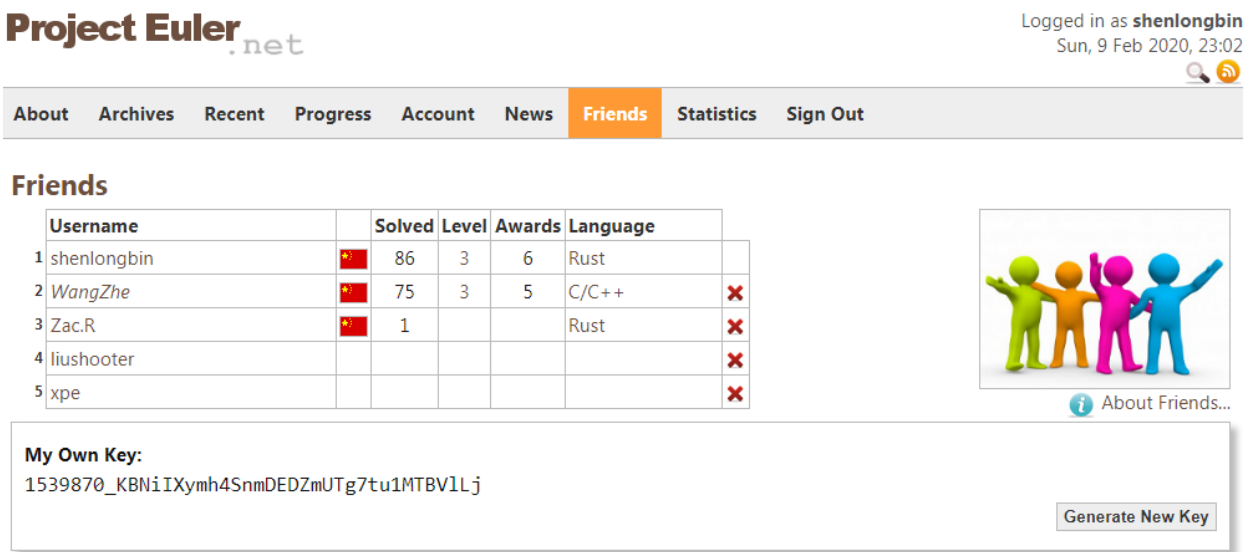
用欧拉计划学 Rust 编程

V1.1



申龙斌

2020 年 2 月 9 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。

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。

我的微信号是：SLOFSLB，快满 5000 好友了，还剩几个位置，暗号：Rust。

也欢迎关注我的微信公众号“申龙斌的程序人生”，不要错过更多精彩内容。

✧ 个人公众号开通 3 周年纪念

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，用官网上的 rustup 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。

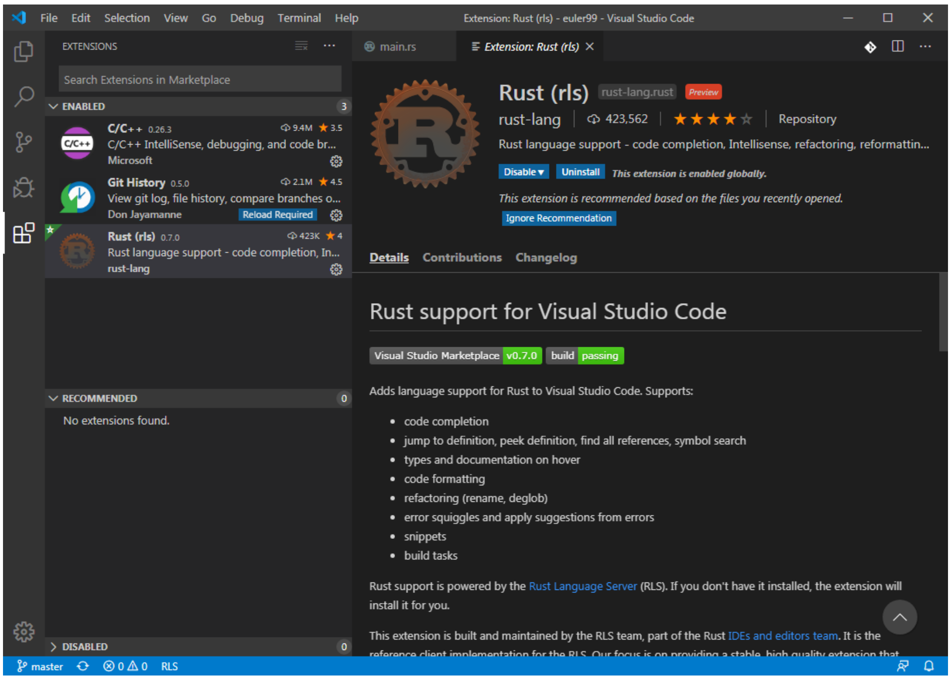
安装：

```
rustup component add clippy
```

