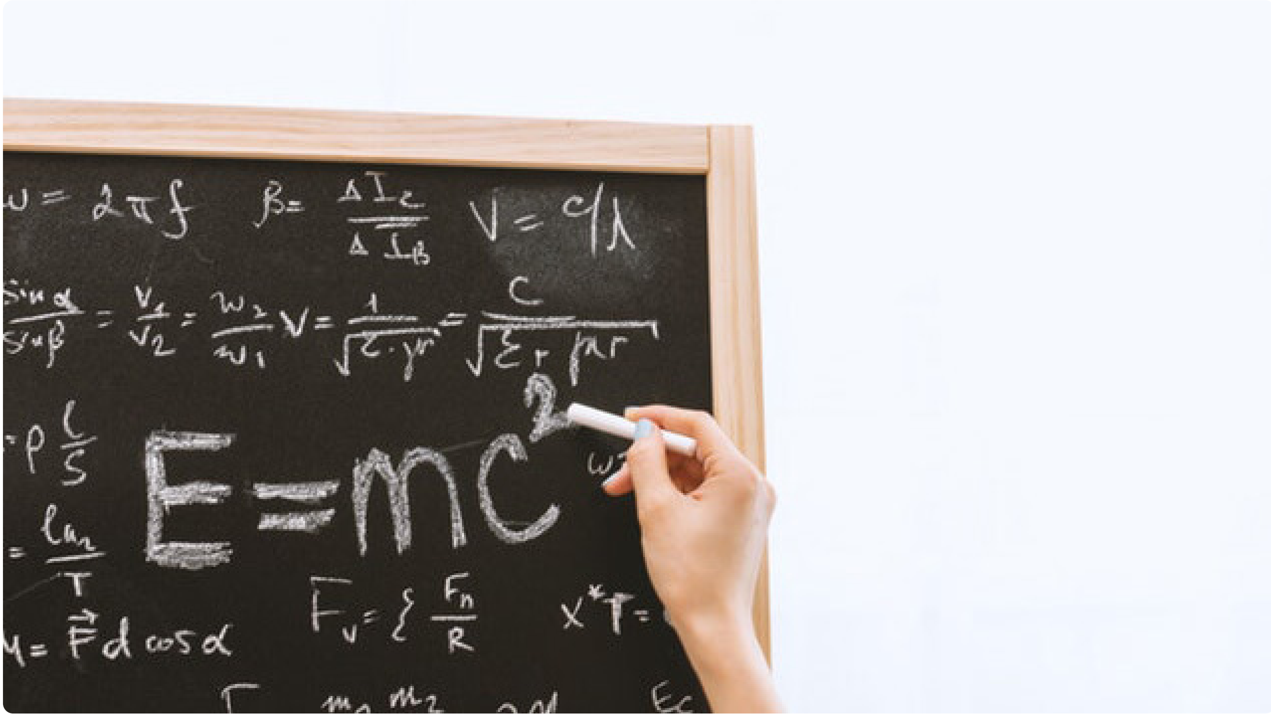
42 专题4：斐波那契数列

更新时间：2019-10-14 10:04:40



> 成功的奥秘在于目标的坚定。
>
> ——迪斯雷利

什么是斐波那契数列？

> 斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：1、1、2、3、5、8、13、21、34、……在数学上，斐波那契数列以如下被以递推的方法定义：F(1)=1，F(2)=1， F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）在现代物理、准晶体结构、化学等领域，斐波纳契数列都有直接的应用
>
> ----以上来自维基百科

上面的内容来自维基百科对于斐波那契数列的定义，大家在学校中应该都学习过斐波那契数列。在编程中斐波那契数列也是一道非常经典的面试题，下面我们来看几个问题来实践一下。

问题一

```
递推方程 f(n) = f(n - 1) + f(n - 2)，已知 f(1) = 1，f(2) = 1，求第n项 （n <= 30）
```

解析：

如果递归求这个问题，将会重复计算大量的状态，出现指数级的时间复杂度。

当然可以使用递推的方式来解决这个方程。这里来一步一步理解记忆化搜索。

理解函数

一个代码片段，将输入的数据通过计算转成输出的数据。函数同样可以表示成y=f(x)，x表示输入数据，也就是状态，y表示函数的输出结果。所有x组成的集合叫状态空间。类比于数学上的函数，函数应当有以下特点：
`唯一性：同一个状态x，函数值是唯一的`
这就要求我们尽可能将函数设计成跟环境无关的，只跟状态相关。

