05讲递归（上）：泛化数学归纳，如何将复杂问题简单化

你好，我是黄申。上一节的结尾，我们用递归模拟了数学归纳法的证明。同时，我也留下了一个问题：**既然递归的函数值返回过程和基于循环的迭代法一致，我们直接用迭代法不就好了，为什么还要用递归的数学思想和编程方法呢**？这是因为，在某些场景下，递归的解法比基于循环的迭代法更容易实现。这是为什么呢？我们继续来看舍罕王赏麦的故事。

如何在限定总和的情况下，求所有可能的加和方式？

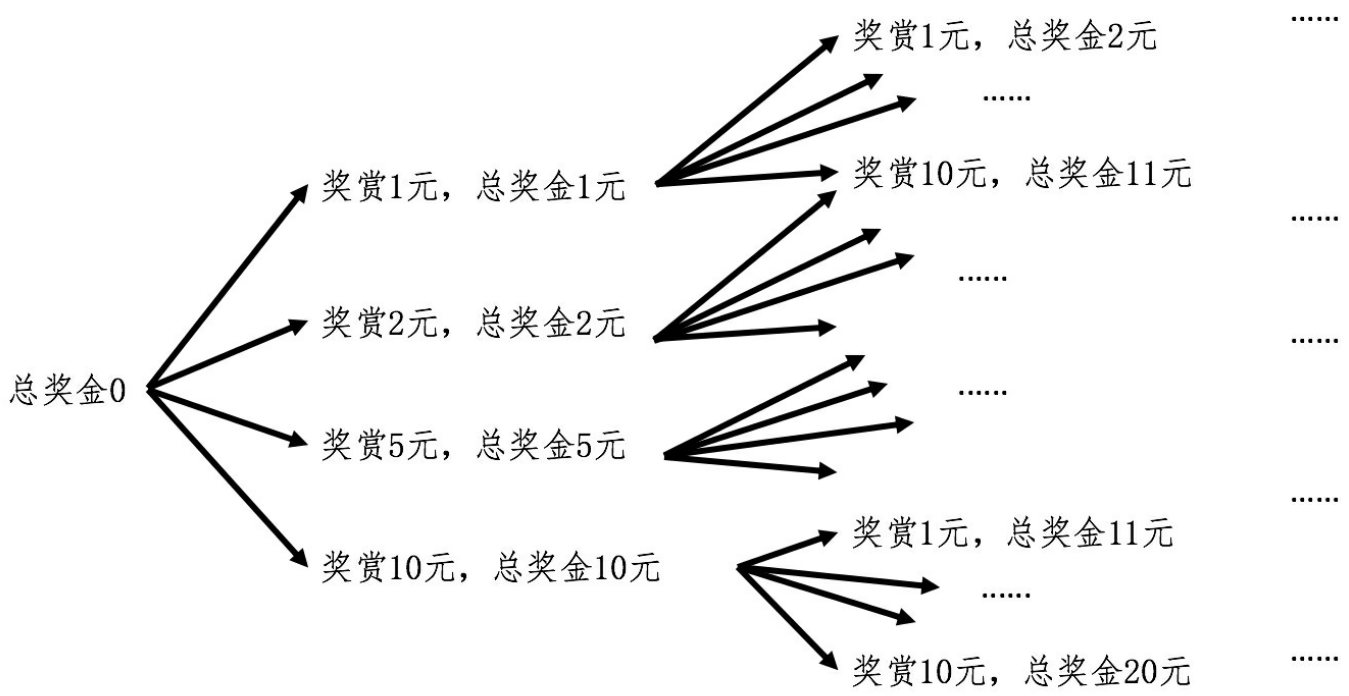
舍罕王和他的宰相西萨·班·达依尔现在来到了当代。这次国王学乖了，他对宰相说：“这次我不用麦子奖赏你了，我直接给你货币。另外，我也不用棋盘了，我直接给你一个固定数额的奖赏。”

宰相思考了一下，回答道：“没问题，陛下，就按照您的意愿。不过，我有个小小的要求。那就是您能否列出所有可能的奖赏方式，让我自己来选呢？假设有四种面额的钱币，1元、2元、5元和10元，而您一共给我10元，那您可以奖赏我1张10元，或者10张1元，或者5张1元外加1张5元等等。如果考虑每次奖赏的金额和先后顺序，那么最终一共有多少种不同的奖赏方式呢？”

