前言

算法，对于初级程序员(**Api Caller**) 而言,可能并不怎么重要，因为平时工作中压根用不到算法。但是要进入高级工程师，就要知道，如何用最优的计算方式来完成同一件任务。比如，腾讯面试经常问到的，给你**一个亿的用户数据，要你从中找到指定的用户信息，如何达到最快**，又或者，**手机QQ本地聊天记录中有1W条数据，如何最快找到包含搜索关键字的聊天记录**，这些都是直接影响到用户体验的功能，又比如，**滴滴打车app，如何进行最优的派单方案，让 所有的注册车辆都有订单**等等.诸如此类的问题，**Api Caller** 的程序员何以解决，拿不到高薪，是有自身的原因的，谁让你对高级算法一无所知。

算法，基于数学，应用于生活中的各种实际问题，它是一个巨大的知识体系，深不可测，仅以一文概论，不现实。本文详解的是，**动态规划算法**，一种解决问题的**通用套路**，学会了套路，相当于入了门，遇到任何问题都可以尝试套用框架,唯一不足的就是**深度和广度的不足，多训练就会有提升**，遇到再难的问题也不至于抓耳挠腮，不知所措。

所谓动态规划算法

以下是来自百度百科的概念，经我整理之后，浓缩如下：

动态规划(dynamic programming)是[运筹学](运筹学)的一个分支，是求解决策过程(decision process)最优化的数学方法. 提到数学，可能很多人都要头疼了，读大学的时候最怕的就是 高等数学，线性代数，解析几何这种。但是，实际上，动态规划算法，在现实生活中很多方面都已经应用到实践，比如，求地图上两个地点的最短路线，求资源的最合理分配方案，以及最优排序算法等问题上。按照动态规划的套路，基本上可以做到一套思维框架，放之四海皆准，每一种问题的之间的差别，都只是前提条件的不同，以及 最终目标的不同。

*能够用动态规划算法解决的问题，往往都会存在一个"**状态转移方程式**"。也就是，在多个不同的阶段，所采取的当前决策。把所有决策整合起来形成一个统一的数学方程式。*

？？？什么玩意？我们还是说点人话吧

动态规划算法，都有一个共同点，那就是求 "最值" ，像 最"短"路径，最"少"耗时，最"少"次数等。这种算法是一种普适性套路算法，它针对所有求最值得问题，都可以用一个统一的思维方式去解决。

状态转移方程，其实也就是一个所谓的 **数学函数公式**(能当程序员，数学应该是都不错的,你肯定知道我在说什么 ^-^！)，比如下文即将讲到的 **斐波拉契数列公式。**

写出**状态转移方程**之后，就可以开始用**程序代码**来解决实际问题。本文采用的编程语言是**Kotlin**,但是我都有些详尽的注释，就算不懂Kotlin，阅读起来应该也不会有太大障碍。

开始这一切之前，有一些算法的相关概念在此声明。

时间复杂度和空间复杂度

一个大的问题，往往可以分解成若干个子问题，大事化小，小事化了。

- **时间复杂度** = **子问题的个数** x **解决一个子问题所需的时间**
- **空间复杂度** = **子问题的个数** x **解决一个子问题所需的内存空间**

时间复杂度 描述算法的执行效率。常见的**时间复杂度**有**O(1),O($\log_2$n),O(n),O(n^2)**. **n**表示子问题的个数，分别表示**0次方，1/2次方，1次方，和2次方**。次数越高，复杂度越大, 解决同样的问题花费的时间就越长，算法的效率也就越低，谁都不希望自己的程序运行效率低。

空间复杂度 描述算法占用内存。越高，说明算法占用的内存空间越大。 它的写法和**时间复杂度**一样，次数越高，算法占用的空间也就越大，耗费内存就越大，无论是服务器上的程序算法，还是手机上的程序算法，都会考虑内存的问题。

时间和空间复杂度都只是描述算法的复杂程度，一般对比只看**大概数字级别**，而不是去**纠结实际的耗时**，比如3ms，6ms 级别上相差不大的基本也就是**同一个时间复杂度**。

