CB数学导论

GROUP-CBUPS 东灯
Github @Lampese　Bilibili @东灯哒
2020.11-12 月　第一版

前言

有人说我不懂 CB,有人说我只会拿着计算机算法的那套说辞去说 CB,去批评各位 CBer,但是也确实如此,因为我看到的就是如此。

这本书是对 CB 里面数学算法的研究,今后我有时间还会在图论、数据结构论上做些文章。

正如之前有人说二分是更快的枚举,我真的看到了圈子的水平。

我是一个 OIer,我的四年的时间一直都在搞算法竞赛,要说对计算机算法也有一些了解,这本导论是概述了一些数学算法在我的世界当中的真实实现。

对,没错,我概述的就是在我的世界 CB 当中的实现,我对本书当中所有的算法都进行了实现,数学理论被发明出来一定是有实用价值的,以此书告诉你们计算机算法与 CB 是有一定联系的。

说回数论,数论是数学的皇后,研究的就是整数的性质,数论从古至今,一直被大量的数学家所研究,数学家创造了无数的定理和公式,计算机源自数学,数学是计算机设计的灵魂,也必定是 CB 的灵魂。

目录

Part 1 本书的逻辑

这里会提到一些书里面可能出现的点,所以称为“本书的逻辑”

1.Log 运算问题

接下来的内容当中,凡是遇到 Log 运算不写底数的,均可视为底数为 2,如 Log N=Log2 N。

但是假如我写明指数,如 Log10 N 或 Lg N,那么就不按照底数为 2 理解。

2.时间复杂度

接下来的内容当中,可能涉及到关于“时间复杂度”的概念,想必各位看各种百科看的也是云里雾里,我就通俗的举个例子。

比如你要枚举一块区域,区域长度为 N,也就是你要枚举 N 次,你的时间复杂度就是 O(N),代表对于长度为 N 的区域,你的复杂度就是 O(N),即执行 N 次。

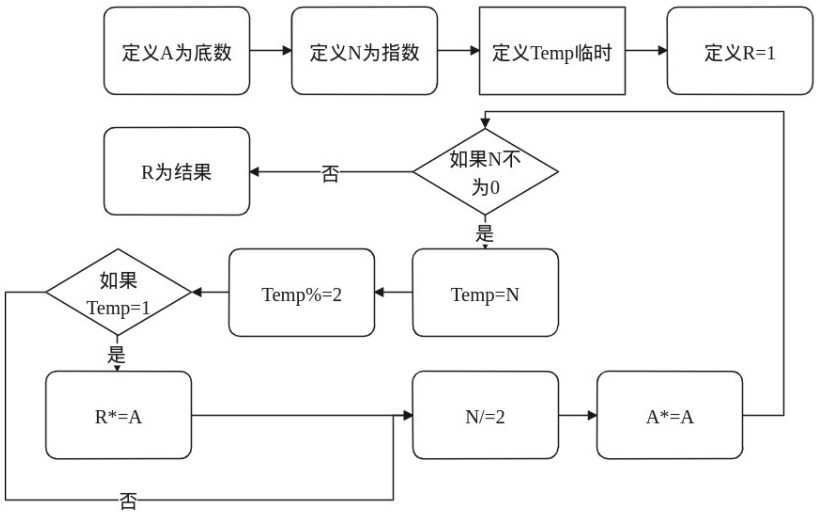
我们熟悉的二分法时间复杂度是多少呢? 是 O(Log N),关于这个各位可以自己演算。

时间复杂度不取最好,只取最坏或全值域。

3.算法实现

本书当中的算法不会用复杂的 CB 指令实现,否则你看不到这么精简的思路和实现,我秉承给出思路,让各位 CBer 来实现。

所有算法的演示都将是使用 CB 可实现的思维导图形式,清晰明了的表述 CB 里面该写什么该打什么,比如下面这张图:



这张图是 Part4 当中不带模快速幂的图例,其实真正的编程语言当中这个过程已经繁琐了,但为了 CB 算法的实现,我把等于、模、除这些在编程语言当中一句就能完成的事情换成了两句,这大幅度方便了 CB 算法的实现。

4.关于后续

这本书采用 MIT 协议稳定在 Github 更新,存储库链接为:

https://github.com/Lampese/CB-Math

除此途径外本书的内容不保证最新,也不对内容的正确性负责,假如您对内容有任何疑问或者有错误需要指出,请直接在存储库内发起新的 Issues。

Part 2 最大公约数(Gcd)和最小公倍数(Lcm)

最大公约数和最小公倍数是各位小学时期就学过的数学概念,不过我想各位还没看过他们两个的数学表达,接下来我简单一提:

Gcd(最大公约数)

有两个正整数 a 与 b,如果有一数 c 满足以下条件

$$c|a \wedge c|b \wedge c \le a \wedge c \le b \wedge c = max\{c|a \wedge c|b\}$$

那么我们认为整数 c 就是 a 和 b 的最大公约数,记作 gcd(a,b)=c。

Lcm(最小公倍数)

有两个正整数 a 与 b,如果有一正数 c 满足以下条件

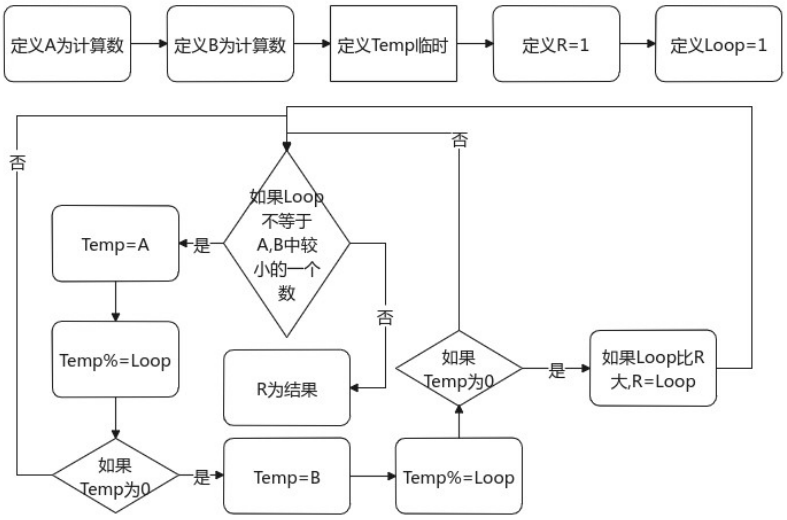
$$a|c \wedge b|c \wedge a \le c \wedge b \le c \wedge c = min\{a|c \wedge b|c\}$$

那么我们认为整数 c 就是 a 和 b 的最小公倍数,记作 lcm(a,b)=c。

在我们在说完数学表达之后,应该如何来求 Gcd 和 Lcm 呢?