让我们再次帮国王想想，如何解决这个难题吧。这个问题和之前的棋盘上放麦粒有所不同，它并不是要求你给出最终的总数，而是**在限定总和的情况下，求所有可能的加和方式。**你可能会想，虽然问题不一样，但是求和的重复性操作仍然是一样的，因此是否可以使用迭代法？好，让我们用迭代法来试一下。

我还是使用迭代法中的术语，考虑k=1,2,3,…,n的情况。在第一步，也就是当n=1的时候，我们可以取四种面额中的任何一种，那么当前的奖赏就是1元、2元、5元和10元。当n=2的时候，奖赏的总和就有很多可能性了。如果第一次奖赏了1元，那么第二次有可能取1、2、5元三种面额（如果取10，总数超过了10元，因此不可能）。

所以，在第一次奖赏1元，第二次奖赏1元后，总和为2元；第一次奖赏1元，第二次奖赏2元后，总和为3元；第一次奖赏1元，第二次奖赏5元后，总和为6元。好吧，这还没有考虑第一次奖赏2元和5元的情况。我来画个图，从图中你就能发现这种可能的情况在快速地“膨胀”。



你应该能看到，虽然迭代法的思想是可行的，但是如果用循环来实现，恐怕要保存好多中间状态及其对应的变量。说到这里，你是不是很容易就想到计算编程常用的**函数递归**?

在递归中，每次嵌套调用都会让函数体生成自己的局部变量，正好可以用来保存不同状态下的数值，为我们省去了大量中间变量的操作，极大地方便了设计和编程。

不过，这里又有新的问题了。之前用递归模拟数学归纳法还是非常直观的。可是，这里不是要计算一个最终的数值，而是要列举出所有的可能性。那应该如何使用递归来解决呢？上一节，我只是用递归编程体现了数学归纳法的思想，但是如果我们把这个思想泛化一下，那么递归就会有更多、更广阔的应用场景。

如何把复杂的问题简单化?

首先，我们来看，**如何将数学归纳法的思想泛化成更一般的情况**？数学归纳法考虑了两种情况：

1. 初始状态，也就是n=1的时候，命题是否成立；
2. 如果n=k-1的时候，命题成立。那么只要证明n=k的时候，命题也成立。其中k为大于1的自然数。

将上述两点顺序更换一下，再抽象化一下，我写出了这样的递推关系：

1. 假设n=k-1的时候，问题已经解决（或者已经找到解）。那么只要求解n=k的时候，问题如何解决（或者解是多少）；
2. 初始状态，就是n=1的时候，问题如何解决（或者解是多少）。

我认为这种思想就是将**复杂的问题，每次都解决一点点，并将剩下的任务转化成为更简单的问题等待下次求解，如此反复，直到最简单的形式**。回到开头的例子，我们再将这种思想具体化。

1. 假设n=k-1的时候，我们已经知道如何去求所有奖赏的组合。那么只要求解n=k的时候，会有哪些金额的选择，以及每种选择后还剩下多少奖金需要支付就可以了。
2. 初始状态，就是n=1的时候，会有多少种奖赏。

有了这个思路，就不难写出这个问题的递归实现。我这里列一个基本的实现。

```
import java.util.ArrayList;

public class Lesson5_1 {

 public static long[] rewards = {1, 2, 5, 10}; // 四种面额的纸币

 /**
    * @Description: 使用函数的递归（嵌套）调用，找出所有可能的奖赏组合
    * @param totalReward-奖赏总金额，result-保存当前的解
    * @return void
    */

    public static void get(long totalReward, ArrayList<Long> result) {

     // 当totalReward = 0时，证明它是满足条件的解，结束嵌套调用，输出解
     if (totalReward == 0) {
      System.out.println(result);
      return;
     }
     // 当totalReward < 0时，证明它不是满足条件的解，不输出
     else if (totalReward < 0) {
      return;
     } else {
      for (int i = 0; i < rewards.length; i++) {
       ArrayList<Long> newResult = (ArrayList<Long>)(result.clone()); // 由于有4种情况，需要clone当前的解并传入被调用
       newResult.add(rewards[i]);      // 记录当前的选择，解决一点问题
       get(totalReward - rewards[i], newResult);  // 剩下的问题，留给嵌套调用去解决
      }
     }

    }

}
```