递归回调

递归，也就是函数对自身的调用，一般都会设定一个退出递归的条件，防止无限回调。动态规划，大问题分解成小问题，一般都会用到递归调用。

初识动态规划：斐波拉契数列

著名的斐波拉契数列, f(1) = f(2) = 1,往后的f(n) 都是前面两个数的和，写成数学公式，也就是所谓"状态转移方程"，如下：

$$f(n) = \begin{cases} 1, n = 1, 2 \\ f(n-1) + f(n-2), n > 2 \end{cases}$$

如果我们要得到一个 **f(40)**，斐波拉契数列上位置**40**的函数值。我们可以完全按照数学公式来写代码。

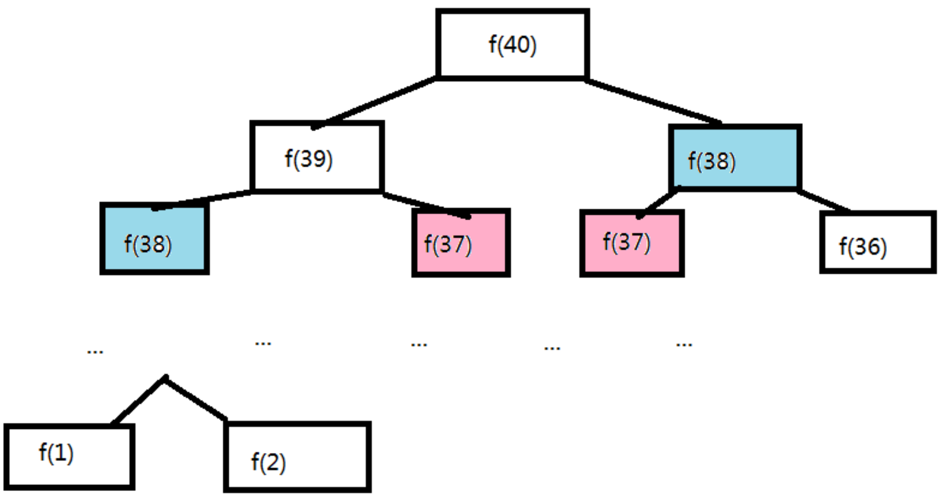
暴力解法

利用递归算法来编程就是如下写法：

```
fun fib1(x: Int): Long {
    return if (x == 1 || x == 2) {
        1
    } else
        fib1(x - 1) + fib1(x - 2)
}
```

如果用上面的算法来计算斐波拉契数列中某个节点的值，我们来算算**时间复杂度**是多少。

如果是**f(40)**,那么**递归树**结构就是：



时间复杂度

- 子问题的个数

  每个子问题都会分裂成2个子问题。所以子问题的个数是 2^n

- 一个子问题耗费的时间

  一个子问题只需要一次加法。所以，解决一个子问题的时间是1

合并起来，时间复杂度就是O(2^n) 指数级别的复杂度，效率极低。

空间复杂度（O(1)）这里不谈，因为都是临时变量，不存在永久保存的数据。

测试一下执行的效率:

```
fun main() {
    val start = System.currentTimeMillis()
    fib1(40)
    val end = System.currentTimeMillis()
    println("${(end - start)}ms")
}
```

结果（受系统的调度影响，多次运行结果可能不同，但是数量级应该是一样的）：

```
311ms
```

从上面的图可以看出，很多子问题都是重复计算的。比如**f(38),f(37)**...这就是上面的算法所带来的低效率问题，在做很多不必要的重复性工作。 那么有没有办法避免这种重复的递归计算呢？

一重改进

既然上面的**f(37),f(38)**等都在重复计算，那么我们用一个备忘录map，记录已经计算出的结果。

比如把上面的**f(38)**的值用map记录下来，下次需要计算的时候，**直接取值**,而不是再去计算

程序变成这样：

```
fun fib2(x: Int): Long {
    if (x < 1) return 0
    val map = HashMap<Int, Long>()
    return helper(map, x)
}

fun helper(map: HashMap<Int, Long>, x: Int): Long {
    if (x == 1 || x == 2) return 1
    if (map[x] != null && map[x] != 0L) //如果已经计算过了，那就是说保存过了，就直接返
回
        return map[x]!!
    // 否则，就计算之后再保存
    map[x] = helper(map, x - 1) + helper(map, x - 2) // 这里依然是递归
    return map[x]!!
}

fun main() {
    val start = System.currentTimeMillis()
    fib2(40)
    println("${System.currentTimeMillis() - start}ms")
}
```

运行结果(受系统的调度影响，多次运行结果可能不同，但是数量级应该是一样的)：

```
3ms
```

执行耗时从 **三位数**变成了**1位数**

这种解法的**时间复杂度**该如何计算？

- 子问题的个数

  上面的做法，显然就是f(38)等这种重复性计算不再存在，所以可以理解为：当需要f(38)的时候，会优先从 map中去取。所以，计算出f(40)，也就最多有40个子问题。所以 f(n)子问题的个数是 **n**,

- 一个子问题耗费的时间

  依然只需要一次加法就能算出一个子问题的结果,所以耗时 **1**

所以**时间复杂度**是 **O(n)**，对比 前面一种解法的$O(2^n)$简直就是降维打击。

而**空间复杂度**：

由于这里使用了 map集合来保存已经计算出来的值，所以，空间复杂度：

- 子问题的个数

  f(n)子问题的个数还是n

- 一个子问题耗费的空间

  每一个子问题的计算结果都会占用1个空间来保存, 所以一个子问题耗费空间1

所以**空间复杂度**= **O(n)**

时间复杂度降低了，程序的**运行效率**提高了，但是，**空间复杂度**却升高了，这就是所谓的**"空间换时间"**.

