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。

承接上文

凑零钱问题，在零钱无限多的情况下，用**"动态规划"**给出了最优解算法。 本文代码扔采用简洁优美的**kotlin**，可读性高。

计算出零钱的组合

上面凑零钱算法只是告诉我们如何得出最小的零钱数目，但是并没有告诉我们零钱是如何组成的。

那么**下一个目标**，我不仅仅想知道零钱需要**多少张**，我还要知道零钱的**组合方式**。

比如：**18块钱总额**，打散成最少数目的零钱，**需要3张5块的**，加上**一张1块**, **1张2块**的，一共**5张**零钱。

```
package com.example.myapplication

val change = arrayOf(1,2,5) // 零钱就是1,2,5, 零钱必须是有序的

fun changeMoney(lastTarget: Int): List<Int>? {
    change.sortDescending()// 我需要从大到小来遍历，所以必须先排序
    // 所以做法就是，从1开始往后推演，从树根往树顶推算
    val hashMap = HashMap<Int, List<Int>>()
    for (currentTarget in 1..lastTarget) {
        getMin(currentTarget, hashMap)
    }
    return if (hashMap[lastTarget]?.sum() == lastTarget) hashMap[lastTarget]
else null
}

fun getMin(currentTarget: Int, map: HashMap<Int, List<Int>>): List<Int> {
    if (map.containsKey(currentTarget) && map[currentTarget]?.isNotEmpty()!!) {
        return map[currentTarget] ?: arrayListOf()
    }

    loop@ for (c in change) {
```

```
        return when {
            currentTarget - c == 0 -> {
                val list = arrayListOf(c)
                map[currentTarget] = list // 目标金额等于当前零钱，返回 1，只需要1张零钱就行
                list
            }
            currentTarget - c > 0 -> {
                val min = getMin(currentTarget - c, map)// 目标金额大于当前零钱，说明需要当前零钱一张之后，剩下的钱要进入下一轮循环

                // 建立一个可变数组，先保存下一轮循环的结果，然后创建不可变数组赋值给总map
                val temp = mutableListOf<Int>()
                temp.addAll(min)
                temp.add(c)

                map[currentTarget] = temp// 保存起来
                min
            }
            else -> {
                continue@loop
            }
        }
    }
    return arrayListOf()
}

/**
 * 我们来把最终结果打印得好看一点，这个不属于算法，只是为了打印好看
 */
fun printRes(changeMoney3: List<Int>?) {
    if (changeMoney3 == null) {
        println("================无法凑成目标金额================")
        return
    }
    // 太难看了，相同的金额应该要累计起来，最后的展现形式应该是 5*21 , 2*2
    val beautifulMap = HashMap<Int, Int>()
    for (c in changeMoney3!!) {// c是当前金额
        if (beautifulMap.keys.contains(c)) {
            beautifulMap[c] = beautifulMap[c]!!.plus(1)
        } else {
            beautifulMap[c] = 1
        }
    }

    println("================需要================")
    // 遍历map
    for (i in beautifulMap) {
        println("${i.key}元零钱: ${i.value}张")
    }
}

fun main() {
    val targetAmount = 18
    println("目标金额是 : $targetAmount")
    val start = System.currentTimeMillis()
    val changeMoney3 = changeMoney(targetAmount)
    printRes(changeMoney3)
```

```
}
```

运行结果：

```
目标金额是 ： 18
================需要=================
1元零钱： 1张
2元零钱： 1张
5元零钱： 3张
```

时间复杂度：

- 子问题的个数

  由于有map在记录当前值，所以如果目标金额是18，也最多就需要计算18次，个数：n

- 一个子问题所花费的时间

  一个子问题最多进行一次减法运算（当前金额减去当前零钱），以及一次 加法运算（结果list的合并），时间：2

所以时间复杂度 ： **O(2n)**

空间复杂度：

- 子问题的个数

  n

- 一个子问题所花费的空间

  每一个子问题最多需要临时创建一个list来保存当前结果，所以空间 1

空间复杂度 ： **O(n)**

零钱数量有限时求零钱组合

其实还可以再扩展，上面的凑零钱，都是在零钱无限多的情况下。但是，现实中零钱不可能无限多。假如：

1元零钱只有23张，2元只有12张，5元只有11张。那么，我要凑成78块钱，最少需要多少张零钱，我要凑成87块钱，需要多少张零钱。

新增难点

之前零钱无限多的时候，由于是1元的存在，任何金额都可以凑出来，但是当零钱有限的时候，还能凑出来么???

