用欧拉计划学 Rust 编程

V2.2

申龙斌

2021 年 6 月 5 日

目　录

修订记录

2019 年 10 月 11 日，V1.0，包括 63 道难度 5%的题

2020 年 2 月 9 日，V1.1，环境准备一节里增加一些小工具的设置，调试器的设置等；补充 2 道难度为 5%的 684 题和 686 题。

2020 年 3 月 25 日，V2.0，增加了难度系数为 10%的 10 道题。

2021 年 6 月 5 日，增加几个未完成整理的题。

前言

2019 年 6 月 18 日，Facebook 发布了数字货币 Libra 的技术白皮书，我也第一时间体验了一下它的智能合约编程语言 MOVE，发现这个 MOVE 是用 Rust 编写的，看来想准确理解 MOVE 的机制，还需要对 Rust 有深刻的理解，所以又开始了 Rust 的快速入门学习。

看了一下网上有关 Rust 的介绍，都说它的学习曲线相当陡峭，曾一度被其吓着，后来发现 Rust 借鉴了 Haskell 等函数式编程语言的优点，而我以前专门学习过 Haskell，经过一段时间的入门学习，我现在已经喜欢上这门神奇的语言。

入门资料我用官方的《The Rust Programming Language》，非常权威，配合着《Rust by example》这本书一起学习，效果非常不错。

学习任何一项技能最怕没有反馈，尤其是学英语、学编程的时候，一定要“用”，学习编程时有一个非常有用的网站，它就是“欧拉计划”，网址：https://projecteuler.net，你可以在这个网站上注册一个账号，当你提交了正确答案后，可以在里面的论坛里进行讨论，借鉴别人的思路和代码。

如果你的英文不过关，有人已经将几乎所有的题目翻译成了中文，网址：http://pe-cn.github.io，本书中的许多题目的描述都照搬了该网站的翻译，感谢这个网站的制作人。

欧拉计划提供了几百道由易到难的数学问题，你可以用任何办法去解决它，当然主要还得靠编程，但编程语言不限，已经有 Java、C#、Python、Lisp、Haskell 等各种解法，当然直接用 google 搜索答案就没什么乐趣了。

学习 Rust 最好先把基本的语法和特性看过一遍，然后就可以动手解题了，解题的过程就是学习、试错、再学习、掌握和巩固的过程，学习进度会大大加快。

源代码下载

最新的源代码将不断同步更新在 github 上，网址：

https://github.com/slofslb/rust-project-euler

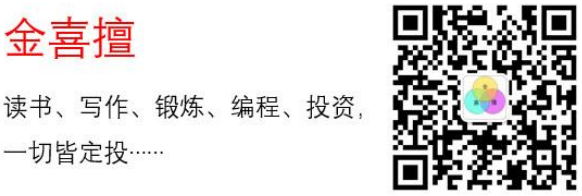
讨论

如果在学习过程中遇到任何问题，欢迎与我讨论交流。

本文档将不定期更新，欧拉计划官网不推荐将 100 题之外的解题过程发布出来，可加我微信索取最新的 PDF 电子书。

我的微信号是：SLOFSLB，暗号：欧拉计划。

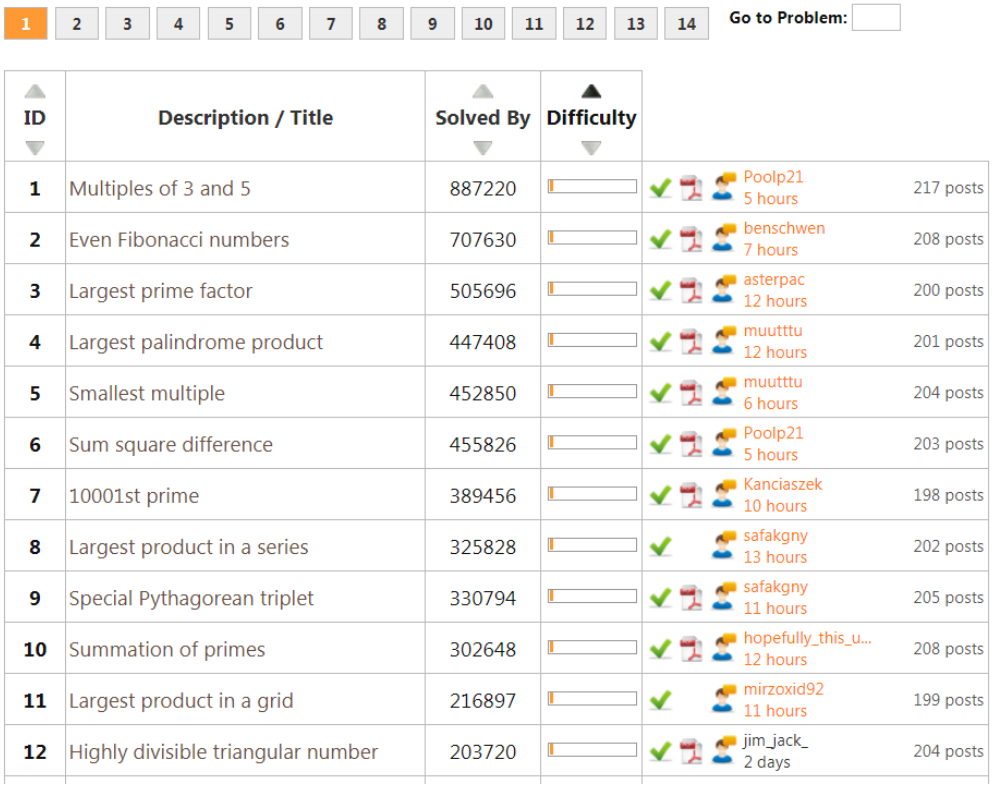
也欢迎关注我的新微信公众号“金喜擅”（原来是“申龙斌的程序人生”），个人独立博客是：http://shenlb.me，还有其它许多精彩内容。



1 题型介绍

欧拉计划中的各题都标出了难度系数，以百分数来表示，5%是其中难度最低的，难度最高的为 100%，截止到 2019 年 10 月 10 日，难题系数为 5%的题共有 63 道，可以作为 Rust 的入门练手题。

题目类型主要涉及整除性质、素数、因子、分数、回文数、阶乘、三角数、大整数、数字序列、路径计算、日期、全排列、组合数、初级密码学等方面，通过解这些题，可以了解 Rust 中的基本数据类型，向量用法，理解 Rust 中特有的所有权体系，体会函数式编程的思维等。



1 2 3 4 5 6 7 8 9 10 11 12 13 14 Go to Problem:

ID	Description / Title	Solved By	Difficulty		
1	Multiples of 3 and 5	887220		Poolp21 5 hours	217 posts
2	Even Fibonacci numbers	707630		benschwen 7 hours	208 posts
3	Largest prime factor	505696		asterpac 12 hours	200 posts
4	Largest palindrome product	447408		muutttu 12 hours	201 posts
5	Smallest multiple	452850		muutttu 6 hours	204 posts
6	Sum square difference	455826		Poolp21 5 hours	203 posts
7	10001st prime	389456		Kanciaszek 10 hours	198 posts
8	Largest product in a series	325828		safakgny 13 hours	202 posts
9	Special Pythagorean triplet	330794		safakgny 11 hours	205 posts
10	Summation of primes	302648		hopefully_this_u... 12 hours	208 posts
11	Largest product in a grid	216897		mirzoxid92 11 hours	199 posts
12	Highly divisible triangular number	203720		jim_jack_ 2 days	204 posts

2 环境准备

在 Windows 下安装，用官网（https://rust-lang.org）上的 rustup-init.exe 直接默认安装即可。

安装完成之后，就有了《The Rust Programming Language》这本书的离线 HTML 版本，直接用命令打开：

```
rustup doc --book
```

还要会使用强大的包管理器：cargo

这个 cargo 好用的另人发指，建项目、编译、运行都用用它：

```
cargo new euler1
cd euler1
cargo build
cargo run
```

由于众所周知的原因，在国内访问 rust 官网有些慢，特别是你在 build 时需要从网站自动下载大量的依赖库时，会非常慢，最好换成国内的镜像服务器，中国科技大学就有这样的镜像服务器。修改办法是，找到 Cargo 安装的文件夹，编辑 config 文件，文件内容：

```
[source.crates-io]
registry = "https://github.com/rust-lang/crates.io-index"
replace-with = 'ustc'
[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

还有一个编码习惯检查的小工具：clippy，可以帮助你检查出来一些写得不太规范的地方，推荐使用。

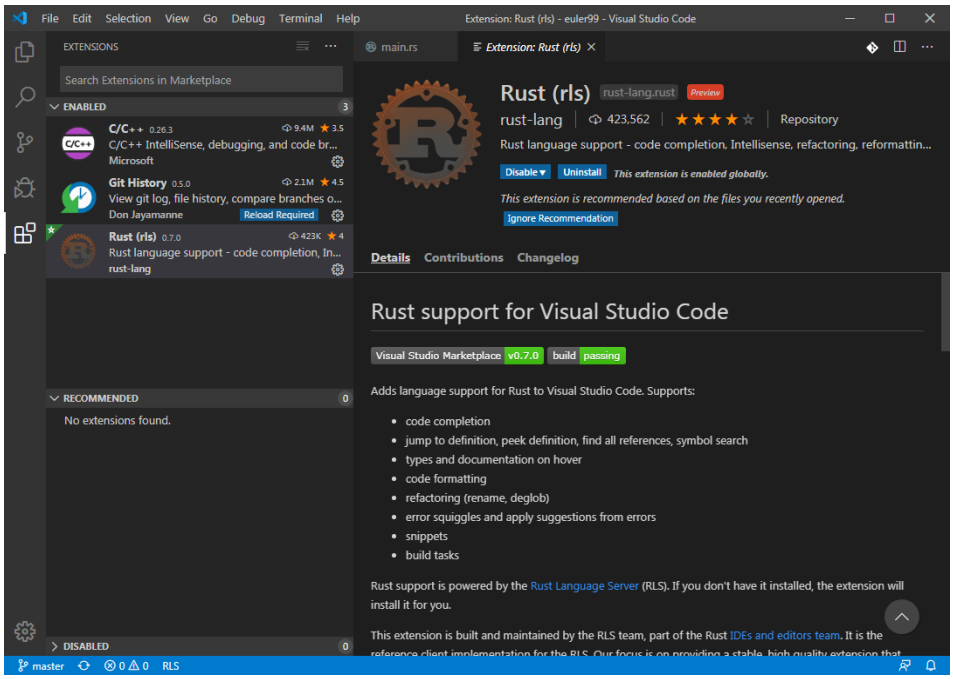
安装：

```
rustup component add clippy
```

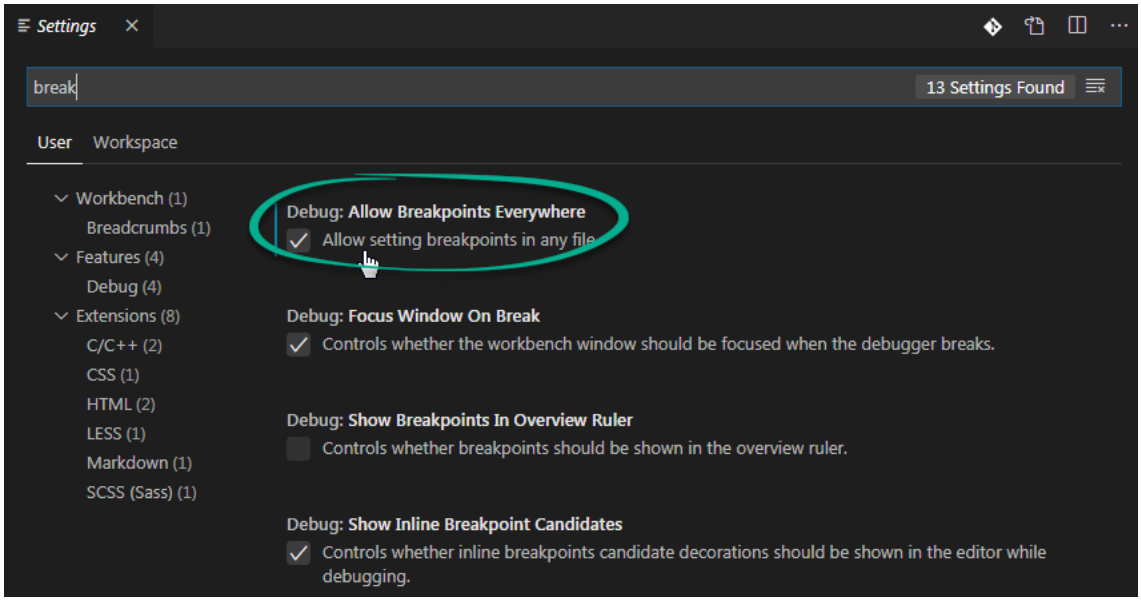
使用：

```
Cargo clippy
```

开发集成环境我推荐微软的 vscode，安装 rust 相关的插件 rls，为了将来的调试，还要安装 C/C++支持。

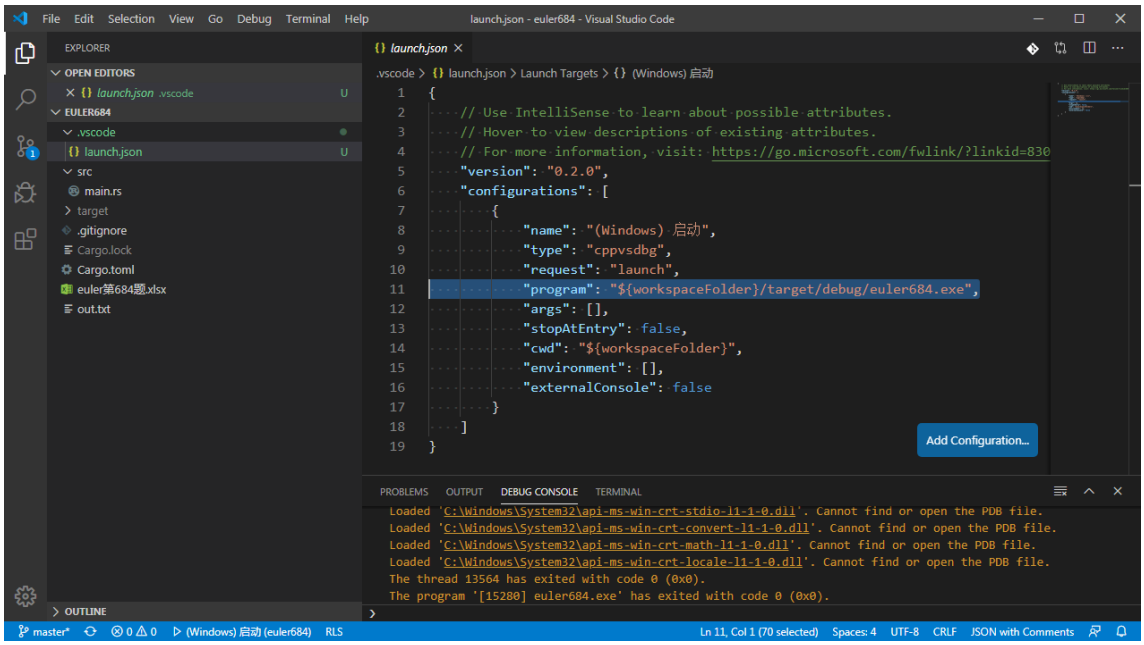


调试程序的时候，首先打开 vscode 里的 debug 设置选项。



再打开菜单 Debug -> Add Configuration，平台选 C/C++ (Windows)，此时需要编辑 launch.json 里的“program”属性，例如：

```
"program": "${workspaceFolder}/target/debug/my_program.exe”,
```


File Edit Selection View Go Debug Terminal Help
launch.json - euler684 - Visual Studio Code
EXPLORER
OPEN EDITORS
launch.json .vscode
EULER684
.vscode
launch.json
src
main.rs
target
.gitignore
Cargo.lock
Cargo.toml
euler第684题.xlsx
out.txt
OUTLINE
launch.json
.vscode > launch.json > Launch Targets > (Windows) 启动
{
// Use IntelliSense to learn about possible attributes.
// Hover to view descriptions of existing attributes.
// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830
"version": "0.2.0",
"configurations": [
{
"name": "(Windows) 启动",
"type": "cppvsdbg",
"request": "launch",
"program": "${workspaceFolder}/target/debug/euler684.exe",
"args": [],
"stopAtEntry": false,
"cwd": "${workspaceFolder}",
"environment": [],
"externalConsole": false
}
]
}
Add Configuration...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Loaded 'C:\Windows\System32\api-ms-win-crt-stdio-l1-1-0.dll'. Cannot find or open the PDB file.
Loaded 'C:\Windows\System32\api-ms-win-crt-convert-l1-1-0.dll'. Cannot find or open the PDB file.
Loaded 'C:\Windows\System32\api-ms-win-crt-math-l1-1-0.dll'. Cannot find or open the PDB file.
Loaded 'C:\Windows\System32\api-ms-win-crt-locale-l1-1-0.dll'. Cannot find or open the PDB file.
The thread 13564 has exited with code 0 (0x0).
The program '[15280] euler684.exe' has exited with code 0 (0x0).
master*
(Windows) 启动 (euler684)
RLS
Ln 11, Col 1 (70 selected)
Spaces: 4
UTF-8
CRLF
JSON with Comments

第一篇 难度系数为 5%

3 小试牛刀

这一部分题型相对简单，可以了解 Rust 的基本数据类型，文件读取和字符串操作。

第 1 题 筛选整数

求小于 1000 的能被 3 或 5 整除的所有整数之和。

熟悉 C 语言和 Python 语言的朋友，可以很快写出来：

```
let mut sum = 0;
for i in 1..1000 {
    if i % 3 == 0 || i % 5 == 0 {
        sum += i;
    }
}
println!("{}", sum);
```

mut 关键字（mutable 的缩写）是 Rust 的一大特色，Rust 中的所有变量默认为不可变的，如果想可变，需要 mut 关键字，否则在 sum += i 时会报编译错误。

for 语句的写法与 Python 的写法类似，也类似 C#中的 foreach 的写法，没有 C 语言中的 for(int i=0; i<1000; i++)三段式写法。

println! 后面有一个叹号，表示这是一个宏，Rust 里的宏也是非常非常强大！与 C 语言里的 define 完全不是一回事，以后再去了解宏的技术细节。

学过 Python 的列表推导（List Comprehension）语法的感觉这种题完全可以用一行语句搞定，Rust 中需要用到 filter()和 sum()函数。

```
// 为了便于阅读，分成多行
println!( "{}",
    (1..1000).filter(|x| x % 3 == 0 || x % 5 == 0)
```

```
                .sum::<u32>()
);
```

.. 这个语法糖表示一个范围，需要注意这是一个开区间，上限不包含 1000，如果想包含 1000，需要这样写：(1..=1000)。

filter 里面的|x|定义了一个闭包函数，关于闭包 closure，又是一个复杂的主题，以后再逐步了解。

sum::() 是一个范型函数，这种两个冒号的语法需要慢慢适应。

Rust 的早期版本没有提供 sum()函数，需要用 fold()函数来实现，是这样写的：

```
println!(
    "{}",
    (1..1000)
        .filter(|x| x % 3 == 0 || x % 5 == 0)
        .fold(0, |s, a| s + a)
);
```

用 collect()函数，可以把这些数全部打印出来。

```
println!(
    "{:?}",
    (1..1000)
        .filter(|x| x % 3 == 0 || x % 5 == 0)
        .collect::<Vec<u32>>()
);
// [3, 5, 6, 9, 10, 12, ... 999]
```

语法知识点：

- ✧ mut 表示可修改的变量
- ✧ println!宏的用法
- ✧ filter()函数和 sum()函数
- ✧ fold()函数
- ✧ collect()函数

第 2 题 偶斐波那契数

求 400 万之内所有偶数的斐波那契数字之和。

算法并不难，需要了解 Rust 中的向量的写法，这里的数列以[1, 2]开始，后面每个数是前面 2 个数字之和：

```
let mut fib = vec![1, 2];

let mut i = 2; // 已经有2个元素
let mut sum = 2;
loop {
    let c = fib[i - 1] + fib[i - 2];
    if c >= 4_000_000 {
        break;
    }
    fib.push(c);
    if c % 2 == 0 {
        sum += c;
    }
    i += 1;
}
println!("{}", sum);
```

这里没有使用函数式编程，大量使用了 mut，无限循环用 loop 语法。

rust 中关于整数的表示提供了多种数据类型，默认的整数类型是 i32，默认浮点类型是 f64。

数字类型中比较有特点的语法是可以用'_'分隔符，让数字更容易读一些，还可以把 u32, i64 等类型作为后缀来指明类型。

let 赋值语句与其它语言也不一样，还可以改变其类型，这个特性称为隐藏 shadowing。

```
let x = 500u16;
let x = x + 1;
let x = 4_000_000_u64;
let x = "slb";
```

这里的 fib 是一个向量，相当于其它语言里的数组、列表。用 vec! 宏可以对向量进行初始化赋值。

这一行：

```
let mut fib = vec![1, 2];
```

与下面三行等价：

```
let mut fib = Vec::new();
fib.push(1);
fib.push(2);
```

push()函数用于给列表增加一个元素。

还可以改进，利用 rust 的**延迟评价**特性，有起始值无终止值的无限序列可以用 for 语句搞定，原来的代码可以再精练一些，这种“2..”的语法在其它语言里是无法想像的。

```
let mut fib = vec![1, 2];

let mut sum = 2;
for i in 2.. {
    let c = fib[i - 1] + fib[i - 2];
    if c >= 4_000_000 {
        break;
    }
    fib.push(c);
    if c % 2 == 0 {
        sum += c;
    }
}
println!("{}", sum);
```

如果再使用函数式编程，还可以更精练一点：

```
let mut fib = vec![1, 2];
for i in 2.. {
    let c = fib[i - 1] + fib[i - 2];
    if c >= 4_000_000 { break; }
    fib.push(c);
}
println!("{}", fib.iter().filter(|&x| x % 2 == 0).sum::<u32>());
```

语法知识点：

- ✧ vec!宏进行向量初始化
- ✧ 往向量里增加一个元素用 push()函数
- ✧ (2..)无终止值的范围表示
- ✧ iter()可以迭代生成向量里的所有元素

第 3 题 最大质因数

找出整数 600851475143 的最大素数因子。

素数就是只能被 1 和本身整除的数，首先定义一个函数 is_prime()，用于判断是否为素数：

```
fn is_prime(num: u64) -> bool {
    for i in 2..(num / 2 + 1) {
        if num % i == 0 {
            return false;
        }
    }
    true
}
```

Rust 是强类型语言，看到函数定义里的 -> bool，这里可以看到 Haskell 语法的身影。

函数最后一行的 true 孤零零的，没有分号，让人感觉很奇怪。Rust 是一个基于表达式的语言，一个语句块的末尾可以是一个表达式，当然也可以用“return true;”表示。

现在可以查找最大的素数因子了：

```
let big_num = 600851475143;
for i in (2..=big_num).rev() {
    if big_num % i == 0 && is_prime(i) {
        println!("{}", i);
        break;
    }
}
```

程序编译没问题，但几分钟也运行不出来结果，试着把数字调小一点，比如：600851，不到 1 秒出来结果，看来程序的效率太差了，主要原因在于判断素数的运算量太大，需要优化算法。

可以尝试把大数进行素数因子分解，找到一个素因子之后，可以用除法缩小搜索的范围，效率得到大幅提升，不到 1 秒得出结果。

```
let mut big_num = 600851475143;
let mut max_prime_factor = 2;

while big_num >= 2 {
    for i in 2..=big_num {
        if big_num % i == 0 && is_prime(i) {
            big_num /= i;
            if i > max_prime_factor  {
                max_prime_factor = i;
                break;
            }
        }
    }
```

```
}
println!("{}", max_prime_factor);
```

Rust 有丰富的函数库可供使用，使我们不用重复发明轮子，在 primes 函数库里有一个 factors_uniq()函数，可以快速得到所有素数因子，主程序只需一条语句。

```
println!("{:?}", primes::factors_uniq(600851475143).last().unwrap
());
```

第 4 题 最大回文乘积

所谓回文数，就是两边读都一样的数，比如：698896。

求两个 3 位数之积最大的回文数。

先写一个判断回文数的函数：

```
fn is_palindromic(n: u64) -> bool {
    let s = n.to_string();
    s.chars().rev().collect::<String>() == s
}
```

把数字转换成字符串，再把字符串反序，如果与原字符串一样，则是回文数。

Rust 中字符串的反序操作好奇怪，竟然不是 s.rev()，先要用 chars()函数，我 google 搜索到了上面那个代码片段。

剩下的逻辑并不复杂，用两重循环可以快速搞定。

```
let mut max = 0;
for x in 100..=999 {
    for y in 100..=999 {
        let prod = x * y;
        if is_palindromic(prod) && prod > max {
            max = prod;
            // println!("{} x {} = {}", x, y, prod);
        }
    }
}
println!("{}", max);
```

我一开始以为只要反序搜索就可以快速找到答案，但找到的数并不是最大，你能发现问题之所在吗？不过，从下面这个错误代码中，我学会了双重循环如何跳出外层循环的语法。

```
// 错误代码
'outer: for x in (100..=999).rev() {
    for y in (100..=999).rev() {
        let prod = x * y;
        if is_palindromic(prod) {
            println!("{} x {} = {}", x, y, prod);
            break 'outer;
        }
    }
}
```

语法知识点：

字符串的反序用 s.chars().rev().collect::<String>()

第 5 题 最小倍数

找出能够被 1，2，3，...，20 整除的最小整数。

代码逻辑很简单，一个一个尝试整除，找到后跳出最外层循环。

```
'outer: for x in (100..).step_by(2) {
    for f in (2..=20).rev() {
        if x % f != 0 {
            break;
        }
        if f == 2 {
            println!("{}", x);
            break 'outer;
        }
    }
}
```

如果你感觉程序运行效率不够高，可以用下面这个命令行运行，差别还是非常大的，感觉与 C 程序的效率相媲美：

```
cargo run -release
```

上面为了跳出外部循环，专门加了一个标签，逻辑上感觉怪怪的，可以先定义一个函数。

```
// 一个数是否可以被1到20整除
fn can_divide_1_to_20(x: u64) -> bool {
    for f in (2..=20).rev() {
        if x % f != 0 {
            return false;
        }
    }
```

```
    true
}
```

主程序的代码的逻辑就清晰多了。

```
for x in (100..).step_by(2) {
    if can_divide_1_to_20(x) {
        println!("{}", x);
        break;
    }
}
```

熟悉函数式编程的，还可以写成一行：

```
print!("{}", (100..).step_by(2).find(|&x| can_divide_1_to_20(x)).unw
rap());
```

第 6 题 平方和与和的平方之差

求 1 到 100 自然数的“和的平方”与“平方和”的差。

用普通的过程式编程方法，这题没有难度，但要尝试一下函数式编程思路，代码会异常简洁。

```
let sum_of_squares = (1..=100).map(|x| x*x).sum::<u32>();
let square_sum = (1..=100).sum::<u32>().pow(2);
println!("{}", square_sum - sum_of_squares);
```

Rust 的较早版本没有提供 sum()函数，要用 fold()函数曲折实现，理解起来稍微困难一些：

```
let sum_of_squares = (1..=100).fold(0, |s, n| s + n * n);
let square_sum = (1..=100_u64).fold(0, |s, n| s + n).pow(2);
println!("{}", square_sum - sum_of_squares);
```

第 8 题 连续数字最大乘积

在 1000 位的大整数里找到相邻的 13 个数字，使其乘积最大。

首先系统内建的 u32, u64 或 u128 整数肯定无法保存 1000 位的大整数，我们用字符串来表示这个大整数，为了让代码好看些，用数组表示，并用 concat()函数合并。

```
let digits = vec![
    "73167176531330624919225119674426574742355349194934",
    "96983520312774506326239578318016984801869478851843",
    "85861560789112949495459501737958331952853208805511",
    "12540698747158523863050715693290963295227443043557",
    "66896648950445244523161731856403098711121722383113",
    "62229893423380308135336276614282806444486645238749",
    "30358907296290491560440772390713810515859307960866",
    "70172427121883998797908792274921901699720888093776",
    "65727333001053367881220235421809751254540594752243",
    "52584907711670556013604839586446706324415722155397",
    "53697817977846174064955149290862569321978468622482",
    "83972241375657056057490261407972968652414535100474",
    "82166370484403199890008895243450658541227588666881",
    "16427171479924442928230863465674813919123162824586",
    "17866458359124566529476545682848912883142607690042",
    "24219022671055626321111109370544217506941658960408",
    "07198403850962455444362981230987879927244284909188",
    "84580156166097919133875499200524063689912560717606",
    "05886116467109405077541002256983155200055935729725",
    "71636269561882670428252483600823257530420752963450",
].concat();
```

找到相邻的 13 个数字，需要用到字符串的切片(slice)功能，比如找到从 i 开始的 13 个字符形成了一个子串。这里面的“&”符号是容易出错的地方，digits 变量有所有权，如果被借用后，就不能再被使用，熟悉 C++的朋友，可以把“&”理解为引用，这样不破坏原来的所有权。

```
let x = &digits[i .. i + 13];
```

现在需要用到函数式编程的思路，将 13 个字符分离出来，并转换成数字，再相乘起来，用到 chars()， map()， to_digit()， unwrap()， fold()等一连串的函数，请自行体会。

```
x.chars()
 .map(|c| c.to_digit(10).unwrap())
 .fold(1_u64, |p, a| p * a as u64);
```

to_digit(10) 可用于将字符转换为数字，例如’9’转换为 9，需要注意这里的转换有可能出现异常，而 rust 处理异常的方式很特别，要重点学习 Option<T>的用法。

用 unwrap()函数可以将 Option<u64>类型转换成 u64 类型。

最后的代码是这样：

```
const ADJACENT_NUMBERS: usize = 13;

let mut max = 0;
```

```
for i in 0..digits.len() - ADJACENT_NUMBERS {
    let x = &digits[i..i + ADJACENT_NUMBERS];
    let prod = x
        .chars()
        .map(|c| c.to_digit(10).unwrap())
        .fold(1_u64, |p, a| p * a as u64);
    if prod > max {
        println!("index: {}   x: {}   prod: {}", i, x, prod);
        max = prod;
    }
}
```

第 17 题 表达数字的英文字母计数

1 到 1000 用英文单词写下来，求总字符个数（空格和连字符不算），例如：342，英文单词是：three hundred and forty-two。

问题分解：

1）数字转换成英文单词

1.1）1 到 19 的拼写

1.2）20 到 99 的拼写

1.3）100 到 999 的拼写

1.4）1000 的拼写

2）单词中去掉空格和连字符

3）求字符总数

1 到 19 的拼写比较特殊，需要分别对待，而超过 20 的数，可以利用递归调用。这里可以学到 String 的语法知识点。Rust 中的字符串有点烦人，list[n].to_string()、"one thousand".to_string()的这种写法让人非常不适应。除了 String 之外，还有字符串切片(slice)、字符常量，需要看基础教程慢慢理解。

```
fn english_number(n: usize) -> String {
    let list0_9 = vec![
        "zero", "one", "two", "three", "four",
        "five", "six", "seven", "eight", "nine",
    ];
    if n <= 9 {
```

```
        return list0_9[n].to_string();
    }
    if n <= 19 {
        let list = vec![
            "ten", "eleven", "twelve", "thirteen", "fourteen",
            "fifteen", "sixteen", "seventeen", "eighteen", "nineteen
",
        ];
        return list[n - 10].to_string();
    }
    if n <= 99 {
        let a: usize = n / 10; // 十位
        let b: usize = n % 10;
        let list = vec![
            "", "", "twenty", "thirty", "forty",
            "fifty", "sixty", "seventy", "eighty", "ninety"
        ];
        let str = list[a].to_string();
        if b > 0 {
            return str + "-" + &english_number(b);
        }
        return str;
    }
    if n <= 999 {
        let a: usize = n / 100; // 百位
        let b: usize = n % 100;
        let str = list0_9[a].to_string() + " hundred";
        if b > 0 {
            return str + " and " + &english_number(b);
        }
        return str;
    }
    if n == 1000 {
        return "one thousand".to_string();
    }
    return "unknown".to_string();
}
```

从字符串里移除特定的字符，要利用函数式编程，还有 filter()和 collect()函数，一气呵成。filter()函数中的*c 又是让人容易写错的地方。

```
fn remove_space(s: &str) -> String {
    s.chars().filter(|c| *c != ' ' && *c != '-').collect()
}
```

主程序就比较容易了，求和即可。

```
let mut sum = 0;
for n in 1..=1000 {
    let s = remove_space(&english_number(n));
    sum += s.len();
    // println!("{}: {} {}", n, english_number(n), s.len());
}
```

```
println!("{}", sum);
```

第 22 题 姓名得分

从文件中读取一堆名字，按字母顺序排序，求名字分总和。名字分 = 顺序号 * 名字中几个字母的序号和。

例如：COLIN，所有字符在字母表中的序号之和，3 + 15 + 12 + 9 + 14 = 53，COLIN 名字排在第 938 个，该名字的得分为 938 × 53 = 49714。

问题分解：

1）读文件，移除引号

2）把名字存储在 Vec 向量中

3）排序

4）求字符在字母表中的序号

5）求单词的分数

6）求总分

正式开始

1）首先把文件读到一个字符串中。

```
use std::fs;

fn main() {
    let data = fs::read_to_string("names.txt")
        .expect("读文件失败");
    println!("{}", data);
}
```

名字中都带着引号，需要移除，可以利用函数式编程，还有 filter()和 collect()函数，一气呵成。filter()函数中的*c 又是让人容易出错的地方。

```
fn remove_quote(s: &str) -> String {
    s.chars().filter(|c| *c !='"').collect()
}
```

2）每个名字是用逗号分开的，所以可以用 split()函数，分解成向量。

```
let data2 = remove_quote(&data);
let names: Vec<&str> = data2.split(",").collect();
println!("{:?}", names);
```

3）向量有专门的排序函数，需要将变量定义为可修改的。

