Scrumptious Scrumping & Matrix Exponentiation

Recently, I wrote a set of Programming questions on hackerrank for the Maynooth Halloween Programmathon (aka the Spookathon). I was particularly happy with the framing for the task *Scrumptious Scrumping*, the fourth task in the main round.

However, people definitely struggled with the question. Moreover, there was a harder variant of the question which nobody got during the contest. To start with, I'll go over the original task, how to solve the initial variant of the question, and then how to handle the tougher version.

Task statement

The contestants, being familiar with the character of Nimrod the mouse, were met with this blurb:

"On his way to the Malicious Monster Mash, Nimrod traverses the deep dark woods. Therein, he is met with a spooky orchard, host to an inconceivably devious array of apple trees.

- To start, Nimrod holds 1 *good* apple. He throws the *good* apple at a tree, which causes the tree to drop another *good* apple. However, his *good* apple is now a *bad* apple.
- Nimrod now holds 1 *good* apple and 1 *bad* apple. Again, he throws his *good* apple at a tree, which causes it to turn into a *bad* apple, and the tree drops a *good* apple. He also throws his *bad* apple, which causes a tree to drop a *good* apple, however on impact the *bad* apple explodes and disappears.
- Nimrod now holds 1 *bad* apple and 2 *good* apples. He repeats this process, throwing all of his apples at trees, where each *bad* apple explodes on impact, and each impact causes a tree to drop a *good* apple. Two steps later, Nimrod holds 3 *bad* apples and 5 *good* apples.

How many *good* apples does he have at the end of the n-th step?"

(This task was inspired by the real world actions of my teammate in procuring ingredients for an apple crumble.)

The contestants are also given example inputs/outputs. So if $n = 3$, the answer is 3, if $n = 5$, the answer is 8 and so on. Let's analyse this question.

Solution

Consider the following 1-indexed arrays:

- G, an array where $G[i]$ is how many *good* apples Nimrod has at step i.
- B, an array where $B[i]$ is how many *bad* apples Nimrod has at step i.

It should be apparent, then, that to solve the question is really to compute $G[n]$, i.e., how many good apples Nimrod has at step n. Because Nimrod starts with just 1 *good* apple, which he throws his at a tree, turning into a *bad* apple, but which yields a *good* apple. So at step one Nimrod has 1 *good* apple and one *bad* apple. Hence, $G[1] = 1$ and $B[1] = 1$. We can follow this kind of reasoning to compute the first few values of G and B, shown below:

$$G = [1, 1, 2, 3, 5, 8, \ldots]$$

$$B = [0, 1, 1, 2, 3, 5, \ldots]$$

You might be able to spot here that B seems to just be G but with an extra 0 at the start, can we show this? Well suppose we've calculated $G[i]$ and $B[i]$ for some step i. The question is, how do we calculate $G[i+1]$ and $B[i+1]$, the next terms in the arrays?

Well, when Nimrod begins a step, he takes all his bad apples and throws them, all of which are destroyed on impact. Then he takes all his good apples and throws them, at which point they all become bad apples. So the only source of bad apples at step $i+1$ are the good apples at step i. Or, put simply, $B[i+1] = G[i]$. So we can see here that B really is just a shifted version of G. But what's $G[i+1]$? Well, Nimrod gets good apples from any tree that any apple has hit. At the start of turn $i+1$, he throws $G[i]$ good apples and $B[i]$ bad apples, which gives a total of $G[i] + B[i]$ throws. Each throw gets a new good apple, so, $G[i+1] = G[i] + B[i]$.

So we can answer the question using these rules:

- $B[1] = 1$,
- $G[1] = 1$,
- $B[i+1] = G[i]$,
- $G[i+1] = G[i] + B[i]$.

Here's the working python3 code for this answer:

```
n = int(input())

G = [0]*(n+1)
B = [0]*(n+1)

G[1] = 1
B[1] = 1

for i in range(1,n):
        B[i+1] = G[i]
        G[i+1] = G[i]+B[i]

print(G[n])
```

Something to note before I continue: The contestants were asked to print the answer modulo 1000000007, as the numbers grow exponentially. So, in truth, the second last line must be rewritten as $G[i+1] = (G[i] + B[i])\%1000000007$ for full points.