朴素 Gcd(枚举 Gcd)

首先根据 Gcd 的定理我们可以得到一种思路,假如给出数字 a 和数字 b,然后我们只需要设一个数字枚举到 a 和 b 当中较小的那个数字,假如 a 和 b 均对一个数字%=0,那么我们就认为这就是符合条件的 c 当中的一个,紧接着将其输出即可,这种方法的时间复杂度是 O(Min(a,b))。



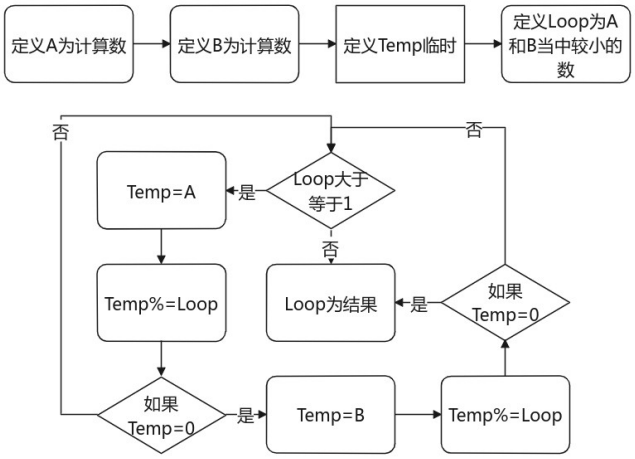
如左图所示,这是朴素求 Gcd 的 CB 过程,下图是 C++ 代码演示。

```cpp
inline int gcd(int a,int b){
  int ans=1;
  for(int i=1;i<=min(a,b);i++){
    if(a%i==0&&b%i==0)
      ans=max(ans,i);
  }
  return ans;
}
```

我们发现朴素 Gcd 事实上还有可以优化的成分,为什么呢?

我们要求的是最大公约数,所以最大公约数从 1 开始枚举必定是慢的,是否可以从 a 和 b 当中较小的那个数开始枚举,遇到第一个公约数就认为它是最大的呢? 答案是,当然可以,而且假如这样做,时间也必定会有提升,这种算法的时间复杂度约为 O(Min(a,b))。

为什么复杂度一样,我还会说速度有所提升呢?因为反向枚举,避免让这个复杂度跑满,时间复杂度只取最坏,所以这两种算法复杂度相同,但实际上反向枚举必然会快。



左图为反向枚举的优化求 Gcd 的过程,这种过程的 C++代码如下。

```
inline int gcd(int a,int b){
  for(int i=min(a,b);i>=1;i--)
    if(a%i==0&&b%i==0)
      return i;
}
```

欧几里德算法(辗转相除法)

上述两种算法的复杂度都不太优秀,遇到非常大的数字的时候复杂度仍然会非常大,会跑很长时间,那么这里就要介绍欧几里德算法求最大公约数。

伟大是数学家欧几里德提出了这个算法,其实这在中国的著作《九章算术》也有所提及“更相减损术”。

首先证明原理:

$$设z|x,z|y,则z|(y-x)$$

$$设z不是x的因子,则z不是x,y-x的公因子$$

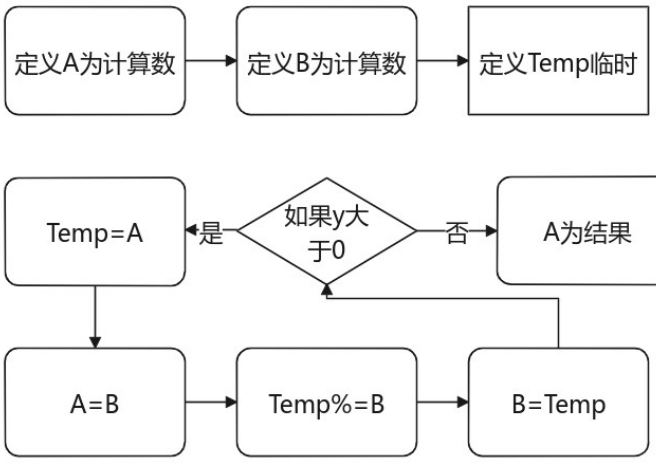
$$设z|x,z不是x,y-x的公因子$$

证明完毕之后,我们可以对这个算法进行分析:

首先由刚才那个定理我们可以得到一句结论:

$$Gcd(a,b)=Gcd(b,a\ mod\ b)$$

由这个结论我们就可以模拟出以下求最大公约数的过程:



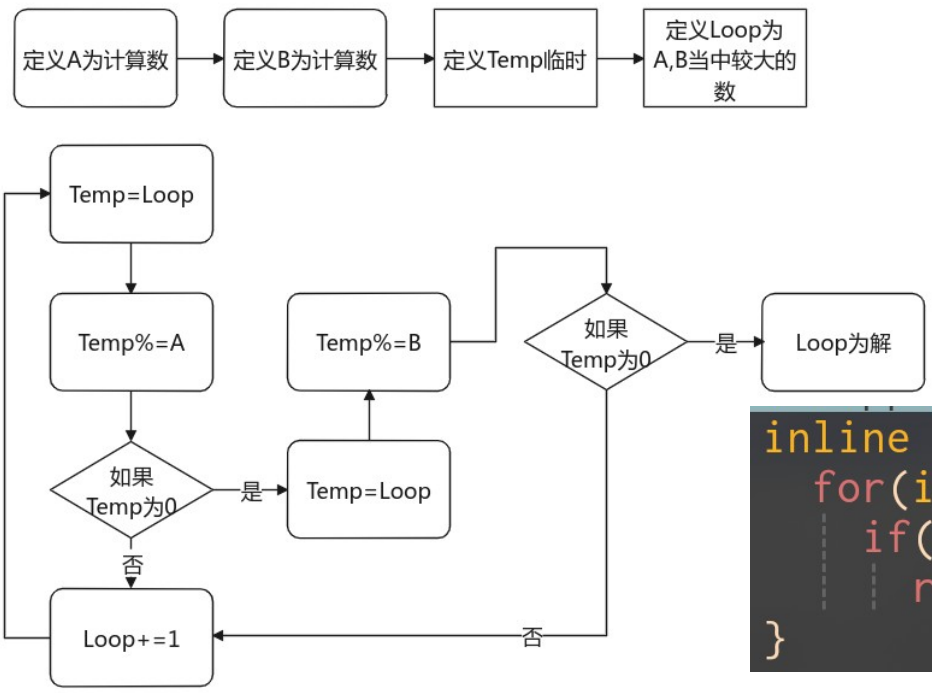
左图为欧几里德的辗转相除法求 Gcd 的过程,下面是 C++的两种求法(循环和递归控制)。

```
inline int gcd(int a,int b){
  int x=a,y=b,temp;
  while(y){
    temp=x;
    x=y;
    y=temp%y;
  }
  return x;
}
inline int gcd_2(int a,int b){
  return y==0?x:gcd_2(b,a%n);
}
```

这种算法的时间复杂度是 O(lg Min(a,b))。

朴素 Lcm(枚举 Lcm)

和刚才的 Gcd 一样,Lcm 也可以进行朴素枚举,从 a 和 b 当中较大的数子开始向上无上界的进行枚举,直到枚举到一个公倍数,就认为是最小。



左图为枚举 Lcm 的 CB 实现流程,下图为 C++实现,该算法复杂度 O(ab)。

```
inline int lcm(int a,int b){
  for(int i=max(a,b);;i++)
    if(i%a==0&&i%b==0)
      return i;
}
```

欧几里德 Lcm(原定理法)

上面的算法复杂度依然并不优秀,所以一定还有更好的算法,我们发现,根据 Gcd 和 Lcm 原本的意义可得:

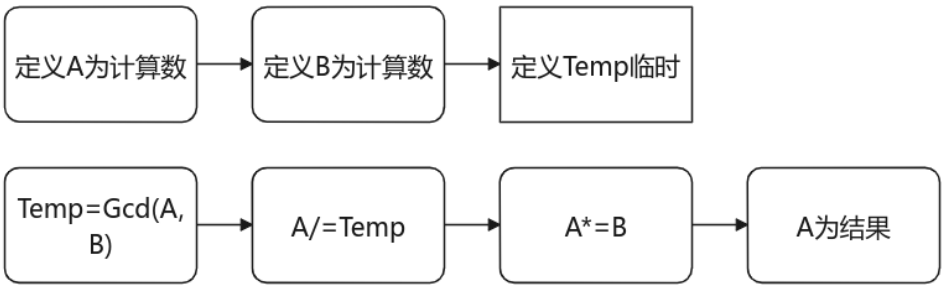
$$Lcm(a,b)=\frac{a*b}{Gcd(a,b)}$$

所以我们只要求出 Gcd(a,b)即可,本算法复杂度其实就是求 Gcd(a,b)的复杂度,即 O(lg min(a,b))。

带入实际问题考虑一下,直接计算 a*b 的值太容易爆范围了,毕竟 CB 里面的数字范围只有 2 的 31 次方,我们把分号上的 b 落下来,一定情况下可以处理的范围更大了:

$$Lcm(a,b)=\frac{a}{Gcd(a,b)}b$$

既然说完了这个方法的证明,那么我们还是来看看实现:



左图是非常简单的快速 Lcm 实现,下图是 C++实现。

```
inline int lcm(int a,int b){
  return a/gcd(a,b)*b;
}
```

在说完 Gcd 和 Lcm 之后,我们就为下一章节奠基,可以使用优秀的拓展欧几里德来求解方程。

Part 3 拓展欧几里德与不定方程

首先看一个方程:

$$ax+bx=Gcd(a,b)$$

第一个问题,这个线性同余方程有解吗?根据裴蜀定理,这个方程一定有解,那我们不妨来尝试证明一下:

裴蜀定理证明和应用

首先,裴蜀有两个推论,即

1.对于正整数 a,b 存在整数 x,y 使得 Gcd(a,b)=ax+by

2.整数 a,b 互质的充分条件是存在整数 x,y 使得 ax+by=1

我们首先开始证明定理 1

已知正整数a与b

设集合$S=\{ax+by|x,y\in\mathbb{Z}\text{并且}ax+by>0\}$

设$d=ax_0+by_0$我们需证明$d=Gcd(a,b)$

设$a=dq+r,r=a\%d$得

$$r=a-dq=a-(ax_0+by_0)q=a(1-x_0q)+b(y_0q)$$

又因为$r\in S\cup\{0\}$并且$0\le r<d$,d是S中最小元素

得$r=0$

所以$d|a$同理可得$d|b$又因为$Gcd(a,b)|d$,所以$d=Gcd(a,b)$

所以$(x+k\frac{b}{d},y-k\frac{a}{d})|k\in\mathbb{Z}$

由于我们并不使用定理 2,所以我们不对此进行证明。

回到我们的问题,对于方程:

$$ax+by=Gcd(a,b)$$

要求给出 a,b 的情况下,求出 x 的两种整数解。

针对问题进行分析:

首先假如 b=0,那么 Gcd(a,b)一定为 a,所以 x=1,y=0 是一组绝对的解。

其他情况如何分析呢?

$$ax+by=Gcd(a,b)$$

$$\because Gcd(a,b)=Gcd(b,a\bmod b)$$

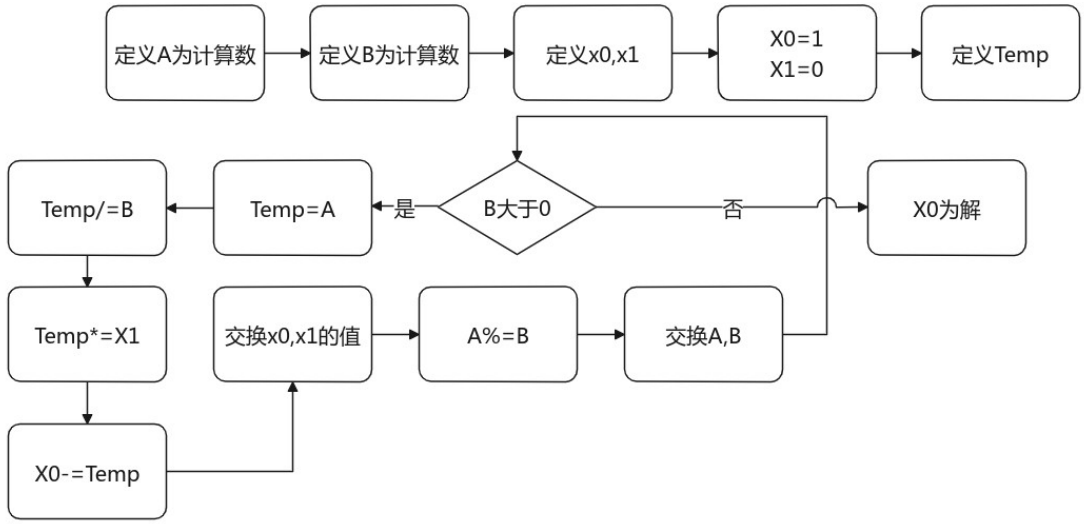
$$\therefore bx_2+(a\bmod b)y_2=Gcd(b,a\bmod b)$$

$$\therefore ax_1+by_1=bx_2+(a\bmod b)y_2$$

$$\because ax_1+by_1=bx_2+(a-\frac{a}{b}b)y2=ay_2+b(x_2-\frac{a}{b}y_2)$$

$$\therefore x_1=y_2,y_1=x_2-\frac{a}{b}y_2$$

由此证毕,我们用 CB 实现一下求解



上图是求线性同余方程的 CB 算法,接下来展示的是对应的 C++算法,分为 exgcd 递归和 exgcd 循环。

```
int exgcd(int a,int b){
    int x0=1,x1=0;
    while(b){
        x0-=a/b*x1;
        swap(x0,x1);
        a %=b;
        swap(a,b);
    }
    return x0;
}
```

```
int exgcd(int a,int b,long long &x,long long &y){
    if(b^1)return x=1,y=0,a;
    int d=exgcd(b,a%b,y,x);
    return y-=a/b*x,d;
}
```

Part 4 快速幂

快速幂实现

快速幂,顾名思义,是快速求幂次方的算法。

正常人求 N^M,通常是把 N*M 次,这样的时间复杂度就是 O(M),明显的,假如次数较多的时候,比如要求 2^31,这时候幂次方算法就会非常慢,那么思考一下,有没有什么办法可以更快地计算幂次方呢?

首先,我们按照一个人人皆知的定论,就是任何一个正整数,都可以被分成若干个 2^n 的形式。

所以快速幂算法就诞生了,快速幂可以每次考虑把指数二分处理,来计算幂次方,举个例子:

我要计算 3^10,那么我们得到

3^10=3*3*3*3*3*3*3*3*3*3

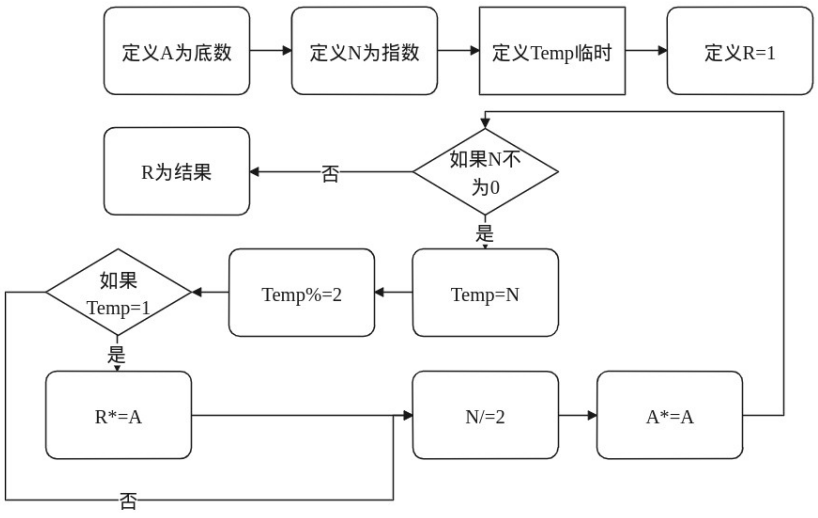
3^10=(3*3)^5

3^10=9^5

9^5=(9^4)*(9^1)

