4.列举你认为是数据思维的实例。

网购时的精准营销以及存货的库存管理优化，短视频平台的内容的精准推送，医院在给病人进行精准医疗以及在大规模采购医疗药品和器械时精准采购，都要把抽象的感觉用具体的海量数据加以支撑，体现数据思维。

5.递归与分治背后的思想是怎么样的？

递归法的思想

递归法的核心思想在于“直接或者间接地调用原算法本身的一种算法”。简单来说，就是算法在解决一个问题时，会将其分解为一个或多个与原问题相似但规模更小的子问题，然后递归地求解这些子问题。递归算法的关键在于必须有一个明确的递归结束条件，即当问题规模缩小到一定程度时，可以直接给出解，而不再继续递归。这样，通过逐层递归求解子问题，最终可以得到原问题的解。

递归法的优点在于算法设计简洁明了，对于具有递归性质的问题，递归算法往往能够给出非常直观的解决方案。然而，递归算法也存在一些缺点，比如可能会增加时间和空间的复杂度，特别是在递归深度较大的情况下，可能会导致栈溢出等问题。

分治法的思想

分治法的思想则是将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。具体来说，分治法包括三个主要步骤：分解、解决和合并。首先，将原问题分解

为若干个规模较小、相互独立、与原问题形式相同的子问题；然后，递归地解决这些子问题；最后，将各个子问题的解合并成原问题的解。

分治法的优点在于能够充分利用问题的分解性质，将复杂的问题简化为一系列简单的问题来解决。此外，由于子问题之间相互独立，因此可以并行处理，提高算法的效率。然而，分治法也要求原问题具有可分解性和子问题的独立性，否则可能无法有效应用。

1.乘积最大

（1）

N=2001 ,数列是全 3 的数列时，满足题意，可以使得乘积最大

(2)

数学原理分析：

对于整数 n 的拆分问题，要最大化各个拆分后的部分的乘积，可以使用以下几条数学原则：

尽量将数字拆成多个尽可能大的因子： 在划分整数时，如果你想让乘积最大，应该尽量使用较多的 2 和 3 进行划分。原因是相对较大的数字会在乘积中产生更大的效果。

优先使用 3 进行划分： 在大多数情况下，划分出的数字应该尽可能包含 3。因为 3 的幂指数比 2 和其他更大的整数能够提供更大的乘积。例如，2 的平方是 4，而 3 的乘积是 9，所以 3 作为

因子会比 2 更优。

注意余数的情况： 如果你无法完全用 3 去划分一个整数，剩下的部分如果是 2，可以保持。如果剩余部分是 1，那就需要调整，比如将 3 拆成 2 和 1，这样可以获得更大的乘积。

比方说，拿 2001 举例，我们尽可能地分出 3 来，就正好发现，答案数列就是一个全 3 的数列。也就是说，解决这个问题的第一步其实就是先看看数据可不可以尽可能的分出 3，但是要注意，这个只对还算比较大的数（比 4 大的数），如果是 4 的话，应当是分成 2 和 2 最大，这是因为我们在分解的时候，应完全避免分出 1 这个因子（这个因子只有坏作用，没有好作用），对于 4 而言，避免分出 1 的话，那最大的就是 2 和 2 了。

（3）python 代码如下，输入整数，输出最大乘积。

```
def integer_break(n):
    # 如果 n 是小于等于 3 的数，直接返回
    if n == 2:
        return 1  # 2 = 1 + 1，最大乘积是 1 * 1 = 1
    if n == 3:
        return 2  # 3 = 2 + 1，最大乘积是 2 * 1 = 2

    # 初始化结果
```

```
    product = 1

    # 优先尽量分成 3 的倍数
    while n > 4:
        product *= 3
        n -= 3

    # 剩余的部分是 2 或者 4，直接乘上
    product *= n

    return product

# 以输入 2001 为例，计算最大乘积
result = integer_break(2001)
print(f"输入 2001 时，最大乘积是：{result}")
```