不一定了. 比如1元的是0张，只有2元的，5元的也是0张，让你凑11元总额，是凑不出来的。所以，现实凑零钱问题，还必须考虑哪些情况下凑不出目标金额。

解法如下：

```
package com.example.myapplication

import java.util.Comparator
```

```
/**
 * 这里表示，1元的有11张，2元的有22张，5元的有11张
 */
var changeCountMap = mapOf<Int, Int>(Pair(1, 11), Pair(2, 22), Pair(5, 11))

// ******************************** 核心函数
*****************************************
fun changeMoney(lastTarget: Int): List<Int>? {
    changeCountMap =// 按照key从大到小的顺序排序
        changeCountMap.toSortedMap(Comparator<Int> { o1, o2 -> o2!! - o1!! })

    // 判断零钱可以提供的总额最大是多少
    var max = 0
    for (key in changeCountMap.keys) {
        max += key * changeCountMap.getValue(key)
    }
    println("可以凑成的最大金额是:$max")
    if (lastTarget > max) {
        return null
    }

    val hashMap = HashMap<Int, List<Int>>()
    for (currentTarget in 1..lastTarget) {
        getMin(currentTarget, hashMap)
    }
    return if (hashMap[lastTarget]?.sum() == lastTarget) hashMap[lastTarget]
else null
}


fun getMin(currentTarget: Int, map: HashMap<Int, List<Int>>): List<Int> {
    if (map.containsKey(currentTarget) && map[currentTarget]?.isNotEmpty()!!) {
        return map[currentTarget]!!
    }
    loop@ for (c in changeCountMap.keys) {
        val max = changeCountMap.getValue(c)
        return when {
            currentTarget - c == 0 -> {
                val list = listOf(c)
                map[currentTarget] = list
                // 上面是假设零钱无限多，但是如果要考虑零钱数量优先
                if (getCount(c, map) <= max && max > 0) {// 数量满足，照旧
                    list
                } else {// 数量不满足
                    map[currentTarget] = listOf()// 回退
                    continue@loop
                }
            }
            currentTarget - c > 0 -> {
                val min = getMin(currentTarget - c, map)
                val temp = mutableListOf<Int>()
                temp.addAll(min)
                temp.add(c)
                map[currentTarget] = temp// 保存起来
                if (getCount(c, map) <= max && max > 0) {// 数量满足，照旧
                    temp
                } else {// 数量不满足
```

```
                    map[currentTarget] = listOf()//要回退
                    continue@loop
                }
            }
            else -> {
                continue@loop
            }
        }
    }
    return listOf()
}

// ********************************* 辅助函数
*****************************************
/**
 * 要做一个函数，计算出当前结果List<Int>中有，指定的金额有多少张，这样才能做出张数限制
 */
fun getCount(changeAmount: Int, map: HashMap<Int, List<Int>>): Int {
    val res = map[map.keys.max()]// 拿到最大的key
    var count = 0
    if (res == null || res.isEmpty()) return 0 // 最大的key没有value，直接返回0
    for (i in res) {// 否则，针对指定金额进行遍历累加计算
        if (i == changeAmount) {
            count++
        }
    }
    return count
}

/**
 * 我们来把最终结果打印得好看一点，这个不属于算法，只是为了打印好看
 */
fun printRes(changeMoney3: List<Int>?) {
    if (changeMoney3 == null || changeMoney3.isEmpty()) {
        println("================无法凑成目标金额================")
        return
    }
    // 太难看了，相同的金额应该要累计起来，最后的展现形式应该是 5*21 , 2*2
    val beautifulMap = HashMap<Int, Int>()
    for (c in changeMoney3) {// c是当前金额
        if (beautifulMap.keys.contains(c)) {
            beautifulMap[c] = beautifulMap[c]!!.plus(1)
        } else {
            beautifulMap[c] = 1
        }
    }

    println("================需要================")
    // 遍历map
    for (i in beautifulMap) {
        println("${i.key}元零钱: ${i.value}张")
    }
}

fun main() {
    val targetAmount = 87
    println("目标金额是 : $targetAmount")
    val start = System.currentTimeMillis()
```

```
    val changeMoney3 = changeMoney(targetAmount)
    println("计算耗时:${System.currentTimeMillis() - start}ms")
    printRes(changeMoney3)
}
```

运行结果：

```
目标金额是 ： 87
可以凑成的最大金额是:110
计算耗时:14ms
================需要=================
2元零钱: 16张
5元零钱: 11张
```

时间复杂度和**空间复杂度**:

- 子问题的个数

  依然没有变化，依然是**n**

- 一个子问题消耗的时间

  与上一章节相比并没有变化，还是只有一次减法和一次数组合并，所以是2

- 一个子问题消耗的空间

  还是在用一个map来保存每一次遍历的目标list一维数组，所以消耗空间1

时间复杂度 **O(2n)**, 空间复杂度：**O(n)**

有没有更简单的办法？

凑零钱问题，必须要动态规划法吗？其实也不一定，其实还有更简单的算法 ！没错，**整数取余**法，说人话:

给你107的目标金额，零钱有1元，2元，5元，10元，数目不限。要追求最少的零钱数目。

那我肯定先用最大额的零钱来凑整，凑到它不能满足余下金额之后，再用较小钞票来做循环。直到最小金额。

换成代码：

```
// 其实解决此类问题还有一个整除取余法
val changes = arrayOf(1, 2, 5, 10)
fun changeMoney2(target: Int): Map<Int, Int> {
    changes.sortDescending()// 从大到小排
    var temp = target
    val map = HashMap<Int, Int>()
    for (c in changes) {
        val v1 = temp / c // 整除的值，表示当前钞票的张数
        map[c] = v1
        temp %= c // 取余的值，表示当前钞票过大之后造成的余数
    }
    return map
}

fun readMap(map: Map<Int, Int>) {
    println("================需要=================")
    // 遍历map
```

```
    for (i in map) {
        println("${i.key}元零钱: ${i.value}张")
    }
}

fun main() {
    val target = 107
    println("目标金额:$target")
    val changeMoney2 = changeMoney2(target)
    readMap(changeMoney2)
}
```

运行结果：

```
目标金额:107
================需要=================
1元零钱: 0张
2元零钱: 1张
5元零钱: 1张
10元零钱: 10张
```

时间复杂度：

- 子事件的个数

  子事件的个数已经和目标金额没有直接关系了，反而是和零钱数量有关，而零钱数量是固定的那么几个，所以事件数目认定为 1

- 解决一个子事件所需时间

  需要一次除法，以及一次取余，也不存在循环和递归，所以时间认为是1

所以时间复杂度：**O(1)**

空间复杂度:

- 子事件的个数

  1

- 解决一个子事件所需空间

  上面我只是用map来记录零钱的张数，并没有参与循环递归的空间开辟，所以 空间 1

空间复杂度也是: **O(1).**

就算考虑零钱有限的情况，整除取余法也有解：

```
// 其实解决此类问题还有一个整除取余法
var changeCountMap = mapOf(Pair(1, 12), Pair(2, 13), Pair(5, 21))

fun changeMoney2(target: Int): Map<Int, Int>? {

    // 依然要判定能够凑出的最大金额
    // 判断零钱可以提供的总额最大是多少
    var max = 0
    for (key in changeCountMap.keys) {
        max += key * changeCountMap.getValue(key)
    }
    println("可以凑成的最大金额是:$max")
    if (target > max) {
```

```
        return null
    }

    changeCountMap =// 按照key从大到小的顺序排序
        changeCountMap.toSortedMap(Comparator<Int> { o1, o2 -> o2!! - o1!! })
    var temp = target
    val map = HashMap<Int, Int>()
    for (c in changeCountMap.keys) {
        val v1 = temp / c // 整除的值，表示当前钞票的张数
        val max = changeCountMap.getValue(c)
        if (v1 > max) {
            map[c] = max // 当前最大值先用完
            temp -= max * c // 剩下的进入下一轮
        } else {
            map[c] = v1
            temp %= c // 取余的值，表示当前钞票过大之后造成的余数
        }

    }

    // 计算出总金额
    var total = 0
    for (key in map.keys) {
        total += key * map[key]!!
    }

    return if (total == target) map else null
}

fun readMap(map: Map<Int, Int>?) {
    if (map == null) {
        println("================无法凑出该金额================")
        return
    }
    println("================需要================")
    // 遍历map
    for (i in map) {
        println("${i.key}元零钱: ${i.value}张")
    }
}

fun main() {
    val target = 101
    println("目标金额:$target")
    val changeMoney2 = changeMoney2(target)
    readMap(changeMoney2)
}
```

运行结果：

```
目标金额:101
可以凑成的最大金额是:143
================需要================
1元零钱: 1张
2元零钱: 0张
5元零钱: 20张
```

如果要凑的是144元,那么运行结果：

```
目标金额:144
可以凑成的最大金额是:143
================无法凑出该金额=================
```

很明显，这种方式更加简洁容易理解,复杂度还更低。

所以说：

为什么我不一开始就用这种简单解法算了？

因为： **解决凑零钱问题有多种方式，但是动态规划算法是一种通用的解题思路，先掌握算法的核心思想，进而可以在遇到其他问题的时候套用思路，举一反三。而 凑零钱的整除取余法，离开了凑零钱问题，可能就不适用了，比如 斐波拉契数列问题，就无法 用这种方法。**

总结

本系列前3篇，从动态规划的基本套路，到现实问题的应用实践，一步一步展示了当问题复杂化时我们算法的变化。其实解决复杂问题的基本套路都是如此，先基于一个简单问题，逐步复杂化，直到达到现实中的复杂问题。

人脑和电脑的区别，是人脑会有逻辑思维，知道用方便快捷省时省力的方式去解决问题，但是电脑做不到，它只能按照人类所想去尽可能的穷举所有的情况，对比之后得出最优解。我们利用计算机程序算法去解决问题，要利用电脑的高效率穷举运算，还要利用人脑的优势，让计算机按照我们的想法去走捷径，执行最优算法，达到最快，最省。

了解了这些，算法还是刚刚入门。复杂的算法问题，如大厂的面试题二分法快速查找数据等.

动态规划算法，最难的，还是写出"状态转移方程", 凑零钱问题的方程算是比较简单的，下一篇，就一个比较复杂的问题(**最长递增子序列问题**)，仍然使用动态规划算法来解决。