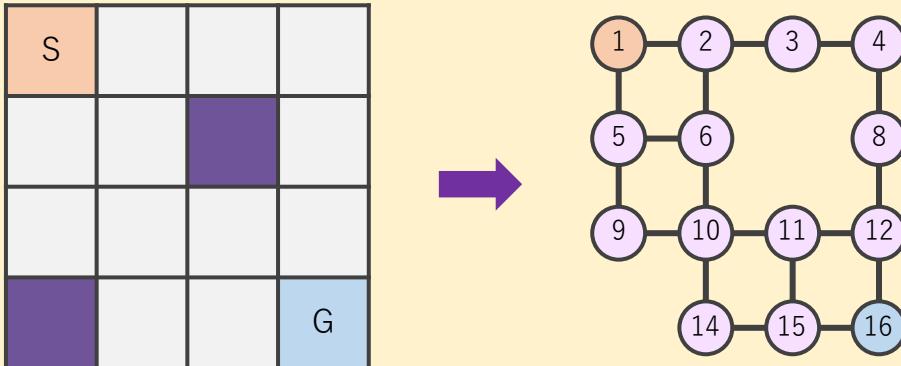
※ Python等原始碼請參閱 chap4-5.md。

問題 4.5.6

若設定頂點編號如下，即可歸納為最短路徑問題（→4.5.7項）。頂點數為 HW ，邊數為 $4HW$ 以下，所以可以在 1 秒內完成執行。。

將從上開始第 i 列，從左開始第 j 個格子的頂點編號設定為 $(i - 1) \times W + j$ 。

$H = 4, W = 4$ 時的具體例如下圖所示。



因此，實作如下可以得到正解。另外，對各格子標出頂點編號般，將資料用數值來表示以識別的方式稱為雜湊。由於是資格考試等常見的問題，有興趣的人可以在網路等進行調查。

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

// 輸入
int H, W;
int sx, sy, start; // 起點的座標 (sx, sy) 與頂點編號 sx*H+sy
int gx, gy, goal; // 終點的座標 (gx, gy) 與頂點編號 gx*W+gy
char c[59][59];

// 圖、最短路徑
int dist[2509];
vector<int> G[2509];

int main() {
    // 輸入
    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];
    }
}
```

```

// 將橫向的邊 [(i, j) - (i, j+1)] 添加到圖中
for (int i = 1; i <= H; i++) {
    for (int j = 1; j <= W - 1; j++) {
        int idx1 = i * W + j; // 頂點 (i, j) 的頂點編號
        int idx2 = i * W + (j+1); // 頂點 (i, j+1) 的頂點編號
        if (c[i][j] == '.' && c[i][j+1] == '.'){
            G[idx1].push_back(idx2);
            G[idx2].push_back(idx1);
        }
    }
}

// 將縱向的邊 [(i, j) - (i+1, j)] 添加到圖中
for (int i = 1; i <= H - 1; i++) {
    for (int j = 1; j <= W; j++) {
        int idx1 = i * W + j; // 頂點 (i, j) 的頂點編號
        int idx2 = (i+1) * W + j; // 頂點 (i+1, j) 的頂點編號
        if (c[i][j] == '.' && c[i+1][j] == '.'){
            G[idx1].push_back(idx2);
            G[idx2].push_back(idx1);
        }
    }
}

// 以下（除了頂點數等）與程式碼 4.5.3 相同
// 廣度優先搜尋的初始化 (dist[i]=-1 時，是未到達的白色頂點)
for (int i = 1; i <= H * W; i++) dist[i] = -1;
queue<int> Q; // 定義佇列 Q
Q.push(start); dist[start] = 0; // 於 Q 添加 1 (操作 1)

// 廣度優先搜尋
while (!Q.empty()) {
    int pos = Q.front(); // 查看 Q 的開頭 (操作 2)
    Q.pop(); // 取出 Q 的開頭 (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); // 於 Q 添加 nex (操作 1)
        }
    }
}

// 輸出答案
cout << dist[goal] << endl;
return 0;
}

```

※ Python等原始碼請參閱 chap4-5.md。

問題 4.5.7

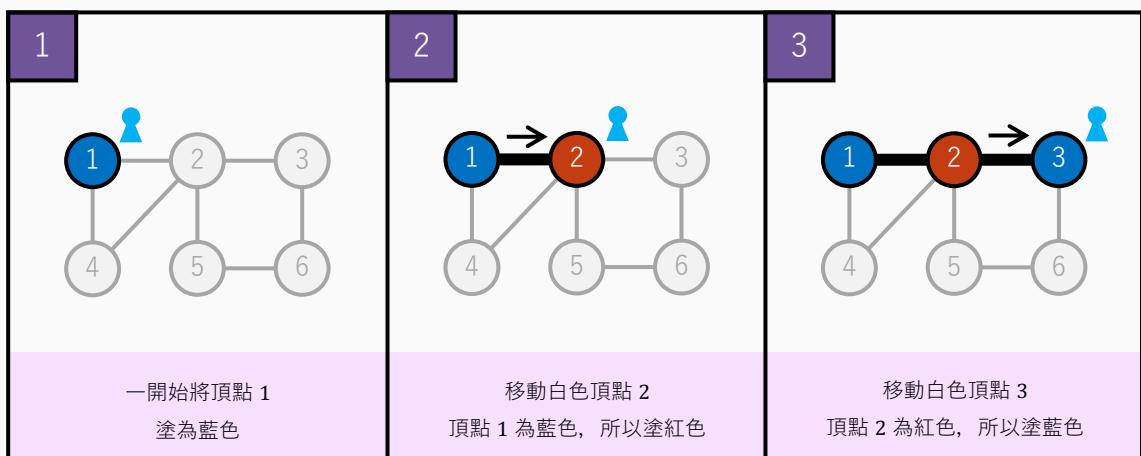
首先，當圖形為連通時，只要決定一個頂點的顏色後，將圖形用藍色和紅色著色的方法最多只有固定一種（如下圖所示）。其原因是必須藍色的旁邊塗紅色、紅色的旁邊塗藍色…如此著色下去。

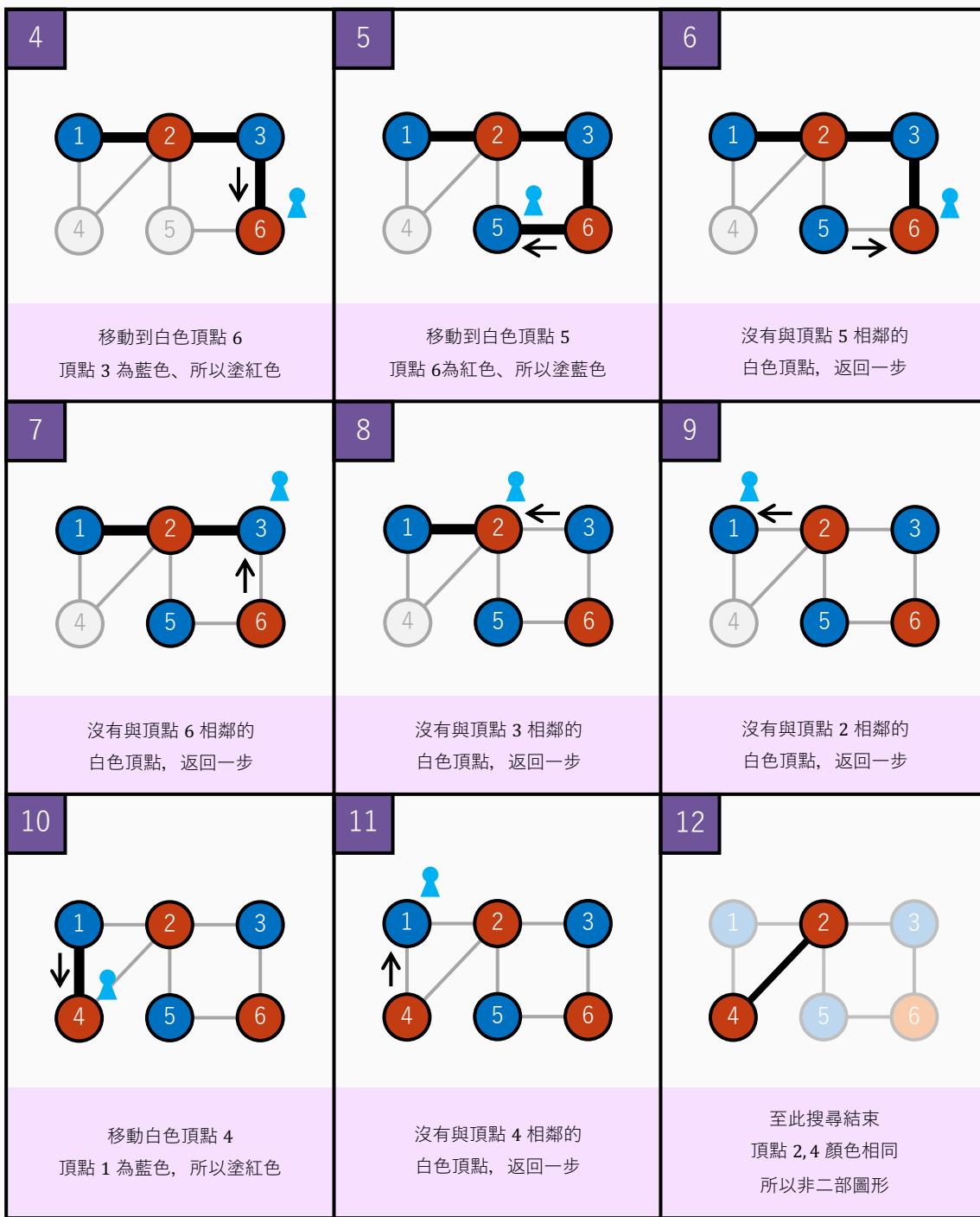


因此，進行以下的深度優先搜尋的話，可以構成一種將圖形用藍色和紅色著色的方法。另外，用藍色文字表示的部分是與4.5.6項所述的連通判斷演算法不同的部分。

1. 將所有頂點塗為白色。
2. 一開始訪問頂點 1，並將頂點 1 塗為**藍色**。
3. 之後，重複以下操作：
 - A) 沒有相鄰的白色頂點：返回一步
 - B) 有相鄰的白色頂點：訪問其中編號最小的頂點。在訪問新的頂點時，用**與前一頂點不同的顏色**著色。
4. 最終，**如果任 2 個相鄰的頂點都以不同顏色著色的話，該圖形為二部圖形。**

將此演算法應用於具體的圖形如下。粗線表示移動路徑。





實作這個演算法如次頁。由於在程式中無法真的將頂點塗為白、藍、紅色，因此設定如下。

- $\text{color}[i]=0$ 時：頂點 i 為白色
- $\text{color}[i]=1$ 時：頂點 i 為藍色
- $\text{color}[i]=2$ 時：頂點 i 為紅色

此外，由於也有輸入的案例為非連通圖形的可能性，注意需要對各連通分量進行相當於步驟2.的操作（參照程式中的「深度優先搜尋」部分）。

```

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

int N, M, A[200009], B[200009]; // N, M ≤ 200000 , 因此使陣列的大小為 200009
vector<int> G[200009];
int color[200009];

void dfs(int pos) {
    for (int i : G[pos]) { // 範圍 for 敘述
        if (color[i] == 0) {
            // color[pos]=1 的時候為 2, color[pos] = 2 的時候為 1
            color[i] = 3 - color[pos];
            dfs(i);
        }
    }
}

int main() {
    // 輸入
    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]);
    }

    // 深度優先搜尋
    for (int i = 1; i <= N; i++) color[i] = 0;
    for (int i = 1; i <= N; i++) {
        if (color[i] == 0) {
            // 頂點 i 為白 (尚未搜尋的連通成分)
            color[i] = 1;
            dfs(i);
        }
    }

    // 是否為二部圖的判斷
    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等原始碼請參閱chap4-5.md。

問題 4.5.8

注意：此問題使用4.5.8項「其他代表性的圖演算法」介紹的Dijkstra演算法。初學者無法解決是自然的，不必擔心。

一般來說，整數可以透過重複進行「乘以 10 再加上 1 至 9 的值」的操作來產生。例如，整數 8691 為：

- 從整數 0 開始
- 乘以 10 再加上 8 ($0 \times 10 + 8 = 8$ になる)
- 乘以 10 再加上 6 ($8 \times 10 + 6 = 86$ になる)
- 乘以 10 再加上 9 ($86 \times 10 + 9 = 869$ になる)
- 乘以 10 再加上 1 ($869 \times 10 + 1 = 8691$ になる)

因此，考慮如下的加權有向圖。頂點 i ($0 \leq i < K$) 表示「除以 K 且餘數為 i 的整數。

關於頂點

- 準備 K 個頂點。
- 令頂點編號分別為 $0, 1, 2, 3, \dots, K - 1$ 。

關於邊

對各頂點 i ($0 \leq i < K$)，添加以下 10 條邊：

- 從頂點 i 到頂點 $(10i + 0) \bmod K$ 的權重為 0 的邊※
- 從頂點 i 到頂點 $(10i + 1) \bmod K$ 的權重為 1 的邊
- 從頂點 i 到頂點 $(10i + 2) \bmod K$ 的權重為 2 的邊
- 從頂點 i 到頂點 $(10i + 3) \bmod K$ 的權重為 3 的邊
- 從頂點 i 到頂點 $(10i + 4) \bmod K$ 的權重為 4 的邊
- 從頂點 i 到頂點 $(10i + 5) \bmod K$ 的權重為 5 的邊
- 從頂點 i 到頂點 $(10i + 6) \bmod K$ 的權重為 6 的邊
- 從頂點 i 到頂點 $(10i + 7) \bmod K$ 的權重為 7 的邊
- 從頂點 i 到頂點 $(10i + 8) \bmod K$ 的權重為 8 的邊
- 從頂點 i 到頂點 $(10i + 9) \bmod K$ 的權重為 9 的邊

※ 為了實作的方便，例外地不添加頂點 $0 \rightarrow 0$ 的邊。

在此，通過一條邊對應於一次操作。例如， $K = 13$ 時將 86 變為 869 的操作是對應於通過從頂點 8 → 頂點 11 的權重 9 的邊 ($86 \bmod 13 = 8$, $869 \bmod 13 = 11$) 。

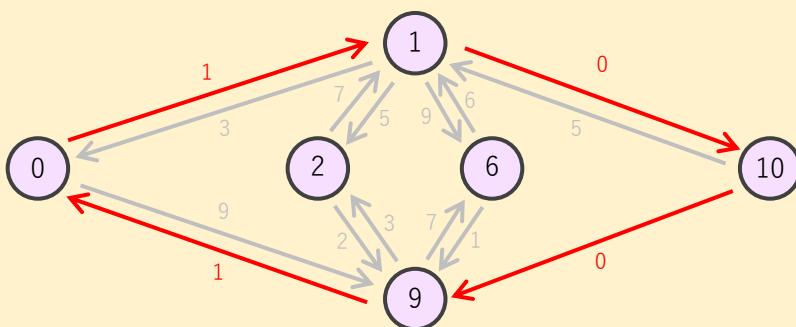


因此，從頂點 0 到頂點 i (≥ 1) 的最短路徑長度為除以 K 餘 i 的數之中，各個位數之和的最小值。

同樣的，從頂點 0 回到頂點 0 的最短路徑長度為 K 的倍數中，各個位數之和的最小值。具體例如下。

例如當 $K = 13$ 時，各個位數之和的最小值是 2 (數 1001)，對應的路徑是 $0 \rightarrow 1 \rightarrow 10 \rightarrow 9 \rightarrow 0$ (補充： $1 \bmod 13 = 1$, $10 \bmod 13 = 10$, $100 \bmod 13 = 9$ 、 $1001 \bmod 13 = 0$) 。

這條路徑是按照前一頁的方法所構成的圖形中的最短路徑。(為了方便閱讀，省略了一些頂點和邊)



因此，使用Dijkstra算法（→4.5.8項）求出加權圖的最短路徑長度即可得到正確答案。

但是，直接實作時從頂點 0 到 0 的最短路徑長度會變為 0 因此需要做一些處理。具體參考次頁的實作例。

※ 也可以參考更詳細的官方解說 (chap4-5.md 中有連結) 。

```

#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;

// Dijkstra法
void dijkstra() {
    // 陣列的初始化等
    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)); // 注意此處勿使 dist[0] = 0 !
}

// 佇列更新
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; // 頂點 0 時為例外
        if (dist[to] > cost) {
            dist[to] = cost;
            Q.push(make_pair(dist[to], to));
        }
    }
}
}

int main() {
    // 輸入
    cin >> K;

    // 添加圖形的邊
    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));
        }
    }

    // Dijkstra法、輸出
    dijkstra();
    cout << dist[0] << endl;
    return 0;
}

```

※ Python等原始碼請參閱 chap4-5.md。

4.6

節末問題 4.6 的解答

問題 4.6.1 (1)

由於乘法運算中，無論在哪個時間點取餘數，答案都不會改變，因此下列兩者相等。

- $21 \times 41 \times 61 \times 81 \times 101 \times 121$ 除以 20 的餘數
- $1 \times 1 \times 1 \times 1 \times 1 \times 1$ 除以 20 的餘數

後者顯然為 1，所以本題的答案是 1。

問題 4.6.1 (2)

即使在計算前取所有數字除以 100 的餘數，答案也不會改變，因此以下會一致。

- 202112^5 除以 100 的餘數
- 12^5 除以 100 的餘數

由於 $12^5 = 248832$ ，答案是 32，但手動計算有點麻煩。因此，可以在計算過程中不斷取餘數，如下：

- $12 \times 12 = 144 \equiv 44 \pmod{100}$
- $44 \times 12 = 528 \equiv 28 \pmod{100}$
- $28 \times 12 = 336 \equiv 36 \pmod{100}$
- $36 \times 12 = 432 \equiv 32 \pmod{100}$

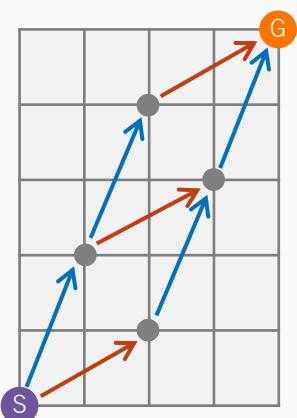
這樣，只需進行最多三位數的計算即可求出答案。

問題 4.6.2

首先從具體例來考慮。要將棋子移動到格子(4, 5)，需要進行：

- $(i, j) \rightarrow (i + 1, j + 2)$ 的移動 2 次
- $(i, j) \rightarrow (i + 2, j + 1)$ 的移動 1 次

反過來，只要滿足這個條件，就一定可以到達目的位置。因為在 3 次移動中有 2 次是朝 $(i + 1, j + 2)$ 的移動，所以移動方法的數量是 ${}_3C_2 = 3$ 種。



接下來，考慮一般的情況。

- $(i, j) \rightarrow (i + 1, j + 2)$ 的移動 a 次（設為移動 A）
- $(i, j) \rightarrow (i + 2, j + 1)$ 的移動 b 次（設為移動 B）

進行以上行動時，為了移動到格子 (X, Y) ，需要滿足以下三個條件：

- a 和 b 是非負整數。
- x 座標的限制： $a + 2b = X$
- y 座標的限制： $2a + b = Y$

在此，同時滿足第 2 及第 3 個條件的只有 $(a, b) = \left(\frac{2Y-X}{3}, \frac{2X-Y}{3}\right)$ 這一組解。

然而，依據第一個條件，也有可能成為答案為 0 種的狀況。

- 因為 a 、 b 為整數，若 $2Y - X$ 及 $2X - Y$ 都不是 3 的倍數，則為 0 種
- 因為 a 、 b 為 0 以上，若 $2Y - X < 0$ 或 $2X - Y < 0$ ，則為 0 種

並非如此的時候，在整體移動次數 $(X + Y)/3$ 次之中，進行 $(2Y - X)/3$ 次移動 A 的話，一定會到達目的格子，因此所求答案為 $\binom{X+Y}{2Y-X}/3$ 種。將程式碼 4.6.5 稍微變更如下實作，可得到正解。

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

const long long mod = 1000000007;
int X, Y;

long long modpow(long long a, long long b, long long m) {
    // 重複平方方法 (p 取如 a^1, a^2, a^4, a^8, ... 的值)
    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;
}

// Division(a, b, m) 為傳回 a ÷ b mod m 的函式
long long Division(long long a, long long b, long long m) {
    return (a * modpow(b, m - 2, m)) % m;
}
```

```

int main() {
    // 輸入
    cin >> X >> Y;

    // 狀況區分 (a, b 會變成負的時候)
    if (2 * Y - X < 0 || 2 * X - Y < 0) {
        cout << "0" << endl;
        return 0;
    }

    // 狀況區分 (a, b 不會成為整數的時候)
    if ((2 * Y - X) % 3 != 0 || (2 * X - Y) % 3 != 0) {
        cout << "0" << endl;
        return 0;
    }

    // 求出二項係數的分子與分母 (步驟 1./步驟 2.)
    long long bunshi = 1, bunbo = 1;
    long long a = (2 * Y - X) / 3, b = (2 * X - Y) / 3;
    for (int i = 1; i <= a + b; i++) { bunshi *= i; bunshi %= mod; }
    for (int i = 1; i <= a; i++) { bunbo *= i; bunbo %= mod; }
    for (int i = 1; i <= b; i++) { bunbo *= i; bunbo %= mod; }

    // 求出答案 (步驟 3.)
    cout << Division(bunshi, bunbo, mod) << endl;
    return 0;
}

```

※ Python等原始碼請參閱chap4-6.md。

問題 4.6.3

首先，下式會成立。例如當 $N = 2$ 時， $4^0 + 4^1 + 4^2 = 1 + 4 + 16 = 21$ ，同時 $(4^{N+1} - 1)/3 = 63 \div 3 = 21$ ，兩個值一致。

$$4^0 + 4^1 + 4^2 + \dots + 4^N = \frac{4^{N+1} - 1}{3}$$

證明雖然較難，但可以如下思考：若將長度為 $4^{N+1}/3$ 的長條每次切去 $3/4$ ，且將此操作重複 $N + 1$ 次，從第 1 次開始依序取出長度 $4^N, 4^{N-1}, \dots, 4^0$ 的部分，則最終只會剩下長度 $1/3$ 。（參考：和的公式 → 2.5.10 項）。



したがって、以下のような方法で答えを割った余りを求めることができます因此，可以用以下方法來求得答案，即除以 $M = 1000000007$ 的餘數：。

- 求出 $4^{N+1} - 1$ 除以 M 的餘數 V (\rightarrow 4.6.7項)
- 求出 $V \div 3$ 除以 M 的餘數 (\rightarrow 4.6.8項)

以下是實作例。其中，函數 `Division(a, b, m)` 是回傳 $a \div b$ 除以 m 的餘數。此外，由於限制是 $N \leq 10^{18}$ ，注意 `modpow` 函數的迴圈次數大約需要 $\log_2(10^{18}) \approx 60$ 次。

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

const long long mod = 1000000007;
long long N;

long long modpow(long long a, long long b, long long m) {
    // 重複平方法 (p 取如 a^1, a^2, a^4, a^8, ... 的值)
    long long p = a, Answer = 1;
    for (int i = 0; i < 60; i++) {
        if ((b & (1LL << i)) != 0) { Answer *= p; Answer %= m; }
        p *= p; p %= m;
    }
    return Answer;
}

// Division(a, b, m) 為傳回 a÷b mod m 的函式
long long Division(long long a, long long b, long long m) {
    return (a * modpow(b, m - 2, m)) % m;
}

int main() {
    // 輸入
    cin >> N;

    // 計算答案
    long long V = modpow(4, N + 1, mod) - 1;
    long long Answer = Division(V, 3, mod);

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

※ Python等原始碼請參閱 chap4-6.md。

4.7

節末問題 4.7 的解答

問題 4.7.1

首先，矩陣的乘法如下所示（矩陣與整數相同，乘法優先計算）。不瞭解的人請回到 4.7.3 項確認。

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} [1 \ 1 \ 1] = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

所求的答案是藍色所示的兩個 2×3 矩陣的和，如下。。

$$\begin{bmatrix} 2 & 0 & 2 \\ 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 3 \\ 3 & 3 & 3 \end{bmatrix}$$

問題 4.7.2

首先，因為 $a_3 = 2a_2 + a_1$ 、 $a_2 = a_2$ ，以下的式會成立。

$$\begin{bmatrix} a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix}$$

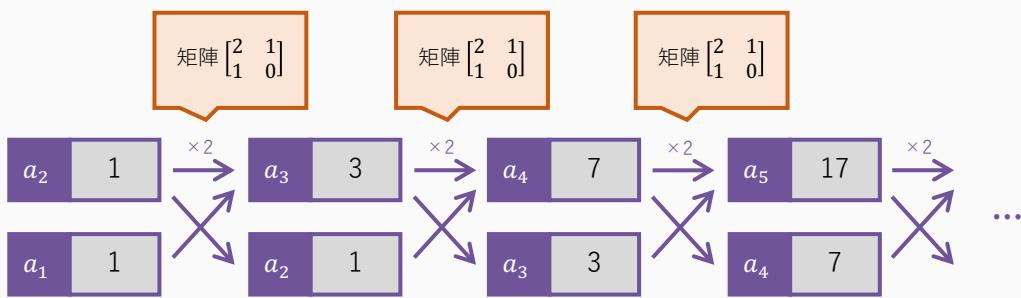
同樣地，因為 $a_4 = 2a_3 + a_2$ 、 $a_3 = a_3$ ，以下的式會成立。

$$\begin{bmatrix} a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \end{bmatrix}$$

對 a_5 之後也重複進行同樣計算的話，會變成如下所示。從此事實可得知，令 A 為由 $2, 1, 1, 0$ 組成的矩陣時， a_N 的值是 A^{N-1} 的 $(2, 1)$ 元素及 $(2, 2)$ 元素相加的值。

$$\begin{bmatrix} a_{N+1} \\ a_N \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}^N \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} = A^{N-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

下圖表示了矩陣與數列 $a = (a_1, a_2, a_3, \dots)$ 的關係。



因此，實作如下的話可以得到正確答案。與程式碼 4.7.1 不同的部分用深藍色標示。
(反過來說，這個程式與程式碼 4.7.1 幾乎相同。)

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

struct Matrix {
    long long p[2][2] = { {0, 0}, {0, 0} };
};

Matrix Multiplication(Matrix A, Matrix B) { // 回傳 2×2 矩陣 A, B 乘積的函式
    Matrix C;
    for (int i = 0; i < 2; i++) {
        for (int k = 0; k < 2; k++) {
            for (int j = 0; j < 2; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // 回傳 A 的 n 次方的函式
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}

int main() {
    // 輸入 → 乘方的計算 (注意若 N 小於 2 的話不會正確動作)
}
```

```

long long N;
cin >> N;

// 製作矩陣 A
Matrix A;
A.p[0][0] = 2; A.p[0][1] = 1; A.p[1][0] = 1;

// B=A^{N-1} 的計算
Matrix B = Power(A, N - 1);

// 輸出答案
cout << (B.p[1][0] + B.p[1][1]) % 1000000007 << endl;
return 0;
}