```
let mut names: Vec<&str> = data2.split(",").collect();
names.sort();
```

4）字符在字母表中的顺序号，可以求 find()，也可以用 position()函数。

```
fn letter_number(ch: char) -> usize {
    let letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    letters.chars().position(|c| c == ch).unwrap() + 1
}
```

5）求一个单词的分数

```
fn word_score(word: &str) -> usize {
    let mut score = 0;
    for ch in word.chars() {
        score += letter_number(ch);
    }
    score
}
```

6)现在可以求总分了,有一个非常有用的 for 循环的用法,可以既得到元素，还可以得到元素的索引号，利用 enumerate()函数。

```
let mut score = 0;
for (i, name) in names.iter().enumerate() {
    let ws = word_score(name);
    println!("{} {} {}", (i+1), name, ws);
    score += ws * (i + 1);
}
println!("{}", score);
```

完整的 main()代码：

```
let data = std::fs::read_to_string("names.txt").expect("读文件失败");
let data2 = remove_quote(&data);
let mut names: Vec<&str> = data2.split(",").collect();

names.sort();
let mut score = 0;
for (i, name) in names.iter().enumerate() {
    let ws = word_score(name);
    println!("{} {} {}", (i + 1), name, ws);
    score += ws * (i + 1);
}
println!("{}", score);
```

语法点：

- ✧ std::fs 读文件
- ✧ 字符串的 split()函数
- ✧ 排序函数 sort()
- ✧ 字符串中查找一个字符的位置
- ✧ enumerate()迭代器，可以产生序号和元素

4 序列

这里需要了解递归函数的写法。

第 14 题 最长考拉兹序列

Collatz 序列的意思是，当一个数 n 是偶数时，下一数为 n/2；当 n 为奇数时，下一个数为 3*n+1。

这种序列有一个猜想，最后都会收敛于 4，2，1。例如：

13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1

从 100 万之内挑一个数作为起始数，生成 Collatz 序列，哪个生成的链最长？

用递归函数是比较简练的。

```
fn collatz_len(x: u64) -> u64 {
    if x == 1 { return 1; }
    let y;
    if x % 2 == 0 {
        y = x / 2;
    } else {
        y = x * 3 + 1;
    }
    collatz_len(y) + 1
}
```

里面有一个关于 y 的分支判断，可以利用类似 C#中的三元表达式 "cond ? a : b"写在一行里，在 Rust 里可以直接用 if 表达式。

```
fn collatz_len(x: u64) -> u64 {
```

```
    if x == 1 { return 1; }
    let y = if x % 2 == 0 { x / 2 } else { x * 3 + 1 };
    collatz_len(y) + 1
}
```

主程序用一个循环暴力搜索就行了：

```
fn main() {
    let mut max = 0;
    for num in 1..1_000_000 {
        let c = collatz_len(num as u64);
        if c > max {
            max = c;
            println!("start num: {}   chain length: {}", num, max);
        }
    }
}
```

程序还可以优化一下性能，将一些运算的结果缓存起来，以后遇到相似的序列时不用重复计算，这里不再展开讨论了。

第 92 题 平方数字链

将一个数的所有数字的平方相加得到一个新的数，不断重复直到新的数已经出现过为止，这构成了一条数字链。

例如，

44 → 32 → 13 → 10 → 1 → 1

85 → 89 → 145 → 42 → 20 → 4 → 16 → 37 → 58 → 89

可见，任何一个到达 1 或 89 的数字链都会陷入无尽的循环。更令人惊奇的是，从任意数开始，最终都会到达 1 或 89。

有多少个小于一千万的数最终会到达 89？

解题思路：

1）各位数字的平方和

```
fn square_sum(n: u64) -> u64 {
    n.to_string()
     .chars()
     .map(|x| x.to_digit(10).unwrap().pow(2) as u64)
     .sum::<u64>()
}
```

可以用整数运算提高效率，改写之后。

```
fn square_sum(n: u64) -> u64 {
    let mut m = n;
    let mut s = 0;
    while m != 0 {
        s += (m % 10) * (m % 10);
        m /= 10;
    }
    s
}
```

2）循环求解

```
fn main() {
    let mut count = 0;
    for i in 1..10_000_000 {
        if square_chain_arrive(i) == 89 {
            count += 1;
        }
    }
    println!("{}", count);
}

fn square_chain_arrive(n: u64) -> u64 {
    let mut x = n;
    while x != 1 && x != 89 {
        x = square_sum(x);
    }
    x
}
```

主程序也可以写成一行。

```
println!("{}", (1..10_000_000).filter(|&x| square_chain_arrive(x) ==
 89).count());
```

5 因子

第 12 题 因子繁多的三角数

第 n 个三角数就是从 1 一直累加到 n 得到的数，比如第 7 个三角数，就是 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28。而 28 一共有 6 个因子：1, 2, 4, 7, 14, 28。

求有超过 500 个因子的三角数。

求所有因子，可以试着取余数就行。

```rust
fn factors(num :u32) -> Vec<u32> {
    (1..=num).filter(|x| num % x == 0).collect::<Vec<u32>>()
}
```

然后暴力尝试即可，但效率非常非常差，10 分钟也没有结果。

```rust
for i in 1.. {
    let num = (1..=i).sum::<u32>();
    let f = factors(num);
    if f.len() > 500 {
        println!("i:{} num:{} len:{} {:?}", i, num, f.len(), f );
        println!("{}", num );
        break;
    }
}
```

可以稍做优化，因为因子都是成对出现的，只要尝试一半的因子就行，速度大幅提高，几秒钟可以计算完成。

```rust
fn main() {
    for i in 2.. {
        let num = (1..=i).sum::<u32>();
        let f = half_factors(num);
        if f.len() * 2 > 500 {
            println!("{}", num );
            break;
        }
    }
}

fn half_factors(num :u32) -> Vec<u32> {
    let s = (num as f32).sqrt() as u32;
    (1..=s).filter(|x| num % x == 0).collect::<Vec<u32>>()
}
```

实际上还可以利用因子的数学性质进一步优化，提高上千倍不止，这里不展开讨论了。

另外，主程序可以写成一行，练习一下：

```rust
println!("{}",
    (2..).map(|i| (1..=i).sum::<u32>())
         .filter(|&x| factors(x).len() * 2 > 500)
         .next()
         .unwrap()
);
```

还可以写成这样：

```rust
println!("{}",
    (2..).map(|i| (1..=i).sum::<u32>())
         .find(|&x| factors(x).len() * 2 > 500)
```

```
        .unwrap()
);
```

第 21 题 亲和数

求 10000 之内的所有亲和数之和。

所谓亲和数，是指两个正整数中，彼此的全部约数之和（本身除外）与另一方相等。比如，220 的因子有 1，2，4，5，10，11，20，22，44，55 和 110，因子之和是 284，而 284 的所有因子是 1，2，4，71 和 142，因子之和是 220。

问题分解：

1）求所有因子

2）因子求和

3）找出亲和数，累加求和。

在第 12 题里已经求出了一半的因子，函数是：

```
fn half_factors(num: u32) -> Vec<u32> {
    let s = (num as f32).sqrt() as u32;
    (1..=s).filter(|x| num % x == 0).collect::<Vec<u32>>()
}
```

很容易补上后面的一半因子。

```
fn proper_divisors(num: u32) -> Vec<u32> {
    let mut v = half_factors(num);
    for i in (1..v.len()).rev() { //不要num自身，所以从1开始
        v.push(num / v[i]);
    }
    v
}
```

所有因子求和：

```
fn proper_divisors_sum(num: u32) -> u32 {
    let divs = proper_divisors(num);
    divs.iter().sum::<u32>()
}
```

主程序暴力循环即可，10000 之内只找到 5 对亲和数：

```
let mut sum = 0;
for a in 1u32..10000 {
    let b = proper_divisors_sum(a);
    if a != b && proper_divisors_sum(b) == a {
        sum += a;
        println!("{} {}", a, b);
    }
}
println!("{}", sum);
```

因为亲和数是成对出现的，还可以优化性能，引入一个布尔数组，这里不再展开讨论了。

主程序也可以用函数式编程，但不好理解，不推荐这种写法。

```
println!(
    "{:?}",
    (1_u32..10000)
        .map(|a| (a, proper_divisors_sum(a)))
        .filter(|&(a, b)| a != b && proper_divisors_sum(b) == a)
        .unzip::<_, _, Vec<_>, Vec<_>>().0
        .iter()
        .sum::<u32>()
)
```

第 23 题 非盈数之和

完全数是指真因数之和等于自身的那些数。例如，28 的真因数之和为 1 + 2 + 4 + 7 + 14 = 28，因此 28 是一个完全数。

一个数 n 被称为**亏数**，如果它的真因数之和小于 n；反之则被称为**盈数**。

由于 12 是最小的盈数，它的真因数之和为 1 + 2 + 3 + 4 + 6 = 16，所以最小的能够表示成两个盈数之和的数是 24。通过数学分析可以得出，所有大于 28 123 的数都可以被写成两个盈数的和；尽管我们知道最大的不能被写成两个盈数的和的数要小于这个值，但这是通过分析所能得到的最好上界。

找出所有不能被写成两个盈数之和的正整数，并求它们的和。

解题思路：

1）求所有因子（不包含自身）

2）判断是否为盈数

3）判断是否可以分解为 2 个盈数之和

4）求解最后的问题

第一步求因子，在第 12 题和第 21 题中已经求过，但这里发现它的一个 BUG，对于 4，9，16，25 这样的完全平方数，因子会多出来一个。修改后是这样：

```
fn proper_divisors(num: u32) -> Vec<u32> {
    let mut v = { // 求一半的因子
        let s = (num as f32).sqrt() as u32;
        (1..=s).filter(|x| num % x == 0).collect::<Vec<u32>>()
    };
    let last = v.last().unwrap();
    if last * last == num {
        // 16的一半因子为1,2,4，另外只差一个8，即16 / 2
        for i in (1..v.len()-1).rev() {
            v.push(num / v[i]);
        }
    }
    else {
        // 12的一半因子为1,2,3，另外一半因子：4,6，分别对应于12/3，12/2
        for i in (1..v.len()).rev() {//不要num自身，所以从1开始
            v.push(num / v[i]);
        }
    }
    v
}
```

第二步判断是否为盈数，逻辑简单。

```
fn is_abundant_number(num: u32) -> bool {
    let proper_divisors_sum = proper_divisors(num).iter().sum::<u32>
();
    proper_divisors_sum > num
}
```

第三步，进行分解判断时，出于性能考虑，需要将盈数的结果缓存在一个数组中。

```
let mut abundant_numbers = vec![false; 28124];
for i in 2usize..abundant_numbers.len() {
    if is_abundant_number(i as u32) {
        abundant_numbers[i] = true;
    }
}
```

判断是否可以分解为 2 个盈数，只需暴力循环。

```
fn can_divide(abundant_numbers: &[bool], num: u32) -> bool {
    for x in 1..=28123 {
        let y = num - x;
```

```
        if y <= 0 {break;}
         if abundant_numbers[x as usize] && abundant_numbers[y as usi
ze] {
            // println!("{} = {} + {}", num, x, y);
            return true;
        }
    }
    false
}
```

第四步，把不可分解的数字求和。

```
let mut sum = 0;
for i in 1..=28123 {
    if !can_divide(&abundant_numbers, i) {
        sum += i;
    }
}
println!("sum: {}", sum);
```

当然这求和的几行语句，也可以用函数式编程把它浓缩在一行：

```
println!("sum: {}",
     (1..=28123).filter(|&x| !can_divide(&abundant_numbers, x))
                .sum::<u32>());
```

语法知识点：

✧ 数组作为函数参数的写法：&[bool]

第 47 题 不同的质因数

首次出现连续两个数均有两个不同的质因数是在：

$$14 = 2 \times 7$$

$$15 = 3 \times 5$$

首次出现连续三个数均有三个不同的质因数是在：

$$644 = 2^2 \times 7 \times 23$$

$$645 = 3 \times 5 \times 43$$

$$646 = 2 \times 17 \times 19$$

首次出现连续四个数均有四个不同的质因数时，其中的第一个数是多少？

解题步骤：

primes 函数库里有一个求不同质因数的函数 factors_uniq()，直接拿来用即可，程序变得异常简单，这个程序中仍有一些重复的质因子计算，还可以进一步优化。

```
fn main() {
    for n in 2.. {
        if has_four_factors_uniq(n) {
            println!("{}", n);
            break;
        }
    }
}

fn has_four_factors_uniq(n: u64) -> bool {
    let xf = primes::factors_uniq(n);
    if xf.len() != 4 {
        return false;
    }
    for i in 1..=3 {
        let yf = primes::factors_uniq(n + i);
        if yf.len() != 4 || xf == yf {
            return false;
        }
    }
    true
}
```

主程序还可以采用 filter()的写法，更简洁一些。

```
let n = (2..).filter(|x| has_four_factors_uniq(*x)).next().unwrap();
println!("{:?}", n);
```

6 素数

欧拉计划中有大量与素数有关的算法题。

第 7 题 第 10001 个素数

求第 10001 个素数。

按通常的逐个试余法，效率极差，需要用著名的**筛子求素数**算法，请自行百度。从网上找来其它语言的源代码，稍做修改即可。

```
let max_number_to_check = 1_000_000;
```

```
let mut prime_mask = vec![true; max_number_to_check];
prime_mask[0] = false;
prime_mask[1] = false;

let mut total_primes_found = 0;

const FIRST_PRIME_NUMBER: usize = 2;
for p in FIRST_PRIME_NUMBER..max_number_to_check {
    if prime_mask[p] {
        // println!("{}", p);
        total_primes_found += 1;
        if total_primes_found == 10001 {
            println!("the 10001st prime number is : {}", p);
            break;
        }
        let mut i = 2 * p;
        while i < max_number_to_check {
            prime_mask[i] = false;
            i += p;
        }
    }
}
```

筛子算法需要提前分配内存空间，所以指定一个足够大的搜索范围 max_number_to_check，还需要一个数组 prime_mask 存放素数的标识位，另外还用 total_primes_found 对找到的素数进行计数。

这里有一个常量声明的语法点：

```
const FIRST_PRIME_NUMBER : usize = 2;
```

第 10 题 素数的和

求小于 2 百万的所有素数之和。

第 7 题已经会求素数，可以重新利用，我们可以将它封装为模块，在 Rust 里称为 mod。

Rust 的开发环境对模块的支持也非常方便，可以新建一个 myprime.rs 文件，稍微修改以前的代码，形成一个 prime_sieve()函数。注意函数的最前面加 pub 关键字，意思是这个函数是公有的，对外界可见。

```
pub fn prime_sieve(max_number_to_check:usize) -> Vec<bool> {
    let mut prime_mask = vec![true; max_number_to_check];
    prime_mask[0] = false;
```

```
    prime_mask[1] = false;

    const FIRST_PRIME_NUMBER: usize = 2;
    for p in FIRST_PRIME_NUMBER..max_number_to_check {
        if prime_mask[p] {
            let mut i = 2 * p;
            while i < max_number_to_check {
                prime_mask[i] = false;
                i += p;
            }
        }
    }
    prime_mask
}
```

主程序 main.rs 里需要加一行语句，告诉程序要用到 myprime 模块。

```
mod myprime;
```

cargo 工具会自动找到 myprime.rs 文件进行编译，并与主程序链接在一起生成 exe 执行文件。有了 myprime 模块，主程序逻辑非常简单。

```
let prime_masks = myprime::prime_sieve(2_000_000);

let mut sum = 0;
for (i, &mask) in prime_masks.iter().enumerate() {
    if mask {
        sum += i as u64;
    }
}
println!("{}", sum);
```

Rust 社区有大量的程序员已经贡献了非常成熟的函数库，我们不再重复发明轮子，可以直接使用别人写好的 primes 函数库，需要在 toml 文件中增加一行依赖项。

```
[dependencies]
primes = "0.2"
```

维护 toml 这个文件也有工具可以搞定，需要安装 cargo-edit，使用命令

```
cargo install cargo-edit
```

安装完成之后，就可以使用下面这些命令来自动维护 toml 文件。

```
cargo add primes
cargo upgrade primes
cargo rm primes
```

站在巨人的肩膀上，这道题可以一行语句搞定。

```
println!("{}", (2..2_000_000).filter(|x| primes::is_prime(*x)).sum::
<u64>());
```

第 27 题 二次多项式生成素数

欧拉发现了这个著名的二次多项式：

$$n^2 + n + 41$$

对于连续的整数 n 从 0 到 39，这个二次多项式生成了 40 个素数。然而，当 n = 40 时，$40^2 + 40 + 41 = 40(40 + 1) + 41$ 能够被 41 整除，同时显然当 n = 41 时，$41^2 + 41 + 41$ 也能被 41 整除。

随后，另一个神奇的多项式 $n^2 - 79n + 1601$ 被发现了，对于连续的整数 n 从 0 到 79，它生成了 80 个素数。这个多项式的系数-79 和 1601 的乘积为-126479。

考虑以下形式的二次多项式：

$n^2 + an + b$，满足$|a| < 1000$ 且$|b| < 1000$

其中$|n|$指 n 的模或绝对值

例如$|11| = 11$ 以及$|-4| = 4$

这其中存在某个二次多项式能够对从 0 开始尽可能多的连续整数 n 都生成素数，求其系数 a 和 b 的乘积。

先借鉴第 7 题中的素数算法，将 2 百万之内的素数都求出来，公式 n*n+*a*n+b* 最大取值不会超过 2 百万。

```
let max_number_to_check = 2_000_000;

let mut prime_mask = vec![true; max_number_to_check];
prime_mask[0] = false;
prime_mask[1] = false;

const FIRST_PRIME_NUMBER: usize = 2;
for p in FIRST_PRIME_NUMBER..max_number_to_check {
    if prime_mask[p] {
        let mut i = 2 * p;
        while i < max_number_to_check {
            prime_mask[i] = false;
            i += p;
        }
```

```
    }
}
```

求方程得到的连续素数的个数。这里要用 isize，因为求值时可能会出现负数，如果用 usize，运行时会发生溢出错误。

```
fn consecutive_primes(prime_mask: &[bool], a: isize, b: isize) -> u32
 {
    for n in 0..1000 {
        let y: isize = n * n + a * n + b;
        if y < 0 || !prime_mask[y as usize] {
            return n as u32;
        }
    }
    0
}
```

最后，进行暴力循环即可，能够连续生成 71 个素数，有点不可思议。

```
let mut max_prime_len = 0;
for a in -999..=999 {
    for b in -1000..=1000 {
        let prime_series_len = consecutive_primes(&prime_mask, a, b);
        if prime_series_len > max_prime_len {
            max_prime_len = prime_series_len;
            println!(
                "primes: {} a: {} b: {}   a * b = {}",
                prime_series_len, a, b, a * b
            );
        }
    }
}
```

使用 primes 函数库的写法：

```
fn main() {
    let mut max_prime_len = 0;
    for a in -999..=999 {
        for b in -1000..=1000 {
            let prime_series_len = consecutive_primes(a, b);
            if prime_series_len > max_prime_len {
                max_prime_len = prime_series_len;
                println!(
                    "primes: {} a: {} b: {}   a * b = {}",
                    prime_series_len,. a, b, a * b
                );
            }
        }
    }
}

// 公式可以生成多少个连续的素数
fn consecutive_primes(a: i64, b: i64) -> u64 {
```

```
    for n in 0..1000 {
        // 这里求值时，可能会出现负数，如果用usize，运行时会出现溢出错误
        let y: i64 = n * n + a * n + b;
        if y < 0 || !primes::is_prime(y as u64) {
            return n as u64;
        }
    }
    0
}
```

第 35 题 旋转素数

数字 197 称为旋转素数，因为它的几个数字经过轮转之后，197、971 和 719 也都是素数，在 100 以内共有 13 个这样的素数：2，3，5，7，11，13，17，31，37，71，73，79，和 97，问 100 万之内有几个旋转素数？

解题思路：

1）旋转一个数

最容易想到的思路是取出最左边的数字，放到字符串的最右侧。

```
fn rotate_v1(n:u64) -> u64 {
    let mut s = n.to_string();
    let ch = s.chars().next().unwrap();
    s = s[1..].to_string();
    s.push(ch);
    s.parse::<u64>().unwrap()
}
```

因为 remove()函数在移除最左侧的字符时，存储了该字符，直接放在右侧即可，代码更简洁和高效一些。

```
fn rotate(n: u64) -> u64 {
    let mut s = n.to_string();
    let ch = s.remove(0);
    s.push(ch);
    s.parse::<u64>().unwrap()
}
```

2）判断是否为旋转素数

另外，要注意处理一下数字里带 0 的特殊情况。

```
fn is_rotate_prime(n: u64) -> bool {
    if n.to_string().contains('0') {
        return false;
    }
```

```
    let mut r = n;
    for _i in 0..n.to_string().len() {
        if !primes::is_prime(r) {
            return false;
        }
        r = rotate(r);
    }
    true
}
```

3）主程序就非常容易了。

```
let mut count_primes = 0;
for n in 2..1_000_000 {
    if is_rotate_prime(n) {
        println!("{}", n);
        count_primes += 1;
    }
}
println!("{}", count_primes);
```

还可以用函数式写法：

```
let count_primes = (2..1_000_000)
        .filter(|&x| is_rotate_prime(x)).count();
println!("{}", count_primes);
```

语法知识点：

1）字符串的 remove()函数和 push()函数。

2）字符串的 parse()函数可以转换成数值类型。

第 37 题 左截和右截素数

3797 有一个有趣的属性，它本身是素数，另外从左向右依次删除一个数字，得到：797，97，和 7，仍是素数，依次从右向左删除一个数字，得到：379，37，和 3，仍是素数。

总共只有 11 个这样的素数，请求它们的和。

注意：2，3，5 和 7 不计算在内。

解题思路

判断是否为左截素数，循环调用 remove()函数即可。

```
fn is_trunc_left_prime(n: u64) -> bool {
    let mut s = n.to_string();
    while s.len() > 0 {
        let p = s.parse::<u64>().unwrap();
        if !primes::is_prime(p) {
            return false;
        }
        s.remove(0);
    }
    true
}
```

类似的，换成 pop()函数，可以判断右截素数。

```
fn is_trunc_right_prime(n: u64) -> bool {
    let mut s = n.to_string();
    while s.len() > 0 {
        let p = s.parse::<u64>().unwrap();
        if !primes::is_prime(p) {
            return false;
        }
        s.pop();
    }
    true
}
```

只用除法也可以生成右截数，效率应该更快一点。

```
fn is_trunc_right_prime(n: u64) -> bool {
    let mut m = n;
    while m > 0 {
        if !primes::is_prime(m) {
            return false;
        }
        m /= 10;
    }
    true
}
```

题目告诉了只有 11 个这样的素数，暴力搜索到 11 个即可，主程序的写法。

```
let mut count = 0;
let mut sum = 0;
for n in 10.. {
    if is_trunc_left_prime(n) && is_trunc_right_prime(n) {
        println!("{}", n);
        count += 1;
        sum += n;
        if count == 11 {break;}
    }
}
println!("sum: {}", sum);
```

还可以练习函数式的一行语句的写法。

```
println!("{}",
     (10..).filter(|&n| is_trunc_left_prime(n) && is_trunc_right_pri
me(n))
    .take(11)
    .sum::<u64>());
```

第 50 题 连续素数的和

素数 41 可以写成六个连续素数的和：

41 = 2 + 3 + 5 + 7 + 11 + 13

在小于一百的素数中，41 能够被写成最多的连续素数的和。

在小于一千的素数中，953 能够被写成最多的连续素数的和，共包含连续 21 个素数。

在小于一百万的素数中，哪个素数能够被写成最多的连续素数的和？

算法比较简单，记录好起始的素数及连续素数的长度，暴力搜索即可。里面有大量的重复求和计算，感兴趣的话，可以继续优化效率。

```
fn main() {
    let limit = 1_000_000;
    // 记录连续素数的长度
    let mut prime_len = 1;
    for start in 2..=limit {
        if !primes::is_prime(start) {
            continue;
        }
        let mut count = 1;
        let mut sum = start;
        for i in start + 1..=limit {
            if primes::is_prime(i) {
                count += 1;
                sum += i;
                if sum >= limit {
                    break;
                }
                if count > prime_len && primes::is_prime(sum) {
                    prime_len = count;
                    println!("start: {} consecutive primes len: {} su
m: {}", start, prime_len, sum);
                }
            }
        }
    }
}
```

第 58 题 螺旋素数

从 1 开始逆时针螺旋着摆放自然数，我们可以构造出一个边长为 7 的螺旋数阵。

```
37 36 35 34 33 32 31
38 17 16 15 14 13 30
39 18  5  4  3 12 29
40 19  6  1  2 11 28
41 20  7  8  9 10 27
42 21 22 23 24 25 26
43 44 45 46 47 48 49
```

可以发现，所有的奇数平方都在这个螺旋方针的右下对角线上，更有趣的是，在所有对角线上一共有 8 个素数，比例达到 8/13 ≈ 62%。

在这个方阵外面完整地再加上一层，就能构造出一个边长为 9 的螺旋方阵。如果不断重复这个过程，当对角线上素数的比例第一次低于 10%时，螺旋数阵的边长是多少？

解题步骤：

观察数字的规律，每一圈的右下角是边长的完全平方数，其它三个角上的数递减，而且差值为（边长-1），这样计算四个角上的数使用一点小技巧。

```
37 36 35 34 33 32 31
38 17 16 15 14 13 30
39 18  5  4  3 12 29
40 19  6  1  2 11 28
41 20  7  8  9 10 27
42 21 22 23 24 25 26
43 44 45 46 47 48 49
```

大量素数的判断时，用 PrimeSet，比 primes::is_prime()要快。

```rust
use primes::PrimeSet;

fn main() {
    let mut pset = PrimeSet::new();
```

```
    let mut count_prime = 0;
    for n in (3..).step_by(2) { // 边长
        let lower_right = n * n;
        let prime_four_corner = (0..4)
            .map(|i| lower_right - (n - 1) * i)
            .filter(|&x| pset.is_prime(x));
        count_prime += prime_four_corner.count();

        let percent = (count_prime as f32) / ((2 * n - 1) as f32);
        if percent < 0.1 {
            println!("{} count: {} percent: {}", n, count_prime, perc
ent);
            break;
        }
    }
}
```

第 97 题 非梅森大素数

1999 年人们发现了第一个超过一百万位的素数，这是一个梅森素数，可以表示为 $2^{6972593}-1$，包含有 2,098,960 位数字。在此之后，更多形如 2^p-1 的梅森素数被发现，其位数也越来越多。

然而，在 2004 年，人们发现了一个巨大的非梅森素数，包含有 2,357,207 位数字：$28433\times2^{7830457}+1$。

找出这个素数的最后十位数字。

解题步骤：

结果只取最后 10 位数字，不用大整数函数库，求模就行。

```
let mut p = 28433_u64;
for _i in 0..7830457 {
    p = p * 2 % 10_000_000_000;
}
println!("{}", p + 1);
```

可以练习一下 fold() 的写法：

```
let mersenne = [2; 7830457]
    .iter()
    .fold(28433_u64, |s, x| s * x % 10_000_000_000)
    + 1;
println!("{}", mersenne);
```

7 数字游戏

第 9 题 特殊勾股数

找到和为 1000 的勾股数，并求积。

简单粗暴地遍历求解即可。

```
for a in 1..1000 {
    for b in a..1000 {
        let c = 1000 - a - b;
        if c > 0 && a*a + b*b == c*c {
            println!("{} = {} x {} x {}", a*b*c, a, b, c);
            return;
        }
    }
}
```

Python 语言中，可以使用列表推导的语法，写成一行语句：

```
print([a*b*(1000-a-b) for a in range(1,1000 ) for b in range(a,1000)
if 1000-a-b>0 and a*a+b*b==(1000-a-b)*(1000-a-b)])
```

第 11 题 方阵中的最大乘积

在一个矩阵里，找到一条线上、相邻的、乘积最大的 4 个数，求积。

先把数值用二维数组表示。

```
let arr = [
  [08,02,22,97,38,15,00,40,00,75,04,05,07,78,52,12,50,77,91,08],
  [49,49,99,40,17,81,18,57,60,87,17,40,98,43,69,48,04,56,62,00],
  [81,49,31,73,55,79,14,29,93,71,40,67,53,88,30,03,49,13,36,65],
  [52,70,95,23,04,60,11,42,69,24,68,56,01,32,56,71,37,02,36,91],
  [22,31,16,71,51,67,63,89,41,92,36,54,22,40,40,28,66,33,13,80],
  [24,47,32,60,99,03,45,02,44,75,33,53,78,36,84,20,35,17,12,50],
  [32,98,81,28,64,23,67,10,26,38,40,67,59,54,70,66,18,38,64,70],
  [67,26,20,68,02,62,12,20,95,63,94,39,63,08,40,91,66,49,94,21],
  [24,55,58,05,66,73,99,26,97,17,78,78,96,83,14,88,34,89,63,72],
  [21,36,23,09,75,00,76,44,20,45,35,14,00,61,33,97,34,31,33,95],
  [78,17,53,28,22,75,31,67,15,94,03,80,04,62,16,14,09,53,56,92],
  [16,39,05,42,96,35,31,47,55,58,88,24,00,17,54,24,36,29,85,57],
  [86,56,00,48,35,71,89,07,05,44,44,37,44,60,21,58,51,54,17,58],
  [19,80,81,68,05,94,47,69,28,73,92,13,86,52,17,77,04,89,55,40],
```

```
  [04,52,08,83,97,35,99,16,07,97,57,32,16,26,26,79,33,27,98,66],
  [88,36,68,87,57,62,20,72,03,46,33,67,46,55,12,32,63,93,53,69],
  [04,42,16,73,38,25,39,11,24,94,72,18,08,46,29,32,40,62,76,36],
  [20,69,36,41,72,30,23,88,34,62,99,69,82,67,59,85,74,04,36,16],
  [20,73,35,29,78,31,90,01,74,31,49,71,48,86,81,16,23,57,05,54],
  [01,70,54,71,83,51,54,69,16,92,33,48,61,43,52,01,89,19,67,48],
];
```

先不考虑代码的啰嗦和美观性，4 个方向都比较一遍，找出最大的即可。

```
let mut max = 0;
for i in 0..20 {
    for j in 0..20 {
        if i+4<=20 {
            let p = arr[i][j] * arr[i+1][j] * arr[i+2][j] * arr[i+3]
[j];
            if p > max {
                max = p;
                println!("下 {} {} {}",i,j, max);
            }
        }
        if j<=20-4 {
            let p = arr[i][j] * arr[i][j+1] * arr[i][j+2] * arr[i][j+
3];
            if p > max {
                max = p;
                println!("右 {} {} {}",i,j, max);
            }
        }
        if i<=20-4 && j<=20-4 {
            let p = arr[i][j] * arr[i+1][j+1] * arr[i+2][j+2] * arr[i
+3][j+3];
            if p > max {
                max = p;
                println!("右下 {} {} {}",i,j, max);
            }
        }
        if i<=20-4 && j>=3 {
            let p = arr[i][j] * arr[i+1][j-1] * arr[i+2][j-2] * arr[i
+3][j-3];
            if p > max {
                max = p;
                println!("左下 {} {} {}",i,j, max);
            }
        }
    }
}
println!("{}", max);
```

代码里存在大量的与 max 比较的重复代码，可以用 k=0,1,2,3 代表四个方向，多一个内层循环，让代码简洁一点。

```
for k in 0..4 {
    let mut p = 0;
    if k == 0 && i+4<=20 {
        p = arr[i][j] * arr[i+1][j] * arr[i+2][j] * arr[i+3][j];
    }
    if k == 1 && j<=20-4 {
        p = arr[i][j] * arr[i][j+1] * arr[i][j+2] * arr[i][j+3];
    }
    if k == 2 && i<=20-4 && j<=20-4 {
        p = arr[i][j] * arr[i+1][j+1] * arr[i+2][j+2] * arr[i+3][j+
3];
    }
    if k == 3 && i<=20-4 && j>=3 {
        p = arr[i][j] * arr[i+1][j-1] * arr[i+2][j-2] * arr[i+3][j-
3];
    }

    if p > max {
        max = p;
        println!("{} {} {}",i,j, max);
    }
}
```

语法知识点：

✧ 二维数组的表示法

第 28 题 螺旋数阵对角线

从 1 开始，按顺时针顺序向右铺开的 5 × 5 螺旋数阵如下所示：

21	22	23	24	**25**
20	**7**	8	**9**	10
19	6	**1**	2	11
18	**5**	4	**3**	12
17	16	15	14	**13**

可以验证，该数阵对角线上的数之和是 101。

以同样方式构成的 1001 × 1001 螺旋数阵对角线上的数之和是多少？

求解思路：

找规律，右上角那个数是完全平方数，其它三个角的数正好递减。

```
fn main() {
```

```
    let mut sum = 1;
    for i in (3..=1001).step_by(2) {
        let upperright = i * i;
        sum += upperright;
        sum += upperright - (i - 1) * 1;
        sum += upperright - (i - 1) * 2;
        sum += upperright - (i - 1) * 3;
    }
    println!("{}", sum);
}
```

知识点：

✧ 步长大于 1 的迭代器用 step_by()。

第 30 题 各位数字的五次幂

令人惊讶的是，只有三个数可以写成它们各位数字的四次幂之和：

$$1634 = 1^4 + 6^4 + 3^4 + 4^4$$

$$8208 = 8^4 + 2^4 + 0^4 + 8^4$$

$$9474 = 9^4 + 4^4 + 7^4 + 4^4$$

由于 $1 = 1^4$ 不是一个和，所以这里并没有把它包括进去。

这些数的和是 1634 + 8208 + 9474 = 19316。

找出所有可以写成它们各位数字的五次幂之和的数，并求这些数的和。

解题思路：

一个关键的表达式，求各位数字的 5 次方，再求和。

```
n.to_string()
 .chars()
 .map(|c| c.to_digit(10).unwrap().pow(5))
 .sum::<u32>();
```

主程序就非常简单了。

```
let mut s = 0;
for n in 2..999999 {
    let sum_pow = n
        .to_string()
```