理解递归

很多人理解递归局限在代码调用代码中，于是很难理解递归什么时候跳出来，变量的值如何变。

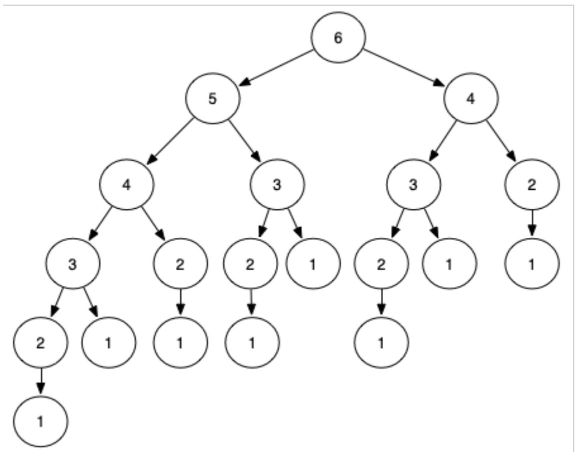
举个函数调用，上下文切换的例子：

假设说你要修电脑，先要用螺丝刀把机箱给打开，然后。。。
等等，螺丝刀呢？没有螺丝刀咋办，买呗，于是你就把电脑丢下，跑出去买了一个螺丝刀。
买回来之后，继续把机箱打开，发现内存条坏了，于是你就把电脑丢下，跑出去买了一个内存条（这里有个细节，机箱你是打开的，有两种做法，一种是打开着的机箱丢在地上，等买完内存条再回来继续，一种是把打开着的机箱装回去，等买完内存条再回来重新拆）。

根据这个例子，再想一下什么是递归。递归本身就是个多层函数调用，只不过函数的代码都一样。大家理解递归调用，其实可以理解成，当前函数调用了`其他`的函数。函数调用就是，计算父任务时，需要子任务一，于是代码中把父任务挂起，先执行子任务，等到子任务执行完之后，再把结果返回到父任务这里。

所以不管是递归，还是函数调用，都是有上下文的。每次调用，状态和环境变量都不一样。

基于上下文的理解，大家可以尝试一下把前中后序遍历的代码从递归改成，用一个栈模拟递归，可以加深对上下文的理解。



图为函数调用

记忆化搜索

看上图，是第6项斐波那契数列的调用图，可以看出，计算第6项需要计算第4项的结果，计算第5项也需要计算第4项的结果，为了避免重复计算，我们需要将计算第4项的结果缓存起来，以便重复使用。

递归过程中将函数的调用缓存起来，叫做记忆化搜索。

记忆化搜索有个条件：需要遵循函数与环境无关的原则，也是说，一个函数参数必须唯一对应一个返回值，否则不能缓存。

```
int cache[N]; //n如果有确切的范围，并且n比较小时，我们可以开个数组来存储函数值，存取函数值只有o(1)的复杂度
// map<int, int> cache; //如果没有确切的范围，或者n比较大时，我们需要利用映射数据结构来存储函数值，比如红黑树、哈希表
memset(cache, -1, sizeof(cache)); //-1表示cache还没计算过
int f(int n) {
    if (n <= 2) {
        return 1;
    }
    if (cache[n] != -1) {
        return cache[n];
    }
    return cache[n] = f(n - 1) + f(n - 2);
}
```

上面是个c++代码，如果用python写，那就更简单了，在函数上写个注解`@lru_cache(maxsize=N)`即可。

问题二

```
递推方程 f(n) = f(n - 1) + f(n - 2)，已知 f(1) = 1，f(2) = 1，求第n项 （n <= 80）
```

解析：

注意数据范围，n从30加到80。这个问题跟上一个问题解法一样，只是注意坑点，f(n)超过int的存储范围，因此需要使用64位整型，java为long，而python本身支持大整数计算，直接忽略这个坑点。

问题三

```
递推方程 f(n) = f(n - 1) + f(n - 2)，已知 f(1) = 1，f(2) = 1，求第n项 （n <= 10000），结果对10^9 + 7取模
```

