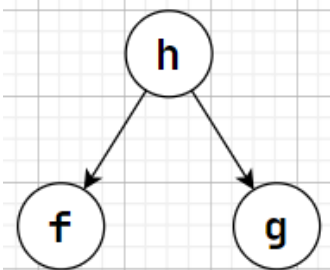


## G. 函式優化

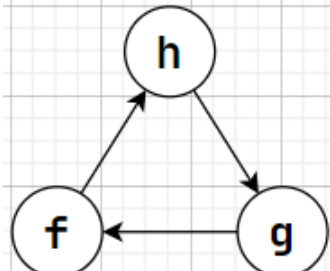
### Description

平時在寫程式的時候，code 裡面可能包含很多函式。這些函式相互之間可能會有依賴關係（比如呼叫某個函式時會需要呼叫另一個函式）。我們可以把函式之間的關係畫成一張有向圖，稱為 Call Graph。圖上各個節點代表函式，而從節點  $u$  連出一條有向邊至節點  $v$  則函式  $u$  時會呼叫函式  $v$ （也就是計算函式  $u$  的結果會需要知道函式  $v$  的結果）。

以下方程式碼為例，一共有 3 個函數  $f, g, h$ ，互相之間的依賴關係如右圖：

Code	Call Graph
<pre> int f(int x) {     return x ^ 123456; } int g(int x) {     return x * x; } int h(int x) {     return f(x) + g(x); } </pre>	 <pre> graph TD     h((h)) --&gt; f((f))     h((h)) --&gt; g((g)) </pre>

撰寫程式時可能會不小心寫出兩個函式互相呼叫或是多個函式之間的依賴關係形成環 (Recursive Calls)，並且沒有適當終止條件，這樣的 bug 會導致程式無法結束執行或是 stack overflow：

Code	Call Graph
<pre> int f() {     return h() ^ 123; } int g() {     return f() ^ 456; } int h() {     return g() ^ 789; } </pre>	 <pre> graph TD     f((f)) --&gt; h((h))     h((h)) --&gt; g((g))     g((g)) --&gt; f((f)) </pre>

而有的時候，儘管 code 沒有出現任何 recursive call，函式之間的依賴關係有可能使得函式被呼叫的次數過多，因而造成程式執行效率顯著降低。

Code	Call Graph
<pre> int f1() { return 1; } int f2() { return 1; } int f3() { return f1() + f2(); } int f4() { return f2() + f3(); } int f5() { return f3() + f4(); } ... int f99() { return f97() + f98(); } int f100() { return f98() + f99(); } </pre>	

細心的你應該已經發現了上方的函式結構和費式數列一樣，費式數列的成長速度為指數量級 ( $O(\phi^n) \approx O(1.618^n)$ )，所有函式的總呼叫次數也會是此量級。當函式越來越多的時候，所需執行時間也會以很可怕的速度增長。

了解這些之後，就來挑戰看看這道問題吧！給你一份程式碼，這份程式碼以三個非負整數  $x, y, z$  作為輸入，經過一連串神秘的運算過程之後，最終輸出一個整數  $ans$ 。這份 code 包含了 200 個函式，運算答案的過程皆會用到這些函式。然而程式執行效率非常低，沒辦法在足夠短的時間內算出答案，你的目標便是優化這份程式碼，使其能夠在 1 秒之內算出答案。思考看看如何減少函式的呼叫次數吧！

## Implementation Details

請下載附件「TLE.cpp」，你需要想辦法優化這份程式碼並上傳。

保證所有函式之間不存在 recursive call。

## Input

第一行有三個非負整數  $x, y, z$ ，以空白間隔，表示這份程式碼的輸入。

- $0 \leq x, y, z < 777771449$

## Output

輸出只有一個整數，表示這份程式碼的輸出。

## Sample 1

Input	Output
-------	--------

1 2 3	701423903
-------	-----------

## Sample 2

Input	Output
89921 0 777771448	614811106