```
        .chars()
        .map(|c| c.to_digit(10).unwrap().pow(5))
        .sum::<u32>();
    if sum_pow == n {
        println!("{}", n);
        s += n;
    }
}
println!("sum: {}", s);
```

也可以写一个函数 is_power_number()，再利用 filter 函数一行完成。

```
fn is_power_number(n: u32) -> bool {
    n == n.to_string()
          .chars()
          .map(|c| c.to_digit(10).unwrap().pow(5))
          .sum::<u32>()
}

fn main() {
    let sum_pow = (2..999999).filter(|&x| is_power_number(x)).sum::<u
32>();
    println!("{}", sum_pow);
}
```

第 32 题 全数字的乘积

如果一个 n 位数包含了 1 至 n 的所有数字恰好一次，我们称它为全数字的；例如，五位数 15234 就是 1 至 5 全数字的。

7254 是一个特殊的乘积，因为在等式 39 × 186 = 7254 中，被乘数、乘数和乘积恰好是 1 至 9 全数字的。

找出所有被乘数、乘数和乘积恰好是 1 至 9 全数字的乘法等式，并求出这些等式中乘积的和。

注意：有些乘积可能从多个乘法等式中得到，但在求和的时候只计算一次。

解题思路：

1）判断一个字符串中只能出现一次的 1 到 9

2）循环尝试，记录每一个满足要求的乘积

3）求和

第一步，先写一个判断字符串里只能出现一次 1 到 9 的函数。

```
fn exists_only_once_1_to_9(s: &str) -> bool {
    let mut has_digit = vec![false; 10];
    for ch in s.to_string().chars() {
        let c = ch.to_digit(10).unwrap() as usize;
        if c == 0 { // 不允许0的存在
            return false;
        }
        if has_digit[c] {
            return false;
        }
        has_digit[c] = true;
    }
    true
}
```

第二步和第三步，比较简单，循环的终止条件要考虑一下，被乘数和乘数都不能超过 4 位数，所以只需要试验到 9876。

```
let mut v: Vec<u32> = vec![];
for a in 2..9876 {
    for b in 2..a {
        let c = a * b;
        let abc = a.to_string() + &b.to_string() + &c.to_string();
        if abc.len() == 9 && exists_only_once_1_to_9(&abc) {
            println!("{} x {} = {}", a, b, c);
            if !v.contains(&c) {
                v.push(c);
            }
        }
    }
}
println!("{}", v.iter().sum::<u32>());
```

程序到这里已经可以跑起来了，但运行起来较慢，发现少了一条重要的优化语句，乘积在大于 9876 时，后面的数都不用试了。

在循环体加一条判断语句，程序在 1 秒之内跑完。

```
if c > 9876 {break;}
```

第 34 题 各位数字的阶乘

145 是个有趣的数，因为 1! + 4! + 5! = 1 + 24 + 120 = 145。

找出所有各位数字的阶乘和等于其本身的数，并求它们的和。

注意：因为 1! = 1 和 2! = 2 不是和的形式，所以它们并不在讨论范围内。

解题思路：

1）求阶乘

2）找出一个数的各位数字

3）循环求解

第一步，阶乘可以用递归实现。

```
fn factorial(n: u32) -> u32 {
    if n <= 1 {
        return 1;
    }
    n * factorial(n - 1)
}
```

1 到 9 的阶乘需要经常用到，用一个数组缓存起来。

```
// [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
let mut fac = vec![1; 10];
for i in 0..=9 {
    fac[i] = factorial(i as u32);
}
```

使用 map()函数，上面几行等价于这样一行：

```
let fac: Vec<u32> = (0..10).map(|x| factorial(x)).collect();
```

后面的逻辑比较简单，我只搜索到了 999999，后面好像不存在满足条件的更大的解。

```
let mut sum = 0;for n in 3..999999 {
    // 各位数字的阶乘之和
    let mut sum_fac = 0;
    for digit in n.to_string().chars()
                  .map(|x| x.to_digit(10).unwrap()) {
        sum_fac += fac[digit as usize];
    }
    if n == sum_fac {
        println!("{}! = {}", n, n);
        sum += n;
    }
}
println!("{}", sum);
```

求 sum_fac 那几行还可以这样写：

```
let sum_fac: u32 = n
    .to_string()
    .chars()
    .map(|x| fac[x.to_digit(10).unwrap() as usize])
    .sum();
```

知识点：

✧ 学会使用 map()函数

✧ chars()和 map()运用，取各位数字

第 36 题 两种进制的回文数

数字 585 是回文数，即从左向右、从右向左读都是一样的，其二进制表示为 1001001001，也是回文数。

请找出 100 万以下的所有回文数，并求和。注意，转换成二进制之后，左侧的 0 不计算在内。

解题步骤：

1）转换成二进制表示

费劲写了一个转换成二进制的函数。

```
fn to_radix2_string(n: u64) -> String {
    let mut d = n;
    let mut s:String = "".to_string();
    while d != 0 {
        let m = d % 2;
        s.push_str(&m.to_string());
        d /= 2;
    }
    s
}
```

发现又在重复造轮子，直接用 format!宏就可以搞定。

```
let s = format!("{:b}", n);
```

2）判断是否回文

比较简单而直接的写法。

```
fn is_palindromic(s: String) -> bool {
    s == s.chars().rev().collect::<String>()
}
```

这里用了 collect()，效率比较低，实际上当发现了首尾不一样的字符时，就应该马上返回 false，而且最多比较一半的字符就行，效率大幅提升。

```
fn is_palindromic(s: String) -> bool {
    let mut c1 = s.chars();
    let mut c2 = s.chars().rev();
    for _i in 0..s.len() / 2 {
        if c1.next().unwrap() != c2.next().unwrap() {
            return false;
        }
    }
    true
}
```

3）最后循环求和

这一步逻辑简单，代码从略。

第 38 题 全数字的倍数

将 192 分别与 1、2、3 相乘：

192 × 1 = 192

192 × 2 = 384

192 × 3 = 576

连接这些乘积，我们得到一个 1 至 9 全数字的数 192384576。我们称 192384576 为 192 和(1,2,3)的连接乘积。

同样地，将 9 分别与 1、2、3、4、5 相乘，得到 1 至 9 全数字的数 918273645，即是 9 和(1,2,3,4,5)的连接乘积。

对于 n > 1，所有某个整数和(1,2, ... ,n)的连接乘积所构成的数中，最大的 1 至 9 全数字的数是多少？

解题步骤：

本题与第 32 题比较相似，有一个判断全数字（有且仅有一次 1 到 9）的函数，可以直接利用。

```
fn exists_only_once_1_to_9(s: &str) -> bool {
    let mut has_digit: Vec<bool> = vec![false; 10];
    for ch in s.to_string().chars() {
        let c = ch.to_digit(10).unwrap() as usize;
        if c == 0 {
            return false;
        }
        if has_digit[c] {
            return false;
        }
        has_digit[c] = true;
    }
    true
}
```

主程序用 2 层循环就可以搞定，运算量不大，不需要优化。

```
fn main() {
    let mut max = "".to_string();
    for a in 1..=9876 {
        let mut s = String::from("");
        for n in 1..=9 {
            let prod = a * n;
            s.push_str(&prod.to_string());
            if !exists_only_once_1_to_9(&s) {
                break;
            }
            if s.len() == 9 && s > max {
                println!("{} x {:?} = {}", a, [1..n], s);
                max = s.clone();
            }
        }
    }
}
```

第 40 题 钱珀瑙恩常数

将所有正整数连接起来构造的一个十进制无理数如下所示：

0.12345678910111213141516171819202 1...

可以看出小数点后第 12 位数字是 1。如果 d_n 表示上述无理数小数点后的第 n 位数字，求下式的值：$d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$

解题思路：

算法很简单，需要留意差 1 的 BUG。

```
let max_digits = 1_000_001;
```

```
let mut digits: Vec<usize> = vec![0; max_digits];
let mut pos = 1;
'a: for i in 1.. {
    for ch in i.to_string().chars() {
        let d = ch.to_digit(10).unwrap() as usize;
        if pos >= max_digits {
            break 'a;
        }
        digits[pos] = d;
        pos += 1;
    }
}
println!("{}", digits[1] * digits[10] * digits[100] * digits[1000]
 * digits[10000] * digits[100000] * digits[1000000]);
```

最后的乘积计算，可以练习一下 map()和 fold()的写法。

```
let d: Vec<usize> = (0..=6)
            .map(|x| digits[10_usize.pow(x)]).collect();
println!("{:?}", d);
let p: usize = (0..=6)
    .map(|x| digits[10_usize.pow(x)])
    .fold(1, |x, a| x * a);
println!("{}", p);
```

第 46 题 哥德巴赫的另一个猜想

克里斯蒂安·哥德巴赫曾经猜想，每个奇合数可以写成一个素数和一个平方的两倍之和。

$$9 = 7 + 2\times1^2$$

$$15 = 7 + 2\times2^2$$

$$21 = 3 + 2\times3^2$$

$$25 = 7 + 2\times3^2$$

$$27 = 19 + 2\times2^2$$

$$33 = 31 + 2\times1^2$$

最终这个猜想被推翻了。

最小的不能写成一个素数和一个平方的两倍之和的奇合数是多少？

解题思路：

程序很简单，暴力搜索即可。

```rust
fn main() {
    for n in (3..).step_by(2) {
        if primes::is_prime(n) {
            continue;
        } // 奇合数
        if !can_divide(n) {
            println!("{}", n);
            break;
        }
    }
}

fn can_divide(n: u64) -> bool {
    let limit = ((n / 2) as f64).sqrt() as u64;
    for i in 1..=limit {
        let p = n - 2 * i * i;
        if primes::is_prime(p) {
            return true;
        }
    }
    false
}
```

第 52 题 重排的倍数

可以看出，125874 和它的两倍 251748 拥有同样的数字，只是排列顺序不同。

有些正整数 x 满足 2x、3x、4x、5x 和 6x 都拥有相同的数字，求其中最小的正整数。

解题思路：

没写程序的时候我已经知道答案了，因为我以前写过《吉利数字》这样一篇文章。

```rust
fn main() {
    for x in 100000..=999999 {
        if is_permuted(x) {
            println!("{}", x);
            break;
        }
    }
}

fn is_permuted(x: u32) -> bool {
```

```
    // 拆成6个数字
    let vx: Vec<u32> = x
        .to_string()
        .chars()
        .map(|c| c.to_digit(10).unwrap())
        .collect();
    for i in 2..=6 {
        let m = i * x;
        let vm: Vec<u32> = m
            .to_string()
            .chars()
            .map(|c| c.to_digit(10).unwrap())
            .collect();
        // 每个数字都在原来的集合里出现过
        for e in vm {
            if !vx.contains(&e) {
                return false;
            }
        }
    }
    true
}
```

第 206 题 被遮挡的平方数

找出唯一一个其平方形如 1_2_3_4_5_6_7_8_9_0 的正整数，其中每个“_”表示一位数字。

解题步骤：

第一种写法：

```
fn main() {
    let start = 1020304050607080900.0_f64.sqrt() as u64;
    let end = 1929394959697989990.0_f64.sqrt() as u64;
    for i in start..=end {
        if i % 10 != 0 {
            //最后1位必须是0
            continue;
        }
        let m = i * i;
        if meet_cond(m) {
            println!("{} {}", i, m);
            break;
        }
    }
}

fn meet_cond(m: u64) -> bool {
```

```
    let chars: Vec<u32> = m
        .to_string()
        .chars()
        .map(|c| c.to_digit(10).unwrap())
        .collect();

    for j in 0..9 {
        if chars[j * 2] != (j + 1) as u32 {
            return false;
        }
    }
    true
}
```

可以优化性能，步长为 10，全用整数的除法和取模运算，性能提升非常大。

```
fn main() {
    let start = (1020304050607080900.0_f64.sqrt() as u64) / 10 * 10;
    let end = 1929394959697989990.0_f64.sqrt() as u64;
    for i in (start..=end).step_by(10) {
        let m = i * i;
        if meet_cond(m) {
            println!("{} {}", i, m);
            break;
        }
    }
}

// 1_2_3_4_5_6_7_8_9_0
fn meet_cond(m: u64) -> bool {
    let mut x = m / 100;
    let mut digit = 9;
    while x != 0 {
        if x % 10 != digit {
            return false;
        }
        x /= 100;
        digit -= 1;
    }
    true
}
```

8 大整数

第 13 题 大整数求和

有 100 个长达 50 位的大整数，求和，只取前 10 位数字。

各种编程语言都有大整数的函数库，直接使用就行了，不用自己造轮子。在 Rust 里一样也有大量的现成的库，称为 crate，这个单词翻译为“柳条箱”，不知道官方的翻译是什么。大整数的官方实现是 num_bigint。

需要修改 Cargo.toml 文件：

```
[dependencies]
num-bigint = "0.2.2"
```

文件头加上相关的引用：

```
extern crate num_bigint;
use num_bigint::BigUint;
```

100 个大整数这里用字符串数组表示。

```
let numbers = [
"371072875339021027987979982208375902465101357402 50",
"463769376774900097126481248969700780504170182605 38",
"743249861995247410594742333095130581237266173096 29",
"229188020587773197198394501808880724296619808111 97",
// 省略了很多行
"771585425020165450904132458097868827789487218596 17",
"721078384350691861554356628840622574736922845095 16",
"208496039801340017239306716668235552452528046097 22",
"535035342264725242508740540755917897812643303316 90",
];
```

这里只用到了正整数 BigUint，由于 Rust 是强类型语言，所以想办法把字符串转换为 BigUint。

```
let mut sum = BigUint::from(0 as u64);
for s in numbers.iter() {
    sum += BigUint::parse_bytes(s.as_bytes(), 10).unwrap();
}
let full_str = sum.to_string();
println!("take 10 digits: {}", &full_str[..10]);
```

结果很长，只取前 10 个数字，用到字符串的切片函数 &full_str[..10]。

第 16 题 幂的数字和

求 2 的 1000 次方的所有数字之和。

同样用到大整数的计算函数库 num_bigint，注意添加依赖项。

```
extern crate num_bigint;
use num_bigint::BigUint;
```

大整数里没有 power()函数，可以把 2 相乘 1000 次。

```
let mut prod = BigUint::from(1 as u64);
for _i in 0..1000 {
    prod *= BigUint::from(2 as u64);
}
let full_str = prod.to_string();
println!("{}", full_str);
```

在 for 循环里变量 i 并没有使用，所有前面添加一个下划线，可以不出现编译警告。

还可以学习一下函数式编程里的 fold()的写法，用一行语句，但理解起来比前面的 4 行语句难一些。

```
let pow2_1000 = (0..1000)
.fold(BigUint::from(1 as u64), |p, _a| p*BigUint::from(2 as u64));
println!("{}", pow2_1000);
```

在第 8 题里学过把字符串切成一个个的数字，这里相加即可。

```
let s = full_str
    .chars()
    .map(|c| c.to_digit(10).unwrap())
    .sum::<u32>();
println!("{}", s);
```

第 20 题 阶乘数字和

求 100 的阶乘中所有数字之和。

本题与第 16 题非常相似，稍微修改就出来。

```
let mut prod = BigUint::from(1 as u64);
for i in 1..=100 {
    prod *= BigUint::from(i as u64);
}

let full_str = prod.to_str_radix(10);
let s = full_str
    .chars()
    .map(|c| c.to_digit(10).unwrap())
    .sum::<u32>();
println!("{}", s);
```

第 25 题 一千位斐波那契数

在斐波那契数列中，第一个有 1000 位数字的是第几项？

本题与第 16 题非常相似，稍微修改就出来，不再详细解释。

```
extern crate num_bigint;
use num_bigint::BigUint;
fn main() {
    let mut prev = BigUint::from(1 as u64);
    let mut cur = BigUint::from(1 as u64);
    for i in 3.. {
        let next = prev + &cur;
        let str = next.to_string();
        if str.len() >= 1000 {
            println!("{} {} {}", i, str, str.len());
            break;
        }
        prev = cur;
        cur = next;
    }
}
```

第 29 题 不同的幂

考虑所有满足 $2 \le a \le 5$ 和 $2 \le b \le 5$ 的整数组合生成的幂 a^b：

$2^2=4$，$2^3=8$，$2^4=16$，$2^5=32$

$3^2=9$，$3^3=27$，$3^4=81$，$3^5=243$

$4^2=16$，$4^3=64$，$4^4=256$，$4^5=1024$

$5^2=25$，$5^3=125$，$5^4=625$，$5^5=3125$

如果把这些幂按照大小排列并去重，我们得到以下由 15 个不同的项组成的序列：

4，8，9，16，25，27，32，64，81，125，243，256，625，1024，3125

在所有满足 $2 \le a \le 100$ 和 $2 \le b \le 100$ 的整数组合生成的幂 a^b 排列并去重所得到的序列中，有多少个不同的项？

解题思路：

利用大整数函数库，写一个 power()函数，Debug 模式下运行 10 秒，Release 模式下不到 1 秒。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut v: Vec<BigUint> = vec![];
    for a in 2..=100 {
        for b in 2..=100 {
            let x = power(a, b);
            if !v.contains(&x) {
                v.push(x);
            }
        }
    }
    println!("{}", v.len());
}

fn power(a: u64, b: u64) -> BigUint {
    let mut prod = BigUint::from(1 as u64);
    for _i in 0..b {
        prod *= BigUint::from(a);
    }
    prod
}
```

有许多重复的 power()运算，可以优化一下：

```
let mut v: Vec<BigUint> = vec![];
for a in 2_u64..=100 {
    let mut prod = BigUint::from(a);
    for _b in 2_u64..=100 {
        prod *= BigUint::from(a);
        if !v.contains(&prod) {
            v.push(prod.clone());
        }
    }
}
println!("{}", v.len());
```

第 48 题 自幂

十项的自幂级数求和为 $1^1 + 2^2 + 3^3 + \ldots + 10^{10} = 10405071317$。

求如下一千项的自幂级数求和的最后 10 位数字：$1^1 + 2^2 + 3^3 + \ldots + 1000^{1000}$。

解题思路：

运用大整数运算函数库，可以暴力解决问题。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut sum: BigUint = BigUint::from(0 as u64);
    for a in 1..=1000 {
        sum += power(a, a);
    }
    let str_sum = sum.to_string();
    println!("{}", &str_sum[str_sum.len()-10..]);
}

fn power(a: u64, b: u64) -> BigUint {
    let mut prod = BigUint::from(1 as u64);
    for _i in 0..b {
        prod *= BigUint::from(a);
    }
    prod
}
```

由于程序只要求最后 10 位数字，可以在每次计算之后，取模，不用任何第三方库也可以完成任务，非常高效。

```
fn main() {
    let mut sum: u64 = 0;
    for a in 1..=1000 {
        sum = (sum + power_last_10(a, a)) % 10_000_000_000;
    }
    println!("{}", sum);
}

fn power_last_10(a: u64, b: u64) -> u64 {
    let mut prod = 1;
    for _i in 0..b {
        prod = prod * a % 10_000_000_000;
    }
    prod
}
```

第 53 题 组合数选择

从五个数 12345 中选择三个恰好有十种方式，分别是：

123、124、125、134、135、145、234、235、245 和 345

在组合数学中，我们记作：$C_5^3 = 10$ 。

一般来说，

$C_n^r = \frac{n!}{r!(n-r)!}$，其中 r ≤ n，n! = n×(n−1)×...×3×2×1，且 0! = 1。

直到 n = 23 时，才出现了超出一百万的组合数：$C_{23}^{10} = 1144066$。

若数值相等形式不同也视为不同，对于 1 ≤ n ≤ 100，有多少个组合数C_n^r超过一百万？

解题步骤：

利用大整数函数库，很容易计算阶乘，再计算组合数。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    // 先把一些阶乘计算好，保存起来
    let mut fact = vec![BigUint::from(1 as u64); 101];
    let mut a = BigUint::from(1 as u64);
    for n in 2..=100 {
        a *= BigUint::from(n as u64);
        fact[n] = a.clone();
    }
    //println!("{:?}", fact);

    let mut count = 0;
    for n in 1..=100 {
        for r in 1..=n {
            let comb = &fact[n] / &fact[r] / &fact[n - r];
            if comb > BigUint::from(1_000_000 as u64) {
                println!("{} {} {}", n, r, comb.to_string());
                count += 1;
            }
        }
    }
    println!("{}", count);
}
```

优化：

再仔细研究一下这些组合数，可以发现一个规律：C(90, 1)，C(90, 2)，C(90, 3)都小于 1 百万，当 C(90, 4)时，值大于 1 百万。

根据组合数的性质，C(90, 86) 一定也肯定大于 1 百万，这样不用进行大量的计算，可以知道 C(90, *)这样的情况中，大于 1 百万的组合有 86 - 4 + 1 = 83 组。

把上面的 `count += 1;` 换成下面两行，可以大幅提升性能：

```
count += n - r - r + 1;
break;
```

第 55 题 利克瑞尔数

将 47 倒序并相加得到 47 + 74 = 121，是一个回文数。

不是所有的数都能像这样迅速地变成回文数。例如，

349 + 943 = 1292

1292 + 2921 = 4213

4213 + 3124 = 7337

也就是说，349 需要迭代三次才能变成回文数。

尽管尚未被证实，但有些数，例如 196，被认为永远不可能变成回文数。如果一个数永远不可能通过倒序并相加变成回文数，就被称为利克瑞尔数。出于理论的限制和问题的要求，在未被证否之前，我们姑且就认为这些数确实是利克瑞尔数。除此之外，已知对于任意一个小于一万的数，它要么在迭代 50 次以内变成回文数，要么就是没有人能够利用现今所有的计算能力将其迭代变成回文数。事实上，10677 是第一个需要超过 50 次迭代变成回文数的数，这个回文数是 4668731596684224866951378664（53 次迭代，28 位数）。

令人惊讶的是，有些回文数本身也是利克瑞尔数数；第一个例子是 4994。

小于一万的数中有多少利克瑞尔数？

注意：2007 年 4 月 24 日，题目略作修改，以强调目前利克瑞尔数理论的限制。

解题思路：

需要用到大整数运算库，前后颠倒相加，再判断是否是回文数，逻辑并不复杂。

```rust
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut count = 0;
    for n in 1..10000 {
        if is_lychrel_number(n) {
            println!("{}", n);
            count += 1;
        }
    }
    println!("count: {}", count);
}

fn is_lychrel_number(n: u64) -> bool {
    let mut x = BigUint::from(n);
    for _i in 0..50 {
        x = lychrel_transform(&x);
        if is_palindromic(&x) {
            return false;
        }
    }
    // 永远变不成回文数，只判断了50次
    true
}

use std::str::FromStr;
// 前后颠倒，求和
fn lychrel_transform(n: &BigUint) -> BigUint {
    let rev_str = n.to_string().chars().rev().collect::<String>();
    let rev_n = BigUint::from_str(&rev_str).unwrap();
    n + rev_n
}

fn is_palindromic(n: &BigUint) -> bool {
    let str_n = n.to_string();
    let rev_str = str_n.chars().rev().collect::<String>();
    str_n == rev_str
}
```

第 56 题 幂的数字和

一古戈尔（10^{100}）是一个巨大的数字：一后面跟着一百个零。100^{100} 则更是无法想像地巨大：一后面跟着两百个零。然而，尽管这两个数如此巨大，各位数字和却都只有 1。

若 a，b < 100，所有能表示为 a^b 的自然数中，最大的各位数字和是多少？

解题步骤：

求各位的数字和，类似的问题出现过许多次。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut max_sum = 0;
    for a in 1..100 {
        for b in 1..100 {
            let s = power(a, b).to_string();
            let sum_digits = s.chars().map(|ch| ch.to_digit(10).unwra
p()).sum::<u32>();
            if sum_digits > max_sum {
                max_sum = sum_digits;
                println!(
                    "{} ^ {} len: {} sum of digits: {}",
                    a,
                    b,
                    s.len(),
                    sum_digits
                );
            }
        }
    }
    println!("{}", max_sum);
}

fn power(a: u64, b: u64) -> BigUint {
    let mut prod = BigUint::from(a as u64);
    for _i in 0..b {
        prod *= BigUint::from(a as u64);
    }
    prod
}
```

第 57 题 平方根逼近

2 的平方根可以用一个无限连分数表示：

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cdots}}}}$$

将连分数计算取前四次迭代展开式分别是：

$$1+\frac{1}{2}=\frac{3}{2}=1.5$$
$$1+\frac{1}{2+\frac{1}{2}}=\frac{7}{5}=1.4$$
$$1+\frac{1}{2+\frac{1}{2+\frac{1}{2}}}=\frac{17}{12}=1.41666\ldots$$
$$1+\frac{1}{2+\frac{1}{2+\frac{1}{2+\frac{1}{2}}}}=\frac{41}{29}=1.41379\ldots$$

接下来的三个迭代展开式分别是 99/70、239/169 和 577/408，但是直到第八个迭代展开式 1393/985，分子的位数第一次超过分母的位数。

在前一千个迭代展开式中，有多少个分数分子的位数多于分母的位数？

解题步骤：

根据第 i 项，很容易推出第 i+1 项。

$$\frac{a_{i+1}}{b_{i+1}}=1+\frac{1}{1+\frac{a_i}{b_i}}$$

数字的位数很多，需要用到大整数计算。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut count = 0;
    let mut a = BigUint::from(3 as u64);
    let mut b = BigUint::from(2 as u64);
    for _i in 2..=1000 {
        let c = &a + &b;
        a = &c + &b;
        b = c;
        if a.to_string().len() > b.to_string().len() {
            count += 1;
            //println!("{} {}", a, b);
        }
    }
    println!("{}", count);
}
```

第 63 题 幂次与位数

五位数 16807=7^5 同时也是一个五次幂。同样的，九位数 134217728=8^9 同时也

是九次幂。

有多少个 n 位正整数同时也是 n 次幂？

解题步骤：

底数 a 肯定不能大于 9，因为 10 的 2 次幂是 100，已经超过 2 位数。

幂数要考虑退出循环的条件，比如：

当 a=4，b=3 时，$4^3 = 64$，只是 2 位数，当幂次增加时，它的位数永远不可能超过幂次，此时不需要再尝试更多的 b，退出内层循环即可。

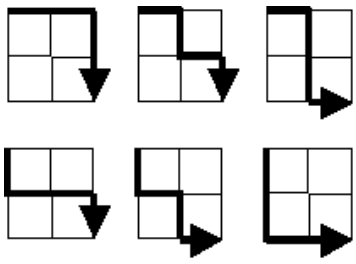
```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut count = 0;
    for a in 1..=9 {
        for b in 1.. {
            let p = power(a, b).to_string();
            if p.len() < b as usize {
                // 位数永远不可能超过幂次了，退出内层循环
                break;
            }
            if p.len() == b as usize {
                count += 1;
                println!("{}:  {} ^ {} = {}", count, a, b, p);
            }
        }
    }
    println!("{}", count);
}

fn power(a: u64, b: u64) -> BigUint {
    let mut p = BigUint::from(a);
    for _i in 1..b {
        p *= BigUint::from(a);
    }
    p
}
```

9 路径

第 15 题 网格路径

已知 2x2 网格中从左上角到右下角共有 6 条可能路径，计算 20x20 网格中，

有多少条可能的路径。

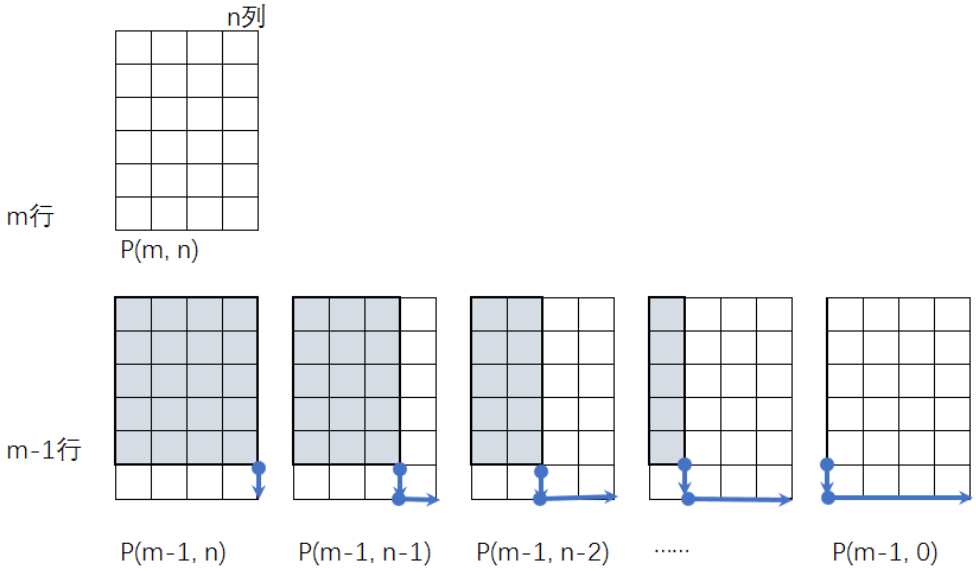


解题思路：

还是用递归的思路。对于 m 行 n 列的网格，可以利用其它网格的路径个数的结果，即：

$$P(m,n) = P(m-1,n) + P(m-1,n-1) + \ldots + P(m-1,1) + P(m-1,0)$$

对于 0 行或者 0 列的网格，路径只有 1 条。



程序就比较容易写出来了：

```rust
fn path_slow(m: usize, n: usize) -> u64 {
    if m == 0 || n == 0 { return 1; }
    let mut sum = 0;
    for j in 0..=n {
        sum += path_slow(m-1, j);
    }
    return sum;
}
```

```rust
fn main() {
    println!("{}", path_slow(12, 12));
    println!("{}", path_slow(20, 20));
}
```

可惜程序的性能很差，对于 12x12 的网格可以秒出，而 20x20 的网格估计 20 分钟也没反应，看来重复的运算量太大了。

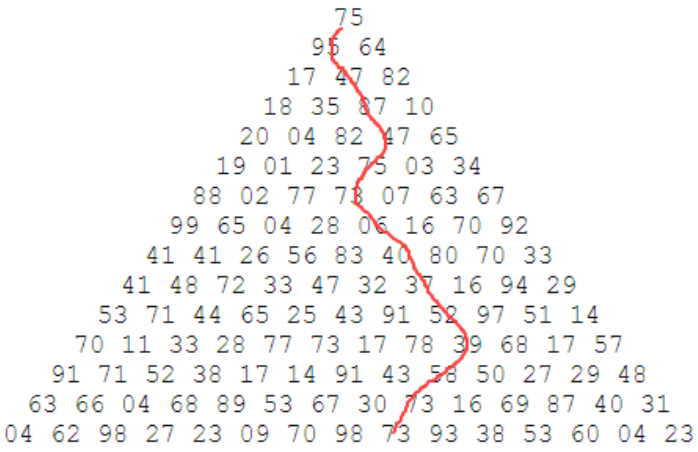
可以把以前计算的结果缓存到一个一维向量中，速度则大幅提升，这里可以学到&mut 传入向量地址的语法知识点，另外初始化 10000 万个零，用 vec![0; 10000]。

```rust
fn main() {
    let mut v: Vec<u64> = vec![0; 10000];
    println!("{}", path_fast(&mut v, 20, 20));
}

fn path_fast(v: &mut Vec<u64>, m: usize, n: usize) -> u64 {
    if m == 0 || n == 0 {
        return 1;
    }
    if v[m * 100 + n] > 0 {
        return v[m * 100 + n];
    } //缓存命中
    let mut sum = 0;
    for j in 0..=n {
        sum += path_fast(v, m - 1, j);
    }
    v[m * 100 + n] = sum; // 加入缓存中
    println!("({},{}) {}", m, n, sum);
    return sum;
}
```

另外，这道题可以推导出一个排列组合的数学公式，当然就体会不到编程的乐趣了。

第 18 题 最大路径和 I

从堆成三角的数字中，找到一条路径，使其和最大，求和。一个节点的下一个点只能是下一层的左节点或右节点。



为了节省内存空间，用一维数组表示这些数，需要准确地计算出各个索引位置的行号，为了方便地计算出左、右子节点，最上一层的行号为 1。

```
let w = [
                                75, // row 1
                              95, 64,  // row 2
                            17, 47, 82,  // row 3
                          18, 35, 87, 10,
                        20, 04, 82, 47, 65,
                      19, 01, 23, 75, 03, 34,
                    88, 02, 77, 73, 07, 63, 67,
                  99, 65, 04, 28, 06, 16, 70, 92,
                41, 41, 26, 56, 83, 40, 80, 70, 33,
              41, 48, 72, 33, 47, 32, 37, 16, 94, 29,
            53, 71, 44, 65, 25, 43, 91, 52, 97, 51, 14,
          70, 11, 33, 28, 77, 73, 17, 78, 39, 68, 17, 57,
        91, 71, 52, 38, 17, 14, 91, 43, 58, 50, 27, 29, 48,
      63, 66, 04, 68, 89, 53, 67, 30, 73, 16, 69, 87, 40, 31,
    04, 62, 98, 27, 23, 09, 70, 98, 73, 93, 38, 53, 60, 04, 23,
];
```

在数组中的第 n 个数，行号是多少？利用了第 r 行一定有 r 个数的性质。

```
fn row(n: usize) -> usize {
    let mut s = 0;
    for r in 1.. {
        s += r;
        if s > n {return r;}
    }
    return 0;
}
```

根据上面行号的性质，可以得出节点 n 的下一层的左节点的编号是 n + r，右节点是 n + r + 1。

求路径的和的时候可以利用递归，终止条件是遇到最底一层的时候，由于原题只让求路径长度，这里没有记下来所走路径的编号。

```
fn max_path_weight(w: &[u32], n: usize) -> u32 {
    if n >= w.len() {return 0;} //越界判断
    let r = row(n);
    let bottom_row = row(w.len() - 1);
    if r == bottom_row { // 递归的退出条件
        return w[n];
    }
    let left = max_path_weight(w, n + r);
    let right = max_path_weight(w, n + r + 1);
    let max = if left > right {left} else {right};
    return w[n] + max;
}
```

主程序调用只需要一行，数组的总层数不多，复杂度不高，没再做进一步的性能优化。

```
println!("{}", max_path_weight(&w, 0));
```