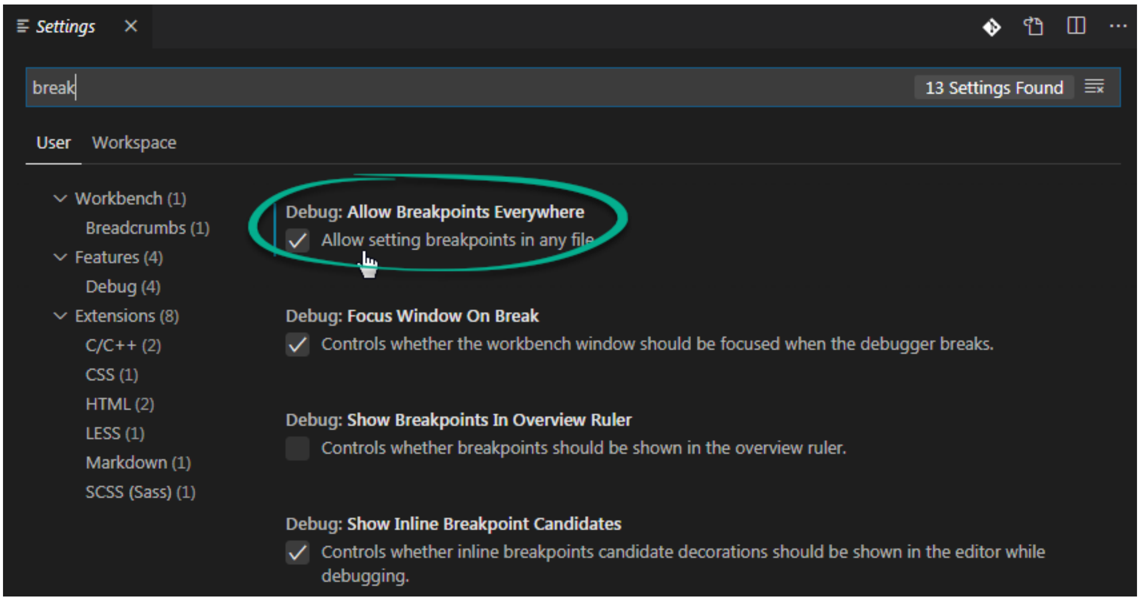
使用：

```
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。

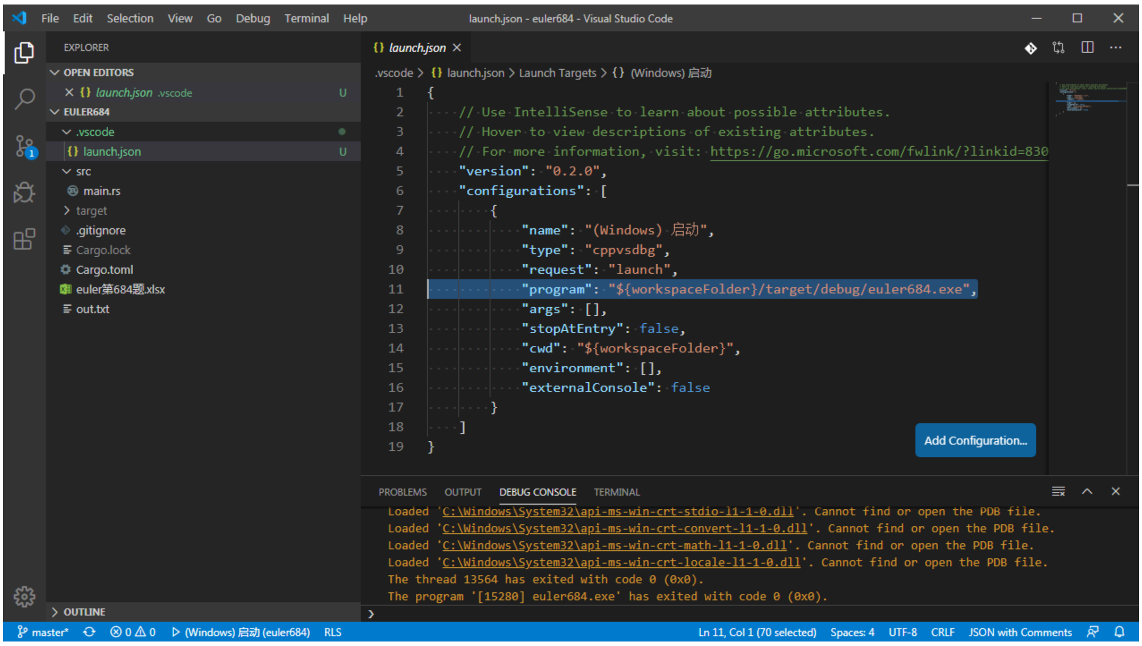


调试程序的时候，首先打开 vscode 里的 debug 设置选项。



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”,
```