我们测试一下总金额为10元的时候，有多少种解。

```
public static void main(String[] args) {

 int totalReward = 10;
 Lesson5_1.get(totalReward, new ArrayList<Long>());

}
```

最终，程序运行后大致是这种结果：

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 1, 1, 2, 1]
[1, 1, 1, 1, 1, 1, 2, 1, 1]
[1, 1, 1, 1, 1, 1, 2, 2]
...
[5, 5]
[10]
```

这里面每一行都是一种可能。例如第一行表示分10次奖赏，每次1元；第二行表示分9次奖赏，最后一次是2元；以此类推。最终结果的数量还是挺多的，一共有129种可能。试想一下，如果总金额为100万的话，会有多少种可能啊！

这个代码还有几点需要留意的地方，我再来解释一下：

1.由于一共只有4种金额的纸币，所以无论是n=1的时候还是n=k的时候，我们只需要关心这4种金额对组合产生的影响，而中间状态和变量的记录和跟踪这些繁琐的事情都由函数的递归调用负责。

2.这个案例的限制条件不再是64个棋格，而是奖赏的总金额，因此判断嵌套调用是否结束的条件其实不是次数k，而是总金额。这个金额确保了递归不会陷入死循环。

3.我这里从奖赏的总金额开始，每次嵌套调用的时候减去一张纸币的金额，直到所剩的金额为0或者少于0，然后结束嵌套调用，开始返回结果值。当然，你也可以反向操作，从金额0开始，每次嵌套调用的时候增加一张纸币的金额，直到累计的金额达到或超过总金额。

小结

递归和循环其实都是迭代法的实现，而且在某些场合下，它们的实现是可以相互转化的。但是，对于某些应用场景，递归确很难被循环取代。我觉得主要有两点原因：

第一，递归的核心思想和数学归纳法类似，并更具有广泛性。这两者的类似之处体现在：**将当前的问题化解为两部分：一个当前所采取的步骤和另一个更简单的问题。**

1.一个当前所采取的步骤。这种步骤可能是进行一次运算（例如每个棋格里的麦粒数是前一格的两倍），或者做一个选择（例如选择不同面额的纸币），或者是不同类型操作的结合（例如今天讲的赏金的案例）等等。

2.另一个更简单的问题。经过上述步骤之后，问题就会变得更加简单一点。这里“简单一点”，指运算的结果离目标值更近（例如赏金的总额），或者是完成了更多的选择（例如纸币的选择）。而“更简单的问题”，又可以通过嵌套调用，进一步简化和求

解，直至达到结束条件。

我们只需要保证递归编程能够体现这种将复杂问题逐步简化的思想，那么它就能帮助我们解决很多类似的问题。

第二，递归会使用计算机的函数嵌套调用。而函数的调用本身，就可以保存很多中间状态和变量值，因此极大的方便了编程的处理。

正是如此，递归在计算机编程领域中有着广泛的应用，而不仅仅局限在求和等运算操作上。在下一节中，我将介绍如何使用递归的思想，进行“分而治之”的处理。

思考题

一个整数可以被分解为多个整数的乘积，例如，6可以分解为2x3。请使用递归编程的方法，为给定的整数n，找到所有可能的分解（1在解中最多只能出现1次）。例如，输入8，输出是可以是1x8, 8x1, 2x4, 4x2, 1x2x2x2, 1x2x4, ……

欢迎在留言区交作业，并写下你今天的学习笔记。你可以点击“请朋友读”，把今天的内容分享给你的好友，和他一起精进。

精选留言

黄申

由于暂时无法追加回复，我这里再回复一下网友debugtalk

我看了Python的实现，果然很简介

奖金的题目没问题。整数的因子分解有点小瑕疵，少了一些可能。比如，输入8，少了

[1, 2, 2, 2]

[1, 2, 4]

[1, 4, 2]

[2, 1, 2, 2]

2018-12-20 04:09

杨景胜

从递归的两个原则到代码实现有点跳跃， 想了半天还是没想明白这个代码的逻辑啊~

2018-12-19 10:16

debugtalk

Python 实现赏金问题：https://github.com/debugtalk/geektime-notes/blob/master/programmers-mathematics/chapter5.md

Python 实现思考题：https://github.com/debugtalk/geektime-notes/blob/master/programmers-mathematics/chapter5.py

2018-12-19 13:40

作者回复

很棒 我稍后sync下来看

2018-12-20 01:28

李尧

思考题：请大神帮忙看下，输出少了个 [1,8]
输出：[2, 2, 2, 1] [2, 4, 1][4, 2, 1][8, 1]