9^5=(6561^1)*(9^1)

这样,我们就在 Log M 的时间内求出了 N^M 的值,所以求 N^M 的快速幂时间复杂度就从 O(M)优化到了 O(Log M),让我们看看 CB 实现。



左图为快速幂算法的 CB 实现,下图为快速幂算法的 C++代码实现。

```
inline long long fastPower(long long base,long long power){
    long long result=1;
    while(power){
        if(power&1)result=result*base;
        power>>=1;
        base=base*base;
    }
    return result;
}
```

```
ull fastpow(ull k,ull mod){
  if(ans[k])return ans[k];
  if(k&1){
    ull temp=fastpow(k>>1,mod);
    ans[k]=(((temp*temp)%mod)<<1)%mod;
  }
  else{
    ull temp=fastpow(k>>1,mod);
    ans[k]=(temp*temp)%mod;
  }
  return ans[k];
}
```

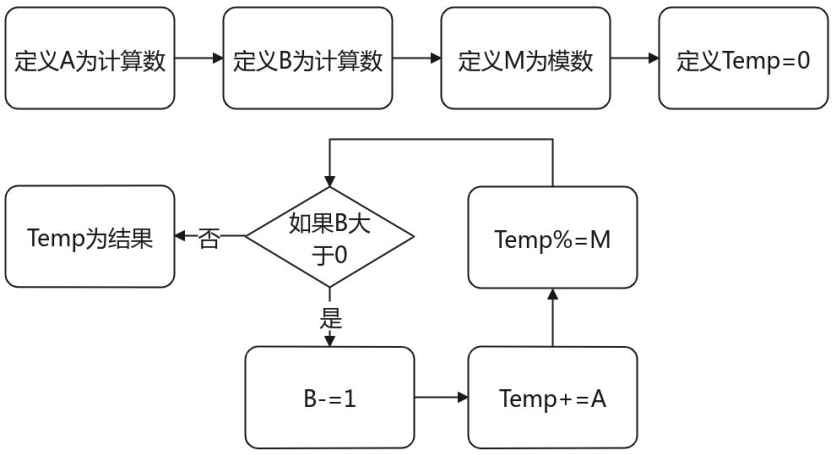
Part 5 大乘法取模

快速乘作用

在我们进行乘法运算的时候,经常会遇到 a*b 超出了数字存储范围,但你又希望他们的结果可以保存下来,这时候你就会对他们的结果进行取模,可是如果我们直接计算(a*b)%m 的话,a*b 就会因为超出范围而被归 0,这时候我们就需要一种叫做快速乘的算法来解决这类问题。

朴素带模乘法

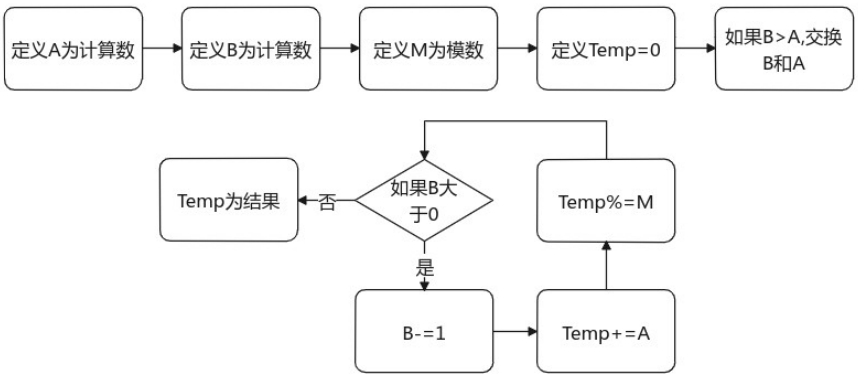
在我们介绍快速乘之前,先介绍一种针对这类问题的普通解法,既然是要算 a*b,那么我们把它转化成加法,每次加完之后进行取模,时间复杂度为 O(Max(a,b))。



左图是朴素带模乘法的流程,下方是这个算法的 C++实现。

```
int p(int a,int b,int m){
  int temp=0;
  for(;b>0;b--){
    temp+=a;
    temp%=m;
  }
  return temp;
}
```

然后用之前做 Gcd 和 Lcm 的老思维思考一下,这里是否也可以优化为,取 a 和 b 中间较小的一个数字进行控制加法,这样复杂度就变成了 O(Min(a,b))。

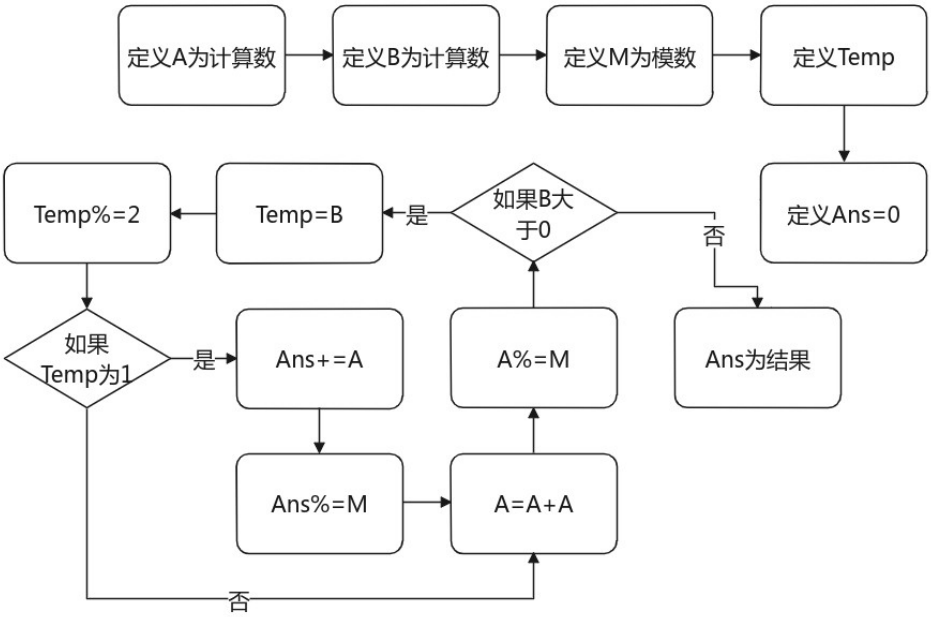


左图是优化后的流程,下方依然是 C++实现。

```
int p(int a,int b,int m){
  int temp=0;
  if(b>a)swap(a,b);
  for(;b>0;b--){
    temp+=a;
    temp%=m;
  }
  return temp;
}
```

