

$$F(N, M)$$

## THE ONE FUNCTION

- 首先是 The One Function 的定義，我們可以把它拆成兩個 case
  - Case 1:  $M \leq N$ 
    - $F(N, M) = 1$
  - Case 2:  $M > N$ 
    - $F(N, M) = F(N, M-1) + F(N, M-2) + \dots + F(N, M-N)$
- 也就是說，固定  $N$  的值，將  $M$  不同的  $F(N, M)$  寫出來，則  $F(N, 1 \sim M)$  會是一個數列，**每項是前  $N$  項值的總和**

$$F(N, M)$$

## THE ONE FUNCTION

- 如果還是有點不懂，舉例而言，若將  $F(2, M)$  寫為一個數列，這個數列名字就叫做**費波納奇數列 (Fibonacci)**

M =	1	2	3	4	5	6	7	8	9	10
F(2, M)	1	1	2	3	5	8	13	21	34	55
F(3, M)	1	1	1	3	5	9	17	31	57	105
F(4, M)	1	1	1	1	4	7	13	25	49	94
F(5, M)	1	1	1	1	1	5	9	17	33	65

# MATRIX

## WHY MATRIX

- 接下來，可能會有些人好奇，好好的數列就相加就好，為什麼要用 matrix 處理呢？
- 原因就是因為，F 的特性是必須從頭開始計算，因此當 N 或 M 極大時，很容易就會超時了
- 而 matrix 能透過優化來加速這個過程
- 但在優化之前，我們首先得知道該怎麼透過矩陣運算來完成這件事



# MATRIX

## HOW TO IMPLEMENT

- 首先，我們期待能達成的矩陣是：每一行總和是我們想要的答案
- 以  $F(3, n)$  為例，初始矩陣會是

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{array}{l} F(3,1) = 1 \\ F(3,2) = 1 \\ F(3,3) = 1 \end{array}$$

- 如上所示，三行的總和分別是  $F(3, 1 \sim 3)$  的答案

# MATRIX

## HOW TO IMPLEMENT

- 而接下來，每次得到的新矩陣，最後一行會是原本矩陣中 N 行的總和
- 也因為矩陣只有  $n \times n$  大小，因此新的一行加入後舊的一行會被丟掉

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} F(3,1) = 1 \\ F(3,2) = 1 \\ F(3,3) = 1 \end{matrix} \quad \rightarrow \quad \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{matrix} F(3,2) = 1 \\ F(3,3) = 1 \\ F(3,4) = 3 \end{matrix}$$

- 如上所示，經過一次運算後，原本  $F(3, 1 \sim 3)$  的初始矩陣，應該被轉化為  $F(3, 2 \sim 4)$ ，之後的運算依此類推

# MATRIX

## HOW TO IMPLEMENT

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} F(4,1) = 1 \\ F(4,2) = 1 \\ F(4,3) = 1 \\ F(4,4) = 1 \end{matrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} F(4,2) = 1 \\ F(4,3) = 1 \\ F(4,4) = 1 \\ F(4,5) = 4 \end{matrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \end{bmatrix} \begin{matrix} F(4,3) = 1 \\ F(4,4) = 1 \\ F(4,5) = 4 \\ F(4,6) = 7 \end{matrix}$$

- 接下來進入正題，怎麼樣的運算能夠達成如上述
  - 每個新矩陣是
    - 上一個矩陣向上移一格
    - 且最新一行是上一個矩陣的總和
- 這樣的定義呢？



# MATRIX

## HOW TO IMPLEMENT

- 這樣的矩陣可以

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- (為了方便，以下都以 `ret` 稱呼這個矩陣)
- $F(N, a+1 \sim a+N+1) = \text{ret} * F(N, a \sim a+N)$
- 同學可以試著找找看，這些 `ret` 有什麼樣的共通點，又為什麼能達成這樣的效果

# MATRIX OPTIMIZATION

- 到目前為止，我們已經了解到
  - 初始的矩陣應該長什麼樣（每一行總和皆為1的矩陣，如單位矩陣）
  - 每一輪運算該做的事情（使  $\text{init} = \text{init} * \text{ret}$ ）
- 但如果只是這麼做，其實跟單純相加是一樣的
- 因此，接下來要討論的事是，如何優化這個矩陣算法



