习题

求100内的素数

```
# 基本做法
# 一个数能被从2开始到自己的平发根的正整数整数整除，就是合数
import math
n = 100
for x in range(2, n):
    for i in range(2, math.ceil(math.sqrt(x))):
        if x % i == 0:
            break
    else:
        print(x)

# 改进1
# 存储质数合数一定可以分解为几个质数的乘积
import math
n = 100
primenumber = []
for x in range(2, n):
    for i in primenumber:
        if x % i == 0:
            break
    else:
        print(x)
        primenumber.append(x)

# 改进2
# 使用列表存储已有的质数，同时增加范围
import math
primenumber = []
flag = False
for x in range(2,100000):
    for i in primenumber:
        if x % i == 0 :
            flag = True
```

```
            break
        if i >= math.ceil(math.sqrt(x)):
            flag = False
            break
    if not flag:
        print(x)
        primenumber.append(x)
```

我们来比较一下

```
import datetime

upper_limit = 100000

start = datetime.datetime.now()
count = 1
for x in range(3, upper_limit, 2):  # 舍弃掉所有偶数
    if x > 10 and x % 10 == 5: # 所有大于10的质数中，个位数只有1,3,7,9。意思就是大于5，结尾是5就能被5整除了
        continue
    for i in range(3, int(x ** 0.5) + 1, 2): # 为什么从3开始，且step为2?
        if x % i == 0:
            break
    else:
        count += 1
        # print(x, count)
        pass

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
print(count)
print("~~~~~~~~~~~~~~~")

start = datetime.datetime.now()
x = 5
step = 2
count = 2
#print(2, 3, sep='\n')
while x < upper_limit:
    for i in range(3, int(x**0.5) + 1, 2): # p和n都是奇数，那么不必和偶数整除
```

```
        if not x % i:
            break
    else:
        #print(x)
        count += 1

    x += step
    step = 4 if step == 2 else 2

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
print(count)

print("~~~~~~~~~~~~~~~~")



# 改进2
# 使用列表存储已有的质数，同时增加范围
import math

start = datetime.datetime.now()
primenumber = []
flag = False
count = 0
for x in range(2,100000):
    for i in primenumber:
        if x % i == 0 :
            flag = True
            break
        if i >= math.ceil(math.sqrt(x)):
            flag = False
            break
    if not flag:
        #print(x)
        count += 1
        primenumber.append(x)

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
```

```
print(count)

print("~~~~~~~~~~~~~~~~")
```

增加了列表并没有大幅度提升，为什么？
进一步修改，去除

```
# 改进2
# 使用列表存储已有的质数，同时增加范围
import math

start = datetime.datetime.now()
primenumber = []
flag = False
count = 1
for x in range(3,100000,2): # 奇数
    for i in primenumber:
        if x % i == 0 :
            flag = True
            break
        if i >= math.ceil(math.sqrt(x)):
            flag = False
            break
    if not flag:
        #print(x)
        count += 1
        primenumber.append(x)

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
print(count)

print("~~~~~~~~~~~~~~~~")
```

