陪伴小学生学习 Python 笔记：2024 年寒假

之一、算法入门练习

说明：

> 本文内容较长，持续更新中，连载于：`github/xiaohaimiao` Python 学习笔记

在 `github` 或 `markdown` 编辑器中阅读本文时，可打开其目录功能方便浏览和跳转。

> - `github` 页面中，点击文档右上角的目录图标；
> - `marktext` 编辑器中，按 `Ctrl-K`

目录

一、算法入门练习（本文）

1. 求素数，初步学习穷举法及优化

> `素数`（`质数`）是指在大于 `1` 的自然数中，除了 `1` 和 `它本身` 以外不再有其他因数的自然数。

题目要求：

> 求 100 之内的素数（质数）
>
> - 写一个函数 `isPrime(number:int)` 判断 `number` 是否是素数，如果是则返回 `True`，否则返回 `False`；
> - 然后用 `1-100` 的循环调用这个函数，判断每一个数字是否是素数并输出，如：
>
> > 素数：2, 3, 5, 7, 11, ... 非素数：1, 4, 6, 8, 9, 10, ...

思路分析：

> 对于 `n`，用从 `2` 到 `(n-1)` 的数去整除 `n`，只要有一个数能整除 `n` ，则 `n` 不是素数，反之是素数。
>
> 也就是说，如果在 `2 ~ (n-1)` 之间找到任意一个数字能够整除 `n`， `n` 就不是素数。
>
> 这种方法，称为`试除法`。

注意要点：

1. **注意要求的范围**：比如题目中 1 到 100，是否包含1、是否包含100？

   > 数学上把 `(1, 100]` 这样的集合称为：前开后闭的集合，范围是 `1 < n ≤ 100`，不包含`1`、包含`100`。类似的集合还有 `(1, 100)`， `[1, 100]`， `(1, 100]`， `[1, 100)`，以此类推。因为 1 不是素数（质数），所以本题的范围应该是？

2. **注意循环的起始和结束的边界范围**：

   > 比如 `for i in range(1, 100)`，对应上面的集合是哪一种？ 注意：不同计算机编程语言中，甚至相同编程语言中不同函数，它们的取值范围可能不一样。 例如， `range(1, 100)` 对应的集合是 `[1, 100)`，而 `range(100)` 对应的集合是 `[0, 100)`。 再例如，Python 中长度为 `n` 的列表的索引是从 `0` 开始到 `n-1`，范围是 `[0, n)`。代码中可用 `range(n)` 或者 `range(0, n)` 获得这个范围的整数值。如果试图用超过这个范围的数值作为索引访问列表中的某一项时，会导致代码运行时出现错误。

3. **注意整除与除的区别**：

> 两种方法判断是否整除，常用“求余数”的方法，“求余”又称为“求模”。 一个数除以另外一个数，余数为零，是整除。
>
> 数学角度：`10 ÷ 3 = 3 ... 1` —— 十除以三，商为三，余数为一， 用求余/求模运算：`10 % 3 = 1`。
>
> 编程角度：`10 / 3 = 3.3333...` 小数点后有几位，取决于不同的数据类型所保留精度的范围，注意这一点，暂不展开。 整除运算：`10 // 3 = 3` —— 十整除三，商为三，没有余数，此为 “整除” 。 在编程中，整除也可以用除后保留整数部分来实现，例如取整函数、将浮点型转换为整型来实现。 因为不同编程语言细节有差异，暂不展开。

4. **逐步优化算法**：先用代码编写出最简单的算法，再进一步优化。

> 求素数 C 代码

编程练习：

Python 参考代码如下，请自行实现后再对照查阅：

```python
def isPrime(number: int):
    if number < 2:
        return False
    for i in range(2, number): # 从 2 到 n-1
        if number % i == 0:
            return False # 是否整除
    return True
```

接下来，使用循环调用这个函数来判断 `1 到 100` 的每一个数字是否为素数，并输出结果。以下是示例代码：

```python
for num in range(1, 100 + 1):
    if isPrime(num):
        print(num, end=" ")
print()  # 输出换行
```

这段代码会遍历 `1 到 100` 的每一个数字，判断是否为素数，如果是素数，则输出该数字。

注意：`range(1, 100 + 1)` 的边界范围。

知识拓展：剪枝，缩小计算范围

本例用到一个循环：用每个数字去试除 `number`，这样的算法叫做 `穷举法`。

但即便是 `穷举法`，也是可以动脑筋 `进行优化` 的。

1. **优化：对上限剪枝、偶数**

在上面的代码中，判断 `1 到 100` 中的素数是可以缩小判断的范围的。

从数学角度来看，一个正整数 `n` 的因数，除了自身之外不可能有大于 n/2 的因数。

所以，上面代码中，采用 `试除法` 的循环可以`缩小一半`：

```
def isPrime(number: int):
    if number < 2:
        return False
    for i in range(2, number//2 + 1): # 从 2 到 n/2
        if number % i == 0:
            return False    # 是否整除
    return True
```

这样的优化，缩小了穷举的范围，减少了计算量，称为 `剪枝`。——好比修剪树木，剪去多余的树枝而并不影响树木的生长。

既然提到了除了自身的因数不可能大于 $n/2$，那么 `2、3、5、7` 的倍数呢？——显然都不是素数，可以简单排除一下偶数，至少在 $n/2$ 的范围内又减少了一半：

```
def isPrime(number: int):
    if number < 2:
        return False
    if number == 2:
        return True
    for i in range(3, number//2 + 1, 2): # 从 3 到 n/2 的奇数，步长为 2 跳过偶数
        if number % i == 0: # 是否整除
            return False
    return True
```

继续看，不仅 2 的倍数，3 的倍数也显然不是质数，2 x 3 = 6 包含了所有它们的公倍数，且起点可以从 5 开始，于是：

```python
def isPrime(number: int):
    if number < 2:
        return False
    if number == 2:
        return True
    # 从 5 到 n/2，步长 6，跳过 2、3 的倍数
    for i in range(5, number//2 + 1, 6):
        if number % i == 0: # 是否整除
            return False
    return True
```

2. 再优化：继续剪枝，平方根

还能不能进一步剪枝呢？

可以。

> 一个数的因数，除了本身之外，不仅不会超过 $n/2$，
>
> 还不会超过 $\sqrt{n}$。

于是，进一步剪枝的范围是：

> 从 3 到 $\sqrt{n}$ 的奇数，前开后闭。

在 Python 中，不引入 math 包的情况下，可用下面代码来计算 $\sqrt{n}$：

`int(number**0.5)`

> `number**n` 表示对 `number` 求 `n 次幂` 的运算，
>
> `number**0.5` 表示对 `number` 求 $1/2$ 次幂——也就是求平方根，
>
> 再将其抛弃小数部分（转换为整数）用到 `int()`，
>
> 合起来就是：`int(number**0.5)`

所以，进一步剪枝后的代码如下：

```python
def isPrime(number: int):
    if number < 2:
        return False
    if number == 2:
        return True
    if number % 2 == 0: # 排除偶数
        return False
    # 从 5 到 n/2，步长 6，跳过 2、3 的倍数
    for i in range(5, int(number**0.5) + 1, 6):
        if number % i == 0: # 是否整除
            return False
    return True
```