While this does answer the question correctly, it risks missing something. Because B is just a copy of G, there's no point in keeping it around. If we want to compute $G[i+2]$ for some step i, we can do the following manipulations: $G[i+2] = G[i+1] + B[i+1]$, but we know $B[i+1] = G[i]$. Hence, $G[i+2] = G[i+1] + G[i]$. Or, in other words, each term of G is just the sum of the previous two terms!

This should be clear from the example above: $1+1=2$, $1+2=3$, $2+3=5$, $3+5=8$, etc. For those who recognise it, this is just the Fibonacci sequence. So this question was really just asking the contestant to find F_n, the n-th Fibonacci number! So the standard code below for fibonacci will give the correct answers:

```
n = int(input())

F = [0]*(n+2)

F[0] = 1
F[1] = 1

for i in range(n):
        F[i+2] = (F[i+1]+F[i])%1000000007

print(F[n])
```

(You may notice that to compute a given term we only need the two terms which came before. With this in mind, you don't actually need to store the *whole* array for F, just store the previous two values. This brings the memory complexity down significantly.)

Harder variant

The solution above works fine in the case where n is reasonably sized. However, if n were as big as 10^9 (1 billion), the for-loop would take too long to run and the contestant would be met with a "Time limit exceeded" result. In order to handle this, we'll need an optimisation trick.

The solution above is a bottom-up dynamic programming solution, with the recurrence relation given by $F_n + F_{n+1} = F_{n+2}$ and initial conditions $F_0 = 1$, $F_1 = 1$. The key to speeding this up is to notice that the recurrence relation is *linear*.

That is to say, F_{n+2} is just a simple linear combination of F_n and F_{n+1}. It would therefore be reasonable to bring linear algebra into the equation. Namely, notice the following:

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

Hopefully you can see from the above expression that, by multiplying the vector $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$ on the left by the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, we effectively "bump" the vector up to the next pair $\begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$. What's more, we can do this as many times as we like:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+3} \\ F_{n+2} \end{pmatrix}.$$

We can use this trick to get to the vector $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$ from the starting vector $\begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}.$$

So, to compute F_n, all we really have to do is to compute $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$, then multiply whatever matrix you get by the vector $\begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, the resulting vector will contain the answer F_n.

Computing powers quickly is a classic problem in competition programming. Here, the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ may be computed quickly using fast squaring, which gives the full solution for the harder variant!

Here's a working python solution for this harder variant:

```python
n = int(input())

# Multiply two 2-by-2 matrices together, take the answer mod 1000000007
def mat_mult(A, B):
        return [[(A[i][0]*B[0][j]+A[i][1]*B[1][j])%1000000007 for j in [0,
1]] for i in [0,1]]
```

```python
# Compute the fast-squaring algorithm
def fast_square(power):
        if power == 0:
                return [[1,0],[0,1]]
        H = fast_square(power//2)
        F = mat_mult(H, H)
        return (F if power%2 == 0 else mat_mult([[1,1],[1,0]], F))

# Answer is the first entry in the resulting matrix
print(fast_square(n)[0][0])
```

This type of optimisation works *anytime* you have a finite linear recurrence relation. So, for example, if you had to compute L_n, where $L_{n+4} = L_n + 2L_{n+1} - L_{n+3}$ was the recurrence relation, you could do the exact same algorithm with the matrix $\begin{pmatrix} -1 & 0 & 2 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ acting on the vector $\begin{pmatrix} L_{n+3} \\ L_{n+2} \\ L_{n+1} \\ L_n \end{pmatrix}$.

This technique even works with so-called inhomogeneous linear recurrences, like $H_{n+2} = H_{n+1} + H_n + 1$, where you can use the matrix $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ acting on the vector $\begin{pmatrix} H_{n+1} \\ H_n \\ 1 \end{pmatrix}$.

Also, finally relating back to the Nimrod story, you may read the above matrix off in the following way, as telling you how to get from one step to the next: The top row $(1, 1)$ says that the amount of *good* apples in the next step is equal to the amount of *good* apples in the previous step plus the amount of *bad* apples in the previous step, and the bottom row $(1, 0)$ says that the amount of *bad* apples in the next step is equal to the amount of *good* apples in the previous step.