第 6 讲　常见算法

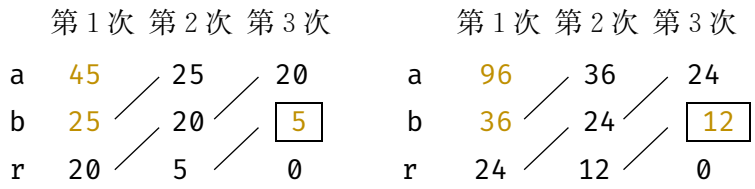
一　使用 while 循环的一些算法

1. 求最大公约数

- 若一个数能同时被两个数整除，则称其为两个数的公约数，公约数可能有多个，最大的称为**最大公约数**
- 一种常用的求两个数的最大公约数的方法是辗转相除法：

> - 首先用较大的 a 整除较小的 b，得到第一个余数
>
> 然后用 b 整除第一个余数，得到第二个余数　　余数会越除越小
>
> 然后用第一个余数整除第二个余数，以此类推，直至余数为 0，这一次的除数（较小的数）就是最大公约数

例：




```
def gcd(a, b):
    if a < b: a, b = b, a   # 保证 a 大于 b
    r = a % b
    while r > 0:
        a = b          # 将 a 更新为下一轮较大的数（上一轮的 b）
        b = r          # 将 b 更新为下一轮较小的数（上一轮的 r）
        r = a % b      # 计算余数
    return b           # 退出 while 时 r 为 0，即上一轮的 b 为最大公约数
```

2. 十进制与二进制转换

- 二进制数 $a_{n-1}\cdots a_2a_1a_0$（a_i 为 1 或 0）对应的十进制数 m 为

$$m=a_{n-1}\times 2^{n-1}+\cdots+a_1\times 2^1+a_0\times 2^0$$

由于前 $n-1$ 项都是 2 的倍数，因此 m 整除 2，余数部分必为 a_0，商为

$$m_1=a_{n-1}\times 2^{n-2}+\cdots+a_1\times 2^0$$

同理，再一次整除 2，余数部分必为 a_1，商为 $m_1=a_{n-1}\times 2^{n-3}+\cdots+a_2\times 2^0$

以此类推，余数将依次为 a_2，a_3……直至 a_{n-1}，此时商为 0

将获得的余数组合起来，就是二进制数的表示

```
def binary(m):
    r = ''                          # 存放余数的字符串
    while m > 0:
        r = str(m % 2) + r          # 获得余数转换字符形式存放到 r 中
        m = m // 2
    return r[::-1]                  # 将字符串 r 逆序
```

二　使用 for 循环的一些问题求解

1.数列求和

· 求一串数列 $a_0+a_1+\cdots+a_{n-1}$ 的和

· 基本思路：观察数列总结出通项公式和下标范围后，通过 for 循环计算出各项并相加

```
s = 0                 # 累加问题中需要一个变量记录累加结果
for i in range(n)     # i 遍历下标 0 ~ n-1
    a = _________     # 计算出单项值
    s += a            # 累加，循环结束后 s 的值就是我们需要的和
```

· 在 python 中，这个问题可以用更短的代码解决：

```
s = sum(______ for i in range(n))  # 通过列表推导式得到各项的值然后直接相加
```

2.枚举素数

· 素数又称质数，是只能被 1 和自身整除的整数

· 判断一个数 n 是否是质数的方法：依次整除 2 至 n−1，如果有能被整除的，n 就不是素数，否则就是

```
def is_prime(n):
    for i in range(2, n):       # i 将遍历 2 至 n-1
        if n % i == 0:
            return False        # 如果被整除，说明不是素数，直接返回 False
    return True                 # 执行到这里说明一个都没被整除，是素数，返回 True
```

思路优化：数学上，如果小于 $\sqrt{n}$ 的数都无法整除 n，那么大于 $\sqrt{n}$ 的数也无法整除 n

因此 `range` 的范围可以只到 `sqrt(n)`，减少一大半的循环量

· 枚举 a 到 b（不含）范围内的所有素数：依次对各个数调用上面的函数即可

```
s = []
for i in range(a, b):       # i 将遍历 2 至 n-1
    if is_prime(i):
        s.append(i)         # 素数将被添加进列表中
```