```
import java.util.ArrayList;

/**
* @Auther: yli
* @Date: 2018/12/19 17:19
* @Description:
*/
public class Iteration {

public static void recursion(long total, ArrayList<Long> result){

if (total == 1){
result.add(1L);
System.out.println(result);
return;
}else {
for(long i = 2; i <= total; i ++){
ArrayList<Long> newList = (ArrayList<Long>)(result.clone());
newList.add(Long.valueOf(i));
if(total%i !=0){
continue;
}
recursion(total/i, newList);
}
}
}

public static void main(String[] args){
long total = 8;
recursion(total, new ArrayList<Long>());
}
}
```

2018-12-19 17:55

作者回复

循环的时候不能少了1，可以在结果中判断是否已经涵盖1，我稍微修改了一下

```
public static void recursion(long total, ArrayList<Long> result) {

if (total == 1) {
if (!result.contains(1L)) result.add(1L);
System.out.println(result);
return;
} else {
for (long i = 1; i <= total; i++) {
if ((i == 1) && result.contains(1L)) continue;
ArrayList<Long> newList = (ArrayList<Long>) (result.clone());
newList.add(Long.valueOf(i));
```

```
if (total % i != 0) {
continue;
}
recursion(total / i, newList);
}
}
}
```

2018-12-20 03:55

文刂氵共超

整数分解 - C++代码

```
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

// 输出函数
void PrintResult(vector<int> &Result)
{
for (size_t i = 0; i < Result.size(); ++i)
{
cout << Result[i] << " ";
}
cout << endl;
}

//数组中是否存在某值
bool IsExit(vector<int> &Result, int value)
{
vector<int>::iterator result = std::find(Result.begin(), Result.end(), value);
if (result == Result.end())
{
return false;
}
else
{
return true;
}
}

//整数分解
void RecursionAlgo(int Num, vector<int> &Result)
{
if (Num == 1)
{
PrintResult(Result);
return;
}
for (int i = 1; i <= Num; ++i)
{
```

```
//判断1是否在解中
if (IsExit(Result, 1))
{
if (i == 1)
{
continue;
}
}
if (Num%i == 0)
{
vector<int> newResult = Result;
newResult.push_back(i);

RecursionAlgo(Num/i, newResult);
}
}
}

int _tmain(int argc, _TCHAR* argv[])
{
int Totalmoney = 10;
vector<int> Result;

RecursionAlgo(Totalmoney, Result);
return 0;
}
```

2018-12-19 12:18

作者回复

回答很棒，下次可以将运行结果也贴出来

2018-12-20 04:39

松原

黄老师，这句“递归和循环其实都是迭代法的实现”我不太理解。

如果递归和循环都属于迭代法，那么就是说递归是从属于迭代法的。而我所理解的迭代法的核心是从1到n的递推，而递归是从n到1的逐步求解的过程，两者应该是并列的关系。希望老师能解答我的这个困惑。

2018-12-19 20:52

作者回复

确实两者的递推方向是不一样的，不过递归在计算机的实现中，是使用的函数调用，在满足条件后，函数开始逐级返回，这时候又是正向递推了，所以我觉得这也是从1到n

2018-12-19 22:47

icy

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 24 14:30:38 2018
@author: icytanz
"""

import copy

def get_mul_factor(num,result=[]):
if(num==1):
total=copy.copy(result)
```

```
if 1 not in total:
total.append(1)
print(total)
return
elif(num<1):
return
else:
n=list(range(num+1))
n.reverse()
if 1 in result:
m=range(len(n)-2)
else:
m=range(len(n)-1)
for i in m:
new_result=copy.copy(result)
if num%n[i]==0:
new_result.append(n[i])
get_mul_factor(num//n[i],new_result)

if __name__=='__main__':
get_mul_factor(2)
#[2, 1]
#[1, 2]

get_mul_factor(1)
#[1]

get_mul_factor(4)
#[4, 1]
#[2, 2, 1]
#[2, 1, 2]
#[1, 4]
#[1, 2, 2]

get_mul_factor(6)
#[6, 1]
#[3, 2, 1]
#[3, 1, 2]
#[2, 3, 1]
#[2, 1, 3]
#[1, 6]
#[1, 3, 2]
#[1, 2, 3]

get_mul_factor(8)
#[8, 1]
#[4, 2, 1]
#[4, 1, 2]
#[2, 4, 1]
#[2, 2, 2, 1]
#[2, 2, 1, 2]
```

