第 4 章

递归和动态规划

斐波那契系列问题的递归和动态规划

【题目】

给定整数 N，返回斐波那契数列的第 N 项。

【补充题目 1】

给定整数 N，代表台阶数，一次可以跨 2 个或者 1 个台阶，返回有多少种走法。

【举例】

N=3，可以三次都跨 1 个台阶；也可以先跨 2 个台阶，再跨 1 个台阶；还可以先跨 1 个台阶，再跨 2 个台阶。所以有三种走法，返回 3。

【补充题目 2】

假设农场中成熟的母牛每年只会生 1 头小母牛，并且永远不会死。第一年农场有 1 只成熟的母牛，从第二年开始，母牛开始生小母牛。每只小母牛 3 年之后成熟又可以生小母牛。给定整数 N，求出 N 年后牛的数量。

【举例】

N=6，第 1 年 1 头成熟母牛记为 a；第 2 年 a 生了新的小母牛，记为 b，总牛数为 2；

第 3 年 a 生了新的小母牛，记为 c，总牛数为 3；第 4 年 a 生了新的小母牛，记为 d，总牛数为 4。第 5 年 b 成熟了，a 和 b 分别生了新的小母牛，总牛数为 6；第 6 年 c 也成熟了，a、b 和 c 分别生了新的小母牛，总牛数为 9，返回 9。

【要求】

对以上所有的问题，请实现时间复杂度 $O(\log N)$的解法。

【难度】

将 ★★★★

【解答】

原问题。$O(2^N)$的方法。斐波那契数列为 1，1，2，3，5，8，…，也就是除第 1 项和第 2 项为 1 以外，对于第 N 项，有 $F(N)=F(N-1)+F(N-2)$，于是很轻松地写出暴力递归的代码。请参看如下代码中的 f1 方法。

```
public int f1(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2) {
                return 1;
        }
        return f1(n - 1) + f1(n - 2);
}
```

$O(N)$的方法。斐波那契数列可以从左到右依次求出每一项的值，那么通过顺序计算求到第 N 项即可。请参看如下代码中的 f2 方法。

```
public int f2(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2) {
                return 1;
        }
        int res = 1;
        int pre = 1;
        int tmp = 0;
        for (int i = 3; i <= n; i++) {
                tmp = res;
                res = res + pre;
```

```
                pre = tmp;
            }
            return res;
    }
```

$O(\log N)$的方法。如果递归式严格遵循 $F(N)=F(N-1)+F(N-2)$，对于求第 N 项的值，有矩阵乘法的方式可以将时间复杂度降至 $O(\log N)$。$F(n)=F(n-1)+F(n-2)$，是一个二阶递推数列，一定可以用矩阵乘法的形式表示，且状态矩阵为 2×2 的矩阵：

$$\big(F(n),F(n-1)\big)=\big(F(n-1),F(n-2)\big)\times\begin{vmatrix}a & b\\ c & d\end{vmatrix}$$

把斐波那契数列的前 4 项 $F(1)==1$，$F(2)==1$，$F(3)==2$，$F(4)==3$ 代入，可以求出状态矩阵：

$$\begin{vmatrix}a & b\\ c & d\end{vmatrix}=\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}$$

求矩阵之后，当 $n>2$ 时，原来的公式可化简为：

$$\big(F(3),F(2)\big)=\big(F(2),F(1)\big)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}=(1,1)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}$$

$$\big(F(4),F(3)\big)=\big(F(3),F(2)\big)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}=(1,1)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}^2$$

$$\vdots$$

$$\big(F(n),F(n-1)\big)=\big(F(n-1),F(n-2)\big)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}=(1,1)\times\begin{vmatrix}1 & 1\\ 1 & 0\end{vmatrix}^{n-2}$$

