ACM-template

langman

April 11, 2018

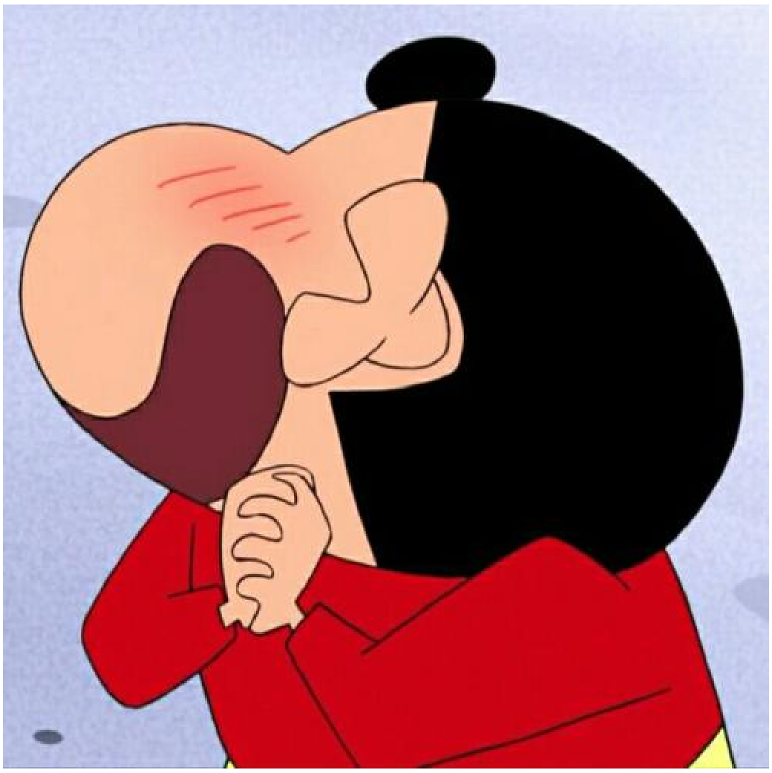

Figure 1: stay hungry stay foolish

Contents

1 头文件

```
#include<iostream>
#include<cstdio>
#include<cstring>
#include<string>
#include<algorithm>
#include<queue>
#include<stack>
#include<vector>
#include<cmath>
#include<set>
#include<cstdlib>
#include<functional>
#include<climits>
#include<cctype>
#include<iomanip>
using namespace std;
typedef long long ll;
#define INF 0x3f3f3f3f
const int mod = 1e9+7 ;
#define clr(a,x) memset(a,x,sizeof(a))
#define cle(a,n) for(int i=1;i<=n;i++) a.clear();
const double eps = 1e-6;
int main()
{
  freopen("in.txt","r",stdin);
  freopen("out.txt","w",stdout);
  //舒服了
  return 0;
}
```

2 图论

2.1 二分图

判断是否二分图

```
int color[N], graph[N][N];

//0为白色，1为黑色
bool bfs(int s, int n) {
    queue<int> q;
    q.push(s);
    color[s] = 1;
    while(!q.empty()) {
        int from = q.front();
        q.pop();
        for(int i = 1; i <= n; i++) {
            if(graph[from][i] && color[i] == -1) {
                q.push(i);
                color[i] = !color[from];//染成不同的颜色
            }
            if(graph[from][i] && color[from] == color[i])//颜色有相同，则不是二分图
                return false;
        }
    }
    return true;
}
```

求出最大匹配数

```
#define N 202
int useif[N];    //记录y中节点是否使用 0表示没有访问过，1为访问过
int link[N];   //记录当前与y节点相连的x的节点
int mat[N][N]; //记录连接x和y的边，如果i和j之间有边则为1，否则为0
int gn,gm;    //二分图中x和y中点的数目
int can(int t)
{
    int i;
    for(i=1;i<=gm;i++)
    {
       if(useif[i]==0 && mat[t][i])
       {
           useif[i]=1;
           if(link[i]==-1 || can(link[i]))
           {
              link[i]=t;
```

```
                return 1;
            }
        }
    }
    return 0;
}
int MaxMatch()
{
    int i,num;
    num=0;
    memset(link,0xff,sizeof(link));
    for(i=1;i<=gn;i++)
    {
      memset(useif,0,sizeof(useif));
       if(can(i)) num++;
    }
    return num;
}
```

2.2 并查集

```
int par[maxn];
int rank[maxn];

void init()
{
  for(int i = 1;i<=n;i++)
  {
    par[i] = i;
    rank[i] = 0;
  }
}

int find(int x)
{
  return x==par[x]?x:find(par[x]);
}
```