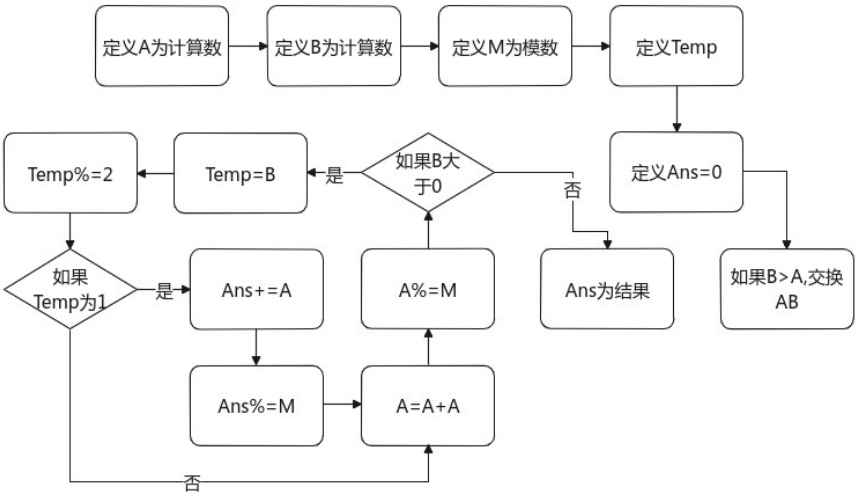
快速乘法

说了这么多,我们进入正题,来讨论一下如何更加优化刚才的两个朴素算法,这里我们可以考虑运用快速幂我们所使用的思想,然后我们发现,加法和求幂是完全一致的,因为同底数幂相乘等于幂次方相加,所以快速幂实际上也运用了这样的思想,很快我们就可以考虑出我们可以把 a 和 b 当中的一个数来进行每次的二分,再维护一个东西来进行每次的+1,流程如下,附 C++代码在右。这样算法的时间复杂度就到了 O(Log Max(a,b))级别。




```cpp
int f(int a,int b,int mod){
  int res=0;
  while(b){
    if(b&1)
    res=(res+a)%mod;
    a=(a+a)%mod;
    b>>=1;
  }
  return res;
}
```

按照刚才优化朴素乘法的思路,我们加一条语句依然可以把复杂度尽量的优化,达到 O(Log Min(a,b))




```cpp
int f(int a,int b,int mod){
  int res=0;
  if(b>a)swap(a,b);
  while(b){
    if(b&1)
    res=(res+a)%mod;
    a=(a+a)%mod;
    b>>=1;
  }
  return res;
}
```

Part 7 素数和 MR 测试

素数,又名质数,概念上是一个大于 1 的正整数,如果除了 1 于他本身之外,正整数范围内没有其他约数,那么就是质数,否则就是和数,伟大的数学家…好吧还是伟大的欧几里德,在《几何原本》里面提到素数有无限个并展开了证明,现代的陈景润院士、王元院士等优秀数学家为素数研究贡献了终身。

素数的分布

目前世界上对素数的分布规律没有定论,素数的分布很不均匀,下面我简单列举一下 1e5 以内的素数分布。

范围	个数
10	4
100	25
1000	168
10000	1229
100000	9592

素数分布只有五大普遍规律,无法进行合理的精确计算,而其中一种规律,就是目前已经广泛用于应用,而且还没有被证伪的——

素数第二对称律(歌德巴赫猜想)

这点接下来的内容当中我也会提到,这里就不多说了。

素数的性质

1.唯一分解定理:若正数 a≥2,那么 a 一定可以表示为若干个素数的乘积。

2.威尔逊定理(草):若 p 为素数,则(p-1)!≡-1(mod p)。

2.威尔逊逆定理:对于一个 p,若(p-1)!≡-1(mod p),那他一定是素数。

3.费马定理:若 p 为素数,a 为任意正正数,并且 a 和 p 互质,那么我们说

$$a^{p-1} \equiv 1 (mod\ p)$$

4.费马小定理:刚才的情况太过于特殊了,所以著名的费马小定理提出:

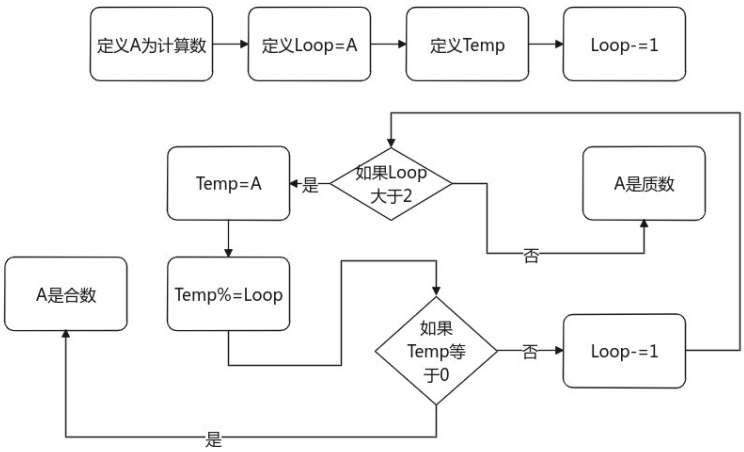
$$a^{p} \equiv a (mod\ p)$$

5.欧拉定理:若 a 与 m 互质,则

$$a^{\varphi(m)} \equiv 1 (mod\ m)$$

素数的判定

是时候来说下素数的判定了,首先根据素数的性质我们可以得到一个算法,即:从 2 开始一直枚举到 n-1,假如没有任何数被 n 整除,那么我们认为 n 是一个素数,这种方法的时间复杂度是 O(n)。



左图为朴素判定算法的 CB 流程,下图为 C++代码实现。

```
bool s(int a){
  for(int i=2;i<a-1;i++)
    if((a%i)^1)
      return false;
  return true;
}
```

刚才的方法非常慢,我们如何优化呢…?

当前数学和程序设计当中使用筛法,比如合数筛,埃氏筛,欧拉筛,州阁筛,杜教筛,Min25 筛,但这些都要耗费大量的内存才能能看到复杂度的成效,所以我们需要一种针对单独的数字筛选。

Miller-Rabin 测试就是一个针对单独素数的合理判定方法,是一个基于费马小定理的速度测试方法,它并不保证筛选出来的一定是一个素数,但是可以保证非常大概率的素数。

这个过程有五步:

(1)计算一个奇数 M,使得 $N=2^r*M+1$ 。

(2)选取随机数 A<N。

(3)对于任何 i<r,若 $A^{2^i*M}\equiv N-1(mod\ N)$,则 N 通过随机数 A 的测试。

(4)或者,若 $A^M\equiv 1(Mod\ N)$,则 N 通过随机数 A 的测试。

(5)让 A 取不同的值对 N 进行 5 次测试,若全部通过那么认为 N 为素数

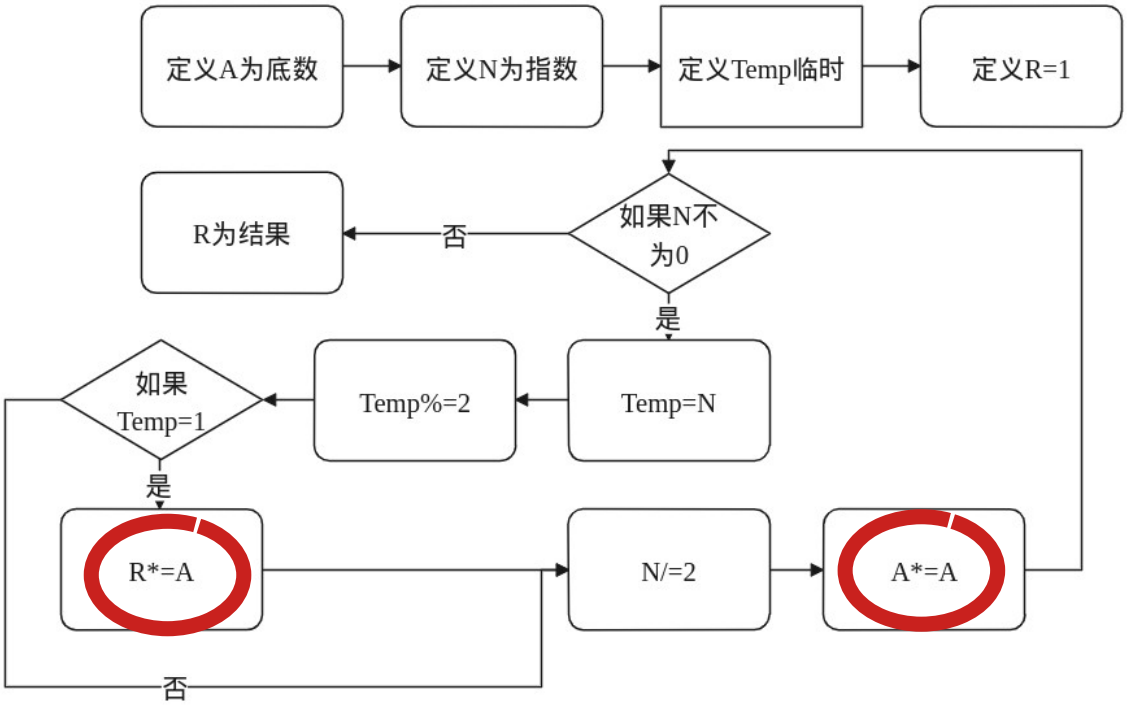
根据定理得到概率,如果 N 通过了 1 次测试,那么 N 不是素数的概率就是 25%,如果按照上述流程通过了 2 次测试,那么 N 不是素数的概率就是 0.125,如果进行 x 次测试,那么 N 不是素数的概率就是 $\frac{1}{4^x}$ 。

根据上述理论,假如我们进行 4 次测试:

$$\frac{1}{4^4}=\frac{1}{256}=0.00390625\approx 0.4\%$$

如果这样的话,N 是素数的概率就超过了 99%,所以来说做四次循环的 MR 测试就是一般合理的,假如有需求可以多做几次,比如做 5 次测试,N 是素数的概率就已经到达了 99.90%,几乎不太可能判定错误。

那么首先,我们需要实现一个带模的快速幂,意思就是对幂出来的结果取模,才能完成这个测试,首先我们在快速幂的流程当中,对每次的结果取模,如图所示,我们针对画红圈的结果进行取模。



问题来了,假如这里面每个乘法的因数的乘积它乘起来本来就爆范围了怎么办呢? 没错,我们用刚才的方法——快速乘来解决大乘法取模的问题!

那么带模的快速幂其实就是把 R*=A 和 A*=A 的步骤更换成“快速乘计算(R*A)%M”和“快速乘计算(A*A)%M”的问题而已,关于这个的 C++代码:

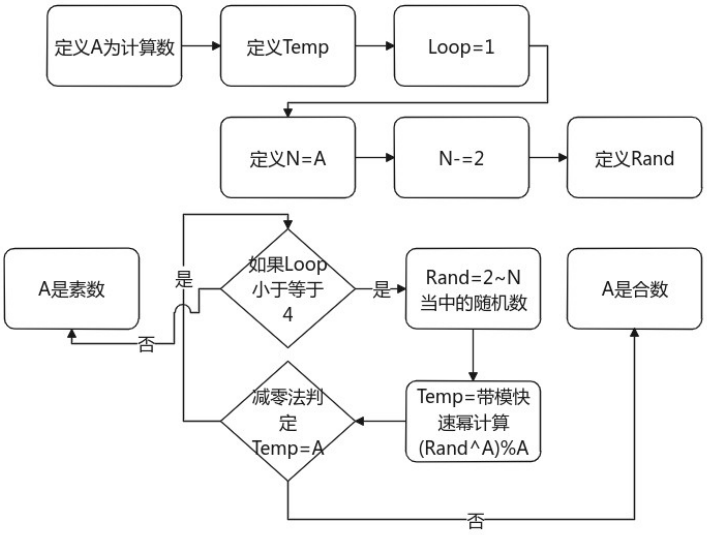
```
int fp(int b,int p,int m){
  //fx指快速乘法
  int r=1;
  while(p){
    if(p&1)r=fx(r,b,m);
    p>>=1;
    b=fx(b,b,m);
  }
  return r;
}
```

```
int fp(int b,int p,int m){
  //常数较小模法
  int r=1;
  while(p){
    if(p&1)r=(r*b)%m;
    p>>=1;
    b=(b*b)%m;
  }
  return r;
}
```

处理完带模快速幂的问题之后,我们才开始实现 MR 测试的主流程,MR 测试当中求测试通过的方法有两种,刚才也说过了,现在我们假设 A 是刚才的随机数,N 是我们要测试的数,在 MR 当中我们只需要以下的测试一种即可:

$$A^n \bmod N = A \vee A^{N-1} \bmod N = 1$$

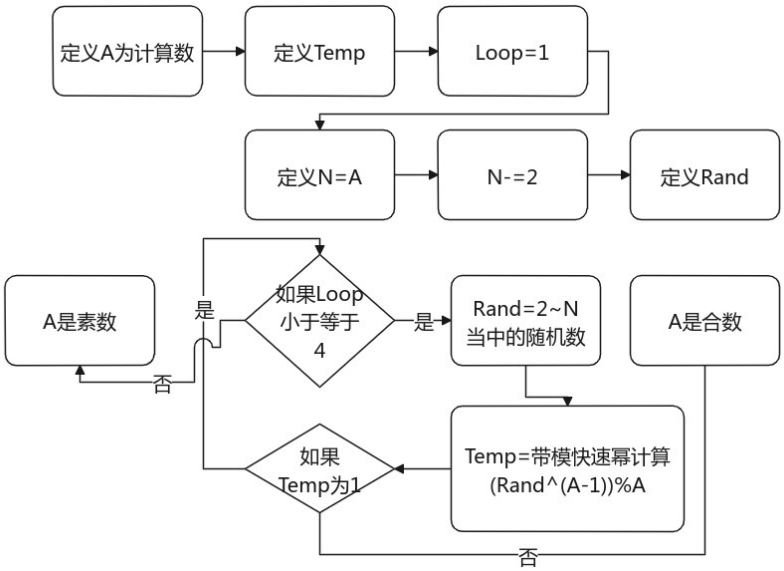
首先是第一种判定方法的实现:




```
bool Miller_Rabin(int n){
  if(n==2)return true;
  for(int i=0;i<count;i++){
    int a=rand()%(n-2)+2;
    if(fp(a,n,n)!=a)
      return false;
  }
  return true;
}
```

这个方法的时间复杂度是十分稳定的 O(4Log N)。

然后是第二种判定方法的实现:




```
bool Miller_Rabin(int n){
  if(n==2)return true;
  for(int i=0;i<count;i++){
    int a=rand()%(n-2)+2;
    if(fp(a,n-1,n)!=1)
      return false;
  }
  return true;
}
```

这个方法的时间复杂度也是 O(4Log N)。

Part 7 欧拉函数和卡特兰数

欧拉函数

欧拉函数其实在刚才也有提及,就在欧拉定理当中:

费马小定理:刚才的情况其实是符合费马小,所以都有

$$a^p \equiv a(mod\ p)$$

5.欧拉定理:若 a 与 m 互质,则

$$a^{\varphi(m)} \equiv 1(mod\ m)$$

欧拉函数的意义是:φ(n)表示小于等于 n 的数中与 n 互质的数的数目。

φ(8)=4,其中和 n 互质的数字有(1,3,5,7)。

那么关于 φ(n)有以下引理:

(1)如果 n 是一个素数 p,那么 $\varphi(p)=p-1$

(1)的证明:傻子都知道。

(2)如果 n 是一个素数 p 的幂次 p^a ,那么 $\varphi(p^a)=p^{a-1}*(p-1)$

(2)的证明:

首先我们知道一点:比 p^a 小的正整数一定有 p^a-1 个

那么我们也知道了所有 p 的因数可以为 $p^{a-1}-1$,而且这些数字全部都和 p^a 互质,所以 $\varphi(p^a)=p^a-1-(p^{a-1}-1)$ 。

(3)如果 n 为任意两个互质的数 a,b 的积,则 $\varphi(a*b)=\varphi(a)*\varphi(b)$

(3)的证明:

在比 a*b 小的 a*b-1 的整数中,只有那些既和 a 互质,又和 b 互质的数,才会和 a*b 互质,而分别有 φ(a)和 φ(b)个,那么 $\varphi(a*b)=\varphi(a)*\varphi(b)$ 。

(4)设 $n=p_1^{a1}*p_2^{a2}*...*p_k^{ak}$ 来表示 N 的素数幂乘积,那么

$$\varphi(n)=n*(1-\frac{1}{p_1})*(1-\frac{1}{p_2})*...*(1-\frac{1}{p_k})$$

(4)证明:

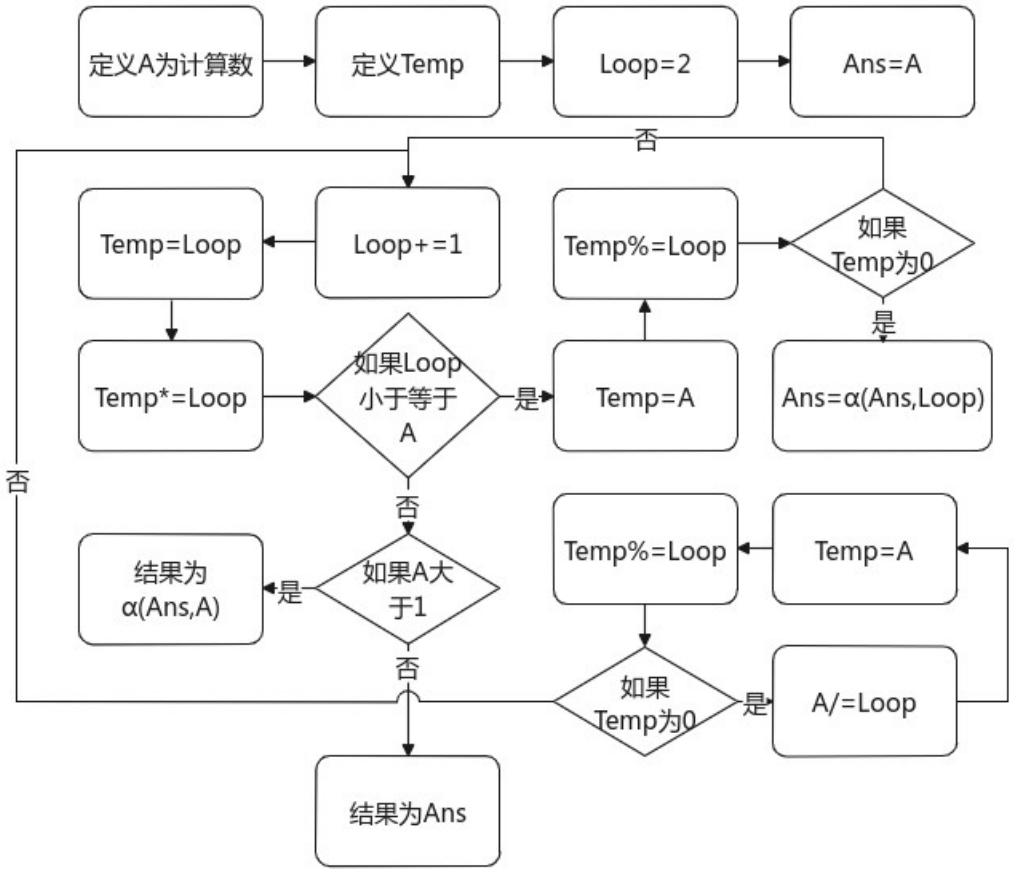
$$\varphi(n)=\varphi(p_1^{a1})*\varphi(p_2^{a2})*...*\varphi(p_k^{ak})=n*(1-\frac{1}{p_1})*(1-\frac{1}{p_2})*...*(1-\frac{1}{p_k})$$

那么根据刚才的定理 4,我们得出 φ(n)的公式:

$$\varphi(n)=n\prod_{i=1}^{k}(1-\frac{1}{p_1})$$

假如用 CB 来求这个公式:

首先,我们把计算 x/y*(y-1)的运算符定义为 α(x,y),因为这太好求了,况且我也要省点纸…上图为 CB 当中此流程的实现,下图为 C++当中对这个算法的实现,此算法复杂度为稳定的 O(φ(n))。




```
int eular(int n){
    int ans=n;
    for(int i=2;i*i<=n;++i)
        if((n%i)^1){
            ans=ans/i*(i-1);
            while((n%i)^1) n/=i;
        }
    return n>1?ans/n*(n-1):ans;
}
```

```
void euler(){//打表欧拉
    for(int i=2;i<maxn;i++){
        if(E[i])continue;
        for(int j=i;j<maxn;j+=i){
            if(!E[j])E[j]=j;
            E[j]=E[j]/i*(i-1);
        }
    }
}
```

卡特兰数(卡塔兰数)

卡特兰数是数论中地位仅次于斐波那契数列的一个数列,实用价值极高,广泛用于数论计数问题和组合数学,应用地点就可以自己查阅和学习,在实际应用当中就可以灵活应用。它的前几项是 1,2,5,14,42,132,429.....

卡特兰数有大量可直接使用的通项公式,如两个递推公式:

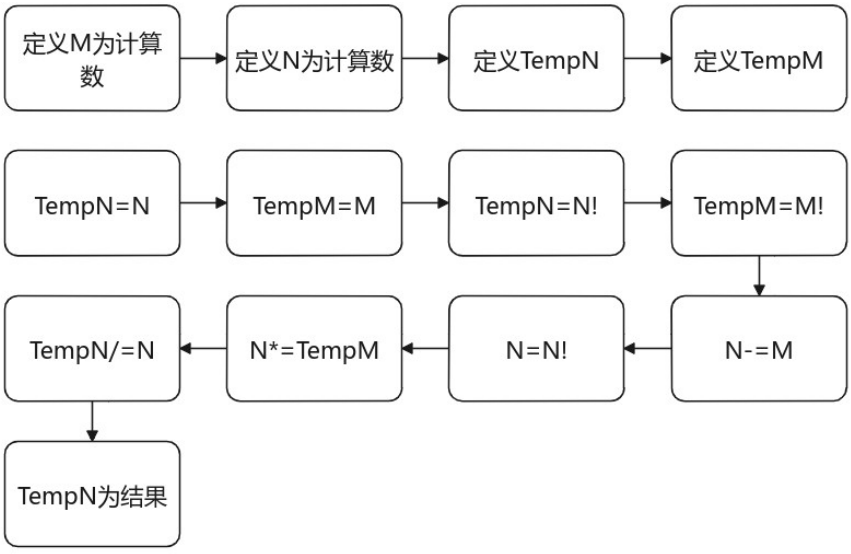
Sum 递推公式:
$$f(n)=\sum_{i=1}^{n-1} f(i)*f(n-i-1)$$