所以，求斐波那契数列第 N 项的问题就变成了如何用最快的方法求一个矩阵的 N 次方的问题，而求矩阵 N 次方的问题明显是一个能够在 $O(\log N)$时间内解决的问题。为了表述方便，我们现在用求一个整数 N 次方的例子来说明，因为只要理解了如何在 $O(\log N)$的时间复杂度内求整数 N 次方的问题，对于求矩阵 N 次方的问题是同理的，区别是矩阵乘法和整数乘法在细节上有些不一样，但对于怎么乘更快，两者的道理相同。

假设一个整数是 10，如何最快地求解 10 的 75 次方。

1．75 的二进制数形式为 1001011。

2．10 的 75 次方$=10^{64}\times10^{8}\times10^{2}\times10^{1}$。

在这个过程中，我们先求出 10^1，然后根据 10^1 求出 10^2，再根据 10^2 求出 10^4，……，最后根据 10^{32} 求出 10^{64}，即 75 的二进制数形式总共有多少位，我们就使用了几次乘法。

3．在步骤 2 进行的过程中，把应该累乘的值相乘即可，比如 10^{64}、10^8、10^2、10^1 应该累乘，因为 64、8、2、1 对应到 75 的二进制数中，相应的位上是 1；而 10^{32}、10^{16}、10^4 不应该累乘，因为 32、16、4 对应到 75 的二进制数中，相应的位上是 0。

对矩阵来说同理，求矩阵 m 的 p 次方请参看如下代码中的 matrixPower 方法。其中 muliMatrix 方法是两个矩阵相乘的具体实现。

```
public int[][] matrixPower(int[][] m, int p) {
	int[][] res = new int[m.length][m[0].length];
	// 先把 res 设为单位矩阵，相当于整数中的 1
	for (int i = 0; i < res.length; i++) {
		res[i][i] = 1;
	}
	int[][] tmp = m;
	for (; p != 0; p >>= 1) {
		if ((p & 1) != 0) {
			res = muliMatrix(res, tmp);        快速幂
		}
		tmp = muliMatrix(tmp, tmp);
	}
	return res;
}

public int[][] muliMatrix(int[][] m1, int[][] m2) {        矩阵相乘
	int[][] res = new int[m1.length][m2[0].length];
	for (int i = 0; i < m2[0].length; i++) {
		for (int j = 0; j < m1.length; j++) {
			for (int k = 0; k < m2.length; k++) {
				res[i][j] += m1[i][k] * m2[k][j];
			}
		}
	}
	return res;
}
```

用矩阵乘法求解斐波那契数列第 N 项的全部过程请参看如下代码中的 f3 方法。

```
public int f3(int n) {
	if (n < 1) {
		return 0;
	}
	if (n == 1 || n == 2) {
		return 1;
	}
	int[][] base = { { 1, 1 }, { 1, 0 } };
	int[][] res = matrixPower(base, n - 2);
	return res[0][0] + res[1][0];
}
```

补充问题 1。如果台阶只有 1 级，方法只有 1 种。如果台阶有 2 级，方法有 2 种。如果台阶有 N 级，最后跳上第 N 级的情况，要么是从 N-2 级台阶直接跨 2 级台阶，要么是从 N-1 级台阶跨 1 级台阶，所以台阶有 N 级的方法数为跨到 N-2 级台阶的方法数加上跨到 N-1

级台阶的方法数，即 $S(N)=S(N\text{-}1)+S(N\text{-}2)$，初始项 $S(1)==1$，$S(2)==2$。所以类似斐波那契数列，唯一的不同就是初始项不同。可以很轻易地写出 $O(2^N)$与 $O(N)$的方法，请参看如下代码中的 s1 和 s2 方法。

```
public int s1(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2) {
                return n;
        }
        return s1(n - 1) + s1(n - 2);
}

public int s2(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2) {
                return n;
        }
        int res = 2;
        int pre = 1;
        int tmp = 0;
        for (int i = 3; i <= n; i++) {
                tmp = res;
                res = res + pre;
                pre = tmp;
        }
        return res;
}
```

$O(\log N)$的方法。表达式 $S(n)=S(n\text{-}1)+S(n\text{-}2)$是一个二阶递推数列，同样用上文矩阵乘法的方法，根据前 4 项 $S(1)==1$，$S(2)==2$，$S(3)==3$，$S(4)==5$，求出状态矩阵：