2.3 最短路

两种算法 但是要注意dijkstra无法处理负边的情况

2.3.1 dijkstra

需要注意的在于 可以更优化 我没写了 而且需要注意重边的情况

```
//dijkstra 算法
//无负权
int map[2005][2005];//记录路径 注意双向
int dp[2005];//单源最短路径记录
bool vis[2005];//记录是否用过
int N;//顶点数
void dijkstra(int s)
{
  clr(dp,INF);
  clr(vis,0);
  dp[s] = 0;
  while(true)
  {
    int v = -1;
    for(int u = 1;u<=N;u++)//从没用过的点中找一个距离最小的顶点
    {
      if(!vis[u] && (v==-1 || dp[u]<dp[v]) )
      v = u;
    }
    if(v == -1) break;
    vis[v] = true;
    for(int i = 1;i<=N;i++)
    {
      dp[i] = min(dp[i],dp[v]+map[v][i]);
    }
  }
}
```

2.3.2 spfa

需要注意的是怎么建边 双向边?

```
// Bellman Ford 存在最短路 这个比较难看感觉
//这个可以有负权
//但是  spfa 是在bellmen Ford 的基础上的加强
//这里用的是用前向星的方法去建图
//这里不用判重边的还是很舒服
int N,M;
```

```
int cnt;
struct edge{
int to,Next,w;
}E[maxn];
int pre[maxn],dp[maxn];//pre 路径结点 dp 最短路
bool vis[maxn];
int in[maxn];//这个的作用在于处理进去过多少次 就能看出是不是存在负环
void addedge(int x,int y,int z)
{
E[++cnt] .to = y;
E[cnt].Next = pre[x];
E[cnt].w = z;
pre[x] = cnt;
return;
}
bool spfa(int s)//这个算法还能判断是否存在负环
{
int i,t,temp;
queue<int>Q;
clr(vis,0);
clr(dp,INF);
clr(in,0);

Q.push(s);
vis[s] = true;
dp[s] = 0;

while(!Q.empty())
{
  t = Q.front();Q.pop();vis[t] = false;
  for(i = pre[t];i;i=E[i].Next)
  {
    temp = E[i].to;
    if(dp[temp] > dp[t]+E[i].w)
    {
      dp[temp] = dp[t]+E[i].w;
      if(!vis[temp])
      {
        Q.push(temp);
        vis[temp] = true;
        if(++in[temp]>N) return false; //负环判定关键
      }
    }
  }
```

```
}
return true;
}
```

2.3.3 Flody

这个就不写了,一个小dp

2.4 最小生成树

这是个什么玩意呢 图里面是吧,找到n-1条边使得生成一颗树,然后他的边权之和最小

```
//prime 算法
//还有一个不想去写了 没有这个必要
//注意重边
// 还有这个算法 在树生成不起来的情况下 需要特判一下
// dfs一遍就行 看是否全联通
int map[maxn][maxn];
int dp[maxn];
int vis[maxn];
int N;
int prime()
{
  clr(dp,INF);
  clr(vis,0);
  dp[1] = 0;
  int res = 0;
  while(true)
  {
    int v = -1;
    for(int u = 1;u<=N;u++)
    {
      if(!vis[u] && (v==-1 || dp[u]<dp[v])) v = u;
    }
    if(v == -1) break;
    vis[v] = 1;
    res += dp[v];
    for(int u = 1;u<=N;u++)
    {
      dp[u] = min(dp[u],map[v][u]);
    }
  }
  return res;
}
```

2.5 最大流

2.5.1 Dinic

板子先存着,坑定用的着

```
//最大流 dinic算法
//记得有时候要建双向边
const int MAXN = 1000;
struct edge{int to,cap,rev;};    //用边来存图
vector<edge>G[MAXN];             //图的链接表表示
int level[MAXN];                 //顶点到源点的距离标号
int iter[MAXN];                  //当前弧在其之前的边已经没有用了

void addedge(int from,int to,int cap)     //为图加一条从from到to的容量为cap的边
{
  G[from].push_back((edge){to,cap,(int)G[to].size()});
  G[to].push_back((edge){from,0,(int)G[from].size()-1});
}
void bfs(int s)                  //bfs计算从源点出发的距离标号
{
  clr(level,-1);
  queue<int>que;
  level[s] = 0;
  que.push(s);
  while(!que.empty())
  {
    int v = que.front();
    que.pop();
    for(int i = 0;i<G[v].size();i++)
    {
      edge &e = G[v][i];
      if(e.cap>0 && level[e.to]<0)
      {
        level[e.to] = level[v]+1;
        que.push(e.to);
      }
    }
  }
}
int dfs(int v, int t,int f)              //通过dfs寻找增广路
{
  if(v == t) return f;
  for(int i = iter[v] ;i<G[v].size();i++)
  {
```

```
    edge &e = G[v][i];
    if(e.cap > 0 && level[v] < level[e.to])
    {
      int d = dfs(e.to,t,min(f,e.cap));
      if(d>0)
      {
        e.cap -=d;
        G[e.to][e.rev].cap += d;
        return d;
      }
    }
  }
  return 0;
}
int max_flow(int s,int t)    //从 s 到 t 的最大流
{
  int flow = 0;
  while(true)
  {
    bfs(s);
    if(level[t] < 0 ) return flow;
    clr(iter,0);
    int f;
    while((f = dfs(s,t,INF)) > 0 )
    flow += f;
  }
}
```

