1. 请完成以下程序设计, 并设计案例验证你的程序.

a.写个函数 sqr(x), 用纯加法实现 x^2 的计算(x 是整数). 你不能用乘法.

b.写个函数 cube(x), 计算 x^3， 函数中可调用 sqr(x)一次. 你不能用乘法， 可以用加法.

c. 写个函数 quad(x), 计算 x^4, 用最简单的方式完成， 函数中可以调用 sqr(x). 你不能用乘法或加法.

a.利用一个循环和加法解决：

```
def sqr(x):
    x = abs(x)
    res = 0
    for i in range(0, x):
        res += x
    return res


# test case
print("case 0:", sqr(5))
print("case 1:", sqr(1))
print("case 2:", sqr(0))
print("case 3:", sqr(-1))
print("case 4:", sqr(-3))
```

Shell:

```
case 0: 25
case 1: 1
case 2: 0
case 3: 1
case 4: 9

Process finished with exit code 0
```

b.将 sqr(x)函数用加法累加，注意下一三次方符号的问题就好了：

```
def sqr(x):
    x = abs(x)
    res = 0
    for i in range(0, x):
        res += x
    return res


def cube(x):
    res = 0
    if x > 0:
        for i in range(0, x):
            res += sqr(x)
    else:
        for i in range(0, x, -1):
            res -= sqr(x)
    return res
```

```
# test case
print("case 0:", cube(5))
print("case 1:", cube(1))
print("case 2:", cube(0))
print("case 3:", cube(-1))
print("case 4:", cube(-3))
```

Shell:

```
case 0: 125
case 1: 1
case 2: 0
case 3: -1
case 4: -27

Process finished with exit code 0
```

c.重复调用 sqr(x)来得到四次方结果:

```
def sqr(x):
    x = abs(x)
    res = 0
    for i in range(0, x):
        res += x
    return res


def quad(x):
    res = sqr(sqr(x))
    return res


# test case
print("case 0:", quad(5))
print("case 1:", quad(1))
print("case 2:", quad(0))
print("case 3:", quad(-1))
print("case 4:", quad(-3))
```

Shell:

```
case 0: 625
case 1: 1
case 2: 0
case 3: 1
case 4: 81

Process finished with exit code 0
```

2. 请完成<计算机导论>书本的练习题 1.2.5，1.2.6.

1.2.5 利用循环完成阶乘（全排列）:

```
def factor(n):
    if n < 0:
        ret = 'error'
    elif n == 0:
        ret = 1
    else:
        ret = 1
        for i in range(1, n + 1):
            ret *= i

    return ret


print(factor(4))
```

Shell:

```
24

Process finished with exit code 0
```

1.2.6 利用组合数的另外的公式得到这个答案，由于

```
def factor(n):
    if n < 0:
        ret = 'error'
    elif n == 0:
        ret = 1
    else:
        ret = 1
        for i in range(1, n + 1):
            ret *= i

    return ret


def combination(n, k):
    return factor(n) // (factor(n - k) * factor(k))


print(combination(10, 4))
```

Shell:

```
210

Process finished with exit code 0
```

3. 请用 Python 试验以下代码:

```
x = 98765432109876543210
print(1 / x + 1 == 1)

x = 9876543210
print(1 / x + 1 == 1)
```

请解释为什么 1 / x + 1 有时候会等于 1? 有时候会不等于 1?

首先观察这个代码的执行结果

```
True
False

Process finished with exit code 0
```

再观察这两个 x 计算出来的结果。
改写代码为:

```
x = 98765432109876543210
print(1 / x)

x = 9876543210
print(1 / x)
```

Shell:

```
1.0124999998860938e-19
1.0124999999873437e-10

Process finished with exit code 0
```