第 67 题 最大路径和 II

从下面展示的三角形的顶端出发，不断移动到在下一行与其相邻的元素，能够得到的最大路径和是 23。

3
7 4
2 **4** 6
8 5 **9** 3

如上图，最大路径和为 3 + 7 + 4 + 9 = 23。

在这个 15K 的文本文件 triangle.txt（右击并选择“目标另存为……”）中包含了一个一百行的三角形，求从其顶端出发到达底部，所能够得到的最大路径和。

注意：这是第 18 题的强化版。由于总路径一共有 2^{99} 条，穷举每条路经来解决这个问题是不可能的！即使你每秒钟能够检查一万亿（10^{12}）条路径，全部检查完也需要两千万年。存在一个非常高效的算法能解决这个问题。

解题步骤：

第 18 题的算法用递归实现，数据量小，没有问题，在这道题中得更换算法。

如果知道一个节点的左、右节点的最大路径，可以很容易地计算出当前节点的最大路径，从底层开始，逐层计算每个节点到底部节点的最大路径上一层的最大路径，所以从每一层中最大路径只与下一层的左、右节点有关。

1）读文件，保存到数组中

这里采用连续存放的策略，节省内存空间。UNIX 和 Windows 中的换行符有一点点区别，replace()时要注意。

```
let data = std::fs::read_to_string("triangle.txt").expect("读文件失败
");
let data2 = data.trim().replace("\r\n", " ").replace("\n", " ");
let w: Vec<usize> = data2
    .split(" ")
    .map(|x| x.parse::<usize>().unwrap())
    .collect();
```

2）计算某一个节点的行号

强行规定顶层的行号为 1，逐层增 1，这样规定有一个好处，就是每层的行号就是每层元素的个数。

```
fn row(n: usize) -> usize {
    let mut s = 0;
    for r in 1.. {
        s += r;
        if s > n {
            return r;
        }
    }
    return 0;
}
```

3）自下而上逐层计算

```
fn compute_path_weight(w: &Vec<usize>) -> Vec<usize> {
    let mut path: Vec<usize> = vec![0; w.len()];
    let max_row: usize = row(w.len() - 1);
    for i in (0..w.len()).rev() { // 从底层向上计算
        let r = row(i);
        if r == max_row { // leaf
            path[i] = w[i];
        } else {
            let left = w[i] + path[i + r];
            let right = w[i] + path[i + r + 1];
            path[i] = if left > right { left } else { right };
        }
    }
    return path;
}
```

4）第 0 个节点的值就是答案

```
let path = compute_path_weight(&w);
println!("{}", path[0]);
```

10 日期

第 19 题 数星期日

求 20 世纪（1901 年 1 月 1 日到 2000 年 12 月 31 日）有多少个月的一号是星期日？

本题当然可以利用闰年的性质，只用数学公式就能算出来，这里用编程办法，熟悉一下 Rust 中如何处理日期和时间。

关于日期的库用 chrono，网上有些资料比较老，建议直接参考官网上的帮助，写得非常详细，少走一些弯路。

在 https://docs.rs 网站上搜索 chrono 即可。

```
use chrono::prelude::*;
use time::Duration;
```

代码简单粗暴，每次加一天，用到 Duration；判断星期几用到 weekday()，判断几号用了 day()，逻辑很简单。

```
let mut count = 0;
let mut d = Utc.ymd(1901, 1, 1);
while d <= Utc.ymd(2000, 12, 31) {
    if d.weekday() == Weekday::Sun && d.day() == 1 {
        println!("{}", d);
        count += 1;
    }
    d = d + Duration::days(1);
}
println!("{}", count);
```

11 排列组合

第 24 题 字典序排列

0，1，2，3，4，5，6，7，8 和 9，每个数字用且只用一次，称为全排列，按

数值大小排序，求第一百万个数是多少？

例如，0，1 和 2 按从小到大只有 6 种排列：012，021，102，120，201，210。

解题思路：

这是一道排列组合类的数学题，在百度文库中有一个 PPT 介绍得不错，链接：https://wk.baidu.com/view/5f4bacf79e31433239689339?pcf=2&fromShare=1&fr=copy&copyfr=copylinkpop

这道题可以利用其中的字典序的算法：

实现这个算法不太麻烦，只是需要细心一些。

```
fn next_perm(v: &mut Vec<u32>) {
    let mut i = v.len() - 2;
    while v[i] > v[i + 1] {
        i -= 1;
    }
    let mut j = v.len() - 1;
    while i < j && v[i] > v[j] {
        j -= 1;
    }
    v.swap(i, j);
    i += 1;
    j = v.len() - 1;
    while i < j {
        v.swap(i, j);
        i += 1;
        j -= 1;
    }
}
```

主程序只需要循环就行了，向量的初始值就是第一个排列，从 2 开始找到第 1

00 万个排列数。

```
fn main() {
    let mut v: Vec<u32> = vec![0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    for i in 2..=1_000_000 {
        next_perm(&mut v);
        //println!("{} {:?}", i, v);
    }
    println!("{:?}", v);
}
```

这里的 v 是向量表示，要转换成一个整数，可以这样：

```
let v_str = v.iter()
             .map(|x| x.to_string())
             .collect::<String>();
println!("{}", v_str.parse::<u64>().unwrap());
```

这里的组合数一直是 10 个数字，也可以换成定长数组的写法，

```
fn main() {
    let mut v = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    for i in 2..=1_000_000 {
        next_perm(&mut v);
    }
    println!("{:?}", v);
}
```

语法知识点：

✧ 注意向量或数组传递到函数里的写法

第 31 题 硬币求和

英国的货币单位包括英镑£和便士 p，在流通中的硬币一共有八种：

1p，2p，5p，10p，20p，50p，£1（100p），£2（200p）

以下是组成£2 的其中一种可行方式：

1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p

不限定使用的硬币数目，组成£2 有多少种不同的方式？

解题步骤：

采取了一种丑陋无比的 8 层循环的写法，暴力解决了问题。

```
fn main() {
    let mut count = 0;
    for a in 0..=200 {
        for b in 0..=100 {
            for c in 0..=40 {
                for d in 0..=20 {
                    for e in 0..=10 {
                        for f in 0..=4 {
                            for g in 0..=2 {
                                for h in 0..=1 {
                                    let price = a
                                        + b * 2
                                        + c * 5
                                        + d * 10
                                        + e * 20
                                        + f * 50
                                        + g * 100
                                        + h * 200;
                                    if price == 200 {
                                        count += 1;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    println!("{}", count);
}
```

用递归算法显得更简洁和优美一些。

```
fn main() {
    println!("{}", ways(200, 0));
}

fn ways(money: isize, maxcoin: usize) -> usize {
    let coins = [200, 100, 50, 20, 10, 5, 2, 1];
    let mut sum = 0;
    if maxcoin == 7 {
        return 1;
    }
    for i in maxcoin..8 {
        if money - coins[i] == 0 {
            sum += 1;
        }
        if money - coins[i] > 0 {
            sum += ways(money - coins[i], i);
        }
    }
    sum
}
```

第 41 题 全数字的素数

如果一个 n 位数恰好使用了 1 至 n 每个数字各一次,我们就称其为全数字的。例如，2143 就是一个 4 位全数字数，同时它恰好也是一个素数。

最大的全数字的素数是多少？

解题步骤：

1）生成全排列

第 24 题中已经有一个全排列的生成算法，增加一个返回值，如果已经到达了最后的一个排列，就返回 false，方便主程序退出循环。

```
fn next_perm(v: &mut [u64]) -> bool {
    let mut i = v.len() - 2;
    while v[i] > v[i + 1] {
        if i == 0 {
            return false;
        }
        i -= 1;
    }
    let mut j = v.len() - 1;
    while i < j && v[i] > v[j] {
        j -= 1;    }
    swap(v, i, j);
    i += 1;
    j = v.len() - 1;
    while i < j {
        swap(v, i, j);
        i += 1;
        j -= 1;
    }
    true
}
```

2）向量转换成数值

为了后面判断素数，需要将[1, 2, 3, 4, 5, 6, 7]这样的向量转换成 1234567。通常的做法是把每个数字转换成字符串，拼在一起，再转换成数值。

```
let v_str = v.iter().map(|x| x.to_string()).collect::<String>();
let d = v_str.parse::<usize>().unwrap();
```

后来发现，用一系列整数运算可以完成这个任务，代码比较简洁，但我没有比

较两种算法的效率，初步估计整数运算的效率会更高一些。

```
let d = v.iter().fold(0, |x, a| 10 * x + a);
```

3）主程序

准备工作完成后，主程序没有难度。我先用 4 位整数验证了程序的正确性，再跑 9 位、8 位的情况，最后在 7 位的时候发现了答案。

```
let mut v = [1, 2, 3, 4, 5, 6, 7];
let mut max_prime = 0;
while next_perm(&mut v) {
    let d = v.iter().fold(0, |x, a| 10 * x + a);
    if primes::is_prime(d as u64) && d > max_prime {
        println!("{}", d);
        max_prime = d;
    }
}
```

4）不重新发明轮子

我以前为了练习算法，自己写了全排列的生成函数，实际上别人已经写好了这类函数库，直接拿来用就行。

```
use permutohedron::heap_recursive;
fn main() {
    let mut max_prime = 0;
    let mut data = [1, 2, 3, 4, 5, 6, 7];
    heap_recursive(&mut data, |permutation| {
        let v = permutation.to_vec();
        let d = v.iter().fold(0, |x, a| 10 * x + a);
        if primes::is_prime(d as u64) && d > max_prime {
            println!("{}", d);
            max_prime = d;
        }
    })
}
```

5）更为高效的算法

因为题目只要求找出最大的全排列素数，前面这些算法都要把所有的排列组合都尝试一遍，效率极低。

最高效的办法是修改最早的全排列生成算法，让 next_perm_desc()降序生成，这样找到的第一个素数就是最终答案。

```
fn main() {
    let mut v = [7, 6, 5, 4, 3, 2, 1];
    loop {
        let d = v.iter().fold(0, |x, a| 10 * x + a);
```

```
        if primes::is_prime(d as u64) {
            println!("{}", d);
            break;
        }
        if !next_perm_desc(&mut v) {
            break;
        }
    }
}

// 降序全排列
fn next_perm_desc(v: &mut [u64]) -> bool {
    let mut i = v.len() - 2;
    while v[i] < v[i + 1] {
        if i == 0 {
            return false;
        }
        i -= 1;
    }

    let mut j = v.len() - 1;
    while i < j && v[i] < v[j] {
        j -= 1;
    }

    swap(v, i, j);

    i += 1;
    j = v.len() - 1;
    while i < j {
        swap(v, i, j);
        i += 1;
        j -= 1;
    }
    true
}

fn swap(v: &mut [u64], i: usize, j: usize) {
    let temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

第 49 题 素数重排

公差为 3330 的三项等差序列 1487、4817、8147 在两个方面非常特别：其一，每一项都是素数；其二，两两都是重新排列的关系。

一位素数、两位素数和三位素数都无法构成满足这些性质的数列，但存在另一个由四位素数构成的递增序列也满足这些性质。

将这个数列的三项连接起来得到的 12 位数是多少？

问题分解：

1）将一个四位数生成全排列，过滤到非素数，保存在集合中

2）从集合中挑出一个数，如果能够按等差数列关系，在集合里找到第三个数，则找到一组答案

```
use permutohedron::heap_recursive;

fn main() {
    // 四位数
    for x in 1000u64..=9999 {
        if !primes::is_prime(x) {
            continue;
        }
        // 拆成4个数字
        let mut vx: Vec<u64> = x
            .to_string()
            .chars()
            .map(|c| c.to_digit(10).unwrap() as u64)
            .collect();

        // 全排列，保存在集合中
        let mut vy = Vec::new();
        heap_recursive(&mut vx, |pt| {
            let y = pt.to_vec().iter().fold(0, |x, a| 10 * x + a);
            if y > x && primes::is_prime(y) && !vy.contains(&y) {
                vy.push(y);
            }
        });
        //println!("{} {:?}", x, vy);

        for &y in vy.iter() {
            // 按等差关系形成第三个数
            let z = (y - x) + y;
            if vy.contains(&z) && primes::is_prime(z) {
                println!("{}{}{}", x, y, z);
            }
        }
    }
}
```

第 43 题 子串的可整除性

1406357289 是一个 0 至 9 全数字数，因为它由 0 到 9 这十个数字排列而成；但除此之外，它还有一个有趣的性质：子串的可整除性。

记 d_1 是它的第一个数字，d_2 是第二个数字，依此类推，我们注意到：

- ✧ $d_2d_3d_4$=406 能被 2 整除
- ✧ $d_3d_4d_5$=063 能被 3 整除
- ✧ $d_4d_5d_6$=635 能被 5 整除
- ✧ $d_5d_6d_7$=357 能被 7 整除
- ✧ $d_6d_7d_8$=572 能被 11 整除
- ✧ $d_7d_8d_9$=728 能被 13 整除
- ✧ $d_8d_9d_{10}$=289 能被 17 整除

找出所有满足同样性质的 0 至 9 全数字数，并求它们的和。

问题分解：

1）找出 0 到 9 的所有全排列

2）找出三位数的子串

3）暴力循环求解

第一步，找全排列

在第 24 题和第 41 题已经了解了全排列的算法，这里直接利用 next_perm()函数即可，需要注意 0 不能出现在最左边。

第二步：取出 3 位数字

这里以取 $d_2d_3d_4$ 三位数字为例：

```
let v_str = v.iter().map(|x| x.to_string()).collect::<String>();
let sub3 = &v_str[1..4];
let d = sub3.parse::<u32>().unwrap();
```

第三步，可以写出主程序：

```
let mut v = [1, 0, 2, 3, 4, 5, 6, 7, 8, 9];
let primes = [2, 3, 5, 7, 11, 13, 17];
let mut sum: u64 = 0;
```

```
loop {
    let v_str = v.iter().map(|x| x.to_string()).collect::<String>();

    for i in 1..=7 {
        let sub3 = &v_str[i..i + 3];
        let d = sub3.parse::<u32>().unwrap();
        if d % primes[i-1] != 0 {
            break;
        }
        if i == 7 {
            println!("{}", v_str);
            sum += v_str.parse::<u64>().unwrap();
        }
    }

    if !next_perm(&mut v) {
        break;
    }
}
println!("sum: {}", sum);
```

第四步：优化

前面的算法中大量在字符串和数字之间进行转换，效率还有点低，实际上可以全部利用整数的运算，效率可以提高很多，最后的代码：

```
fn main() {
    let mut v = [0, 9, 8, 7, 6, 5, 4, 3, 2, 1];
    let mut sum: u64 = 0;
    while next_perm(&mut v) {
        let num = v.iter().fold(0, |x, a| 10 * x + a);
        if is_divisibility(num) {
            println!("{}", num);
            sum += num;
        }
    }
    println!("sum: {}", sum);
}

fn is_divisibility(num: u64) -> bool {
    let primes = [2, 3, 5, 7, 11, 13, 17];
    let mut n = num % 1_000_000_000;
    for i in (0..=6).rev() {
        let sub3 = n % 1000;
        if sub3 % primes[i] != 0 {
            return false;
        }
        n = n / 10;
    }
    true
}
```

12 分数

第 26 题 倒数的循环节

单位分数指分子为 1 的分数。

1/6= 0.1(6) 表示 0.166666…，括号内表示有一位循环节。

1/7= 0.(142857)，1/7 有六位循环节。

找出正整数 d < 1000，其倒数的十进制表示小数部分有最长的循环节。

解题思路：

通过手算寻找规律，当分母为 7 时：

n = 7

	余数		商
	1		0.
1 * 10 % 7 =	3	1 * 10 / 7 =	1
3 * 10 % 7 =	2	3 * 10 / 7 =	4
2 * 10 % 7 =	6	2 * 10 / 7 =	2
6 * 10 % 7 =	4	6 * 10 / 7 =	8
4 * 10 % 7 =	5	4 * 10 / 7 =	5
5 * 10 % 7 =	1	5 * 10 / 7 =	7
1 * 10 % 7 =	3	1 * 10 / 7 =	1
3 * 10 % 7 =	2	3 * 10 / 7 =	4
2 * 10 % 7 =	6	2 * 10 / 7 =	2
6 * 10 % 7 =	4	6 * 10 / 7 =	8
4 * 10 % 7 =	5	4 * 10 / 7 =	5
5 * 10 % 7 =	1	5 * 10 / 7 =	7

再找一个分母大于 10 的情况：

n = 13

	余数
1 * 10 % 13 =	10
10 * 10 % 13 =	9
9 * 10 % 13 =	12
12 * 10 % 13 =	3
3 * 10 % 13 =	4
4 * 10 % 13 =	1
1 * 10 % 13 =	10
10 * 10 % 13 =	9
9 * 10 % 13 =	12
12 * 10 % 13 =	3
3 * 10 % 13 =	4
4 * 10 % 13 =	1
1 * 10 % 13 =	10
10 * 10 % 13 =	9

0.(076923)

	商
1 * 10 / 13 =	0
10 * 10 / 13 =	7
9 * 10 / 13 =	6
12 * 10 / 13 =	9
3 * 10 / 13 =	2
4 * 10 / 13 =	3
1 * 10 / 13 =	0
10 * 10 / 13 =	7
9 * 10 / 13 =	6
12 * 10 / 13 =	9
3 * 10 / 13 =	2
4 * 10 / 13 =	3
1 * 10 / 13 =	0
10 * 10 / 13 =	7

再找一个能除尽的：

n = 16

	余数
1 * 10 % 16 =	10
10 * 10 % 16 =	4
4 * 10 % 16 =	8
8 * 10 % 16 =	0
0 * 10 % 16 =	0
0 * 10 % 16 =	0
0 * 10 % 16 =	0
0 * 10 % 16 =	0
0 * 10 % 16 =	0

	商
1 * 10 / 16 =	0
10 * 10 / 16 =	6
4 * 10 / 16 =	2
8 * 10 / 16 =	5
0 * 10 / 16 =	0
0 * 10 / 16 =	0
0 * 10 / 16 =	0
0 * 10 / 16 =	0
0 * 10 / 16 =	0

可以发现几个规律：

1）分子为 1，表示一开始的余数为 1

2）余数为 0 时，表示可以除尽，循环要终止

3）当余数重复出现时，表示找到了循环节，两次重复出现的位置就是循环节

按照这个逻辑，循环节的长度可以求出，这里用两个向量分别存储余数 remainders 和商 digits。

```
fn reciprocal_cycle(d: u32) -> u32 {
```

```
    let mut remainders : Vec<u32> = vec![1]; //余数
    let mut digits : Vec<u32> = vec![]; //商

    let mut numerator = 1; //分子
    while numerator != 0 {
        digits.push(numerator * 10 / d);
        numerator = numerator * 10 % d; //余数
        let pos = remainders.iter().position(|&x| x==numerator);
        match pos {
            Some(x) => { //余数重复出现时
                return (digits.len() - x) as u32;
            }
            None => {
                remainders.push(numerator);
            }
        }
    }
    0 //除尽的时候，表示循环节为0
}
```

这里在向量里查找一个元素的位置索引时用了 position 函数，返回是一个 Option<T>类型，用 match 语句针对不同的情况进行处理。

主程序就简单了：

```
let mut max_cycle: u32 = 0;
for n in 2..1000 {
    let rc = reciprocal_cycle(n);
    if rc > max_cycle {
        println!("n={} cycle={}", n, rc);
        max_cycle = rc;
    }
}
println!("max reciprocal cycle: {}", max_cycle);
```

优化：实际上商并不需要存储，可以减少一个向量，求循环节的函数还可以精简一下。

```
fn reciprocal_cycle(d: u32) -> u32 {
    let mut remainders : Vec<u32> = vec![1]; //余数
    let mut numerator = 1; //分子
    while { numerator = numerator * 10 % d; numerator != 0} {
        let pos = remainders.iter().position(|&x| x==numerator);
        match pos {
            Some(x) => { //余数重复出现时
                return (remainders.len() - x) as u32;
            }
            None => {
                remainders.push(numerator);
```

```
            }
        }
    }
    0 //除尽的时候，表示循环节为0
}
```

第 33 题 消去数字的分数

49/98 是一个有趣的分数，因为缺乏经验的数学家可能在约简时错误地认为，等式 49/98 = 4/8 之所以成立，是因为在分数线上下同时抹除了 9 的缘故。

我们也会想到，存在诸如 30/50 = 3/5 这样的平凡解。

这类有趣的分数恰好有四个非平凡的例子，它们的分数值小于 1，且分子和分母都是两位数。

将这四个分数的乘积写成最简分数，求此时分母的值。

求解思路：

四个分数很容易求出，我这里没有进行通分运算，后面手算即可。

```
for ab in 10..=99 {
    let a = ab / 10;
    let b = ab % 10;
    for cd in (ab+1)..=99 {
        let c = cd / 10;
        let d = cd % 10;
        if b == c && ab * d == cd * a {
            println!("{} / {} = {} / {}", ab, cd, a, d);
        }
    }
}
```

13 三角形数

第 39 题 直角三角形

周长 p 的 120 的整数直角三角形共有 3 个：{20, 48, 52}， {24, 45, 51}， {30, 40, 50}，如果周长 p<=1000，当 p 取何整数值时，满足条件的直角三角数个数最多？

解题思路

先对给定的周长，把所有的直角三角形求出来。为了不输出重复项，第二重循环的取值范围要注意。

```
fn count_right_triangles(p: isize) -> isize {
    let mut count = 0;
    for a in 1..p {
        for b in a..p {
            let c = p - a - b;
            if c > 0 && a * a + b * b == c * c {
                count += 1;
                print!("{{{}, {}, {}}}, ", a, b, c);
            }
        }
    }
    count
}
```

主程序比较简单。

```
let mut max_count = 1;
let mut max_p = 0;
for p in 2..1000 {
    let count = count_right_triangles(p);
    if count > max_count {
        max_count = count;
        max_p = p;
        println!("p: {} max: {}", p, max_count);
    }
}
println!("p: {}", max_p);
```

第 42 题 编码三角形数

三角形数序列的第 n 项由公式 $t_n = n(n+1)/2$ 给出；因此前十个三角形数是：

1，3，6，10，15，21，28，36，45，55，…

将一个单词的每个字母分别转化为其在字母表中的顺序并相加，我们可以计算出一个单词的值。例如，单词 SKY 的值就是 19 + 11 + 25 = 55 = t_{10}。如果一个单词的值是一个三角形数，我们就称这个单词为三角形单词。

在这个 16K 的文本文件 words.txt （右击并选择“目标另存为……”）中包含有将近两千个常用英文单词，这其中有多少个三角形单词？

解题思路：

1）读文件内容到数组中

```
let data = std::fs::read_to_string("words.txt").expect("读文件失败");
// 删除引号
let data2: String = data.chars().filter(|c| *c != '"').collect();
let names: Vec<&str> = data2.split(",").collect();
```

2）准备足够的三角数，以备将来进行快速判断

```
let mut tri_numbers = vec![];for i in 1..=100 {
    tri_numbers.push(i * (i + 1) / 2);
}
//println!("{:?}", tri_numbers);
```

3）字符在字母表中的顺序号

最早是用查找方式实现的。

```
let letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";letters.chars().position
(|c| c == ch).unwrap() + 1
```

在 Rust 中一个字符可以直接转换成 u8 类型，就是其 ASCII 编码值，'A'的编码为 65，字符与'A'的差值就是编号，更加高效。

```
// 一个字符在字母表中分数，'A' -> 1，'B' -> 2
fn letter_number(ch: char) -> u8 {
    (ch as u8) - 64
}
```

4）单词的得分

常规的写法：

```
fn word_score(word: &str) -> usize {
    let mut score = 0;
    for ch in word.chars() {
        score += letter_number(ch) as usize;
    }
    score
}
```

可以用函数式编程的写法：

```
fn word_score(word: &str) -> usize {
    word.chars().map(|ch| letter_number(ch) as usize).sum()
}
```

5）主循环算分求和即可。

前面的函数都比较简单，可以写在一行，最后的主程序也可以很精炼。

```
let mut count = 0;
for name in names {
    let word_score = name.chars().map(|ch| ch as usize - 64).sum();
    if tri_numbers.contains(&word_score) {
        //println!("{} {}", name, word_score);
        count += 1;
    }
}
println!("{}", count);
```

第 44 题 五边形数

五边形数由公式 $P_n=n(3n-1)/2$ 生成。前十个五边形数是：

$$1,\ 5,\ 12,\ 22,\ 35,\ 51,\ 70,\ 92,\ 117,\ 145,\ \cdots$$

可以看出 $P_4 + P_7 = 22 + 70 = 92 = P_8$。然而，它们的差 $70 - 22 = 48$ 并不是五边形数。

在所有和差均为五边形数的五边形数对 P_j 和 P_k 中，找出使 $D = |P_k - P_j|$ 最小的一对；此时 D 的值是多少？

问题分解：

1）生成足够的五边形数，备用

一开始不知道最终答案的范围，先生成 1 万个备用。

```
let mut penta: Vec<u64> = vec![0];
for i in 1..10000 {
    penta.push(i * (3 * i - 1) / 2);
}
```

2）双重循环搜索

暴力搜索，判断和与差是否也是五边形数，竟然只找到一个满足条件的解，输入到 projecteuler 网站，发现竟然就是正确答案。

```
for k in 2..3000 {
    for j in (1..k).rev() {
        let d = penta[k] - penta[j];
        let sum = penta[k] + penta[j];
        if penta.contains(&d) && penta.contains(&sum) {
            println!("j:{} k:{} pj:{} pk:{} diff:{}",
```

```
                    j, k, penta[j], penta[k], d);
            }
        }
    }
```

3）优化

判断一个数是不是五边形数用 contains()效率并不高，特别是集合的元素非常多时，得把高中的二次函数的求解公式翻出来。

$$\frac{x\cdot(3x-1)}{2}=p \qquad \frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

$$x(3x-1)=2p$$

$$3x^2-x-2p=0$$

$$x=\frac{1\pm\sqrt{1^2-4\times3\times(-2p)}}{2\times3}=\frac{1\pm\sqrt{1+24p}}{6}$$

舍去负数的解，得：

$$x=\frac{1+\sqrt{1+24p}}{6}$$

如果 p 是五边形数，1+24*p 应该是完全平方数，分子还要能被 6 整除。

```
fn is_penta(p: u64) -> bool {
    let t = ((1 + 24 * p) as f64).sqrt() as u64;
    if t * t != (1 + 24 * p) {
        return false;
    }
    (t + 1) % 6 == 0
}
```

主程序仍用双重循环搜索。

```
let mut min_d = std::u64::MAX; // 改为2.. 可以证明结果的正确性，但要运行相当长的时间
for k in 2..10000 {
     let pk = penta(k);
    if pk - penta(k-1) > min_d {break;}
    for j in (1..k).rev() {
        let pj = penta(j);
        let d = pk - pj;
        if d < min_d && is_penta(d) && is_penta(pk + pj) {
            println!("j:{} k:{} pj:{} pk:{} diff:{}",
                     j, k, pj, pk, d);
            min_d = d;
            break;
        }
    }
}
```

这个题找到一个答案的速度非常快，但并不能充分地证明它就是最小的 d，应该再继续向后搜索五边形数，当相邻的五边形数的差都大于 min_d 时，才证明了的确找到了最小的 d，但需要运行相当长的时间。

第 45 题 三角形数、五边形数和六角形数

三角形数、五边形数和六角形数分别由以下公式给出：

三角形数	$T_n=n(n+1)/2$	1，3，6，10，15，…
五边形数	$P_n=n(3n-1)/2$	1，5，12，22，35，…
六边形数	$H_n=n(2n-1)$	1，6，15，28，45，…

可以验证，$T_{285} = P_{165} = H_{143} = 40755$。

找出下一个同时是三角形数、五边形数和六角形数的数。

问题求解：

这个题我没有采用第 44 题的二次函数求解的公式，准备好 10 万个三角数和五边形数，暴力搜索找到了答案。

```
let mut tri: Vec<u64> = vec![];
for i in 1..100000 {
    tri.push(i * (i + 1) / 2);
}
let mut penta: Vec<u64> = vec![];
for i in 1..100000 {
    penta.push(i * (3 * i - 1) / 2);
}

for i in 2..30000 {
    let hex = i * (2 * i - 1);
    if tri.contains(&hex) && penta.contains(&hex) {
        println!("i: {}  hexagonal: {}", i, hex);
    }
}
```

14 密码学

第 59 题 异或解密

计算机上的每个字符都被指定了一个独特的代码，其中被广泛使用的一种是 ASCII 码（美国信息交换标准代码）。例如，大写字母 A = 65，星号（*） = 42，小写字母 k = 107。

一种现代加密方法是将一个文本文档中的符号先转化为 ASCII 码，然后将每个字节异或一个根据密钥确定的值。使用异或进行加密的好处在于，只需对密文使用相同的密钥再加密一次就能得到明文，例如，65 XOR 42 = 107，而 107 XOR 42 = 65。

为了使加密难以破解，密钥要和明文一样长，由随机的字节构成。用户会把加密过的信息和密钥放置在不同的地方，解密时二者缺一不可。

不幸的是，这种方法对大多数人来说并不实用，因此一个略有改进的方法是使用一个密码作为密钥。密码的长度很有可能比信息要短，这时候就循环重复使用这个密码进行加密。这种方法需要达到一种平衡，一方面密码要足够长才能保证安全，另一方面需要充分短以方便记忆。

你的破解任务要简单得多，因为密钥只由三个小写字母构成。文本文档 cipher.txt（右击并选择“目标另存为……”）中包含了加密后的 ASCII 码，并且已知明文包含的一定是常见的英文单词，解密这条消息并求出原文的 ASCII 码之和。

解题步骤：

1）读文件，保存在数组中

cipher.txt 文件中是 ASCII 码数值，转换成 u8 类型存储。

```
let data = std::fs::read_to_string("cipher.txt").expect("文件无法打开
");
let xs: Vec<&str> = data.split(",").collect();
let letters: Vec<u8> = xs.iter().map(|x| x.parse::<u8>().unwrap()).c
ollect();
```

2）统计

解密的文本是常见的英文单词，而且密钥是小写字母，只需用这 26 个小写字

母分别与这些文本进行 XOR，统计分别得到的英文单词的个数，哪个最多哪个就最可能是正确的密码。

3 个小写字母密钥针对不同的位置进行 XOR 操作，index 取 0，1，2，表示位置索引。

```
fn guess_pass(letters: &Vec<u8>, index: usize) -> char {
    let mut key = '*';
    let mut max_count = 0;
    for pass in ('a' as u8)..=('z' as u8) {
        let mut count = 0;
        for (i, ch) in letters.iter().enumerate() {
            if i % 3 == index {
                let a = (ch ^ pass) as char;
                if ('A'..='Z').contains(&a) || ('a'..='z').contains(&
a) {
                    count += 1;
                }
            }
        }
        if count > max_count {
            max_count = count;
            key = pass as char;
        }
    }
    key
}
```

3）猜三个位置上的密码

```
let key = [
    guess_pass(&letters, 0),
    guess_pass(&letters, 1),
    guess_pass(&letters, 2),
];
println!("key: {:?}", key);
```

4）解码，求和

```
let mut sum: u32 = 0;
for (i, ch) in letters.iter().enumerate() {
    let a = ch ^ (key[i % 3] as u8);
    sum += a as u32;
    print!("{}", a as char);
}
println!("\n");
println!("{}", sum);
```

第 79 题 密码推断

网上银行常用的一种密保手段是向用户询问密码中的随机三位字符。例如，如果密码是 531278，询问第 2、3、5 位字符，正确的回复应当是 317。

在文本文件 keylog.txt 中包含了 50 次成功登陆的正确回复。

假设三个字符总是按顺序询问的，分析这个文本文件，给出这个未知长度的密码最短的一种可能。

解题步骤

1）分析 keylog.txt

文件中前几个数：319、680、180、690、129、620，根据一个 3 位数回复，比如 319，能够知道 3 出现在 1 前面，1 出现在 9 前面。

读文件，把这种信息用元组保存起来。

```
let data = std::fs::read_to_string("keylog.txt").expect("打开文件失败
");
let data2 = data.trim().replace("\r", "");
let lines = data2.split("\n");

let mut set: Vec<(&str, &str)> = vec![];
for line in lines {
    let a = &line[..1];
    let b = &line[1..=1];
    let c = &line[2..];
    if !set.contains(&(a, b)) {
        set.push((a, b));
    }
    if !set.contains(&(b, c)) {
        set.push((b, c));
    }
}
```

2）GraphViz

排除掉重复的元素，手工分析这些先后顺序，也可以得到一个初步结果。如果你知道有一个 GraphViz 这个画图利器，把刚才的元素关系生成图元命令。

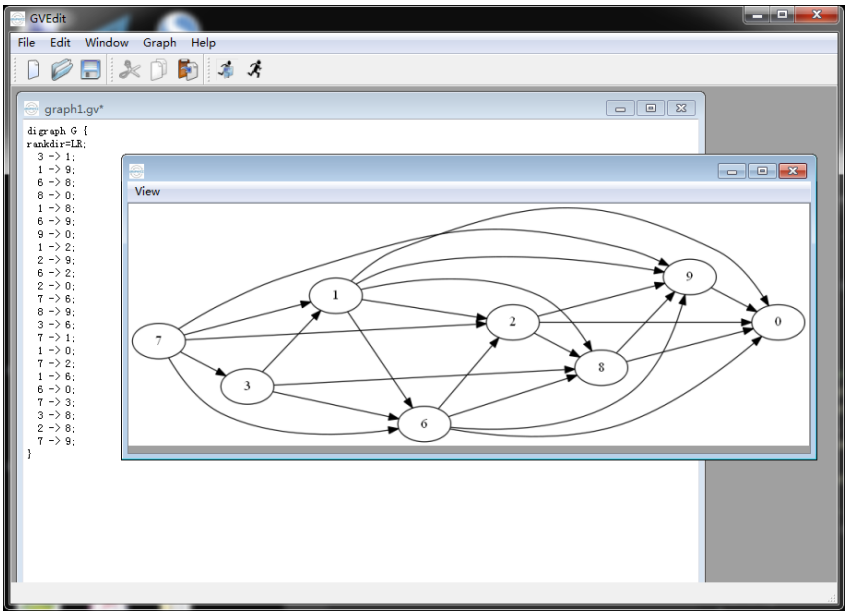
```
println!("digraph G {{");
println!("  rankdir=LR;");
for (a, b) in set {
    println!("  {} -> {};", a, b);
```

