1 记忆化搜索

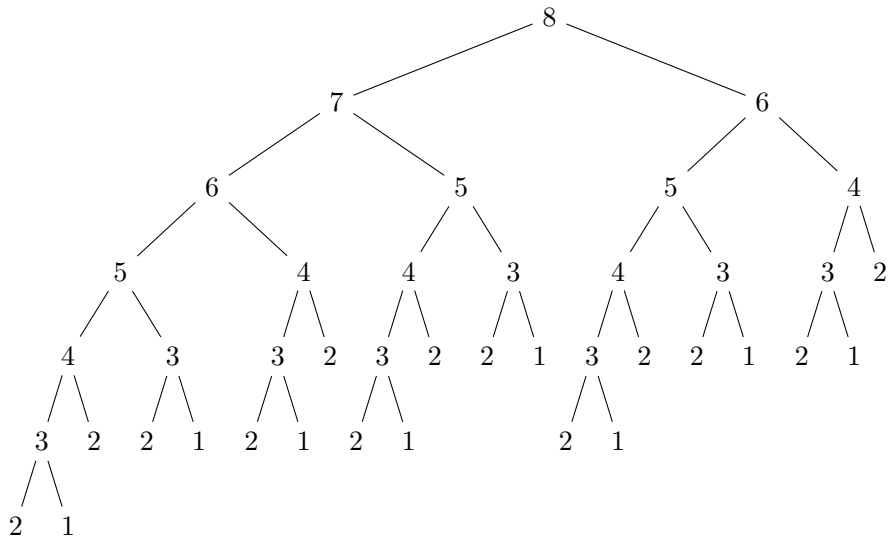
1.1 我们为什么需要记忆化搜索？

这一章我们讲记忆化搜索，记忆化搜索这个东西其实和普通的搜索没有太大的区别，是一个所谓“以空间换时间”的操作。

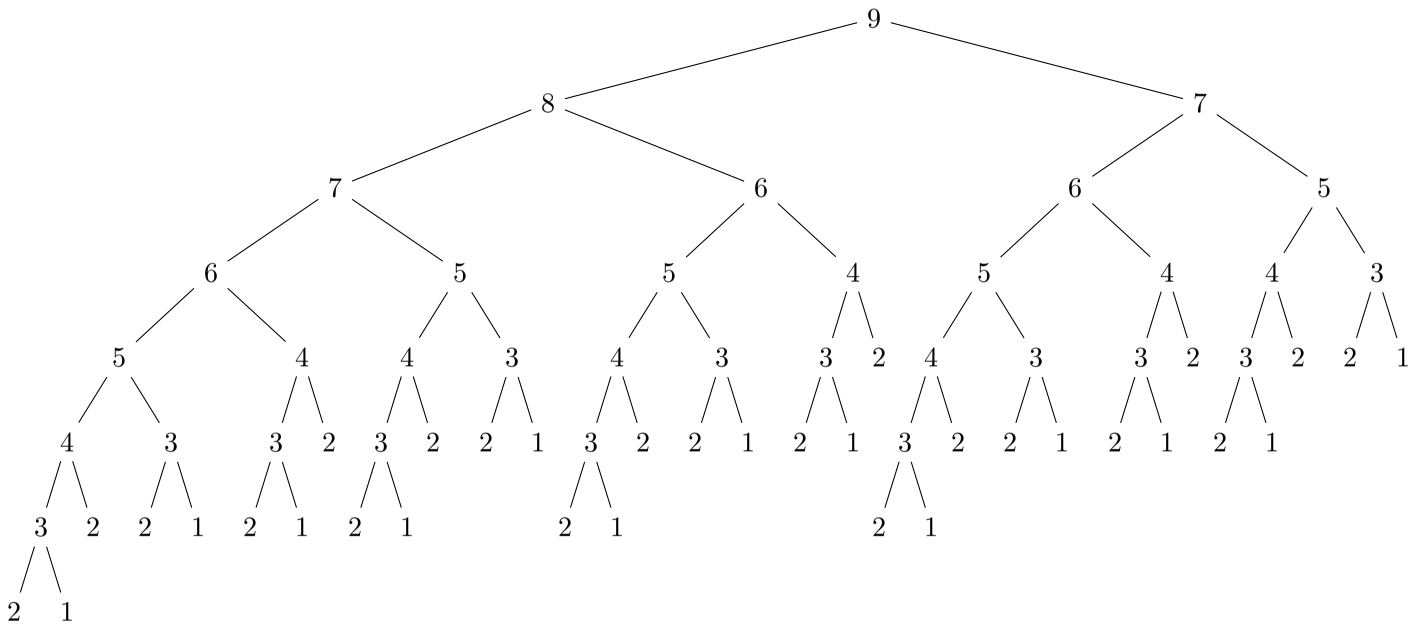
举个例子，我们要计算斐波那契数列的第 n 项。普通的递归可能会这么写：

```
int fib(int x)
{
    if(x==1||x==2)return 1;
    return fib(x-1)+fib(x-2);
}
```