#[2, 1, 4]
#[2, 1, 2, 2]
#[1, 8]
#[1, 4, 2]
#[1, 2, 4]
#[1, 2, 2, 2]

2018-12-24 15:06

郑晨Cc

```
package 二进制;

import java.util.ArrayList;

public class Lesson5Test {

public int counter = 0;



public void chosen(int num,ArrayList list){

if(null==list){
list = new ArrayList();
list.add(1);
list.add(num);
}

if(num==1){ //递归终止
return;
}

for(int i=1;i<=num;i++ ){
if(i==1){
System.out.println(list);
counter++;
continue;

}
if(i==num){
list.add(1);
list.remove(0);
System.out.println(list);
counter++;
continue;

}

if(num%i==0){
int a = num/i;
ArrayList list2 = new ArrayList(list);
list2.remove(list.size()-1);
list2.add(i);
```

```
list2.add(a);
chosen(num/i,list2);
}
}

}

public static void main(String[] args){

Lesson5Test too = new Lesson5Test();

too.chosen(8, null);
System.out.println("总数："+too.counter);

}

}
```

控制台输出：
[1, 8]
[1, 2, 4]
[1, 2, 2, 2]
[2, 2, 2, 1]
[2, 4, 1]
[1, 4, 2]
[4, 2, 1]
[8, 1]
总数：8
2018-12-20 14:10

debugtalk
感谢 Sean 的指正，之前整数因子分解的解答的确存在问题，没有完整考虑整数 1 在各种位置的情况。
现已更正：https://github.com/debugtalk/geektime-notes/blob/master/programmers-mathematics/chapter5.py
2018-12-20 12:59

杨景胜
//因数分解

```
public static void getMultiFactors(long multi,ArrayList<Long> result){
if (multi == 1){
result.add(multi);
System.out.println(result);
}else{
for (long i = 2; i <= multi; i++) {
if(multi % i == 0){
ArrayList<Long> newResult = (ArrayList<Long>) result.clone();
newResult.add(i);
getMultiFactors(multi / i,newResult);
}
}
}
}
```

2018-12-19 11:21

作者回复

如果循环从2开始，可能会漏掉一些情况，请参考为我给另一位网友李尧的回复

2018-12-20 04:43

方得始终

参考老师和其他同学的留言, 下面是Pythoni实现的思考题, 应该是个较为简洁的版本.

```
import copy

def prod_factors(num, result=[]):
if num == 1:
print('x'.join([str(_) for _ in result]))
if 1 not in result:
result.append(1)
print('x'.join([str(_) for _ in result]))
elif num < 0:
return
else:
for i in range(1, num+1):
if (i == 1 and i not in result) or (i !=1 and num % i == 0):
newresult = copy.copy(result)
newresult.append(i)
prod_factors(num/i, newresult)


prod_factors(8)
1x2x2x2
1x2x4
1x4x2
1x8
2x1x2x2
2x1x4
2x2x1x2
2x2x2
2x2x2x1
2x4
2x4x1
4x1x2
4x2
4x2x1
8
8x1
```

2018-12-29 07:10

作者回复

同时考虑了1出现和不出现的情况

2018-12-31 07:03

清如許

这里贴代码对Python这种缩进语言来讲很不友好啊，建议可以提交图片或者支持md格式。说下自己的思路：一个整数num 对 num 到 1 之间的整数 i 分别求余，如果余数为0，说明这是一个因子，将 i 添加到结果列表里，然后再让num 对 i 取整，得到 k ，对整数 K 再次递归求解。退出条件，如果 num == 1，那么将 1 添加到结果列表里，并打印结果列表。这里要注意下，如果 i == 1 ,也是退出条件，此时将 num 加入结果列表并打印。因为，一个大于1的数据除以1，永远得不到1 ，也就达不到前面的退出条件。源代码见 https://github.com/somenzz/geekbang/blob/master/mathOfProgramer/chapter05_recursion_1.py