```
}
println!("}}");
```

程序输出这样一组文本，它们是 GraphViz 的图元命令：

```
digraph G {
rankdir=LR;
  3 -> 1;
  1 -> 9;
  6 -> 8;
  8 -> 0;
  1 -> 8;
  6 -> 9;
  9 -> 0;
  1 -> 2;
  2 -> 9;
  6 -> 2;
  2 -> 0;
  7 -> 6;
  8 -> 9;
  3 -> 6;
  7 -> 1;
  1 -> 0;
  7 -> 2;
  1 -> 6;
  6 -> 0;
  7 -> 3;
  3 -> 8;
  2 -> 8;
  7 -> 9;
}
```

从 graphviz.org 网站下载并安装 GraphViz 软件，用上面的几行文本可以快速生成图形。



73162890，多么醒目的答案。

第二篇 难度为 10%

这一部分是 10%的难度。

第 54 题 扑克手牌

在扑克游戏中，玩家的手牌由五张牌组成，其等级由低到高分别为：

- ✧ 单牌：牌面最大的一张牌。
- ✧ 对子：两张牌面一样的牌。
- ✧ 两对：两个不同的对子。
- ✧ 三条：三张牌面一样的牌。
- ✧ 顺子：五张牌的牌面是连续的。
- ✧ 同花：五张牌是同一花色。
- ✧ 葫芦：三条带一个对子。
- ✧ 四条：四张牌面一样的牌。
- ✧ 同花顺：五张牌的牌面是连续的且为同一花色。
- ✧ 同花大顺：同一花色的 10、J、Q、K、A。

牌面由小到大的顺序是：2、3、4、5、6、7、8、9、10、J、Q、K、A。如果两名玩家的手牌处于同一等级，那么牌面较大的一方获胜；例如，一对 8 胜过一对 5（参见例 1）；如果牌面相同，例如双方各有一对 Q，那么就比较玩家剩余的牌中最大的牌（参见例 4）；如果 最大的牌相同，则比较次大的牌，依此类推。

S 代表黑桃(Spade)，H 表示红桃(Heart)，D 表示方块(Diamond)，C 表示梅花(Club)，T 表示 10(Ten)，考虑以下五局游戏中双方的手牌：

手牌	玩家1	玩家2	胜者
1	5H 5C 6S 7S KD	2C 3S 8S 8D TD	玩家2
	一对5	一对8	
2	5D 8C 9S JS AC	2C 5C 7D 8S QH	玩家1
	单牌A	单牌Q	
3	2D 9C AS AH AC	3D 6D 7D TD QD	玩家2
	三条A	同花方片	

4	4D 6S 9H QH QC	3D 6D 7H QD QS	玩家1
	一对Q	一对Q	
	最大单牌9	最大单牌7	
5	2H 2D 4C 4D 4S	3C 3D 3S 9S 9D	玩家1
	葫芦	葫芦	
	（三条4）	（三条3）	

在 poker.txt 文本文件中，包含有两名玩家一千局的手牌。每一行包含有 10 张牌（均用一个空格隔 开）：前 5 张牌属于玩家 1，后 5 张牌属于玩家 2。你可以假定所有的手牌都是有效的（没有无效的字符 或是重复的牌），每个玩家的手牌不一定按顺序排列，且每一局都有确定的赢家。

其中有多少局玩家 1 获胜?

第一步： 先读文件，将玩家 1 和玩家 2 的牌分开。

第 22 题里已经学会了读文件，并且将字符串分隔成向量，再利用切片功能将前 5 个赋给玩家 1，后 5 个 赋给玩家 2。

```
let data = std::fs::read_to_string("poker.txt").expect("打开文件出错
");
let data2 = data.replace("\r\n", "\n");
let lines = data2.trim().split('\n');
for line in lines {
    let hand1 = &line[..14];
    let hand2 = &line[15..];
    println!("{:?} {:?}", hand1, hand2);
}
```

第二步： 多文件管理

这个项目涉及到手牌、牌张、花色等概念，适合用面向对象的编程思路。Rust 项目对多个源文件的功能支持也相当不错，main.rs 放主程序，poker.rs 放扑克相关的模块。

一手牌 Hand 由多张牌 Card 组成，一个 Card 由牌点（用 8 位整数表示）和花色 Suit 构成，花色只有 4 种，适合用枚举表示。Rust 里的枚举看上去与 C/C#/Java 等语言的枚举很像，但实际上它的功能远远不是一个简单的枚举。

```
//文件poker.rs
```

```
pub enum Suit {
    Spade,   // 黑桃
    Heart,   // 红桃
    Diamond, // 方块
    Club,    // 梅花
}

pub struct Card {
    value: u8, // 用2到14表示2, 3, ..., 10, J, Q, K, A
    suit: Suit,
}

pub struct Hand {
    cards: Vec<Card>,
}
```

main.rs 需要加一行语句，告诉主程序要使用 poker.rs 中定义的模块。

```
mod poker;
```

这个时候，程序可以编译，会给出几个警告，提示 Hand，Card 和 Suit 这些类型从来没用过。 第三步： 构建一张牌 Card 我们的任务要通过一个字符串构建出一个 Card 对象。比如，"8C"构建出梅花 8，"TS"构建出黑桃 10，"KC"为梅花 K，"9H"为红桃 9，"4S"为黑桃 4。

这个时候要先学会 Rust 中的 Trait 概念，Trait 这个东西很像 Java/C#里的接口，但又不是。Rust 内置不 支持构造函数，下面这段代码相当于给 Card 定义了一个静态方法 new()，相当于其它语言里的构造函数。

```
impl Card {
    pub fn new(str_card: &str) -> Card {
        let first_char = str_card.chars().next().unwrap();
        let card_value = "..23456789TJQKA"
            .chars()
            .position(|c| c == first_char)
            .unwrap() as u8;
        let second_char = str_card.chars().nth(1).unwrap();
        let card_suit = if second_char == 'S' {
            Suit::Spade
        } else if second_char == 'H' {
            Suit::Heart
        } else if second_char == 'D' {
            Suit::Diamond
        } else {
            Suit::Club
        };
        Card {
            value: card_value,
            suit: card_suit,
        }
```

```
    }
}
```

主程序里可以构造一个 card(这里用梅花 8)，尝试打印出来。

```
let card = poker::Card::new("8C");
println!("{:?}", card);
```

编译时 Rust 会给出相当清楚的错误信息，还给出了修改建议

```
error[E0277]: `poker::Card` doesn't implement `std::fmt::Debug`
 - -> src\main.rs:14:22
    |
14 |   println!("{:?}", card);
    |   ^^^^ `poker::Card` cannot be formatted using `{:?}`
    |
= help: the trait `std::fmt::Debug` is not implemented for `poker::Ca
rd`
= note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug
`
= note: required by `std::fmt::Debug::fmt`
```

我们声明了一个新类型 Card，但系统并不知道如何把它转换成字符串显示出来。按照提示，我们在 Card 和 Suit 前面各加上一行语句，让系统帮我们自动实现一个输出格式。

```
#[derive(Debug)]
pub enum Suit {
    Spade,   // 黑桃
    Heart,   // 红桃
    Diamond, // 方块
    Club,    // 梅花
}

#[derive(Debug)]
pub struct Card {
    value: u8, // 用2到14表示2, 3, ..., 10, J, Q, K, A
    suit: Suit,
}
```

此时程序可以顺利编译，运行可以得到如下结果：

```
Card { value: 8, suit: Club }
```

输出得虽然有点复杂，但容易理解。如果我们就想输出"8C"这样的字符串，则需要实现 Display 这个 trait 里的 fmt()函数。注意 write!语句后面千万别习惯性地加个分号，否则出现的编译错误让人好困惑！

```
use std::fmt::{Display, Error, Formatter};
```

```
impl Display for Suit {
    // 只用一个字母表示: S,H,D,C
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error> {
        let name = format!("{:?}", self);
        write!(f, "{}", &name[..1])
    }
}

impl Display for Card {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error> {
        let first_char = "..23456789TJQKA".chars().nth(self.value
as usize).unwrap();
        write!(f, "{}{}", first_char, self.suit)
    }
}
```

现在构建 5 张牌，输出出来。

```
let card1 = poker::Card::new("8C");
let card2 = poker::Card::new("TS");
let card3 = poker::Card::new("KC");
let card4 = poker::Card::new("9H");
let card5 = poker::Card::new("4S");
println!("{} {} {} {} {}", card1, card2, card3, card4, card5);
```

第四步： 构建一手牌 Hand

对于 Hand，也要实现 fmt()函数，还要实现一个构造函数 new()。

```
use itertools::Itertools;
impl Display for Hand {
    fn fmt(&self, f: &mut Formatter) -> Result<(), Error> {
        let str_cards = self.cards.iter().map(|x| x.to_string()).j
oin(" ");
        write!(f, "{}", str_cards)
    }
}

impl Hand {
    pub fn new(str_cards: &str) -> Hand {
        let mut v = vec![];
        for s in str_cards.split(' ') {
            v.push(Card::new(s));
        }
        Hand { cards: v }
    }
}
```

主程序这样写：

```
let hand = poker::Hand::new("8C TS KC 9H 4S");
println!("{}", hand);
```

第五步： 比较两个对象的大小

现在我们想比较两手牌的大小，主程序写成这样。

```
let hand1 = poker::Hand::new("8C TS KC 9H 4S");
let hand2 = poker::Hand::new("7D 2S 5D 3S AC");
if hand1 > hand2 {
    println!("player1 wins" );
}
```

想让两个对象能够相互比较大小，需要实现四个 trait（Ord、PartialOrd、Eq 和 PartialEq）中的几个函数。

```
use std::cmp::{Ord, Ordering};
impl Ord for Hand {
    fn cmp(&self, other: &Self) -> Ordering {
        self.to_string().cmp(&other.to_string())
    }
}

impl PartialOrd for Hand {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl Eq for Hand {}

impl PartialEq for Hand {
    fn eq(&self, other: &Self) -> bool {
        self.to_string().eq(&other.to_string())
    }
}
```

现在，这里的比较逻辑还没有实现，暂时用字符串的比较代替。有几点要留意：

1）Ord 里的 cmp()函数，PartialOrd 里的 partial_cmp()函数，一个是表示全序，一个表示偏序。

2）cmp()和 partial_cmp()两个函数的返回值有点区别，后面的多 Option<>

3）Eq 里的内容是空的，但必须要写

4）PartialEq 里的函数名是 eq()

5）实现了这些 trait 后，程序会自动理解“<”、“>”、“==”这些比较运算符。

第六步：比较两手牌的大小

这时需要细心了，判断同花、顺子、四条、三条、对子等情况，为了后面的比较，我声明了一个枚举 enum，用来区分各种牌型，从这里可以领略 Rust 里枚举的强大。

```
pub enum HandType {
    HighCard(u8, u8, u8, u8, u8), // 单牌
    //对子
    OnePair {
        value_pair: u8,
        max_remain: u8, // 除了对子之外，剩下最大的牌点
    },
    //两对
    TwoPairs {
        high_pair: u8,  // 最大的一对
        low_pair: u8,   // 最小的一对
        max_remain: u8, // 除了两对之外，剩下最大的牌点
    },
    KindThree(u8), // 三条
    Straight(u8),  // 顺子
    Flush(u8),     // 同花
    //葫芦，即三条带一个对子
    FullHouse {
        value_kind_three: u8,
        value_pair: u8,
    },
    KindFour(u8),      // 四条
    StraightFlush(u8), // 同花顺
    RoyalFlush,        // 同花大顺
}
```

再声明两个函数 ranking1()和 ranking2()，两次比较后能够区分大小。

```
impl HandType {
    pub fn ranking1(&self) -> u8 {
        match self {
            HighCard(_, _, _, _, _) => 0,
            OnePair { .. } => 1,
            TwoPairs { .. } => 2,
            KindThree(_) => 3,
            Straight(_) => 4,
            Flush(_) => 5,
            FullHouse { .. } => 6,
            KindFour(_) => 7,
            StraightFlush(_) => 8,
            RoyalFlush => 9,
        }
    }
```

```
    pub fn ranking2(&self) -> u64 {
    // ...
    }
}
```

这里有一个"..."的语法点，忽略结构体里的内容。

```
FullHouse { .. } => 6,
```

相当于：

```
FullHouse {
    value_kind_three: _,
    value_pair: _,
} => 6,
```

Ord 里的 cmp()和 PartialEq 里的 eq()的逻辑也要相应修改一下。

```
impl Ord for HandType {
    fn cmp(&self, other: &Self) -> Ordering {
        self.ranking1()
            .cmp(&other.ranking1())
            .then(self.ranking2().cmp(&other.ranking2()))
    }
}
impl PartialEq for HandType {
    fn eq(&self, other: &Self) -> bool {
        self.ranking1() == other.ranking1() && self.ranking2() ==
other.ranking2()
    }
}
```

剩下就是依次判断各种情况，需要足够的耐心和测试。

```
use itertools::Itertools;
impl Hand {
    //是否五张牌同一个花色。
    fn is_flush(&self) -> bool {
        self.cards.iter().map(|card| card.suit).all_equal()
    }

    //判断五张牌是否连号，先将牌面数值从小到大排序，两两之差为1就是顺
子。
    fn is_straight(&self) -> bool {
        let mut v: Vec<u8> = self.cards.iter().map(|x| x.value).co
llect();
        v.sort();
        (0..4).all(|i| v[i + 1] - v[i] == 1) //两两之差都为1
    }
}
```

第七步：将相关类整理到一个文件夹下

源文件较多时，可以放在一个文件夹下，模块里还可以有子模块。比如这样组织文件：

```
src/
+---main.rs
+---poker/
    +---card.rs
    +---hand.rs
    +---hand_type.rs
    +---mod.rs
    +---suit.rs
```

poker 文件夹可以自动识别为一个 mod，需要 mod.rs 文件的配合，这里声明用到的子模块，编译器可以自动找到相应的源文件。

```
pub mod card;
pub mod hand;
pub mod hand_type;
pub mod suit;
```

hand.rs 文件里使用其它模块的内容时，需要用 use 语句。

```
use super::card::*;
use super::hand_type::*;
```

第 69 题 欧拉总计函数与最大值

在小于 n 的数中，与 n 互质的数的数目记为欧拉总计函数 φ(n)（有时也称为 φ 函数）。例如，因为 1、2、4、5、7 和 8 均小于 9 且与 9 互质，故 φ(9)=6。

n	互质的数	ϕ(n)	n/ϕ(n)
2	1	1	2
3	1，2	2	1.5
4	1，3	2	2
5	1，2，3，4	4	1.25
6	1，5	2	3
7	1，2，3，4，5，6	6	1.1666…
8	1，3，5，7	4	2
9	1，2，4，5，7，8	6	1.5
10	1，3，7，9	4	2.5

可以看出，对于 n ≤ 10，当 n=6 时 n/φ(n)取得最大值。

当 n ≤ 1,000,000 时，求使得 n/φ(n)取得最大值的 n。

解题过程：

遇到一个复杂的问题，可以先从简单的情况入手，寻找一些规律，再慢慢逼近最终的问题。

第一步： 暴力求解

先根据题意暴力求解。

```
fn main() {
    let mut max_ratio = 0_f64;
    for n in 2..=1_000_000 {
        // 最大公约数为1，表示互质
        let phi = (1..n).filter(|&x| gcd(x, n) == 1).count();
        let ratio = (n as f64) / (phi as f64);
        if ratio > max_ratio {
            println!("n= {:6}  phi={:6}  n/phi= {:.4}", n, phi, ra
tio);
            max_ratio = ratio;
        }
    }
}

// 最大公约数
fn gcd(a: u64, b: u64) -> u64 {
    if b == 0 {
        a
    } else {
        gcd(b, a % b)
    }
}
```

可以得到部分结果，计算到 2310 非常快，计算到 30030 则要 1 分钟，计算到 1 百万估计得几天。

```
n=      2  phi=     1  n/phi= 2.0000
n=      6  phi=     2  n/phi= 3.0000
n=     30  phi=     8  n/phi= 3.7500
n=    210  phi=    48  n/phi= 4.3750
n=   2310  phi=   480  n/phi= 4.8125
n=  30030  phi=  5760  n/phi= 5.2135
```

第二步： 找规律

n	互质的数	φ(n)	真因子	n/φ(n)
2	*1*	1		2
3	*1,2*	2		1.5
4	*1,3*	2	2	2
5	*1,2,3,4*	4		1.25
6	*1,5*	2	2*3	3
7	*1,2,3,4,5,6*	6		1.1666...
8	*1,3,5,7*	4	2^3	2
9	*1,2,4,5,7,8*	6	3^2	1.5
10	*1,3,7,9*	4	2*5	2.5
30	*1,3,7,11*	8	2*3*5	3.75
210		48	2*3*5*7	4.375
2310		480	2*3*5*7*11	4.8125
30030		5760	2*3*5*7*11*13	5.2135

通过前面几个结果，可以发现是连续几个素数的乘积，所以问题可以变为求哪些连续素数的乘积小于 1 百万。

```
// 2 * 3 * 5 * 7 * 11 * ... 找到最多的素数，乘积在1百万以内
let mut pset = PrimeSet::new();
let mut prod = 1;
for p in pset.iter() {
    if prod * p > 1_000_000 {
        break;
    }
    prod *= p;
    println!("prime={} {}", p, prod);
}
println!("{}", prod);
```

这里可以练习一下函数式编程，好像可读性很差。

```
let mut some_primes = (2..).filter(|&x| primes::is_prime(x));
let mut prod = 1;
let some_products = std::iter::repeat_with(|| {
    let tmp = prod;
    prod *= some_primes.next().unwrap();
    tmp}).take_while(|&x| x <= 1_000_000)
         .collect::<Vec<_>>();
println!("{:?}", some_products);
println!("{:?}", some_products.last().unwrap());
```

用 itertools 写一下，这样好一些。

```
let mut some_primes = (2..).filter(|&x| primes::is_prime(x));
let result = itertools::iterate(1, |&prod| prod * some_primes.next
```

```
().unwrap())
    .take_while(|&x| x <= 1_000_000)
    .last()
    .unwrap();
println!("{:?}", result);
```

第三步： 补一下数学知识

这个欧拉总计函数，可以推导出数学公式，网上很容易搜到一大堆相关内容。

若$n = p_1^{k_1} p_2^{k_2} \cdots p_r^{k_r}$

则$\varphi(n) = \prod_{i=1}^{r} p_i^{k_i - 1}(p_i - 1) = \prod_{p|n} p^{\alpha_p - 1}(p-1) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$。

其中α_p是使得p^{α}整除n的最大整数α（这里$\alpha_{p_i} = k_i$）。

例如$\varphi(72) = \varphi(2^3 \times 3^2) = 2^{3-1}(2-1) \times 3^{2-1}(3-1) = 2^2 \times 1 \times 3 \times 2 = 24$

```
let mut max_ratio = 0_f64;
for n in 2..=1_000_000 {
    let all_factors = primes::factors(n);
    let uniq_factors = primes::factors_uniq(n);

    let mut phi = 1;
    for p in uniq_factors {
        let k = all_factors.iter().filter(|&x| *x == p).count();
        phi *= p.pow(k as u32 - 1) * (p-1);
    }
    let ratio = (n as f64) / (phi as f64);
    if ratio > max_ratio {
        println!("n= {:6}  phi={:6}  n/phi= {:.4}", n, phi, rati
o);
        max_ratio = ratio;
    }
}
```

看看后一种写法的例子：

$$\varphi(x) = x\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\left(1 - \frac{1}{p3}\right)\cdots\cdots\left(1 - \frac{1}{p_n}\right)$$

其中p1, p2……pn为x的所有质因数，x是不为0的整数。

- ψ（10）=10×（1−1/2）×（1−1/5）=4；
- ψ（30）=30×（1−1/2）×（1−1/3）×（1−1/5）=8；
- ψ（49）=49×（1−1/7）=42。

用这个公式，程序写起来更简单。

```
let mut max_ratio = 0_f64;
for n in 2..=1_000_000 {
    let uniq_factors = primes::factors_uniq(n);
    let mut phi = n;
    for p in uniq_factors {
        phi = phi * (p-1) / p;
    }
    let ratio = (n as f64) / (phi as f64);
    if ratio > max_ratio {
        println!("n= {:6}  phi={:6}  n/phi= {:.4}", n, phi, rati
o);
        max_ratio = ratio;
    }
}
```

第 71 题 有序分数

考虑形如 n/d 的分数，其中 n 和 d 均为正整数。如果 n < d 且其最大公约数为 1，则该分数称为最简真分数。

如果我们将 d ≤ 8 的最简真分数构成的集合按大小升序列出，我们得到：

1/8，1/7，1/6，1/5，1/4，2/7，1/3，3/8，2/5，3/7，1/2，4/7，3/5，5/8，2/3，5/7，3/4，4/5，5/6，6/7，7/8 可以看出 2/5 是 3/7 直接左邻的分数。

将所有 d ≤ 1,000,000 的最简真分数按大小升序排列，求此时 3/7 直接左邻的分数的分子。

解题过程：

这道题实在太简单，根据分母找到最接近 3/7 的分子就行了，直接上代码。

```
fn main() {
    let mut max = 0.0;
    let mut numerator = 0;
    for d in 2..=1_000_000 { // 分母
        let n = d * 3 / 7;
        if n * 7 == d * 3 {
            continue;
        }
        let q = (n as f64) / (d as f64);
        if q > max {
            max = q;
            numerator = n;
```

```
            //println!("{} / {} = {}", numerator, denom, q);
        }
    }
    println!("{}", numerator);
}
```

第 76 题 加和计数

将 5 写成整数的和有 6 种不同的方式:

4 + 1

3 + 2

3 + 1 + 1

2 + 2 + 1

2 + 1 + 1 + 1

1 + 1 + 1 + 1 + 1

将 100 写成整数的和有多少种不同的方式?

算法 1：递归算法

这种题首先能够想到的是递归算法，从简单的情况分析入手，对于 6 的分解，假设第一个数字确定，后面的数字肯定不能大于第 1 个数，并且还有总和的限制。

s							i	后面几个数的和s-i	后面的所有数都不能大于	
6	5	1					首数是5	1	min(5, 1)	w(1,1)
6	4	2					首数是4	2	min(4, 2)	w(2,2)
6	4	1	1							
6	3	3					首数是3	3	min(3, 3)	w(3,3)
6	3	2	1							
6	3	1	1	1						
6	2	2	2				首数是2	4	min(2, 4)	w(4,2)
6	2	2	1	1						
6	2	1	1	1	1					
6	1	1	1	1	1	1	首数是1	5	min(1, 5)	w(5,1)

这样写出递归函数，第一个参数是总和，第二个参数表示里面的每个数都不能超过 limit。

```
fn summation_ways(s: u32, limit: u32) -> u32 {
    if limit <= 1 {
        return 1;
    }
    let mut count = 0;
    for i in 1..=limit {
        count += summation_ways(s - i, i.min(s - i));
    }
    count
}
```

主程序只需要这样调用即可：

```
    println!("{}", summation_ways(100, 99));
```

这种算法效率比较低，有大量的重复计算，可以缓存部分结果进一步提高效率。

算法 2：利用 Partition Function 函数

这个序列在数学上也有专门的名字，称为 Partition Function，参考网站：

https://mathworld.wolfram.com/PartitionFunctionP.html

http://oeis.org/A000041

这个序列有非常多有趣的性质，估计花几个月也琢磨不完，其中有一个递归公式可以直接拿来用。

$$P(n)=\sum_{k=1}^{n}(-1)^{k+1}\left[P\left(n-\tfrac{1}{2}k(3k-1)\right)+P\left(n-\tfrac{1}{2}k(3k+1)\right)\right]$$

注意这个序列与原题的初始值、结果有差 1 的区别，还要引入缓存，否则计算量也非常大。

```
fn p(cache: &mut [i64], n: i64) -> i64 {
    if n < 0 {
        return 0;
    }
    if n == 0 {
        return 1;
    }
    if cache[n as usize] > 0 {
        return cache[n as usize];
    }
    let mut s = 0;
    for k in 1..=n {
        let n1 = n - k * (3 * k - 1) / 2;
        let n2 = n - k * (3 * k + 1) / 2;
```

```
        let sign = if (k + 1) % 2 == 0 { 1 } else { -1 };
        s += sign * (p(cache, n1) + p(cache, n2));
    }
    cache[n as usize] = s;
    s
}
```

第 81 题 两个方向的路径和

在如下的 5 乘 5 矩阵中，从左上方到右下方始终只向右或向下移动的最小路径和为 2427，由标粗的路径给出。

131	673	234	103	18
201	**96**	**342**	965	150
630	803	**746**	**422**	111
537	699	497	**121**	956
805	732	524	**37**	**331**

在这个 31K 的文本文件 matrix.txt 中包含了一个 80 乘 80 的矩阵，求出从该矩阵的左上方到右下方始终只向右和向下移动的最小路径和。

解题过程：

遇到一个复杂的问题，可以先尝试解决简单的情况，然后慢慢逼近最终的问题。

第一步： 先把文件内容读到二维数组中

以前写过读文件到一维数组中，这次稍微有点变化。

```
let mut mat = vec![vec![0; 80]; 80];
let data = std::fs::read_to_string("matrix.txt").expect("读文件失败");
let lines = data.trim().split('\n');
for (i,line) in lines.enumerate() {
    let lines_data = line.split(',')
        .map(|x| x.parse::<usize>().unwrap());
    for (j, w) in lines_data.enumerate() {
        mat[i][j] = w;
```

```
    }
}
println!("{:?}", mat);
```

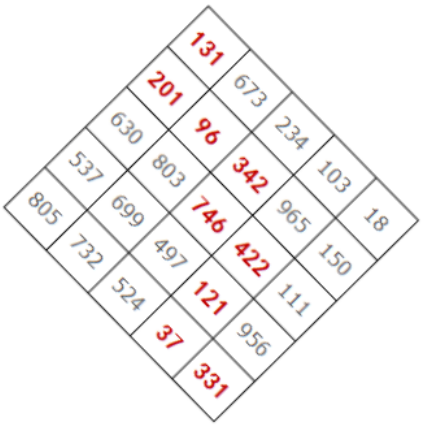
更通用的写法是这样：

```
let f = BufReader::new(File::open("matrix.txt").unwrap());
let arr: Vec<Vec<usize>> = f
    .lines()
    .map(|line| {
        line.unwrap()
            .split(',')
            .map(|s| s.parse().unwrap())
            .collect()
    })
    .collect();
println!("{:?}", arr);
```

第二步：

这道题类似于第 67 题（求最大路径和 II），只需要把这个矩阵旋转 45 度就可以变为相同的问题。

131	673	234	103	18
201	**96**	**342**	965	150
630	803	**746**	**422**	111
537	699	497	**121**	956
805	732	524	**37**	**331**



https://note.youdao.com/yws/public/resource/28aed5663443f7988df82ac9c7fe1ace/D0C888F6CCCF476888E27C20807A3674?ynotemdtimestamp=1583646589046

从底向上，从右向左依次计算即可。

```
fn compute_min_path(w: &Vec<Vec<usize>>) -> Vec<Vec<usize>> {
    let cols = w[0].len();
    let rows = w.len();
    let mut path: Vec<Vec<usize>> = vec![vec![0; cols]; rows];
    for i in (0..rows).rev() {
        for j in (0..cols).rev() {
            if i == rows - 1 && j == cols - 1 {
                // 最右下角
```

```
                path[i][j] = w[i][j];
            } else if i == rows - 1 {
                // 最底行，只有右节点
                path[i][j] = w[i][j] + path[i][j + 1];
            } else if j == cols - 1 {
                // 最右列，只有下节点
                path[i][j] = w[i][j] + path[i + 1][j];
            } else {
                path[i][j] = w[i][j] + path[i][j + 1].min(path[i + 1][j]);
            }
        }
    }
    path
}
```

最后的结果存储在矩阵的最左上角。

```
let min_path = compute_min_path(&arr);
println!("{}", min_path[0][0]);
```

第 99 题 最大的幂

比较两个如 2^{11} 和 3^7 这样写成幂的形式的数并不难，因为 $2^{11} = 2048 < 3^7 = 2187$。

然而，想要验证 $632382^{518061} > 519432^{525806}$ 就会变得非常困难，因为这两个数都包含有超过三百万位数字。

22K 的文本文件 base_exp.txt 有一千行，每一行有一对底数和指数，找出哪一行给出的幂的值最大（注意：文件的前两行就是上述两个例子）。

解题过程：

只需应用简单的对数知识就行。

```
let data = std::fs::read_to_string("base_exp.txt").expect("读文件失败
");
let lines = data.trim().split('\n');

let mut max = 0_f64;
for (i, line_str) in lines.enumerate() {
    let line_data: Vec<usize> = line_str
        .split(',')
```

```
        .map(|x| x.parse::<usize>().unwrap())
        .collect();
    let (base, exp) = (line_data[0], line_data[1]);
    let v = (exp as f64) * (base as f64).log10();
    if v > max {
        println!("{} {} {}", i + 1, line_str, v);
        max = v;
    }
}
```

第 357 题 生成素数的整数

考虑 30 的因子：1、2、3、5、6、10、15、30。可以看出，对于 30 的每个因子 d，d+30/d 都是素数。

存在不超过 100,000,000 的正整数 n，使得对于 n 的每个因子 d，d+n/d 都是素数，求所有这样的数 n 的和。

第一步：根据题意，暴力求解

利用第 10 题完成的求素数的 myprime 模块，第 12 题的求一半因子的函数，很容易写出程序。

```
mod myprime;

fn main() {
    let large_num = 100_000_000;
    // 为防止判断 d + n / d 是否为素数时越界，多分配一点空间
    let prime_mask = myprime::prime_sieve(large_num + 2);
    println!("finished prime generation");

    let mut sum = 0;
    for n in 1..=large_num {
        if is_divisor_prime(&prime_mask, n) {
            sum += n;
        }
    }
    println!("{}", sum);
}

fn is_divisor_prime(prime_mask: &[bool], n: usize) -> bool {
    let s = (n as f32).sqrt() as usize;

    // 求一半因子
    let half_divisors: Vec<usize> = (1..=s).filter(|x| n % x == 0).c
ollect();

    for d in half_divisors {
        let p = d + n / d;
```

```
        if !prime_mask[p] {
            return false;
        }
    }
    true
}
```

程序的运行效率很差，计算一百万之内的结果也需在 7 秒多，计算 1 个亿很长时间没结果。

第二步：优化性能

问题主要出在求一半因子那条语句，一个数的因子非常非常多，但大部分情况下，只需计算前几个因子，就可以发现(d + n / d)并不是素数，之后的那些因子根本不需要计算出来。

只需修改一行语句，就可以大幅提高计算效率。

```
    let half_divisors = (1..=s).filter(|x| n % x == 0);
```

这条语句中没有执行 collect()函数，也就是说只是定义了一个迭代器，还没有计算一个因子，有延迟评价的效果。后面在 for 循环中调用迭代器，很多情况下直接 reture false 返回函数结果了。

改进后计算 1 亿的结果需要 7 秒多。

第 387 题 哈沙德数

能够被其各位数字和整除的数被称为**哈沙德数**或奈文数。

201 是一个哈沙德数，因为它能被 3 整除（其各位数字和是 3）。如果我们截去 201 的最后一位数字，我们得到 20，同样是一个哈沙德数。如果我们截去 20 的最后一位数字，我们得到 2，仍然是一个哈沙德数。如果一个哈沙德数不断截去最后一位数字的结果始终是哈沙德数，我们称之为**可右截哈沙德数**。

此外：201 / 3 = 67 是一个素数。如果一个哈沙德数被其各位数字除的结果是一个素数，我们称之为**强哈沙德数**。

现在我们取素数 2011。如果我们截去 2011 的最后一位数字，我们得到 201，一个可右截的强哈沙德数。我们称这样的素数为**可右截强哈沙德素数**。

已知所有小于 10000 的可右截强哈沙德素数之和为 90619。求所有小于 10^{14} 的

可右截强哈沙德素数之和。

第一步：先解决简单的 10000 以内的问题

根据题意，将各个函数写出来，右截哈沙德数的判断稍微注意一下，为了效率，这里的 digits 没有执行 collect()函数。

```
#[test]
fn test_10000() {
    let h:Vec<u64> = (1..10000).filter(|x| is_right_truc_strong_hars
had_prime(*x)).collect();
    println!("{:?}", h);
    assert_eq!(h.iter().sum::<u64>(), 90619);
}

fn sum_of_digits(n: u64) -> u64 {
    n.to_string()
        .chars()
        .map(|c| c.to_digit(10).unwrap())
        .sum::<u32>() as u64
}

// 哈沙德数：能够被其各位数字和整除
fn is_harshad(n: u64) -> bool {
    n % sum_of_digits(n) == 0
}

