24 点游戏

王镇宇 P18106035

1.引言

24 点是一种可以用排列组合解决的游戏，它的具体玩法如下：

给玩家 4 张牌，每张牌的牌面值在 1 - 13 之间，允许其有相同的牌。采用加、减、乘、除四则运算，允许中间存在小数，且可使用括号，每张牌仅可使用一次，尝试构造一个表达式，使其运算结果为 24。

组合数学的魅力在于将数字和运算符巧妙地组合之后，可以用来解决日常生活中遇到的很多排列组合问题。24 点就是一种常见的排列组合问题，不过在此之上我们涉及到了数值的运算以及运算符优先级等组合问题。

本文从排列组合问题的角度出发，给出了 24 点问题的两种解法。第二节分析解决思路，第三节介绍两种方法，第四节结果分析，第五节总结。

2.问题分析

2.1 输入输出

根据上述游戏规则构造一个 24 点游戏算法，输入输出要求如下：

输入：n_1, n_2, n_3, n_4。

输出：若能得到运算结果 24，输出“Success”和任意一个对应的表达式，否 则输出 Fail。

如：

输入：11, 8, 3, 5

输出：Success

(11 - 8) * (3 + 5)

2.2 分析

如果使用穷举法，那问题就变成了求排列组合的所有情况。在算数运算的过程中我们需要考虑到数值的排列、运算符的先后以及括号对运算优先级的影响。分析如下：

1. 每个数字只能使用一次，故给定的四个数字共有 4! = 24 种排列。
2. 四个数需要三个运算符，每个运算符有四种可能，对于同一个排列，则运算符共有 $4^3 = 64$ 种组合。
3. 考虑括号对运算优先级的影响，对于四个输入 A, B, C, D, 共有五种括号组合方式：((AB)(CD)), (((AB)C)D), ((A(BC))D), (A((BC)D)), (A(B(CD)))。

   综上，共有$4! \times 4^3 \times 5 = 7680$ 种排列组合。

3. 问题解法

3.1 分治递归

通过分析，很容易就能想到一种解法，即遍历运算符、数字和括号的所有排列组合形式，在计算相应的输出值，若符合结果则返回相应的组合。

为了简化运算过程，使得代码更简洁易懂，可以采用分治递归的方法来优化，对于给定集合 S={A, B, C, D}，定义函数 α(S) 为对集合 S 中所有元素进行可能的四则运算所得到的值的集合。首先从 S 中取出任意两个数值，如取出 A 和 B，对 A 和 B 进行四则运算，所有的可能为 A+B、A-B、B-A、A×B 、A/B、B/A。将这些值赋给 A，再将 S[-1]即 D 值赋给 B，可得到$S_1 = \{A + B, D, C\}$, $S_2 = \{A - B, D, C\}$, $S_3 = \{B - A, D, C\}$, $S_4 = \{A * B, D, C\}$,$S_5 = \left\{\frac{A}{B}, D, C\right\}$,$S_6 = \left\{\frac{B}{A}, D, C\right\}$ 六个新集合，这样就使得 S 缩减成为长度为 3 的集合，可将其认为是集合 S 的子集，有关系式 $\alpha(S) = \bigcup_{i=1}^{6} \alpha(S_i)$。

如上方法可对问题进行分治，将 4 个数的运算简化成 3 个数的六个子问题之和，再对子问题进行递归求解，算法如下，仅列举 S_1 递归方法，其余子集方法相同。

```
# Recursive Solution #
# num 定义 S 集合中元素个数
def GameAnalysis(self, num):
    if num == 1:
        if number[0] == TargetResult:
```

```
                return True
            else:
                return False
    for i in range(num):
        for j in range(i + 1, num):
            a, b = number[i], number[j]
            number[j] = number[num - 1]
            str_a, str_b = result[i], result[j]
            result[j] = result[num - 1]
            # a + b (b + a)
            result[i] = '(' + str_a + '+' + str_b + ')'
            number[i] = a + b
            if self.GameAnalysis(num - 1):
                return True
    return False
```

3.2 动态规划

上述解法很清晰也很容易理解，但从时间复杂度和空间复杂度来考虑，穷举法无疑都是效率相当差的，虽然已经已经采用了递归分治的思想来简化，但其中很多一部分冗余计算都未做有效优化，如未考虑加法的重复计算和交换律等。这一节应用分治和动态规划的方法，优化存储结构，构造了一种更加高效的解法。

首先从分治法出发，对于给定集合 S={A, B, C, D}，可分为 4 种：{∅}∪{A, B, C, D}, {A}∪{B, C, D}, {A, B}∪{C, D}, {A, B, C}∪{D}。划分若包含∅则转化原问题，可使用分治递推法解决，故此处不考虑包含∅的划分。则我们队集合的划分只需考虑三种情况，对划分{A}∪{B, C, D}，显然{B, C, D}可以继续进行划分为{B}∪{C, D}或{B, C}∪{D}，当划分中的元素个数为 2 时，则划分结束。对划分结束的集合进行计算，如对 P={A, B}，返回 A 与 B 进行四则运算的六种结果（注意考虑被除数为 0 的情况），若 P 划分中元素个数为 1 时，则返回 P。

显然对三元划分{A, B, C}和{B, C, D}，会出现重复的划分{B, C}，重复计算然是降低程序效率的重要因素，计算过的二元划分值我们用字典将其存储，在对二元划分进行计算时则对字典进行检索，时间复杂度 O(1)。

定义的基本方法如下：

```
# Dynamic program #
# 对二元划分处理，取值或计算并存入字典
def PairCalculate(self, curlist)
# 对划分结束后各划分的值进行遍历计算
def Calculate(self, llist, rlist)
# 在字典中寻找二元划分
def FindPair(self, curlist)
# 按照当前划分长度进行递归处理
def PartiAnalysis(self, curlist)
# 初始化生成所有四元组，进行三种基本划分后进行 PartiAnalysis
def OptimizedAnalysis(self, numberlist)
```

4. 结果分析

4.1 测试平台

PC: MacBook Pro
CPU: 2.9 GHz Intel Core i5
Python version: 3.6.5 64-bit

4.2 测试结果

Data	Recursive Solution (s)	Dynamic Program (s)	Result
3,8,2,11	0.002866983413696289	0.0007429122924804688	(((3+8)+11)+2)
7,9,1,10	0.013345956802368164	0.0012960433959960938	((7-10)*(1-9))
3,5,8,9	0.013949155807495117	0.0007441043853759766	(((3*9)+5)-8)

由上表数据易见采用动态规划法及优化过的数据存储方法性能总体远优于分支递归的方法，性能提升 4~10 倍。

5. 总结

本文从实际生活问题出发，实现了游戏 24 点的解决方案。从提出问题，在运用排列组合相关知识先对问题的基本解决思路有了大体把握，再通过简单的递归算法实现了第一种解决方案。仅仅实现功能远不能满足现实中出现的规模更大数据更复杂的问题，因此对递归算法进行了分析，着重解决其性能瓶颈，对数据的重复运算浪费大量时间和资源，针对游戏机制分析进行了更细的元组划分，采用字典机构存储二元划分，以此来提升数据的运算效率，利用动态规划的思想来更一步优化了程序性能。

动态规划法虽然性能优于递归法，但缺点显而易见，扩展性很差，递归法可以在给定 N 各元素的情况下算出得出结果 S 的排列组合，而前者却不行，在 N=5 时划分就不再是 3 种，代码需要重构，可以设计一种平衡性能和扩展性的算法来满足。