在 Python 中，浮点数除法结果的指数$e = \left|digit_{int_1} - digit_{int_2}\right| + 1$，并且有效数字依旧保留为 17 位（Python 标准库采用的是 IEEE754 标准的双精度浮点数，接下来对浮点数展示将以十进制形式呈现）。那么，当对这些结果分别+1 时，就会有不同结果：

$$
\begin{array}{rc}
 & e = 0 \quad s = 1.0000000000000000 \\
+ & e = -19 \quad s = 1.0124999998860938 \\
\hline
 & e = 0 \quad s = 1.0000000000000000 \\
+ & e = 0 \quad s = 0.\underbrace{0 \ldots 0}_{18\text{ 个}}10124999998860938\text{（对齐转换后）} \\
\hline
 & e = 0 \quad s = 1.\underbrace{0 \ldots 0}_{18\text{ 个}}10124999998860938\text{（真实值）} \\
 & e = 0 \quad s = 1.0000000000000000\text{（经过舍入和规格化后）}
\end{array}
$$

因而第一个 x 得出的结果就为 **True**

$$
\begin{array}{rc}
 & e = 0 \quad s = 1.0000000000000000 \\
+ & e = -10 \quad s = 1.0124999999873437 \\
\hline
 & e = 0 \quad s = 1.0000000000000000
\end{array}
$$

$$+ \qquad e=0 \quad s=0.\underbrace{0\dots0}_{9\text{个}}12499999873437 \quad \text{（对齐转换后）}$$

$$e=0 \quad s=1.\underbrace{0\dots0}_{9\text{个}}101249999998860938 \quad \text{（真实值）}$$

$$e=0 \quad s=1.0000000001012500 \quad \text{（经过舍入和规格化后）}$$

因而第二个 x 得出的结果就为 **False**

4. 请完成<计算机导论>书本的练习题1.3.2.

```
print(2 ** 10)
print(2 ** 20)
print(2 ** 30)
print(2 ** 40)
print(2 ** 50)
```

Shell:

```
1024
1048576
1073741824
1099511627776
1125899906842624

Process finished with exit code 0
```

增长很迅速

5. <计算机导论>类似练习题 1.3.4, 请将第一种 Python 程序 for 循环改写为 while 循环, 使得一旦找到所要的 g 就跳出循环. 这样可以减少不必要的循环.（请你的回答不要用到 break 语句）

```python
def square_root(c):
    i = 0  # display the times of loop
    g = 0
    j = 0
    while j * j < c:
        g = j
        j += 1
    # loop over
    while abs(g * g - c) > 0.0001:
        g += 0.00001
        i = i + 1
        print("%d:g=%.5f" % (i, g))

# define the function


square_root(10)
```

Shell:

```
1:g=3.00001
2:g=3.00002
......
16226:g=3.16226
16227:g=3.16227

Process finished with exit code 0
```

6. <计算机导论>类似练习题 1.3.3, 如何改写第二种<u>二分法</u>的 Python 程序, 使得当 $c<1$ 时, 例如 $c=0.9$, 也能算出正确的平方根. 提示: 更改 m_{max} 的起始值.

```
def square_root_improved(c):
    i = 0
    if c >= 1:
        m_max = c
        m_min = 0
    else:
        m_max = 1
        m_min = c
    g = (m_max + m_min) / 2
    while abs(g * g - c) > 0.00000000001:
        if g * g < c:
            m_min = g
        else:
            m_max = g
        g = (m_max + m_min) / 2
        i = i + 1
        print("%d:%.13f" % (i, g))


square_root_improved(0.9)
```

Shell:

```
1:0.9250000000000
2:0.9375000000000
……
29:0.9486832980998
30:0.9486832980532
```

7. 请完成以下程序设计, 并设计案例验证你的程序.

a.写出$\sqrt[3]{c}$的牛顿迭代法计算式. (用数学语言表达, 不需要程序)

b.利用牛顿法, 写出 Python 程序, 完成$\sqrt[3]{c}$的计算. 试验 c=10

c.利用二分法完成对$\sqrt[3]{c}$, c=10 计算的 Python 程序.