```

※ Python等原始碼請參閱chap4-7.md。

問題 4.7.3

首先，根據 $a_4 = a_3 + a_2 + a_1, a_3 = a_3, a_2 = a_2$ ，以下的式子成立。

$$\begin{bmatrix} a_4 \\ a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix}$$

同樣地，根據 $a_5 = a_4 + a_3 + a_2, a_4 = a_4, a_3 = a_3$ ，以下的式子成立。

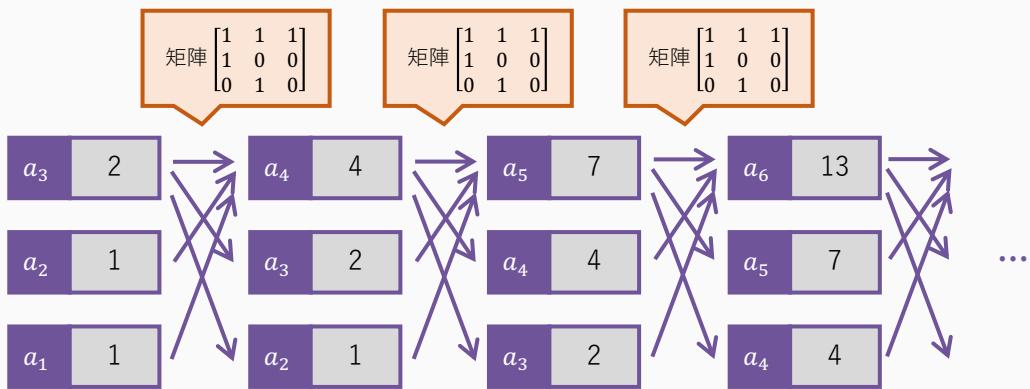
$$\begin{bmatrix} a_5 \\ a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_4 \\ a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^2 \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix}$$

對 a_6 之後也重複進行同樣計算的話，會變成如下所示。其中，將由 $1, 1, 1, 1, 0, 0, 0, 1, 0$ 組成的 3×3 矩陣設為 A 。

$$\begin{bmatrix} a_{N+2} \\ a_{N+1} \\ a_N \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{N-1} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix} = A^{N-1} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

從此可知， a_N 的值可以用 $[A^{N-1} \text{ 的 } (3, 1) \text{ 元素} \times 2 + (3, 2) \text{ 元素} + (3, 3) \text{ 元素}]$ 表示。

下圖表示了矩陣與數列 $a = (a_1, a_2, a_3, \dots)$ 的關係。



因此，如下實作可以得到正確答案。注意矩陣的大小變為 3×3 。（與程式碼 4.7.1 不同的部分用深藍色標示。）

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

struct Matrix {
    long long p[3][3] = { {0, 0, 0}, {0, 0, 0}, {0, 0, 0} };
};

Matrix Multiplication(Matrix A, Matrix B) { // 回傳  $3 \times 3$  矩陣 A, B 乘積的函式
    Matrix C;
    for (int i = 0; i < 3; i++) {
        for (int k = 0; k < 3; k++) {
            for (int j = 0; j < 3; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // 回傳 A 的 n 次方的函式
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}
```

```

int main() {
    // 輸入 → 乘方的計算（注意若 N 小於 2 的話不會正確動作）
    long long N;
    cin >> N;

    // 製作矩陣 A
    Matrix A;
    A.p[0][0] = 1; A.p[0][1] = 1; A.p[0][2] = 1; A.p[1][0] = 1; A.p[2][1] = 1;

    // B=A^{N-1} 的計算
    Matrix B = Power(A, N - 1);

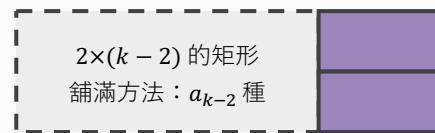
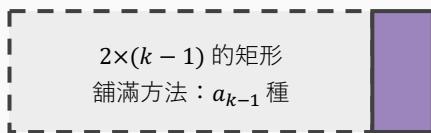
    // 輸出答案
    cout << (2LL * B.p[2][0] + B.p[2][1] + B.p[2][2]) % 1000000007 << endl;
    return 0;
}

```

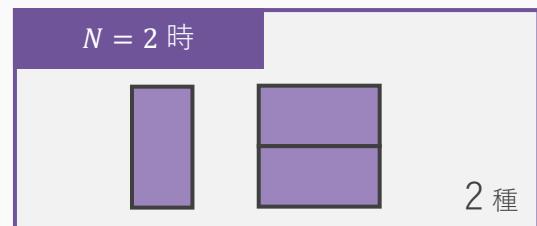
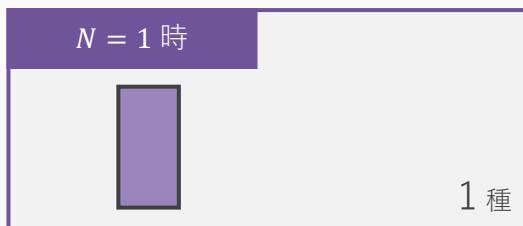
※ Python 等原始碼請參閱 chap4-7.md。

問題 4.7.4 (1)

當從左開始依序舖滿 $2 \times k$ 的矩形時，最後放置的部分是以下兩種情況之一（當 $k \geq 2$ 時）：



因此，將舖滿 $2 \times k$ 的矩形的方法數設為 a_k 時， $a_k = a_{k-1} + a_{k-2}$ 的遞迴式成立。而且 $N = 1$ 時的答案為 $a_1 = 1$ ， $N = 2$ 時的答案為 $a_2 = 1$ 。

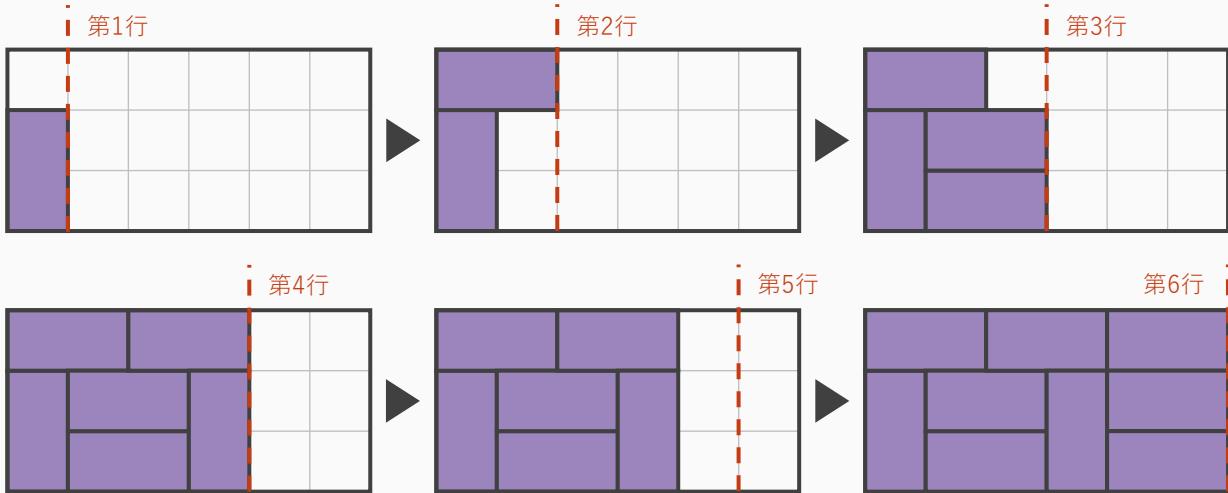


因此， $2 \times N$ 時的答案為以下的費波那契數的第 $N + 1$ 項，製作將其輸出的程式（→ 4.7.1項）即可得到正確答案。。

N	1	2	3	4	5	6	7	8
a_N	1	2	3	5	8	13	21	34

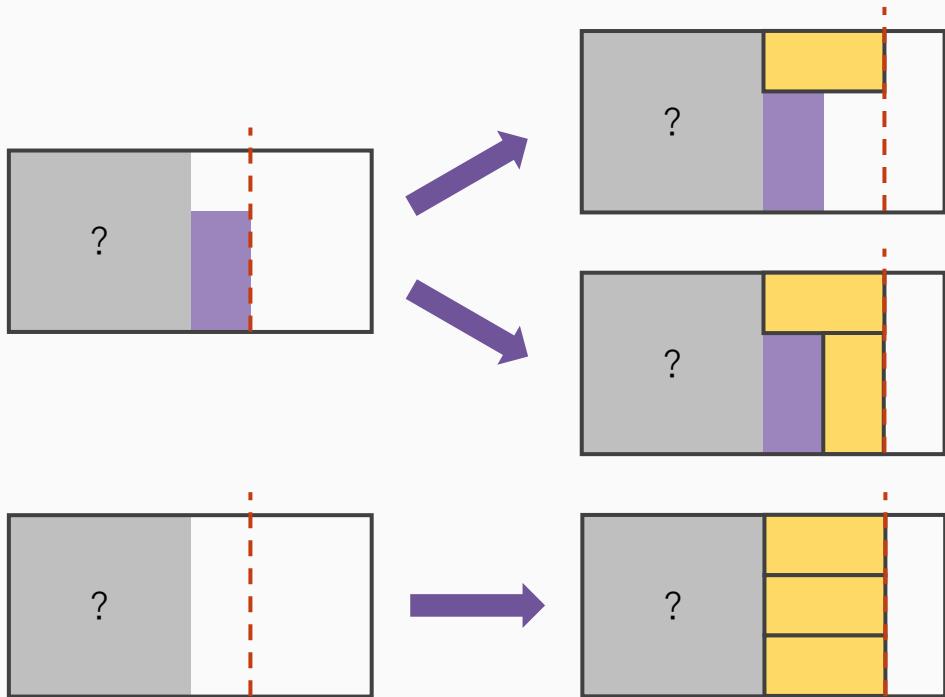
問題 4.7.4 (2), (3)

如下，考慮從左開始一行行舖滿。當舖到第 x 行為止時，橫跨第 x 行和第 $x + 1$ 行的矩形不含在內。

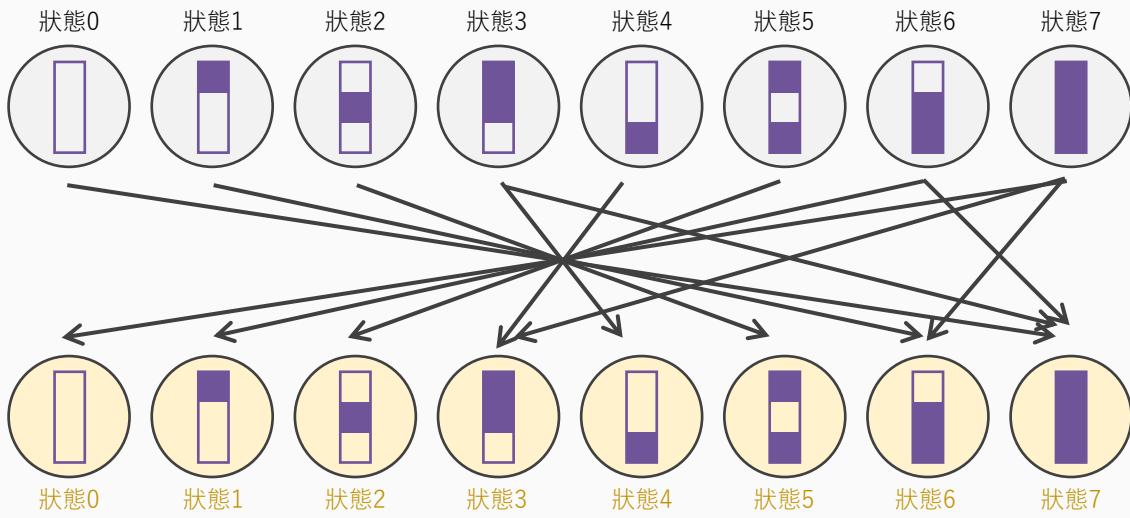


此時，「舖到第 x 行為止時，一定已將第 $x - 1$ 行前全部蓋住」的重要性質會成立，因此在此之後的舖滿方式只依賴於最後一行（第 x 行）。

例如，當 $K = 3$ 且下方的 2 列已經舖滿時，有以下兩種放置方式。其他情況也可以同樣思考。



那麼，考慮「在第 x 行的階段中最後一行為○○時，使第 $x + 1$ 行的階段中最後一行為△△的方法有幾種？」吧。例如，當 $K = 3$ 時，如下圖所示，每個箭頭代表 1 種方法。



因此，將最後一行的狀態編號為 $0, 1, \dots, 2^K - 1$ 時，令 $dp[x][y]$ 為「在舖到第 x 行的時間點時為狀態 y 的情況數」，則以下式子成立。

其中， $A_{p,q}$ 為從狀態 p 轉換到狀態 q 的方法數（例如，若無箭頭，則 $A_{p,q} = 0$ ）。並且， $L = 2^K - 1$ 。

$$\begin{bmatrix} dp[x+1][0] \\ dp[x+1][1] \\ dp[x+1][2] \\ \vdots \\ dp[x+1][L] \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,L} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,L} \\ A_{2,0} & A_{2,1} & A_{2,2} & \cdots & A_{2,L} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{L,0} & A_{L,1} & A_{L,2} & \cdots & A_{L,L} \end{bmatrix} \begin{bmatrix} dp[x][0] \\ dp[x][1] \\ dp[x][2] \\ \vdots \\ dp[x][L] \end{bmatrix}$$

這個式子略顯複雜，但可以如下思考：例如，左邊 $(1, 1)$ 元素的 $dp[x+1][0]$ 的值是將「填滿到第 x 行且為狀態○○的情況數 \times 從狀態○○轉換到狀態 0 的方法數」對所有的狀態相加的總和。

因此， $dp[N][0], dp[N][1], \dots, dp[N][N-1]$ 的值如下（反覆應用上述公式即可）。

$$\begin{bmatrix} dp[N][0] \\ dp[N][1] \\ dp[N][2] \\ \vdots \\ dp[N][L] \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \cdots & A_{0,L} \\ A_{1,0} & A_{1,1} & A_{1,2} & \cdots & A_{1,L} \\ A_{2,0} & A_{2,1} & A_{2,2} & \cdots & A_{2,L} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{L,0} & A_{L,1} & A_{L,2} & \cdots & A_{L,L} \end{bmatrix}^N \begin{bmatrix} dp[0][0] \\ dp[0][1] \\ dp[0][2] \\ \vdots \\ dp[0][L] \end{bmatrix} = A^N \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

例如，當 $K = 3$ 時， $dp[N][0], dp[N][1], dp[N][2], \dots, dp[N][7]$ 的值如下（ $K = 4$ 時因為過長故省略）。

$$\begin{bmatrix} dp[N][0] \\ dp[N][1] \\ dp[N][2] \\ dp[N][3] \\ dp[N][4] \\ dp[N][5] \\ dp[N][6] \\ dp[N][7] \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}^N \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

此時，最後一列（第 N 列）必須完全舖滿，故所求答案為 $dp[N][L] = dp[N][2^K - 1]$ 。

因此，使用重複平方法來計算 $2^K \times 2^K$ 矩陣的乘方，可以在計算複雜度為 $O((2^K)^3 \times \log N)$ 的情況下解決這個問題。以下是 C++ 的實作例。

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

// K=2 時的轉移
long long Mat2[4][4] = {
    {0, 0, 0, 1},
    {0, 0, 1, 0},
    {0, 1, 0, 0},
    {1, 0, 0, 1}
};

// K=3 時的轉移
long long Mat3[8][8] = {
    {0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 1, 0, 0, 0},
    {0, 0, 0, 1, 0, 0, 0, 1},
    {0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 1, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0, 0, 1}
};

// K=4 時的轉移
long long Mat4[16][16] = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0}
};
```

```

    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0}
};

struct Matrix {
    int size_ = 0; // 矩陣的大小 (設為 size_ × size_ 的正方矩陣)
    long long p[16][16];
};

Matrix Multiplication(Matrix A, Matrix B) { // 回傳矩陣 A, B 乘積的函式
    Matrix C;

    // 矩陣 C 的初始化
    C.size_ = A.size_;
    for (int i = 0; i < A.size_; i++) {
        for (int j = 0; j < A.size_; j++) C.p[i][j] = 0;
    }

    // 矩陣的乘法
    for (int i = 0; i < A.size_; i++) {
        for (int k = 0; k < A.size_; k++) {
            for (int j = 0; j < A.size_; j++) {
                C.p[i][j] += A.p[i][k] * B.p[k][j];
                C.p[i][j] %= 1000000007;
            }
        }
    }
    return C;
}

Matrix Power(Matrix A, long long n) { // 回傳 A 的 n 次方的函式
    Matrix P = A, Q;
    bool flag = false;
    for (int i = 0; i < 60; i++) {
        if ((n & (1LL << i)) != 0LL) {
            if (flag == false) { Q = P; flag = true; }
            else { Q = Multiplication(Q, P); }
        }
        P = Multiplication(P, P);
    }
    return Q;
}

```

```
int main() {
    // 輸入
    long long K, N;
    cin >> K >> N;

    // 製作矩陣 A
    Matrix A; A.size_ = (1 << K);
    for (int i = 0; i < (1 << K); i++) {
        for (int j = 0; j < (1 << K); j++) {
            if (K == 2) A.p[i][j] = Mat2[i][j];
            if (K == 3) A.p[i][j] = Mat3[i][j];
            if (K == 4) A.p[i][j] = Mat4[i][j];
        }
    }

    // B=A^N 的計算
    Matrix B = Power(A, N);

    // 輸出答案
    cout << B.p[(1 << K) - 1][(1 << K) - 1] << endl;
    return 0;
}
```

第 5 章

用以解決問題的
數學思考

5.2

節末問題 5.2 的解答

問題 5.2.1

以下為費波那契數到第 12 項為止的數及其除以 4 的餘數。。

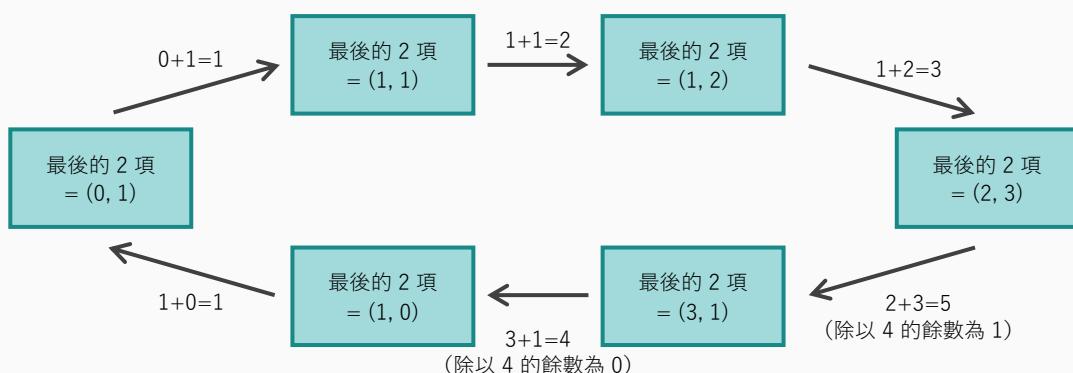
$1 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow \dots$ 會週期性地重複。

N	1	2	3	4	5	6	7	8	9	10	11	12
第 N 項	1	1	2	3	5	8	13	21	34	55	89	144
除以 4 的餘數	1	1	2	3	1	0	1	1	2	3	1	0

此週期性在 N 增大時是否依然成立呢？實際上可以從以下兩點來證明：

- 在費波那契數中，每一項的值只由前兩項的值決定
- (第 1 項，第 2 項) 與 (第 7 項，第 8 項) 一致

示意圖如下。



因此，費波那契數的第 N 項除以 4 的餘數，與第 $(N \bmod 6)$ 項除以 4 的餘數相同（當 N 是 6 的倍數時是與第 6 項相同）。

$10000 \bmod 6 = 4$ ，所以費波那契數的第 10000 項除以 4 的餘數是第 4 項除以 4 的餘數，即 3。

(解說在下一頁繼續)

問題 5.2.2

首先，石頭為 1 個的狀態下，無法再拿石頭，因此 $N = 1$ 是敗北狀態（後手必勝）。

另一方面，當有 2 個石頭時，若先手取 1 個石頭，後手無法再行動，因此 $N = 2$ 是獲勝狀態。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
狀態	敗	勝													

接下來考慮 $N = 3$ 的情況。一般而言，遊戲只有在能轉換到敗北狀態時才能獲勝 (\rightarrow 5.2.2項)。但由於「取 1 個石頭，減少到剩下 2 個（獲勝狀態）」是唯一能做的操作，因此 $N = 3$ 是敗北狀態。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
狀態	敗	勝	敗												



接下來，當石頭有 4, 5, 6 個的狀態時，可以一步減少到 3 個石頭（敗北狀態），因此這些是獲勝狀態。此外，當石頭有 7 個的狀態，有以下三種操作：

- 取 1 個石頭，剩下 6 個（獲勝狀態）
- 取 2 個石頭，剩下 5 個（獲勝狀態）
- 取 3 個石頭，剩下 4 個（獲勝狀態）

但所有操作都轉換到獲勝狀態。因此 $N = 7$ 是敗北狀態。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
狀態	敗	勝	敗	勝	勝	勝	敗								



進行同樣的考察的話，可以知道 $N = 8, 9, 10, 11, 12, 13, 14$ 是獲勝狀態，而 $N = 15$ 是敗北狀態。

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
狀態	敗	勝	敗	勝	勝	勝	敗	勝	勝	勝	勝	勝	勝	勝	敗



至此，我們發現 $1, 3, 7, 15$ 是敗北狀態，敏銳的人可能會想到「是否只有以 $2^k - 1$ 表示的數字是敗北狀態呢？」的週期性。（如果沒想到， $N = 16$ 之後的情況也請調查看看）。

事實上，這種週期性在 N 增大時依然成立（證明略）。因此，撰寫以下程式，在 $1 \leq k \leq 60$ 的範圍內進行全搜尋以判斷是否為 $N = 2^k - 1$ ，可以得到正解。

此外，本問題的限制 $N \leq 10^{18}$ ，因為 $2^{60} > 10^{18}$ ，所以僅需搜尋到 $k \leq 60$ 為止即可。

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

int main() {
    // 輸入
    long long N;
    cin >> N;

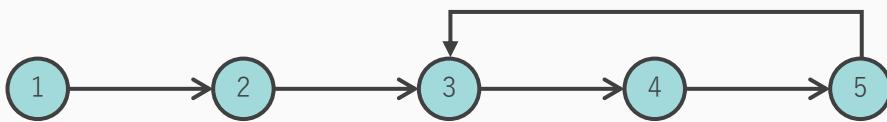
    // 檢查是否以 N = 2^k-1 的形式表示
    bool flag = false;
    for (int k = 1; k <= 60; k++) {
        if (N == (1LL << k) - 1LL) flag = true;
    }

    // 輸出
    if (flag == true) cout << "Second" << endl;
    else cout << "First" << endl;
    return 0;
}
```

※ Python等原始碼請參閱chap5-2.md。

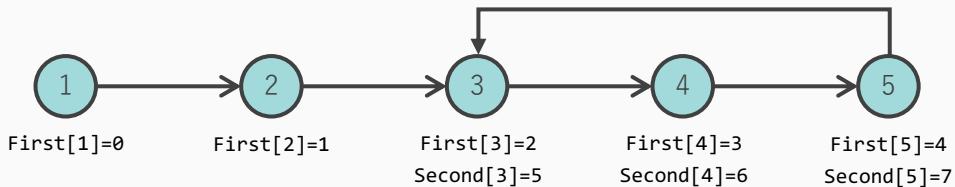
問題 5.2.3

這個問題稍微有點複雜，因此首先思考 $N = 5, A = (2, 3, 4, 5, 3)$ 的情況。發信機的轉發如下圖所示，從城鎮 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow \dots$ 週期性地移動。



此週期性在一般情況下也成立。可以證明在 N 次移動以內，會回到已訪問過的城市，之後將以週期性方式進行移動。

在此，假設第一次訪問城鎮 u 時，使用了 $First[u]$ 次發信機，第二次訪問時，使用了 $Second[u]$ 次發信機吧。（下圖為具體例）



設 $L = Second[u] - First[u]$ （週期的長度），則在使用了第 $First[u], First[u] + L, First[u] + 2L, \dots$ 次發信機時訪問城鎮 u 。上述例子中，

- 城鎮 3：在使用第 $2, 5, 8, 11, 14, 17, \dots$ 次發信機時訪問
- 城鎮 4：在使用第 $3, 6, 9, 12, 15, 18, \dots$ 次發信機時訪問
- 城鎮 5：在使用第 $4, 7, 10, 13, 16, 19, \dots$ 次發信機時訪問

因此，當 $(K - First[u]) \bmod L = 0$ 時，在第 K 次移動到達城市 u 。提出可以檢查這種 u 的程式來得到正解。

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

long long N, K;
long long A[200009];
long long First[200009], Second[200009];

int main() {
    // 輸入
    cin >> N >> K;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 陣列的初始化
    for (int i = 1; i <= N; i++) First[i] = -1;
    for (int i = 1; i <= N; i++) Second[i] = -1;

    // 求出答案
    // cur 為現在所在城鎮的編號
    long long cnt = 0, cur = 1;
    while (true) {
        // First, Second 的更新
        if (cur == 1) {
```

```
if (First[cur] == -1) First[cur] = cnt;
else if (Second[cur] == -1) Second[cur] = cnt;

// 判斷在 K 次移動後是否位在城鎮 cur
if (cnt == K) {
    cout << cur << endl;
    return 0;
}
else if (Second[cur] != -1 && (K-First[cur]) % (Second[cur]-First[cur]) == 0) {
    cout << cur << endl;
    return 0;
}

// 位置更新
cur = A[cur];
cnt += 1;
}
return 0;
}
```