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

问题描述：

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，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

第 2 题 偶斐波那契数

问题描述：

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

问题描述：

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，一个语句块的最后可以是一个表达式，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

问题描述：

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。真是没有白走的弯路。

```
// 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

问题描述：

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)).unwrap());
```

第 6 题 平方和与和的平方之差

问题描述：

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

问题描述：

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，digit

s 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

问题描述：

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

问题描述：

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

问题描述：

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

题目描述：

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

问题描述：

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。

```
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。

```
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。

```
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：

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

还可以写成这样：

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

第 21 题 亲和数

问题描述：

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

问题描述：

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 28123 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

问题描述：

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

问题描述：

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

问题描述：

求小于 2 百万的所有素数之和。

在第 7 题的基础上稍做修改即可，为防止溢出，需要用 u64 保存累计值 sum。

```
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;
```

```
let mut sum: u64 = 0;

const FIRST_PRIME_NUMBER: usize = 2;
for p in FIRST_PRIME_NUMBER..max_number_to_check {
    if prime_mask[p] {
        sum += p as u64;
        let mut i = 2 * p;
        while i < max_number_to_check {
            prime_mask[i] = false;
            i += p;
        }
    }
}
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

writing `&Vec<_>` instead of `&[_]` involves one more reference and cannot be used with non-Vec-based slices.

```
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

问题描述：

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：

$$n^2 + an + b, \text{ 满足 } |a| < 1000 \text{ 且 } |b| < 1000$$

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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	$\boxed{9}$	10	27
42	21	22	23	24	$\boxed{25}$	26
43	44	45	46	47	48	$\boxed{49}$

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

```
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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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：

0.123456789101112131415161718192021...

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

问题描述：

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。

```
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

问题描述：

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。

```
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

问题描述：

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。

```rust
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
}
```