你的 Python 程序的答案请准确到小数点后 10 位, 请打印出循环次数.

a.经过一系列运算后，可以得到g_1和g_0的关系：

$$g_1 = (\frac{c}{3g_0^2} + \frac{2g_0}{3})$$

b.

```
def cube_root(c):
    g = 2 / 3 * c
    i = 1
    while g ** 3 - c > 0.00000000001:
        g = (2 / 3 * g + (c / g ** 2) / 3)
        print("%d:%.10f" % (i, g))
        i += 1
cube_root(10)
```

Shell:

```
1:4.5194444444
2:3.1761586365
3:2.4478652136
4:2.1882033331
5:2.1549531315
6:2.1544348147
7:2.1544346900

Process finished with exit code 0
```

c.参考之前利用二分法求二次根号的代码修改即可：

```
def cube_root(c):
    i = 0
    m_max = c
    m_min = 0
    g = (m_max + m_min) / 2
    while abs(g * g * g- c) > 0.00000000001:
        if g * g * g< c:
            m_min = g
        else:
            m_max = g
        g = (m_min + m_max) / 2
        i = i + 1
        print("%d:%.10f" % (i, g))
cube_root(10)
```

Shell:

```
1:2.5000000000
2:1.2500000000
3:1.8750000000
......
36:2.1544346901
37:2.1544346901
38:2.1544346900

Process finished with exit code 0
```

8. 用 Python 程序完成<编程导论>的练习题: 1.5.7, 1.5.9. 对于 1.5.9 请尽量自己想出另外一种方式来完成. 然后与书中的程序做个比较, 当 n 很大的时候, 哪个比较快? 请分析.

1.5.7

```
L = [4, 2, -11, 3, 1, 5, ]
a = L[0]
L1 = []
L2 = []
for i in range(1, len(L)):
    if L[i] > a:
        L2.append(L[i])
    else:
        L1.append(L[i])
print("This list is", L1 + [a] + L2)
```

Shell:

```
This list is [2, -11, 3, 1, 4, 5]

Process finished with exit code 0
```

1.5.9

首先是一种思考方式，就是将书上的方法改良，即仅仅修改代码形式

```
def binomial_expansion():
    row = (1, )
    while True:
        yield row
        pairs = zip(row, row[1:])
        new = (x + y for x, y in pairs)
        row = (1, *new, 1)


n = int(input())
coefficient = binomial_expansion()
for i in range(n):
    coefficient.__next__()
print(list(coefficient.send(None)))
```

Shell:

```
10
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

Process finished with exit code 0
```

书中的代码为:

```
n = 10
L = [1, 1]
for i in range(1, n):
    L0 = [0] + L
    L = L + [0]
    for j in range(len(L)):
        L[j] = L[j] + L0[j]
print(L)
```

为了测量运行时间，添加了以下代码

```
import time
start = time.time()
……
elapsed = (time.time() - start)
print(elapsed)
```

经过测量，当 n=10000 时(这个时候 n 的值就不需要输入了)，自己代码运行花费了 10.548508644104004s，而书上的代码花费了 12.115744590759277s。

具体原因，一是由于自己代码使用的是生成器和迭代器，可以占用更小的内存空间以及运算得更快。二是由于自己代码使用的是元组（元组在 python 中有很多优化）。

但是这个代码与书上代码采用的思想是一致的，也就是生成了 n 次方之前的项来使得 n 可以得到（这个代码也可以很快地改成生成杨辉三角）。可就解决这个问题而言，这个算法本身是不高效的，因为计算了一些不必要的数据。

现在考虑一个组合数迭代公式：

$$C_n^k = C_n^{k-1} \times \frac{n-k+1}{k}$$

按照这个思路，改写代码就可以为：

```
n = int(input())
row = (1,)
coefficient = 1
for k in range(1, n + 1):
    coefficient = coefficient * (n - k + 1) // k
    row += (coefficient,)
print(list(row))
```

Shell:

```
10
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]

Process finished with exit code 0
```