// 可右截哈沙德数：不断截去最后一位数字的结果始终是哈沙德数
fn is_right_truc_harshad(n: u64) -> bool {
    let s = n.to_string();
    let digits = s.chars().map(|c| c.to_digit(10).unwrap() as u64);
    let mut num: u64 = 0;
    let mut sum: u64 = 0;
    for d in digits {
        num = num * 10 + d;
        sum += d;
        if num % sum != 0 {
            return false;
        }
    }
    true
}

use num_integer;
fn is_strong_harshad(n: u64) -> bool {
    let sum_digits = sum_of_digits(n);
    let (q, r) = num_integer::div_rem(n, sum_digits);
    r == 0 && primes::is_prime(q)
}

fn is_right_truc_strong_harshad_prime(n: u64) -> bool {
```

```
    let m = n / 10;
    m!= 0 && primes::is_prime(n) && is_strong_harshad(m) && is_right
_truc_harshad(m)
}
```

单元测试很容易通过，但在计算 10^9 的可右截强哈沙德素数之和的时候，效率就已经非常差了，题目要求 10^{14} 的可右截强哈沙德素数，得考虑优化算法。

第二步：寻找规律，改进算法

把一百万之内的可右截强哈沙德素数打印出来，看看有什么规律。为了便于观察，将 3 位数、4 位数、5 位数、6 位数换行显示。

[181, 211, 271, 277, 421, 457, 631,

2011, 2017, 2099, 2473, 2477, 4021, 4027, 4073, 4079, 4231, 4813, 4817, 6037, 8011, 8017, 8039, 8461, 8467,

20071, 20431, 40867, 48091, 84061, 84067,

400237, 400277]

好像并没有特别明显的规律，只是发现满足条件的数并不太多。

把可右截的哈沙德数打印出来，按不同位数放在电子表格的不同列中，可以发现规律。左侧存在的字头，右侧才可以继续存在，左侧有 120，右侧有 1200，1204，1206 等，左侧不存在 123，右侧就不会有 123？这样的四位数存在。

1	10	100	1000
			1002
			1008
		102	1020
			1026
		108	1080
			1088
	12	120	1200
			1204
			1206
		126	1260
	18	180	1800
2	20	200	2000
			2001
			2004
			2007
		201	2010
			2016
		204	2040
			2043
		207	2070
		209	2090
	21	210	2100
			2106
		216	2160
	24	240	2400
			2401
			2403
			2408
		243	2430
		247	2470
			2478
	27	270	2700
			2704

现在可以写出递归程序，根据字头，尝试在后面添加 0, 1, 2, …, 9，如果是可右截哈沙德数，再进一步去找强哈沙德素数，注意字头只需搜索到 9_999_999_999_999_999 即可。

```
fn main() {
    let mut sum = 0;
    for i in 1..=9 {
        sum += harshad_loop(i);
    }
    println!("{}", sum);
}
```

```
fn harshad_loop(head: u64) -> u64 {
    if head >= 10_u64.pow(13) {
        return 0;
    }
    let mut sum = 0;
    if is_right_truc_harshad(head) {
        if is_strong_harshad(head) {
            // 素数只能以1，3，5，7，9结尾
            let harshad_primes = (1..=9)
                .step_by(2)
                .map(|i| head * 10 + i)
                .filter(|&x| primes::is_prime(x))
                .collect::<Vec<u64>>();
            if !harshad_primes.is_empty() {
                // println!("{:?}", harshad_primes);
            }
            sum += harshad_primes.iter().sum::<u64>();
        }
        for i in 0..=9 {
            sum += harshad_loop(head * 10 + i);
        }
    }
    sum
}
```

现在只需 1、2 秒就可以计算出结果。

第 493 题 彩虹之下

在容器中装有 70 个球，分别染上彩虹的七种颜色，每种颜色各有 10 个。

从容器中随机取出 20 个球，这些球中出现不同颜色球的数量的期望值是多少？

你的答案应当保留到小数点后 9 位小数（a.bcdefghij）。

第一步：模拟选球的过程

先模拟从 70 个球里选 20 个球的过程，可以大概一个大概的数值，虽然难于精确到小数点后 9 位，但可以给出一个粗略的估计。

假设 70 个球的编号是 0 到 69，颜色编号从 0 到 6，每 10 个一种颜色，x/10 则可以表示球的颜色编号。rand 模块中有一个 sample_iter()函数可以完成从 70 个球里选 20 个球的过程，itertools 里的 unique()函数可以统计出不重复的颜色值。

模拟一百万次，可以得到大约为 6.818 的结果。

```
use rand::{seq, thread_rng, Rng};
use itertools::Itertools;
fn simulate() {
    let mut rng = thread_rng();
    let total_samples = 1_000_000_u64;
    let mut sum = 0;
    for _i in 0..total_samples {
        let balls20 = seq::sample_iter(&mut rng, 0..70, 20).unwrap();
        let distinct_colors = balls20
            .into_iter()
            .map(|x| x / 10)
            .unique()
            .collect::<Vec<u32>>();
        //println!("{:?}", distinct_colors);
        sum += distinct_colors.len();
    }
    println!("{}", (sum as f64) / (total_samples as f64));
}
```

第二步：利用概率论的知识

原问题等价于：20 个球里出现至少 1 个红球的概率 + 至少 1 个绿球的概率 + ... + 至少 1 个紫球的概率。

由于出现红、绿、...、紫 7 种颜色球的概率是一样的，所以问题又等价于：

7 * {20 个球里出现至少 1 个红球的概率}

7 * (1 – {20 个球里没有出现 1 个红球的概率})

7 * (1 – {20 个球里没有红球出现的所有可能组合} / {70 个球里选 20 个球的所有可能组合})

没有红球出现，即从 70 个球里除掉 10 个红球，还有 60 个其它颜色的球，从里面选 20 个的情况共有 C(60, 20)种，所以得到：

$$7 * (1 - C_{60}^{20} / C_{70}^{20})$$

还可以进一步化简，得到：$7 * (1 - \frac{41*42*...*50}{61*62*...*70})$

现在用计算器也可以得到正确答案。

```
let mut prob = 1.0;
for i in 41..=50 {
    prob *= (i as f64) / ((i + 20) as f64);
```

```
}
println!("{:.9}", (1.0 - prob) * 7.0);
```

第三篇 难度为 15%

第 51 题（待完成）素数数字替换

将两位数*3 的第一个数字代换为任意数字，在九个可能值中有六个是素数：13、23、43、53、73 和 83。

将五位数 56**3 的第三和第四位数字代换为相同的任意数字，就得到了十个可能值中有七个是素数的最小例子，这个素数族是：56003、56113、56333、56443、56663、56773 和 56993。56003 作为这一族中最小的成员，也是最小的满足这个性质的素数。

通过将部分数字（不一定相邻）代换为相同的任意数字，有时能够得到八个素数，求满足这一性质的最小素数。

第一步：先实现两位数中替换掉一个数字的情况

一个整数 n 换掉某位上的数字（pos 为 0 表示个位，1 表示十位...）。

```
fn replace(n: u64, pos: usize, new_digit: u64) -> u64 {
    let p = 10_u64.pow(pos as u32);
    let d = n / p % 10;    // 取出这个位置上的数字
    n - d * p + new_digit * p
}
```

一个数 n 换掉某位(pos)形成所有素数的集合。

```
fn gen_prime_family(n: u64, pos: usize) -> Vec<u64> {
    let mut prime_family = vec![];
    for digit in 0..=9 {
        let generated = replace(n, pos, digit);
        if primes::is_prime(generated)
            && !prime_family.contains(&generated)
            && generated.to_string().len() == n.to_string().len()
        {
            prime_family.push(generated);
        }
    }
    prime_family
}
```

寻找所有两位字，就可以找到结果[13, 23, 43, 53, 73, 83]。

```
fn find_prime_family_2digits(num_primes: usize) -> Vec<u64> {
    for n in 10..=99 {
        // 先替换个位，再替换十位
        for pos in 0..=1 {
            let prime_family = gen_prime_family(n, pos);
```

```
            if prime_family.len() == num_primes {
                println!("seed: {} pos:{} {:?}", n, pos, prime_famil
y);
                return prime_family;
            }
        }
    }
    vec![]
}
```

第 62 题 立方数重排（待完成）

立方数 41063625（345^3）可以重排为另外两个立方数：56623104（384^3）和 66430125（405^3）。实际上，41063625 是重排中恰好有三个立方数的最小立方数。

求重排中恰好有五个立方数的最小立方数。

第 65 题（待完成）

第 73 题 统计一定范围内的分数

题目描述：

考虑形如 n/d 的分数，其中 n 和 d 均为正整数。如果 n < d 且其最大公约数为 1，则该分数称为最简真分数。

如果我们将 d ≤ 8 的最简真分数构成的集合按大小升序列出，我们得到：

1/8, 1/7, 1/6, 1/5, 1/4, 2/7, 1/3, 3/8, 2/5, 3/7, 1/2, 4/7, 3/5, 5/8, 2/3, 5/7, 3/4, 4/5, 5/6, 6/7, 7/8 可以看出在 1/3 和 1/2 之间有 3 个分数。

将 d ≤ 12,000 的最简真分数构成的集合排序后，在 1/3 和 1/2 之间有多少个分数？

解题过程：

第一步：直接根据题意，暴力求解

1/3 < n/d < 1/2

可以推导出 2 * n < d && d < 3 * n

找到最大公约数后，求出真分数，放在一个集合里。

```
let mut count = 0;
let mut v:Vec<(usize, usize)> = vec![];
for d in 2..=1200 {
    for n in 1..d {
        if 2 * n < d && d < 3 * n {
            let g = num::integer::gcd(n, d);
            let r = (n / g, d / g);
            if !v.contains(&r) {
                v.push(r);
                count += 1;
            }
        }
    }
}
println!("{}", count);
```

当 d 为 1000 时，可以很容易地计算出结果。但当 d 逐渐增大时，求解速度越来越慢，主要原因是数组中元素越来越多，判断一个元素是否在数组中，速度越来越慢。

第二步：

找最大公约数，并且维护一个庞大的数组，代价较大，当找到一个真分数时，可以排除掉很多(n,d)，比如找到 2/5 时，可以排除 4/10, 6/15, ... , 4800/12000。

针对[1, 12000]中的每一个 d，维护一个 exclude_list 数组，其中的元素约分后都已经统计过了，可以直接忽略掉。

```
let mut exclude_list = vec![vec![]; 12001];

let mut count = 0;
for d in 2..=12000 {
    for n in 1..d {
        if 2 * n < d && d < 3 * n
            && !exclude_list[d].contains(&n) {
            count += 1;
            for i in 2..=12000 / d {
                exclude_list[d * i].push(n * i);
            }
```

```
            }
        }
    }
println!("{}", count);
```

4 秒得到结果。

当 d 更大时，还可以进一步优化算法，这里不再讨论。

第 74 题（待完成）

第 85 题（待完成）

第 102 题（待完成）

第 345 题 矩阵和

题目描述：

在一个矩阵中选择部分元素，使得每个元素都是所在的行和列中唯一被选中的元素，这些元素的和的最大值定义为矩阵的矩阵和。例如，如下矩阵的矩阵和为 3315 (= 863 + 383 + 343 + 959 + 767)：

```
  7   53  183  439   863
497  383  563   79   973
287   63  343  169   583
627  343  773  959   943
767  473  103  699   303
```

求如下矩阵的矩阵和：

```
  7  53 183 439 863 497 383 563  79 973 287  63 343 169 583
627 343 773 959 943 767 473 103 699 303 957 703 583 639 913
447 283 463  29  23 487 463 993 119 883 327 493 423 159 743
217 623   3 399 853 407 103 983  89 463 290 516 212 462 350
960 376 682 962 300 780 486 502 912 800 250 346 172 812 350
870 456 192 162 593 473 915  45 989 873 823 965 425 329 803
973 965 905 919 133 673 665 235 509 613 673 815 165 992 326
322 148 972 962 286 255 941 541 265 323 925 281 601  95 973
445 721  11 525 473  65 511 164 138 672  18 428 154 448 848
414 456 310 312 798 104 566 520 302 248 694 976 430 392 198
184 829 373 181 631 101 969 613 840 740 778 458 284 760 390
821 461 843 513  17 901 711 993 293 157 274  94 192 156 574
 34 124   4 878 450 476 712 914 838 669 875 299 823 329 699
815 559 813 459 522 788 168 586 966 232 308 833 251 631 107
813 883 451 509 615  77 281 613 459 205 380 274 302  35 805
```

解题过程：

遇到一个复杂的问题，可以先尝试解决简单的情况，然后慢慢逼近最终的问题。

第一步： 先解决 5x5 矩阵的简单问题

首先能够想到的是递归的办法，5x5 的矩阵和可以转换成求 4x4 的矩阵和，递归下去就可以找到答案。

依次划掉第 0 列的 5 个元素，可以得到相应的剩余矩阵，可以看到剩余矩阵的维数变成 4x4，一直递推到 1x1。

行 \ 列	0	1	2	3	4
0	~~7~~	~~53~~	~~183~~	~~439~~	~~863~~
1	~~497~~	383	563	79	973
2	~~287~~	63	343	169	583
3	~~627~~	343	773	959	943
4	~~767~~	473	103	699	303

剩余矩阵

7 去掉(第0列, 第0行)

383	563	79	973
63	343	169	583
343	773	959	943
473	103	699	303

~~7~~	53	183	439	863
~~497~~	~~383~~	~~563~~	~~79~~	~~973~~
~~287~~	63	343	169	583
~~627~~	343	773	959	943
~~767~~	473	103	699	303

497 去掉(第0列, 第1行)

53	183	439	863
63	343	169	583
343	773	959	943
473	103	699	303

~~7~~	53	183	439	863
~~497~~	383	563	79	973
~~287~~	~~63~~	~~343~~	~~169~~	~~583~~
~~627~~	343	773	959	943
~~767~~	473	103	699	303

287 去掉(第0列, 第2行)

53	183	439	863
383	563	79	973
343	773	959	943
473	103	699	303

~~7~~	53	183	439	863
~~497~~	383	563	79	973
~~287~~	63	343	169	583
~~627~~	~~343~~	~~773~~	~~959~~	~~943~~
~~767~~	473	103	699	303

627 去掉(第0列, 第3行)

53	183	439	863
383	563	79	973
63	343	169	583
473	103	699	303

~~7~~	53	183	439	863
~~497~~	383	563	79	973
~~287~~	63	343	169	583
~~627~~	343	773	959	943
~~767~~	~~473~~	~~103~~	~~699~~	~~303~~

767 去掉(第0列, 第4行)

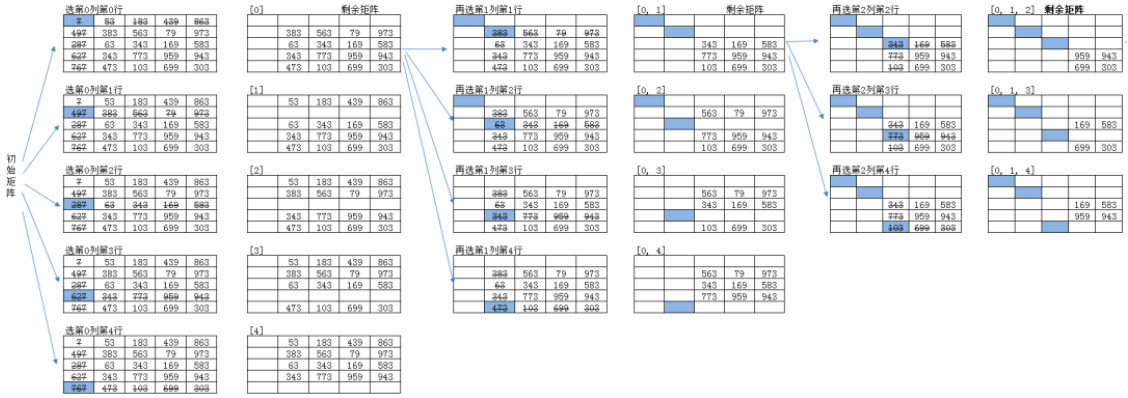
53	183	439	863
383	563	79	973
63	343	169	583
343	773	959	943

为了代码写起来方便，还有将来的性能考虑，用一维数组。

先求剩余矩阵。

```
fn residual_matrix(mat: &Vec<usize>, row:usize, col:usize) -> Vec<
usize> {
    let dim = (mat.len() as f64).sqrt() as usize;
    let mut m2 = vec![];
    for index in 0..mat.len() {
        let r = index / dim;
        let c = index % dim;
        if row != r && col != c {
```

```
                m2.push(mat[index]);
            }
        }
        m2
}
```



递归程序很容易写出来，运行后得到 3315，正确答案。

```
fn main() {
    let mat = vec![
        7, 53, 183, 439, 863, // row 0
        497, 383, 563, 79, 973, // row 1
        287, 63, 343, 169, 583, // row 2
        627, 343, 773, 959, 943, // row 3
        767, 473, 103, 699, 303, // row 4
    ];
    println!("{}", matrix_sum(&mat));
}

fn matrix_sum(mat: &Vec<usize>) -> usize {
    if mat.len() == 1 {
        return mat[0];
    }

    let dim = (mat.len() as f64).sqrt() as usize;
    let mut max_sum = 0;
    for row in 0..dim {
        let m2 = residual_matrix(&mat, row, 0);
        let temp = mat[row*dim] + matrix_sum(&m2);
        if temp > max_sum {
            max_sum = temp;
        }
    }
    max_sum
}
```

把 15x15 的矩阵换上，程序运行非常慢，打印了几个中间变量，发现这种算法用几个小时也算不出来。

第二步： 优化性能

15 层递归，中间有大量重复的计算，例如下面的两种情况，剩余矩阵是相同的，但会重复计算，类似的情况非常多，所以性能非常差。得对中间结果进行缓存处理。

~~7~~	~~53~~	~~183~~	~~439~~	~~863~~
~~497~~	~~383~~	~~563~~	~~79~~	~~973~~
~~287~~	~~63~~	343	169	583
~~627~~	~~343~~	773	959	943
~~767~~	~~473~~	103	699	303

~~7~~	~~53~~	~~183~~	~~439~~	~~863~~
~~497~~	~~383~~	~~563~~	~~79~~	~~973~~
~~287~~	~~63~~	343	169	583
~~627~~	~~343~~	773	959	943
~~767~~	~~473~~	103	699	303

首先想到一种缓存办法，生成一个与原始矩阵一样大小的矩阵，里面充满很大的随机数。每个元素也相应位置上的随机数进行异或运算，生成哈希值。这样会保证相同的矩阵生成的哈希值肯定是相同的，只要哈希表的容量足够大（0xFFFFFFFF），就不会有哈希冲突的情况发生。

```
fn main() {
    let mat = vec![
        7, 53, 183, 439, 863, 497, 383, 563, 79, 973, 287, 63, 34
3, 169, 583, 627, 343, 773, 959,
        943, 767, 473, 103, 699, 303, 957, 703, 583, 639, 913, 44
7, 283, 463, 29, 23, 487, 463,
        993, 119, 883, 327, 493, 423, 159, 743, 217, 623, 3, 399,
853, 407, 103, 983, 89, 463, 290,
        516, 212, 462, 350, 960, 376, 682, 962, 300, 780, 486, 50
2, 912, 800, 250, 346, 172, 812,
        350, 870, 456, 192, 162, 593, 473, 915, 45, 989, 873, 823,
 965, 425, 329, 803, 973, 965,
        905, 919, 133, 673, 665, 235, 509, 613, 673, 815, 165, 99
2, 326, 322, 148, 972, 962, 286,
        255, 941, 541, 265, 323, 925, 281, 601, 95, 973, 445, 721,
 11, 525, 473, 65, 511, 164, 138,
        672, 18, 428, 154, 448, 848, 414, 456, 310, 312, 798, 104,
 566, 520, 302, 248, 694, 976,
        430, 392, 198, 184, 829, 373, 181, 631, 101, 969, 613, 84
0, 740, 778, 458, 284, 760, 390,
        821, 461, 843, 513, 17, 901, 711, 993, 293, 157, 274, 94,
192, 156, 574, 34, 124, 4, 878,
        450, 476, 712, 914, 838, 669, 875, 299, 823, 329, 699, 81
```

```
5, 559, 813, 459, 522, 788, 168,
        586, 966, 232, 308, 833, 251, 631, 107, 813, 883, 451, 50
9, 615, 77, 281, 613, 459, 205,
        380, 274, 302, 35, 805,
    ];

    let mut hash_mat = vec![];
    for _i in 0..mat.len() {
        hash_mat.push(rand::random::<usize>());
    }
    //println!("{:?}", hash_mat );

    let mut cache = vec![0; 0xFFFFFFFF];
    println!("{}", matrix_sum(&mat, &hash_mat, &mut cache));
}

fn matrix_sum(mat: &Vec<usize>, hash_mat: &Vec<usize>, cache: &mut
 Vec<usize>) -> usize {
    if mat.len() == 1 {
        return mat[0];
    }

    let hash = hash_code(mat, hash_mat);
    if cache[hash] > 0 {
        return cache[hash]; // 缓存命中
    }

    let dim = (mat.len() as f64).sqrt() as usize;
    let mut max_sum = 0;
    for row in 0..dim {
        let m2 = residual_matrix(&mat, row, 0);
        let hash_mat2 = residual_matrix(&hash_mat, row, 0);
        let temp = mat[row * dim] + matrix_sum(&m2, &hash_mat2, ca
che);
        if temp > max_sum {
            max_sum = temp;
        }
    }
    cache[hash] = max_sum; // 写入缓存
    max_sum
}

// 剩余矩阵
fn residual_matrix(mat: &Vec<usize>, row: usize, col: usize) -> Ve
c<usize> {
    let dim = (mat.len() as f64).sqrt() as usize;
    let mut m2 = vec![];
    for index in 0..mat.len() {
        let r = index / dim;
        let c = index % dim;
        if row != r && col != c {
            m2.push(mat[index]);
        }
```

```
    }
    m2
}

fn hash_code(mat: &Vec<usize>, hash_mat: &Vec<usize>) -> usize {
    let mut hash = 0;
    for i in 0..mat.len() {
        hash ^= mat[i] ^ hash_mat[i];
    }
    hash & 0xFFFFFFFF
}
```

改进的程序速度很快，可以在 1、2 秒内得到结果。

第三步： 改进一下哈希算法

这里的哈希算法不太美观，引入了一个哈希表矩阵，为了不引起哈希冲突，需要较大内存。这个算法能够计算出最大的矩阵和，但具体选择了哪些位置上的元素？没有显示的答案。

这里引入一种简便的记号，[4, 1, 2, 3, 0]表示选中了(第 0 列, 第 4 行)、(1, 1)、(2, 2)、(3, 3)和(4, 0)这几个位置上的元素，这样用一维向量就可以记录选中的位置。

(x,y) (0, 4) (1, 1) (2, 2) (3, 3) (4, 0)

path: [4, 1, 2, 3, 0]

行 \ 列	0	1	2	3	4
0	7	53	183	439	863
1	497	383	563	79	973
2	287	63	343	169	583
3	627	343	773	959	943
4	767	473	103	699	303

如果某个位置被选中，可以用二进制 1 表示，这样哈希编码后很省空间，2^15 就够用。

```
fn hash_code(path: &Vec<usize>) -> usize {
    let mut hash = 0;
    for p in path {
        hash += 1_usize << p;
    }
    hash
}
```

当然，也可以写成一行

```
fn hash_code(path: &Vec<usize>) -> usize {
    path.iter().map(|x| 1_usize << x).sum()
}
```

最后的程序：

```
fn main() {
    let mat = vec![
        7, 53, 183, 439, 863, 497,
        /* … … */
        380, 274, 302, 35, 805,
    ];

    let mut cache = vec![0; 2_usize.pow(15)];
    let mut path = vec![];
    println!("{}", matrix_sum(&mut cache, &mat, &mut path));
}

fn matrix_sum(cache: &mut [usize], mat: &Vec<usize>, path: &mut Ve
c<usize>) -> usize {
    // 总列数
    let dim = (mat.len() as f64).sqrt() as usize;
    // 准备填充的列号
    let cur_col = path.len();
    if cur_col == dim {
        return 0;
    }

    let hash = hash_code(path);
    if cache[hash] > 0 {
        return cache[hash];
    }

    let mut max_sum = 0;
    for row in 0..dim {
        if !path.contains(&row) {
            path.push(row);
            let temp = mat[row * dim + cur_col] + matrix_sum(cach
e, &mat, path);
            if temp > max_sum {
                max_sum = temp;
            }
            path.pop();
        }
    }
    cache[hash] = max_sum;
    //println!("{:?} {} {}", &path, hash, max_sum);
    max_sum
}

fn hash_code(path: &Vec<usize>) -> usize {
    path.iter().map(|x| 1_usize<<x).sum()
```

```
}
```

这里能够得到最大矩阵和，但不知道最后选中了哪些元素，我想到了一个笨办法，由于缓存里有大量的中间结果，可以从第 0 列到第 n 列再计算一遍，把这条路径找出来。这个输出路径的函数不太美观，但能完成任务。

```
fn matrix_output(cache: &Vec<usize>, mat: &Vec<usize>) {
    let dim = (mat.len() as f64).sqrt() as usize;
    let mut path = vec![];
    for col in 0..dim {
        let mut selected_row = 999;
        let mut max = 0;
        for row in 0..dim {
            if !path.contains(&row) {
                path.push(row);
                let hash = hash_code(&path);
                if mat[row * dim + col] + cache[hash] > max {
                    max = mat[row * dim + col] + cache[hash];
                    selected_row = row;
                }
                path.pop();
            }
        }
        path.push(selected_row);
    }
    println!("path: {:?}", path);
}
```

第四步：遗传算法

除了暴力的递归算法外，还有其它算法，比如遗传算法，每次生成一定数量的后代，对矩阵上相应元素求和，和较大的属于优质后代，淘汰掉那些矩阵和较小的，不断迭代之后，就可以得到最优结果。

任意交换两个元素就可以产生一种选择元素的方法。

[0, 1, 2, 3, 4]

7	53	183	439	863
497	383	563	79	973
287	63	343	169	583
627	343	773	959	943
767	473	103	699	303

[0, 1, 3, 2, 4]

7	53	183	439	863
497	383	563	79	973
287	63	343	169	583
627	343	773	959	943
767	473	103	699	303

算法并不复杂，不需严格排序，将矩阵和最大的放在队列的最前面（用 insert 语句），其它的扔在后面就行（用 push 语句），是金子早晚会发光的，经过几代淘汰，优质的后代会跑到队列的前面。

```
fn main() {
    let mat = vec![
        7, 53, 183, 439, 863,
        /*  … … … …*/
        380, 274, 302, 35, 805,
    ];

    let reproduce_num = 7;
    let mut paths = vec![vec![0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14]];
    let mut max = 0;
    for _generation in 0..20000 {
        for _i in 0..reproduce_num * reproduce_num {
            let path = &paths[rand::random::<usize>() % paths.len
()];
            let path_new = swap(
                &path,
                rand::random::<usize>() % 15,
                rand::random::<usize>() % 15,
            );
            if !paths.contains(&path_new) {
                let ev = eval(&mat, &path_new);
                if ev > max {
                    max = ev;
                    paths.insert(0, path_new);
                } else {
```

```
                    paths.push(path_new);
                }
            }
        }
        // 只保留前面几个优质的
        if paths.len() > reproduce_num {
            for p in (reproduce_num..paths.len()).rev() {
                paths.remove(p);
            }
        }
    }
    println!("{} {:?}", max, paths[0]);
}

fn swap(path: &Vec<usize>, i: usize, j: usize) -> Vec<usize> {
    let mut path2 = path.clone();
    path2[i] = path[j];
    path2[j] = path[i];
    return path2;
}

fn eval(mat: &Vec<usize>, path: &Vec<usize>) -> usize {
    let dim = (mat.len() as f64).sqrt() as usize;
    let mut sum = 0;
    for (col, row) in path.iter().enumerate() {
        let index = row * dim + col;
        sum += mat[index];
    }
    sum
}
```

算法中繁殖系数用了 7，尽量随机生成下一代，繁衍 2 万次之后，99%的情况下都会得到正确答案，提高繁衍次数会 99.99%的得到正确答案。

遗传算法的缺点是迭代次数不够的时候，算法就不太稳定，可能无法得到最优解，只能得到局部较优解。但这种算法也有优点，可以进行规模化，当矩阵更大时，算法仍然有效。只需考虑如何让后代有一定概率的变异性，后代更优异。

第五步： 更复杂的算法

这类问题还可以用动态规划，匈牙利算法等，算法比较复杂，感兴趣的朋友自行探索吧。

第 346 题 强循环单位数

7 是一个特别的数，因为 7 在 2 进制下写作 111，在 6 进制下写作 11（也即 $7_{10} = 11_6 = 111_2$）。换句话说，7 在至少两种 b > 1 进制下为循环单位数。

我们称拥有这种性质的正整数为强循环单位数。可以验证，有 8 个小于 50 的强循环单位数：$\{1, 7, 13, 15, 21, 31, 40, 43\}$。

进一步地，所有小于 1000 的强循环单位数之和是 15864。

求所有小于 10^{12} 的强循环单位数之和。

解题过程：

尝试把 50 以内的强循环数列出来，找找规律。

n=2 这一行非常特殊，里面包括了从 3 开始向后的所有自然数。问题转化为求第 3 行之后小于 50 的各个数的和。

		进制b = 2	3	4	5	6	7	8
1	n=1	1	1	1	1	1	1	1
11	n=2	3	4	5	6	7	8	9
111	n=3	7	13	21	31	43	57	73
1111	n=4	15	40	85	156	259	400	585
11111	n=5	31	121	341	781	1555	2801	4681
111111	n=6	63	364	1365	3906	9331	19608	37449
1111111	n=7	127	1093	5461	19531	55987	137257	299593

手算这张表比较麻烦，可以写个小程序打印这张表。

```
for n in 1..14 {
    print!("n={}", n);
    for b in 2_u64..10 {
        let t = (b.pow(n) - 1) / (b - 1);
        print!("{:15}", t);
    }
    println!();
}
```

对于 b 进制的第 n 行的数值可以推导出公式

$$1 + b + b^2 + b^3 + \ldots + b^{n-1} = \frac{b^n - 1}{b - 1}$$

前 n 行之和：

$$\frac{b-1}{b-1}+\frac{b^2-1}{b-1}+\frac{b^3-1}{b-1}+\ldots+\frac{b^n-1}{b-1}$$
$$=\frac{b+b^2+b^3+\ldots+b^n-n}{b-1}$$
$$=\frac{b\left(1+b+b^2+\ldots+b^{n-1}\right)-n}{b-1}$$
$$=\frac{b\cdot\frac{b^n-1}{b-1}-n}{b-1}$$

$$\frac{b^n-1}{b-1}<\text{limit}$$
$$b^n-1<\text{limit}\cdot(b-1)$$
$$b^n<1+\text{limit}\cdot(b-1)$$
$$\log\left(b^n\right)<\log\left(1+\text{limit}\cdot(b-1)\right)$$
$$n<\frac{\log\left(1+\text{limit}\cdot(b-1)\right)}{\log(b)}$$

		进制b = 2	3	4	5	6	7	8	9	10		90		
1	n=1	1	1	1	1	1	1	1	1	1		1	1	(b-1)/(b-1)
11	n=2	3	4	5	6	7	8	9	10	11		91	1+b	(b^2-1)/(b-1)
111	n=3	7	13	21	31	43	57	73	91	111		8191	1+b+b^2	(b^3-1)/(b-1)
1111	n=4	15	40	85	156	###	###	###	###	1111			1+b+b^2+b^3	(b^4-1)/(b-1)
11111	n=5	31	121	341	781	###	###	###	###	11111				(b^5-1)/(b-1)
111111	n=6	63	364	1365	3906	###	###	###	###	111111				
1111111	n=7	127	1093	5461	19531	###	###	###	###	1111111				
	n=8	255	3280	21845	97656	###	###	###	###	11111111				
	n=9	511	9841	87381	488281	###	###	###	###	111111111				
	n=10	1023	29524	349525	2441406	###	###	###	###	1111111111				
	n=11	2047	88573	1398101	12207031	###	###	###	###	11111111111				
	n=12	4095	265720	5592405	61035156	###	###	###	###	111111111111				
	n=13	8191	797161	22369621	305175781	###	###	###	###	1111111111111				(b^n-1)/(b-1)

第四篇 难度为 20%

这一部分是 20%的难度。

第 60 题 素数对的集合

3、7、109 和 673 是非常特别的一组素数。任取其中的两个并且以任意顺序连接起来，其结果仍然是个素数。例如，选择 7 和 109，我们得到 7109 和 1097 均为素数。这四个素数的和是 792，这是满足这个性质的一组四个素数的最小和。

若有一组五个素数，任取其中的两个并且以任意顺序连接起来，其结果仍然是个素数，求这样一组素数的最小和。

解题过程：

题目中已经知道有 4 个素数，3、7、109 和 673，可以随意组合仍是素数，先假设就是这个 4 个，再找第 5 个素数，仍满足条件。

第一步： 假设已经 4 个，找第 5 个

这个代码比较容易写。

```
fn main() {
    let mut pset = PrimeSet::new();
    for p in pset.iter() {
        if is_concat_primes(p, 3) && is_concat_primes(3, p)
            && is_concat_primes(p, 7) && is_concat_primes(7, p)
            && is_concat_primes(p, 109) && is_concat_primes(109, p)
            && is_concat_primes(p, 673) && is_concat_primes(673, p)
        {
            println!("{}", p);
            break;
        }
    }
}

// concating prime a and prime b, the result is prime?
// for example, 7, 109 => 7109 is prime, so return true
fn is_concat_primes(a: u64, b: u64) -> bool {
    let c = format!("{}{}", a, b);
    let c = c.parse::<u64>().unwrap();
    primes::is_prime(c)
}
```

程序运行很长时间没有结果，最后还真能找到一组答案：129976621，但明显这个答案有点太大了，应该错过了更小的 5 个素数组合。

第二步： 暴力搜索

准备一个列表，有 5 个空位，逐个向里面添加满足条件的素数，为了限定搜索

范围，先假定 5 个素数都不超过 10000。

假设我们已经找到了三个素数：[191, 227, 281]，第四个素数的搜索范围为 282..10000，写五重循环就可以完成任务，代码不太好看，但可以快速检验算法。

主程序：

```
let max = 10000;
for p1 in prime_pairs(max, &[]) {
    for p2 in prime_pairs(max, &p1) {
        for p3 in prime_pairs(max, &p2) {
            for p4 in prime_pairs(max, &p3) {
                let p5 = prime_pairs(max, &p4);
                if !p5.is_empty() {
                    println!("{:?}", p5[0]);
                    println!("{:?}", p5[0].iter().sum::<u64>());
                }
            }
        }
    }
}
```

另外两个辅助函数：

```
// try to append a prime to list_primes, don't exceed max value
fn prime_pairs(max: u64, list_primes: &[u64]) -> Vec<Vec<u64>> {
    let mut pairs = vec![];

    let mut pset = PrimeSet::new();
    for p in pset.iter().take_while(|&x| x <= max) {
        if list_primes.is_empty() {
            pairs.push(vec![p]);
            continue;
        }
        if p > *list_primes.last().unwrap() && all_primes(list_primes, p) {
            let mut v = list_primes.to_vec();
            v.push(p);
            pairs.push(v);
        }
    }
    pairs
}