——为什么上限是 $\sqrt{n}$？

在试除法判断素数时，有一个重要的推论：

> 如果一个数 `n` 不是素数，那么它一定可以被两个因数 `a` 和 `b` 相乘得到，其中 $a <= b$。
>
> 假设 `n` 可以被 `a` 整除（显然 `n` 不是素数），即 $n = a \times b$。
>
> 如果 `a` 大于 `n` 的平方根 `c`，那么 `b` 一定小于 `n` 的平方根 `c`，否则 $a \times b$ 就会大于 $c^2 = n$，与前提矛盾。
>
> 因此，在试除法中尝试的除数范围上限不超过 `n` 的平方根。

对于小学生小朋友，换个角度，借助几何来理解可能容易一些：

> - `n` 是某个长方形的面积。该长方形的长和宽假设为 `a, b`，则 $n = a \times b$;
> - 当两条边边长相等 $a = b = c$ 时，`n` 是一个边长为 `c` 的正方形 $n = c^2$;
> - 当两条边长度不相等时，$a < c < b$ 或 $a > c > b$。也就是说，其中一边长度肯定超过正方形边长，另外一边则小于正方形边长——否则 $a \times b > c^2$。
>
> ——也就是说，`a 或 b 必然有一边小于等于 c`，
>
> 所以剪枝的上限可以设为 $c = \sqrt{n}$，也就是 `n 的平方根`。
>
> 因为，判断素数只需要有一边能整除 `n` 即可 ——按循环来看，显然先找到小的数字就不必再找成对的大的数字了。

- [ ] （待补充：这里可以引入 Geogebra 的动态公式）

扩展思考：

$n/2$ 与 $\sqrt{n}$ 哪个大？——如何求证？

提示：想想 `边长为 1 的正方形` 的 `边` 和 `对角线`，哪个更长？

---

2. 求回文数，练习拆分数字和进制转换

> `回文数` `Palindrome Number` 是对称数，指正向读与反向读是相同的，如 `12321`，`33433` 等。

题目要求：

> - 编写函数 `isPalindrome(number:int)`，判断传入的参数 `number` 是否是回文数。如果是，返回 `True`，否则返回 `False`；
> - 再调用这个函数，找出四位数中的回文数，打印并计算个数。

> 求回文数 C 代码

思路分析：

先不要看，自己想了再对照看思路，看你能不能想到更多：

> 可以利用正向与反向相同或对称位数字相同来判断。

1. 反转字符串：利用**字符串反转**，**数值转字符串**，比较反转前后的**字符串的数值**；
    1. Python 特有的语法，在其它语言中**非通用**；
    2. Python 列表自带的函数，在不支持类似方法的语言中**非通用**。虽然其它语言也有类似方法，但很简单不妨自己试试编写；
    3. 通用方法：尝试自己编写一个反转字符串的函数。
2. 转字符串：**判断对称位置字符是否相同**：
    1. 转为列表，用索引取列表中对称位置的字符**进行字符比较**；
    2. 不转为列表，用字符串操作函数取对称位置的的字符**进行字符比较**；
3. 不转字符串，**用数学方法取数值的对称位置的数字**（digit）**进行数值比较**。

    ——都要掌握，尤其必须掌握 解法思路3

编程练习1：用基本思路实现要求

对照上面的思路分析，编写多个函数，分别完成以下练习：

1. 熟悉数值转字符串、字符串转数值的操作；
2. 对比 Python 中两种取字符的方式：
    1. 用 `for` 枚举字符串中的每个字符；
    2. 用 `for i in range(n)` 循环，并用索引方式访问字符串中的每个字符的方式 `list[i]` ——此种方式更通用，可适用于其它语言，如 C/C++、Java/C# 等。
3. 熟悉 Python 特有的取逆向字符串的方式，但要学会用其它方法实现：
    1. Python 特有/**非通用**，**反向取字符串**：`string[::-1]` ——大部分编程语言不支持。

```
def isPalindrome1(number:int):
# 获得数字对应的字符串
digits = str(number)
# 逆向取字符串
reverseDigits = digits[ : :-1]
# 返回两个列表是否相等的逻辑值
return digits == reverseDigits
```

    2. Python 特有/**非通用**，将列表反序： `list.reverse()` ——部分语言内置函数库或开发框架也有类似方法。

```
def isPalindrome2(number:int):
# 获得数字对应的字符串
digits = str(number)
# 从字符串构造列表
digitList = list(digits) # ['1', '2', '3', '4', '5']
# 逆向取新的列表
reverseList = digitList.reverse() # ['5', '4', '3', '2', '1']
#text = "".join(reverseList) # 可将数组元素连接为字符串用于显示等
# 返回两个列表是否相等的逻辑值
return digitList == reverseList
```

    3. **通用方法，语言无关**：用字符串操作判断对应位置字符是否相同。

        注：练习使用正确的索引值范围，避免出错。

        注：不同编程语言的字符串取值操作方式和函数不同。

        ——略，请自行练习。

4. **通用方法，语言无关**：用列表操作判断对应位置字符是否相同。

   注：与上一项区别不大，需要用到字符串转列表的函数 `list(str(number))`

   注：不同语言的字符串取值操作方式和函数不同。

   ——略，请自行练习。

知识拓展1：用数学方法取数值的某位数字

通用方法，语言无关

> 十进制数的每一位，对应的数学含义是 `10 的 n-1 次方` 有 `n` 个。
>
> 例如：12345
>
> `千位` 对应着 `10 的 3-1 次方`，也就是 `100`，有 `3` 个；
>
> 12345 的 `千位是 2`，意味着 `2 个 1000`； `百位是 3`，意味着 `3 个 100`，以此类推。
>
> 例如：`12345` 整个数字拆分为：
>
> $$10000 + 2000 + 300 + 40 + 5 = 12345$$
>
> 也就是：
>
> $$1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

1. **思考：** 如何求一个数字的个位数字？ ——先自己想想再看下面内容：

> 如果要取末位（个位）上的数字，只需对 `10` 求模即可，即除以十的余数。
>
> 例如：`12345` 的个位，是：`12345 % 10 = 5`

2. **思考：** 求一个数字上任意位的数字？ ——先自己想想再看下面内容：

> 求某个数字的某位上的数字是几，可以将该数字缩小 `10` 的 n-1 次方倍并取整数部分，然后求其除以 `10` 的余数。
>
> 例如：`12345` 的百位上的数字是：`(12345 // 100) % 10 = 3`
>
> 求 `12345` 百位上的数字，则先将 `12345 缩小 100 倍`（`10 的 2 次方`），抛弃小数点后的部分得到 `123`（直接整除），然后除以 10 求余数（对 10 求模）。
>
> 类似，求 `12345` 千位上的数字：`(12345 // 1000) % 10 = 2`

编程练习2：用数学方法取数判断是否是回文数

根据知识拓展1，请自行完成以下编程练习：

```
# 用数学方法求指定位置上的数字，比如 12345，求第右边数第2位（十位）和第4位（千位）
def getDigitByPos(number:int, pos:int):
    x = number    # 比如 12345
    # 略，考验你的时候到了，请自行补全代码

    return x

def isPalindrome(number:int):
    # 请自行补全代码，加上循环对比每一位与对称位置的数字
    #for i in ...
        #digit1 = getDigitByPos(n, i??)
        #digit2 = getDigitByPos(n, i??)
        #print(digit1, digit2, digit1 == digit2)
        # 如果不相等，则

        # 否则
    #return True/False ?
```

知识拓展2：进制转换

> 因为不同进制的"数字"的定义的区别在于"`逢几进位`"——比如二进制是`逢2进位`，八进制是`逢8进位`，十进制是`逢10进位`，十六进制是`逢16进位`，等等。
>
> 所以，在任意进制之间转换的方法是相同的，因为：
>
> 每一位代表着该进制的 `n-1` 次方
>
> 每一位上的数字是几，意味着有几个 `n-1` 次方

> 例如十进制的 `12345` 意味着：
>
> $1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$
>
> $= 1 \times 10000 + 2 \times 1000 + 3 \times 100 + 4 \times 10 + 5 \times 1 = 12345$

> 八进制的 `12345` 意味着（转为十进制）：
>
> $1 \times 8^4 + 2 \times 8^3 + 3 \times 8^2 + 4 \times 8^1 + 5 \times 8^0$
>
> $= 1 \times 4096 + 2 \times 512 + 3 \times 64 + 4 \times 8 + 5 \times 1 = 5349$

> 十六进制的 `12345` 意味着（转为十进制）：
>
> $1 \times 16^4 + 2 \times 16^3 + 3 \times 16^2 + 4 \times 16^1 + 5 \times 16^0$
>
> = ... 不妨自己试一试?