解析：

注意数据范围，n从30加到10000。这个问题仍然跟上一个问题的解法一样，由于记忆化搜索复杂度是o(n)，对于10000范围内，妥妥的没问题。

问题四

```
递推方程 f(n) = f(n - 1) + f(n - 2)，已知 f(1) = 1，f(2) = 1，求第n项 （n <= 10^18），结果对10^9 + 7取模
```

解析：

n的数据范围超级大，达到10^18，已经无法用记忆化搜索或者递推来做，这就需要另一种做法。

快速幂算法

问题：求解

$$a^b \% k$$

的值，**b <= 10 ^ 18**。

循环b次，不断累乘a的做法肯定是行不通的，因为b特别大。

我们推导一下：

当b为偶数时，令b = 2c，

那么

$$a^b = a^{2c} = (a^c)^2$$

，我们只需要求

$$a^c$$

，就可以以o(1)的时间复杂度得到

$$a^b$$

当b为奇数时，令b = 2c + 1，

那么

$$a^b = a^{2c+1} = (a^c)^2 a$$

，我们同样只需要求

$$a^c$$

，就可以以o(1)的时间复杂度得到

$$a^b$$

不断地求解子问题，从而得到

$$a^b$$

的结果。由于子问题

$$a^c$$

的c是问题a^ b的b的一半，因此这个算法复杂度是o(log b)。

代码：

```
int f(int a, int b, int k) {
    if  (b == 1)  { return a % k; }
    //求解a ^ (b / 2)
    int ret = f(a, b >> 1, k);
    //求解 (a ^ (b / 2)) ^ 2
    ret = (long long) ret * ret % k;
    if (b & 1) {
        //处理b是奇数的情况，多乘了一次a
        ret = (long long) ret *a % k;
    }
    return ret;
}
```

斐波那契数列的快速幂算法

将递推方程转化一下：

$$\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \end{pmatrix}$$

大家应该能够理解矩阵的吧，矩阵上一行表示f(n) = f(n - 1) + f(n - 2)，下一行表示f(n-1) = f(n-1)。

那么我们将这个转化后的递推方程，转化成矩阵的幂：

$$\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} a_2 \\ a_1 \end{pmatrix}$$

套用上面介绍的快速幂算法，我们可以很快求出矩阵的n次幂。

至此斐波那契数列的4个问题，都解决：

```
Matrix mod(Matrix& A, int k);
Matrix multiply(Matrix& A, Matrix& B, int k);
int pow(Matrix& A, long long b, int k) {
    if  (b == 1)  { return mod(A, k); }
    Matrix ret = pow(a, b >> 1, k);
    ret = multiply(ret, ret, k);
    if (b & 1) {
        ret = multiply(ret, a, k);
    }
    return ret;
}
```

拓展 Fibonacci 的DP版本

对于DP的不同理解造成不同的写法。记忆化通常会增加你的时间复杂度，从而增加空间复杂度（例如，使用制表法，你可以自由地放弃计算，例如使用Fib的制表法可以使用O（1）空间，而使用Fib的制表法则可以使用O（N）堆栈空间）。

> 详细可以参考下面两份文档：
>
> Dynamic programming and memoization: bottom-up vs top-down approaches
>
> Tabulation vs Memoizatation

- top-down(memorize)：

```
def memorize_fib(n): # n为第几个Fibonacci数
    memo = {1:1, 2:1}
    if n in memo:
        return memo[n]
    else:
        memo[n] = memorize_fib(n-1) + memorize_fib(n-2)
        return memo[n]

print(memorize_fib(4)) # 输出3
```

- bottom up(tabulation)：

```
def tabulation_fib(n):  # n为第几个Fibonacci数
    fib = [1, 1, 2]
    if n < 4:
        return fib[n-1]
    for k in range(3, n+1):
        fib[2] = fib[0] + fib[1]
        fib[0], fib[1] = fib[1], fib[2]
    return fib[2]

print(tabulation_fib(4)) # 输出3
```

这里memo用dict，用array也一样。当然用bottom up还有一点，可以只存每次最后两个数，可以save space.，这样就只用到constant space，还可以直接借用工具。

```
import functools


@functools.lru_cache(maxsize=None)
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


print(fib(10)) # 55=
```

小结

这节课我们认识了下斐波那契数列，同学们可以自己动手实现一下，这个知识点在面试的时候还是很常见的。我在很多家公司的面试题中都有看到过，大家一定要掌握。

}

41 专题3：二分算法

43 专题5：大整数