当 n = 10000 时，代码竟然只花费了 0.8524680137634277s，这个是多么大的进步。可以说由于少算了前面 9999 个答案，让这个问题的解决速度快了一个数量级左右（尤其是更大的情况）

有一个小细节，对于

`coefficient = coefficient * (n - k + 1) // k`

千万不能写成

`coefficient *= (n - k + 1) // k`

因为运算顺序有区别，会导致结果出错。

综上，**少算不必要算的东西，也是加快算法的方法**

9. 定义一个函数 mod(a, b, x), 返回 a^b mod x 的值, 其中 a, b, x 均为整数. 例如 mod(2, 5, 6）=2^5 mod 6 的结果为 2, mod(3, 4, 6)的结果为 3. 请设计 Python 函数, 利用你的函数计算:

a. mod(33,12345678,166)

b. mod(33,123456789,166)

c. mod(33,123456789000,166)

d. 计算 a 从 30 到 39，输入到 mod(a,12345678,166), 10 个结果累加的总和值, 再对 166 取模。

请注意, 如果 b 非常大, 你会无法短时间完成 a^b 的计算. 想想看, 要怎么办?

Hint: 从数学中你知道(xy) mod c = (x mod c)(y mod c) mod c

刚看到这个题目的时候，考虑了一个很“数学”的方式：

定义 欧拉函数（*Euler's totient function*）

欧拉函数$\varphi(n)$的值就是 n 的简化剩余系的元素个数

若 n 有标准分解$p_1^{k_1}p_2^{k_2}\ \dots\ p_r^{k_r}$（其中各$p_i$为互异的质因子，各$k_i$大于 1 为质因子的次数），那么

$$\varphi(n) = n\prod_{i=1}^{r}\left(1-\frac{1}{p_i}\right)$$

定理 欧拉定理（*Euler theorem*）

若 n,a 均为正整数，且 gcd(a,n) = 1，则

$$a^{\varphi(n)} \equiv 1(mod\ n)$$

其中，$\varphi(n)$被称为欧拉函数

注意到欧拉定义的使用情况中，强调了互质，那么，对于一些不能互质情况，我们需要稍微做一些处理（这个也是代码中比较特殊的地方）

按照以上的数学定理，定义了第一种 *mod(a,b,x)*函数（当然，为了让乘方更快，采用了快速幂）

```python
import math
# define the function of Euler_phi
def euler_phi(x):
    ret = x
    i = 2
    while i * i <= x:
        if x % i == 0:
            ret = ret // i * (i - 1)
            while x % i == 0:
                x //= i
        i += 1
    if x > 1:
        ret = ret // x * (x - 1)
    return ret
# define the fast_pow
def fast_pow(m, n):
    if n == 1:
        ret = m
    else:
        temp = fast_pow(m, n // 2)
        if n % 2 == 1:
            ret = m * temp * temp
        else:
            ret = temp * temp
```

```
    return ret
# define the function of mod
def mod(a, b, x):
    if math.gcd(a, x) == 1:
        ret = fast_pow(a, (b % euler_phi(x))) % x
    else:
        ret = fast_pow((a * math.gcd(a, x)), euler_phi(x // math.gcd(a, x))) % x
        ret = (ret * fast_pow(a, (b % euler_phi(x)))) % x
    return ret
```

这个算法最大的优点在于对于 a，b 的值并不敏感（就算 a，b 是个天文数字，这个时间也不会变化剧烈），决定这个计算最主要是 x 的大小（计算欧拉函数是一个时间复杂度为$O(\sqrt{n})$的算法），经过测试，在 x 不太大（约小于 2^{16} 次方）时，几乎都可以瞬间计算出答案。