> 二进制的 `12345` 呢?
>
> ——坑，**二进制中没有2及以上的数字**：因为逢2进位，十进制的 `2` 在二进制中表示为 `10`。

现在，你学会了十进制与八进制、十六进制的转换，与二进制如何转换也难不倒你了。

动手练习：进制转换

请动手在草稿纸上试一试、练一练，在二进制与十进制之间的转换。

编程练习3：进制转换函数

> 请分别设计函数，实现二进制、八进制、十六进制、十进制之间的相互转换。

补充一点有意思的：

二进制 与 八进制、十六进制 之间的转换非常简单，远远不需要像十进制与它们之间的转换那么麻烦。

> **请思考：** 为什么简单?
>
> ——**请想一想二、八、十六之间的关系，可以如何简化?**

---

3. 求水仙花数，了解枚举和迭代

背景知识：

> `水仙花数` 指的是一个 `n 位数`，它的每个位上的数字的 `n 次幂之和` 等于它本身。例如，`153` 是一个 `水仙花数`，因为 $1^3 + 5^3 + 3^3 = 153$。

> `水仙花数` 又称为 `阿姆斯特朗数` `Armstrong Number` 得名于美国数学家 `迈克尔·D·阿姆斯特朗` `Michael D. Armstrong`。他在1969年的一篇论文中提到了这个概念。
>
> 虽然 `水仙花数` 以 `迈克尔·D·阿姆斯特朗` 的名字命名，但他并不是首个研究这个概念的人。早在1917年，印度数学家 `斯里尼瓦瑟·拉马努金` `Srinivasa Ramanujan` 就提到了类似的数学问题。

题目要求：

> - 编写一个函数 `isArmstrongNumber(number:int)`，判断传入的参数 `number` 是否是水仙花数。如果是，返回 `True`，否则返回 `False`；
> - 调用这个函数，判断 `10000` 以内的水仙花数并输出。

> 求水仙花数 C 代码

思路分析：

很显然，又需要拆分数字了：

- 拆分数字的每一位；
- 将每一位的三次方求和；
- 将和与这个数本身进行比较;
- 注意范围：题目要求 `10000` 以内，结合水仙花数的定义，可以简单剪枝。

有了上一次`判断回文数`的经验，拆分数字有很多种方法。

这一次的练习，重点在复习和巩固的基础上，学习和练习 Python 特有的`推导式`，理解和掌握`迭代/可枚举类型`。

> 注意：部分编程语言有类似的方式实现类似推导式的功能，但 Python 将之内置在 “语言” 级别提供，而不需要依赖外部函数等。
>
> 所以，在享受 `Python 语法糖` 之余，一定要掌握 `通用方法`。

编程练习：注意对比以下各版本的差异

请先自行实现判断一个数字是否是水仙花数的函数， 再对照看下面的代码如何演进。

1. **版本 1.0：** 通用解法

```python
# 判断一个数字是否是水仙花数，这里默认按每位上的数字的三次方计算
def is_Armstrong_Number1(number:int, power:int = 3):
    # 将整数值的 number 转为字符串 digits
    digits = str(number)
    result = 0    # 这一句可以省略：Python 中不需要定义变量和赋予初始值
    for i in range(len(digits)):        # 循环，用 number 的长度作为循环次数
        item = int(digits[i])**power    # 计算每一位的 power 次方
        result += item                  # 累加

    return result == number    # 是否是水仙花数    # 累加
```

2. **版本 2：** 基于索引的取值方式改为对集合元素的直接访问

   将 `循环获取每一位数字字符并求 power 次幂后进行累加` 这部分改为

```python
for digit in digits:            # 迭代 number 的每一位
    item = int(digit)**power    # 计算每一位的 p
```

关于迭代和可枚举类型

> 迭代允许我们对一个数据集合中的每个元素执行相同的操作，或者按照一定的规则重复执行某段代码。迭代是一种在编程中常见且重要的概念，它可以帮助我们处理大量的数据、执行重复的任务或实现算法的逻辑。比如：
>
> `循环访问数据集的每一项元素` 又称为 `遍历数据集`；
>
> `遍历某个数据集并进行特定的处理`，这个过程叫做 `迭代` ；

遍历或者迭代数据集的方法，常见有两类：

- 用 `for i in range(n)` 循环，和 `item = items[i]` 用索引访问每一项；
- 用 `for item in items` 的方式遍历数据集 `items`，而不必用数值索引去访问数据集 `items` 的每一项：
  - `item` 已经是数据集的每一项，不必再用 `items[i]` 访问；
  - 数据集 `items` 属于 `可枚举类型` 才能用这样的方法；
  - Python 中的 `列表、元组、字典` 是 `可枚举类型`；
  - 注意：**部分语言不支持对可枚举类型的 for in 方式迭代，所以还是需要掌握通用方法。**

```python
# 判断一个数字是否是水仙花数，这里默认按每位上的数字的三次方计算
def is_Armstrong_Number2(number:int, power:int = 3):
    # 将整数值的 number 转为字符串 digits
    digits = str(number)

    result = 0    # 这一句可以省略：Python 中不需要定义变量和赋予初始值
    for digit in digits:           # 迭代 number 的每一位
        item = int(digit)**power   # 计算每一位的 power 次方
        result += item            # 累加

    return result == number    # 是否是水仙花数
```

3. **版本 3：** Python 推导式

   用 Python 推导式，将 `digits` 字符串中的每一位转换成数值再求 `power` 次幂。

```python
# 判断一个数字是否是水仙花数，这里默认按每位上的数字的三次方计算
def is_Armstrong_Number3(number:int, power:int = 3):
    # 将整数值的 number 转为字符串 digits
    digits = str(number)
    # 用推导式生成一个列表，列表中的每个元素是 number 的每一位的 power 次方
    list = [int(digit)**power for digit in digits]
    # digit 是式子中的临时变量

    result = 0    # 这一句可以省略：Python 中不需要定义变量和赋予初始值
    for item in list:           # 迭代列表中的每一个元素，对应 number 的每一位的 power 次
方
        result += item         # 累加

    return result == number    # 是否是水仙花数
```

4. **版本4：** 将迭代列表并累加求和部分，改为 Python 内置函数 `sum()`

   注意：`sum()` 函数可以对 `列表、元组、字典` 进行求和。——注意观察：它们是 `可枚举类型`，所以，……，请多想一想。

```python
# 判断一个数字是否是水仙花数，这里默认按每位上的数字的三次方计算
def is_Armstrong_Number4(number:int, power:int = 3):
    digits = str(number)
    list = [int(digit)**power for digit in digits] #推导式， digit 是式子中的临时变量
    result = sum(list)         # 用 Python 内置的 sum 函数计算列表中所有元素的和

    return result == number    # 是否是水仙花数素的和
```