// each prime in list concating with b, are prime
fn all_primes(primes_list: &[u64], b: u64) -> bool {
    for &a in primes_list.iter() {
        if !is_concat_primes(a, b) || !is_concat_primes(b, a) {
            return false;
        }
    }
    true
}
```

没想到程序竟然很快计算出了结果，还是正确答案。实际上还需要更多的计算，

来验证它们就是和最小的一组素数。

第三步： 优化

看论坛精华帖子，发现有一个思路非常不错，利用了数字和整除 3 的性质，把素数分为两组，可以减少搜索范围，提高速度。

一个素数的所有数字和除以 3 之后只能出现两种情况，余数或者为 1，或者为 2，可以分为两组 prime1，prime2。这样 5 个素数肯定全部在 prime1 里，或者全在 prime2 里。

第四步： 更专业的解法

原来这种问题称为复杂网络中的 k-clique 问题。clique 在复杂网络中翻译为“派系”，在数据结构中翻译为“团”。刚才的问题就是求 5-clique 的问题。有大量的文章讨论这种算法。Python 语言里有一个 NetworkX 软件包专门处理这种问题，但在 Rust 里还没有找到相关类库。

可以参考的几篇文章：

https://www.cnblogs.com/singlee/archive/2012/02/25/2765023.html

https://en.wikipedia.org/wiki/Clique_problem

https://medium.com/100-days-of-algorithms/day-64-k-clique-c03fdc565b1e

https://iq.opengenus.org/algorithm-to-find-cliques-of-a-given-size-k/

第 61 题 循环的多边形数

三角形数、正方形数、五边形数、六边形数、七边形数和八边形数统称为多边形数。它们分别由如下的公式给出：

三角形数	P3,n=n(n+1)/2	1, 3, 6, 10, 15, ...
正方形数	P4,n=n2	1, 4, 9, 16, 25, ...
五边形数	P5,n=n(3n−1)/2	1, 5, 12, 22, 35, ...

六边形数	P6,n=n(2n−1)	1, 6, 15, 28, 45, ...
七边形数	P7,n=n(5n−3)/2	1, 7, 18, 34, 55, ...
八边形数	P8,n=n(3n−2)	1, 8, 21, 40, 65, ...

由三个 4 位数 8128、2882、8281 构成的有序集有如下三个有趣的性质。

✧ 这个集合是循环的，每个数的后两位是后一个数的前两位（最后一个数的后两位也是第一个数的前两位）。

✧ 每种多边形数——三角形数（P3,127=8128）、正方形数（P4,91=8281）和五边形数（P5,44=2882）——在其中各有一个代表。

✧ 这是唯一一组满足上述性质的 4 位数有序集。

存在唯一一个包含六个 4 位数的有序循环集，每种多边形数——三角形数、正方形数、五边形数、六边形数、七边形数和八边形数——在其中各有一个代表。求这个集合的元素和。

解题过程：

把复杂的问题分解为一个一个的简单问题。

第一步： 先把所有四位数求出来

利用 map()、filter()和 take_while()等函数，可以用一行语句将数组生成。

```
let p3: Vec<u32> = (1..)
    .map(|n| n * (n + 1) / 2)
    .filter(|&n| n > 1000)
    .take_while(|&n| n <= 9999)
    .collect();

let p4: Vec<u32> = (1..)
    .map(|n| n * n)
    .filter(|&n| n > 1000)
    .take_while(|&n| n <= 9999)
    .collect();
```

如果这样生成六种多边形数，会有大量的重复代码，可以闭包作为函数的参数，统一写一个函数，这样代码非常简练。

```
let p3 = gen_poly_numbers(1000, 9999, |n| n * (n + 1) / 2);
let p4 = gen_poly_numbers(1000, 9999, |n| n * n);
let p5 = gen_poly_numbers(1000, 9999, |n| n * (3 * n - 1) / 2);
let p6 = gen_poly_numbers(1000, 9999, |n| n * (2 * n - 1));
let p7 = gen_poly_numbers(1000, 9999, |n| n * (5 * n - 3) / 2);
let p8 = gen_poly_numbers(1000, 9999, |n| n * (3 * n - 2));


fn gen_poly_numbers(start: u32, end: u32, f: fn(u32) -> u32) -> Vec<u32> {
    (1..).map(f)
         .filter(|&n| n >= start)
         .take_while(|&n| n <= end)
         .collect()
}
```

这里的一个语法知识点是 f: fn(u32) -> u32 的写法，把一个函数作为参数传递给另一个函数。

第二步： 递归算法

现在仍不要直接奔向最后的问题，先从简单的情况入手，用题目中给出的三个多边形数[8128, 8281, 2882]，实现一个递归算法，验证算法的正确性。

算法描述：

1）从 p3 中的取一个数 for n in p3

2）假设这个数是 8128，记录到 found 数组中，我们下一步的任务是从[p4, p5]数组中寻找 28 开头，81 结尾的两个数，伪代码：find([p4, p5], 28, 81, [8128])

3）先在 p4 中找，再在 p5 中找 for cur_list in lists

4）在数组中寻找以 28 开头的数 for n in cur_list

5）假设是在 p5 中找到了 2882，此时 found 数组变为[8128, 2882]，递归调用，在[p4]中查找以 82 开头，81 结尾的一个数 find([p4], 82, 81, [8128, 2882])

5.1）在 p4 里会找到一个以 82 开头的数，即 8281，更新 found，再调用 find([], 81, 81 [8128, 2882, 8281]

5.2）从这里可以发现递归函数的退出机制，待查的数组已经为空，要找的开头与结尾正好相等。

递归函数的源代码：

```rust
fn find(lists: &[&[u32]], start: u32, end: u32, found: &[u32])  {
    if lists.is_empty() && start == end {
        let result = found.iter().sum::<u32>();
        println!("{:?} {}", found, result);
    }

    for (i, &cur_list) in lists.iter().enumerate() {
        for &n in cur_list {
            if head(n) == start {
                let mut lists_copy = lists.to_vec();
                lists_copy.remove(i);
                let mut found_copy = found.to_vec();
                found_copy.push(n);
                find(&lists_copy, tail(n), end, &found_copy);
            }
        }
    }
}
```

lists.copy.remove(i)的含义，假设当前搜索的数组为[p4, p5, p6, p7, p8]，如果在 p5 中找到了满足条件的数，后续的搜索范围将是[p4, p6, p7, p8]，这里的 i 记住数组中的位置，remove(i)将删除 p5。

主程序中的代码：

```rust
for n in p3 {
    find(&[&p4, &p5], n % 100, n / 100, &[n]);
}
```

第三步：解决最终的问题

搜索 3 个数的代码测试通过后，再搜索 6 个数。

```rust
for n in p3 {
    find(&[&p4, &p5, &p6, &p7, &p8], n % 100, n / 100, &[n]);
}
```

由于 p8 中的数据元素较少，先搜索 p8 可能会稍快一点。

第 323 题 随机整数按位或运算

记 y_0、y_1、y_2、……是一系列 32 位的无符号随机整数，也就是说，$0 \le y_i < 2^{32}$，其中每个数等概率出现。

序列 x_i 按如下方式递归定义：

$x_0 = 0$

对于所有 $i > 0$，$x_i = x_{i-1} | y_{i-1}$（|指按位或运算符）

可以看出，存在下标 N，使得对于所有 $i \ge N$，$x_i = 2^{32}-1$（所有比特位均为 1）。

求 N 的期望值。将你的答案保留小数点后十位小数。

理解题意，一个数初始值为 0，不断与一个随机数进行‘或’运算，直到 32 个二进制位都为 1，所需要步数记为 N。进行无数数这样的试验，问 N 的平均值是多少。

第一步：

先用蒙特卡罗模拟方法试验一下，看看答案的大概范围。

```
fn trial() -> u32 {
    let mut x = 0_u32;
    for i in 1.. {
        x |= rand::random::<u32>();
        if x == 0xFFFFFFFF {
            return i;
        }
    }
    0
}
```

运行这个 trail()函数 10 万次，取平均值，可以得到答案在 6.35 附近。

第二步：

原题的 x 是 32 位整数，可以从最简单的情况入手，假设 x 为 1 位的整数。

记此时的期望值为 E(1)。

进行第一步操作之后，有两种可能：

第一种情况，1/2 的概率，直接变成了 1，即一步完成

第二种情况，1/2 的概率，仍是 0，回到了初始情况，即需要 1+E(1)步完成。

这时有：E(1) = 1/2 + 1/2 * (1 + E(1))

可以解出：E(1) = 2

第三步：

假设 x 是 2 位整数，此时的期望值记为 E(2)。

进行第一步操作之后，有三种可能：

第一种情况，1/4 的概率，直接变成了$(11)_2$，即一步完成

第二种情况，2/4 的概率，有一位变为 1，即$(01)_2$或$(10)_2$,这时是 E(1)时的情况，期望值为 1+E(1)。

第三种情况，1/4 的概率，仍为 0，这时是 1+E(2)。

列出式子，E(2) = 1/4 + 2/4 * (1+ E(1)) + 1/4 * (1 + E(2))

3/4 * E(2) = 8/4

E(2) = 8/3

第四步：

这时，你可能还没有总结出规律，再来一次，假设 x 是 3 位整数，此时的期望值记为 E(3)。

进行第一步操作之后，有四种可能：

第一种情况，1/8 的概率，直接变成了$(111)_2$，即一步完成

第二种情况，3/8 的概率，有两位变为 1，即$(011)_2$或$(101)_2$或$(110)_2$,这时是 E(1)时的情况，期望值为 1+E(1)。

第三种情况，3/8 的概率，有一位变为 1，即$(001)_2$或$(010)_2$或$(100)_2$,这时是 E(2)时的情况，期望值为 1+E(2)。

第四种情况，仍为 0，这时是 1+E(3)。

列出式子，E(3) = 1/8 + 3/8 * (1+ E(1)) + 3/8 * (1 + E(2)) + 1/8 * (1 + E(3))

7/8 * E(3) = 1/8 + 3/8 * (1+ E(1)) + 3/8 * (1 + E(2)) + 1/8

7/8 * E(3) = 1/8 + 3/8 * 3 + 3/8 * 11/3 + 1/8

E(3) = 22/7

第五步，现在可以总结出最一般的规律：

E(N) = 1/2^N + C(N,1)/2^N * (1+E(1)) + C(N,2)/2^N * (1+E(2)) + ... + C(N,N-1)/2^N * (1+E(N-1)) + 1/2^N (1+E(N))

这里的 C(N, i)是排列组合数，从 N 个数里取 i 个数的组合数。

耐心地把右侧的 E(N)挪到左侧：

(2^N-1)/2^N * E(N) = 1/2^N + C(N,1)/2^N * (1+E(1)) + C(N,2)/2^N * (1+E(2)) + ... + C(N,N-1)/2^N * (1+E(N-1)) + 1/2^N

最后可以得到：

E(N) = [1 + C(N,1) * (1+E(1)) + C(N,2) * (1+E(2)) + ... + C(N,N-1) * (1+E(N-1)) + 1] / (2^N-1)

我们知道 E(1)=2，不断套用上面的公式，就可以得到最终的答案。

求组合数的函数：

```
fn comb(n: usize, r: usize) -> f64 {
    let mut c = 1_f64;
    for i in 0..r {
        c *= ((n - i) as f64) / (r - i) as f64;
    }
    c
}
```

主程序：

```
let mut e = vec![0.0, 2.0];
for n in 2..=32 {
    let mut f = 1.0 as f64;
    for r in 1..=n {
```

```
            f += comb(n, r) * (1.0 + e[n - r]);
        }
        f = f / (2_f64.powf(n as f64) - 1.0);
        e.push(f);
        println!("E({}) = {:.10}", n, f);
    }
```

第五篇　500 题之后的一些题

第 500 题 500!

120 的因子个数为 16，事实上 120 是最小的有 16 个因子的数。

找出最小的有 2^{500500} 个因子的数，给出这个数除以 500500507 的余数。

解题过程：

直接看最终的问题，2^500500 是个天文数字，肯定不能用蛮力。遇到一个复杂的问题，可以先尝试解决简单的情况，然后慢慢逼近最终的问题。

第一步： 从简单的情况入手找规律：

第 650 题里解决过因子个数的公式，还可以计算出所有因子之和。

```rust
fn min_number_has_factors(x: u64) -> u64 {
    for n in 2.. {
        let groups = factors_group(n);
        let factors_num = groups.iter().map(|(_, x)| x + 1).product::<u64>();
        if factors_num == x {
            println!("{}, divisors num: {}", n, factors_num);
            print_factors_group(groups);
            return n;
        }
    }
    0
}

// 如果一个数有这些因子：[2, 2, 3, 3, 3, 3, 5, 7]
// 则得到：[(2,2), (3,4), (5,1), (7,1)]
fn factors_group(n: u64) -> Vec<(u64, u64)> {
    let factors = primes::factors(n);
    let groups = factors
```

```
        .iter()
        .group_by(|e| **e)
        .into_iter()
        .map(|(k, group)| (k, group.count() as u64))
        .collect::<Vec<(u64, u64)>>();
    groups
}

fn print_factors_group(groups: Vec<(u64, u64)>) {
    println!(
        "{}",
        &groups
            .iter()
            .map(|(k, v)| k.to_string() + &"^" + &v.to_string())
            .join(" * ")
    );
    println!(
        "divisors num:  {}",
        &groups
            .iter()
            .map(|(_, v)| "(".to_string() + &v.to_string() + &"+1)")
            .join(" * ")
    );
}
```

现在先尝试计算几个，慢慢寻找规律。

```
min_number_has_factors(4); // 2^2
min_number_has_factors(8); // 2^3
min_number_has_factors(16); // 2^4
min_number_has_factors(32); // 2^5
min_number_has_factors(64); // 2^6
min_number_has_factors(128); // 2^7
min_number_has_factors(256); // 2^8
```

结果有：

```
6 = 2^1 * 3^1
因子个数 4=  (1+1) * (1+1)

24 = 2^3 * 3^1
因子个数 8 = (3+1) * (1+1)

120 = 2^3 * 3^1 * 5^1
因子个数 16 =  (3+1) * (1+1) * (1+1)

840 = 2^3 * 3^1 * 5^1 * 7^1
因子个数 32 = (3+1) * (1+1) * (1+1) * (1+1)

7560 = 2^3 * 3^3 * 5^1 * 7^1
因子个数 64 = (3+1) * (3+1) * (1+1) * (1+1)

83160 = 2^3 * 3^3 * 5^1 * 7^1 * 11^1
因子个数 128 = (3+1) * (3+1) * (1+1) * (1+1) * (1+1)
```

```
1081080 = 2^3 * 3^3 * 5^1 * 7^1 * 11^1 * 13^1
因子个数 256 = (3+1) * (3+1) * (1+1) * (1+1) * (1+1) * (1+1)
```

第二步： 努力寻找规律

通过分析几个简单的特例，将一般性的公式推导出来，需要运用基础的数学知识。

一个数 n 可以分解成如下形式，其中 pi 为素数因子。

$$p_0^{a_0} * p_1^{a_1} * \ldots * p_i^{a_i}$$

那么，它的因子个数为：

$$(a_0 + 1) * (a_1 + 1) * \ldots * (a_i + 1)$$

最终的因子个数可以表示为 2^{500500} 形式，令：

$$a_i = 2^{b_i} - 1$$

则有：

$$\begin{gathered} n = p_0^{2^{b_0}-1} * p_1^{2^{b_1}-1} * \ldots * p_i^{2^{b_i}-1} \\ 2^{b_0} * 2^{b_1} * \ldots * 2^{b_i} = 2^{500500} \\ b_0 + b_1 + \ldots + b_i = 500500 \end{gathered}$$

最终的结果要让[b0, b1, b2...bi]的和为 500500。现在来看一下这个数组是如何变化的，找出递推的规律。

$$\begin{gathered} 120 = 2^3 * 3^1 * 5^1 \\ f(120) = 16 = (3+1) * (1+1) * (1+1) \\ = 2^2 * 2^1 * 2^1 \\ \text{也就是说} b_0, b_1, b_2 = [2, 1, 1] \\ 840 = 2^3 * 3^1 * 5^1 * 7^1 \\ f(840) = 32 = (3+1) * (1+1) * (1+1) * (1+1) \\ = 2^2 * 2^1 * 2^1 * 2^1 \\ \text{也就是说} b_0, b_1, b_2, b_3 = [2, 1, 1, 1] \end{gathered}$$

```
因子个数 2 =  (2^1)
[b0] = [1]

因子个数 4 =  (2^1) * (2^1)
[b0,b1] = [1,1]

因子个数 8 = (2^2) * (2^1)
[b0,b1] = [2,1]

因子个数 16 =  (2^2) * (2^1) * (2^1)
[b0,b1,b2] = [2,1,1]

因子个数 32 = (2^2) * (2^1) * (2^1) * (2^1)
[b0,b1,b2] = [2,2,1]

因子个数 64 = (2^2) * (2^2) * (2^1) * (2^1)
[b0,b1,b2,b3] = [2,2,1,1]

因子个数 128 = (2^2) * (2^2) * (2^1) * (2^1) * (2^1)
[b0,b1,b2,b3,b4] = [2,2,1,1,1]

因子个数 256 =  (2^2) * (2^2) * (2^1) * (2^1) * (2^1) * (2^1)
[b0,b1,b2,b3,b4,b5] = [2,2,1,1,1,1]
```

这里需要足够的耐心，这个 bi 数组或者在末尾增加一个元素 1，或者在前面的某个位置上数值增 1。

$$\text{要使其最小}\quad n = p_0^{2^{b_0}-1} * p_1^{2^{b_1}-1} * \ldots * p_i^{2^{b_i}-1}$$

直接比较大小容易溢出，取对数：

$$(2^{b_0}-1) * log(p_0) + (2^{b_1}-1) * log(p_1) + \ldots + (2^{b_i}-1) * log(p_i)$$

如果其中的某一项增 1，则数值增加：

$$2^{b_i} * log(p_i)$$

如果尾部增加一项，数值增加：

$$log(p_{i+1})$$

上面的数值中，哪一项更小，则表示或者在尾部增加一个，或者原数组中的数值增 1。

最后的代码：

```rust
fn p500(n: usize) -> u64 {
```

```
    let mut pset = PrimeSet::new();
    let primes: Vec<_> = pset.iter().take(n).collect();
    let primes_log: Vec<_> = primes.iter().map(|x| (*x as f64).log10()).collect();

    let mut b = vec![1];
    for _i in 2..=n {
        let mut min = primes_log[b.len()];
        let mut pos = b.len(); // 默认尾部增加一个
        for j in 0..b.len() {
            let temp = 2_f64.powf(b[j] as f64) * primes_log[j];
            if temp < min {
                pos = j;
                min = temp;
            }
            if b[j] == 1 {
                break; // 后面的都不用判断了
            }
        }
        if pos == b.len() {
            b.push(1);
        } else {
            b[pos] += 1;
        }
    }

    let mut result = 1_u64;
    for i in 0..b.len() {
        let exp = 2_u32.pow(b[i]) - 1;
        for _j in 0..exp {
            result = result * primes[i] % 500500507;
        }
    }
    result
}
```

第 622 题 完美洗牌

有这样一种洗牌：将牌张平分为两份，左手拿上半部分牌张，右手拿下半部分牌张，然后，将右手的牌严格地交叉到左手的牌张中，也就是右手的第 1 张牌处于左手的第 1 张牌后面，右手的第 2 张牌处于左手的第 2 张牌后面，依次类推。（注意，这种洗牌法不会改变顶底的两张牌）

记 s(n) 为使牌恢复原状的最少连续洗牌次数，这里的 n 为偶数。

令人惊奇的是，52 张的标准扑克牌只需 8 次洗牌就可以恢复原状，因此有：s(52) = 8，同样可以验证，86 张的扑克牌也只需 8 次洗牌恢复原状，将所有满足 s(n)=8 的 n 求和可以得到 412。

求满足 s(n)=60 的所有 n 的和。

解题过程：

遇到一个复杂的问题，可以先尝试解决简单的情况，然后慢慢逼近最终的问题。

第一步： 从简单的情况入手找规律：

将牌从 0 开始编号，模拟洗牌的过程，看重复几次能够恢复原状。

```rust
#[test]
fn test_shuffle_52_cards()
{
    assert_eq!(8, shuffle_times(52));
}

#[test]
fn test_shuffle_86_cards()
{
    assert_eq!(8, shuffle_times(86));
}

fn shuffle_times(deck_size: u32) -> u32 {
    let cards: Vec<u32> = (0..deck_size).collect();
    let mut shuffled = perfect_shuffle(&cards);
    let mut count = 1;
    while cards.to_vec() != shuffled {
        //println!("{:?}", shuffled);
        shuffled = perfect_shuffle(&shuffled);
        count += 1;
    }
    count
}

fn perfect_shuffle(cards: &[u32]) -> Vec<u32> {
    let half = cards.len() / 2;
    let mut shuffled = Vec::with_capacity(cards.len());
    for i in 0..half {
        shuffled.push(cards[i]);
        shuffled.push(cards[half + i]);
    }
    shuffled
    /* 另一种装逼的写法
    let c1 = &cards[0..half];
    let c2 = &cards[half..];
    let z: Vec<(_, _)> = c1.iter().zip(c2.iter()).map(|(&a, &b)| (a, b)).collec
t();
    z.iter().flat_map(|tup| vec![tup.0, tup.1]).collect()
    */
}
```

现在可以求出满足所有 s(n)=8 的数值和，但非常慢，计算到 1 万都非常吃力。

```rust
fn sum_s(n: u64) -> u64 {
    let mut sum = 0_u64;
    for i in (4..999).step_by(2) {
        let t = shuffle_times(i) as u64;
        if t == n {
            sum += i as u64;
            println!("s({}) = {}", i, t)
        }
    }
    sum
}
```

第二步： 优化

打印出洗牌过程中各个牌的位置变化，找一下变化规律，发现只需跟踪数字 1 的位置变化即可，一开始它处于第 1 号位置（从 0 开始），经过一次洗牌后，它变到位置 2，再洗，变为位置 4，依次推导下去，再回到位置 1 时就表示恢复原状。

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ..., 51]
[0, 26, 1, 27, 2, 28, 3, 29, 4,..., 51]
[0, 13, 26, 39, 1, 14, 27, 40, 2, ..., 51]
[0, 32, 13, 45, 26, 7, 39, 20, 1, 33, ..., 51]
...
[0, 2, 4, ... , 50, 1, 3, 5, ..., 51]
```

修改算法，不需要构建复杂的数组，只需要整数运算，速度快了许多倍，可以轻松计算到几十万的情况，但还是远远不够。

```rust
fn shuffle_pos(deck_size: u64, i: u64) -> u64 {
    if i < deck_size / 2 {
        i * 2
    } else {
        i * 2 - deck_size + 1
    }
}

fn shuffle_times(deck_size: u64) -> u32 {
    let mut pos = 1;
    pos = shuffle_pos(deck_size, pos);
    let mut count = 1;
    while pos != 1 {
        //print!("{} ", pos);
        pos = shuffle_pos(deck_size, pos);
        count += 1;
    }
    //println!();
    count
}
```

等于 60 的情况非常非常多，而且看不到尽头，还得改进算法。

```
s(755756) = 60
s(769576) = 60
s(772366) = 60
s(772786) = 60
s(775776) = 60
s(787450) = 60
s(787816) = 60
s(811240) = 60
...
```

第三步：再优化

后面我又想了一些优化办法，但陷入了一个思维误区，那个 shuffle_pos()函数可以变换成一个简单的数学表达式。可以看这篇文章：

http://goatleaps.xyz/euler/maths/Project-Euler-622.html

对于 n 张牌的情况，连续洗牌 i 次后，1 的位置为 $2^i \bmod (n-1)$。如果连续洗牌 60 次恢复原状，就有：

$2^{60} \bmod (n-1) = 1$

也就是说$(2^{60} - 1)$肯定能够被(n-1)整除，也就是说(n-1)肯定是$(2^{60} - 1)$的因子，一下子，满足条件的 n 范围限制在$(2^{60} - 1)$的所有因子中。

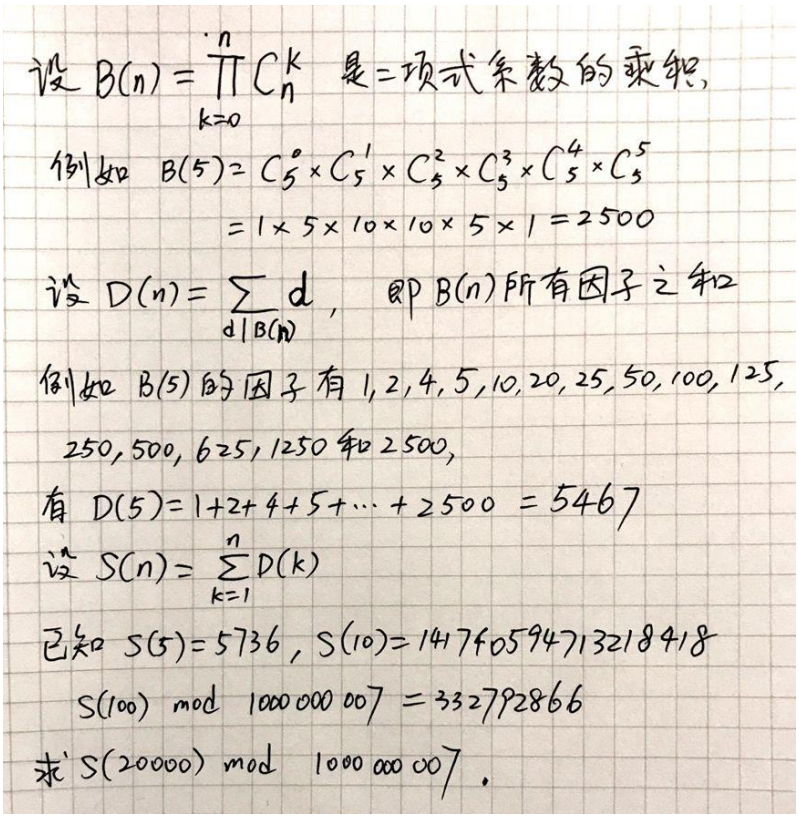
现在，只需要找到$(2^{60} - 1)$的所有素数因子，再排列组合出所有因子，一个个检查是否需要 60 次洗牌复原，可以得到最终结果。

运行时间小于 1 秒！

```rust
fn sum_s(n: u32) ->u64 {
    let p = 2_u64.pow(n) - 1;
    let factors = primes::factors(p);
    // 2 ^ 64 - 1 的因子:  [3, 3, 5, 5, 7, 11, 13, 31, 41, 61, 151, 331, 1321]

    let mut v = vec![];
    // 用了一种笨办法实现所有的排列组合
    // 二进制位为1表示含这个因子，0表示不包含
    for i in 1..=2_u64.pow(factors.len() as u32) {
        let s = format!("{:0width$b}", i, width = factors.len());
        let prod = s
            .chars()
            .enumerate()
            .map(|(index, x)| { // 某个位为1，则取该位对应的素数因子
                if x.to_digit(10).unwrap() == 1 {
                    factors[index]
                } else {
                    1
                }
            })
```

```
            .fold(1_u64, |p, a| p * a as u64) //求乘积
            + 1; // (n-1)肯定是(2 ^ 60 - 1)的因子，别忘了+1
        if !v.contains(&prod) {
            v.push(prod);
        }
    }
    //v.sort();
    //println!("{:?}",v );

    v.iter().filter(|&x| shuffle_times(*x) == n).sum::<u64>()
}
```

第 650 题 二项式系数的因子



解题过程：

遇到一个复杂的问题，可以尝试先解决简单的情况，然后慢慢逼近最终的问题。

第一步： 手工试算几个，找找规律

B(1) = 1
D(1) = 1 S(1) = 1

B(2) = $C_2^0 * C_2^1 * C_2^2 = 2$
D(2) = 1 + 2 = 3 = (2^2 - 1)/(2-1) S(2) = 1 + 3 = 4

B(3) = $C_3^0 * C_3^1 * C_3^2 * C_3^3$
= 1 * 3 * 3 * 1 = 3 ^ 2 = 9 S(3) = D(1) + D(2) + D(3)
D(3) = 1 + 3 + 9 = (3^3-1)/(3-1) = 13 S(3) = 1 + 3 + 13 = 17

B(4) = $C_4^0 * C_4^1 * C_4^2 * C_4^3 * C_4^4$
= $1 * \frac{4}{1} * \frac{4*3}{1*2} * \frac{4}{1} * 1$
= 1 * 4 * 6 * 4 * 1
= (2^5) * (3^1)
D(4) = 1 + 2 + 4 + 8 + 16 + 32
+ 3 + 6 + 12 + 24 + 48 + 96 = 252 S(4) = 1 + 3 + 13 + 252 = 269
= (2^6-1)/(2-1) * (3^2-1)/(3-1)

B(5) = $C_5^0 * C_5^1 * C_5^2 * C_5^3 * C_5^4 * C_5^5$
= $1 * \frac{5}{1} * \frac{5*4}{1*2} * \frac{5*4}{1*2} * \frac{5}{1} * 1$
= 1 * 5 * 10 * 10 * 5 * 1
= (2^2) * (5^4)
D(5) = 1 + 2 + 4
+ 5 + 10 + 20 S(5) = 1 + 3 + 13 + 252 + 5467
+ 25 + 50 + 100 =5736
+ 125 + 250 + 500
+ 625 + 1250 + 2500
= (2^3-1)/(2-1) * (5^5-1)/(5-1)
= 7*781
= 5467

可以看到 B(n)的求解与排列组合数的因子有关系。

```
B(2) = 2^1
B(3) = 3^2
B(4) = (2^5) * (3^1)
B(5) = (2^2) * (5^4)
```

如果一个数能够分解成几个素数乘积，利用等比数列的求和公式，可以推导出所有因子之和的公式：

(p1^a1) * (p2^a2)

1	p1	(p1^2)	…	(p1^a1)
p2 * 1	p2 * p1	p2 * (p1^2)	…	p2 * (p1^a1)
(p2^a2) * 1	(p2^a2) * p1	(p2^a2) * (p1^2)	…	(p2^a2) * (p1^a1)

$$因子和 = \frac{p1^{a1+1}-1}{p1-1} * \frac{p2^{a2+1}-1}{p2-1}$$

公式中只有 2 个不同的质因子，可以推广到更多质因子的情况。

现在可以暴力计算了：

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    // 先把一些阶乘计算好，保存起来
    let mut fact = vec![BigUint::from(1 as u64); 101];
    let mut a = BigUint::from(1 as u64);
    for n in 2..=100 {
        a *= BigUint::from(n as u64);
        fact[n] = a.clone();
    }
    //println!("{:?}", fact);

    let mut s = 0;
    for n in 1..=10 {
        let mut prod = BigUint::from(1_u64);
        for r in 1..=n {
            let comb = &fact[n] / &fact[r] / &fact[n - r];
            prod *= &comb;
            //println!("{} {} {}", n, r, comb.to_string());
        }
        let b = prod.to_string().parse::<u64>().unwrap();
        println!("B({}) = {}", n, b);

        let f_all = primes::factors(b);
        let f_uniq = primes::factors_uniq(b);
        //println!("{:?}", f_all);
        //println!("{:?}", f_uniq);

        let mut d = 1;
        for f in f_uniq {
            let c = f_all.iter().filter(|&n| *n == f).count();
            d *= (f.pow(c as u32 + 1) - 1) / (f-1);
        }
        //println!("D({}) = {}", n, d);
```

```
        s += d;
        println!("S({}) = {}", n, s);
    }
}
```

可以正确地计算出 S(10)，但在计算 S(11)时就会出现溢出错误。

第二步：

溢出发生在 pow()函数的计算上，求排列组合数和求阶乘的运算量太大，没必要把乘积计算出来，可以将因子保存在一个向量中，不断添加和删除相应的元素即可。

```
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut s = 0;
    for n in 1..=100 {
        let mut factors = vec![];
        for i in 1..=n {
            let mut f = comb_factors(n, i);
            factors.append(&mut f);
        }
        factors.sort();

        let d = factors_sum(&factors);
        println!("D({}) = {:?}", n, d);

        s = (s + d) % 1_000_000_007_u64;
        println!("S({}) = {}", n, s);
    }
}

fn factors_sum(v: &Vec<u64>) -> u64 {
    let mut uniq = v.clone();
    uniq.dedup();

    let mut prod = BigUint::from(1_u64);
    for p in uniq {
        let c = v.iter().filter(|&x| *x == p).count() as u64;
        let t = (big_pow(p, c + 1) - BigUint::from(1_u64)) / (BigU
int::from(p - 1));
        //println!("{} {} {}", p, c, t);
        prod = prod * t % 1_000_000_007_u64;
    }
    let prod = prod % 1_000_000_007_u64;
    prod.to_string().parse::<u64>().unwrap()
}

fn big_pow(a: u64, b: u64) -> BigUint {
```

```
    let mut prod = BigUint::from(1 as u64);
    for _i in 0..b {
        prod *= BigUint::from(a as u64);
    }
    prod
}

// va中元素已经排好序
fn vec_remove(va: &mut Vec<u64>, vb: &Vec<u64>) {
    for item in vb {
        let index = va.binary_search(&item).unwrap();
        //println!("{:?} {:?} {}", va, vb, index);
        va.remove(index);
    }
}

fn comb_factors(m: u64, n: u64) -> Vec<u64> {
    let mut factors = vec![];
    let mut x = m;
    for i in 0..n {
        let mut f = primes::factors(x);
        factors.append(&mut f);
        x -= 1;
    }
    factors.sort();
    //println!("{:?}", factors);
    for i in 2..=n {
        let f = primes::factors(i);
        //println!("{} {:?}", n, f);
        vec_remove(&mut factors, &f);
    }
    factors.to_vec()
}
```

这次可以很容易地计算出 S(100)，但要计算 S(1000)则要花相当长的时间，主要是因为数组越来越大，数组排序太花时间。

还得优化。

第三步 用 HashMap

数组中元素的排序太慢，尝试换成字典来存储因子，在 Rust 中用 HashMap 来实现。比如，B(10)中含有 12 个 2，10 个 3，8 个 5，3 个 7。

而 B(100)中含有如下因子，字典中 key 是质因子，value 是个数。

