24/10/27

T1 阿姆斯特朗数

题目

“阿姆斯特朗数”（Armstrong number），也称为“自幂数”。一个n位数是阿姆斯特朗数，如果它等于其各位数字的n次方和。例如，153是一个3位的阿姆斯特朗数，因为 153=13+53+33 。
请编写一个程序，找出 [0,999] 范围内所有的阿姆斯特朗数，并打印它们。

说明

1. 请使用for/while循环、for循环实现。
2. 数字之间以换行符 `\n` 分割。

Hint

C库函数：`double pow(double x,double y);` （包含头文件 `math.h`）

参考代码

```c
#include <stdio.h>
#include <math.h>

int main() {
    int num, sum, temp, remainder, n;

    // 遍历范围 [0, 999] 的每一个数
    for (num = 0; num <= 999; num++) {

        temp = num;
        sum = 0;

        // 计算当前数的位数
        n = 0;
        while (temp != 0) {
            temp /= 10;
            n++;
        }

        // 还原 temp 为 num
        temp = num;

        // 计算每一位数字的 n 次方和
        while (temp != 0) {
            remainder = temp % 10;
            sum += pow(remainder, n);
            temp /= 10;
        }

        // 检查是否是阿姆斯特朗数
        if (sum == num) {
            printf("%d\n", num);
        }
```

```
    }

    return 0;
}
```

代码解释

1. 库函数

```
#include <stdio.h>    // 用于输入和输出等的基本C语言库函数
#include <math.h>     // 用于使用pow()函数计算幂
```

2. 变量说明

- `num`：用于遍历和保存当前正在检查的数字。
- `sum`：用于保存各位数字的n次方和。
- `temp`：用于保存num的副本，以便进行数字拆分，同时避免误操作原num（一个基本的C语言编程思想）。
- `remainder`：用于保存拆分出来的当前位数字。
- `n`：表示当前数字的位数。

3. 主逻辑

1. **遍历所有数字**：使用`for`循环从`0`到`999`逐个检查每个数。
2. **计算位数**：
   - 使用一个`while`循环将`temp`除以10，直到`temp`为0为止，计算出该数字的位数`n`。
3. **计算各位数字的n次方和**：
   - 将`temp`重置为`num`，然后在`while`循环中，将`temp`的每个位取出（`temp % 10`），计算其`n`次方并累加到`sum`中。（关键是利用`int`类型的特性）
   - 将`temp`除以10，以进入下一位。
4. **判断阿姆斯特朗数**：
   - 如果`sum`等于`num`，则它是阿姆斯特朗数。

T2 取模小Trick

题目

我们定义一个 X 数列：

1. $a\_1=1$
2. $a\_2=2$
3. $a\_3=3$
4. ...
5. $a\_n=2*a\_(n-1)+a\_(n-3)$ $(n>3)$

给出一个正整数 k，请输出 X 数列的第 k 项 a_k 除以 `32767` 的余数。

输入格式

输入的第一行是一个整数 k ($1 \leq k \leq 40$)

输出格式

输出一行为 X 数列的第 k 项 a_k 除以 32767 的余数

参考代码

```c
#include <stdio.h>

int main() {
    int k;
    int a[41];  // 数组 a 用于保存数列 X 的前 40 项

    scanf("%d", &k);

    // 初始项
    a[1] = 1;
    a[2] = 2;
    a[3] = 3;

    // 递推计算 X 数列的第 4 项到第 k 项
    for (int i = 4; i <= k; i++) {
        a[i] = 2 * a[i - 1] + a[i - 3];
        a[i] %= 32767;  // 取模运算以防止数值过大
    }

    printf("%d\n", a[k]);

    return 0;
}
```

代码解释

1. 变量说明

- `k`：存储输入的正整数 (k)。
- `a[41]`：用于保存数列 (X) 的前 40 项。（定义为 41 个元素的原因是数组索引从 0 开始）

2. 主要逻辑

1. **输入**：读取一个正整数 (k)。
2. **初始化初始项**：将数列 (X) 的前三项初始化为 (a[1] = 1)、(a[2] = 2)、(a[3] = 3)。
3. **递推计算**：
   - 从第 4 项开始，使用递推公式计算每一项，并在每次计算后对结果取模 32767。
   - 取模运算 `a[i] %= 32767;` 可以防止计算过程中的数值溢出。（由于输出格式即为余数，故不影响结果）
4. **输出结果**：输出第 (k) 项的结果。

T3 杨辉三角

题目

打印杨辉三角。

输入

输入只有一行，表示需要打印的杨辉三角的行数 $n(1\leq n\leq 20)$。

输出

输出为打印的杨辉三角，其中，需要打印出类似等腰三角形的样式。每行各个数字之间用**一个空格**隔开（不考虑数字位数对齐），最后一个数后面输出**换行符**。

Hint

杨辉三角，是**二项式系数**在三角形中的一种几何排列，性质如下：

1. 每个数等于它上方两数之和。
2. 每行数字左右对称，由1开始逐渐变大。
3. 第n行的数字有**n**项。
4. 前n行共 **[n(1+n)]/2** 个数。
5. 第n行的m个数可表示为**C(n-1，m-1)**，即为从n-1个不同元素中取m-1个元素的组合数。