5. **版本5：** 基于 `Python 语法糖` 的**极简 “优化”**

```python
# 判断一个数字是否是水仙花数：极简版本
def is_Armstrong_Number5(number:int, power:int = 3):
    return number == sum(int(digit)**power for digit in str(number))
```

特别注意：版本5 中的“极简”版本函数，在实际软件开发中并不提倡，主要原因如下：

- 不利于 “阅读” 代码，尤其　团队开发往往需要交叉评审，时间长了甚至自己也未必记得；
- 不利于 调试，比如加断点——不是不能加，是不便加；
- 不利于维护，因为后续维护代码的人未必是你，谁都害怕遇到 “屎山代码”。

所以，面对这类技巧——掌握它，可以用，但不要滥用，不要给自己和别人留下麻烦。

如果有人用来 “炫技”，记住：`不与夏虫语冰`，呵呵一笑而过。

参考上一节 《求回文数》中学习到的 `用数学方法拆分每一位上的数字` 的方法，重新实现求水仙花数的代码，请自行尝试，参考代码如下：

```python
def is_Armstrong_Number(number, power:int = 3):
    n = len(str(number))
    result = 0
    temp = number
    while temp > 0:
        digit = temp % 10         # 求末位上的数字
        result += digit ** power # 累加到结果变量
        temp //= 10               # 缩小十倍，以便求下一位
    return result == number
```

扩展编程：

> 请仿照 `sum()` 自行编写一个函数 `summation()`，用来对 `列表、元组、字典` 求和。记住：Python 中的 `列表、元组、字典` 是 `可枚举类型`。
>
> 再注意一下，`对 字典 求和`，与 `对 列表、元组 求和` 有什么不同之处？

---

4. 求阶和、阶乘，学习递推和递归

背景知识：

1. **阶和：**

> `阶和` 是指从 `1` 到某个正整数 `n` 的 `所有正整数的和`。
>
> `阶和` 可以用公式表示为：
>
> $1 + 2 + 3 + ... + n$

2. 阶乘：

> `阶乘` 是指一个正整数 `n` 与 `小于等于 n` 的所有 `正整数` 的 `乘积`。
>
> `阶乘` 可以用符号 `!` 表示，例如5的阶乘可以表示为 `5!`，其计算方法为：
>
> $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
>
> `阶乘` 在数学和计算机中经常被使用，常用于排列组合、概率统计、递归算法等领域。

编程练习：分别编写两个函数，计算 n 的阶和与阶乘

1. 递推方式：

> 求阶和阶乘 C 代码

```
# 计算 n 的阶和
def sum_of_Factorials(n):
    sum = 0
    for i in range(1, n+1):
        sum += i
    return sum

# 计算 n 的阶乘
def factorial(n):
    fact = 1
    for i in range(1, n+1):
        fact *= i
    return fact
```

例如，如果想计算数字 5 的阶和和阶乘，可以调用函数：

```
n = 5
print(sum_of_Factorials(n))   # 输出：15
print(factorial(n))           # 输出：120
```

2. 递归方式：

上面的代码中，解决问题所采用的是 递推 的思维方式。

> 递推是指根据已知的初始条件和递推公式，通过迭代计算来求解问题的方法。在递推中，问题的解是通过不断迭代计算前一项或多项得到的。

进一步思考，很显然，求 `5!` 先求 `4!`，因为 `5! = 4! × 5`，阶和类似。

那么，泛化推理可以得到结论：求 `n!` 就相当于先求 `(n-1)!` 然后乘以 `n`，可以表示为：

> $n! = f(n-1) \times n$

那么，在 `factorial()` 函数里，可以简化为：

```
# 计算阶和的递归函数
def sum_of_Factorials_Recursive(n):
    return n + sum_of_factorials(n-1)

# 计算阶乘的递归函数
def factorial_Recursive(n):
    return n * factorial_Recursive(n-1)
```

但因为参数 `n` 可能为负数，所以需要加上限制条件，否则就会无穷无尽、无法终止了。

```
# 计算阶和的递归函数
def sum_of_Factorials_Recursive(n:int):
    if n <= 0 return 0  # 终止递归（以后还要学习处理错误的方式）
    if n == 1:
        return 1
    else:
        return n + sum_of_Factorials_Recursive(n-1)

# 计算阶乘的递归函数
def factorial_Recursive(n:int):
    if n < 0 return 0 # 终止递归（以后还要学习处理错误的方式）
    if n == 1:
        return 1
    else:
        return n * factorial_Recursive(n-1)
```

于是，将求阶和、求阶乘的方式，从循环 `1 - n` 的递推方式，改为了 **"自己调用自己" 的递归方式。**

——对比可见，采用 `递归方式` 的代码逻辑，理解起来简单多了。

但一定注意：采用递归方式必须确保有 “终止条件”，否则就会陷入 “无穷无尽” 的 `无限递归`。

逻辑上的“无穷无尽”，在计算机编程中，`无限递归` 会导致运行程序的计算机为了避免耗尽内存资源而终止该程序——**你的程序将被强迫终止。**

特别注意：实际应用中，并不采用递归的方式来求解`阶和`、`阶乘`以及后面的`斐波那契数列`等等，有更优的方法。学习`递归`的主要目的是**掌握递归的思维方式和处理好其中的推出条件。**

> 递归是指一个函数在其定义中调用自身的过程。在递归中，函数通过将问题分解为更简单的子问题来解决复杂的问题。
>
> 递归通常包含两个要素：
>
> 1. 递归基础（base case）：定义递归的终止条件，当满足终止条件时，递归将停止。
> 2. 递归步骤（recursive step）：将原始问题分解为一个或多个更小的子问题，并通过调用自身来解决这些子问题。
>
> 递归在解决问题时可以提供简洁而优雅的解决方案，特别是对于那些可以分解为相同类型的子问题的问题。然而，递归也需要小心使用，确保终止条件正确设置，避免进入无限递归的循环。
>
> 需要注意的是，递归不仅限于函数调用自身，也可以是函数调用其他函数，只要函数最终可以通过一系列调用解决问题。

3. 对比递推与递归：

> 递推是指根据已知的初始条件和递推公式，通过迭代计算来求解问题的方法。在递推中，问题的解是通过不断迭代计算前一项或多项得到的。
>
> 递推与递归有以下区别和特点：
>
> 1. **调用方式**：递归是函数调用自身，而递推是通过循环迭代计算得到结果。
> 2. **结构**：递归是将问题分解为子问题并通过递归调用来解决，而递推是根据已知的初始条件和递推公式来计算得到结果。
> 3. **性能**：递推通常比递归具有更好的性能，因为递推不需要频繁地进行函数调用，而是通过循环迭代计算得到结果。
> 4. **空间复杂度**：递归在计算过程中需要使用函数调用栈来保存每一次递归调用的状态，可能会**占用较多的内存空间。**而递推只需要使用有限的变量来保存计算结果，空间复杂度相对较低。
> 5. **适用场景**：递归通常适用于问题可以自然地分解为子问题的情况，而递推则适用于问题可以通过迭代计算得到结果的情况。
>
> 总的来说，在实际应用中，需要根据问题的特点和需求选择适合的方法。