但是结果并没有提高计算速度，为什么？

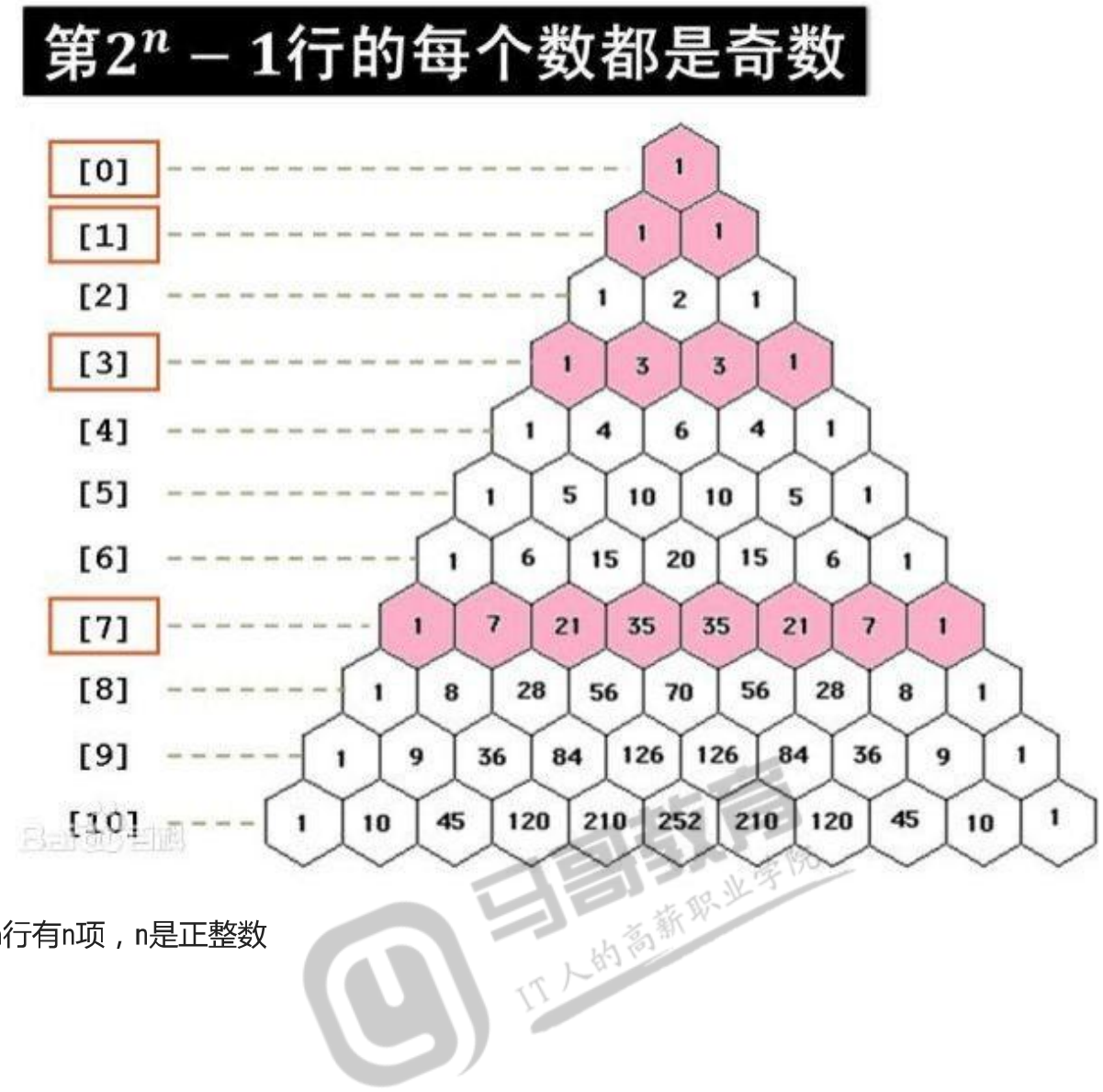
经过分析，得到下面的代码

```
# 改进2
# 使用列表存储已有的质数，同时缩小取模范围
import math
```