思路优化：如果一个数不是素数，那么它一定能分解成素数的乘积

在判断 n 是否为素数时，我们已经得到了比它小的所有素数的列表

如果这些素数都不能整除 n，说明 n 是素数，这样我们就无需遍历 2~sqrt(n)的每一个整数

```
def is_prime(n, s):             # s 是素数列表
    for i in s:
        if n % i == 0:
            return False        # 如果被整除，说明不是素数，直接返回 False
    return True                 # 执行到这里说明一个都没被整除，是素数，返回 True
```

3. 打印图案

- 此类问题要求打印出字符或数字组成的图形
- 一般思路：将图形拆分成行，观察每行的字符组成，得到其与行数的关系

  然后，要么通过两层嵌套循环输出，要么直接生成一行字符，通过一轮循环输出

  例如，上面这个图形有 n 行，第 i 行要输出 2i+1 个*号，两边各输出 n−i 个空格

```
for i in range(n):
    print(' ' * (n-i) + (2*i+1) * '*' +  ' ' * (n-i))
```

4. 杨辉三角

- 杨辉三角如图所示，它的特点是每行首末数字都是 1，中间的数字是上方数字和左上数字之和

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
……
```

- 此类问题通过创建二维列表求解：创建一个 n 行二维列表，其中第 i 行列表有 i 个元素，且初始化为 0

  然后将每行列表的第 1 个和第 i 个元素改为 1

```
matrix = []
for row in range(n):
    line = [0] * (row + 1)        # 下标为 row 的列表是第 row + 1 行
    line[0] = line[row] = 1       # 第 1 个和最后 1 个元素改为 1
    matrix.append(line)           # 添加到列表中
```

  然后开始处理中间的元素，第 i 行第 1～i−1 个元素需要处理，共 i−2 个元素

  显然，我们只需要从第三行开始处理

```
for row in range(2, n):
    for i in range(1, row):
        # 当前计算的是 row 行 i 列的元素，等于上一行（row-1）的同位置（i）和前一位置（i-1）之和
        matrix[row][i] = matrix[row-1][i] + matrix[row-1][i-1]
```

两个部分可以合并，从而在单个嵌套循环内完成

```
matrix = []
for row in range(n):
    line = [0] * (row + 1)
    line[0] = line[row] = 1
    for row in range(2, n):
    for i in range(1, row):
        line[i] = matrix[row-1][i] + matrix[row-1][i-1]
    matrix.append(line)
```

三　查找算法

在一组数据中查找某个数是否存在，并返回它的位置

1. 线性查找

· 从第 1 个数据开始，依次比较数据和给定的查找值，如果相同则找到，并得到该数据的位置

```
found = -1
for i in range(len(a)):       # a 是存储数字的列表
    if key == a[i]:
        found = i             # 如果找到了，就更新 found
        break
print(found)
```

因为要找出下标，所以迭代变量选择下标而非元素本身，实际上，python 可以同时迭代这两个变量：

```
found = -1
for i, item in enumerate(a):       # a 是存储数字的列表
    if key == item:
        found = i             # 如果找到了，就更新 found
        break
print(found)
```

2. 二分查找

· 二分查找需要**数据已经按照大小排序**，其每轮查找都将 key 与最中间位置的数据做比较

如果 key 等于中间值，那么查找结束

如果 key 大于或小于中间值，说明这个数如果存在，一定在该中间值的某一侧，从而搜索范围减半

```
left, right = 0, len(a)-1           # a 是存储数字的列表，由小到大排序
found = -1
while left <= right:                # left~right 是搜索下标范围
    mid = (left+right) // 2         # 获得中间位置
    if a[mid] > key:
        right = mid -1              # 中间值大于 key，则右侧的所有数都被排除
    elif a[mid] < key:
        left = mid + 1              # 中间值小于 key，则左侧的所有数都被排除
    else:
        found = mid
        break
```

3. 查找最大/小值

· 基本思路：设第一个值为最大/小值，记录在一个变量中

从第二个值开始，依次与最值比较，如果比最值更最，就更新最值变量（变成这个值）

· 结果：所有数据比完后，最值变量存储的就是最值

```
max_idx = 0                     # 记录最大值下标的变量
for i in range(len(a)):
    if a[i] > a[maxidx]:
        max_idx = i             # 更新最值
```

四　排序算法

· 将一组顺序混乱的数据按照大小顺序重新排列

1. 选择排序

一组从大到小排序的数据中，第 1 个数是所有数中最大的，第 2 个数是除第 1 个数外最大的，……，以此类推，第 i 个数是第 i 至最后一个数中最大的。选择排序的思路，就是依次找出最大，第 2 大，……的数据并放在对应的位置，从而实现排序。**前 i 个数排好序放在前 i 个位置后，剩余数中的最大值，就是第 i+1 大的数，放在第 i+1 个位置**。

· 从大到小排序：找出 max 放在前面 或 找出 min 放在后面

从小到大排序：找出 min 放在前面 或 找出 max 放在后面

