Задача

Напишете програма на C или C++, която прочита от стандартния си вход десетичен запис на естествено число $1 \leq N \leq 10^{10000}$ и отпечатва последните 9 цифри на N-тото число на Фибоначи.

Примери:

За вход "10", програмата трябва да отпечатва "000000055".

За вход 7777 седмици, програмата трябва да отпечатва "590199457", за 7776 седмици и 8 - "332126689", за 7776 седмици и 9 - "922326146".

Изисквания:

Дължина до 2000 символа и време до 2 секунди. Печели най-бързото решение.

Анализ на задачата

Разглеждаме редицата от наредени двойки $\langle F_0 \bmod n, F_1 \bmod n\rangle$, $\langle F_1 \bmod n, F_2 \bmod n\rangle$, ..., където $a \bmod b$ означава остатък при целочислено делене на a с b. Тъй като има само краен брой такива двойки, то съществува минимално $c > 0$, такова че $\langle F_0 \bmod n, F_1 \bmod n\rangle = \langle F_c \bmod n, F_{c+1} \bmod n\rangle$. Оттук следва, че $F_k \bmod n = F_{k+c} \bmod n$ за всяко k и в частност за да намерим $F_k \bmod n$ е достатъчно да намерим $F_{k \bmod c} \bmod n$. Търсеното c е известно като *период на Пизано за* n и при $n = 1\ 000\ 000\ 000$, неговата стойност е 1 500 000 000.

Първата част от задачата е намирането на остатъка на входа по модул 1 500 000 000. Това може да се изчисли ефективно по следния начин: Нека α е остатъкът при делене с 3 на сумата от цифрите без последните 8 или 0, ако няма такива и нека β е остатъкът при делене с 5 на 9-тата цифра отзад напред или 0, ако няма такава. Сега остатъкът при делене с 1 500 000 000 има синтактичния вид xy, където x е $\beta + 5.((3 + \beta - \alpha) \bmod 3)$, а y са последните най-много 8 цифри от входа.

Оставям коректността на това твърдение за упражнение на читателите!

Сега се интересуваме от m-тото число на Фибоначи. Бърз алгоритъм за това е следният:

```
func Fibonacci(m):
	Let a := 0, b := 1, x := 2
	Let bin := the binary representation of m

	for each bit q in bin:
		Let a' := a * a
		Let b' := b * b

		a := 2 * b' - 3 * a' - x
		b := a' + b'

		if (q is set):
			Let t := a + b
			a := b
			b := t
			x := -2
		else:
			x := 2

	return a
```

Защо алгоритъмът е коректен?

Лесно се доказва с индукция, че:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

, откъдето

$$(-1)^n = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^n = \begin{vmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{vmatrix} = F_{n+1}F_{n-1} - F_n^2 \Longrightarrow F_{n+1}F_n = F_{n+1}^2 - F_n^2 - (-1)^n$$

Освен това:

$$\begin{pmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{2n} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}^2$$

Сега:

$$F_{2n+1} = F_{n+1}^2 + F_n^2$$

$$F_{2n} = F_nF_{n+1} + F_{n-1}F_n = 2.F_{n+1}F_n - F_n^2 = 2.F_{n+1}^2 - 3.F_n^2 - 2.(-1)^n$$

Основното наблюдение относно коректността на алгоритъма е, че на всяка стъпка от цикъла a съдържа F_k, b съдържа F_{k+1}, а x съдържа $2.(-1)^k$, където k е числото, образувано от битовете на m до текущо разглеждания.

- В началото няма битове, така че $k = 0$, $a = F_0$, $b = F_1$, $x = 2$.
- Ако текущият бит не е сетнат, то от F_k и F_{k+1} директно получаваме F_{2k}, F_{2k+1} на горния принцип и инвариантата се запазва, включително и за x.
- Ако текущият бит е сетнат, то правим още едно събиране за да получм F_{2k+1}, F_{2k+2} и инвариантата отново се запазва на следващата итерация, пак включително и за x.
- След края на цикъла a съдържа търсеното число на Фибоначи.

Коментари:

- Посоченият метод може лесно да се трансформира до търсене на остатък по модул като това става при пресмятането на a, b и t. Единственото, което заслужава внимание е, че преди да вземем остатъка трябва да прибавим достатъчно голямо число към a, така че то да остане неотрицателно.
- Алгоритъмът е подходящ за намиране на много големи числа на Фибоначи, тъй като прави само $2.\log_2 m$ умножения на големи числа. Това е сравнително скъпа операция при работа с такива данни и затова е необходимо тя да се използва колкото се може по-малко.
- Равенството $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$ може да се използва директно за намиране на F_n например чрез ***Exponentiation by squaring*** метод. Това обаче става за сметка на поне 5 умножения на стъпка.
- Равенствата $F_{2n+1} = F_{n+1}^2 + F_n^2$ и $F_{2n} = 2.F_{n+1}F_n - F_n^2$ могат да се ползват и в този вид (този метод е известен като ***Fibonacci fast doubling***), но това е за сметка на 3 умножения на стъпка.