后续，继续用递推、递归的方式解决求斐波那契数列的练习。

> **单词助手：**
>
> - `recursion n. 递归`， `recursive adj. 递归的、递归式的`
> - `iteration n. 递推`， `iterative adj. 迭代的、循环的、递推的`
>
> > 注意：**虽然递推和迭代用同一个单词**，严格意义上，**递推不等于迭代**，**迭代也不是简单的循环**：
> >
> > - 迭代主要强调`重复执行特定的处理`，计算机编程领域往往用循环来实现迭代，所以`有时候会混用 循环 和 迭代 这两个概念`；
> > - 而`"递推" 则是一种数学或逻辑推理的方法`，通过已知的初始条件和递推关系式，逐步推导出后续的数值或解决方案。所以，`在某些情况下会混用 递推 和 迭代 这两个概念`。

---

5. 求斐波那契数列，学习递推和递归

背景知识：

> 斐波那契数列是一个经典的数学序列，以数学家列昂纳多·斐波那契（`Leonardo Fibonacci`）的名字命名。这个序列的前两个数字是0和1，后续的每个数字都是前两个数字之和。换句话说，第三个数字是前两个数字的和，第四个数字是前两个数字之和，以此类推。
>
> 斐波那契数列的前几个数字依次是`0、1、1、2、3、5、8、13、21、34`等等。这个序列在数学和计算机科学中具有广泛的应用。它的特点包括：
>
> 1. 递归性质：斐波那契数列的定义本身就是递归的，每个数字都是前两个数字之和。这种递归性质也可以通过递归函数来实现。
> 2. 快速增长：斐波那契数列的每个数字都比前一个数字大，增长速度非常快。随着序列的增长，相邻两个数字的比值趋近于黄金分割比例（约为1.618）。
> 3. 自相似性：斐波那契数列具有自相似性，即序列的一部分可以看作是整个序列的缩小版。例如，去掉前两个数字后剩下的部分仍然是一个斐波那契数列。
>
> 斐波那契数列在数学、编程、金融等领域都有广泛的应用，例如在算法设计、动态规划、金融建模、自然科学模型等方面。

题目要求：

> 请用递推和递归方式编写一个 Python 函数，接收一个整数 `n` 作为参数，返回斐波那契数列的前 `n` 个数字。
>
> - 用递推方式实现的函数名为 `fibonacci_iterative(n)`，返回一个包含斐波那契数列前 `n` 个数字的列表。
> - 用递归方式实现的函数名为 `fibonacci_recursive(n)`，返回一个包含斐波那契数列前 `n` 个数字的列表。

思路分析：

> - 斐波那契数列的前两个数字为 `0 和 1`，后续的每个数字都是`前两个数字之和`；
> - 递推方式需要使用循环来计算斐波那契数列的每个数字；
> - 递归方式需要使用函数自身来计算斐波那契数列的每个数字。

编程练习：

> 求斐波那契数列 C 代码

请自行编程实现，分别用递推和递归方式实现的参考代码如下：

1. **递推方式：**

```python
# 用递推方式求斐波那契数列
def fibonacci_Iterative(n:int):
    if n <= 0:
        return "输入的数必须是正整数"
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:    # 从 3 开始采用递推，小于 3 直接返回结果
        fib_sequence = [0, 1]
        for _ in range(3, n+1):
            fib_next = fib_sequence[-1] + fib_sequence[-2]
            fib_sequence.append(fib_next)
        return fib_sequence
```

函数 `fibonacci_iterative(n)` 使用循环迭代的方式来计算斐波那契数列的前 `n` 个数。它从初始的前两个斐波那契数开始，通过迭代计算得到前 `n` 个斐波那契数，并将它们存储在一个列表中返回。

2. **递归方式：**

```python
# 用递归方式求斐波那契数列
def fibonacci_Recursive(n:int):
    if n <= 0:
        return "输入的数必须是正整数"
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:    # 从 3 开始采用递归，小于 3 终止递归，直接返回结果
        fib_sequence = fibonacci_Recursive(n-1)
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
        return fib_sequence
```

函数`fibonacci_Recursive(n)`使用递归的方式来计算斐波那契数列的前`n`个数。它通过将问题分解为两个子问题：计算前 $n-1$ 个斐波那契数，并将结果存储在一个列表中，然后将最后两个数相加得到第`n`个斐波那契数，并将其添加到列表中返回。

请注意，递归函数中一定要设置终止递归的条件。本例中，当 $n<3$ 时，将不再“**自己调用自己**”从而终止继续递归。

可以通过调用这两个函数来计算斐波那契数列的前`n`个数。例如，计算斐波那契数列的前`10`个数，可以这样调用函数：

```python
n = 10
print(fibonacci_Iterative(n)) # 输出: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
print(fibonacci_Recursive(n)) # 输出: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

上面的例子中，可以用推导式简化列表部分，有兴趣的话请自行练习，就不展开了。

> 单词助手：
>
> - `iterative 迭代的、循环的、递推的`
> - `recursive 递归的、递归式的`

---

6. 辗转相除法求最大公约数，继续练习递推和递归

这一节继续练习递推和递归，用辗转相除法求两个数字的最大公约数。

背景知识：

> 辗转相除法（欧几里德算法）是一种用于计算两个整数的最大公约数的方法。
>
> 辗转相除法来自于古希腊数学家欧几里德 `Euclidean`，他在公元前300年左右的著作《几何原本》中首次描述了这个算法，因此也被称为欧几里德算法（`Euclidean algorithm`）。
>
> 据传，欧几里德的弟子乌克拉底 `Uclides` 向欧几里德请教如何找到两个数的最大公约数。欧几里德很快就给了他一个简单而巧妙的方法。他告诉乌克拉底，如果两个数能够整除，那么其中较小的数就是最大公约数。但是，如果两个数不能整除，那么我们用较小的数去除以较大的数，然后再用余数去除以刚才的较小数。这个过程一直重复下去，直到找到一个能整除的数，这个数就是最大公约数。
>
> 辗转相除法的思想是利用两个整数的除法运算和取余运算来逐步缩小问题的规模，直到找到最大公约数。这个算法的重要性在于它是一种高效的方法，即使在大整数的情况下也能很快地求解最大公约数。
>
> （小学阶段学习的求两个数的最大公因数的方法是`短除法`，可另行练习）

题目要求：

> 请编写一个函数 `gcd(a, b)`，其中 `a` 和 `b` 分别是两个整数。函数要求使用欧几里德算法（辗转相除法）求解 `a` 和 `b` 的最大公约数，并返回结果。 请确保函数的输入和输出与上述要求一致。

思路分析：

> 欧几里德算法的关键在于不断用较小的数去除以较大的数，然后用余数去除以刚才的较小数，直到找到一个能整除的数，该数即为最大公约数。 可以使用循环或递归来实现欧几里德算法。

编程练习：递推与递归

> 求最大公约数 C 代码

请自行实现，参考代码如下，分别用递推和递归方式：

```python
# 用递推方式实现欧几里德算法
def gcd_Iterative(a:int, b:int):
    if a < b:
        a, b = b, a  # 交换 a 和 b 的值，确保 a 大于等于 b
    while b != 0:    # 循环的终止条件
        a, b = b, a % b
    return a

# 用递归方式实现欧几里德算法
def gcd_Recursive(a:int, b:int):
    if a < b:
        a, b = b, a   # 交换 a 和 b 的值，确保 a 大于等于 b
    if b == 0:        # 递归的终止条件
        return a
    else:
        # 将 b 和 a 除以 b 的余数，作为新的 'a, b' 递归调用自己
        return gcd_Recursive(b, a % b)
```

这两个函数使用递归的方式来计算两个整数 a 和 b 的最大公约数：

> 在开始之前，先判断了 `a` 和 `b` 的大小关系，并在需要时交换它们的值，以`确保 a 大于等于 b`。**注意：确保严谨。**
>
> 然后，在递推的循环，或递归过程中，反复将较大的数除以较小的数的余数，直到`余数为 0`，此时较小的数即为最大公约数。