```
{97: 93, 19: 65, 83: 65, 41: 44, 59: 17, 79: 57, 61: 21, 67: 33,
2: 335, 3: 192, 37: 20, 23: 56, 31: 69, 47: 80, 13: 21, 71: 41, 7:
 148, 73: 45, 11: 81,
43: 56, 17: 5, 53: 5, 5: 176, 89: 77, 29: 45}
```

修改后的代码：

```
extern crate num_bigint;
use num_bigint::BigUint;

use std::collections::HashMap;
fn main() {
    let mut s = 0;
    for n in 1..=10000 {
        let map = comb_factors_hash_map(n);
        //println!("{:?}", map);

        let temp = factors_hash_map_sum(&map);
        //println!("D({}) = {:?}", n, temp);

        s = (s + temp) % 1_000_000_007_u64;
        println!("S({}) = {}", n, s);
    }
}

fn big_pow(a: u64, b: u64) -> BigUint {
    let mut prod = BigUint::from(1 as u64);
    for _i in 0..b {
        prod *= BigUint::from(a as u64);
    }
    prod
}

fn factors_hash_map_sum(map: &HashMap<u64, u64>) -> u64 {
    let mut prod = BigUint::from(1_u64);
    for (&f, count) in map {
        let t = (big_pow(f, count + 1) - BigUint::from(1_u64)) /
(BigUint::from(f - 1));
        prod = prod * t % 1_000_000_007_u64;
    }
    let prod = prod % 1_000_000_007_u64;
    prod.to_string().parse::<u64>().unwrap()
}

fn comb_factors_hash_map(x: u64) -> HashMap<u64, u64> {
    let mut map = HashMap::new();

    let mut count = x as i64 - 1;
    for n in (2..=x).rev() {
        let f = primes::factors(n);
        let a = factors_to_hash_map(&f);
        if count >= 0 {
            hash_map_add_count(&mut map, &a, count as u64);
        } else {
            hash_map_substract_count(&mut map, &a, count.abs() as
u64);
        }
        count -= 2;
```

```
    }
    map
}

fn factors_to_hash_map(factors: &Vec<u64>) -> HashMap<u64, u64> {
    let mut map = HashMap::new();
    for f in factors {
        let v = map.get(f).cloned(); // 如果不写cloned()，有警告，
不理解原因
        match v {
            Some(x) => {
                map.insert(*f, x + 1);
            }
            None => {
                map.insert(*f, 1);
            }
        }
    }
    map
}

fn hash_map_add_count(map: &mut HashMap<u64, u64>, a: &HashMap<u6
4, u64>, times: u64) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x + count * times);
            }
            None => {
                map.insert(*f, *count * times);
            }
        }
    }
}

fn hash_map_substract_count(map: &mut HashMap<u64, u64>, a: &HashM
ap<u64, u64>, times: u64) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x - count * times);
            }
            None => {}
        }
    }
}
```

这次可以轻松地计算到 S(1000)，但后面越来越慢。还得继续优化。

第四步 加速 pow()运算

计算一个数的 n 次方，有一个常用的算法可以加速，称为 Exponentiation by squaring 算法，相关网站：

https://en.wikipedia.org/wiki/Exponentiation_by_squaring

修改后程序运行速度提高，多运行一段时间，可以计算出 S(5000)。

```
fn big_pow(base: u64, exp: u64) -> BigUint {
    let mut result = BigUint::from(1 as u64);
    let mut b = BigUint::from(base);
    let mut e = exp;
    while e > 0 {
        if e % 2 == 1 {
            result = result * &b;
        }
        e = e >> 1;
        b = &b * &b;
    }
    result
}
```

第五步 递推公式

程序运行几个小时，应该能够计算出 S(20000)，在它计算的过程中，我还要尝试进一步的优化。但在这之后，我开始走弯路了，尝试缓存一些中间的计算结果来进行加速，效果都不理想。

google "euler project problem 650"，发现一篇文章。

http://goatleaps.xyz/euler/maths/Project-Euler-650.html

里面提到一个 B(n-1)递推出 B(n)的公式，可以进一步优化。

我也用笨办法推导出了这条公式：

$$B(n) = C_n^0 * C_n^1 * C_n^2 * C_n^3 * \cdots * C_n^{n-1} * C_n^n$$

$$= 1 * \frac{n}{1} * \frac{n*(n-1)}{1*2} * \frac{n*(n-1)*(n-2)}{1*2*3} * \cdots * \frac{n}{1} * 1$$

$$= 1 * \frac{n-1}{1} * \frac{n}{n-1} * \frac{(n-1)*(n-2)}{1*2} * \frac{n}{n-2} * \frac{(n-1)*(n-2)*(n-3)}{1*2*3} * \frac{n}{n-3} * \cdots * 1 * \frac{n}{1} * 1$$

$$= 1 * C_{n-1}^1 * \frac{n}{n-1} * C_{n-1}^2 * \frac{n}{n-2} * C_{n-1}^3 * \frac{n}{n-3} * \cdots * C_{n-1}^{n-1} * \frac{n}{1} * 1$$

$$= B(n-1) * \frac{n\hat{}(n-1)}{(n-1)!}$$

$$B(n) = B(n-1) * \frac{n\hat{}n}{n!}$$

```
use std::collections::HashMap;
```

```
const MODULUS: u64 = 1_000_000_007_u64;
extern crate num_bigint;
use num_bigint::BigUint;

fn main() {
    let mut factorial_factors = HashMap::new();
    let mut map = HashMap::new();
    let mut s = 1;
    for n in 2..=20000 {
        let factor_n = factors_map(n);
        map_add_count(&mut map, &factor_n, n);

        // 缓存 n! 阶乘的因子
        map_add(&mut factorial_factors, &factor_n);

        map_substract(&mut map, &factorial_factors);

        //println!("{:?}", &map);
        let d = map_sum(&map);

        s = (s + d) % MODULUS;
        if n == 10 {
            assert_eq!(s, 141740594713218418 % MODULUS);
        }
        if n == 100 {
            assert_eq!(s, 332792866_u64);
        }
        if n % 100 == 0 {
            println!("D({}) = {:?} \t S({}) = {}", n, d, n, s);
        }
    }
    println!("{}", s);

}

fn map_sum(map: &HashMap<u64, u64>) -> u64 {
    let mut prod = BigUint::from(1_u64);
    for (&f, count) in map {
        let t = (big_pow(f, count + 1) - BigUint::from(1_u64)) /
(BigUint::from(f - 1));
        prod = prod * t % 1_000_000_007_u64;
    }
    let prod = prod % 1_000_000_007_u64;
    prod.to_string().parse::<u64>().unwrap()
}

fn big_pow(base: u64, exp: u64) -> BigUint {
    let mut result = BigUint::from(1 as u64);
    let mut b = BigUint::from(base);
    let mut e = exp;
    while e > 0 {
        if e % 2 == 1 {
            result = result * &b;
```

```
        }
        e = e >> 1;
        b = &b * &b;
    }
    result
}

fn factors_map(n:u64) -> HashMap<u64, u64> {
    let mut map = HashMap::new();
    let all_factors = primes::factors(n);
    for f in &all_factors {
        let v = map.get(f).cloned(); // 如果不写cloned()，有警告，
不理解原因
        match v {
            Some(x) => {
                map.insert(*f, x + 1);
            }
            None => {
                map.insert(*f, 1);
            }
        }
    }
    map
}

fn map_add(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u64>) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x + count);
            }
            None => {
                map.insert(*f, *count);
            }
        }
    }
}

fn map_add_count(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u6
4>, times: u64) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x + count * times);
            }
            None => {
                map.insert(*f, *count * times);
            }
        }
    }
}
```

```
fn map_substract(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u6
4>) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x - count);
            }
            None => {}
        }
    }
}
```

程序并没有想像中提高运行速度，还得想办法。

第六步 倒数的模运算

问题仍出在计算因子和的公式上，计算乘方时用大整数库，当数字越来越大时，速度越来越慢。

$$\text{因子和} = \frac{p1^{a1+1}-1}{p1-1} * \frac{p2^{a2+1}-1}{p2-1}$$

这时得利用模运算的数学知识了，模运算有一些基本性质：如果 a ≡ b (mod m) 且有 c ≡ d (mod m)，那么下面的模运算律成立：

```
a + c ≡ b + d (mod m)
a − c ≡ b − d (mod m)
a × c ≡ b × d (mod m)
```

这里可以看到加、减、乘都满足这种同余定理，但没有除法！

问题没完，在数学运算中：a / b == a * (1 / b)，其中 1 / b 叫 b 的倒数。那么在模运算中，有没有这种类似于倒数的存在呢？答案是有！它就是**乘法逆元**！

相关文章链接： http://conw.net/archives/6/

有人已经写好了 Rust 的代码，直接可以拿来用。

http://rosettacode.org/wiki/Modular_inverse#Rust

pow()乘方函数也有带模运算的 Rust 代码，直接拿来用。

https://rob.co.bb/posts/2019-02-10-modular-exponentiation-in-rust/

这次优化彻底去掉了大整数运算，速度提升了不知多少倍。

在我的笔记本电脑上，8 秒钟计算出 S(20000)！

最终的源代码：

```
use std::collections::HashMap;

const MODULUS: u64 = 1_000_000_007_u64;

fn main() {
    let mut factorial_factors = HashMap::new();
    let mut map = HashMap::new();
    let mut s = 1;
    for n in 2..=20000 {
        let factor_n = factors_map(n);
        map_add_count(&mut map, &factor_n, n);

        // 缓存 n! 阶乘的因子
        map_add(&mut factorial_factors, &factor_n);

        map_substract(&mut map, &factorial_factors);

        //println!("{:?}", &map);
        let d = map_sum(&map);

        s = (s + d) % MODULUS;
        if n == 10 {
            assert_eq!(s, 141740594713218418 % MODULUS);
        }
        if n == 100 {
            assert_eq!(s, 332792866_u64);
        }
        if n % 100 == 0 {
            println!("D({}) = {:?} \t S({}) = {}", n, d, n, s);
        }
    }
    println!("{}", s);

}

// http://rosettacode.org/wiki/Modular_inverse#Rust
// 求乘法逆元
fn mod_inv(a: isize, module: isize) -> isize {
    let mut mn = (module, a);
    let mut xy = (0, 1);

    while mn.1 != 0 {
        xy = (xy.1, xy.0 - (mn.0 / mn.1) * xy.1);
        mn = (mn.1, mn.0 % mn.1);
    }
```

```
    while xy.0 < 0 {
        xy.0 += module;
    }
    xy.0
}

// https://rob.co.bb/posts/2019-02-10-modular-exponentiation-in-r
ust/
fn mod_pow(mut base: u64, mut exp: u64, modulus: u64) -> u64 {
    let mut result = 1;
    base = base % modulus;
    while exp > 0 {
        if exp % 2 == 1 {
            result = result * base % modulus;
        }
        exp = exp >> 1;
        base = base * base % modulus
    }
    result
}


fn map_sum(map: &HashMap<u64, u64>) -> u64 {
    let mut prod = 1;
    for (&f, count) in map {
        if *count > 0 {
            // 计算 f^(count+1) / (f-1)
            let temp = mod_pow(f, count + 1, MODULUS) - 1;
            // 计算(f-1)的乘法逆元
            let inv = mod_inv(f as isize - 1, MODULUS as isize) as
 u64;
            //println!("inv:{}", inv);
            let temp = temp * inv % MODULUS;
            prod = prod * temp % MODULUS;
            //println!("f:{} count+1:{} prod: {}", f, count+1, pro
d);
        }
    }
    prod
}

fn factors_map(n:u64) -> HashMap<u64, u64> {
    let mut map = HashMap::new();
    let all_factors = primes::factors(n);
    for f in &all_factors {
        let v = map.get(f).cloned(); // 如果不写cloned()，有警告，
不理解原因
        match v {
            Some(x) => {
                map.insert(*f, x + 1);
            }
            None => {
```

```
                map.insert(*f, 1);
            }
        }
    }
    map
}

fn map_add(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u64>) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x + count);
            }
            None => {
                map.insert(*f, *count);
            }
        }
    }
}

fn map_add_count(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u6
4>, times: u64) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x + count * times);
            }
            None => {
                map.insert(*f, *count * times);
            }
        }
    }
}

fn map_substract(map: &mut HashMap<u64, u64>, a: &HashMap<u64, u6
4>) {
    for (f, count) in a {
        let v = map.get(f).cloned();
        match v {
            Some(x) => {
                map.insert(*f, x - count);
            }
            None => {}
        }
    }
}
```

优化要点：

1）不要轻易用大整数运算库

2）质因子分解

3）同余定理

4）Exponentiation by squaring 乘方运算加速

5）找到递推公式

6）乘法逆元

卡了我好几天的优化算法是因为以前不知道“乘法逆元”的存在。

第 684 题 逆数字和

定义 s(n)是数字和为 n 的最小整数，例如 s(10)=19。

记 S(k) = s(1) + s(2) + ... + s(k)，可以知道 S(20)=1074。

设斐波那契数列 f(n)按如下方式定义：

$$f_0 = 0$$
$$f_1 = 1$$
$$f_i = f_{i-2} + f_{i-1} \quad (i \geq 2)$$

求：

$$\sum_{i=2}^{90} S(f_i) \mod 1\,000\,000\,007$$

解题过程：

遇到一个复杂的问题，首先可以尝试先解决简单的情况，然后慢慢逼近最终的问题。

第一步：

题目中知道 S(20)=1074，那就先求 S(20)。

手算前 20 个，发现一个规律。每 9 个为一组，后面的数字是 9，前面的数字从 0 到 8。

可以推导出一个公式：s(n) = (a+1) * 10^b - 1

s(0)	0	
s(1)	1	
s(2)	2	
s(3)	3	
s(4)	4	
s(5)	5	
s(6)	6	
s(7)	7	
s(8)	8	
s(9)	9	= 10 - 1
s(10)	19	= 20 - 1
s(11)	29	= 30 - 1
s(12)	39	= 40 - 1
s(13)	49	= 50 - 1
s(14)	59	= 60 - 1
s(15)	69	= 70 - 1
s(16)	79	= 80 - 1
s(17)	89	= 90 - 1
s(18)	99	= 100 - 1
s(19)	199	= 200 - 1
s(20)	299	= 300 - 1

s(n) a999…9 1个a，b个9
a=n%9
b=n/9 取整
s(n) = (a+1) * 10^b - 1

S(20)很容易求出来：

```
let mut ss = 0;
for n in 1..=20 {
    let a = n % 9;
    let b = n / 9;
    let s = (a + 1) * 10_u32.pow(b) - 1;
    println!("{}", s);
    ss += s;
}
println!("S(20): {}", ss);
```

但求 S(200)时就会溢出，因为 10^b 是一个超出 u64 范围的大整数。

第二步

把 90 个斐波那契数求出来，以后要用。

```
let mut fib = [0_u64; 91];
fib[1] = 1;
for i in 2..=90 {
    fib[i] = fib[i - 1] + fib[i - 2];
```

```
    println!("fib({}): {}", i, fib[i]);
}
```

运行一下，可以发现第 90 个数非常非常大。

```
fib(88): 1100087778366101931
fib(89): 1779979416004714189
fib(90): 2880067194370816120
```

第三步

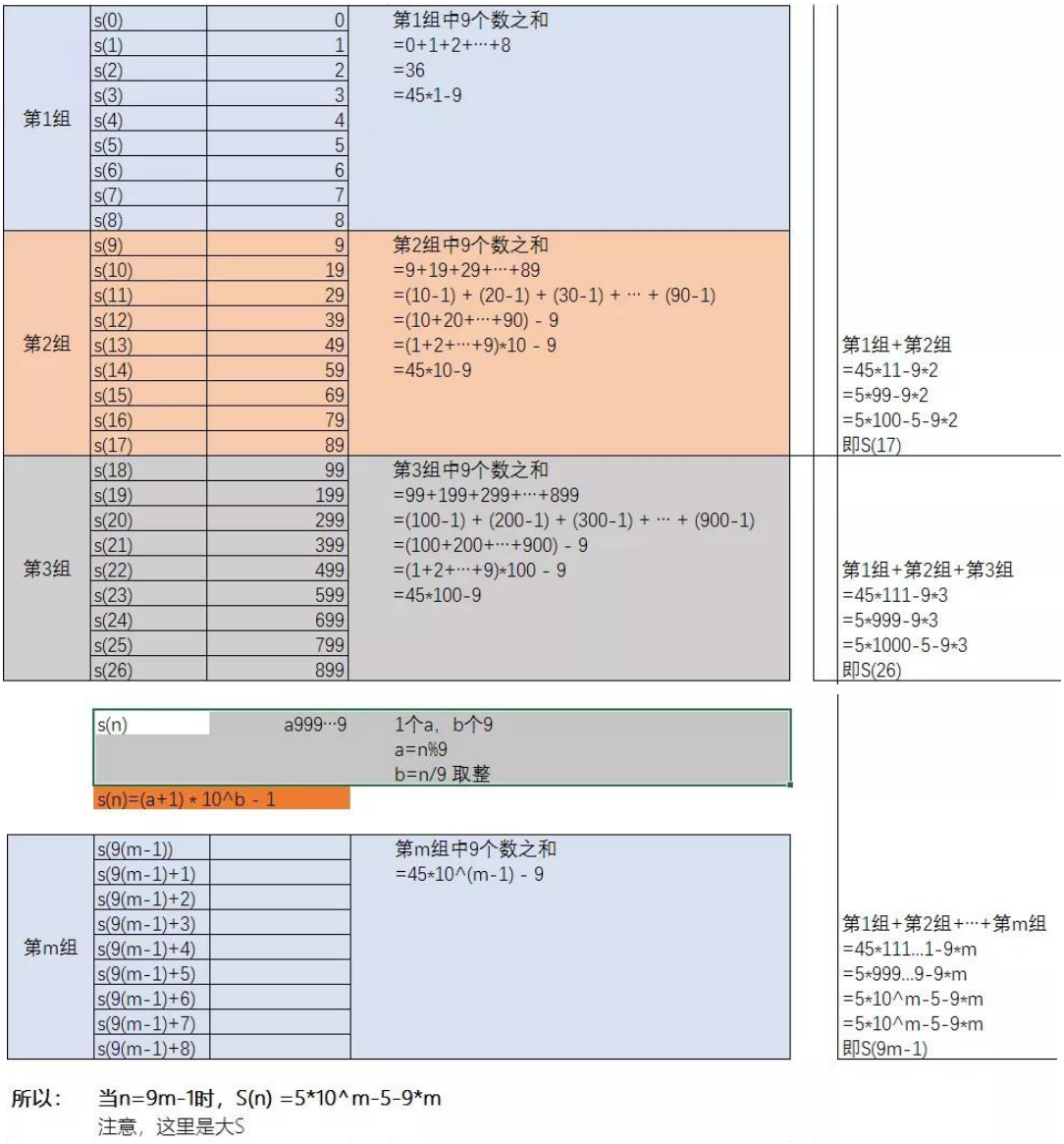
首先能够想到的是用大整数库 num-bigint 优化算法。

```
fn fs(n: u64) -> u64 {
    let a = n % 9;
    let b = n / 9;
    let mut s = BigUint::from(a + 1);
    for i in 0..b {
        s = s * BigUint::from(10_u64);
    }
    s = s - BigUint::from(1_u64);
    let result = s % BigUint::from(1_000_000_007_u64);
    result.to_string().parse::<u64>().unwrap()
}
```

现在可以比较快地计算出 s(20000)和 S(20000)，但离目标 288006719437081 6120 还有相当大的距离，bigint 这条路不通，还得改进算法。

第四步

看看 S(n)的计算是否还有其它规律。每 9 个为一组，计算 9 个数之和，可以找到规律。



可以发现，当 n = 9 * m - 1 时，有公式：

$S(n) = 5 * 10^m - 5 - 9 * m$

有了这个公式，在计算 S(20)时，可以先快速计算出 S(17)，再加上 s(18)+s(19)+s(20)就可以得到最终结果，算法复杂度相当于计算四次 s(n)。

现在仍有一个关键问题没有解决，10^m 是一个非常大的数，必须找到快速计算 $10^m \bmod 1\_000\_000\_007\_u64$ 的办法。

这里要利用费马小定理：

如果 p 是一个素数，而整数 a 不是 p 的倍数，则有 $a^{(p-1)} \bmod p = 1$。

看一个特殊的例子，a=10，p=7 时，有助于大致理解费马小定理的含义。

10^0	1 % 7=	1
10^1	10 % 7=	3
10^2	100 % 7=	2
10^3	1000 % 7=	6
10^4	10000 % 7=	4
10^5	100000 % 7=	5
10^6	1000000 % 7=	1
10^7	10000000 % 7=	3
	...	
对于素数p		
10^(p-1)	10^(p-1) % p =	1

有个这个定理，$10^m \bmod 1\_000\_000\_007\_u64 = 10^{(m \bmod 1\_000\_000\_006\_u64)} \bmod 1\_000\_000\_007\_u64$，可以大大加速计算过程，25 秒计算出结果。

最后的源代码：

```rust
use std::time::SystemTime;

const PRIME: u64 = 1_000_000_007_u64;

#[macro_use]
extern crate lazy_static;
lazy_static! {
    static ref ARRAY: Vec<u64> = {
        println!("initializing ARRAY ...");
        let mut arr = vec![1];
        let mut x = 1;
        for _i in 1..PRIME - 1 {
            x = x * 10 % PRIME;
            arr.push(x as u64);
        }
        arr
    };
}

fn main() {
    let start = SystemTime::now();
    let mut result = 0;
    let mut fib = vec![0_u64, 1];
    for i in 2..=90 {
        let n = fib[i - 1] + fib[i - 2];
        fib.push(n);
        let ss = fss(n);
        result = (result + ss) % PRIME;
        println!("n:{} S:{} result: {}", n, ss, result);
    }
    println!("{:?}", start.elapsed());
}
```

```rust
fn ten_power_mod(n: u64) -> u64 {
    let m = n % (PRIME - 1);
    ARRAY[m as usize]
}

fn fs(n: u64) -> u64 {
    let a = n % 9;
    let b = n / 9;
    let s = (a + 1) * ten_power_mod(b) - 1;
    s % PRIME
}

fn sum_group(m: u64) -> u64 {
    let temp = (9 * m) % PRIME;
    let s = 5 * ten_power_mod(m) + PRIME - temp - 5;
    s % PRIME
}

fn fss(n: u64) -> u64 {
    let m = n / 9;
    let mut s = sum_group(m);
    for i in 9 * m..=n {
        s += fs(i);
    }
    s % PRIME
}
```

这道题的难度系数虽然被归在 5%一类中，但还是相当有挑战的。

第 686 题 2 的幂

$2^7 = 128$，在 2 的 n 次方中，首次遇到前 2 位数字为“12”，下一次再遇到“12”的情况是 2^{80}。 考虑 2^j 用 10 进制表示，定义 p(L, n)为第 n 个满足前导数字为 L 的最小 j 值。

即有：

p(12, 1)=7

p(12, 2)=80

我们已知 p(123, 45)=12710，求 p(123, 678910)。

解题过程：

遇到一个复杂的问题，首先可以尝试先解决简单的情况，然后慢慢逼近最终的问题。

第一步：

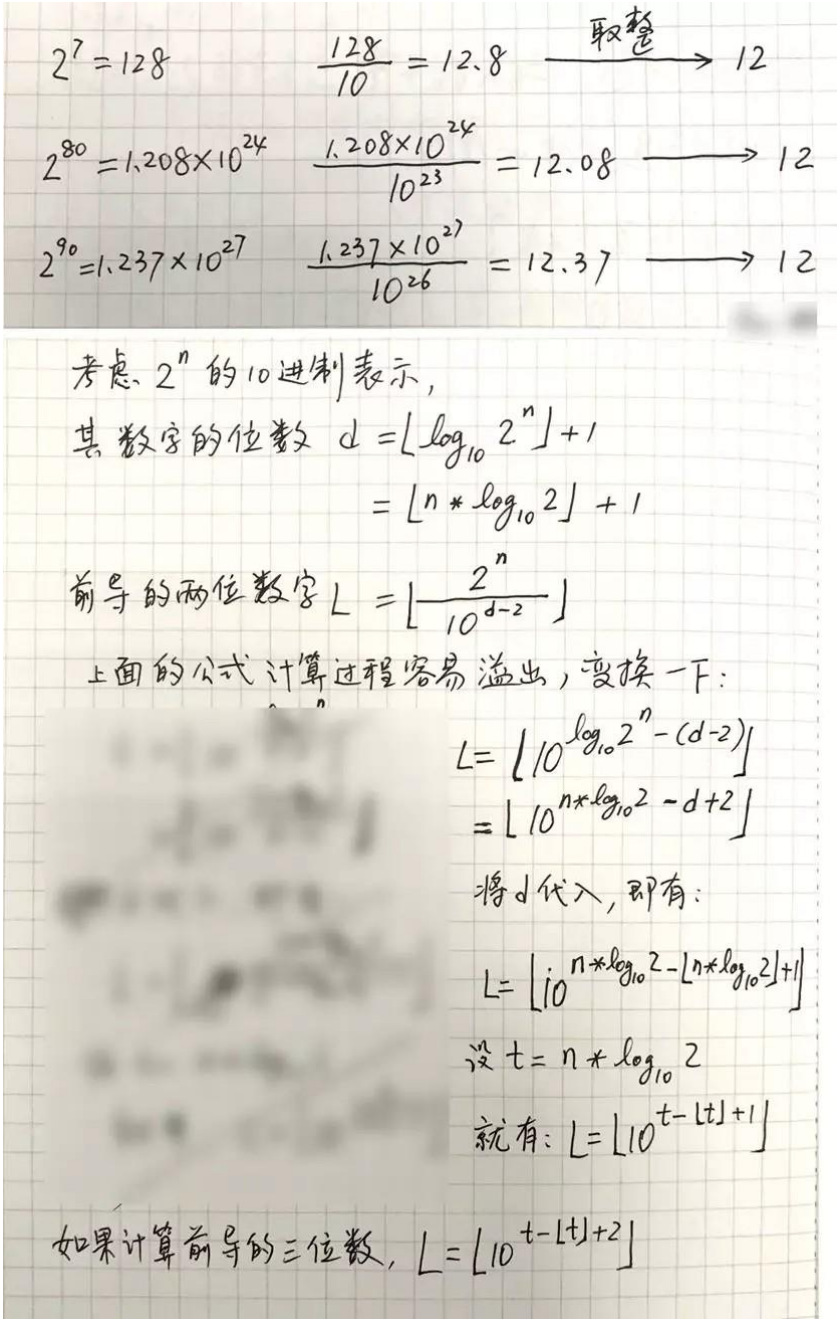
首先用 excel 演算一下。

n	2 ^ n	数字个数d	10^(d-2)		
1	2	1			
2	4	1			
3	8	1			
4	16	2			
5	32	2			
6	64	2			
7	128	3	10	12.8	12
8	256	3	10	25.6	25
9	512	3	10	51.2	51
10	1024	4	100	10.24	10
…	…				
78	3.02231E+23	24	1E+22	30.22314549	30
79	6.04463E+23	24	1E+22	60.44629098	60
80	1.20893E+24	25	1E+23	12.0892582	12
81	2.41785E+24	25	1E+23	24.17851639	24
82	4.8357E+24	25	1E+23	48.35703278	48
83	9.67141E+24	25	1E+23	96.71406557	96
84	1.93428E+25	26	1E+24	19.34281311	19
85	3.86856E+25	26	1E+24	38.68562623	38
86	7.73713E+25	26	1E+24	77.37125246	77
87	1.54743E+26	27	1E+25	15.47425049	15
88	3.09485E+26	27	1E+25	30.94850098	30
89	6.1897E+26	27	1E+25	61.89700196	61
90	1.23794E+27	28	1E+26	12.37940039	12
91	2.47588E+27	28	1E+26	24.75880079	24

能够很快找到规律，前导的两位数字可以比较容易的计算出来。

第二步：

数学推导的过程并不复杂，需要一点点对数方面的知识。



先用已知的 2 个答案检查算法的正确性。

```
let mut count = 0;
for n in 7..100 {
    let t = (n as f64) * 2_f64.log10();
    let m = t - t.floor() + 1.0;
    let m = 10_f64.powf(m).floor() as u64;
    if m == 12 {
        count += 1;
        println!("p(12, {}) = {} ", count, n);
        if count == 1 {
            assert_eq!(n, 7, "p(12,1) = 7");
        }
        if count == 2 {
```

```
            assert_eq!(n, 80, "p(12,2) = 80");
        }
    }
}
```

第三步

前导数字为 123，公式稍微有一点点变化，t - t.floor() + 1 变成 t - t.floor() + 2，然后暴力求解最终的问题即可，根据机器性能，需要几秒到几十秒的时间。

```
let mut count = 0;
for n in 7.. {
    let t = (n as f64) * 2_f64.log10();
    let m = t - t.floor() + 2.0;
    let head = 10_f64.powf(m) as u64;
    if head == 123 {
        count += 1;
        println!("p(123, {}) = {} ", count, n);
        if count == 45 {
            assert_eq!(n, 12710, "p(123, 45) = 12710");
        }
        if count == 678910 {
            break;
        }
    }
}
```

第 700 题 Eulercoin

欧拉诞生于 1707 年 4 月 15 日，对于序列(1504170715041707 * n) mod 4503599627370517，如果一个元素小于前面发现的所有 Eulercoin，则其称为 Eulercoin。

例如，第一个元素是 1504170715041707，为第一个 Eulercoin，第二个元素为 3008341430083414，由于它大于 1504170715041707，所以不是 Eulercoin。然而，第三个元素为 8912517754604，比前面的数都小，被称为 Eulercoin。前两个 Eulercoins 之和是 1513083232796311。

求所有 Eulercoin 之和。

解题过程：

第一步，先根据题意暴力求解

```
const INC: u64 = 1504170715041707_u64;
const MOD: u64 = 4503599627370517_u64;

let mut x = 0_u64;
let mut min = INC;
for n in 1_u64.. {
    x = (x + INC) % MOD;
    if x <= min {
        min = x;
        println!("{:20} {:20}", n, x);
    }
}
```

运行这个程序，可以输出如下结果。

```
                   1     1504170715041707
                   3        8912517754604
                 506        2044785486369
                2527        1311409677241
                4548         578033868113
               11117         422691927098
               17686         267349986083
               24255         112008045068
               55079          68674149121
               85903          25340253174
              202630           7346610401
              724617           4046188430
             1246604            745766459
             6755007            428410324
            12263410            111054189
            42298633             15806432
           326125654             15397267
           609952675             14988102
           893779696             14578937
          1177606717             14169772
          1461433738             13760607
          1745260759             13351442
          2029087780             12942277
          2312914801             12533112
          2596741822             12123947
          2880568843             11714782
          3164395864             11305617
          3448222885             10896452
          3732049906             10487287
          4015876927             10078122
          4299703948              9668957
          4583530969              9259792
          4867357990              8850627
          5151185011              8441462
          5435012032              8032297
```

```
          5718839053          7623132
          6002666074          7213967
          6286493095          6804802
          6570320116          6395637
          6854147137          5986472
          7137974158          5577307
          7421801179          5168142
          7705628200          4758977
          7989455221          4349812
          8273282242          3940647
          8557109263          3531482
          8840936284          3122317
          9124763305          2713152
          9408590326          2303987
          9692417347          1894822
          9976244368          1485657
         10260071389          1076492
         10543898410           667327
         10827725431           258162
             … … … …             … … … …
```

越往后，想找到一个 Eulercoin 愈发困难，必须得改进算法。

第二步，找规律

在 n=2527 之后，后一个 eulercoin 与前一个 eulercoin 有着递推的关系，补上一些数字之后，就能发现更为明显的关系。比如，n=2021, 6569, 30824, 116727...时，得到的数虽然不是 eulercoin，但数值非常大，与 4503599627370517 越来越接近。

	n	eulercoin
	1	1504170715041707
	3	8912517754604
	503	4496731895102282
	506	2044785486369
	2021	4502866251561389
+506=	2527	1311409677241
+2021=	4548	578033868113
+2021=	6569	4503444285429502
+4548=	11117	422691927098
+6569=	17686	267349986083
+6569=	24255	112008045068
+6569=	30824	4503556293474570
+24255=	55079	68674149121
+30824=	85903	25340253174
+30824=	116727	4503581633727744
+85903=	202630	7346610401

根据发现的规律，需要保存好最小的数和最大的数，不用一个数一个数的计算，效率非常高，不到 1 秒计算完成。

```
let mut sum = 1504170715041707_u64 + 8912517754604_u64 + 204478548636
9_u64 + 1311409677241_u64;
```

```
let mut n_max = 2021_u64;
let mut max = 4502866251561389_u64;
let mut n = 2527_u64;
let mut x = 1311409677241_u64;
let mut min = x;

while min > 0 {
    let temp_n = n + n_max;
    let temp_x = (x + max) % 4503599627370517_u64;
    if temp_x <= min {
        n = temp_n;
        x = temp_x;
        min = x;
        sum += x;
        println!("{:20} {:12} {:20}", n, x, sum);
    }
    if temp_x > max {
        n_max = temp_n;
        max = temp_x;
        //println!("max: {} {} ", n_max, max);
    }
}
```

再优化

看了欧拉论坛中的优秀贴子，发现 n 的索引值并不需要记录，代码还可以更优美一些。

```
let inc = 1504170715041707_u64;
let modular = 4503599627370517_u64;
let mut low = inc;
let mut high = inc;
let mut sum = inc;
while low > 0 {
    let next = (low + high) % modular;
    if next < low {
        low = next;
        sum += low;
        println!("{:20} {:20}", low, sum);
    } else {
        high = next;
    }
}
println!("{}", sum);
```

后记

后来主要使用 Python 编程来解决欧拉计划的难题，主要放在 CSDN 上。