Python 数据结构与算法分析（第四章 递归）

1. 基本概念

- **递归**：递归是解决问题的一种方法，它将问题不断地分成更小的子问题，直到子问题可以用普通的方法解决。通常情况下，递归会使用一个不停调用自己的函数。
- **递归三原则**：
  - 递归算法必须有基本情况；
  - 递归算法必须改变其状态并向基本情况靠近；
  - 递归算法必须递归地调用自己。

2. 复杂递归问题

（1）汉诺塔问题

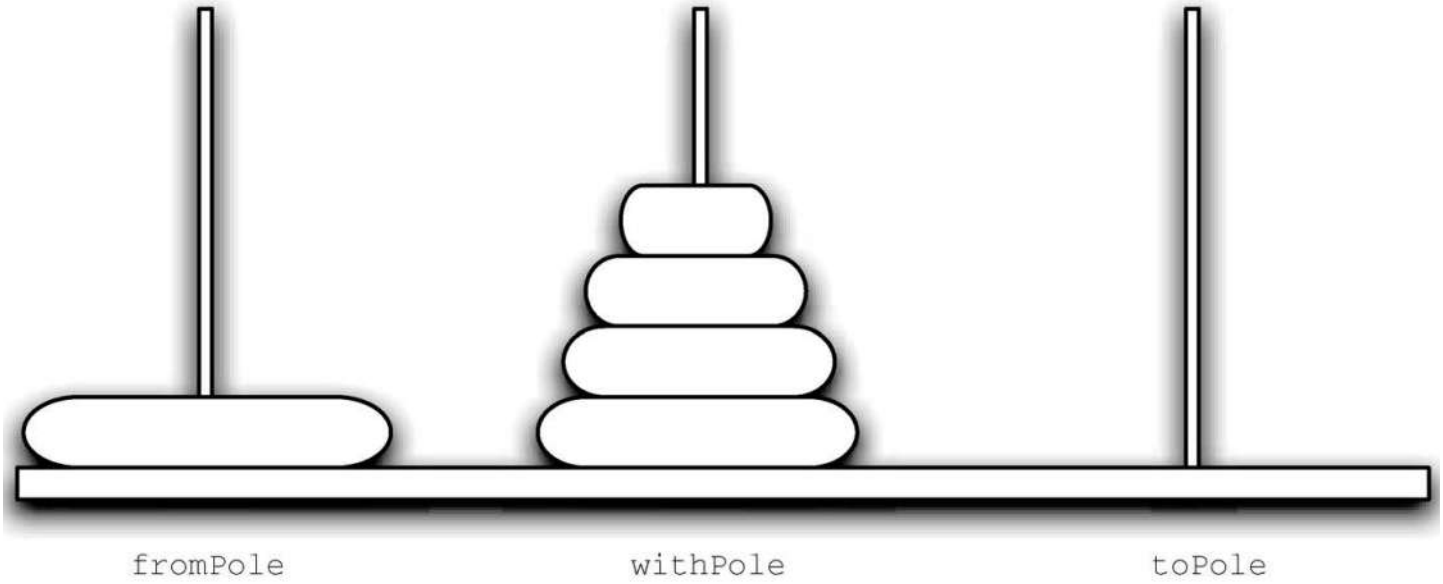

图 4-11　汉诺塔问题示例

要求将fromPole柱子上的n个圆盘移动至toPole柱子上，该问题可由递归算法完美解决。

对于n个圆盘，我们可以将其视为两个圆盘，即最底层圆盘和上层$n-1$个圆盘。因此，要将n个圆盘移动至toPole柱子上，其可通过如下三个步骤完成：

- Step1. 将$n-1$圆盘从fromPole柱子移动至withPole柱子上；
- Step2. 将最底层圆盘从fromPole柱子移动至toPole柱子上；
- Step3. 将$n-1$圆盘从withPole柱子移动至toPole柱子上；

而将$n-1$圆盘从fromPole柱子移动至toPole上，又可以视为将$n-1$最底层圆盘移动至toPole柱子上和将$n-2$圆盘移动至toPole柱子上。故，该过程可由递归实现。

```
def moveTower(n, fromPole, toPole, withPole):
    if n >= 1:
        moveTower(n-1, fromPole, withPole, toPole)  ## 将n-1圆盘从fromPole柱子经由toPole柱子移动至withPole柱子
        print("moving disk from %d to %d\n" % (fromPole, toPole))  ## 将n圆盘从fromPole柱子移动至toPole柱子
        moveTower(n-1, withPole, toPole, fromPole)  ## 将n-1圆盘从withPole柱子经由fromPole柱子移动至toPole柱子
    else:
        print('Please input again.')

moveTower(4, 1, 3, 2)
```

```
moving disk from 1 to 2
moving disk from 1 to 3
moving disk from 2 to 3
moving disk from 1 to 2
moving disk from 3 to 1
moving disk from 3 to 2
moving disk from 1 to 2
moving disk from 1 to 3
moving disk from 2 to 3
moving disk from 2 to 1
moving disk from 3 to 1
moving disk from 2 to 3
moving disk from 1 to 2
moving disk from 1 to 3
moving disk from 2 to 3
```

（2）贪心算法

考虑如下问题，假设某个自动售货机制造商希望在每笔交易中给出最少的硬币。一个顾客使用一张一美元的纸币购买了价值 37 美分的物品，最少需要找给该顾客多少硬币呢?