2018-12-20 09:02

作者回复

少了一些可能。比如，输入8，少了

[1, 2, 2, 2]

[1, 2, 4]

[1, 4, 2]

[2, 1, 2, 2]

2018-12-20 11:27

文刂氵共超

使用C++实现赏金问题

```
#include <vector>
#include <iostream>

using namespace std;

// totalmoney : 总金额
// select : 钱币面额的选择列表
// Result : 结果列表
void RecursionAlgo(int Totalmoney, vector<int> &Select, vector<int> &Result)
{
//每次递归需要用总金额减去使用的面额，直到Totalmoney=0，表示找到解
if (Totalmoney == 0)
{
for (size_t i = 0; i < Result.size(); ++i)
{
cout << Result[i] << " ";
}
cout << endl;
return;
}
//Totalmoney < 0, 不符合标准 返回
else if (Totalmoney < 0)
{
return;
}
else
{
for (size_t i = 0; i < Select.size(); ++i)
{
vector<int> newResult = Result;
newResult.push_back(Select[i]);
RecursionAlgo(Totalmoney-Select[i], Select, newResult);
}
}
}
int _tmain(int argc, _TCHAR* argv[])
{
int Totalmoney = 10;

int ia[] = {1, 2, 5, 10};
vector<int> Select(ia, ia+4);
```

```
vector<int> Result;

RecursionAlgo(Totalmoney, Select, Result);
return 0;
}
```

2018-12-19 11:34

唐瑞甫

我来答下吧，案例中的n代表已经迭代的次数，在这个案例中，限制了每轮迭代只有四种选择

2018-12-19 08:56

cocu

这个案例中的n到底是什么，是奖励总金额，还是取的纸币数?

2018-12-19 07:22

作者回复

这里我统一回答一下，是当前迭代的次数，也就是取的纸币数量

2018-12-19 09:17

杰之7

通过这一节的阅读和听音频学习，对递归有了一个理解。其中的思想就是解决当前的一个问题，并把问题进行转化下去，使其更容易解决。

递归和循环都是不断进行迭代，直到最终能得到我们需要的结果。其中的不同是递归需要保留中间处理的数值。

老师，现在我编程基础相对较弱，这篇Java实现的递归中间我还是有些地方没有完全理解。当然老师已经写了注解。

2019-02-11 17:04

作者回复

没关系，多多上手写几遍，尝试不同的写法，慢慢就会熟悉。

2019-02-12 01:27

崇宗

参考了大家的实现后再修改的版本（之前的实现忘记列举出所有的可能性）

```
def factorization(num):
if(num <= 0):
return []
elif(num == 1):
return [[1]]
else:
res = []
res.append([1, num])
for i in range(2, num + 1):
if((num % i) == 0):
temp_res = factorization(num // i)
for j in range(len(temp_res)):
temp_res[j].append(i)
res = res + temp_res
return deEquivalentLists(res)



def factorization_print(num):
res = factorization(num)
char = 'x'
res_str = [char.join([str(num) for num in ls]) for ls in res]
for string in res_str:
print(string)
```

```
def arrange(ls):
if len(ls) == 0:
return [[]]
elif len(ls) == 1:
return [ls]
else:
res = []
for i in range(len(ls)):
for arr_res in arrange(ls[:i] + ls[i + 1:]):
res.append([ls[i]] + arr_res)
return res


def deEquivalentLists(lists):
res = []
for ls in lists:
ls.sort()
if ls not in res:
res.append(ls)
return res


def deDuplicatedLists(lists):
res = []
for ls in lists:
if ls not in res:
res.append(ls)
return res

if __name__ == '__main__':
number = 8
res = factorization(number)
arrange_res = []
for ls in res:
arrange_res = arrange_res + arrange(ls)
final_res = deDuplicatedLists(arrange_res)
print(final_res)
print(len(final_res))
```