# MATRIX OPTIMIZATION

- 初始矩陣稱作  $init$ ，每輪要乘的矩陣稱作  $ret$ ，那麼當我們要計算  $F(N, N + 16)$  時，我們做的事情是
  - 執行 16 次  $init = init \cdot ret$ ，也就是  $init = init \cdot ret^{16}$
  - 但這麼做太久了，總共要 16 次運算
  - 有沒有辦法結合一些重複的運算，讓運算次數少一些呢？

# MATRIX OPTIMIZATION

- 我們把  $F(N, N + 16)$  改為
  - $ret = ret \cdot ret$  (此時的  $ret$  是原本的  $ret^2$ )
  - $ret = ret \cdot ret$  (此時的  $ret$  是原本的  $ret^4$ )
  - $ret = ret \cdot ret$  (此時的  $ret$  是原本的  $ret^8$ )
  - $ret = ret \cdot ret$  (此時的  $ret$  是原本的  $ret^{16}$ )
  - $init = init \cdot ret$  (此時的  $init$  是原本的  $init \cdot ret^{16}$ )
- 這麼一來只要 5 次運算就能代替原本的 16 次運算了！

# MATRIX OPTIMIZATION

- 這個算法的關鍵點在於，把  $ret^n$  拆成  $ret^{2^a} \cdot ret^{2^b} \cdot ret^{2^c} \cdot \dots$
- 這麼一來就能把原本  $n$  次的運算大大減少成  $a+b+c+\dots$  了
- Warning !
  - 這段由於比較複雜，下一張 ppt 會給同學參考用的 code，想要多一點思考空間的同學，可以直接按掉下一張以免被暴雷



# MATRIX OPTIMIZATION

- 也就是相較於原本的

```
while(n > 0) {  
    init = init * ret;  
    n - -;  
}
```

- 以類似於

```
while(n > 0) {  
    if (n %2 == 1) init = init * ret;  
    ret = ret * ret;  
    n /= 2;  
}
```

- 這樣的寫法，是能將時間從  $O(n)$  提升至  $O(\log n)$  的喔

# MATRIX OPTIMIZATION

- 於是，在了解這題需要的
  - 初始值
  - 迭代運算方式
  - 優化方式
- 以後，我們來看過一次各個 function，整理一下各自需要做什麼

# MATRIX

## FUNCTION IMPLEMENT

- `Matrix::Matrix(int n)`
  - 傳入矩陣大小後，constructor 應該要建出一個  $n * n$  大小的矩陣
  - 矩陣能以  $n * n$  二維陣列的概念來建立
- `long long& Matrix::operator()(const int& row, const int& column) const`
  - 回傳這個矩陣在這個位置所存的數字



# MATRIX

## FUNCTION IMPLEMENT

- `void Matrix::toidentity()`
  - 將矩陣轉為單位矩陣，也就是左上到右下對角線為 1，其餘為 0 的矩陣
  - 可以用來當作初始矩陣（因為各行總和為 1）
- `Matrix constructMatrix(int n)`
  - 建立 ret 矩陣並回傳給 base，使得後續能夠順利執行 base.power

# MATRIX

## FUNCTION IMPLEMENT

- `Matrix Matrix::power(int k) const`
  - 運算過程存在的地方，需要完成  $init = ret^k$  的運算，再將 `init` 作為運算結果回傳
  - 使得 `main` 能夠透過加總第 `N-1` 行來得到 `F(N, M)` 的答案
  - Hint: 由於執行 `power` 的 `base` 是經過 `constructMatrix` 的（也就是 `base` 相當於 `ret`），因此在 `power` 中，應該要先獲取一個 `identity matrix`（相當於 `init`），接著在迴圈中進行多次運算以得到答案，最後將其回傳

# MATRIX

## FUNCTION IMPLEMENT

- 以上就是關於 12767 - The One Function and The Power Of Matrix 的提示
- 如果在看完後還是有什麼問題，歡迎再到討論區或是寄信給助教討論喔
- 祝同學們期中順利！