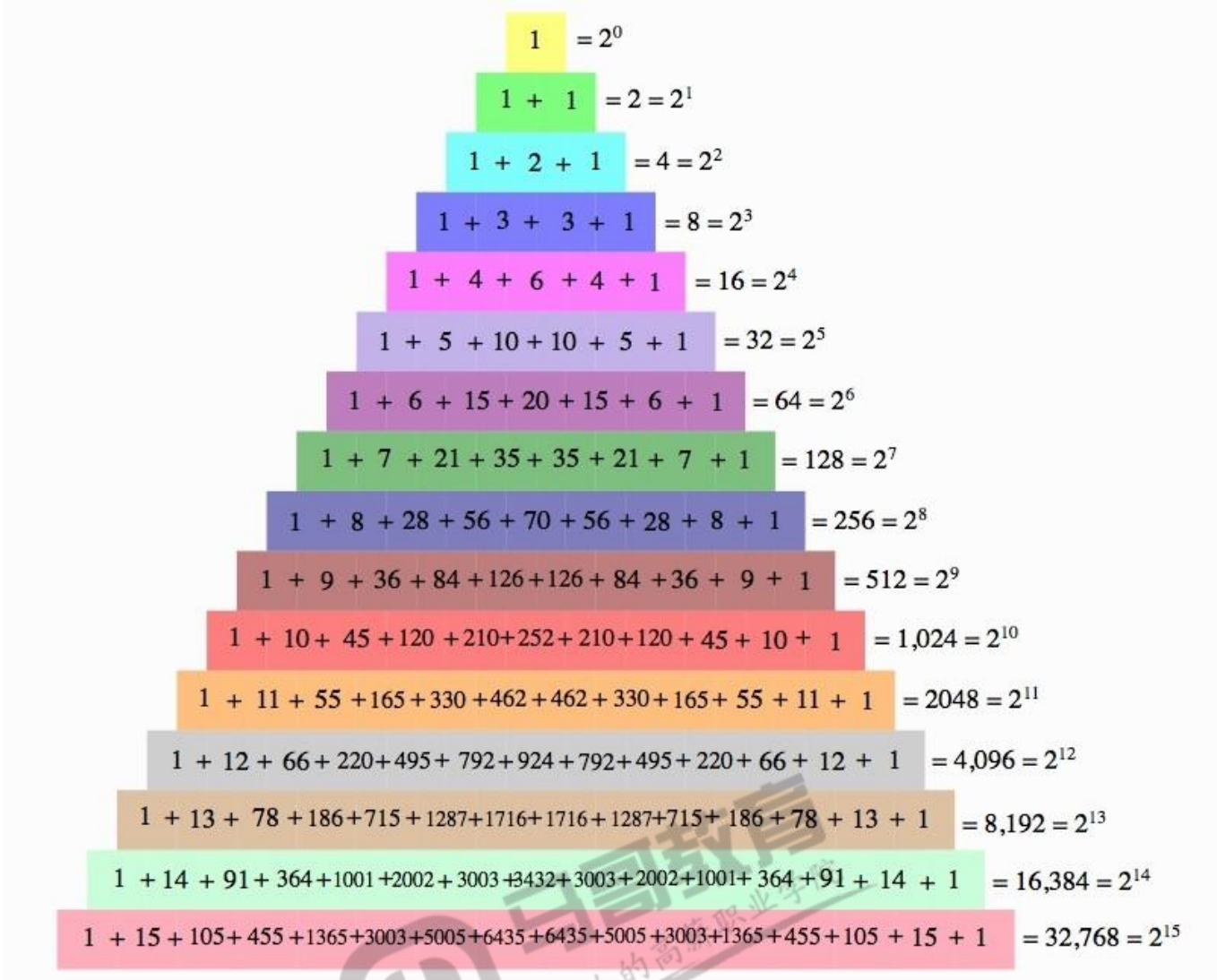
```
start = datetime.datetime.now()
primenumber = []
flag = False
count = 1
for x in range(3, 100000, 2):
    edge = math.ceil(math.sqrt(x))
    for i in primenumber:
        if x % i == 0 :
            flag = True
            break
        if i >= edge:
            flag = False
            break
    if not flag:
        #print(x)
        count += 1
        primenumber.append(x)

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
print(count)

print("~~~~~~~~~~~~~~~~")
```

直接提速，计算速度第一位。
本质上就是空间换时间。

计算杨辉三角前6行



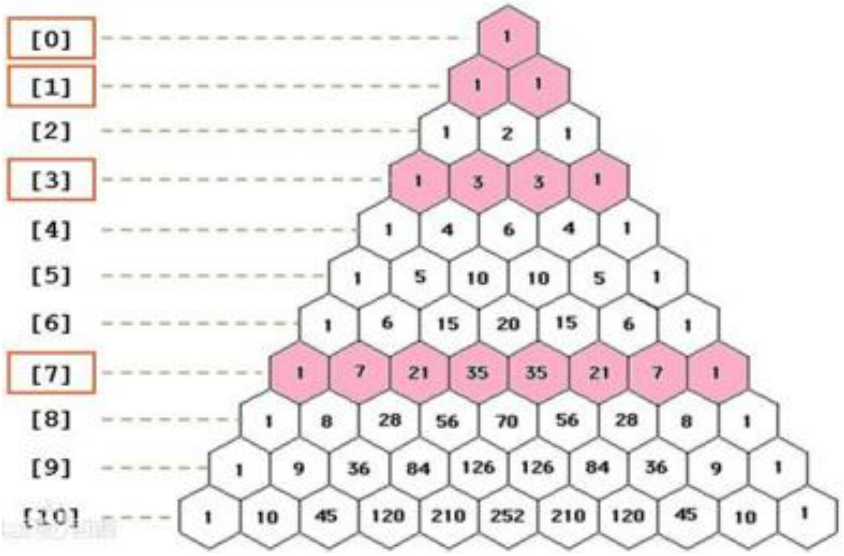
第n行有n项，n是正整数



第n行数字之和为2**(n-1)