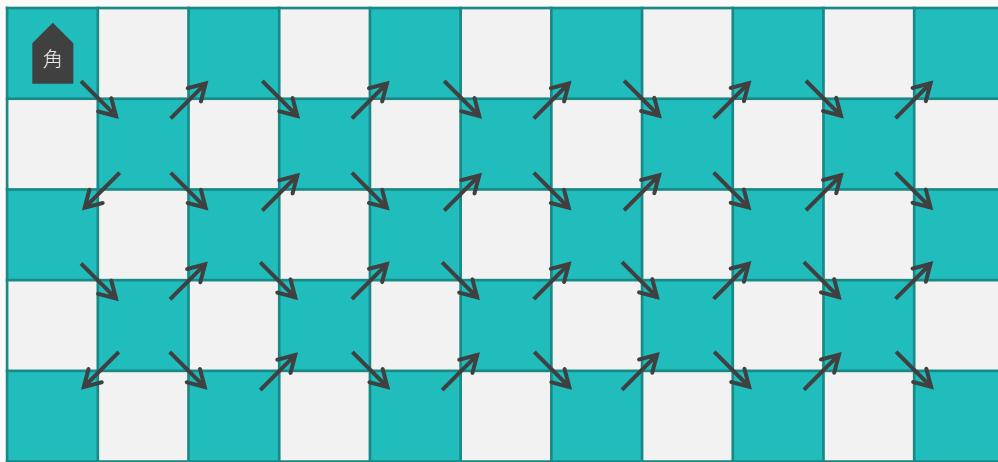
※ Python等原始碼請參閱 chap5-2.md。

5.3

節末問題 5.3 的解答

問題 5.3.1

首先思考 $H = 5, W = 11$ 的情況。令從上開始第 x 列，從左開始第 y 行的格子為 (x, y) 時，只能到達 $x + y$ 為偶數的格子。



實際上，當 $H \geq 2, W \geq 2$ 時，僅能到達 $x + y$ 為偶數的格子（總共 $\lceil HW/2 \rceil$ 個）。不能移動到奇數格子的理由如下。

因為角行只能朝斜向移動，只考慮相鄰的格子時，可進行的移動如下。

- 從格子 $(x, y) \rightarrow$ 格子 $(x + 1, y + 1)$
- 從格子 $(x, y) \rightarrow$ 格子 $(x + 1, y - 1)$
- 從格子 $(x, y) \rightarrow$ 格子 $(x - 1, y + 1)$
- 從格子 $(x, y) \rightarrow$ 格子 $(x - 1, y - 1)$

然而，移動後， $[x$ 座標] + $[y$ 座標]的值的增減為 $-2, 0, 2$ 任一者，因此奇偶不會變。

由於起始位置（左上的格子）為 $[x$ 座標] + $[y$ 座標] = 2（偶數），所以無法移動到奇數格。

因此，如次頁進行實作的話可以得到正解。又，注意當 $H = 1$ 或 $W = 1$ 時，答案為 1。這種需要區分的情況稱為「邊角案例」。

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

int main() {
    // 輸入
    long long H, W;
    cin >> H >> W;

    // 區分狀況
    if (H == 1 || W == 1) {
        cout << "1" << endl;
    }
    else {
        cout << (H * W + 1) / 2 << endl;
    }
    return 0;
}
```

※ Python 等原始碼請參閱 chap5-3.md。。

問題 5.3.2

思考用以下步驟來決定數字的選擇方法：

- **步驟 1**：決定 2, 3, 4, 5, 6, 7, 8, 9, 10 的選擇方法
- **步驟 2**：決定 1 的選擇方法

首先，步驟 1 的選擇方法共有 $2^9 = 512$ 種（→ 3.3.2項）。另一方面，在步驟 1 結束時的最終選擇數字的總和為奇數時，步驟 2 的選擇方法的必定只有 1 種。



因此，所求答案為 $512 \times 1 = \boxed{512}$ 種（正好全體的一半）。

5.4

節末問題 5.4 的解答

問題 5.4.1

首先，「至少出現一個 6 點」的餘事件是「全部都是 5 點以下」，因此下式會成立。

$$(\text{至少出現一個6 點的機率}) = 1 - (\text{全部都是 5 點以下的機率})$$

因此，根據乘法原理（→ 3.3.2項）全部都是 5 點以下的機率為 $(5/6) \times (5/6) \times (5/6) = 125/216$ 。於是，答案如下。

$$1 - \frac{125}{216} = \frac{\textcolor{red}{91}}{\textcolor{red}{216}}$$

問題 5.4.2

依照 5.4.4 項的解說進行實作即可。以下為實作例。各變數和陣列的意義如下：

- `gyou[i]`：第 i 列的總和
- `retu[j]`：第 j 行的總和
- `Answer[i][j]`：對於第 i 列、第 j 行格子的答案

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

int H, W, A[2009][2009];
int gyou[2009]; // 列的總和
int retu[2009]; // 行的總和
int Answer[2009][2009];

int main() {
    // 輸入
    cin >> H >> W;
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) cin >> A[i][j];
    }

    // 計算列的總和
    for (int i = 1; i <= H; i++) {
```

```

        gyou[i] = 0;
        for (int j = 1; j <= W; j++) gyou[i] += A[i][j];
    }

    // 計算行的總和
    for (int j = 1; j <= W; j++) {
        retu[j] = 0;
        for (int i = 1; i <= H; i++) retu[j] += A[i][j];
    }

    // 對各格計算答案
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) {
            Answer[i][j] = gyou[i] + retu[j] - A[i][j];
        }
    }

    // 以空格區隔來輸出
    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) {
            if (j >= 2) cout << " ";
            cout << Answer[i][j];
        }
        cout << endl;
    }
    return 0;
}

```

※ Python等原始碼請參閱chap5-4.md。

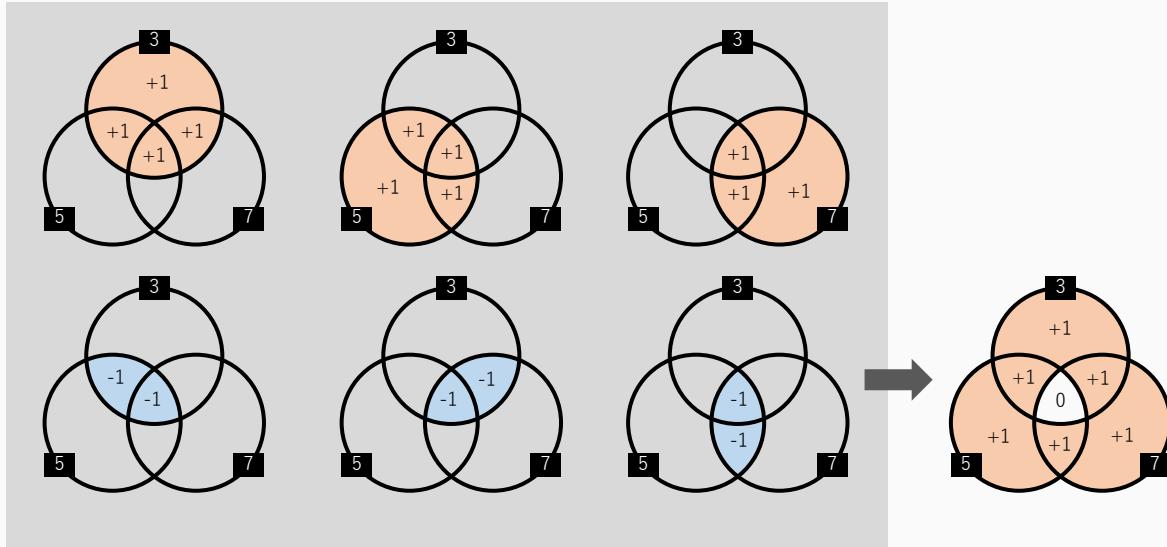
問題 5.4.3 (1)

一般而言， N 以下的整數中， M 的倍數的數量為 $\lfloor N/M \rfloor$ 個，因此：

- 3 的倍數有 $A_1 = \lfloor 1000 \div 3 \rfloor = \text{333 個}$
- 5 的倍數有 $A_2 = \lfloor 1000 \div 5 \rfloor = \text{200 個}$
- 7 的倍數有 $A_3 = \lfloor 1000 \div 7 \rfloor = \text{142 個}$
- 15 的倍數有 $A_4 = \lfloor 1000 \div 15 \rfloor = \text{66 個}$
- 21 的倍數有 $A_5 = \lfloor 1000 \div 21 \rfloor = \text{47 個}$
- 35 的倍數有 $A_6 = \lfloor 1000 \div 35 \rfloor = \text{28 個}$
- 105 的倍數有 $A_7 = \lfloor 1000 \div 105 \rfloor = \text{9 個}$

問題 5.4.3 (2), (3), (4), (5)

- (2) 計算 $A_1 + A_2 + A_3$ 時，例如 **15 這個數字** 被計算了 2 次（在 A_1 和 A_2 都被算到）。
- (3) 計算 $A_1 + A_2 + A_3 - A_4 - A_5 - A_6$ 時，**105 這個數字** 在 A_1, A_2, A_3 被計算了 3 次，但是在 A_4, A_5, A_6 也被扣除了 3 次，因此完全沒有被計算到。。
- 如下圖所示，未正確計算到的數字只有 105 的倍數。。



- (4) 將在 (3) 中未計算到的 105 的倍數加上即可。答案為 **$A_1 + A_2 + A_3 - A_4 - A_5 - A_6 + A_7$** 。
- (5) 根據 (1) 的答案， $333 + 200 + 142 - 66 - 47 - 28 + 9 = \mathbf{543}$ 個。

問題 5.4.4

集合 $S_1, S_2, S_3, \dots, S_N$ 的聯集（任一個有被包含的部分）的元素數量如下式表示。

從 N 個集合中選擇至少一個的方法有 $2^N - 1$ 種，對全部的方法加總以下的值。

- 當選擇奇數個集合時：所選集合共同部分的元素數量
- 當選擇偶數個集合時：所選集合共同部分的元素數量 $\times (-1)$

例如，集合 S_1, S_2, S_3 的聯集的元素數量為以下所有值的和。。

- S_1 的元素數量
- S_2 的元素數量
- S_3 的元素數量
- S_1 和 S_2 的共同部分 $\times (-1)$
- S_1 和 S_3 的共同部分 $\times (-1)$
- S_2 和 S_3 的共同部分 $\times (-1)$
- S_1 和 S_2 和 S_3 的共同部分 $\times (-1)$

將「1000 以下的 3 的倍數」當成 S_1 ，「1000 以下的 5 的倍數」當成 S_2 ，「1000 以下的 7 的倍數」當成 S_3 試試看。應該會是與問題 5.4.3 相同結果。

問題 5.4.5

設集合 S_1 為「 N 以下 V_1 的倍數」，集合 S_2 為「 N 以下 V_2 的倍數」，...，集合 S_K 為「 N 以下 V_K 的倍數」時，所求答案為 S_1, S_2, \dots, S_K 的共同部分。

因此，撰寫以下程式，將 N 個集合的選擇方法（選擇哪些倍數）進行 $2^N - 1$ 種全搜尋的話，即可得到正解。使用了位元全搜尋（→專欄1）這種實作方法。

此外， N 以下且為 P_1, P_2, \dots, P_M 所有值的倍數的整數個數以下式表示。

$$\frac{N}{P_1, P_2, P_3, \dots, P_M \text{ 的最小公倍數}}$$

(3 個以上的最小公倍數的求法 →節末問題3.2.3)

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

long long N, K;
long long V[20];
long long Answer = 0;

// 回傳最大公因數的函式
long long GCD(long long A, long long B) {
    if (B == 0) return A;
    return GCD(B, A % B);
}
```

```

// 回傳最小公倍數的函式
long long LCM(long long A, long long B) {
    return (A / GCD(A, B)) * B;
}

int main() {
    // 輸入
    cin >> N >> K;
    for (int i = 1; i <= K; i++) cin >> V[i];

    // 位元全搜尋
    for (int i = 1; i < (1 << K); i++) {
        long long cnt = 0; // 選擇數字的個數
        long long lcm = 1; // 最小公倍數
        for (int j = 0; j < K; j++) {
            if ((i & (1 << j)) != 0) {
                cnt += 1;
                lcm = LCM(lcm, V[j + 1]);
            }
        }
        long long num = N / lcm; // 數的個數，此數是被選擇的所有數字的倍數
        if (cnt % 2 == 1) Answer += num;
        if (cnt % 2 == 0) Answer -= num;
    }

    // 輸出
    cout << Answer << endl;
    return 0;
}

```

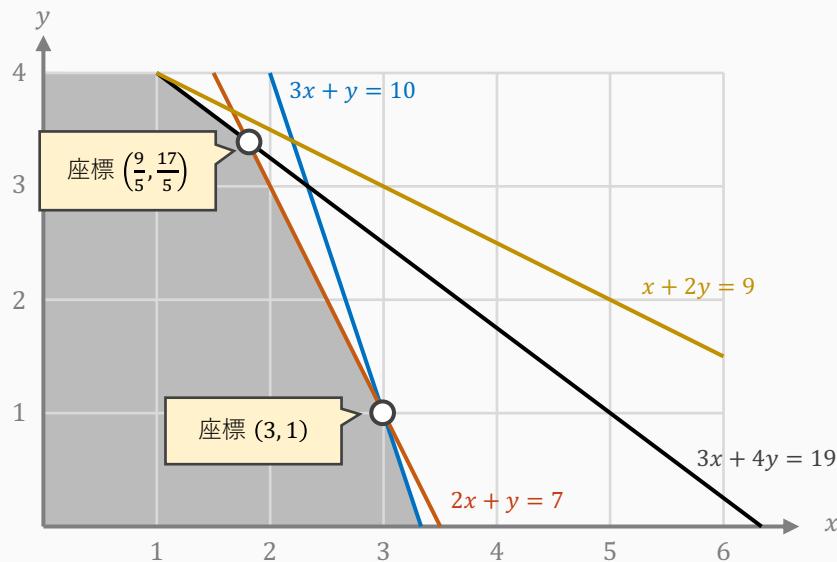
※ Python等原始碼請參閱 chap5-4.md。

5.5

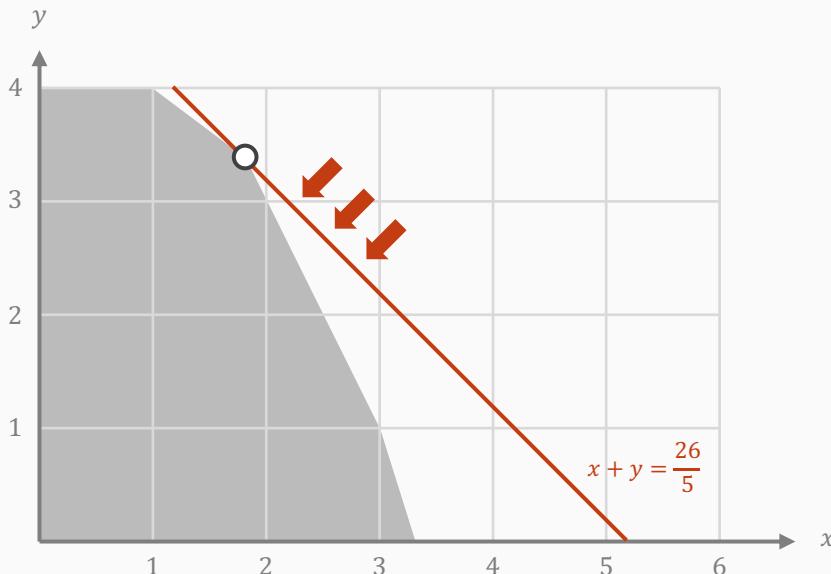
節末問題 5.5 的解答

問題 5.5.1

首先，將 $3x + y = 10$, $2x + y = 7$, $3x + 4y = 19$, $x + 2y = 9$ 的圖畫在同一座標平面上，如下圖所示。灰色區域表示問題中四個條件全部滿足的區域。

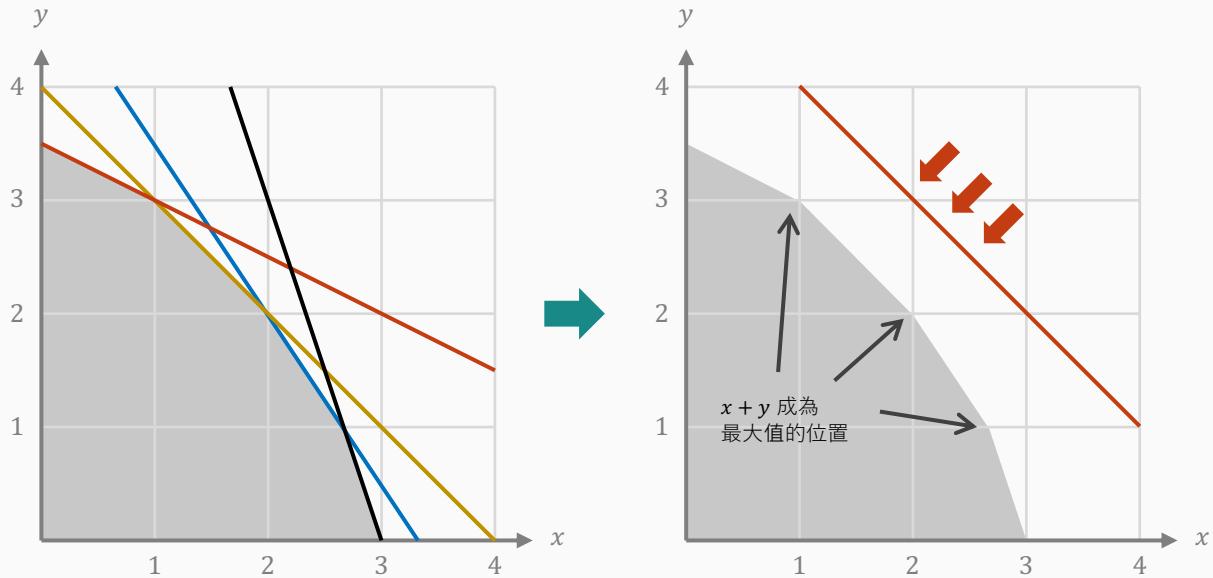


因此，將直線 $x + y = a$ 逐步向下移動吧（將 a 減少看看）。如此一來，由於當 $a = 26/5$ 時首次碰到灰色區域，因此 $x + y$ 的最大值為 $26/5$ 。



問題 5.5.2

在線性規劃問題中， $x + y$ 在兩條直線的交點為最大值。其原因為，滿足條件的部分（相當於下圖的灰色區域）只會在兩條直線的交點處發生轉折。。



因此，可以透過以下演算法求出答案。注意，限制條件中保證在 x, y 為正實數時取最大值。（若非如此，則可能會出現最大值為無限大的情況，需要另外區分）

對於所有整數組 (i, j) $[1 \leq i < j \leq N]$ 進行以下操作：

- 求出直線 $a_i x + b_i y + c_i$ 和直線 $a_j x + b_j y = c_j$ 的交點
- 判斷是否 N 個條件式全部滿足

在滿足條件的交點中， $(x$ 座標 $) + (y$ 座標 $)$ 的值最大的即為答案。

另外，直線 $a_i x + b_i y + c_i$ 和直線 $a_j x + b_j y = c_j$ 的交點可透過以下方法求得：

- $a_i b_j = a_j b_i$ 時：兩條直線平行（不相交）
- $a_i b_j \neq a_j b_i$ 時：交點的座標如下（使用本書未探討之聯立方程式即可導出，有興趣的讀者可以自行調查）

$$\left(\frac{c_i b_j - c_j b_i}{a_i b_j - a_j b_i}, \quad \frac{c_i a_j - c_j a_i}{b_i a_j - b_j a_i} \right)$$

因此，撰寫如下程式即可獲得正解。函數 `check(x, y)` 會在實數組 (x, y) 滿足 N 個條件式全部時回傳 `true`，否則回傳 `false`。

這段程式呼叫 `check` 函數 $O(N^2)$ 次，由於 `check` 函數的計算複雜度為 $O(N)$ ，因此程序的總計算複雜度為 $O(N^3)$ 。

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

int N;
double A[509], B[509], C[509];

bool check(double x, double y) {
    for (int i = 1; i <= N; i++) {
        if (A[i] * x + B[i] * y > C[i]) return false;
    }
    return true;
}

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i] >> B[i] >> C[i];

    // 將交點進行全搜尋
    double Answer = 0.0;
    for (int i = 1; i <= N; i++) {
        for (int j = i + 1; j <= N; j++) {
            // 不具有交點時
            if (A[i] * B[j] == A[j] * B[i]) continue;

            // 求出第 i 條直線（條件式的邊界）與第 j 條直線（條件式的邊界）的交點
            double px = (C[i] * B[j] - C[j] * B[i]) / (A[i] * B[j] - A[j] * B[i]);
            double py = (C[i] * A[j] - C[j] * A[i]) / (B[i] * A[j] - B[j] * A[i]);
            bool ret = check(px - 0.00000001, py - 0.00000001);
            if (ret == true) {
                Answer = max(Answer, px + py);
            }
        }
    }

    // 輸出答案
    printf("%.12lf\n", Answer);
    return 0;
}
```

※ Python等原始碼請參閱 chap5-5.md。

5.6

節末問題 5.6 的解答

問題 5.6.1 (1), (2), (3)

使用 2 枚、1 枚、0 枚 500 日圓硬幣時，100 日圓硬幣和 50 日圓硬幣合計金額分別為 0 日圓、500 日圓、1000 日圓。

因此，各種情況下的支付方法數量如下圖所示。



問題 5.6.1 (4)

思考將問題如右圖所示以「500 日圓硬幣的枚數」來分解。。

如此，根據 (1), (2), (3) 的結果，可知答案為 $1 + 6 + 11 = \textcolor{red}{18}$ 種 方法。

原始問題

使用 500, 100,
50 日圓硬幣
支付 1000
日圓的方法
數量

小問題1

500 日圓硬幣 2 枚

小問題2

500 日圓硬幣 1 枚

小問題3

500 日圓硬幣 0 枚

問題 5.6.2

首先，考慮將問題分解如下。

- 小問題 1：已選擇整數的最大值為 A_1 的選擇方法有幾種？
- 小問題 2：已選擇整數的最大值為 A_2 的選擇方法有幾種？
- 小問題 3：已選擇整數的最大值為 A_3 的選擇方法有幾種？
- \vdots
- 小問題 N ：已選擇整數的最大值為 A_N 的選擇方法有幾種？

此時，所求的答案如下所示。

$$(\text{小問題 1 的答案}) \times A_1 + \dots + (\text{小問題 } N \text{ 的答案}) \times A_N$$

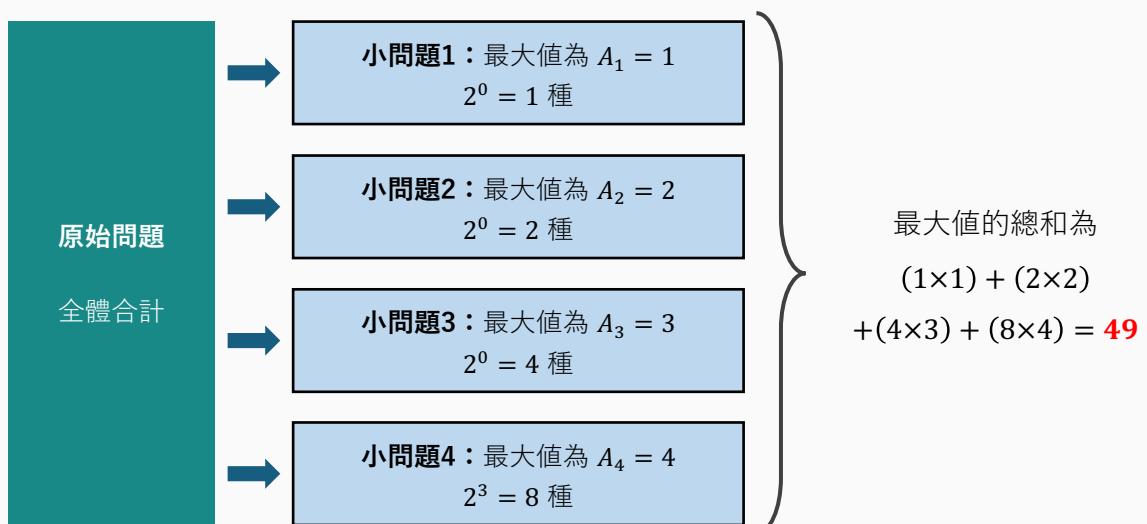
因此，已選擇整數的最大值為 A_i 的選擇方法條件如下。。

- 選擇 A_i 。
- 從 $A_1, A_2, A_3, \dots, A_{i-1}$ 中選擇 0 個以上的整數。

對於 $i - 1$ 個整數可以進行 Yes/No 的選擇，因此根據乘法原理（→ 3.3.2項），選擇方法總共有 2^{i-1} 種。因此，所求答案如下。

$$\sum_{i=1}^N 2^{i-1} \times A_i = (2^0 \times A_1) + (2^1 \times A_2) + \dots + (2^{N-1} \times A_N)$$

例如，當 $N = 4, (A_1, A_2, A_3, A_4) = (1, 2, 3, 4)$ 時，如下圖所示可知答案為 49。



將其以程式實作如下。注意，變數 `power[i]` 是 2^i 除以 1000000007 的餘數。

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

const long long mod = 1000000007;
long long N;
long long A[300009];
long long power[300009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 求出  $2^i$ 
    power[0] = 1;
    for (int i = 1; i <= N; i++) {
        power[i] = (2 * power[i - 1]) % mod;
    }