```python
a = 30
b = 45
print(gcd_Iterative(a, b))   # 输出：15
print(gcd_Recursive(a, b))   # 输出：15
```

编程拓展：

注意：在上面的函数中，交换两个数值采用了 Python 特有的语法：

`a, b = b, a` 对应标准的编程方式应该引入中间变量，为：

```python
c = a
a = b
b = c
```

思考：对于**两个正整数**，是否可以不引入新的变量完成交换？

先不要看下面的答案，利用数学特性：两个数的和，减去其中一个得到另外一个。

```
a = 10
b = 20

a = a + b
b = a - b
a = a - b

print(a)  # 输出：20
print(b)  # 输出：10
```

以后，还会学到另外一种计算方法：`异或`，先简单看一下：

```
a = 10
b = 20

a = a ^ b
b = a ^ b
a = a ^ b

print(a)  # 输出：20
print(b)  # 输出：10
```

在这个方法中，使用了异或运算符 `^` 来完成交换。

> 异或运算符有一个特性：对于两个相同的数进行异或运算会得到 0。
>
> 所以，通过连续进行异或运算，可以完成两个变量的交换。
>
> 首先，将 a 与 b 进行异或运算，结果存储在 a 中； 然后，再次将 a 与 b 进行异或运算，这时 a 存储的是原先的 b 的值； 最后，再次将 a 与 b 进行异或运算，这时 a 存储的是原先的 a 的值。
>
> 经过这三次异或运算，a 和 b 的值完成了交换。

请注意，这种方法**只适用于正整数，对于其他类型的数据可能会产生不可预测的结果**。

---

7. 递归求解汉诺塔问题

背景知识：

> 汉诺塔问题源于古代印度的一个传说。传说中，有一个古老的寺庙里放着三根柱子，柱子上有64个大小不同的金盘。这些金盘从小到大按顺序叠放在一起，最大的盘子在最底下，最小的盘子在最上面。寺庙里的僧侣们每天都在默默地进行着一项神圣的任务，他们要将这64个金盘从寺庙的一个柱子上移动到另一个柱子上，但是要遵守以下规则：
>
> 1. 每次只能移动一个盘子；
> 2. 移动过程中，大的盘子不能放在小的盘子上面；
> 3. 可以借助第三根柱子作为中转。

题目要求：

> 编写一个 Python 函数 `hanoi_Recursive(n, source, target, auxiliary)`，用来求解汉诺塔问题。函数的参数如下：
>
> - `n`：整数，表示金盘的数量，范围为 1 到 64；
> - `source`：字符串，表示起始柱子的名称；
> - `target`：字符串，表示目标柱子的名称；
> - `auxiliary`：字符串，表示辅助柱子的名称
>
> 函数要求按照规则将金盘从起始柱子移动到目标柱子，并输出每次移动的步骤。
>
> > 单词助手：`source 来源、起源、原始`；`target 目标、靶子`；`auxiliary 辅助的、附加的`

调用示例：

```
n = 3
source = "A"
target = "C"
auxiliary = "B"

print("递归实现：")
hanoi_Recursive(n, source, target, auxiliary)
```

输出：

> 移动 1 号盘，从 A 到 C 移动 2 号盘，从 A 到 B 移动 1 号盘，从 C 到 B ... ...

思路分析：可以使用递归的方式来解决汉诺塔问题。

> 每次递归调用时，将问题拆分为三个子问题：
>
> 1. 将 n-1 个金盘从起始柱子移动到辅助柱子。
> 2. 将第 n 个金盘从起始柱子移动到目标柱子。
> 3. 将 n-1 个金盘从辅助柱子移动到目标柱子。
>
> 递归的基本情况是当金盘的数量为 1 时，直接将它从起始柱子移动到目标柱子。

编程练习：

> 求解汉诺塔问题 C 代码

请自行实现函数 `hanoi_Recursive()`，参考代码如下：

```
# 递归方式求解汉诺塔问题，source 是起点柱，target 是目标柱，auxiliary 是辅助柱
def hanoi_Recursive(n:int, source, target, auxiliary):
    if n == 1:
        print(f"移动 1 号盘，从 {source} 到 {target}")
    else:
        # 这里如何注释？
        hanoi_Recursive(n-1, source, auxiliary, target)
        print(f"移动 {n} 号盘，从 {source} 到 {target}")
        # 这里如何注释？
        hanoi_Recursive(n-1, auxiliary, target, source)
```

在每次递归调用时，先检查盘子的数量。如果只有一个盘子，直接将它从起始柱子移动到目标柱子。否则，递归地调用函数来解决两个子问题：

1. 将 $n-1$ 个盘子从起始柱子移动到辅助柱子，将第 n 个盘子从起始柱子移动到目标柱子。
2. 然后，再次递归地调用函数来将 $n-1$ 个盘子从辅助柱子移动到目标柱子。

总结：

用递归方式求解汉诺塔问题，是非常好的用递归思维解决问题的例子，化繁为简。在实际的编程竞赛中，不仅仅要求会编写出递归方式求解的代码，还要求能够推理、演算出每一步。比如，**问 $n=5$ 时，程序输出的内容是什么？变量 A 的值是多少？** 等等。

这时，就会涉及到更深入的理解：计算机内部是如何实现递归的调用的？

——这将在后续练习中进一步加强。

知识拓展：分治与递归

综上，汉诺塔问题被拆分为两个步骤：

- 将当前盘子从起始柱子移动到辅助柱子；
- 将当前盘子从辅助柱子移动到目标柱子

然后，递归调用 $n-1$ 号盘子。

这是典型的采用 `分治` 的思维方式处理问题：

> **分治：**
>
> 在软件开发中，"分治" 是一种思维方法，指的是 `将一个复杂的问题划分为多个相互独立且较小的子问题，然后将这些子问题分别解决，并最终将它们的结果合并起来得到原问题的解决方案` 。这种方法的核心思想是 `将大问题分解为小问题来解决` ，从而简化问题的复杂性。注意， `分治` 与 `分解任务` 略有不同。
>
> **分治算法通常会使用递归来实现**：
>
> 在分治算法中， `问题的划分和解决过程` 通常会使用 `递归` 的方式进行。具体来说，分治算法将问题划分为多个子问题，然后递归地对每个子问题进行解决，最后将子问题的解合并起来得到原问题的解。递归在分治算法中的使用可以简化问题的处理过程，并且使得代码更加清晰和易于理解。
>
> 总结来说， `分治是一种思维方法` ， `递归是一种实现方式` ，分治使用递归来解决问题。
>
> > 思考和分析的时候如此，在实际执行和实现环节，出于 `性能或空间优化` 的目的，往往会 `用 递推 代替 递归` 。

除了软件开发会用到 `分治` 的思维方式，生活中又何尝不是呢？

分治、分解、递推、递归无处不在。

——要注意区分它们之间的细微区别。

编程拓展：

在上面的例子中，使用了一个参数 `auxiliary` ，是否可以省略这个参数？

> 假如，我们把 `A、B、C` 三个柱子换成 `1 号 2 号 3 号`，用编号表示。 那么，已知任意两个柱子 `x，y` 的编号，例如 `x = 1，y = 3` 可以知道第三个柱子 `z` 的编号一定是：`6 - x - y = 2`。
>
> 于是，代码可以变为：