乘法计算公式:
$$f(n)=\frac{f(n-1)*(4*n*2)}{n+1}$$

通项公式也有两个:
$$f(n)=\frac{C_{2n}^{n}}{n+1}$$
$$f(n)=C_{2n}^{n}-C_{2n}^{n-1}$$

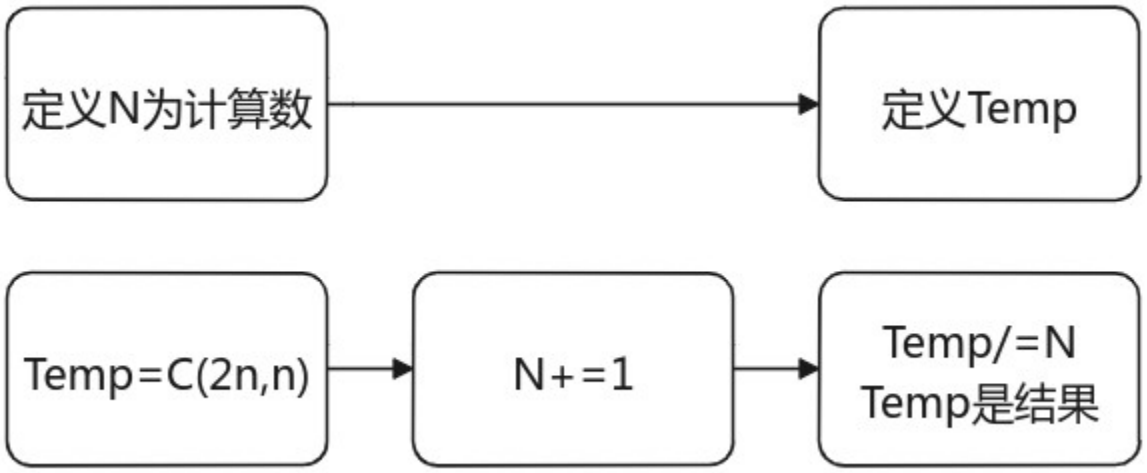
上述公式都可以求出卡特兰数,但是由于前两种方法对于 CB 来说过于苛刻,因为 CB 无法实现大量数据的存储,自然也是非常依赖于通项公式的,那么我们就终点来介绍下面两种,首先我们要了解组合数学当中的 C:
$$C_n^m=\frac{A_n^m}{m!}=\frac{n!}{m!(n-m)!}$$

随后我们就出了一个 C 的计算流程,很快啊!

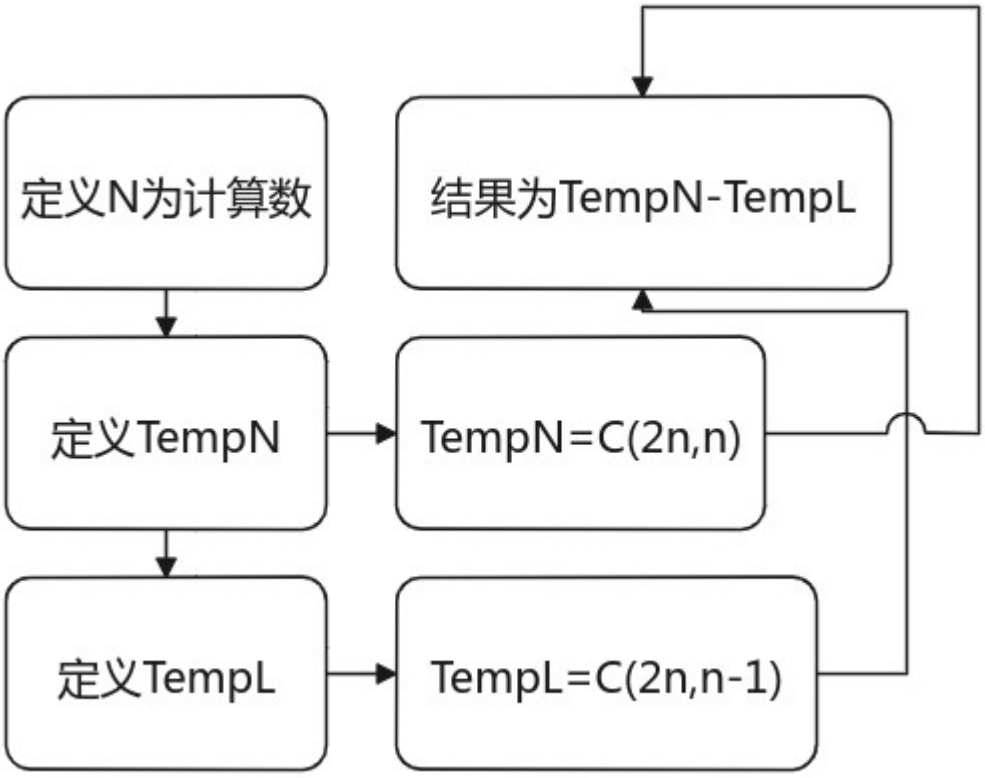


计算流程出来之后那自然是中华传统算法的点到为止,直接开始进行计算,这块的复杂度和 C++代码摆出来也没意义了吧:

如图是第一个通项公式计算的方法。



如图是第二个通项公式的计算方法。



Part 8 任意多边形面积

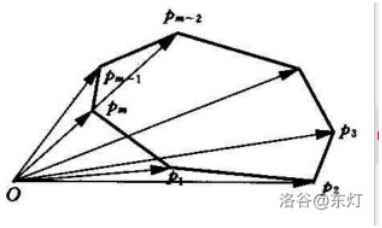
在我卑微的数论计算几何知识当中,求任意多边形面积的公式一直在我心里是非常实用的,但是因为 CB 流程当中实现又过于繁琐,所以我只为各位写出证明过程,实现就作为本书的结束,敬请各位学成的 CBer 自由实现!

证明有微积分证法和向量积证明方法,我挑选向量积为各位进行证明。

设Ω是m边形(如图1)，顶点$p_k(k=1,2,...m)$沿边界Ω正方向排列，$p_{m+1}=p_i$，坐标依次为

$$(x_1,y_1),(x_2,y_2),...,(x_m,y_m)$$

建立多边形Ω的多边形区域向量图。



由图可知坐标原点与多边形任意相邻的两个顶点构成一个三角形，而三角形的面积可由三个顶点构成的两个平面向量的外积求得，当$\vec{p_k}\vec{p_{k+1}}$为正方向时，外积值为正。

设向量

$$\vec{op_k}=\{x_k,y_k,0\}$$

$$op_{k+1}=\{\vec{x_{k+1}},y_{k+1},0\}$$

由向量外积计算得

$$\vec{op_k}*\vec{op_{k+1}}=\{0,0,x_ky_{k+1}-x_{k+1}y_k\}$$

所以任意多边形面积公式为

$$S_\Omega=\sum_{k=1}^{m}S_{\triangle p_kp_{k+1}}=\frac{1}{2}|\sum_{k=1}^{m}\vec{op}_k*\vec{op}_{l+1}|=\frac{1}{2}\sum_{k=1}^{m}(x_ky_{k+1}-x_{k+1}y_k)$$

***此证明过程摘自我自己的 Markdown 编辑器**

The END