Анализ на предоставените решения

Най-бързите получени решения са 4 - DA (използва матричното равенство и итеративна версия на Exponentiation by squaring), BS (използва рекурсивна версия на Fibonacci fast doubling), LH (използва рекурсивна версия на вариант на Fibonacci fast doubling) и IB (отново итеративен Exponentiation by squaring).

Истинско предизвикателство за автора се оказа сравнението на посочените програми по бързодействие, тъй като работата на всяка беше до 1 ms върху всеки от предварителните тестове. С цел значително по-голяма точност се наложи да модифицирам решенията, така че времето за изчисление да се отпечатва от самите програми. Резултатните програми са компилирани (g++ v10.2.0 с флагове -O3 и -std=c++17) и тествани в Cygwin 2.908 среда на 64-битов Windows 10 Pro с процесор Intel Core i5-9400F. Тестовете представляват 20 файла, всеки с 9999 цифрено число.

Получените резултати са както следва:

Тест	Очакван изход	LH			DA			BS			IB			SF		
		✓	All	Comp	✓	All	Comp	✓	All	Comp	✓	All	Comp	✓	All	Comp
t1	394598269	ok	2323500	10900	ok	613600	8200	ok	1722600	28900	ok	203600	39900	ok	2320500	5700
t2	453931885	ok	2336500	10000	ok	575400	8100	ok	2006800	31100	ok	174100	34900	ok	2304300	5600
t3	217825896	ok	2321600	9800	ok	520200	7200	ok	1815200	30400	ok	156300	31000	ok	2381100	5800
t4	873605602	ok	2372700	10300	ok	515700	6300	ok	1808300	25700	ok	149100	28700	ok	2369300	5900
t5	200832369	ok	2362200	10000	ok	452300	5900	ok	1755700	27600	ok	148400	28600	ok	2380800	5800
t6	951126785	ok	2369600	10100	ok	447400	5900	ok	1890300	28200	ok	148200	28600	ok	2383400	5700
t7	620811681	ok	2375300	6800	ok	441900	5900	ok	1819200	23000	ok	143400	28500	ok	2429200	5700
t8	853602437	ok	2384600	10000	ok	443800	5800	ok	1789400	26100	ok	149800	28600	ok	2350900	5800
t9	606647175	ok	2384900	11900	ok	445200	5900	ok	1773100	27700	ok	147900	28600	ok	2325700	5800
t10	795056930	ok	2358700	10200	ok	450600	5900	ok	1782800	27800	ok	148100	28600	ok	2352200	5800
t11	188734031	ok	2354200	9900	ok	492900	5900	ok	1780400	27300	ok	143300	28600	ok	2362800	5800
t12	974008987	ok	2381200	9900	ok	472400	6100	ok	1855500	27000	ok	146400	28700	ok	2351800	5700
t13	807527775	ok	2367200	9900	ok	475900	5900	ok	1769000	28200	ok	144100	28700	ok	2374200	5800
t14	398609403	ok	2549200	10700	ok	456900	5800	ok	1742900	25800	ok	145500	28600	ok	2367000	5700
t15	595101161	ok	2524800	10700	ok	475500	5800	ok	1764800	25100	ok	144000	28600	ok	2329900	5800
t16	763714037	ok	2535800	9300	ok	480300	6100	ok	1807500	24100	ok	148000	28600	ok	2351400	5700
t17	254064975	ok	2577900	10400	ok	451700	5900	ok	1752000	29000	ok	156400	28700	ok	2369000	5900
t18	644227217	ok	2365000	9900	ok	462700	5900	ok	1761200	29000	ok	142700	28600	ok	2406700	5800
t19	882893301	ok	2363200	10100	ok	459500	5800	ok	1909000	26900	ok	145600	28600	ok	2356700	5800
t20	290867985	ok	2346600	10400	ok	455400	5900	ok	1737500	28100	ok	146000	28700	ok	2384100	5800
Средно време			2397735	10060		479465	6210		1802160	27350		151545	29620		2362550	5770

Легенда:
All – Общо време за изпълнение на програмата в ns.
Comp – Време за изпълнение без входно-изходните операции в ns.
SF – авторско решение (не участва в състезанието)

Най-бързото решение относно цялостно изпълнение на програмата е IB (средно време 151545 ns или около 0.15 ms), а най-бързото решение относно самото намиране на резултата (без входно-изходни операции) е DA (средно време 6210 ns или около 0.006 ms).

Всички решения и тестове могат да се видят на:
https://github.com/mr-fotev/mr-fotev.github.io/tree/master/MNKnowledge/excFibo