    // 求出答案
    long long Answer = 0;
    for (int i = 1; i <= N; i++) {
        Answer += power[i - 1] * A[i];
        Answer %= mod;
    }

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

※ Python等原始碼請參閱 chap5-6.md。

5.7

節末問題 5.7 的解答

問題 5.7.1

加法式中 2021 有 4 個， 1234 有 5 個。

因此，所求答案為 $2021 \times 4 + 1234 \times 5 = 8084 + 6170 = \mathbf{14254}$ 。

問題 5.7.2

考慮將優美程度的期望值分解為以下的 ${}_6C_2 = 15$ 個部分。

- 部分 1：由第 1 個和第 2 個骰子的結果算出的優美程度
- 部分 2：由第 1 個和第 3 個骰子的結果算出的優美程度
- 部分 3：由第 1 個和第 4 個骰子的結果算出的優美程度
- 部分 4：由第 1 個和第 5 個骰子的結果算出的優美程度
- ⋮
- 部分 15：由第 5 個和第 6 個骰子的結果算出的優美程度

因此，由於以下原因，每個部分中「算出的優美程度」的期望值為 $1/6$ 。

骰子的點數有右圖中的 $6 \times 6 = 36$ 種等機率的情況。

另一方面，兩個骰子點數相同的情況有 6 種，所以其機率為 $6/36 = 1/6$ 。不瞭解的話請回到 3.4 節確認。

		骰子 2					
		1	2	3	4	5	6
骰子 1	1	○	✗	✗	✗	✗	✗
	2	✗	○	✗	✗	✗	✗
	3	✗	✗	○	✗	✗	✗
	4	✗	✗	✗	○	✗	✗
	5	✗	✗	✗	✗	○	✗
	6	✗	✗	✗	✗	✗	○

因此，所求答案為 $1/6 \times 15 = \mathbf{5/2}$ 。（根據期望值的線性性 [→ 3.4 節]，整體優美的期望值為各部分期望值的和）

問題 5.7.3

首先，考慮 $A_1 \leq A_2 \leq A_3 \leq \dots \leq A_N$ 的情況。這時，以下的式子成立，因此答案與例題 2 (\rightarrow 5.7.3項) 相同。。

$$|A_j - A_i| = A_j - A_i \quad (1 \leq i \leq j \leq N)$$

よって

$$\sum_{i=1}^N \sum_{j=i+1}^N |A_j - A_i| = \sum_{i=1}^N \sum_{j=i+1}^N A_j - A_i$$

另一方面，所求答案是「將不同的兩個元素的差全部相加的值」，所以 $A_1, A_2, A_3, \dots, A_N$ 的順序調換後答案不變。。

例如，當 $A = (1, 4, 2, 3)$ 時的答案是 10，將其排序為 $A = (1, 2, 3, 4)$ 後答案仍是 10。

因此，將數列 $A = (A_1, A_2, \dots, A_N)$ 升冪排序 (\rightarrow 3.6節) 後，進行與例題 2 相同的處理方式，製作以下程式即可得到正解。

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

long long N, A[200009];
long long Answer = 0;

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];

    // 排序 (從程式碼 5.7.1 唯一增加的部分)
    sort(A + 1, A + N + 1);

    // 求出答案 → 輸出答案
    for (int i = 1; i <= N; i++) Answer += A[i] * (-N + 2LL * i - 1LL);
    cout << Answer << endl;
    return 0;
}
```

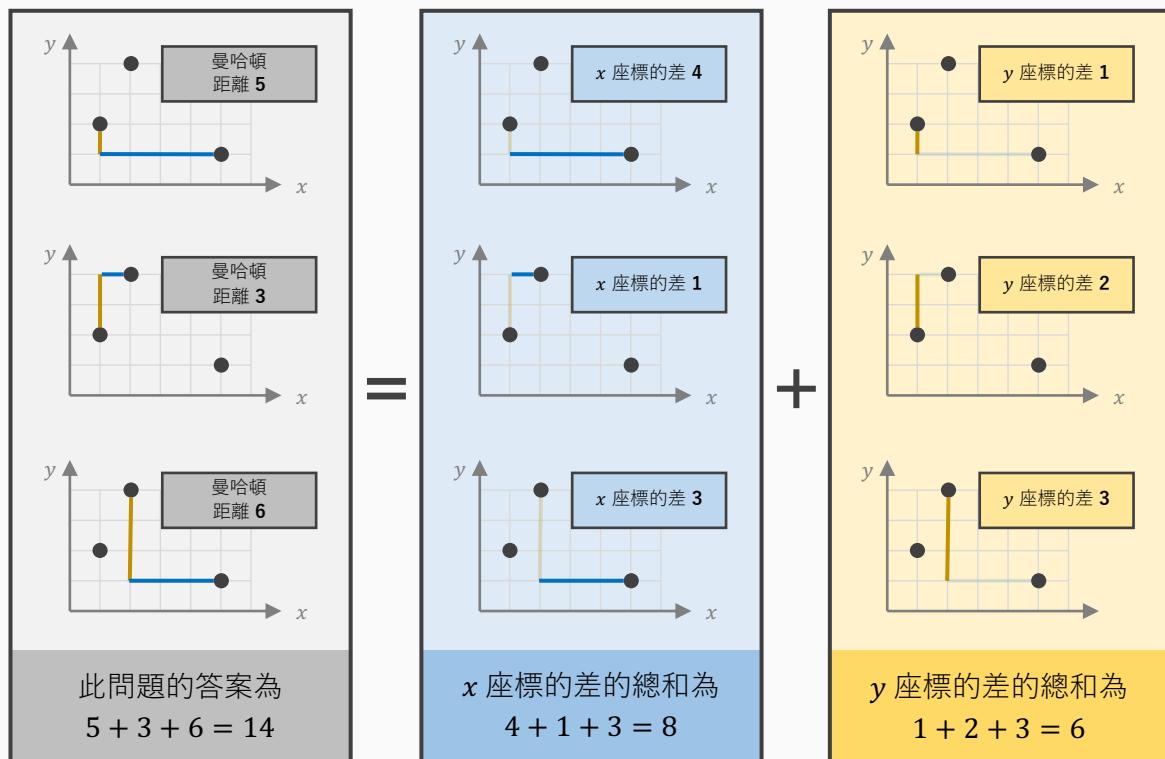
※ Python 等原始碼請參閱 chap5-7.md。

問題 5.7.4

首先，兩點間的曼哈頓距離是 x 座標的差的絕對值與 y 座標的差的絕對值相加的值。因此，所求的「曼哈頓距離的總和」是以下兩部分的答案相加的值。

- 部分 1： x 座標的差的絕對值的總和
- 部分 2： y 座標的差的絕對值的總和

例如，考慮在座標 $(1, 2), (5, 1), (2, 4)$ 的點。曼哈頓距離的總和為 $5 + 3 + 6 = 14$ ， x 座標的差的絕對值的總和為 8， y 座標的差的絕對值的總和為 6。



因此，部分 1 和部分 2 的答案可以用以下式子表示。這與節末問題 5.7.3 相同，若將 (x_1, x_2, \dots, x_N) 和 (y_1, y_2, \dots, y_N) 從小到大排序，之後可用 $O(N)$ 的計算複雜度內求出式子的值。

$$Part1 = \sum_{i=1}^N \sum_{j=i+1}^N |x_i - x_j|$$

$$Part2 = \sum_{i=1}^N \sum_{j=i+1}^N |y_i - y_j|$$

因此，撰寫以下程式即可得到正解。在 C++ 中，可以藉由使用標準庫 `std::sort` 來將陣列元素從小到大排序。（→ 3.6.1項）

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

long long N;
long long X[200009], Y[200009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> X[i] >> Y[i];

    // 將陣列排序
    sort(X + 1, X + N + 1);
    sort(Y + 1, Y + N + 1);

    // 部分 1 的答案 (x 座標的差的絕對值總和)
    long long Part1 = 0;
    for (int i = 1; i <= N; i++) Part1 += X[i] * (-N + 2LL * i - 1LL);

    // 部分 2 的答案 (y 座標的差的絕對值總和)
    long long Part2 = 0;
    for (int i = 1; i <= N; i++) Part2 += Y[i] * (-N + 2LL * i - 1LL);

    // 輸出
    cout << Part1 + Part2 << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap5-7.md。

5.8

節末問題 5.8 的解答

問題 5.8.1

如 5.8.3 項所述，在表示年齡差關係的圖中，令頂點 1 到頂點 i 的最短路徑長度 $dist[i]$ ，則人 i 的年齡最大 $\min(dist[i], 120)$ 。

因此，對於求出最短路徑長度的程式（程式碼 4.5.3），只變更輸出部分並提出如下程式，即可得到正解。

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

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

int main() {
    // 輸入
    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]);
    }

    // 廣度優先搜尋的初始化 (dist[i]==-1時，為未到達的白色頂點)
    for (int i = 1; i <= N; i++) dist[i] = -1;
    queue<int> Q; // 定義佇列 Q
    Q.push(1); dist[1] = 0; // 將 1 添加到 Q 中 (操作 1)

    // 廣度優先搜尋
    while (!Q.empty()) {
        int pos = Q.front(); // 查看 Q 的開頭 (操作2)
        Q.pop(); // 取出 Q 的開頭 (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); // 將 nex 添加到 Q 中 (操作 1)
            }
        }
    }
}
```

```

    }
}

// 輸出從頂點 1 到各頂點的最短距離
// 無法從頂點 1 到達的時候可設為 120 歲
for (int i = 1; i <= N; i++) {
    if (dist[i] == -1) cout << "120" << endl;
    else cout << min(dist[i], 120) << endl;
}
return 0;
}

```

← 從程式碼 4.5.3 變更的部分

※ Python 等原始碼請參閱 chap5-8.md。

問題 5.8.2

這個問題不容易看出解法，所以讓我們先調查 N 較小的情況吧。

- 當 $N = 2$ 時，以 $(P_1, P_2) = (2, 1)$ 的最大分數為 1
- 當 $N = 3$ 時，以 $(P_1, P_2, P_3) = (2, 3, 1)$ 的最大分數為 3
- 當 $N = 4$ 時，以 $(P_1, P_2, P_3, P_4) = (2, 3, 4, 1)$ 的最大分數為 6

(藉由進行全搜尋，用手算也可得知)

$N \geq 5$ 時也一樣，令 $(P_1, P_2, \dots, P_{N-1}, P_N) = (2, 3, \dots, N, 1)$ ，則分數如下（和的公式 → 2.5.10項）。

$$\sum_{i=1}^N (i \bmod P_i) = 1 + 2 + 3 + \dots + (N-1) + 0 = \frac{N(N-1)}{2}$$

但是，這真的是最大值嗎？答案是肯定的，可以證明如下。

首先，為了簡單起見，證明 $N = 4$ 的最大值為 6。

$$\sum_{i=1}^4 (i \bmod P_i) = (1 \bmod P_1) + (2 \bmod P_2) + (3 \bmod P_3) + (4 \bmod P_4)$$

雖然分數是計算如上式，但依據 mod 的性質可以說明以下幾點：

- $1 \bmod P_1$ 為 $P_1 - 1$ 以下
- $2 \bmod P_2$ 為 $P_2 - 1$ 以下
- $3 \bmod P_3$ 為 $P_3 - 1$ 以下
- $4 \bmod P_4$ 為 $P_4 - 1$ 以下

因此，分數是將這些相加的值 $P_1 + P_2 + P_3 + P_4 - 4$ 以下。

但是， (P_1, P_2, P_3, P_4) 是 $(1, 2, 3, 4)$ 的調換排列，因此如下所示：

$$P_1 + P_2 + P_3 + P_4 = 1 + 2 + 3 + 4 = 10$$

$$P_1 + P_2 + P_3 + P_4 - 4 = 1 + 2 + 3 + 4 - 4 = 6$$

可知分數為 6 以下。所以，分數在 $(P_1, P_2, P_3, P_4) = (2, 3, 4, 1)$ 時的最大值為 6。

一般情況的證明

可以用和 $N = 4$ 時相同的方法證明。首先， $i \bmod P_i \leq P_i - 1$ 成立，故分數為 $(P_1 - 1) + (P_2 - 1) + \dots + (P_N - 1) = P_1 + \dots + P_N - N$ 以下。

另一方面， $(P_1, P_2, P_3, \dots, P_N)$ 是 $(1, 2, 3, \dots, N)$ 的調換排列，因此如下所示：

$$P_1 + P_2 + \dots + P_N = 1 + 2 + \dots + N = \frac{N(N+1)}{2}$$

$$P_1 + P_2 + \dots + P_N - N = \frac{N(N+1)}{2} - N = \frac{N(N-1)}{2}$$

可知分數為 $N(N - 1)/2$ 以下。

因此，提出將 $N(N - 1)/2$ 輸出的程式即可得到正解。

本問題的限制為 $N \leq 10^9$ 之大，答案可能超過 10^{17} ，因此注意使用 `int` 型態等 32 位元整數的話可能會發生溢出。

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

int main() {
    // 輸入
    long long N;
    cin >> N;

    // 輸出
    cout << N * (N - 1) / 2 << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap5-8.md。

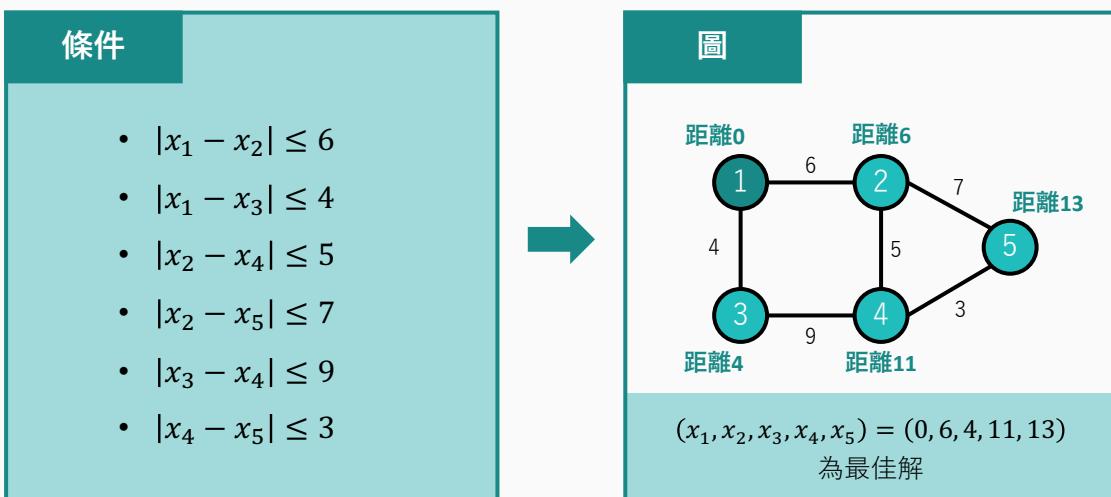
問題 5.8.3

注意：此問題使用4.5.8項「其他代表性的圖演算法」介紹的Dijkstra演算法。初學者無法解決是自然的，不必擔心。

首先，思考如下的加權無向圖。

- 有 N 個頂點，從 1 編號到 N 。
- 有 M 條邊，根據條件 $|x_{A_i} - x_{B_i}| \leq C_i$ ，添加連接頂點 A_i 和 B_i 的邊，權重為 C_i 。

這時， x_N 的最大值為從頂點 1 到頂點 N 的最短路徑長度 $dist[N]$ 。具體例如下，令 $x_i = dist[i]$ ，則確實滿足 M 個條件式。（本解說不會探討詳細證明，但可使用與無權重圖相同的方法進行證明）



因此，使用Dijkstra法求出頂點 1 到頂點 N 的最短路徑長度 $dist[N]$ ，並提出將其輸出的程式如下，即可得到正解。

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

long long N, M;
long long A[500009], B[500009], C[500009];

// 圖
bool used[500009];
long long dist[500009]; // dist[i] 為頂點 1 → 頂點 i 的最短路徑長度
vector<pair<int, long long>> G[500009];
priority_queue<pair<long long, int>, vector<pair<long long, int>>, greater<pair<long long, int>>> Q;
```

```

// Dijkstra法
void dijkstra() {
    // 陣列的初始化等
    for (int i = 1; i <= N; i++) dist[i] = (1LL << 60);
    for (int i = 1; i <= M; i++) used[i] = false;
    dist[1] = 0;
    Q.push(make_pair(0, 1));

    // 佇列更新
    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; // 頂點 0 的時候是例外
            if (dist[to] > cost) {
                dist[to] = cost;
                Q.push(make_pair(dist[to], to));
            }
        }
    }
}

int main() {
    // 輸入、添加圖的邊
    cin >> N >> M;
    for (int i = 1; i <= M; i++) {
        cin >> A[i] >> B[i] >> C[i];
        G[A[i]].push_back(make_pair(B[i], C[i]));
        G[B[i]].push_back(make_pair(A[i], C[i]));
    }

    // Dijkstra法
    dijkstra();

    // 輸出答案
    if (dist[N] == (1LL << 60)) cout << "-1" << endl;
    else cout << dist[N] << endl;
    return 0;
}

```

※ Python等原始碼請參閱 chap5-8.md。

5.9

節末問題 5.9 的解答

問題 5.9.1

在程式碼 5.9.1 中，如下使用了 while 敘述來計算「支付到極限時的紙幣數量」。

```
while (N >= 10000) { N -= 10000; Answer += 1; }
while (N >= 5000) { N -= 5000; Answer += 1; }
while (N >= 1) { N -= 1000; Answer += 1; }
```

這也可以用除法來計算。當剩餘金額為 N 日元時，可用的紙幣數量最大值如下表所示。

紙幣種類	10000 日圓鈔	5000 日圓鈔	1000 日圓鈔
可用最大數量	$\left\lfloor \frac{N}{10000} \right\rfloor$ 枚	$\left\lfloor \frac{N}{5000} \right\rfloor$ 枚	$\left\lfloor \frac{N}{1000} \right\rfloor$ 枚
支付到極限時的 剩餘金額	$N \bmod 10000$ 日圓	$N \bmod 5000$ 日圓	$N \bmod 1000$ 日圓

因此，撰寫如下程式，即可以計算複雜度 $O(1)$ 求出答案。即使是 $N = 10^{18}$ 程度的輸入，也能瞬間執行完畢。

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

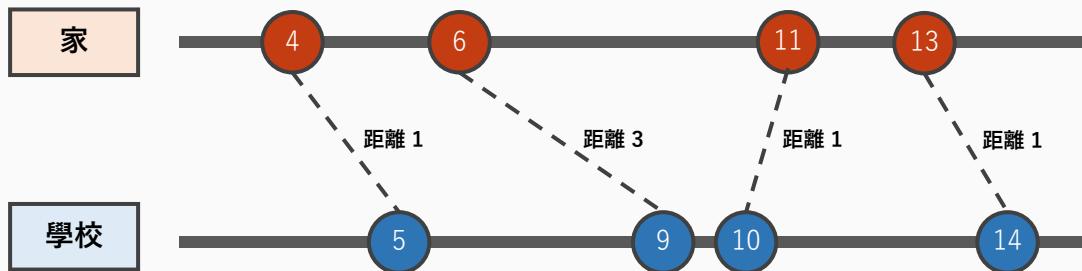
int main() {
    // 輸入
    long long N, Answer = 0;
    cin >> N;

    // 支付 10000 日圓鈔
    Answer += (N / 10000); N %= 10000;
    // 支付 5000 日圓鈔
    Answer += (N / 5000); N %= 5000;
    // 支付 1000 日圓鈔
    Answer += (N / 1000); N %= 1000;

    // 輸出答案
    cout << Answer << endl;
    return 0;
}
```

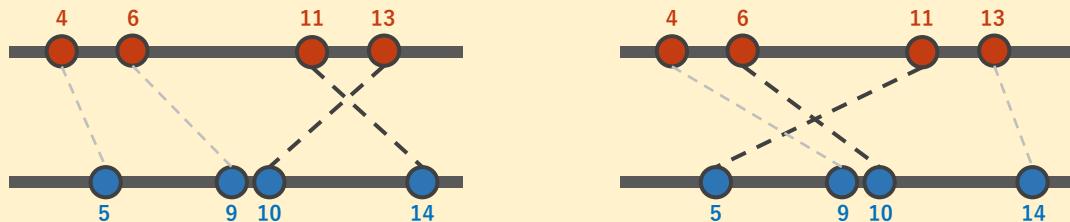
問題 5.9.2

如下圖所示，若將「從左數來第 1 家與從左數來第 1 所小學」、「從左數來第 2 家與從左數來第 2 所小學」……「從左數來第 N 家與從左數來第 N 所小學」連接，則家與就讀學校的距離總和為最小。

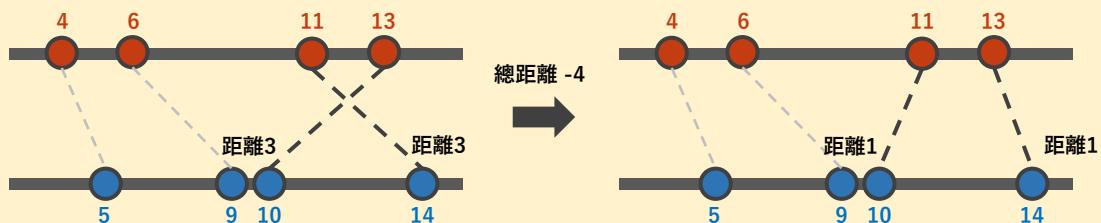


這一事實可以如下證明（因為難度有點高，可跳過不讀）。

首先，除了上述方法（以下記為方法 A）之外，所有的連接方法都至少存在一處「交叉點」。



此外，若將交叉點的連接更換而消除交叉，總距離會減少或不變。下圖為其中一例。



然後，反覆進行消除交叉的操作直到無法進行，則最終一定會變成方法 A^{*}。因此，可以證明不存在比方法 A 距離更短的連接方式（家及學校的分配方式）。

^{*}可以從交叉點線的組數（最大 $N C_2$ 組）一定減少 1 組以上來證明。

因此，將陣列 (A_1, A_2, \dots, A_N) 依小到大的順序排列為 $(A'_1, A'_2, \dots, A'_N)$ ，將陣列 (B_1, B_2, \dots, B_N) 依小到大的順序排列為 $(B'_1, B'_2, \dots, B'_N)$ ，距離總和可以用下式表示。

$$\sum_{i=1}^N |A'_i - B'_i| = \underbrace{|A'_1 - B'_1|}_{\text{從左數來第1家與從左數來第1所小學的距離}} + |A'_2 - B'_2| + \dots + \underbrace{|A'_N - B'_N|}_{\text{從左數來第N家與從左數來第N所小學的距離}}$$

因此，提出如下程式即可得到正解。

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

long long N;
long long A[100009], B[100009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = 1; i <= N; i++) cin >> B[i];

    // 排序
    sort(A + 1, A + N + 1);
    sort(B + 1, B + N + 1);

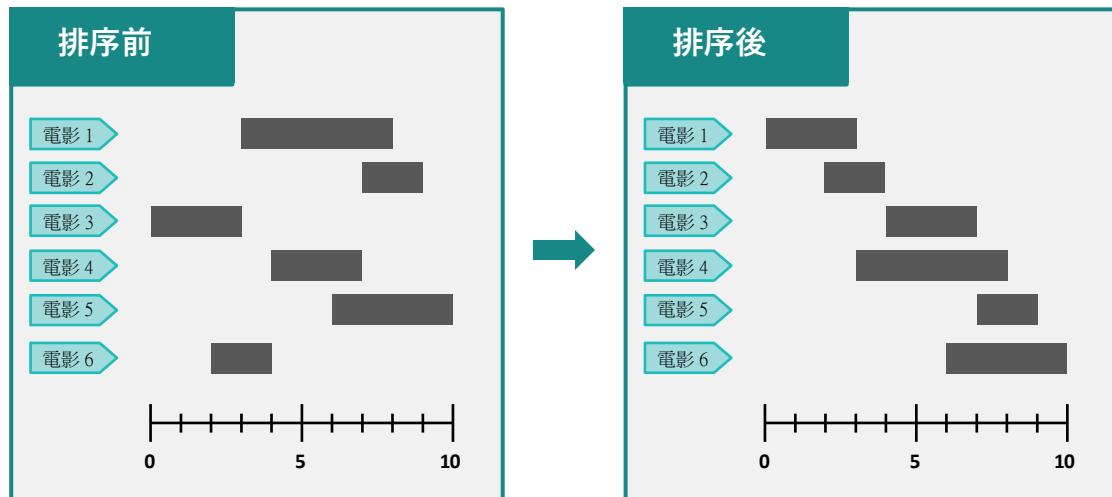
    // 求出答案
    long long Answer = 0;
    for (int i = 1; i <= N; i++) Answer += abs(A[i] - B[i]);
    cout << Answer << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap5-9.md。

問題 5.9.3

程式碼 5.9.2 的演算法確實可以得到正確答案，但計算速度較慢。因為在調查「目前可選擇的電影之中結束時間最早的電影」時需要計算複雜度 $O(N)$ ，所以在能夠選擇總共 N 部電影的情況下，計算複雜度為 $O(N_2)$ 。究竟該如何提高速度呢？。