```
# 递归方式求解汉诺塔问题，source 是起点柱，target 是目标柱
def hanoi_Recursive2(n:int, source:int, target:int):
    if n == 1:
        print(f"移动 1 号盘，从 {source} 号柱到 {target} 号柱")
    else:
        hanoi_Recursive2(n-1, source, 6 - source - target)
        print(f"移动 {n} 号盘，从 {source} 号柱到 {target} 号柱")
        hanoi_Recursive2(n-1, 6 - source - target, target)

n = 3
source = 1
target = 3

print("递归实现：")
hanoi_Recursive2(n, source, target)
```

---

8. 求解迷宫问题，初步学习回溯

题目要求：

> 有一个字符组成的迷宫，需要找到从起点到终点的路径，路径只能由相邻的空间组成。
>
> 如下面的示例数据 `maze`： `0` 表示没有障碍物的 `空间`， `1` 表示障碍物 `墙壁`。
>
> 请用递归方式编写一个 Python 函数，解决走迷宫问题。
>
> - 函数名为 `solve_Maze(maze, start, end)`，接收三个参数：
>   - `maze` 是代表迷宫的二维列表，其中 `0` 表示 `空间`， `1` 表示 `墙壁`；
>   - `start` 是表示 `起点位置` 的元组，例如 `(x, y)`，其中 `x` 和 `y` 分别表示 `起点` 的 `行` 和 `列`。
>   - `end` 是表示 `终点位置` 的元组，例如 `(x, y)`，其中 `x` 和 `y` 分别表示 `终点` 的 `行` 和 `列`。
> - 函数应该返回一个布尔值，表示是否存在从起点到终点的路径；
> - 扩展：请输出找到的路径的坐标序列。
>
> 单词助手： `maze 迷宫`， `solve 解决、求解、解答`

示例数据和调用代码：

```
# 迷宫示例，0表示空格，1表示墙壁
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0]
]

start = (0, 0)  # 起点位置
end = (4, 4)  # 终点位置

# 调用解答函数
if solve_Maze(maze, start, end):
    print("存在从起点到终点的路径")
else:
    print("不存在从起点到终点的路径")
```

思路分析：

> 用递归来解决走迷宫问题。
>
> - 从起点开始，检查当前位置是否为终点，如果是，则返回 `True` 表示找到了一条路径。否则，我们尝试向上、向下、向左和向右四个方向移动，并递归调用函数来探索下一个位置。
>   - 如果任何一个方向上找到了一条路径，则返回 `True`。
>   - 如果所有方向都没有找到路径，则返回 `False`。
> - 在递归函数中，需要考虑边界情况，如迷宫的边界和墙壁的位置。还需要标记已经访问过的位置，以避免陷入无限递归的循环中。

编程练习：

> 求解迷宫问题 C 代码

下面是递归方式的解答和调用示例代码，请自行编程实现再参考：

```python
def solve_Maze(maze, start, end):
    # 获取迷宫的行数和列数
    rows = len(maze)
    cols = len(maze[0])

    # 检查当前位置是否在迷宫范围内
    if start[0] < 0 or start[0] >= rows or start[1] < 0 or start[1] >= cols:
        return False

    # 检查当前位置是否为墙壁或已经访问过
    if maze[start[0]][start[1]] == 1 or maze[start[0]][start[1]] == -1:
        return False

    # 检查当前位置是否为终点
    if start == end:
        print("出口: ", end)
        return True

    # 标记当前位置为已访问
    maze[start[0]][start[1]] = -1
    print("走到: ", start)

    # 尝试向上、向下、向左和向右四个方向移动
    if solve_Maze(maze, (start[0] - 1, start[1]), end):  # 向上移动
        return True
    if solve_Maze(maze, (start[0] + 1, start[1]), end):  # 向下移动
        return True
    if solve_Maze(maze, (start[0], start[1] - 1), end):  # 向左移动
        return True
    if solve_Maze(maze, (start[0], start[1] + 1), end):  # 向右移动
        return True

    # 没有找到路径，返回 False
    return False
```

这段代码通过将迷宫的状态进行标记，避免重复访问和死循环。

在调用示例中，给出了一个迷宫的示例，起点为 `(0, 0)` ，终点为 `(4, 4)` 。程序会打印出是否存在从起点到终点的路径。

知识拓展：回溯

本题中，为了在走迷宫的过程中避免重复，采用了 `回溯` 。

> **回溯**
>
> 在编程中，回溯是一种算法思想，用于解决在搜索问题中找到所有可能解的情况。
>
> 它通过尝试每一种可能的选择，并**在发现当前选择无法得到正确解时返回上一步进行其它选择**，以找到所有可能的解。
>
> 在走迷宫问题中，从起点开始，尝试向上、向下、向左和向右四个方向移动，如果某个方向可以走则继续前进，如果某个方向无法继续前进则返回上一步，尝试其它方向。这样不断地尝试和回退，直到找到一条通往终点的路径或者所有可能的路径都被尝试过。
>
> 回溯在走迷宫问题中的关键是在每一步尝试之后记录当前位置（本例中利用标记达到同样目的），一旦遇到墙或已经走过的位置，则一层层 `返回到有其它未尝试过的选项的那一层` ——这里是借助 `递归` 实现的，其原理在后续讲解 `递归的实现原理` 和 `栈` 这一数据结构时将讲到。
>
> 因此，回溯算法可以 `穷尽` 所有可能的路径，找到通往终点的路径——如果存在的话。当然，修改退出的判断条件，也可以把所有可能存在的路径都找出来。

> 总结来说，**回溯是一种搜索算法**，用于解决搜索问题中找到所有可能解的情况。

知识拓展：深度优先搜索、广度优先搜索

> `深度优先搜索` 和 `广度优先搜索` 是两种常用的解决走迷宫问题的**搜索策略**。
>
> **深度优先搜索** `Depth First Search` DFS

> `深度优先搜索` 是一种 **先深后广** 的 **搜索策略**。
>
> 它从起点开始，选择一个方向尽可能深入地探索，直到无法继续前进时才返回上一步，继续探索其他方向。比如，约定一个规则 "逢路口先尝试左转"，直到走不通则返回上一个路口试试看右转——按这个规则递归执行。
>
> `DFS` 使用 `栈` 来保存当前路径，以便在回溯时能够返回上一步。 `DFS` 的特点是能够快速地到达远离起点的区域，但可能会陷入无限循环或者过早地走入死胡同。而且， `DFS` 找到的第一个解，未必是最优解——迷宫问题里是最短路径。

> **广度优先搜索** `Breadth First Search` BFS
>
> `广度优先搜索` 是一种 **先广后深** 的 **搜索策略。**
>
> 它从起点开始，依次探索与当前位置相邻的所有未访问过的位置，然后再探索与这些相邻位置相邻的位置，以此类推，直到找到终点或者遍历完所有可达的位置。
>
> `BFS` 使用队列来保存当前层级的位置，以便按照层级顺序进行探索。BFS的特点是能够找到最优解——迷宫问题里是最短路径，但可能会占用较多的内存空间。

> **与回溯的关系**
>
> 回溯算法中，可以用 `DFS` 也可以用 `BFS` 作为**搜索策略**，取决于目的：最快找到解法，还是找到最优解。
>
> 在走迷宫问题中，先尝试下一层，还是先向左右方向尝试后再进入下一层，就是不同的搜索策略。

编程拓展：尝试用广度优先的方式求解迷宫问题

请先尝试自行解答，参考代码如下：