参考代码

```
#include <stdio.h>

int main() {
    int n;
    int triangle[20][20] = {0}; // 初始化杨辉三角的二维数组，所有元素初始为0

    // 输入杨辉三角的行数
    scanf("%d", &n);

    // 构造杨辉三角
    for (int i = 0; i < n; i++) {
        // 每行的第一个和最后一个元素赋值为1
        triangle[i][0] = 1;
        triangle[i][i] = 1;

        // 填充中间的值
        for (int j = 1; j < i; j++) {
            triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
        }
    }

    // 打印杨辉三角，格式化输出成等腰三角形
    for (int i = 0; i < n; i++) {
        // 输出前导空格
        for (int k = 0; k < n - i - 1; k++) {
            printf(" ");
        }
```

```
        // 输出当前行的数值
        for (int j = 0; j <= i; j++) {
            printf("%d", triangle[i][j]);
            if (j != i) {  // 行内的数字之间加空格，但最后一个数字不加
                printf(" ");
            }
        }
        printf("\n");  // 换行
    }

    return 0;
}
```

代码解释

1. 变量说明

- `n`：杨辉三角的行数。
- `triangle[20][20]`：二维数组用于存储杨辉三角的每一项。（定义20行20列是因为题目中行数不超过20）

2. 程序逻辑

1. **输入**：读取一个正整数`n`，表示需要打印的杨辉三角的行数。
2. **构造杨辉三角**：
   - 对于每一行`i`，先将第一个和最后一个位置赋值为1。
   - 对于每行的中间元素，使用递推关系计算：`triangle[i][j] = triangle[i-1][j-1] + triangle[i-1][j]`。
3. **打印输出**：
   - **前导空格**：通过`for`循环打印前导空格，使得输出符合等腰三角形格式。
   - **数字输出**：逐行输出杨辉三角中的每一个元素，每行数字之间用空格隔开，行末没有多余空格。

T4 奶牛分群

题目

Lucy养了 N(1≤N≤1,000,000,000) 头奶牛，但因为奶牛太多了，牧场的草很快就要被吃完了。这天，Lucy决定带她的奶牛们出发去探索牧场四周的土地。

从牧场出来的路只有一条。奶牛沿着这一条路走，一直走到一个三岔路口（可以认为所有的路口都是这样的）。这时候，这一群奶牛可能会分成三群，分别沿着这三条路继续走。如果她们再次走到三岔路口，那么仍有可能继续分成三群继续走。

奶牛的分群方式按照一定的规则：

1. 如果这群奶牛**能均分成三部分** 且 **每部分的牛数都不少于 2 头**，则这群奶牛分成三群，继续走下去。
2. 如果这群奶牛**不能均分为三部分**：但当**分出 2 头奶牛留在岔路口吃草**后，剩余的奶牛能满足条件`1`，则**剩余的奶牛分成三群**，继续走下去。

3. 如果这群奶牛均不符合上面两个规则，则这群奶牛不分裂，留在岔路口处一起吃草。

请计算，最终将会有多少群奶牛在平静地吃草。

输入

输入只有一行，为一个整数 N 。

输出

输出只有一行，为一个整数，表示最终的牛群数量

输入示例

```
14
```

输出示例

```
4
```

Hint

14 头奶牛不能均分成三部分；但是留出一群 2 头奶牛后，剩余的可以均分为三群奶牛，每群 4 头奶牛，之后无法再分裂。所以一共四群奶牛。

```
2 - 14
   /|\
  4 4 4
```

参考代码

```c
#include <stdio.h>

int countGroups(int N) {
    // 如果当前奶牛数量小于 2，则不能再分裂，直接返回 1 群
    if (N < 2) {
        return 1;
    }

    // 检查是否能均分成三部分且每部分不少于 2 头
    if (N % 3 == 0 && N / 3 >= 2) {
        return 3 * countGroups(N / 3);
    }

    // 检查是否能在分出 2 头奶牛后，剩下的奶牛能均分成三部分且每部分不少于 2 头
    if ((N - 2) % 3 == 0 && (N - 2) / 3 >= 2) {
        return 1 + 3 * countGroups((N - 2) / 3);
    }

    // 否则无法分裂，返回 1 群
    return 1;
}

int main() {
    int N;
```

```
    scanf("%d", &N);
    printf("%d\n", countGroups(N));
    return 0;
}
```

代码解释

1. 函数定义

- `countGroups(int N)`：递归计算包含 `N` 头奶牛的群数。

2. 递归过程

1. **基础情况**：当奶牛数 `N < 2` 时，返回 1，表示无法分裂。
2. **均分三群**：如果奶牛数 `N` 可以均分为三部分并且每部分不少于 2 头，则递归计算每群的数量，并乘以 3，返回 `3 * countGroups(N / 3)`。
3. **分出 2 头奶牛后均分三群**：若 `N - 2` 可以均分为三部分，每部分不少于 2 头，则递归计算余下的三群数量，加上分出的 1 群，返回 `1 + 3 * countGroups((N - 2) / 3)`。
4. **无法分裂**：如果不满足以上两种情况，则返回 1，表示无法再分裂，停在此处吃草。

3. 主函数

- 读取输入的奶牛总数 `N`，调用 `countGroups(N)` 计算并输出结果。