但人能够胜任的事情（利用欧拉定理算大数取模），计算机不一定快。这个计算过程不够“笨拙”，计算机计算的时候，花了大量的时间去计算欧拉函数（对于人来讲计算欧拉函数有很多取巧的地方，这些都是不适合计算机的）。现在考虑以下同余的一些性质，思考：**能不能边计算边取模，让计算的数时刻保持很小呢？（就是 hint）**
按照这个想法，又可以定义第二种 *mod(a,b,x)*函数

```
import math
def mod(a, b, x):
    temp_ret = 1
    temp_b = b
    while temp_b >= 1:
        if a != 1:
            i = int(math.log(x, a)) + 1
        else:
            i = 2
        temp_a = a
        a = a ** i % x
        temp_b = b // i
        b_rest = b - temp_b * i
        b = temp_b
        temp_ret *= temp_a ** b_rest
    ret = temp_ret % x
    return ret
```

每计算一定步骤后，就会把相同的计算部分取同余后给去除掉（利用同余的性质）。并且，这个算法可以保证在 a，b，x 均极大的时候可以短时间算出答案。若没有 if 判断语句，对于 $mod(3,4,5) = 1$ 与 $mod(3,100,7) = 4$ 的数据是无法处理的，因为它的$\delta_x(a)$（即 a 对摸 x 的指数（或阶））≤b，会导致程序直接报错（1 不能作为对数的底数）。
按照第二种算法编写函数，以下的代码略去对 mod(a,b,x)的定义

a.

```
……
print(mod(33, 12345678, 166))
```

Shell:

```
17

Process finished with exit code 0
```

b.

```
……
```

```python
print(mod(33, 123456789, 166))
```

Shell:

```
123

Process finished with exit code 0
```

c.

```python
......
print(mod(33, 123456789000, 166))
```

Shell:

```
63

Process finished with exit code 0
```

d.

```python
all_sum = 0
for i in range(30, 40):
    all_sum += mod(i, 12345678, 166)
res = all_sum % 166
print(res)
```

Shell:

```
99

Process finished with exit code 0
```

10. 请基于你目前掌握的知识尽可能回答以下问题, 答案越详细越好.

a. 操作系统的功能是什么?

b. 请你解释 CPU 是如何执行程序 x=x+3 的?

a.

首先，操作系统可以统筹所有的硬件设备，让所有硬件可以有条不紊地运行：

（1）对于输入输出设备，操作系统本身会有大量的 I /O 模型，允许不同厂商依此编写驱动程序，并加载到操作系统中；

（2）对于 CPU 来说，由于现代的 CPU 多是多核，操作系统需要统筹调度各种任务，使得这些计算资源不会被浪费；

（3）对于内存来说，操作系统可以管理内存，使得多个任务可以共享内存资源，同时也会将超出内存存储的任务存储在硬盘中。

另外，操作系统可以管理各种软件与文件系统。所有的软件操作都需要经过操作系统来与硬件交互，用户的所有操作都不能直接接触操作系统与硬件，以此来保证计算机中存储数据的安全。

总而言之，操作系统处于软件和硬件之间的层次。

b.

第一步：首先要将变量 x 存储在内存的某个位置（这个地址假设为 0x114514）。然后，x = x + 3 语句至少应分为：

指令 1　读取 x 的数值（假设为 1）到 CPU 的某个寄存器中（假设为 R1）;

指令 2　将 R1 寄存器的数值+3;

指令 3　将 R1 寄存器值存回 a。

以上三句存储在内存地址为 0x810，0x814，0x818 处（假设在 32 位机器上）。CPU 中的 PC（程序计数器）指向地址 0x810。

第二步：PC 值为 0x810，CPU 从地址 0x810 处读取“指令 1”到 IR（指令寄存器），解读执行该指令。从内存中 0x114514 处读取 x 的值，通过总线将它传入到 R1 中，PC 值加 4。

第三步：PC 值为 0x814，CPU 从地址 0x814 处读取“指令 2”到 IR，解读执行该指令。这个时候 R1 的数值会输入到 ALU（算术逻辑单元）中进行计算（加上 3），完成后将结果返回到 R1 中，PC 值加 4。

第四步：PC 值为 0x818，CPU 从 0x818 处读取“指令 3”到 IR，解读执行该指令。R1 中 x 的数值（此时为 4）通过总线被存回内存地址 0x114514 处。