杨辉三角的基本实现（方法1）

下一行依赖上一行所有元素，是上一行所有元素的两两相加的和，再在两头各加1



预先构建前两行，从而推导出后面的所有行

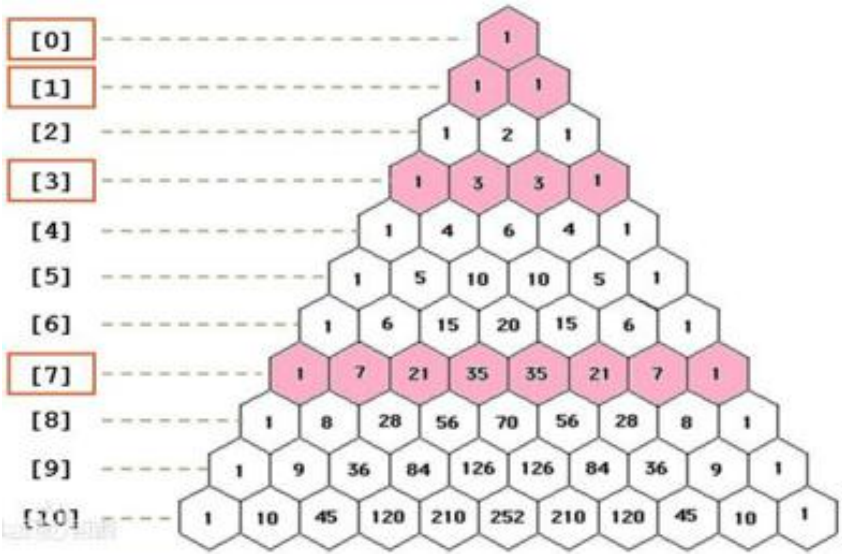
```
triangle = [[1], [1, 1]]

for i in range(2, 6):
    cur = [1]
    pre = triangle[i-1]
    for j in range(len(pre)-1):
        cur.append(pre[j] + pre[j+1])
    cur.append(1)
    triangle.append(cur)
print(triangle)
```

变体
从第一行开始

```
triangle = []
n = 6
for i in range(n):
    cur = [1]
    triangle.append(cur)

    if i == 0:
        continue

    pre = triangle[i-1]
    for j in range(len(pre)-1):
        cur.append(pre[j] + pre[j+1])
    cur.append(1)

print(triangle)
```


补零（方法2）

除了第一行以外，每一行每一个元素（包括两头的1）都是由上一行的元素相加得到。如何得到两头的1呢？
目标是打印指定的行，所以算出一行就打印一行，不需要用一个大空间存储所有已经算出的行。



while循环实现

```
n = 6
newline = [1] # 相当于计算好的第一行
print(newline)

for i in range(1, n):
    oldline = newline.copy() # 浅拷贝并补0
    oldline.append(0) # 尾部补0相当于两端补0
    newline.clear() # 使用append，所以要清除

    offset = 0
    while offset <= i:
        newline.append(oldline[offset-1] + oldline[offset])
        offset += 1
    print(newline)
```

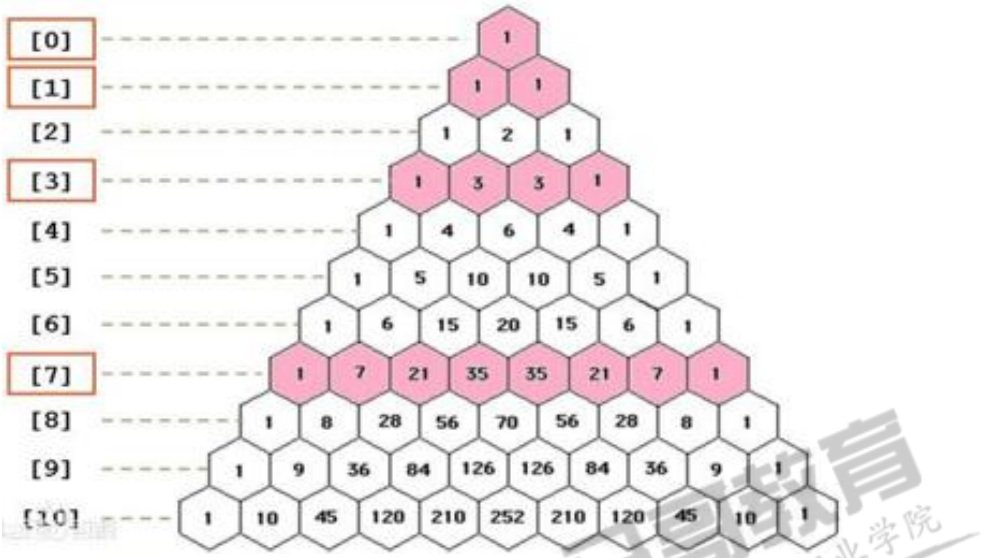
for循环实现

```
n = 6
newline = [1] # 相当于计算好的第一行
print(newline)

for i in range(1, n):
    oldline = newline.copy() # 浅拷贝并补0
    oldline.append(0) # 尾部补0相当于两端补0
    newline.clear() # 使用append，所以要清除

    for j in range(i+1):
```

```
        newline.append(oldline[j - 1] + oldline[j])
    print(newline)
```