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}$$

同样根据上文的过程得到：

$$\big(S(n), S(n-1)\big) = \big(S(2), S(1)\big) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2} = (2,1) \times \begin{vmatrix} 1 & 1 \\ 1 & 0 \end{vmatrix}^{n-2}$$

全部的实现请参看如下代码中的 s3 方法。

```
public int s3(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2) {
                return n;
```

```
		}
		int[][] base = { { 1, 1 }, { 1, 0 } };
		int[][] res = matrixPower(base, n - 2);
		return 2 * res[0][0] + res[1][0];
	}
```

补充问题 2。所有的牛都不会死，所以第 N-1 年的牛会毫无损失地活到第 N 年。同时所有成熟的牛都会生 1 头新的牛，那么成熟牛的数量如何估计？就是第 N-3 年的所有牛，到第 N 年肯定都是成熟的牛，其间出生的牛肯定都没有成熟。所以 $C(n)=C(n-1)+C(n-3)$，初始项为 C(1)==1，C(2)==2，C(3)==3。这个和斐波那契数列又十分类似，只不过 $C(n)$依赖 $C(n-1)$和 $C(n-3)$的值，而斐波那契数列 $F(n)$依赖 $F(n-1)$和 $F(n-2)$的值。同样可以很轻易地写出 $O(2^N)$与 $O(N)$的方法，请参看如下代码中的 c1 和 c2 方法。

```
	public int c1(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2 || n == 3) {
			return n;
		}
		return c1(n - 1) + c1(n - 3);
	}

	public int c2(int n) {
		if (n < 1) {
			return 0;
		}
		if (n == 1 || n == 2 || n == 3) {
			return n;
		}
		int res = 3;
		int pre = 2;
		int prepre = 1;
		int tmp1 = 0;
		int tmp2 = 0;
		for (int i = 4; i <= n; i++) {
			tmp1 = res;
			tmp2 = pre;
			res = res + prepre;
			pre = tmp1;
			prepre = tmp2;
		}
		return res;
	}
```

$O(\log N)$的方法。$C(n)=C(n-1)+C(n-3)$是一个三阶递推数列，一定可以用矩阵乘法的形式表示，且状态矩阵为 3×3 的矩阵。

$$(C_n, C_{n-1}, C_{n-2}) = (C_{n-1}, C_{n-2}, C_{n-3}) \times \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

把前 5 项 C(1)==1，C(2)==2，C(3)==3，C(4)==4，C(5)==6 代入，求出状态矩阵：

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}$$

求矩阵之后，当 n>3 时，原来的公式可化简为：

$$(C_n, C_{n-1}, C_{n-2}) = (C_3, C_2, C_1) \times \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}^{n-3} = (3,2,1) \times \begin{vmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{vmatrix}^{n-3}$$

接下来的过程又是利用加速矩阵乘法的方式进行实现，具体请参看如下代码中的c3方法。

```
public int c3(int n) {
        if (n < 1) {
                return 0;
        }
        if (n == 1 || n == 2 || n == 3) {
                return n;
        }
        int[][] base = { { 1, 1, 0 }, { 0, 0, 1 }, { 1, 0, 0 } };
        int[][] res = matrixPower(base, n - 3);
        return 3 * res[0][0] + 2 * res[1][0] + res[2][0];
}
```

如果递归式严格符合 $F(n)=a\times F(n-1)+b\times F(n-2)+...+k\times F(n-i)$，那么它就是一个 i 阶的递推式，必然有与 $i\times i$ 的状态矩阵有关的矩阵乘法的表达。一律可以用加速矩阵乘法的动态规划将时间复杂度降为 $O(\log N)$。

矩阵的最小路径和

【题目】

给定一个矩阵 m，从左上角开始每次只能向右或者向下走，最后到达右下角的位置，路径上所有的数字累加起来就是路径和，返回所有的路径中最小的路径和。

【举例】

如果给定的 m 如下：

1　3　5　9