二重改进

不知道有没有人跟我有一样的感觉！那就是，这种递归算法， 自上而下的思维方式有点反人类，容易把人绕晕。如果可以的话，我宁愿顺着正常人的思维去思考，比如，自下而上，先得出 f(1),f(2),再得出f(3),f(4)...一直到我们要求的f(40).

程序写成下面这样:

```kotlin
fun fib3(x: Int): Long {

    val map = HashMap<Int, Long>() // 初始化一个map

    map[1] = 1
    map[2] = 1

    for (i in 3..x) {
        val pre1 = map[i - 1] ?: 0
        val pre2 = map[i - 2] ?: 0
        map[i] = pre1 + pre2
    }
    return map[x] ?: 0
}

fun main() {
    val start = System.currentTimeMillis()
    fib3(40)
```

```
    println("${System.currentTimeMillis() - start}ms")
}
```

时间复杂度：

- 子问题的个数

  只是思考的方向反了，子问题仍然是n个

- 一个子问题耗费的空间

  处理一个子问题的，仍然是一次加法，时间依然是1

所以时间复杂度依然是：**O(n)**

空间复杂度:

- 子问题的个数

  n

- 处理一个子问题，需要保存一个数据，所以占空间是1

空间复杂度为：**O(n)**

这种算法，把自上而下的递归，变成了自下而上的遍历，时间和空间复杂度都没有变化，但是写法更容易理解，执行效率也如预料中一样,并没有变化:

```
4ms
```

三重改进

上面一种改进方法，我们用map保存了已经计算出来的值，自上而下递归计算，和自下而上遍历计算，时间和空间复杂度都相同。但是，自下而上的计算依然有优化空间。

思考:**是否可以不需要map来保存数据**？

上一节的改进，f(n)依赖的仅仅是f(n-1)和f(n-2). 而不需要另外的数据一直保存。

```
/**
 * 最后优化
 */
fun fib4(x: Int): Long {
    if (x == 1 || x == 2) return 1L
    var cur = 1L
    var pre = 1L
    for (i in 3..x) {
        val sum = cur + pre
        pre = cur
        cur = sum
    }
    return cur
}

fun main() {
    val start = System.currentTimeMillis()
    fib4(40)
    println("${System.currentTimeMillis() - start}ms")
}
```

时间复杂度：

- 子问题的个数

  仍然是 n

- 一个子问题耗费的时间

  一个子问题只需要1次加法，所以耗时 1

所以时间复杂度是**O(n)**

空间复杂度：

- 子问题的个数是n
- 一个子问题所占用的空间是1，但是 都是临时占用，函数执行完毕之后就会释放，算法中的临时变量所占空间并不会累加，

所以空间复杂度，是**O(1)**

运行结果,时间耗费上依然没变：

```
3ms
```

至此，斐波拉契数列问题才算圆满解决，时间和空间复杂度都达到了最优。

总结

动态规划算法，解决问题的一般思路为：

- **写出状态转移方程式**
- 直接按照方程式写出**暴力解法**程序，可能效率会很低
- 使用集合保存已经计算出的子问题的结果，**优化子问题的个数**，得出**优化算法**优化**时间复杂度**
- 使用**顺序遍历**的思路来替代**逆序递归**，让程序代码更好理解
- 尝试能否使用**临时变量**替代**集合**，优化**空间复杂度**

但是，聪明的你们应该已经发现了好像哪里不对。

没错，最前面说过，动态规划算法，是用来解决最值问题的算法， 目的是得出最优解。 显然，斐波拉契数列并不是求最值。从这里可以看出，这种解法思路，不仅仅针对求最值问题，有一些非此类问题，也可以尝试使用动态规划去思考。

本文使用**斐波拉契数列**，是因为它的**状态转移方程**是 公开的，大家入门都很容易理解。但是现实生活中的实际问题，它的状态转移方程，要根据实际情况，分情况列出，如果方程列错了，就算算法再精良，结果也不是我们想要的。

所以，使用动态规划解决问题，还是万事开头难。只要列出了正确的**状态转移方程**，后续，我们才可以按照套路来，一步一步优化算法。 所以千万不要瞧不起 效率最低的**暴力解法**，它代表的是最原始的也是，**最标准的解法。**计算结果出现问题，我们首先要审视的就是 **状态转移方程**是否正确，方程正确，才能去查后面的问题。

下一篇将以一个实际案例(**凑零钱问题**)，来一步步展示动态规划算法的实际应用。