对称性（方法3）

思路：
能不能一次性开辟空间，可以使用列表解析式或者循环迭代的方式。
能不能减少一半的数字计算。



中点的确定
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
把整个杨辉三角看成左对齐的二维矩阵。
i==2时，在第**3**行，中点的列索引**j==1**
i==3时，在第**4**行，无中点
i==4时，在第**5**行，中点的列索引**j==2**
得到以下规律，如果有**i==2j**，则有中点

```
triangle = []
n = 6
for i in range(n):
    row = [1] # 开始的1
    for k in range(i): # 中间填0，尾部填1
        row.append(1) if k == i-1 else row.append(0)
```

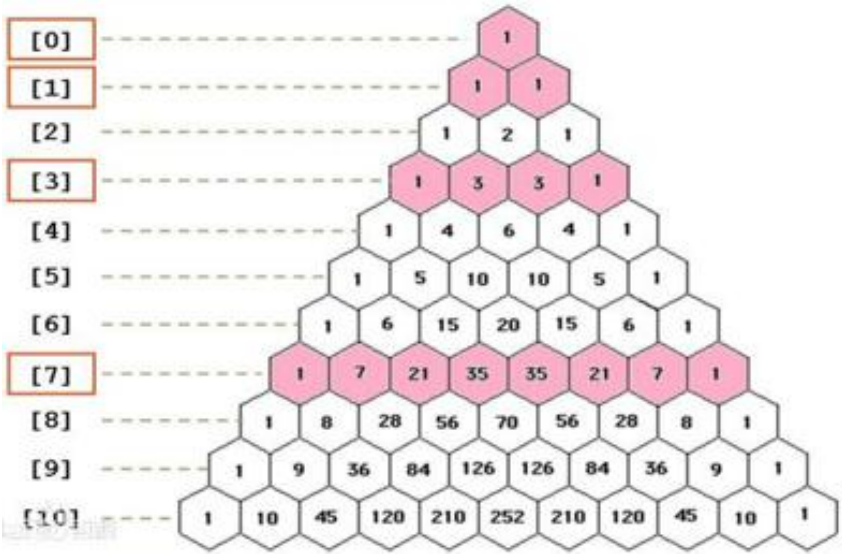
```
    triangle.append(row)
    if i == 0:
        continue
    for j in range(1,i//2+1): # i=2第三行才能进来
        #print(i, j)
        val = triangle[i - 1][j-1] + triangle[i - 1][j]
        row[j] = val
        # i为2，j为0 1 2，循环1次
        # i为3，j为0 1 2 3，循环1次
        # i为4，j为0 1 2 3 4，循环2次
        if i != 2*j: # 奇数个数的中点跳过
            row[-j-1] = val
print(triangle)
```

上面的代码看似不错，但行初始化的代码明显繁琐了，进一步简化

```
triangle = []
n = 6
for i in range(n):
    row = [1] * (i+1) # 一次性开辟
    triangle.append(row)
    for j in range(1,i//2+1): # i=2第三行才能进来
        #print(i, j)
        val = triangle[i - 1][j-1] + triangle[i - 1][j]
        row[j] = val
        if i != 2*j: # 奇数个数的中点跳过
            row[-j-1] = val
print(triangle)
```

单行覆盖（方法4）

方法2每次都要清除列表，有点浪费时间。
能够用上方法3的对称性的同时，只开辟1个列表实现吗？



首先我们明确的知道所求最大行的元素个数，例如前6行的最大行元素个数为6个。
下一行等于首元素不变，覆盖中间元素。

```
n = 6
row = [1] * n # 一次性开辟足够的空间

for i in range(n):
    offset = n - i
    z = 1 # 因为会有覆盖影响计算，所以引入一个临时变量
    for j in range(1, i//2+1): # 对称性
        val = z + row[j]
        row[j], z = val, row[j]
        if i != 2*j:
            row[-j-offset] = val
    print(row[:i+1])
```