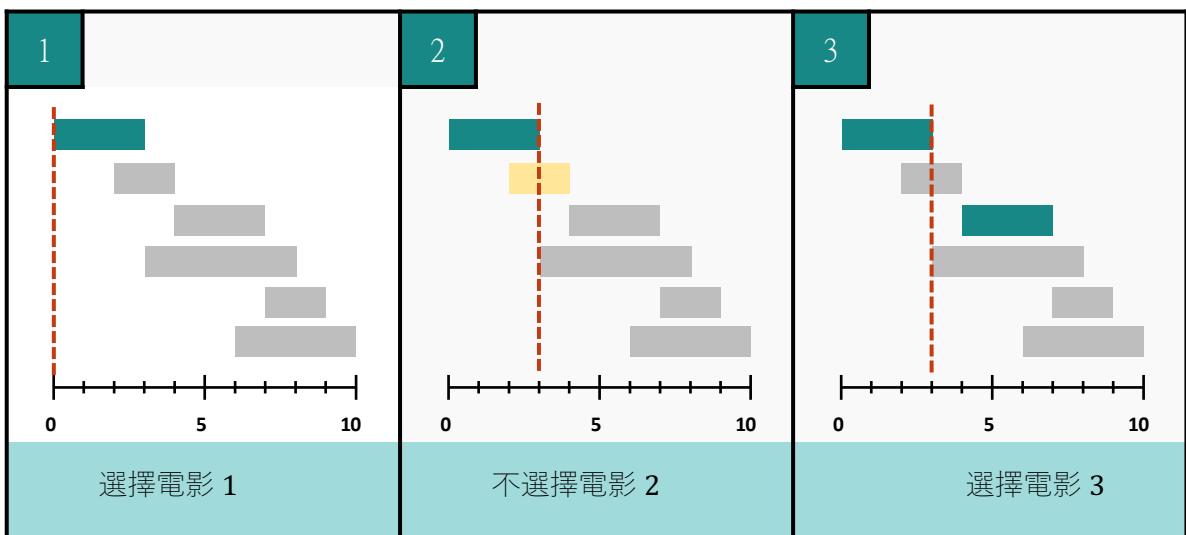
因此，將電影按結束時間早晚排序，將最早結束的電影設為「電影 1」，最晚結束的設為「電影N」。

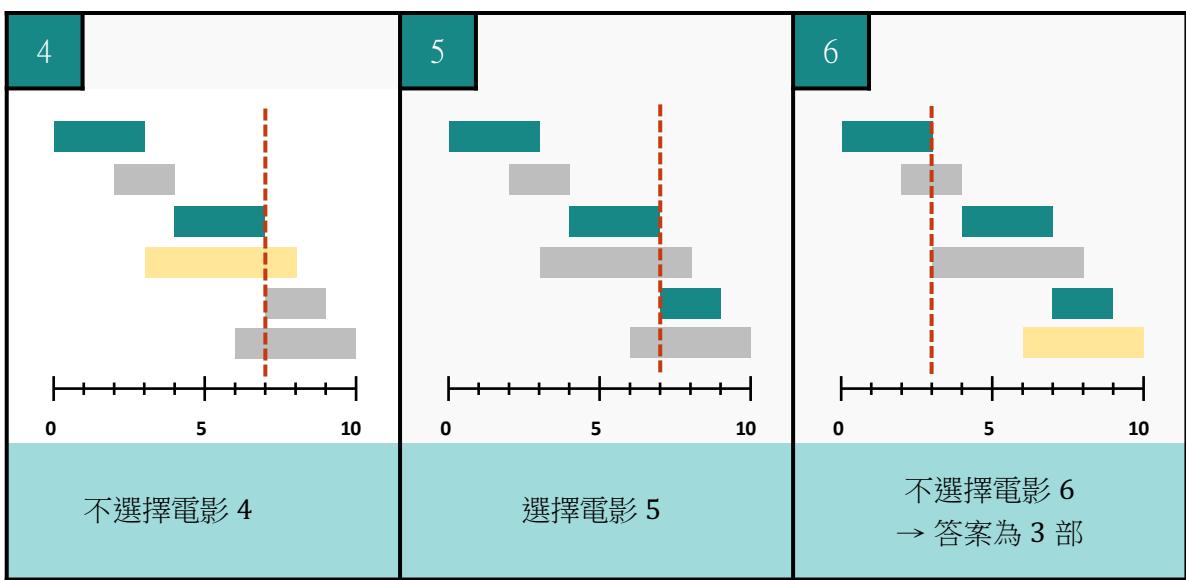


如此，用以下的演算法，可以有效率地持續選擇「最早結束的電影」。

- 選擇電影 1。
- 若（在開始時間）可以選擇電影 2，則選擇。
- 若（在開始時間）可以選擇電影 3，則選擇。
- :
- 若（在開始時間）可以選擇電影 N，則選擇。

若將此演算法運用於具體的例子中，會如下所示。





例如，C++ 的實作例如下。為了依照結束時間早晚排序，使用了 Movie 型態。當 Movie 型態的變數 A 存在時，代表：

- A.l：電影的開始時間
- A.r：電影的結束時間

A.r較大者會被判定為「大」（`bool operator<`的部分）。因此，透過 `sort` 函數將 A.r 按小到大排序。

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

// Movie 型態
struct Movie {
    int l, r;
};

// Movie 型態的比較函數
bool operator<(const Movie &a1, const Movie &a2)
{
    if (a1.r < a2.r) return true;
    if (a1.r > a2.r) return false;
    if (a1.l < a2.l) return true;
    return false;
}

int N;
Movie A[300009];
int CurrentTime = 0; // 現在時間（最後所選電影的結束時間）
int Answer = 0; // 在現在已看過的電影數

int main() {
    // 輸入
    cin >> N;
```

```
for (int i = 1; i <= N; i++) cin >> A[i].l >> A[i].r;

// 排序
sort(A + 1, A + N + 1);

// 持續選擇結束時間最早的電影
for (int i = 1; i <= N; i++) {
    if (CurrentTime <= A[i].l) {
        CurrentTime = A[i].r;
        Answer += 1;
    }
}

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

※ Python 等原始碼請參閱 chap5-9.md 。

5.10

節末問題 5.10 的解答

問題 5.10.1

使用分配律（[→5.10.2項](#)）可以如以下所示輕鬆地計算而解決。

問題 (1)

$$\begin{aligned} & 37 \times 39 + 37 \times 61 \\ &= 37 \times (39 + 61) \\ &= 37 \times 100 \\ &= \mathbf{3700} \end{aligned}$$

問題 (2)

$$\begin{aligned} & 2021 \times 333 + 2021 \times 333 + 2021 \times 334 \\ &= 2021 \times (333 + 333 + 334) \\ &= 2021 \times 1000 \\ &= \mathbf{2021000} \end{aligned}$$

問題 5.10.2

$$1 + 2 + \cdots + N \times 1 + 2 + \cdots + N =$$

本問題是例題2（[→5.10.2項](#)）的一般化。使用分配律，可知以下有關求和符號式子的特徵： \times

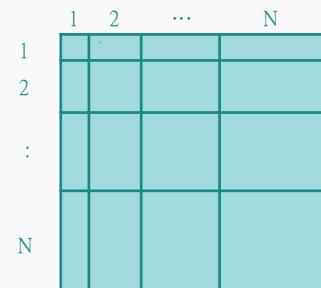
- 當 $i = 1$ 時的總和： $(1 \times 1) + (1 \times 2) + \cdots + (1 \times N) = \mathbf{1 \times (1 + 2 + \cdots + N)}$
- 當 $i = 2$ 時的總和： $(2 \times 1) + (2 \times 2) + \cdots + (2 \times N) = \mathbf{2 \times (1 + 2 + \cdots + N)}$
- \vdots
- 當 $i = N$ 時的總和： $N \times 1 + N \times 2 + \cdots + N \times N = \mathbf{N \times (1 + 2 + \cdots + N)}$

所求的答案是藍色顯示的值的總和，因此，根據分配律，結果如下。

$$\sum_{i=1}^N \sum_{j=1}^N ij = (1 + 2 + \cdots + N) \times (1 + 2 + \cdots + N) = \frac{N(N+1)}{2} \times \frac{N(N+1)}{2}$$

對此沒有概念的話，可以思考右圖中的正方形面積。

此正方形的縱長是 $N(N+1)/2$ ，橫長也是 $N(N+1)/2$ 。



因此，提出如以下輸出答案的程式即為正解。此外，這個問題的限制是 $N \leq 10^9$ 之大，因此 $N(N+1)/2 \times N(N+1)/2$ 的值可能超過 10^{30} 。注意若不進行在計算過程中取餘數（→4.6.1項）等處理，即使使用 `long long` 型態的 64 位元整數，也有可能會溢出。

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

const long long mod = 1000000007;
long long N;

int main() {
    // 輸入
    cin >> N;

    // 求出答案
    long long val = N * (N + 1) / 2;
    val %= mod;
    cout << val * val % mod << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 `chap5-10.md`。

問題 5.10.3

對此思考如下的立方體，可知本問題的答案是：

$$\sum_{i=1}^A \sum_{j=1}^B \sum_{k=1}^N ijk = (1 + \dots + A)(1 + \dots + B)(1 + \dots + C)$$

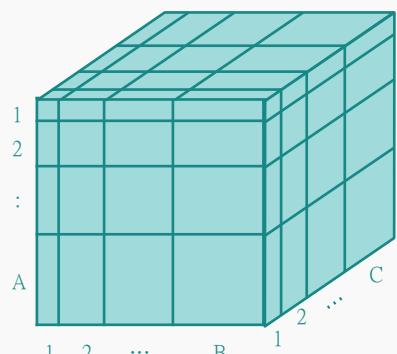
因此，根據和的公式（→2.5.10項）以下式子會成立。

$$1 + 2 + \dots + A = \frac{A(A + 1)}{2}$$

$$1 + 2 + \dots + B = \frac{B(B + 1)}{2}$$

$$1 + 2 + \dots + C = \frac{C(C + 1)}{2}$$

所以答案為 $\frac{A(A + 1)}{2} \times \frac{B(B + 1)}{2} \times \frac{C(C + 1)}{2}$ 。



因此，提出如下將答案輸出的程式即為正解。另外，本問題的限制是 $A, B, C \leq 10^9$ ，因為很大，為了防止溢出，將變數設定如下：

- $D = A(A + 1)/2$
- $E = B(B + 1)/2$
- $F = C(C + 1)/2$

並進行在計算過程中取餘數等處理。

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

const long long mod = 998244353;
long long A, B, C;

int main() {
    // 輸入
    cin >> A >> B >> C;

    // 計算
    long long D = A * (A + 1) / 2; D %= mod;
    long long E = B * (B + 1) / 2; E %= mod;
    long long F = C * (C + 1) / 2; F %= mod;

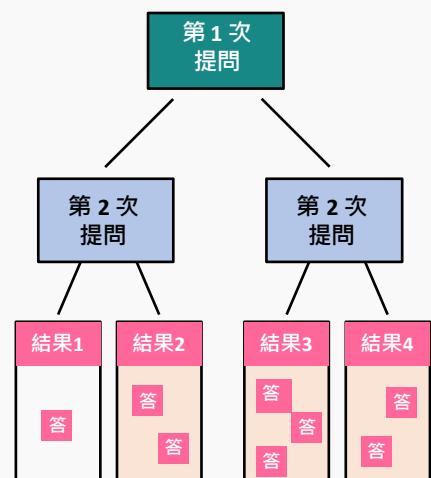
    // 輸出答案
    // 即使在此計算  $(D * E * F) \% mod$ ，可能在途中處理到  $10^{27}$ 
    // 因此，注意即使是 long long 型態也會發生溢出！
    cout << (D * E % mod) * F % mod << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap5–10.md 。

問題 5.10.4

首先，太郎想到的數字有 8 種可能性，但對於兩次提問的回答組合只有「Yes→Yes」、「Yes→No」、「No→Yes」、「No→No」這四種。

由於 $8 > 4$ ，因此如右圖所示，「不確定結果為1種的情況」一定會存在。因此，無法透過兩次問題來確定答案。



問題 5.10.5

若自然地進行實作，會如下所示。

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

int main() {
    // 輸入
    double a, b, c;
    cin >> a >> b >> c;

    // 計算左邊和右邊
    double v1 = log2(a);
    double v2 = b * log2(c);

    // 輸出
    if (v1 < v2) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

但是，若提出這個程式，100 個案例中有 15 個案例是不正確的。其原因是誤差（→5.10.1項）。

例如，在 $(a, b, c) = (10^{18} - 1, 18, 10)$ 的情況下，真正的答案是 Yes，但卻錯誤的輸出了 No。實際上如下。

$$\log_2 a = 59.7947057079725222602 \dots$$

$$b \log_2 c = 59.7947057079725222616 \dots$$

左邊和右邊的相對誤差約為 10^{-19} 左右。因為過於接近，超出了電腦的極限，被判定成「是相同的數字」。

改善方法①

左邊那麼，該如何防止由誤差導致的不正確呢？一種方法是全部用整數來處理。根據對數的性質（→2.3.10項）以下成立。

因為 $b \log_2 c = \log_2(c^b)$

當 $\log_2 a < b \log_2 c$ 時， $\log_2 a < \log_2(c^b)$

因此， $a < c^b$

即使只取 log 的中心大小關係也不會改變

若 $a < c^b$ 則輸出 Yes，若非如此則輸出 No。

將此方法進行實作如下：

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

int main() {
    // 輸入
    long long a, b, c;
    cin >> a >> b >> c;

    // 計算右邊 ( c 的 b 次方 )
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        v *= c;
    }

    // 輸出
    if (a < v) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

但是，這會成為非正解。原因是溢出（[→5.10.1項](#)）。此程式會直接計算 c^b 的值，但由於限制是 $a, b, c \leq 10^{18}$ ，在最壞情況下需要計算 10^{18} 的 10^{18} 次方。不用說C++了，連Python也無法進行計算。

改善方法②

接著，如何防止溢出呢？典型的方法是「在計算過程中取餘數」等，但這個問題不是餘數計算，因此這種方法不適用。

因此，由於在計算乘方的過程中，右邊的值超過 a的當下就可以確定為 Yes，故將迴圈中止的對策是有效的。自然地實作如下：

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

int main() {
    // 輸入
```

```

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

int main() {
    // 輸入
    long long a, b, c;
    cin >> a >> b >> c;

    // 計算右邊 (c 的 b 次方)
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        if (a / c < v) {
            // 輸此條件分支只是將 a < (v * c) 換句話說
            // 行條件換算的理由是 v, c 可能會達到 10^18 左右
            // 若進行 a < v * c · 最差的情況下 v * c = 10^36 而產生溢出
            // 注：long long 型態的極限為2^{63}-1 (約 10^{19})
            cout << "Yes" << endl;
            return 0;
        }
        v *= c;
    }

    // 迴圈無法中止的情況
    cout << "No" << endl;
    return 0;
}

```

但是，這個程式在 100 個測試案例中有 2 個超過了執行時間限制 (TLE)。原因是 $c = 1$ 的案例。

若例如 $(a, b, c) = (2, 10^{18}, 1)$ 的情況。1 的任何次方都是 1，故「目前右邊的值 v 超過 a 就中止」的處理沒有作用。因此，仍然進行了 $b = 10^{18}$ 次迴圈。

此外，由於 $2^{60} > 10^{18}$ ，因此當 $c \geq 2$ 時，一定可以在 60 次迴圈內完成處理。

改善方法③

最後，對於 $c = 1$ 的情況區分處理。由於此問題的限制是 $a \geq 1$ ，因此當 $c^b = 1$ 時，答案一定是 No。因此，撰寫如下程式後，總算可獲得正解。

```

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

int main() {
    // 輸入
    long long a, b, c;
    cin >> a >> b >> c;

    // 區分 c = 1 時的狀況
    if (c == 1) {
        cout << "No" << endl;
        return 0;
    }

    // 計算右邊 (c 的 b 次方)
    long long v = 1;
    for (long long i = 1; i <= b; i++) {
        if (a / c < v) {
            // 此條件分支只是將 a < (v * c) 換句話說
            // 進行條件換算的理由是 v, c 可能會達到 10^18 左右
            // 若進行 a < v * c · 最差的情況下 v * c = 10^36 而產生溢出
            cout << "Yes" << endl;
            return 0;
        }
        v *= c;
    }

    // 迴圈無法中止的情況
    cout << "No" << endl;
    return 0;
}

```

※ Python 等原始碼請參閱 chap5-10.md 。

問題 5.10.6

若首先，可以考慮分別對 $m = 1, 2, \dots, N$ 進行調查的方法，但由於限制是 $N \leq 10^{11}$ 之大，執行時間限制會超過（TLE）。

也就是說， m 可能的模式數遠大於 $f(m)$ 可能的模式數，故以下演算法更有效率。

- 列舉所有可能的 $f(m)$ 候選。
- 若確定 $f(m)$ 則也確定 $m = f(m) + B$ ，因此對每個候選檢查 m 的每位數字的乘積是否與 $f(m)$ 一致。

那麼該如何列舉 $f(m)$ 的候選呢？其實，只需對單調增加數字 m ，如 1123 或 12233599 的 $f(m)$ 即可。因為即使數字順序調換也不失普遍性，例如以下結果全部相同。

- $m = 1123$ のとき $f(m) = 1 \times 1 \times 2 \times 3 = 6$
- $m = 2131$ のとき $f(m) = 2 \times 1 \times 3 \times 1 = 6$
- $m = 3112$ のとき $f(m) = 3 \times 1 \times 1 \times 2 = 6$

此外，單調增長的 11 位以內的數字約有 30 萬個，全部列舉是現實可行的。

因此，撰寫如下程式可以得到正解。又，單調增長數字 m 用遞迴函數 `func(digit, m)` 進行全列舉，`digit` 代表目前的位數。若不瞭解遞迴函數，可以回到第3.6節確認。

此外，函數 `product(m)` 會回傳整數 m 每位數字的乘積。藉由不斷將數字除以10來計算每位數字。與轉換成二進制的演算法（[→2.1.9項](#)）類似。

※注意：此 C++ 程式使用了本書未討論的 `set` 型態。不瞭解的人可以上網調查。
(GitHub 上刊載的 Python、JAVA 的原始碼也使用了 `set` 型態)

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

// 作為 f(m) 可能有的候選
// 關於 set 型態，請在網路上查詢！
set<long long> fm_cand;

// 回傳 m 的各個位數乘積的函式
long long product(long long m) {
    if (m == 0) {
        return 0;
    }
    else {
        long long ans = 1;
        while (m >= 1) {
            ans *= (m % 10);
            m /= 10;
        }
        return ans;
    }
}
```

```

    }

}

void func(int digit, long long m) {
    // m 的位數為 11 位以下
    // 注：如果用 1 填充剩餘的位數，可以假設全部都是 11 位。
    int min_value = (m % 10);
    for (int i = min_value; i <= 9; i++) {
        // 10 * m + i 是在 m 之後附帶數字 i 的值
        func(digit + 1, 10 * m + i);
    }
}

int main() {
    // 列舉 f(m) 的候選
    func(0, 0);

    // 輸入
    long long N, B;
    cin >> N >> B;

    // 檢查是否為 m - f(m) == B
    long long Answer = 0;
    for (long long fm : fm_cand) {
        long long m = fm + B;
        long long prod_m = product(m);
        if (m - prod_m == B && m <= N)
        {
            Answer += 1;
        }
    }

    // 輸出
    cout << Answer << endl;
    return 0;
}

```

※ Python 等原始碼請參閱 chap5–10.md 。

問題 5.10.7 (1)

此問題可以想到多種解法，因此介紹其中一種。

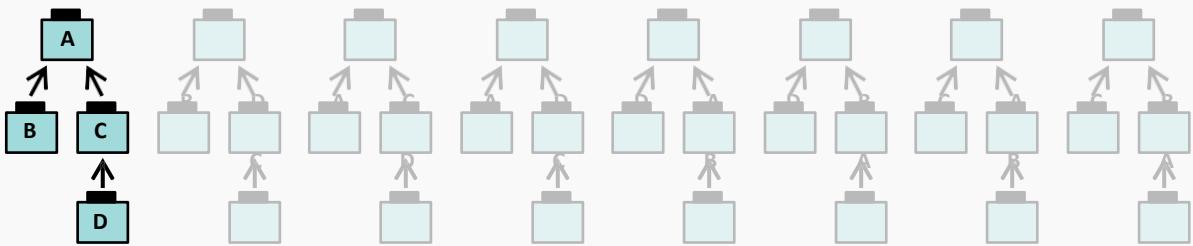
另外，為了方便說明，在每個砝碼上標記 A、B、C、D、E。



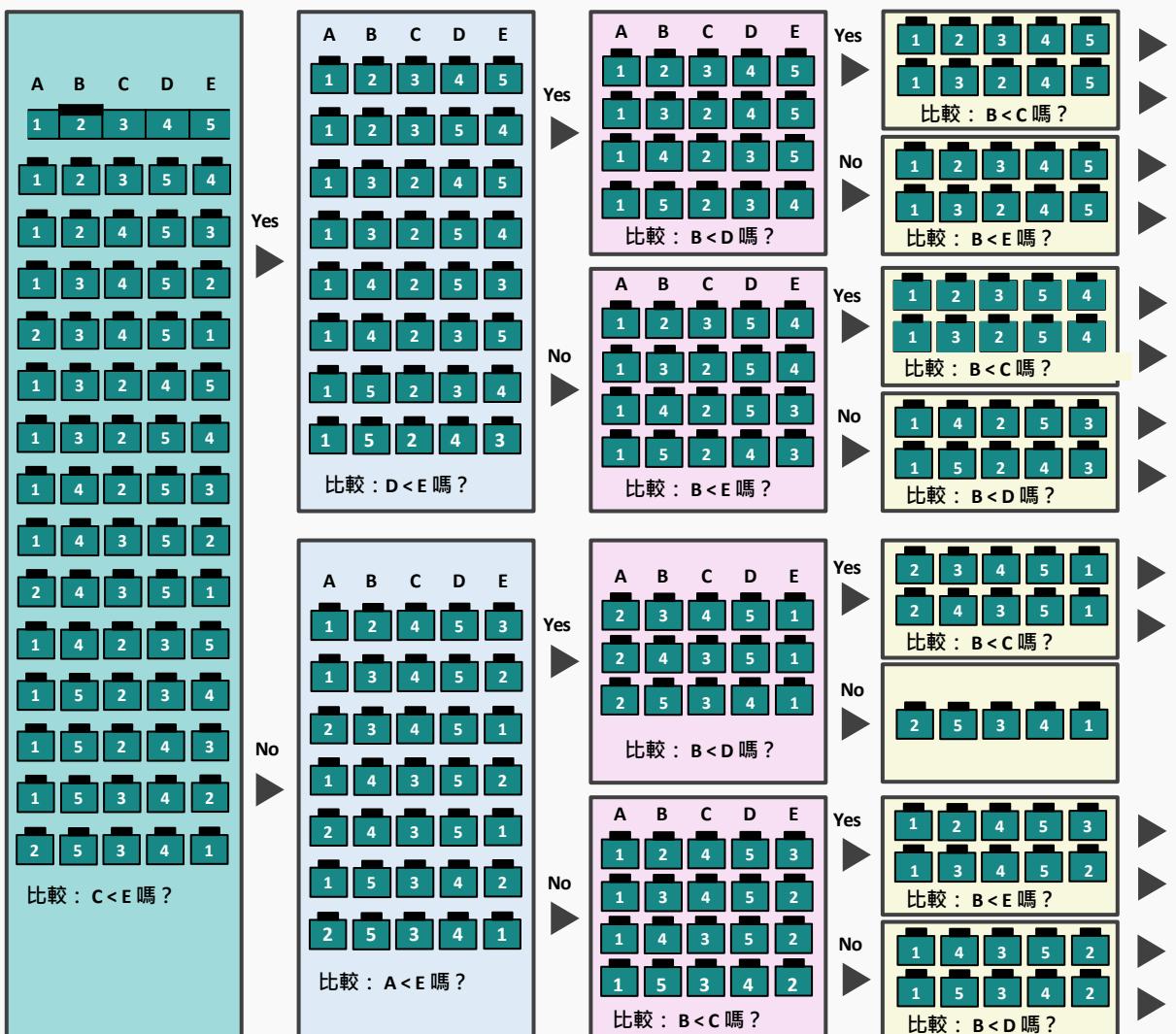
首先，前三次進行如下比較：

1. 比較砝碼 A 和砝碼 B。
2. 比較砝碼 C 和砝碼 D。
3. 比較 1 較輕的一方和 2 較輕的一方

首 3 次提問的結果有以下 8 種可能，但根據對稱性，即使假設「砝碼 A 最輕，砝碼 D 比砝碼 C 重」這種最左邊的模式，也不失普遍性。



接下來，最左邊的模式下的砝碼重量組合有 15 種，但藉由以下的比較，一定可以在 4 次確定答案。（數字為重量 [kg]）



問題 5.10.7 (2)

砝碼的重量組合有 $5! = 120$ 種，6 次比較結果（左側、右側哪一邊比較重）的組合有 $2^6 = 64$ 種。因為後者較小，故無法用在 6 次內確定答案。

問題 5.10.7 (3)

首先，如以下說明，可證明最小次數為 45 次以上。

設砝碼的重量組合有 P 種，為了進行 L 次比較，必須滿足 $2^L \geq P$ ，也就是 $L \geq \log_2 P$ 。

因此，砝碼為 16 個時， $\log_2 P = \log_2 16! = 44.2501 \dots$ ，因此至少需要 45 次比較。

那麼，最小次數是多少呢？首先，將合併排序（→3.6節）運用到本問題上，進行以下操作：

- 「合併兩列包含 1 個砝碼的序列」 × 8 次
- 「合併兩列包含 2 個砝碼的序列」 × 4 次
- 「合併兩列包含 4 個砝碼的序列」 × 2 次
- 「合併兩列包含 8 個砝碼的序列」 × 1 次

回對包含 l 個砝碼的序列進行的合併操作需 $2l - 1$ 次比較（→3.6.10項），因此總計比較次數 $(1 \times 8) + (3 \times 4) + (7 \times 2) + (15 \times 1) = 49$ 次。

此外，截至 2021 年 12 月，研究出了在 46 次以內確定的方法。想知道詳細內容的讀者，請參閱以下論文。

- Peczarski, Marcin (2011). "Towards Optimal Sorting of 16 Elements". *Acta Universitatis Sapientiae*. 4 (2): 215–224.