```
n = len(a)
for i in range(n):
    max_idx = i
    for j in range(i, n):        # i 至 n-1 为未排序的数，这之中的最大值是第 i 大的数
        if a[j] > a[max_idx]:
            max_idx = j
    a[i], a[max_idx] = a[max_idx], a[i]    # 第 i 大的数应该放在第 i 个位置
```

* 请关注排序过程中数据顺序的变化（例题和习题），这个是考试的常考点

2. 冒泡排序

冒泡排序的思路是：从第一个数开始，依次将各个位置的数与下一个数作比较，如果比下一个数大，就交换两个数的位置。由于较大的数会移动到下一个数的位置，又能参与下一次比较，因此最大的数一直在右移，最终移动到末尾，也就是最大的数应该在的位置。然后重复一轮，结果使第二大的数移动到正确位置，继续重复……最终所有数都会移动到对应的位置，完成排序。

```
n = len(a)
for i in range(n):
    for j in range(n-i):              # 此时有 i 个数已正确排在后面，前 n-i 个数没排好
        if a[j] > a[j+1]:
            a[j], a[j+1] = a[j+1], a[j]
```

· 冒泡排序的几个注意点：

① 从小到大排序，需要小的往前冒泡或者大的往后冒泡

从大到小排序，需要大的往前冒泡或者小的往后冒泡

② 往前冒泡要求内循环是从右往左迭代，即 `range(n-1, i, -1)`

· 同样也要熟悉排序过程中数字位置的变化（例题和习题），考试常考

五　递归算法

递归是指一个函数调用自身的行为，如果一个问题可以化解成求解多个小的同类型问题，就可以用递归

1. 阶乘

整数 n 的阶乘可以看成 $n! = n \times (n-1)!$，也就是化为了求 $(n-1)!$ 的问题，因此可以用递归

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n * fact(n-1)
```

如何理解递归的调用？比如现在调用 `fact(10)`：

→ 程序会建立一个 `fact(10)`的区域，跳转到该区域开始执行，执行至 `fact(n-1)`，也就是调用 `fact(9)`

→ 程序会再建立一个新的 `fact(9)`区域，跳转到该区域开始执行，然后执行到 `fact(n-1)`，也就是调用 `fact(8)`

→ ……

→ 直到调用 `fact(1)`，建立一个新的 `fact(1)`区域，跳转到该区域开始执行，执行到 `return 1`，该函数执行结束

→ 程序销毁 `fact(1)`区域，带着返回值 1 跳转回 `fact(2)`区域，继续执行 `fact(2)` 至 `return 2 * 1`

……

→ 程序销毁 `fact(9)`区域，带着返回值 `9!`跳转回 `fact(10)`区域，继续执行 `fact(10)` 至 `return 10 * 9!`

→ 程序销毁 `fact(10)`区域，带着返回值 `10!`回来

2. 斐波那契数列

该数列的特点是前两个数字为 1，1，后面开始第 i 位的数字满足 $a_i = a_{i-1} + a_{i-2}$，因此也能用递归

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

可以看到递归的代码思路很简洁，但是函数区域的建立是需要占用内存的。n 为 100 的时候，有 100 个函数空间同时存在，效率很低。一种解决方法是用字典或列表存储已经算过的斐波那契数，避免反复算同一个数

3. 排列组合枚举

现在有一个由 n 个不同的字符组成的字符串，列举出所有排列的情况。全排列的枚举思路是，对于 n 个字符的全排列，取 1 个字符放在首位，则剩下 n−1 个字符的全排列就是该字符位于首位的全部情况，依次取遍所有字符，就得到 n 个字符全排列的全部情况。

```
def perm(choice, selected=[])        # choice 是还没被选择的字母，selected 是已经被选择的
    if len(choice) == 0:
        print(''.join(selected))
    else:
        for i in range(len(choice)):
            perm(choice[:i] + choice[i+1:], selected.copy() + [choice[i]])
            #      将 choice[i]去掉后的字符串      选好的字符串上加上 choice[i]
```