该问题是贪婪算法的典型应用，即从面值最大的硬币开始，使用尽可能多的硬币，然后尽可能多地使用面值第2大的硬币。其它类似问题还包括如背包问题，即背包容量有限，如何选择宝石组合使得所装宝石价值最大等。

然而若需要枚举出所有找零的可能，该问题即可使用递归算法求解。

- step1. 输入需换零金额，返回不同的找零方案后对应的还需换零金额，入队列。如换零金额26，则经[1, 5, 10, 25]找零后还需换零的金额[25, 21, 16, 1]。
- step2. 队列取数，递归Step1. 至还需换零金额为0。

```
from treelib import Tree
from queue import Queue

combine_selection = [1, 5, 10, 25]
combine_number = 8


def minus(input_num):
    one_return = None
    five_return = None
    ten_return = None
    quarter_return = None
    if input_num >= 25:
        quarter_return = input_num - 25
    if input_num >= 10:
        ten_return = input_num - 10
    if input_num >= 5:
        five_return = input_num - 5
    if input_num >= 1:
        one_return = input_num - 1
    return [one_return, five_return, ten_return, quarter_return]


tree = Tree()
tree.create_node(tag=combine_number, identifier=0)
queue = Queue()
queue.put(combine_number)
while not queue.empty():
    input_temp = queue.get()
    parents = []
    temp_result = minus(input_temp)
    temp_result = [i for i in temp_result if i != None]
    for i in tree.leaves():
        if i.tag == input_temp:
            parents.append(i.identifier)
    for parent_temp in parents:
        for temp_node in temp_result:
            tree.create_node(tag=temp_node, parent=parent_temp)
    for temp_in in temp_result:
        queue.put(temp_in)

print(tree.show())
```

```
8
├── 3
│   └── 2
│   └── 1
│   └── 0
└── 7
├── 2
│   └── 1
│   └── 0
└── 6
├── 1
│   └── 0
└── 5
├── 0
└── 4
└── 3
└── 2
```

> └── 1
> └── 0

基于递归的暴力枚举法在时间成本上开销较大，对于较大的数字是不可接受的。对此，这里还可以考虑使用动态规划的方法。动态规划算法会从 1 分找零开始，然后系统地一直计算到所需的找零金额。这样即可保证每一次的找零策略均以上一次的找零策略为基础，如23找零将以22找零为基础。同时，上一次的找零策略又将是最优的策略。因此，每一次找零均为最优策略。

```
def minus_once(number):
    if number == 5:
        return 5, number-5
    elif number == 10:
        return 10, number-10
    elif number == 25:
        return 25, number-25
    elif number == 1:
        return 1, number-1
    else:
        for minus_index, i in enumerate([1, 5, 10, 25]):
            if number-i < 0:
                break
        return [1, 5, 10, 25][minus_index-1], number-[1, 5, 10, 25][minus_index-1]


def dp(number):
    dict_op = {}
    dict_op[0] = [0]
    for number_temp in range(1, number+1):
        charge, minus_result = minus_once(number_temp)
        dict_op[number_temp] = [charge] + dict_op[minus_result]
    return dict_op

print(dp(26))
```

> {0: [0], 1: [1, 0], 2: [1, 1, 0], 3: [1, 1, 1, 0], 4: [1, 1, 1, 1, 0], 5: [5, 0], 6: [5, 1, 0], 7: [5, 1, 1, 0], 8: [5, 1, 1, 1, 0], 9: [5, 1, 1, 1, 1, 0], 10: [10, 0], 11: [10, 1, 0], 12: [10, 1, 1, 0], 13: [10, 1, 1, 1, 0], 14: [10, 1, 1, 1, 1, 0], 15: [10, 5, 0], 16: [10, 5, 1, 0], 17: [10, 5, 1, 1, 0], 18: [10, 5, 1, 1, 1, 0], 19: [10, 5, 1, 1, 1, 1, 0], 20: [10, 10, 0], 21: [10, 10, 1, 0], 22: [10, 10, 1, 1, 0], 23: [10, 10, 1, 1, 1, 0], 24: [10, 10, 1, 1, 1, 1, 0], 25: [25, 0], 26: [10, 10, 5, 1, 0]}

3. 参考文献

Python数据结构与算法分析（第2版）