然而，最小次數到底是 45 次還是 46 次還未確定。有興趣的話請試著挑戰此未解決問題吧。

第 6 章

最終確認問題

6.1

最終確認問題 1-5 的解答

問題 1

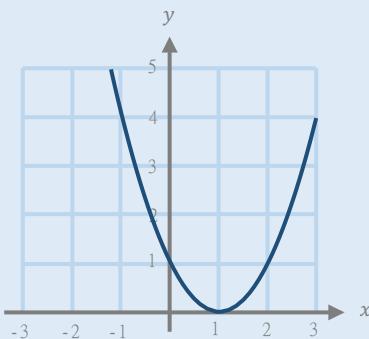
1. $a + 2b + 3c + 4d = (1 \times 12) + (2 \times 34) + (3 \times 56) + (4 \times 78)$
 $= 12 + 68 + 168 + 312$
 $= \mathbf{560}$
2. $a^2 + b^2 + c^2 + d^2 = 12^2 + 34^2 + 56^2 + 78^2$
 $= 144 + 1156 + 3136 + 6084$
 $= \mathbf{10520}$
3. $abcd \bmod 10 = (12 \times 34 \times 56 \times 78) \bmod 10$
 $= (2 \times 4 \times 6 \times 8) \bmod 10$
 $= 384 \bmod 10 = \mathbf{4}$
4. $\sqrt{b+d-a} = \sqrt{34+78-12}$
 $= \sqrt{100} = \mathbf{10}$

此外，在 3 中，使用了在計算途中取餘數也能得到正確答案的性質（[→4.6.1項](#)）。

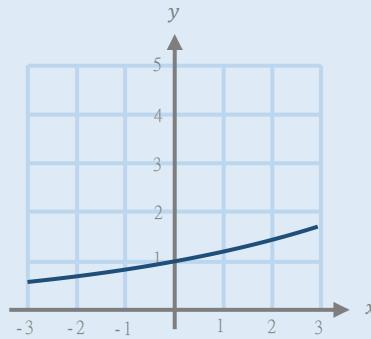
問題 2

答案如下。不瞭解的人可以回到函數（[→2.3節](#)）確認。此外，(4) 的 $y = 2^{3x}$ 與 $y = 8^x$ 相同。

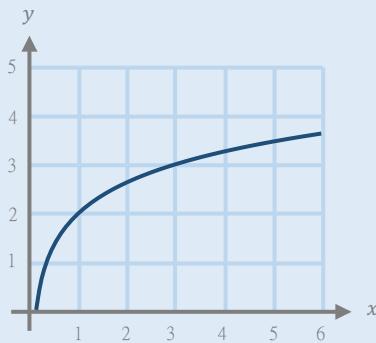
函數 $y = x^2 - 2x + 1$



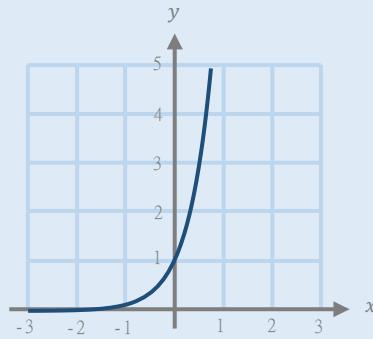
函數 $y = 1.2^x$



函數 $y = \log_3 x + 2$



函數 $y = 2^{3x}$



問題3 (1), (2)

1. ${}_4P_3 = (4 \times 3 \times 2) = \mathbf{24}$

${}_{10}P_5 = (10 \times 9 \times 8 \times 7 \times 6) = \mathbf{30240}$

${}_{2021}P_1 = \mathbf{2021}$

2. ${}_4C_3 = {}_4P_3 \div 3! = 24 \div 6 = \mathbf{4}$

${}_{10}C_5 = {}_{10}P_5 \div 5! = 30240 \div 120 = \mathbf{252}$

${}_{2021}C_1 = {}_{2021}P_1 \div 1! = 2021 \div 1 = \mathbf{2021}$

${}_{2021}C_{2020} = \underline{\underline{{}_{2021}C_1}} = \mathbf{2021}$

因為 ${}_{2021}C_{2020} = \frac{2021!}{2020! \times 1!}$

${}_{2021}C_1 = \frac{2021!}{1! \times 2020!}$ °

問題3 (3), (4)), (5)

3. 根據乘法原理（→3.3.2項），選擇方法的總數為 $160 \times 250 \times 300 = \mathbf{12000000}$ 種

（1200萬種）。

4. 根據乘法原理（→3.3.2項），寫法總數為 $4^5 = \mathbf{1024}$ 種。

5. 將 N 個物品排列的方法數量 $N! = 1 \times 2 \times \cdots \times N$ 種（→3.3.3項），所以長度為 8 的數列總共有 $8! = \mathbf{40320}$ 種。

問題4

首先，平均值（→3.5.4項）如下。

$$\frac{182 + 182 + 188 + 191 + 192 + 195 + 197 + 200 + 205 + 217}{10} = \mathbf{195 \text{ cm}}$$

接著計算標準差。各隊員的身高與平均的差如下表所示。

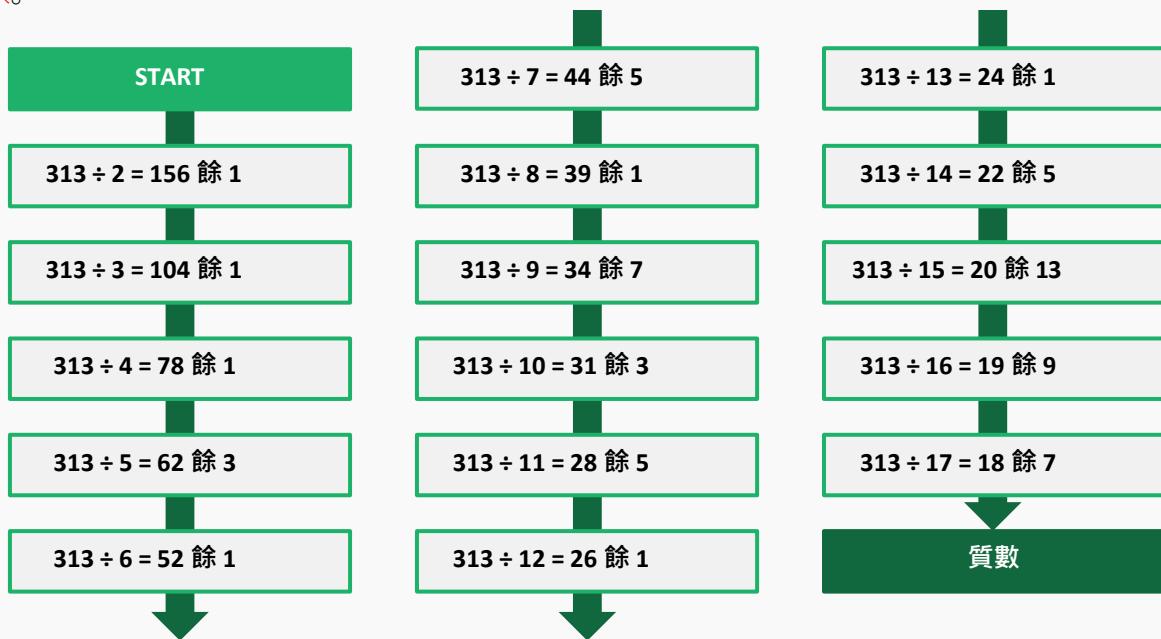
身長	182	183	188	191	192	195	197	200	205	217
與平均的差	13	12	7	4	3	0	2	5	10	22

因此，標準差（→3.5.4項）如下。

$$\sqrt{\frac{13^2 + 12^2 + 7^2 + 4^2 + 3^2 + 0^2 + 2^2 + 5^2 + 10^2 + 22^2}{10}} = \mathbf{10\text{ cm}}$$

問題5 (1)

因為 $\sqrt{313} = 17.69 \dots$ ，可以從 2 到 17 進行除法如下。由於都不能整除，所以 313 是質數。



問題 5 (2)

用輾轉相除法求 723 和 207 的最大公因數，結果如下。求得的最大公因數是 3。



$$723 \div 207 = 3 \text{ 餘 } 102$$

$$207 \div 102 = 2 \text{ 餘 } 3$$

$$102 \div 3 = 34 \text{ 餘 } 0$$

6.2

最終確認問題 6-10 的解答

問題 6

(1) 根據 $1/x$ 的積分 ([→4.3.4項](#))，可知如下。

$$\int_1^{10000} \frac{1}{x} dx = \log_e 10000 = 9.2103 \dots$$

將其四捨五入到小數點後一位，答案是 9。

(2) 各個 i 之程式的循環次數如下：。

- 當 $i = 1$ 時： $\lfloor N/1 \rfloor + 1000$ 次
- 當 $i = 2$ 時： $\lfloor N/2 \rfloor + 1000$ 次
- 當 $i = 3$ 時： $\lfloor N/3 \rfloor + 1000$ 次
- ⋮
- 當 $i = N$ 時： $\lfloor N/N \rfloor + 1000$ 次

因此，整體的循環次數 L 如下：。

$$\underbrace{\left\lfloor \frac{N}{1} \right\rfloor + \left\lfloor \frac{N}{2} \right\rfloor + \left\lfloor \frac{N}{3} \right\rfloor + \cdots + \left\lfloor \frac{N}{N} \right\rfloor}_{+ 1000N}$$

根據倒數和的性質 ([→4.4.4項](#)) 底線部分的總和約為 $N \log_e N$ 。 $L \approx N \log_e N + 1000N$ 。

若使用蘭道大 O 記法來表示，計算複雜度為 $O(N \log N)$ 。

問題 7

答案如下表所示。

函數	N^2	N^3	2^N	3^N	$N!$
1 億	10000	465	27	17	12
5 億	22361	794	29	19	13
10 億	31623	1000	30	19	13

問題 8

(1) 答案如下。可以從前面開始逐個計算。 (→3.7.1項)

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
1	1	5	9	29	65	181	441	1165	2929

(2) 答案如下。不瞭解的人可以回到 4.7 節確認。

$$A + B = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 5 & 8 \\ 10 & 20 \end{bmatrix} = \begin{bmatrix} 1+5 & 4+8 \\ 1+10 & 0+20 \end{bmatrix} = \begin{bmatrix} 6 & 12 \\ 11 & 20 \end{bmatrix}$$

$$AB = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & 8 \\ 10 & 20 \end{bmatrix} = \begin{bmatrix} 1 \times 5 + 4 \times 10 & 1 \times 8 + 4 \times 20 \\ 1 \times 5 + 0 \times 10 & 1 \times 8 + 0 \times 20 \end{bmatrix} = \begin{bmatrix} 45 & 88 \\ 5 & 8 \end{bmatrix}$$

(3) 答案如下。不瞭解的人可以回到 4.7 節確認。

$$A^2 = A \times A = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 1 & 4 \end{bmatrix}$$

$$A^3 = A^2 \times A = \begin{bmatrix} 5 & 4 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 9 & 20 \\ 5 & 4 \end{bmatrix}$$

$$A^4 = A^3 \times A = \begin{bmatrix} 9 & 20 \\ 5 & 4 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 29 & 36 \\ 9 & 20 \end{bmatrix}$$

$$A^5 = A^4 \times A = \begin{bmatrix} 29 & 36 \\ 9 & 20 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 65 & 116 \\ 29 & 36 \end{bmatrix}$$

(4) 這與用矩陣乘方表示費波那契數列的理由相似。首先，根據 $a_n = a_{n-1} + 4a_{n-2}$ 、 $a_{n-1} = a_{n-2}$ ，下式會成立。

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix}$$

重複運用此式，可以用乘方的式子表示如下。

$$\begin{bmatrix} a_n \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} a_{n-2} \\ a_{n-3} \end{bmatrix}$$

= ...

$$= \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} a_2 \\ a_1 \end{bmatrix} = A^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

因此， a_n 的值等於 A^{n-2} 的 $(1, 1)$ 元素和 $(1, 2)$ 元素相加的值。

(下頁繼續)

由於下式成立，「 A^{n-2} 的(1,1)元素和(1,2)元素相加的值」會與「 A^{n-1} 的(1,1)元素」一致。

$$\begin{aligned} A^{n-2} \times A &= \begin{bmatrix} (1,1) \text{ 元素} & (1,2) \text{ 元素} \\ (2,1) \text{ 元素} & (2,2) \text{ 元素} \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} (1,1) \text{ 元素} + (1,2) \text{ 元素} & (1,1) \text{ 元素} \times 4 \\ (2,1) \text{ 元素} + (2,2) \text{ 元素} & (2,1) \text{ 元素} \times 4 \end{bmatrix} \end{aligned}$$

這就是為什麼(1,1)元素的值會出現在數列中的原因。

問題 9

對於 $1 \text{ XOR } 2 \text{ XOR } 3 \text{ XOR } \cdots \text{ XOR } N$ 的值，直接以 $N = 1000000007$ 來求出答案是困難的，因此首先從較小的情況來調查看看吧（→5.2節）。

N	1	2	3	4	5	6	7	8	9	10	11
答案	1	3	0	4	1	7	0	8	1	11	0

在 $N = 3, 7, 11$ 的案例， $1 \text{ XOR } 2 \text{ XOR } \cdots \text{ XOR } N = 0$ ，故此時可以想到「是否為 N 除以4的餘數為3」的規律。

那麼，這個規律在 N 很大的時候是否也成立呢？答案是Yes，可以如下證明。

首先，令 x 為4的倍數時，以下式子成立。

$$\begin{aligned} &x \text{ XOR } (x+1) \text{ XOR } (x+2) \text{ XOR } (x+3) \\ &= \underline{x \text{ XOR } (x+1)} \text{ XOR } \underline{(x+2) \text{ XOR } (x+3)} \\ &= 1 \text{ XOR } 1 = 0 \end{aligned}$$

因此，當 $N \bmod 4 = 3$ 時，所求的值如下。

$$\begin{aligned} &\cancel{1 \text{ XOR } 2 \text{ XOR } 3} \text{ XOR } \cancel{4 \text{ XOR } 5 \text{ XOR } 6 \text{ XOR } 7} \text{ XOR } \cdots \text{ XOR } \cancel{(N-3) \text{ XOR } (N-2) \text{ XOR } (N-1) \text{ XOR } N} \\ &= \quad \downarrow \quad \quad \quad \downarrow \quad \quad \quad \quad \quad \downarrow \\ &= \quad 0 \quad \text{XOR} \quad 0 \quad \text{XOR} \cdots \text{XOR} \quad 0 \\ &= \quad 0 \end{aligned}$$

由於 $1000000007 \bmod 4 = 3$ ，答案是0。

問題 10

本問題直接計算也可以解開，但不使用程式會比較麻煩。這裡使用從上數來第 i 列、從左數來第 j 行的格子寫著 $4i + j$ 這一事實，來分解各個格子的值。

4	4	4	4	4	4	4	4
8	8	8	8	8	8	8	8
12	12	12	12	12	12	12	12
16	16	16	16	16	16	16	16
20	20	20	20	20	20	20	20
24	24	24	24	24	24	24	24
28	28	28	28	28	28	28	28
32	32	32	32	32	32	32	32



1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

所有格子的總和

64 個格子整體來看，將 $[4, 8, 12, 16, 20, 24, 28, 32]$ 相加各 8 次， $[1, 2, 3, 4, 5, 6, 7, 8]$ 相加各 8 次。因此，所求答案如下。

$$\begin{aligned} & (4 + 8 + 12 + 16 + 20 + 24 + 28 + 32) \times (8 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8) \times 8 \\ &= 144 \times 8 + 36 \times 8 \\ &= \mathbf{1440} \end{aligned}$$

綠色格子的總和

對於所有列及所有行，8 個中有 4 個（一半）被塗成綠色。也就是說， $[4, 8, 12, 16, \dots]$ 等被加總的次數也會減半，所求答案為 $1440 \div 2 = \mathbf{720}$ 。

※如果不明白相加次數的技巧，可以參考 5.7 節。

6.5

最終確認問題11-15的解答

問題 11

1. 乘法原理也可以運用於機率，所以答案是 $(1/2)^8 = \mathbf{1/256}$ 。

2. 首先，某個格子變成白色或黑色的機率都是 $1/2$ ，因此：

- 白色圓點的期望值： $64 \times (1/2) = 32$
- 黑色圓點的期望值： $64 \times (1/2) = 32$

根據期望值的線性性質（**→3.4.3項**），[白色個數的期望值] $\times 2 +$ [黑色個數的期望值] $= 32 \times 2 + 32 = \mathbf{96}$ 。

3. 根據期望值的線性性質，所求的答案以下式表示：

$$\begin{aligned} [\text{答案}] &= [\text{第 1 行全部變白的機率}] + \dots + [\text{第 8 行全部變白的機率}] \\ &\quad + [\text{第 1 列全部變白的機率}] + \dots + [\text{第 8 列全部變白的機率}] \\ &\quad + [\text{第 1 條對角線全部變白的機率}] \\ &\quad + [\text{第 2 條對角線全部變白的機率}] \end{aligned}$$

所有的行、列、對角線都由 8 個格子構成，因此它們全部變白的機率是 $(1/2)^8 = 1/256$ 。因此，答案 $1/256 + 1/256 + \dots + 1/256 = 18 \times (1/256) = \mathbf{9/128}$ 。

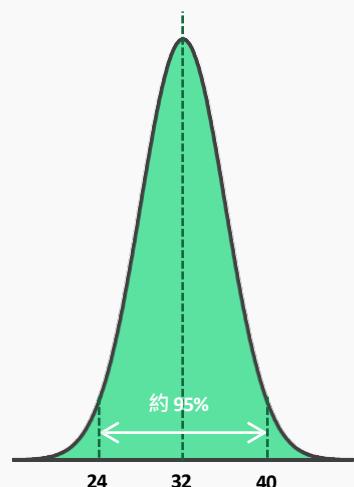
4. 各個格子變白的機率是 $p = 0.5$ ，格子數全部有 $n = 64$ 個，因此白色格子的個數近似於正態分布（**→節末問題 3.5.1**）。

$$\text{平均 } \mu : np = 64 \times 0.5 = 32$$

$$\text{標準偏差 } \sigma : \sqrt{np(1-p)} = \sqrt{64 \times 0.5 \times 0.5} = 4$$

因此， $\mu - 2\sigma = 24$ ， $\mu + 2\sigma = 40$ ，根據 68-95-99.7 法則，

白色格子數在 $24 \sim 40$ 個之間的機率為 **約 95%**。（參見右圖）



問題 12

考慮「不包含 123 的數的數量」很難，因此我們考慮相當於其餘事件（[→5.4.1項](#)）的「包含 123 的數的數量」。。

首先，包含 123 且為 999999 以下的數中，有以下 4 種模式。為了減少區別，將 5 位以下的數的前面用 0 填充（例如 $1237 \rightarrow 001237$ ）。

1 123 ??? 決定後三位數 故1000種（僅123）	2 ?123 ?? 決定三個位數 故1000種（僅123）	3 ??123 ? 決定三個位數 故1000種（僅123）	4 ???123 決定三個位數 故1000種（僅123）
--	--	--	---------------------------------------

針對所有模式，三個位數可在 $0 \sim 9$ 的範圍內自由選擇，根據乘法原理（[→3.3.2項](#)），有 $10 \times 10 \times 10 = 1000$ 種情況。因此，可能會認為「包含 123 的數全部有 4000 個」。

但是，像 123123 這樣的數在情況 1 和 4 中均被計入，因此實際數量為 $4000 - 1 = 3999$ 個。

1 123123	?123 ??	??123 ?	4 123123
-------------	---------	---------	-------------

因此，「包含 123 的數的數量」為全體的模式數 999999 減掉「包含 123 的數的數量（3999 個）」的值，即 **996000 個**。

問題 13

令函數 $\text{func}(N)$ 的計算時間為 a_N ，由於此函數依序呼叫了 $\text{func}(N-1)$ 、 $\text{func}(N-2)$ 、 $\text{func}(N-3)$ 、 $\text{func}(N-3)$ ，則下式成立。

$$a_N = a_{N-1} + a_{N-2} + a_{N-3} + a_{N-3}$$

在此，令 $a_N = 2^N$ ，則 $2^N = 2^{N-1} + 2^{N-2} + 2^{N-3} + 2^{N-3}$ ，正好左右一致。因此，呼叫 $\text{func}(N)$ 的計算複雜度為 $O(2^N)$ 。

*註：對於覺得「為什麼會想到假設 $a_N = 2^N$ 」的讀者，實際測量 $\text{func}(N)$ 的執行時間可能會得到提示。例如，在作者的環境中， $\text{func}(25)$ 需時 0.128 秒， $\text{func}(26)$ 需時 0.259 秒，幾乎為 2 倍差異。

問題 14 (1)

5 人排名的組合有 $5! = 120$ 種，但 4 次提問結果（誰最快）的組合只有 $3^4 = 81$ 種組合。因為後者較小，因此無法在 4 次內確定。（→**5.10.6項**）

問題 14 (2)

此問題有各種解法，以下介紹其中一種。

步驟 1

首先，藉由以下方法，調查 5 人中最快的選手。

1. . 選擇 A、B、C，詢問誰最快。
2. 選擇 D、E（在1. 中最快的選手），詢問誰最快。

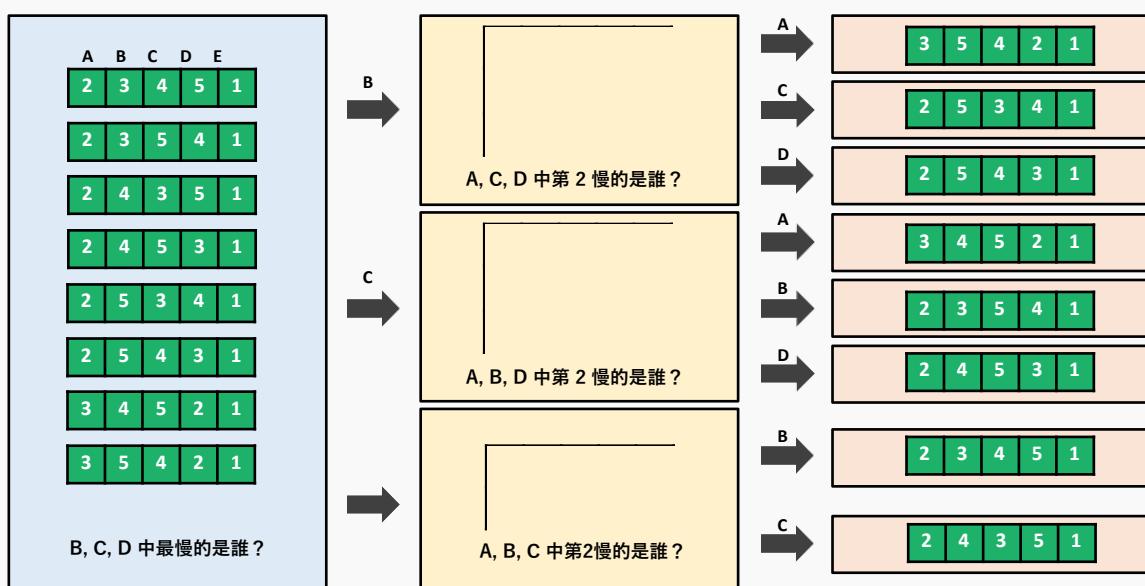
如此，會變成剩下 4 人選手的排名在 3 次提問中確定的問題。之後，假設最快的選手為 E。

步驟 2

詢問 A、B、C 中誰是最快的選手。若答案為 A，則排名的組合會縮小至以下 8 種情況（數字為排名）。

A 2 3 4 5 1	A 2 3 5 4 1	A 2 4 3 5 1	A 2 4 5 3 1
A 2 5 3 4 1	A 2 5 4 3 1	A 3 4 5 2 1	A 3 5 4 2 1

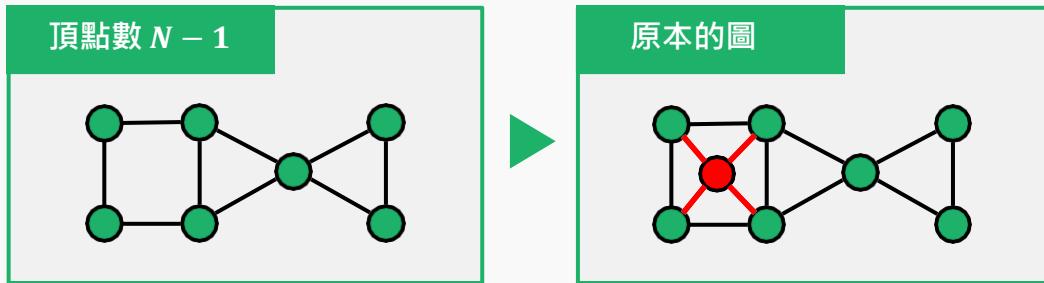
接下來藉由進行以下提問，必定可在 2 次內確定排名。根據這樣的流程，可在總計 5 次提問中確定全部 5 人的排名。



問題 15

首先，平面圖具有「頂點數小於邊數的 3 倍」這一性質，因此至少存在一個度為 5 以下的頂點。

因此，藉由在頂點數為 $N - 1$ 的平面圖中添加度為 5 以下的頂點，可以構成原本的圖。



以下在此，證明：假設在頂點數 $N - 1$ 的平面圖可以用 5 色著色時，添加頂點後的圖也能用 5 色著色（★）。如下。

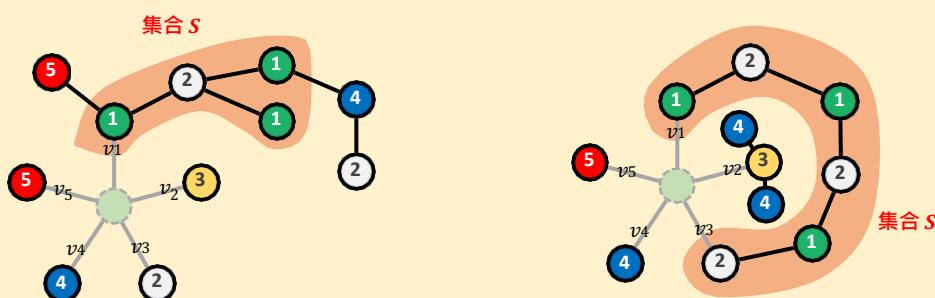
模式 1：添加的頂點 u 的度為 4 以下

如下圖所示，塗上與任一相鄰頂點都不同的顏色即可。（有 5 種顏色可選擇，因此一定存在像這樣的顏色）



模式 2：添加的頂點 u 的度為 5

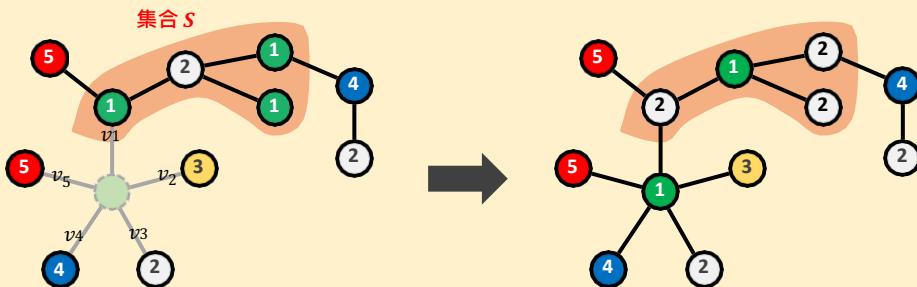
將與 u 相鄰的頂點依順時針設為 v_1, v_2, v_3, v_4, v_5 ，令 v_1 的顏色為 1， v_3 的顏色為 2。另外，（在頂點數 $N - 1$ 的圖中）將從頂點 v_1 開始，只通過顏色 1、2 的頂點而到達的頂點的集合設為 S 。