第 684 题 逆数字和

问题描述：

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。

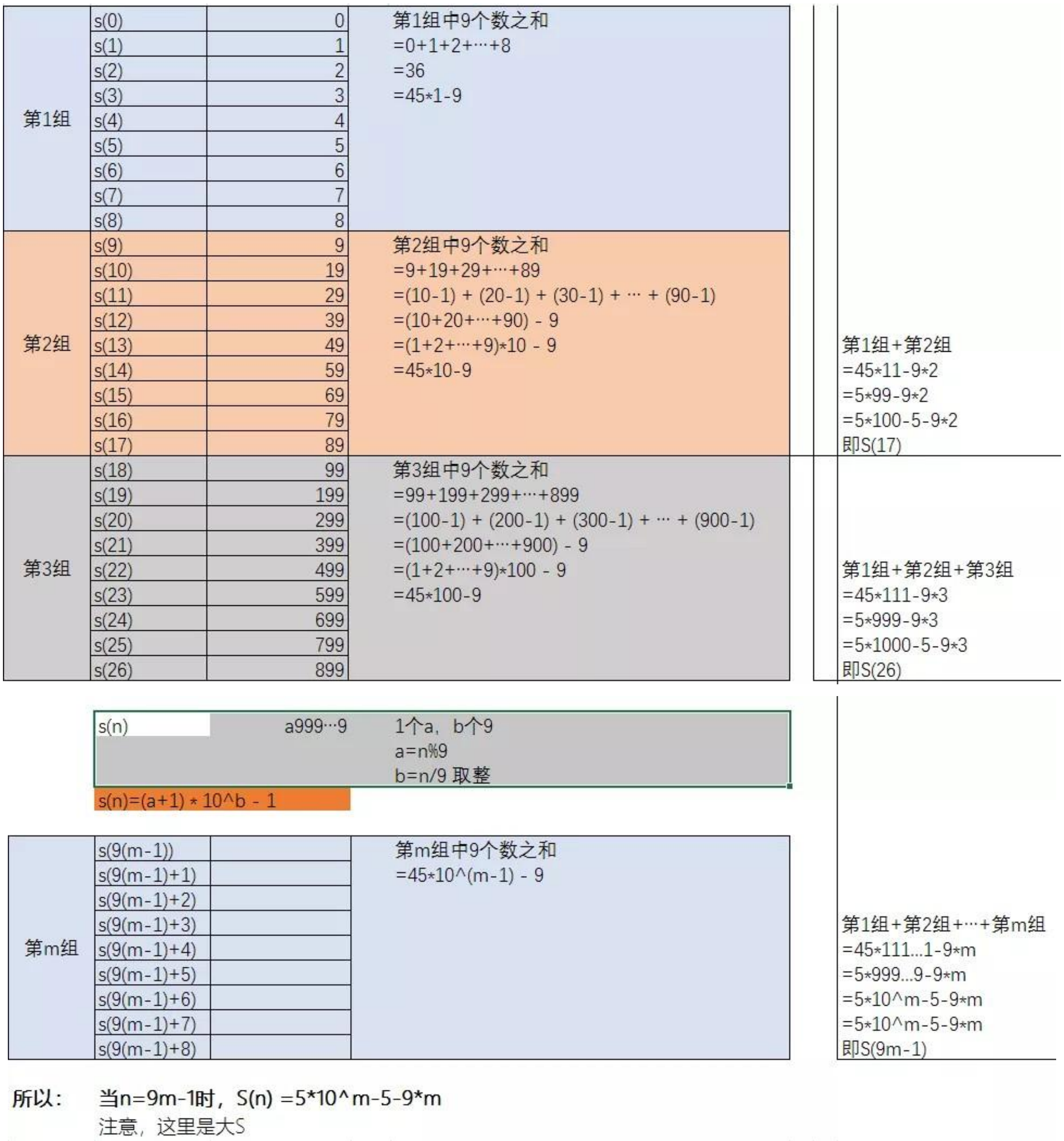
```
fn fs(n: u64) -> u64 {
    let a = n % 9;
    let b = n / 9;
    let mut s = BigUint::from(a + 1);
    for i in 0..b {
        s = s * BigUint::from(10_u64);
    }
    s = s - BigUint::from(1_u64);
    let result = s % BigUint::from(1_000_000_007_u64);
    result.to_string().parse::<u64>().unwrap()
}
```

现在可以比较快地计算出 s(20000)和 S(20000)，但离目标 2880067194370816120 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 mod 1_000_000_007_u64 的办法。

