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。

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

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

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。

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

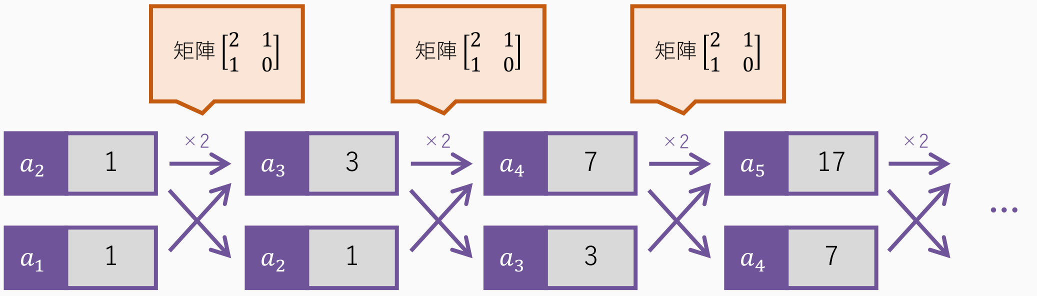
同樣地，因為 $a_4 = 2a_3 + a_2$、$a_3 = a_3$，以下的式會成立。

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

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。

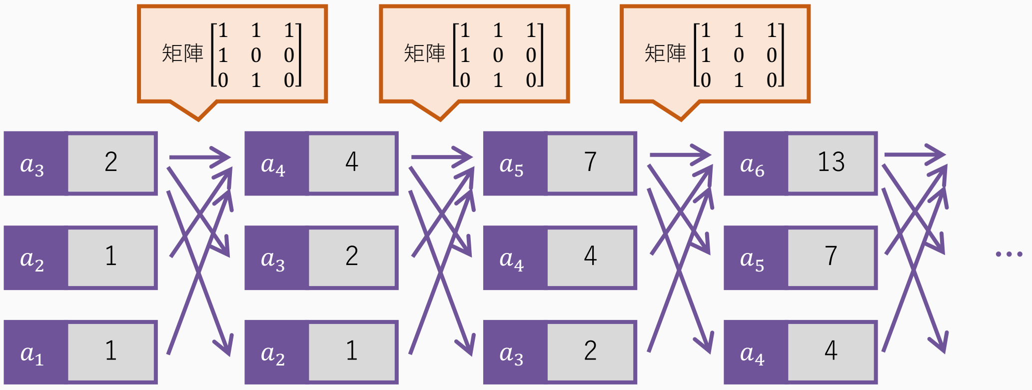
$$\begin{bmatrix} a_5 \\ a_4 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_4 \\ a_3 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^2 \begin{bmatrix} a_3 \\ a_2 \\ a_1 \end{bmatrix}$$

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 $[A^{N-1}$ 的 $(3,1)$ 元素×2 + $(3,2)$ 元素 + $(3,3)$元素] 表示。

下圖表示了矩陣與數列 $a = (a_1, a_2, a_3, \ldots)$ 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 3×3 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;
}
```

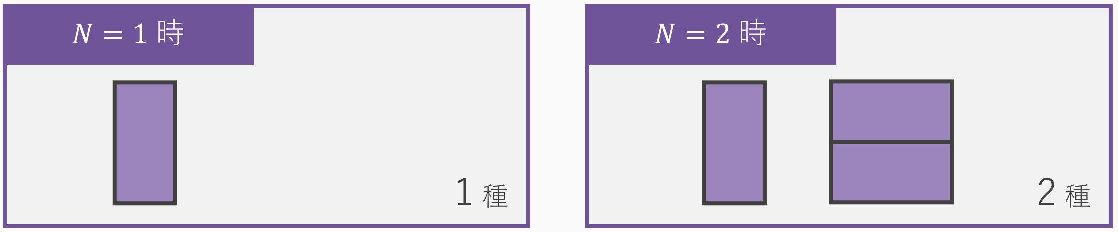
```
int main() {
    // 輸入 → 乘方的計算（注意若 N 小於 2 的話不會正確動作）
    long long N;
    cin >> N;

    // 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)

當從左開始依序鋪滿 $2\times k$ 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，將鋪滿 $2\times k$ 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 $a_2 = 1$。