```python
def solve_Maze_BFS(maze, start, end):
    # 获取迷宫的行数和列数
    rows = len(maze)
    cols = len(maze[0])

    # 定义用于表示每个空间是否被访问过的二维列表
    visited = [[False] * cols for _ in range(rows)]

    # 定义用于表示每个空间的前驱空间的二维列表
    prev = [[None] * cols for _ in range(rows)]

    # 定义用于表示每个空间到起点的距离的二维列表
    distance = [[float('inf')] * cols for _ in range(rows)]

    # 定义方向向量，用于计算相邻空间的位置
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    # 获取起点和终点的行列坐标
    start_row, start_col = start
    end_row, end_col = end

    # 将起点标记为已访问，并将距离设为0
    visited[start_row][start_col] = True
    distance[start_row][start_col] = 0

    # 创建一个列表，并将起点加入队列
    queue = []
    queue.append(start)

    # 使用广度优先搜索算法遍历迷宫
    while queue:
        current_row, current_col = queue.pop(0)

        # 如果当前位置是终点，则找到了路径，返回True
        if current_row == end_row and current_col == end_col:
            return True

        # 遍历所有相邻的空间
        for direction in directions:
            next_row = current_row + direction[0]
            next_col = current_col + direction[1]

            # 如果下一个位置不在迷宫范围内或者是墙壁，则跳过
            if next_row < 0 or next_row >= rows or next_col < 0 or next_col >= cols or
maze[next_row][next_col] == 1:
                continue

            # 如果下一个位置未被访问过，则将其加入队列，并更新距离和前驱空间
```

```
            if not visited[next_row][next_col]:
                visited[next_row][next_col] = True
                distance[next_row][next_col] = distance[current_row][current_col] + 1
                prev[next_row][next_col] = (current_row, current_col)
                queue.append((next_row, next_col))

    # 如果遍历完所有空间后没有找到路径，则返回False
    return False

# 示例数据和调用代码：
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0]
]

start = (0, 0)  # 起点位置
end = (4, 4)  # 终点位置

# 调用解答函数
if solve_Maze_BFS(maze, start, end):
    print("存在从起点到终点的路径")
else:
    print("不存在从起点到终点的路径")
```

补充说明：走迷宫与八皇后

从编程学习的角度来看，走迷宫问题和八皇后问题都是经典的回溯算法问题，但在解决方法和难度上有一些不同。

1. 解决方法：
    - 走迷宫问题是在一个迷宫中找到从起点到终点的路径。解决这个问题需要在迷宫中尝试不同的路径，通过回溯算法进行探索，直到找到一条通向终点的路径或者确定不存在路径。
    - 八皇后问题要求在一个 8x8 的棋盘上放置 8个皇后，使得它们互相不能攻击到对方。解决这个问题需要考虑每个皇后的位置，以确保它们不在同一行、同一列和同一对角线上。
2. 难度：
    - 走迷宫问题相对来说比较容易，因为只需要找到一条从起点到终点的路径即可。解决走迷宫问题同样需要运用回溯算法，但通常情况下不需要像八皇后问题那样复杂的约束条件。

- 八皇后问题相对来说略为复杂，因为需要考虑多个皇后之间的互斥关系，并且需要满足一定的约束条件。解决八皇后问题需要运用回溯算法和剪枝技巧，设计合适的数据结构和算法来搜索解空间。

总的来说，走迷宫问题和八皇后问题都是很好的练习`回溯算法`和解决问题能力的题目。

通过解决这些问题，可以提高对算法思维和编程技巧的理解，并培养解决复杂问题的能力。

- 在非算法类的比赛中属于较高级别才会要求的能力；
- 在算法竞赛类的编程比赛则属于必须掌握的基本能力。

---

9. 求解八皇后问题，继续练习回溯（草稿）

背景知识

> 八皇后问题是一个经典的数学和计算机科学问题，旨在找到在一个8x8的棋盘上放置8个皇后，使得它们互相不能攻击到对方的情况下，所有皇后的位置组合。
>
> > 在国际象棋中，皇后是最强大的棋子，可以在横、竖和对角线上任意距离移动。因此，八皇后问题要求在8x8的棋盘上放置8个皇后，使得每个皇后都不在同一行、同一列和同一对角线上。
> >
> 由于皇后的特殊移动能力，这个问题本质上是一个组合优化问题。解决八皇后问题的一种常见方法是使用回溯算法。回溯算法通过尝试不同的解决方案，并进行适当的剪枝，直到找到所有满足条件的解决方案。

八皇后问题是一个经典的练习问题，它有助于提高问题解决和编程技巧。通过解决这个问题，可以加深对回溯算法和组合优化问题的理解。

题目要求：

> 请用递推和递归方式编写一个函数，解决八皇后问题：
>
> - 用递推方式实现的函数名为`eight_queens_iterative()`，它应该返回一个包含所有可行解的列表。每个可行解都是一个列表，其中每个元素表示每行皇后所在的列号。
> - 用递归方式实现的函数名为`eight_queens_recursive()`，它应该返回一个包含所有可行解的列表。每个可行解都是一个列表，其中每个元素表示每行皇后所在的列号。

思路分析：

> - 要确保每个皇后在同一行、同一列和同一对角线上都没有其他皇后；
> - 递推方式可以使用循环来尝试不同的列号，逐行放置皇后；
> - 递归方式可以使用递归函数来尝试不同的列号，逐行放置皇后。

编程练习：

当解决八皇后问题时，递推方式和递归方式的实现略有不同。

> 求解八皇后问题 C 代码

下面是递推和递归方式的解答和调用示例代码，请自行编程实现再参考：

> 单词助手：
>
> - `stack 栈`——一种基础的数据结构，类似列表
> - `solution 解决方案、解答`

1. **递推方式：**

```
def eight_Queens_Iterative():
    solutions = []
    stack = [(0, [])]  # 使用栈来存储每一行的状态，每个元素为 (row, queens)，其中 row 表示当前行数，queens 表示已放置皇后的列号列表

    while stack:
        row, queens = stack.pop()

        if row == 8:  # 找到一个解
            solutions.append(queens)
        else:
            for col in range(8):
                if all(col != q and abs(row - i) != abs(col - q) for i, q in enumerate(queens)):
                    stack.append((row + 1, queens + [col]))

    return solutions

solutions = eight_queens_iterative()
for solution in solutions:
    print(solution)
```

2. **递归方式：**

```
def eight_Queens_Recursive(row=0, queens=[]):
    if row == 8:
        return [queens]

    solutions = []
    for col in range(8):
        if all(col != q and abs(row - i) != abs(col - q) for i, q in
enumerate(queens)):
            solutions.extend(eight_Queens_Recursive(row + 1, queens + [col]))

    return solutions

solutions = eight_Queens_Recursive()
for solution in solutions:
    print(solution)
```

调用函数的示例代码：

```
# 调用递推方式的八皇后问题解答函数
solutions_Iterative = eight_Queens_Iterative()
print("递推方式解答八皇后问题:")
for solution in solutions_Iterative:
    print(solution)

# 调用递归方式的八皇后问题解答函数
solutions_Recursive = eight_Queens_Recursive()
print("递归方式解答八皇后问题:")
for solution in solutions_Recursive:
    print(solution)
```

注意：八皇后问题的解决方案可能有多个，每个解决方案都是一个列表，其中每个元素表示每行皇后所在的列号。因此，可能会有多行输出。