这里要利用费马小定理：

如果 p 是一个质数，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 mod 1_000_000_007_u64 = $10^{(m \bmod 1\_000\_000\_006\_u64)}$ mod 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：

```
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());
}

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

问题描述

$2^7 = 128$，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。

$2^7 = 128$　　$\frac{128}{10} = 12.8 \xrightarrow{取整} 12$

$2^{80} = 1.208\times10^{24}$　　$\frac{1.208\times10^{24}}{10^{23}} = 12.08 \longrightarrow 12$

$2^{90} = 1.237\times10^{27}$　　$\frac{1.237\times10^{27}}{10^{26}} = 12.37 \longrightarrow 12$

考虑 2^n 的10进制表示，

其数字的位数 $d = \lfloor \log_{10} 2^n \rfloor + 1$
$= \lfloor n * \log_{10} 2 \rfloor + 1$

前导的两位数字 $L = \lfloor \frac{2^n}{10^{d-2}} \rfloor$

上面的公式计算过程容易溢出，变换一下：

$L = \lfloor 10^{\log_{10} 2^n - (d-2)} \rfloor$
$= \lfloor 10^{n*\log_{10}2 - d + 2} \rfloor$

将d代入，即有：

$L = \lfloor 10^{n*\log_{10}2 - \lfloor n*\log_{10}2 \rfloor + 1} \rfloor$

设 $t = n * \log_{10} 2$

就有：$L = \lfloor 10^{t - \lfloor t \rfloor + 1} \rfloor$

如果计算前导的三位数，$L = \lfloor 10^{t - \lfloor t \rfloor + 2} \rfloor$

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;
        }
    }
}
```

8 大整数

第 13 题 大整数求和

问题描述：

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 = [
"37107287533902102798797998220837590246510135740250",
"46376937677490009712648124896970078050417018260538",
"74324986199524741059474233309513058123726617309629",
"22918802058777319719839450180888072429661980811197",
// 省略了很多行
"77158542502016545090413245809786882778948721859617",
"72107838435069186155435662884062257473692284509516",
"20849603980134001723930671666823555245252804609722",
"53503534226472524250874054075591789781264330331690",
];
```

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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。

```
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

问题描述：

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

问题描述：

2 的平方根可以用一个无限连分数表示：

$$\sqrt{2} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \dots}}}$$

将连分数计算取前四次迭代展开式分别是：

$$\begin{aligned}
&1 + \frac{1}{2} = \frac{3}{2} = 1.5 \\
&1 + \frac{1}{2 + \frac{1}{2}} = \frac{7}{5} = 1.4 \\
&1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}} = \frac{17}{12} = 1.41666\dots \\
&1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}} = \frac{41}{29} = 1.41379\dots
\end{aligned}$$

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

问题描述：

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。

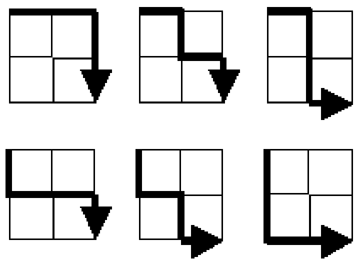
```rust
extern crate num_bigint;
use num_bigint::BigUint;

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

问题描述：

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。

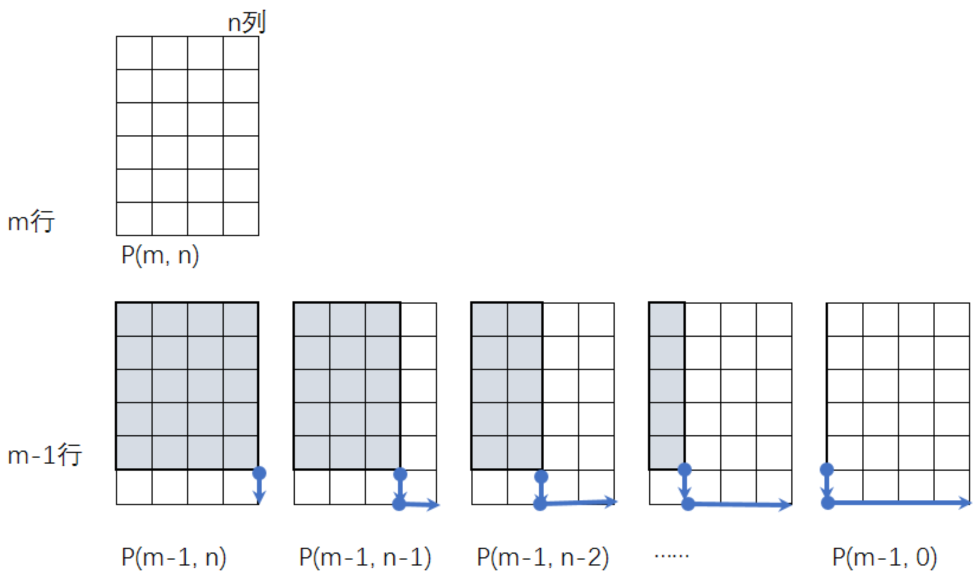


解题思路：

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) + ... + 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：