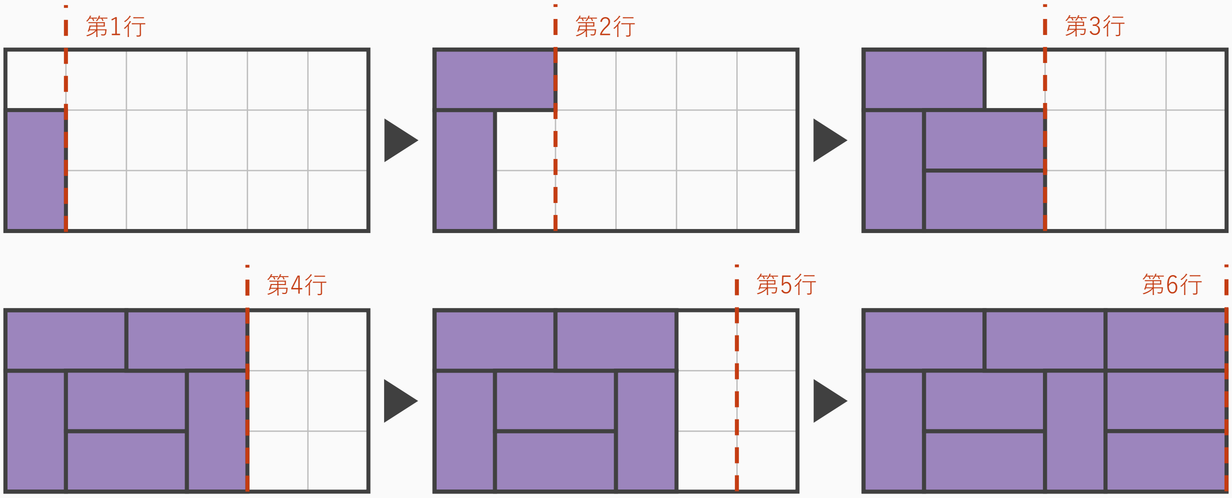


因此，$2\times N$ 時的答案為以下的費波那契數的第 $N+1$ 項，製作將其輸出的程式（→**4.7.1項**）即可得到正確答案。。

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

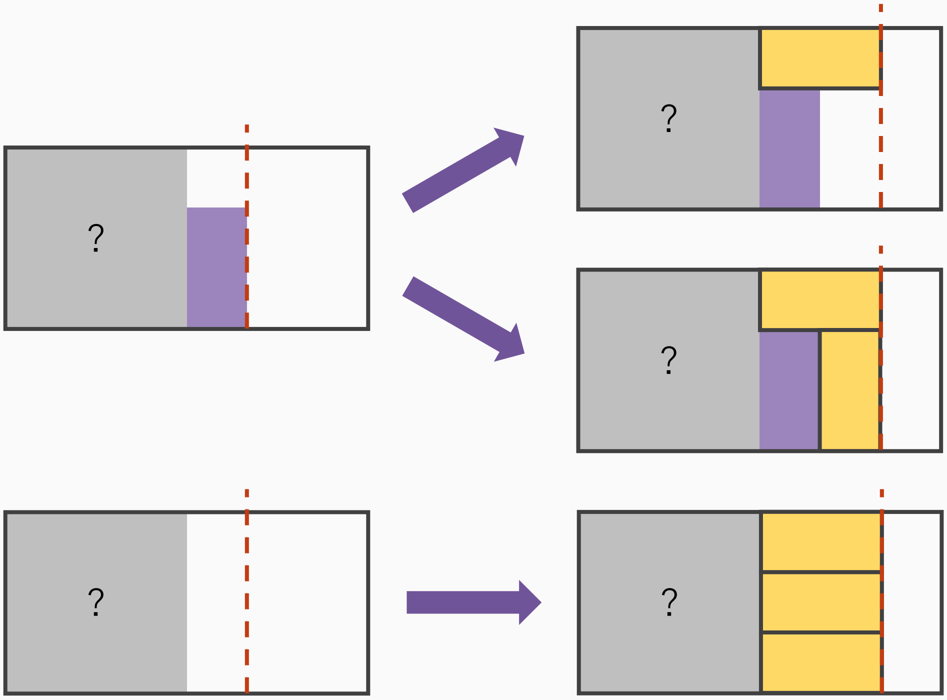
問題 4.7.4 (2), (3)

如下，考慮從左開始一行行鋪滿。當鋪到第 x 行為止時，橫跨第 x 行和第 $x+1$ 行的矩形不含在內。

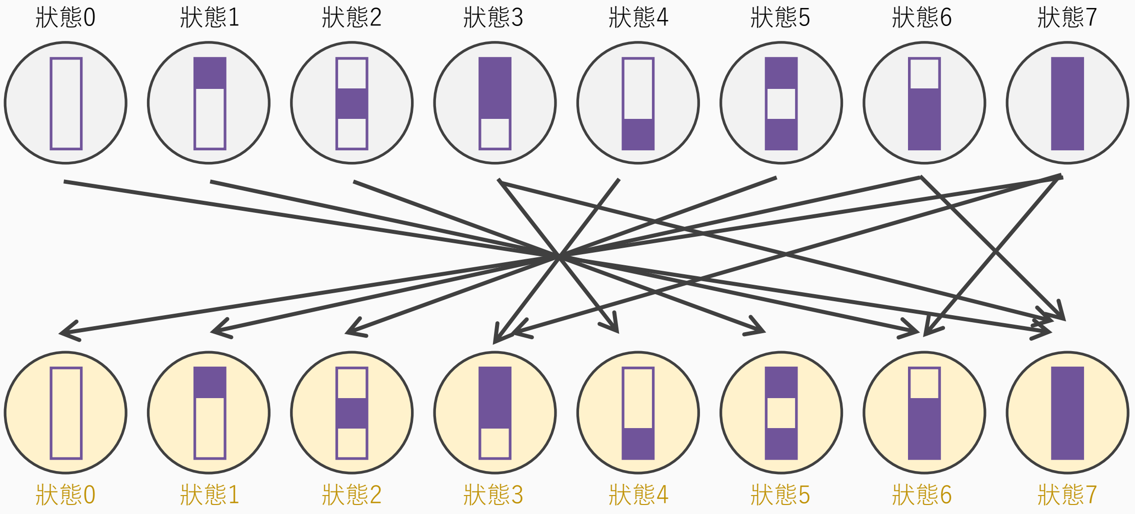


此時，「鋪到第 x 行為止時，一定已將第 $x-1$ 行前全部蓋住」的重要性質會成立，因此在此之後的鋪滿方式只依賴於最後一行（第 x 行）。

例如，當 $K=3$ 且下方的 2 列已經鋪滿時，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〇〇時，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 $0,1,\ldots,2^K-1$ 時，令 $dp[x][y]$ 為「在鋪到第 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，$L=2^K-1$。

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

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 行且為狀態〇〇的情況數 × 從狀態〇〇轉換到狀態 0 的方法數」對所有的狀態相加的總和。

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

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

例如，當 $K=3$ 時， $dp[N][0], dp[N][1], dp[N][2], \ldots, dp[N][7]$ 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）必須完全鋪滿，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, 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}
};

// 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, 1},
```

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

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;
}
```

```cpp
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;
}
```