此時，當 v_3 不包含在集合 S 中時，進行以下操作，可使顏色 1 空餘（代表添加頂點 u 可設為顏色 1）。

- 將 S 中所含顏色 1 的頂點改為顏色 2。
- 將 S 中所含顏色 2 的頂點改為顏色 1。

下圖顯示此操作的具體例。



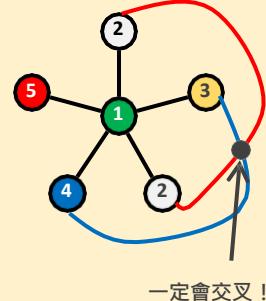
另一方面，若 v_3 包含在集合 S 中時，進行與頂點 v_2 （顏色 3）和頂點 v_4 （顏色 4）相同的操作即可。具體而言，將從頂點 v_2 出發，只經過顏色 3 和 4 的頂點到達的頂點集合設為 T ，進行以下操作：

- 將 T 中所含顏色 3 的頂點改為顏色 4。
- 將 T 中所含顏色 4 的頂點改為顏色 3。

則顏色 3 會空餘。另外，以下路徑一定會交叉：

- 從頂點 v_1 到 v_3 ，且不經過頂點 u 的路徑。
- 從頂點 v_2 到 v_4 ，且不經過頂點 u 的路徑。

因此 v_4 絕對不包含在集合 T 中。



最後，因為頂點數 1 的圖可用 5 色著色，根據 (★) 可知，頂點數 2 圖可以，頂點數 3 的圖可以，頂點數 4 的圖可以……，最終證明原本的圖也能用 5 色著色。。

若想知道更容易理解的證明，請參見 chap6-11_15.md 中刊載的「高校數學之美」網站。

6.4

最終確認問題 16-20 的解答

問題 16

將可能的所有模式逐一調查的方法稱為「全搜尋」（→2.4.5 項）。

在本問題中，對於滿足 $1 \leq a < b < c \leq N$, 且 $a + b + c = X$ 的整數 (a, b, c) 的組合，要完全一點不漏地列舉出來絕非易事。然而，將滿足 $1 \leq a < b < c \leq N$ 的 (a, b, c) 進行列舉是簡單的，將其全體進行測試並計算滿足 $a + b + c = X$ 的個數，這是最簡單的方法。。

將此解法以 C++ 實作如下。 (a, b, c) 的迴圈處理與程式碼3.3.1相似

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

int main() {
    // 輸入
    int N, X;
    cin >> N >> X;

    // 嘗試所有 (a, b, c) 的組合
    int answer = 0;
    for (int a = 1; a <= N; a++) {
        for (int b = a + 1; b <= N; b++) {
            for (int c = b + 1; c <= N; c++) {
                if (a + b + c == X) {
                    answer += 1;
                }
            }
        }
    }

    // 輸出答案
    cout << answer << endl;
    return 0;
}
```

※ Python等原始碼請參閱 chap6-16_20.md。

問題 17

矩形的面積是以（縱長） \times （橫長）計算的。由於這兩者都是整數，所以為了使面積為N，其縱長和橫長必須是「N的因數」。

36 的因數 … 1, 2, 3, 4, 6, 9, 12, 18, 36



因此，若用**3.1.5 項**的方法列舉約數，可以調查所有可能的矩形。從其中找出周長最小的。

另外，也有更簡單的解法。利用假設（縱長） \leq （橫長）也不會失去一般性（ \rightarrow **5.10.4 項**）的性質，則（縱長） $\leq \sqrt{N}$ 。因此，對縱長x在1以上到 \sqrt{N} 以下的範圍進行全搜尋，找出以下條件：

- 橫長 $N \div x$ 是整數的情況
- 在其中找出周長 $2x + 2(N \div x)$ 為最小的情況

可以用計算複雜度 $O(\sqrt{N})$ 以，C++ 實作如下。

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

int main() {
    // 輸入
    long long N;
    cin >> N;

    // 將垂直長度從 1 到  $\sqrt{N}$  為止進行全搜尋
    long long answer = (1LL << 60);
    for (long long x = 1; x * x <= N; x++) {
        if (N % x == 0) {
            answer = min(answer, 2 * x + 2 * (N / x));
        }
    }

    // 輸出答案
    cout << answer << endl;
    return 0;
}
```

※ Python等原始碼請參閱chap6_16_20.md。

問題 18

整數 A 、 B 的最大公因數為 $\text{GCD}(A, B)$ ，最小公倍數為 $\text{LCM}(A, B)$ 。因此， $A \times B = \text{GCD}(A, B) \times \text{LCM}(A, B)$ 的關係（→2.5.2 項）會成立。因此，若可以用輾轉相除法（→3.2 節）計 $\text{GCD}(A, B)$ ，則最小公倍數可以用下式求出。

$$\text{LCM}(A, B) = A \times B \div \text{GCD}(A, B)$$

計算複雜度是 $O(\log(A + B))$ 。

然而，直接計算可能會在 C++ 等語言中發生溢出。因為即使使用 long long 型態，也只能處理大約 19 位的數字，而在計算 $A \times B$ 時可能超過這個範圍。這個問題可以透過以下兩種方法來應對：

- 1 將計算順序從「 $A \times B \div \text{GCD}(A, B)$ 」變為「 $A \div \text{GCD}(A, B) \times B$ 」
- 2 用巧妙的方法判斷最小公倍數是否超過 10^{18}

關於2.，判斷 $A \times B \div \text{GCD}(A, B) > 10^{18}$ ，即 $A \div \text{GCD}(A, B) > 10^{18} \div B$ 即可。因為左邊必定是整數，所以右邊取整數部分也能順利進行。因此，撰寫如下程式即可得到正解。

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

long long GCD(long long A, long long B) {
    if (B == 0) return A;
    return GCD(B, A % B);
}

int main() {
    // 輸入
    long long A, B;
    cin >> A >> B;

    // 判斷最小公倍數是否超過 10^18
    if (A / GCD(A, B) > 10000000000000000 / B) {
        cout << "Large" << endl;
    }
    else {
        cout << A / GCD(A, B) * B << endl;
    }
    return 0;
}
```

※ Python等原始碼請參閱 chap6-16_20.md。

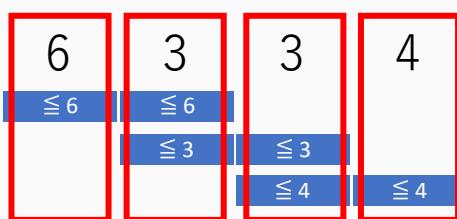
問題 19

這個問題可以使用考慮上限 (\rightarrow 5.8 節) 的技巧來解決。

考慮一般的情況並不容易，因此首先以具體例 $N = 4, B_1 = 6, B_2 = 3, B_3 = 4$ 來思考。此時，可以換句話說如下。

- $\max(A_1, A_2) \leq 6 \cdots$ 「 A_1 和 A_2 都為 6 以下」
- $\max(A_2, A_3) \leq 3 \cdots$ 「 A_2 和 A_3 都為 3 以下」
- $\max(A_3, A_4) \leq 4 \cdots$ 「 A_3 和 A_4 都為 4 以下」

因此，可知 A_1 為 6 以下， A_2 為「6 以下且 3 以下」，所以是 3 以下， A_3 為「3 以下且 4 以下」，所以是 3 以下， A_4 為 4 以下。



一般情況也一樣進行，得到 $A_1 \leq B_1, A_i \leq \min(B_{i-1}, B_i)$ ($2 \leq i \leq N - 1$), $A_N \leq B_{N-1}$ 的上限。實際上，符合這個上限的數列 A 滿足條件，撰寫求其總和的程式即為正解。以下是 C++ 的實作例。

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

int N, B[109];
int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N - 1; i++) {
        cin >> B[i];
    }

    // 求出數列 A 的元素總和 → 輸出答案
    int answer = B[1] + B[N - 1];
    for (int i = 2; i <= N - 1; i++) {
        answer += min(B[i - 1], B[i]);
    }
    cout << answer << endl;
    return 0;
}
```

※ Python 等原始碼請參閱 chap6-16_20.md。

問題 20

對於各問題，若如以下程式求出總和，則計算複雜度是 $O(NQ)$ ，會超過執行時間限制 (TLE)。

```
int answer1 = 0, answer2 = 0;
for (int i = L; i <= R; i++) {
    if (C[i] == 1) answer1 += P[i];
    if (C[i] == 2) answer2 += P[i];
}
```

為了高速化，可以使用累積和（→ 4.2 節）。首先，只計算一班的總得分。假設學號 i 的學生屬於一班時為 $A_i = P_i$ ，屬於二班時為 $A_i = 0$ ，則一班的總得分是 $A_L + A_{L+1} + \dots + A_R$ 。因此，使用累積和的話可以用 $O(1)$ 來回答各問題。

學號、班級	1	2	3	4	5
得分 P_i	50	80	100	30	40
一班的得分 A_i	50	80	0	30	0
累積和	50	130	130	160	160

... 一班
... 二班

對二班也可以進行同樣操作。如此，以整體計算複雜度 $O(N + Q)$ 來解決此問題。將此解法以 C++ 實作如下。

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

int N, C[100009], P[100009], L[100009], R[100009], S1[100009], S2[100009];
int main() {
    // 輸入 → 求出累積和
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> C[i] >> P[i];
    for (int i = 1; i <= N; i++) S1[i] = S1[i - 1] + (C[i] == 1 ? P[i] : 0);
    for (int i = 1; i <= N; i++) S2[i] = S2[i - 1] + (C[i] == 2 ? P[i] : 0);

    // 回答問題
    cin >> Q;
    for (int i = 1; i <= Q; i++) {
        cin >> L[i] >> R[i];
        cout << S1[R[i]] - S1[L[i] - 1] << " " << S2[R[i]] - S2[L[i] - 1] << endl;
    }
    return 0;
}
```

※ Python 等原始碼請參閱 chap6-16_20.md。

問題 21 (1)

對於函數 $f(x) = e^x$ ，具有其微分 $f'(x)$ 也是 e^x 的性質。因此，於 $y = e^x$ 的點 $(1, e)$ 的切線斜率是 $f'(1) = e^1 = e$ 。因此，切線方程式可以用 $y = ex + b$ 的形式表示，但其中通過點 $(1, e)$ 的是 $b = 0$ 時。

因此，於點 $(1, e)$ 的切線方程式是 $\textcolor{red}{y = ex}$ 。

問題 21 (2)

當切線 $y = ex$ 與直線 $y = 2$ 相交時， $ex = 2$ ， $x = \frac{2}{e}$ 。因此，這兩條直線的交點座標是 $\left(\frac{2}{e}, 2\right)$ 。

將此點的 x 座標以小數表示為 $0.735758882 \dots$ 。

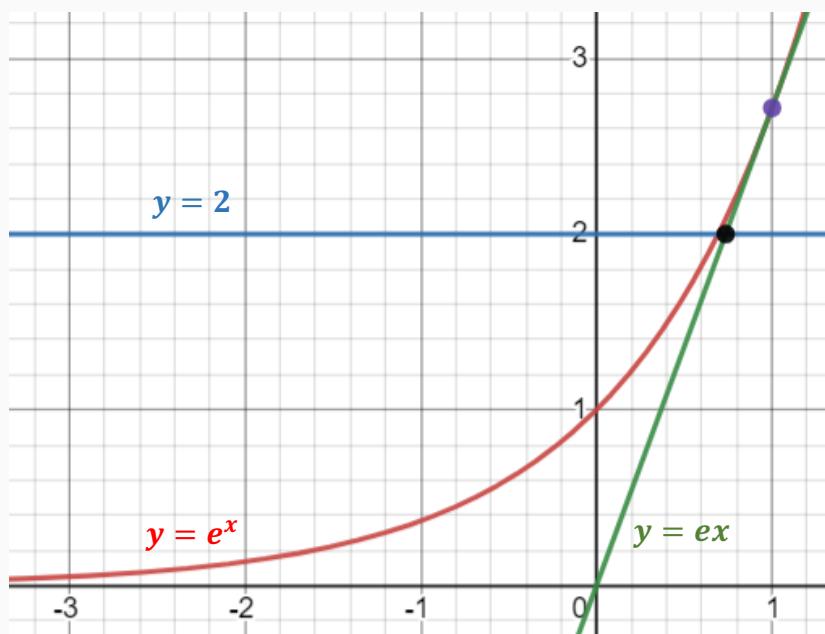


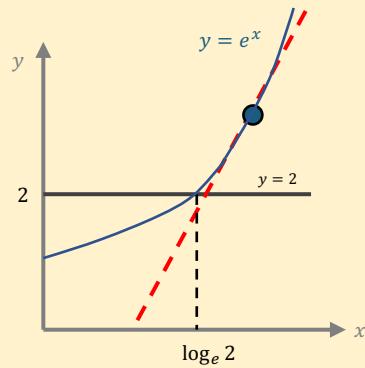
圖 $y = e^x$ 與其切線等的圖（用desmos.com描繪）

問題 21 (3)

$\log_e 2$ 的值可用牛頓法 (\rightarrow 4.3 節) 求得。由於曲線 $y = e^x$ 之 y 座標為 2 的點的 x 座標是 $\log_e 2$ ，可以藉以下方針求出。

- 設 $f(x) = e^x$ ，此時 $f'(x) = e^x$ 。
- 首先，任意設定初始值 a 。
- 之後，將 a 的值持續更新成以下。

點 $(a, f(a))$ 之切線與直線 $y = 2$
的交點 x 座標



因此，撰寫如下程式即可。另外， $\exp(x)$ 是回傳 e^x 的函數。作為別的方法，亦可用 $\text{pow}(2.718281828, x)$ 來取代，會得到幾乎相同的結果。

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

int main() {
    double r = 2.0; // 因為所求為  $y = e^x$  和  $y = 2$  的交點
    double a = 1.0; // 將初始值任意設定為 1.0

    for (int i = 1; i <= 5; i++) {
        // 求出點  $(a, f(a))$  的  $x$  座標與  $y$  座標
        double zahyou_x = a;
        double zahyou_y = exp(a);      ← 從程式碼 4.3.1 改變的部分

        // 求出切線式  $y = \text{sessen\_a} * x + \text{sessen\_b}$ 
        double sessen_a = zahyou_y;    ← 從程式碼 4.3.1 改變的部分
        double sessen_b = zahyou_y - sessen_a * zahyou_x;

        // 求求出下個  $a$  的值 next_a
        double next_a = (r - sessen_b) / sessen_a;
        printf("Step #%d: a = %.15lf -> %.15lf\n", i, a, next_a);
        a = next_a;
    }
    return 0;
}
```

此時，輸出會如下所示。迅速收斂至 $\log_e 2 = 0.693147180559945 \dots$ ，僅 5 次即可與小數點後 15 位數一致。

```
Step #1: a = 1.000000000000000 -> 0.735758882342885  
Step #2: a = 0.735758882342885 -> 0.694042299918915  
Step #3: a = 0.694042299918915 -> 0.693147581059771  
Step #4: a = 0.693147581059771 -> 0.693147180560026  
Step #5: a = 0.693147180560026 -> 0.693147180559945
```

Python、JAVA、C的原始碼請參閱 GitHub 的 chap6-21_25.md。

問題 22

使用兩台烤箱，稱作「烤箱A」和「烤箱B」。在烤箱A上耗費的時間為 a ，在烤箱B上耗費的時間為 b ，則整個料理所需的時間為 $\max(a, b)$ 。在此，不論如何分配烤箱， $a + b$ 的值仍為 $sumT = T_1 + T_2 + \dots + T_N$ 不變，所以，一旦 a 確定， $b = sumT - a$ 也會自動確定，料理所需的整體時間會變成 $\max(a, sumT - a)$ 。



那麼，什麼樣的 a 是「可實現」的呢？在上圖的例子中，將可實現的 a 全部列舉的話，有 0、6、7、8、11、12、13、14、15、17、18、19、20、21、23、24、25、26、27、29、30、31、32、33、36、37、38、44 分鐘。

事實上，使用與**節末問題 3.7.4** 非常相似的動態規劃法的演算法，可以將可實現的 a 以計算複雜度 $O(N \times sumT)$ 全部列出。

準備的陣列（二維陣列）

$dp[i][j]$: 從到料理 i 為止之中，若存在使烤箱A耗費的時間和（以下簡稱耗費時間）為 j 的組合，則為 `true`，否則為 `false`。

動態規劃法的轉換 ($i = 0$)

明顯地，只有「什麼都不選」這一種方法存在。

- $dp[0][j] = \text{true} (j = 0)$
- $dp[0][j] = \text{false} (j \neq 0)$

動態規劃法的轉換（以 $i = 1, 2, \dots, N$ 的順序計算）

從到料理 i 為止之中進行選擇，以使總和為 j 的方法有以下兩種（依最後的行動 [烹飪料理 i 的烤箱] 區分）：

- 到料理 $i - 1$ 為止的耗費時間為 $j - A_i$ ，且料理 i 用烤箱 A 烹飪
- 到料理 $i - 1$ 為止的耗費時間為 j ，且料理 i 不用烤箱 A 烹飪

因此，若 $dp[i - 1][j - A_i]$ ，或 $dp[i - 1][j]$ 之中至少一方為 `true`，則 $dp[i][j] = \text{true}$ ，否則為 `false`。

最終，當 $dp[N][x] = \text{true}$ 時， $a = x$ 是可實現的。在可實現的 a 中， $\max(a, sumT - a)$ 為最小的即為答案。

此解法以C++實作如下。

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

int N, T[109]; bool dp[109][100009];

int main() {
    // 輸入
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> T[i];

    // 陣列的初始化
    int sumT = 0;
    for (int i = 1; i <= N; i++) sumT += T[i];
    for (int i = 1; i <= sumT; i++) dp[0][i] = false;
    dp[0][0] = true;
```

```

// 動態規劃法
for (int i = 1; i <= N; i++) {
    for (int j = 0; j <= sumT; j++) {
        if (j < T[i]) {
            if (dp[i - 1][j] == true) dp[i][j] = true;
            else dp[i][j] = false;
        }
        if (j >= T[i]) {
            if (dp[i-1][j] == true || dp[i-1][j-T[i]] == true) dp[i][j] = true;
            else dp[i][j] = false;
        }
    }
}

// 計算答案並輸出
int answer = (1 << 30);
for (int i = 0; i <= sumT; i++) {
    if (dp[N][i] == true) {
        int cooking_time = max(i, sumT - i);
        answer = min(answer, cooking_time);
    }
}
cout << answer << endl;
return 0;
}

```

Python、JAVA、C的原始碼請參閱 GitHub 的 chap6-21_25.md。

問題 23

使用埃拉托斯特尼篩法（→**4.4.1 項**），可以在 $O(N \log \log N)$ 時間內列出 N 以下的質數。然而，這個問題要求列出的是「 L 以上 R 以下的質數」。可以如下將演算法稍微改變試試看。）

1. 首先，寫出整數 $L, L + 1, \dots, R$ 。
2. 把寫出的所有的 2 的倍數標記 \times 。作為例外，對 2 不標記 \times 。
3. 把寫出的所有的 3 的倍數標記 \times 。作為例外，對 3 不標記 \times 。
4. 把寫出的所有的 4 的倍數標記 \times 。作為例外，對 4 不標記 \times 。
5. （中略）
6. 把寫出的所有的 $\lfloor \sqrt{R} \rfloor$ 的倍數標記 \times 。作為例外，對 $\lfloor \sqrt{R} \rfloor$ 不標記 \times 。
7. 剩下沒有被標記的整數就是質數。

接下來，考慮實作方法。由於程式中無法直接寫出整數，因此取而代之可以使用長度為 $R - L + 1$ 的陣列 `prime`，於 `prime[x]` 中記錄「整數 $x + L$ 是否為無標記」（若無標記則 `true`，若被標記 `×` 的話為 `false`），如此一來可順利實作。

注意在第 i 步操作中，被標記 `×` 的數為 $[L/i] \times i, [L/i + 1] \times i, \dots, [R/i] \times i$ 。由於被標記 `×` 的數大約為 $(R - L)/i$ 個，因此總體計算複雜度如下。

$$\frac{R - L}{2} + \frac{R - L}{3} + \dots + \frac{R - L}{\sqrt{R}} = O((R - L) \log \sqrt{R})$$

此證明可以參考 4.4.4 節 和 4.4.5 節。

將此解法以 C++ 實作如下。

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

long long L, R; bool prime[500009];

int main() {
    // 輸入
    cin >> L >> R;

    // 陣列的初始化、區分出 L = 1 時的狀況（邊角案例）
    for (long long i = 0; i <= R - L; i++) {
        prime[i] = true;
    }
    if (L == 1) prime[0] = false;

    // 篩選
    for (long long i = 2; i * i <= R; i++) {
        long long min_value = ((L + i - 1) / i) * i; // 在 L 以上之最小的 i 的倍數
        // 將 L 以上 R 以下的所有（除 i 以外的）i 的倍數標記為 ×
        for (long long j = min_value; j <= R; j += i) {
            if (j == i) continue;
            prime[j - L] = false;
        }
    }

    // 計數個數並輸出
    long long answer = 0;
    for (long long i = 0; i <= R - L; i++) {
        if (prime[i] == true) answer += 1;
    }
    cout << answer << endl;
    return 0;
}
```

若做更多努力，能夠以計算複雜度 $O\left(\left(\sqrt{R} + (R - L)\right) \log \log \sqrt{R}\right)$ 來解。若事先用埃拉托斯特尼篩法找出 \sqrt{R} 以下的質數，可以節省「合數（4, 6, 8, 9, …）的倍數被標記」的多餘操作，進而達到此計算複雜度。

Python、JAVA、C 的原始碼請參閱 GitHub 的 chap6-21_25.md。

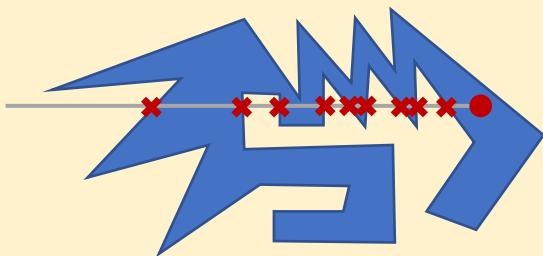
問題 24

此為計算幾何學（→ 4.1 節）的問題。這個問題中，多角形不一定為凸（內角都小於 180 度的多角形），所以有時會有複雜的結構。包括這樣的情況在內，要判斷點是否在多角形的內部，該如何做呢？

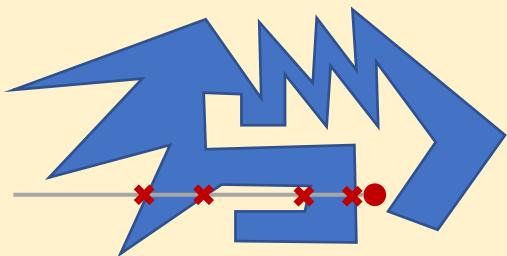
其實，可以使用以下簡單的方法來判斷。

點 (A, B) 是否被包含在多角形內部的判斷

1. 從點 (A, B) 向左拉出半直線。
2. 計算此半直線與多角形的邊相交的次數。若是奇數次，則點 (A, B) 在多角形的內部，若是偶數次，則點 (A, B) 在多角形的外部。



9 次相交所以是 **內部**

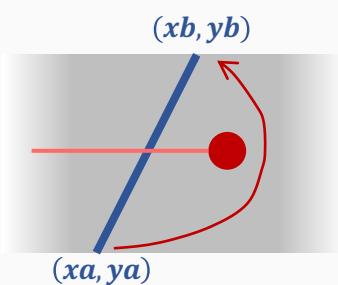


4 次相交所以是 **外部**

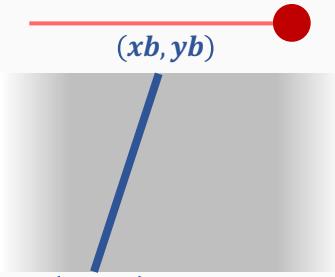
也就是說，需要判斷從點 (A, B) 向左拉出的半直線與多角形的各邊是否相交。假設邊是將 (xa, ya) 和 (xb, yb) 連接的線段（其中 $ya < yb$ ），基本上當滿足以下條件時會相交。

1 $ya \leq B \leq yb$ 。

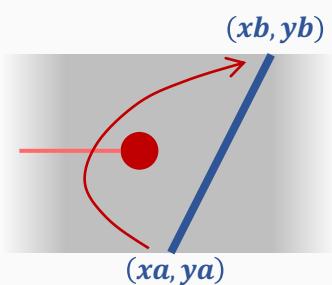
2 點 (xa, ya) 、點 (A, B) 、點 (xb, yb) 按順序為逆時針方向。



滿足條件 1、2 兩方

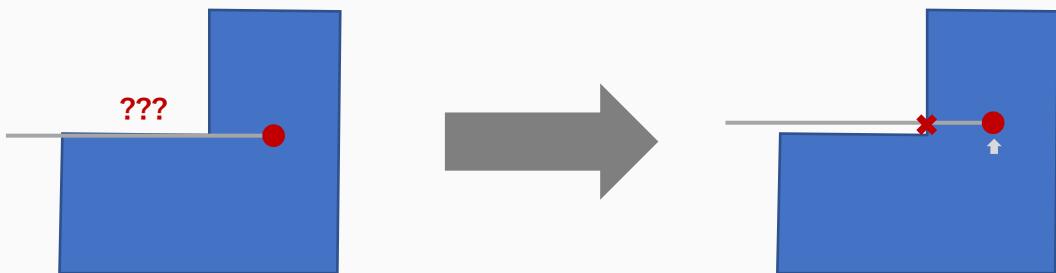


不滿足條件 1
($\nexists ya \leq B \leq yb$)