下图是当 $n = 8$ 时递归树的情况，可以看到，在调用过程中出现了大量的重叠子问题，而此时我们的程序不得不进行重复递归调用，这消耗了大量的时间。



再来更加极端一点的例子，下页图显示了当 n 的值扩大 1 时，问题的规模就几乎扩大了一倍。这几乎是不可接受的。



$n = 9$ 时的递归树，可以看到重叠子问题更多了，问题规模几乎扩大了一倍。

n 的取值	递归调用次数
.....	.....
40	204668309
41	331160281
42	535828591
43	866988873
44	1402817465
45	2269806339
.....	.....

上表展示了当 n 的取值较大时递归调用次数的变化情况，经模拟测算可以发现增长率几乎是指数级别的增长，事实证明也如此。

这样的时间复杂度几乎是不可接受的。所以我们需要使用记忆化搜索这一手段来优化。记忆化搜索，又称为带备忘的搜索，顾名思义，就是把已经搜索过的结果记录下来，作为一个“备忘”，当程序需要再次调用这个搜索过程时就直接调用这个结果即可，不需要重复搜索了。

1.2 如何实现记忆化搜索？

还是以刚才的斐波那契数列为例。我们可以开一个数组 $f[]$，用于记录程序已经计算过的斐波那契数列的项的值。然后需要用的时候直接判断一下 f 数组里有没有这个元素（是否为初始值），如果有就直接返回这个值，没有再递归调用。代码如下：

```
int f[1000];//初始值为 0，因为斐波那契数列中不可能出现 0，所以 0是非法的。
int fib(int x)
{
    if(x==1||x==2)return 1;//递归边界条件
    if(f[x]!=0)return f[x];//如果已经计算过了就返回结果。
    return f[x]=fib(x-1)+fib(x-2);//如果还没有计算过就递归计算并保存结果。
}
```

下表展示了记忆化搜索递归调用的次数：

n 的取值	递归调用次数
……	……
40	3
41	3
42	3
43	3
44	3
45	3
……	……

可见，记忆化搜索能够帮我们减少递归的调用，提高代码的效率，以上代码即使在 n 非常大的情况下仍能较快的给出结果。

同时，不可忽略的一点是记忆化搜索需要有一个与状态等大的数组（当然你也可以用 `map` 或者`set` 之类的东西，只不过稍慢一些），空间占用很成问题。这也是记忆化搜索（包括 DP）的弊端所在。当然由于 DP 在某些情况下可以优化空间复杂度（压维之类的），所以存在一类题目，可以使得 DP 通过而记忆化搜索 MLE 或 TLE。

记忆化搜索通常适用于对于 DFS 的优化，对于 BFS 就不是那么实用了。当你在考场上写不出来 DP 但又非常确定这道题目是个 DP 的时候，不妨先打个暴力，然后转成记忆化搜索，可以帮助你拿到很多分数，还是非常实用的一种技巧。

下面给出普通搜索转记忆化搜索的模板：

```
int f[1000][1000];//与搜索的参数个数相同
void dfs(int a,int b)
{
    if(找到了这个问题的解) return ans;
    if(f[a][b]!=nil) return f[a][b];//这里的 nil 表示一个无效的值，可以认为是在正常情况下
    ↪  不可能出现的一个值
    //状态转移过程被省略一部分
    return f[a][b]=dfs(...,...);//这一句原来是 return dfs(...,...);
}
```