```
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;
}

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]。

```
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

问题描述：

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。

```
                              75
                            95 64
                          17 47 82
                        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
```

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

问题描述：

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。;o)

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

```rust
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

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

10 日期

第 19 题 数星期日

问题描述：

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。

```rust
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

问题描述：

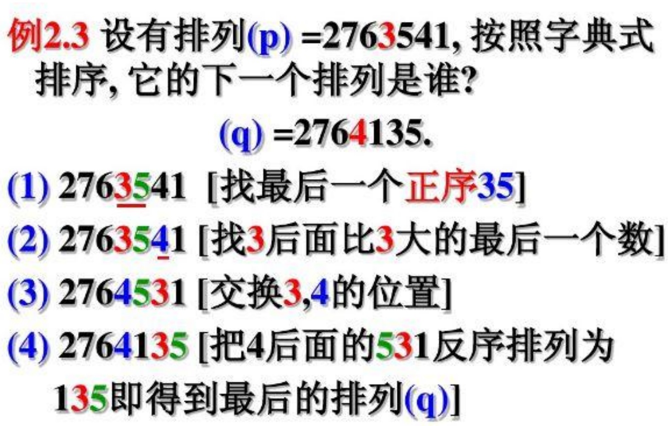
0，1，2，3，4，5，6，7，8 和 9，每个数字用且只用一次，称为全排列，按数值大小排序，求第一百万个数是多少？

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 100 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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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　　0.(076923)

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

再找一个能除尽的：

n = 16

余数

1

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

0.

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

问题描述：

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

问题描述：

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

问题描述：

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

问题描述：

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, …

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

问题描述：

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

问题描述：

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

问题描述：

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。

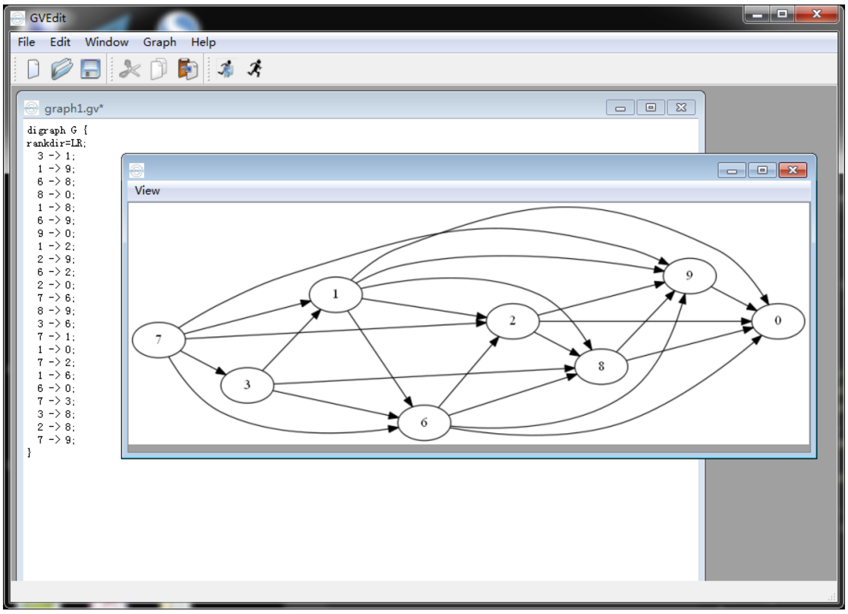
```
println!("digraph G {{");
println!("  rankdir=LR;");
for (a, b) in set {
    println!("  {} -> {};", a, b);
}
println!("}}");
```

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。