不滿足條件 2
(順時針方向)

不過，如果多角形的邊在 $y = B$ 即水平的情況下是例外的，只看這條邊無法判斷是否相交。為了避免這種情況，可以將點稍微向上移動一點（這樣做不會改變結果）。



這樣的話，條件 1 會稍微改變，成為以下形式。這是因為條件變成「 $ya \leq B + \epsilon \leq yb$ 」(ϵ 是無限小的數字)」。

- 1 $ya \leq B < yb$ 。
- 2 點 (xa, ya) 、點 (A, B) 、點 (xb, yb) 按順序為逆時針方向。

條件 2 可以使用外積來判斷（→4.1.5項）。計數多角形的邊中滿足此條件的邊，藉由判斷此邊數是奇數還是偶數來解決這個問題。

將此解法以C++實作如下。

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

int N; long long X[100009], Y[100009], A, B;

int main() {
```

```

// 輸入
cin >> N;
for (int i = 0; i < N; i++) cin >> X[i] >> Y[i];
cin >> A >> B;

// 計數交叉的次數
int cnt = 0;
for (int i = 0; i < N; i++) {
    long long xa = X[i] - A, ya = Y[i] - B;
    long long xb = X[(i + 1) % N] - A, yb = Y[(i + 1) % N] - B;
    if (ya > yb) {
        swap(xa, xb);
        swap(ya, yb);
    }
    if (ya <= 0 && 0 < yb && xa * yb - xb * ya < 0) {
        cnt += 1;
    }
}
}

// 輸出答案
if (cnt % 2 == 1) cout << "INSIDE" << endl;
else cout << "OUTSIDE" << endl;

return 0;
}

```

Python、JAVA、C 的原始碼請參閱 GitHub 的 chap6-21_25.md。。

問題 25

這個問題中，如果使用廣度優先搜尋（→ 4.5.7 項）來尋找最短路徑，計算複雜度需要花費 $O(N^2)$ ，但若巧妙地使用「考慮相加的次數」這個技巧（→ 5.7 節），可以將計算複雜度降到 $O(N)$ 來解決。

首先，以具體例來說，考慮以下 5 頂點的樹。頂點配對有 10 種，每個配對的最短路徑如下圖所示。

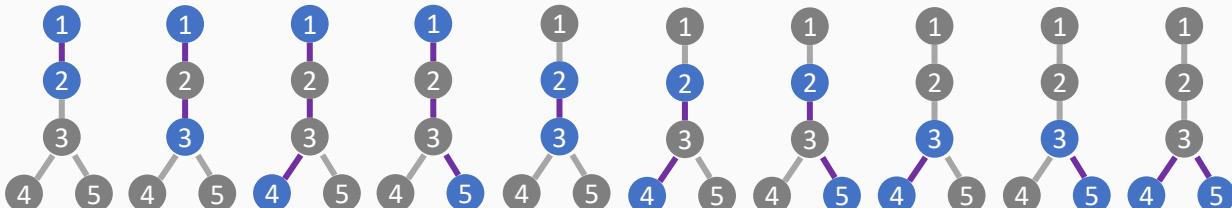


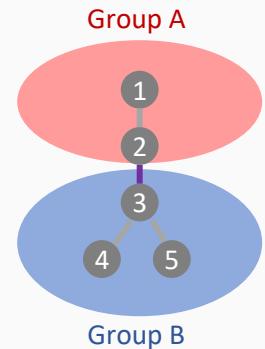
圖 10 種配對的各個最短路徑

從此可知如下幾點。

- 連接頂點 1-2 的邊包含在4條最短路徑（第1、2、3、4條）中。
- 連接頂點 2-3 的邊包含在6條最短路徑（第2、3、4、5、6、7條）中。
- 連接頂點 3-4 的邊包含在4條最短路徑（第3、6、8、10條）中。
- 連接頂點 3-5 的邊包含在4條最短路徑（第4、7、9、10條）中。

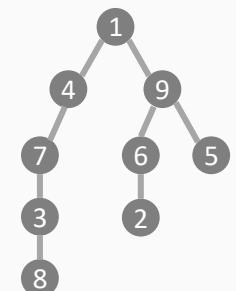
因此，答案是 $4 + 6 + 4 + 4 = 18$ ，這樣的求解方針就是利用「考慮相加的次數」技巧的做法。

那麼，如何知道特定的邊包含在幾條最短路徑中呢？如右圖所示，邊會將樹分成兩個部分。如果將其分成 A 個頂點的群組和 $N - A$ 個頂點的群組，邊會包含在 $A \times (N - A)$ 條最短路徑中。因為從 Group A 的頂點到 Group B 的頂點的最短路徑中一定會包含這條邊。



因此，對每條邊，若能求出分開的兩個群組各自有幾個頂點，就能解決這個問題。

接下來，思考提著頂點1並將此樹懸掛起來。如此，就得到了右圖所示由根開始的圖的形狀。這樣的樹稱為「以頂點1為根的有根樹」（例如，上司和下屬的關係圖（→ 4.5.3 項）可以表示為有根樹）。

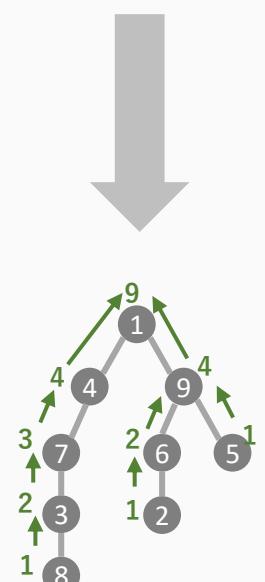


假設頂點 v 的子樹（包含其本身）有 c_v 個頂點。則連接頂點 v 和其正上方頂點的邊會將圖分成 c_v 個頂點和 $N - c_v$ 個頂點的群組。

c_v 可以使用動態規劃法來求出。設頂點 v 的正下方頂點為 s_1, s_2, \dots, s_k ，則可計算出下式。

$$c_v = c_{s_1} + c_{s_2} + \dots + c_{s_k} + 1$$

因此，若從有根樹的底部開始依序求 c_v ，可以在計算複雜度 $O(N)$ 下求得所有 c_v 。本問題的答案是 $c_v \times (N - c_v)$ ($v = 2, 3, \dots, N$) 的總和。



將此解法以C++實作如下。另外，動態規劃法的部分若使用與深度優先搜尋（→ 4.5.6項）相同的遞迴函數，可以較簡單地實作。

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

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

int dp[100009]; bool visited[100009];
void dfs(int pos) {
    visited[pos] = true;
    dp[pos] = 1;
    for (int i : G[pos]) {
        if (visited[i] == false) {
            dfs(i);
            dp[pos] += dp[i];
        }
    }
}

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

    // 使用深度優先搜尋 (DFS) 的動態規劃法
    for (int i = 1; i <= N; i++) {
        visited[i] = false;
    }
    dfs(1);

    // 計算答案並輸出
    long long answer = 0;
    for (int i = 2; i <= N; i++) {
        answer += 1LL * dp[i] * (N - dp[i]);
    }
    cout << answer << endl;

    return 0;
}
```

Python、JAVA、C 的原始碼請參閱 GitHub的chap6-21_25.md。。

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 }
    };
};
```

```

// 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 (\rightarrow 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 3 顆球的方法有幾種?
- (中略)
- 選取使差為 k 以上的 $[N/k]$ 顆球的方法有幾種?

可以在 $[N/k]$ 個中止的理由是因為即使選取再好，最多也只能選取 $[N/k]$ 顆球。這樣，問題 k 可以分解成 $[N/k]$ 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 $O(N \log N)$ (\rightarrow 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。

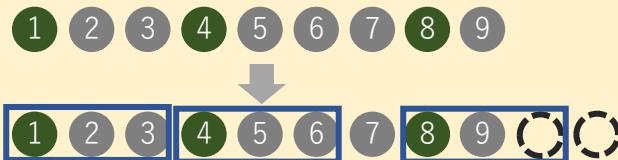
「小問題」是怎樣的問題？

從寫有 $1, 2, \dots, N$ 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 $x, x+1, \dots, x+m-1$ 框起來思考的話，會變成從球 $1, 2, \dots, N+m-1$ 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。

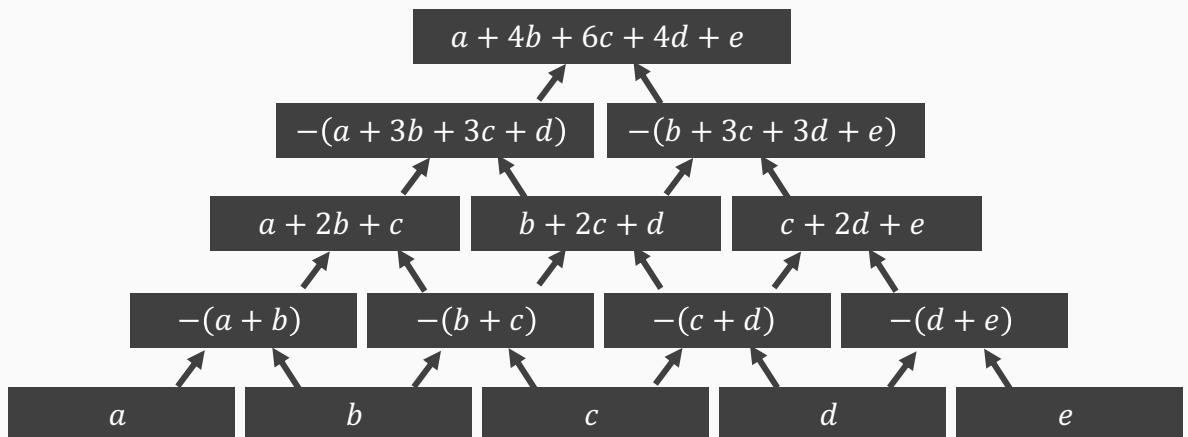
- 「藍」 $\rightarrow 0$ 、「白」 $\rightarrow 1$ 、「紅」 $\rightarrow 2$

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。

$x \backslash y$	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 $\bmod 3$ 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 (\rightarrow 4.6.8 項)。

假設將 n 以三進制表示為 $n_{d-1}n_{d-2}\dots n_1n_0$ ， k 以三進制表示為 $k_{d-1}k_{d-2}\dots k_1k_0$ 。此時，

使用盧卡斯定理吧。

將 n 以三進制表示為 $n_{d-1}n_{d-2}\dots n_1n_0$ ， r 以三進制表示為 $r_{d-1}r_{d-2}\dots r_1r_0$ 。此時， $nCr \bmod 3$ 如下計算：

$$(n_{d-1}Cr_{d-1} \times n_{d-2}Cr_{d-2} \times \dots \times n_1Cr_1 \times n_0Cr_0) \bmod 3$$

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 <iostream>
#include <string>
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;
}

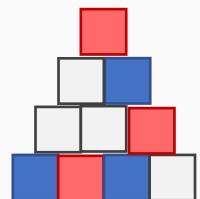
```

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

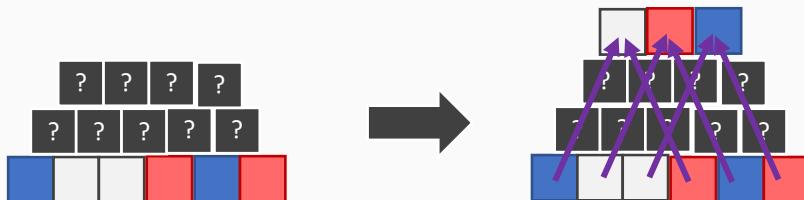
問題 28 — 別解

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 $N = 28, 82, 244, 730, 2188, 6562, \dots$ 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。。

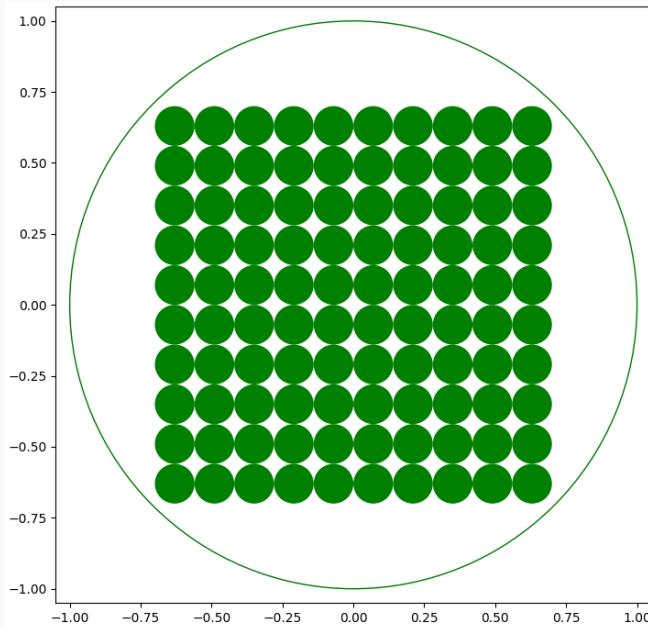
實作程式碼因篇幅問題省略。

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。。

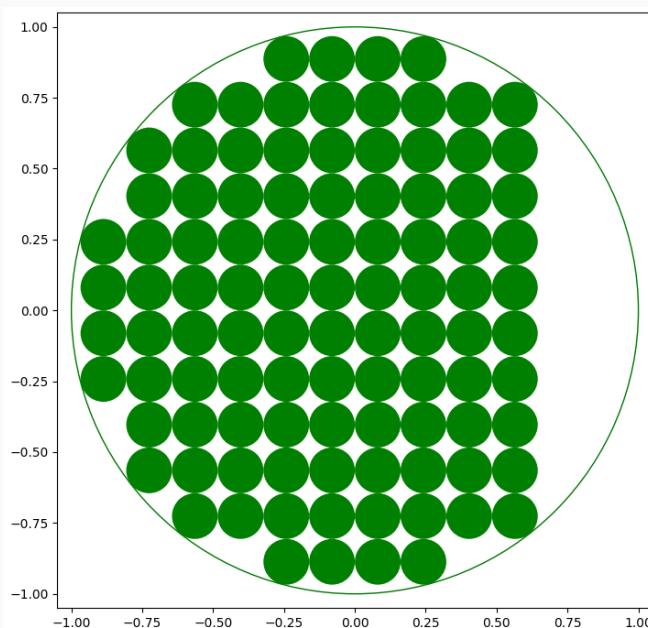
解法 0 — AtCoder上的輸出例($R = 0.07$)

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



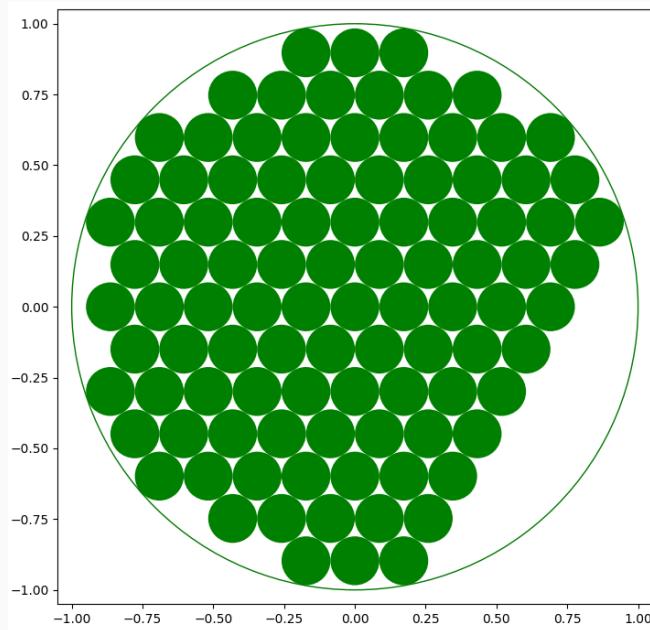
解法 1 — 填滿成正方形 ($R=0.0806$)

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 $R = 0.0806$ 。



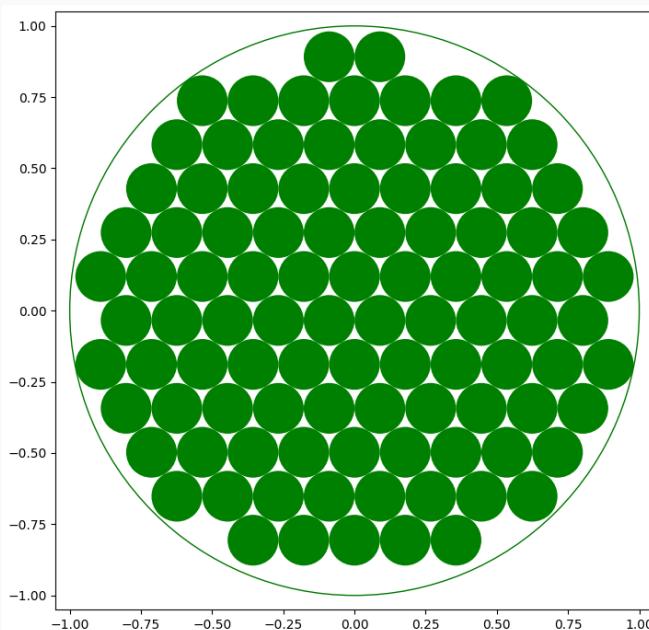
解法 2 — 填滿成六角形($R = 0.0863$)

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。



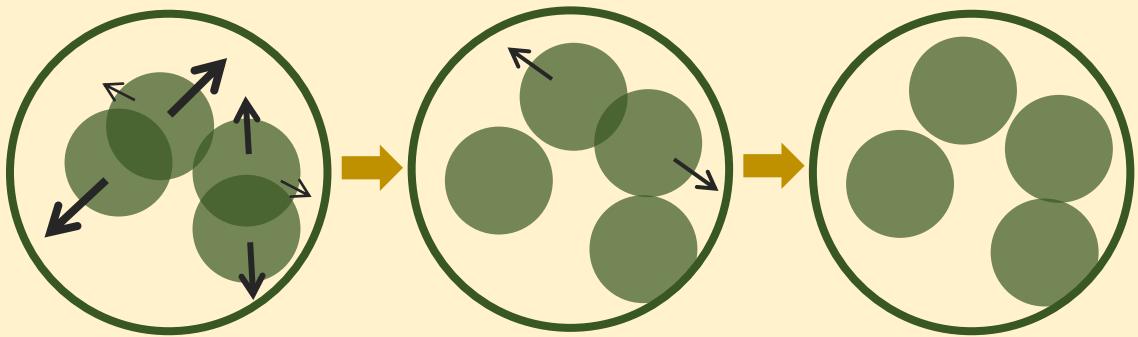
解法4—使用梯度下降法 ($R = 0.0899$)

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。



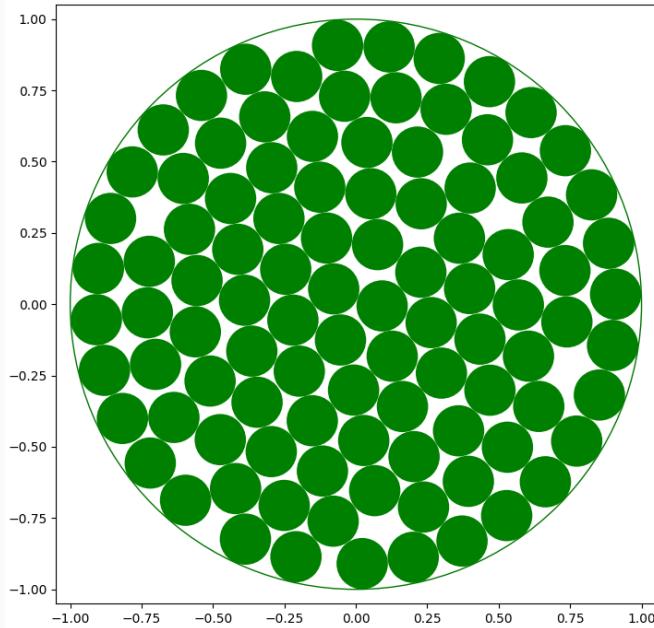
這可以使用梯度下降法來實作。例如，設定**懲罰**函數等如下：

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

朝使 P 盡可能減小的方向，將 $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ 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。

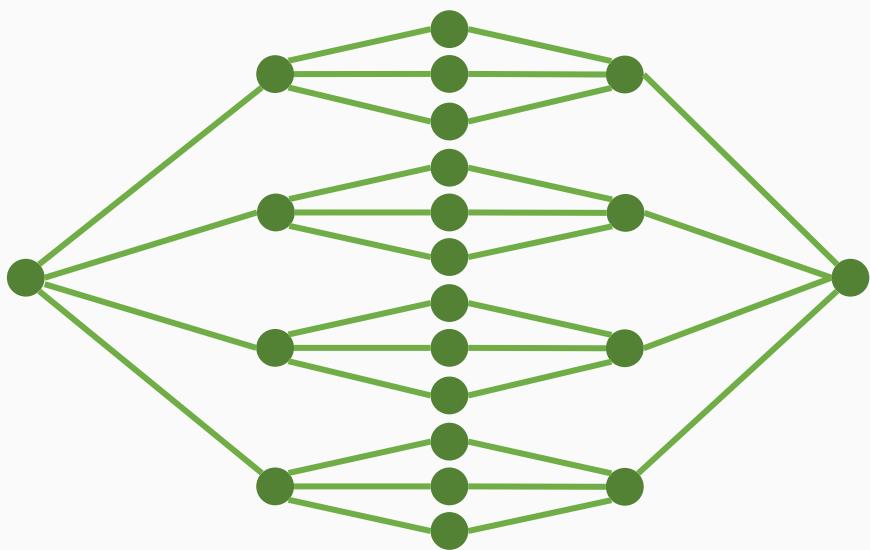
有興趣的話，請務必嘗試挑戰這個未解決的問題吧。

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。

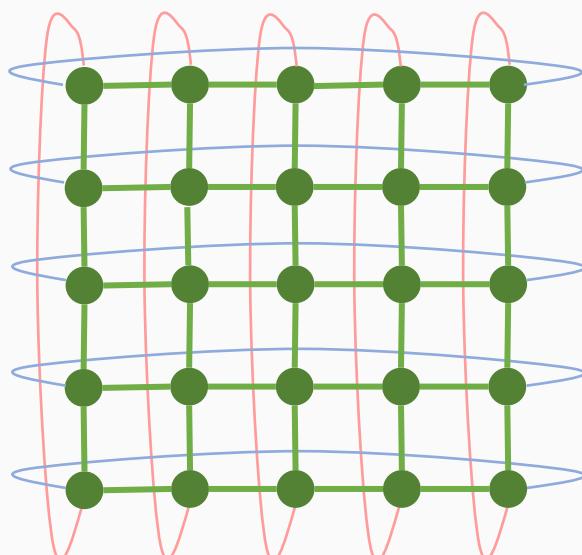
解法 0 — 問題描述中的例（22頂點）

這是在書中的問題描述亦有示範的包含22個頂點的圖。



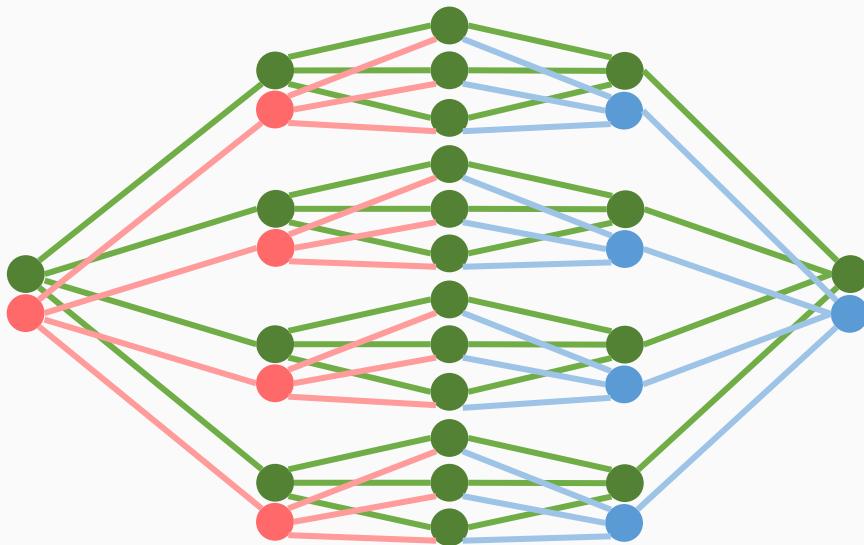
解法 1 — 5×5 的格子狀圖（25頂點）

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。

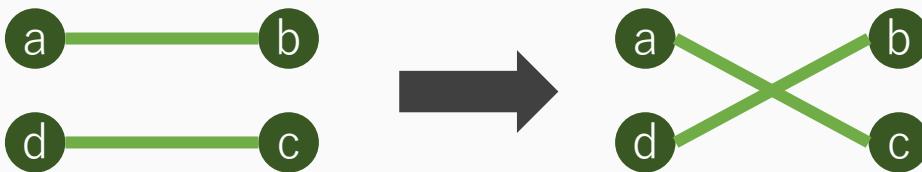
登山法的方針如下：

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 $(\text{度}d, \text{直徑}k) = (4, 4)$ 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。

第7章

結語

全部 184 頁的解說到此結束。雖然恐怕也有難以理解的部分，但是感謝閱讀到最後。

此外，練習問題的份量十分龐大，完成所有問題是不是需要相當大的精力呢？實際上節末問題和最終確認問題總計有 148 題，平均每題花費 30 分鐘的話，也需要花費 70 個小時以上。但是，如果透過這些問題，哪怕只有提升一點點實力，能對讀者有所幫助的話真的會很高興。

真的是最後了，在製作解說時獲得了米田寬峻先生（square1001）的協助。整體約2成左右是由他所撰寫的。非常感謝。

2021 年 12 月 29 日

米田 優峻

「演算法 × 數學」全彩圖解學習全指南

從基礎開始，一次學會 24 種必學演算法與背後的關鍵數學知識及應用

2021 年 12 月 29 日 バージョン 1 作成

製作者 米田 優峻

協助 米田 寛峻

發行 GitHub