Result:
[[1, 8], [8, 1], [1, 2, 4], [1, 4, 2], [2, 1, 4], [2, 4, 1], [4, 1, 2], [4, 2, 1], [1, 2, 2, 2], [2, 1, 2, 2], [2, 2, 1, 2], [2, 2, 2, 1]]
12

2019-02-01 22:33

悬炫

JavaScript实现：
思考题：

```
/**
*
* @description 求某个整数的公约数，不包括1,因为1会导致死循环
```

```
* @param {*} n
* @returns
*/
function commonDivisor(n) {
if (!Number.isInteger(n) || n <= 0) {
throw new Error("请输入正自然数");
}
let list = [];
for (let i = 2; i <= n; i++) {
let value = n / i;
if (Math.trunc(value) === value) {
list.push(i);
}
}
return list;
}

/**
*
* @description 循环递归求解不包含约数1的可能拆分方式
* @param {number} inputNumber //目标整数
* @param {array} commonDivisorList //公约数列表
* @param {array} resuleList //可能拆分方式的集合
* @param {array} [result=[]] //保存当前的拆分方式
*/
function get(inputNumber, commonDivisorList,resuleList,result=[]) {
// 当 inputNumber = 0 时，证明它是满足条件的解，结束嵌套调用，输出解
if (inputNumber === 1) {
// console.log(result);
resuleList.push(result)
// 当 inputNumber < 0 时，证明它不是满足条件的解，不输出
} else if (inputNumber < 1) {
return
} else {
for (let index = 0; index < commonDivisorList.length; index++) {
let newResult = [...result]; // 由于有 4 种情况，需要 clone 当前的解并传入被调用的...
newResult.push(commonDivisorList[index]); // 记录当前的选择，解决一点问题
get(
inputNumber / commonDivisorList[index],
commonDivisorList,
resuleList,
newResult
); // 剩下的问题，留给嵌套调用去解决
}
}
}

/**
*
* @description 输出所有可能拆分方式，包括约数1
* @param {*} n //目标整数
```

```
* @param {array} [resuleList=[]] //保存当前的拆分方式
*/
function division(n, resuleList = []) {
//求得公约数列表
let commonDivisorList = commonDivisor(n);
//求得不包含约数1的可能拆分方式
get(n, commonDivisorList, resuleList)
let list = [...resuleList]
let listLength = list.length;
list.forEach((element) => {
let length = element.length;
for (let index = 0; index < length; index++) {
let list=element.slice()
list.splice(index, 0, 1)
resuleList.push(list)
}
resuleList.push(element.concat(1))
})
resuleList=resuleList.slice(listLength)
console.log(resuleList);

}
//结果有12种 ，篇幅原因，无法一一列出，大家可以直接在控制台跑一下
division(8);
```

2019-01-28 13:42

作者回复

结果和思路都是正确的

2019-01-29 01:51

悬炫

JavaScript的实现：

赏金问题：

```
/**
*
* @description 赏金问题
* @param {number} totalReward 奖赏总金额
* @param {array} result 保存当前的解
* @param {array} rewards 可供选择的面额
* @returns void
*/
function get(totalReward, result=[],rewards = [1, 2, 5, 10]) {
// 当 totalReward = 0 时，证明它是满足条件的解，结束嵌套调用，输出解
if (totalReward === 0) {
console.log(result.toString());
// 当 totalReward < 0 时，证明它不是满足条件的解，不输出
} else if(totalReward<0) {
return;
} else {
for (let index = 0; index < rewards.length; index++) {
let newResult =[...result];// 由于有 4 种情况，需要 clone 当前的解并传入被调用的函数
newResult.push(rewards[index]);// 记录当前的选择，解决一点问题
```

```
get(totalReward - rewards[index], newResult);// 剩下的问题，留给嵌套调用去解决
}
}
}
```

2019-01-28 13:41

金博

JavaScript的实现方式：

```
function recursion(n, result) {
if (Number.isInteger(n) && n > 1) {
for (var i = n; i > 1; i--) {
var newResult = result.concat();
newResult.push(i);
recursion(n / i, newResult);
}
}
else if (n === 1) {
console.log(result);
}
else {
return;
}
}
```

2019-01-16 17:10