2．打印 2 的若干次方

```
print (2 ** 10)
print(2 ** 20 )
print(2 ** 30)
print(2 ** 40)
print(2 ** 50)以此类推
```

呈现指数级增长（例如上一题中，3 的 667 次方就已经是一个超级大的数了）。由于数据的位数太多，在这里就不展示了。

3.渡河问题的最优解

Python 实现代码如下：

```
def search(Step):
    # 若该步骤能使各值均为 1，则输出结果，进入回归步骤
    if a[Step][0] + a[Step][1] + a[Step][2] + a[Step][3] == 4:
        for i in range(Step + 1):  # 能够依次输出不同的方案
            print('east:', end=' ')
            if a[i][0] == 0:
                print('wolf', end=' ')
            if a[i][1] == 0:
                print('goat', end=' ')
            if a[i][2] == 0:
                print('cabbage', end=' ')
            if a[i][3] == 0:
                print('farmer', end=' ')
            if a[i][0] and a[i][1] and a[i][2] and a[i][3]:
                print("none", end='')
            print(end=' ')
            print('west:', end=' ')
            if a[i][0] == 1:
```

```
            print("wolf", end=' ')
        if a[i][1] == 1:
            print('goat', end=' ')
        if a[i][2] == 1:
            print('cabbage', end=' ')
        if a[i][3] == 1:
            print('farmer', end=' ')
        if not (a[i][0] or a[i][1] or a[i][2] or a[i][3]):
            print('none', end='')
        print('\n')

        if i < Step:
            print(' the %d time' % (i + 1))

        if i > 0 and i < Step:
            if a[i][3] == 0:  # 农夫在本岸
                print(" ------> farmer ", end='')
                print(name[b[i] + 1])
            else:  # 农夫在对岸
                print(" <------ farmer ", end='')
                print(name[b[i] + 1])
        print('\n\n\n')
```

```
            return

        for i in range(Step):
            if a[i] == a[Step]:  # 若该步与以前的步骤相同，取消操作
                return
        # 若羊和农夫不在一起而狼和羊或者羊和白菜在一起，则取消操作
        if a[Step][1] != a[Step][3] and (a[Step][2] == a[Step][1] or a[Step][0] == a[Step][1]):
            return

        # 递归，从带第一种对象开始依次向下循环，同时限定递归的界限
        for i in range(-1, 3):
            b[Step] = i  # 记录农夫渡河的方式
            a[Step + 1] = a[Step][:]  # 复制上一步的状态，进行下一步移动
            a[Step + 1][3] = 1 - a[Step + 1][3]  # 农夫过去或者回来
            if i == -1:
                search(Step + 1)  # 进行第一步
```

```
            elif a[Step][i] == a[Step][3]:  # 若该物与农夫同岸，带回
                a[Step + 1][i] = a[Step + 1][3]  # 带回该物
                search(Step + 1)  # 进行下一步

if __name__ == '__main__':
    N = 15
    a = [[0] * 4 for i in range(N)]
    b = [0] * N
    name = [" ", "and wolf", "and goat", "and cabbage"]
    print(' 农夫过河问题，解决方案如下：\n')
    search(0)
```

4.用笨方法求解根号二，使得一旦找到需要的 g 就跳出循环，减少不必要的循环次数。

Python 代码如下：

```
def sqrt_newton(number, tolerance=1e-10):
    """
    使用牛顿法求解平方根
    :param number: 需要开平方的数
    :param tolerance: 容忍误差，用于判断何时停止迭代
    :return: 平方根的近似值
```

```
    """
    if number < 0:
        raise ValueError("Cannot compute the square root of a negative number")

    # 初始猜测值
    x = number / 2.0

    while True:
        next_x = (x + number / x) / 2
        if abs(x - next_x) < tolerance:
            break
        x = next_x

    return x

# 求解 sqrt(2)  可以单独设置误差 tolerance，默认是 1e-10
sqrt_2 = sqrt_newton(2)
print(sqrt_2)
```