3　常用数据结构

3.1　STL

```
//头文件
#include<queue>
#include<stack>
#include<string>
#include<set>
#include<map>
struct cmp1{
    bool operator ()(int &a,int &b){
        return a>b;//最小值优先
    }
};
int main()
{
  //优先队列
  priority_queue<int,vector<int>,cmp1>que1;
  que1.push(i);que1.top();que1.pop();que1.empty();que1.clear();//进，顶，出，空，删，
  //排序
  sort(shu,shu+n,cmp);//范围，排序方式
  //容器
  vector<int>q2;vector<int> q2_1{1,2,3,4};
  q2.push_back(i);q2.top();q2.pop();q2.empty();q2.clear();q2.size();
  //string一些初始化方法
  char shu[100];
  char s1[] = {"dadaa"};
  string a("sssss"),string s = "qqqq",string s1(s,3,4);//s1是s从下标3开始4个字符的拷
  string s2(s,2);//从s2的第二个字符开始拷贝
  string s3(shu,3);//复制字符串cs的前3个字符到s当中

}
```

3.2　树状数组

适用于的方面是，单点更新，多次查询。

```
//一维的
int tree[maxn];
int lowbit(int t)
{
  return t&(-t);
}
void add(int x,int y)
```

```
{
  for(int i=x;i<=n;i+=lowbit(i))
  tree[i]+=y;
}
int getsum(int x)
{
  int ans=0;
  for(int i=x;i>0;i-=lowbit(i))
    ans+=tree[i];
  return ans;
}
//二维的，自己感觉一下，需要容斥一下
int data[MAX][MAX], n;
int lowbit(int x) {
    return x&-x;
}
void Add(int x, int y, int w) {
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= n; j += lowbit(j)) {
            data[i][j] += w;
        }
    }
}
int Sum(int x, int y) {
    int ans = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ans += data[i][j];
        }
    }
    return ans;
}
```

3.3 前缀和

适用于多次区间更新，一次查询，代码就不搞上去了。

3.4 线段树

现在的我对于这方面还不够熟悉，先留个板子

```
long long int node[mod];//注意数据大小，小心爆了，本渣渣就爆了好久
void build(int l,int r,int root,int n)
{
    int mid;
```

```
    if(l==r)
    {
        scanf("%lld",&node[root]);
        return;
    }
    mid=(l+r)>>1;
    build(l,mid,root<<1,n);
    build(mid+1,r,root<<1|1,n);
}
void update(int L,int R,long long int add,int l,int r,int root)
{
    if(L<=l && R>=r)
    {
        node[root]+=add;
        return ;
    }
    int mid=(l+r)>>1;
    if(L<=mid) update(L,R,add,l,mid,root<<1);
    if(mid<R) update(L,R,add,mid+1,r,root<<1|1);
}
void query(int l,int r,int root,long long int k)
{
    if(l==r)
    {
        if(l==1)printf("%lld",node[root]+k) ;
        else printf(" %lld",node[root]+k) ;
        return ;
    }
    int mid=(l+r)>>1;
    query(l,mid,root<<1,k+node[root]);
    query(mid+1,r,root<<1|1,k+node[root]);
}
```

3.5 高精度

我个人习惯使用Java。

```
import java.util.*;
import java.math.*;
import java.util.Scanner;

public class Main {
        public static void main(String[] args) {
         Scanner cin = new Scanner(System.in);
         while(cin.hasNext())
```

```
    {
      int n = cin.nextInt();//输入的方式比较的麻烦
      BigInteger a = new BigInteger(); //里面是字符串
      BigDecimal b = new BigDecimal();
      //接下来是比较常见的函数以及用法。
      a = a.add(a);a = a.multiply(a);
      BigInteger c = new BigInteger.valueOf(n);//类型转换
      subtract();//减法
      multiply();
      divide();     //相除取整
      remainder(); //取余
      pow();    a.pow(b)=a^b
      gcd();    //最大公约数
      abs(); //绝对值
      negate(); //取反数
      mod(); a.mod(b)=a%b=a.remainder(b);
    }
  }
}
```

4 数学方面

4.1 三个特别的数

4.1.1 Fib 数列

$$f(x) = f(x-1) + f(x-2)$$

$$f(0) = 0, f(1) = 1$$

4.1.2 卡特兰 数

$$\sum_{i=1}^{n} f_i * f_{n-i} = f_n$$

$$h(n) = C_{2n}^{n} - C_{2n-1}^{n}$$

注意它这个数字来自于什么情况。

4.1.3 斯大林公式

$$\sqrt{2 * PI * n} * (\frac{n}{e})^n = n!$$

4.2 数论

第一个自然是最基础的欧几里得算法，欧几里得算法的用处有很多，求最大公倍数，解方程，很多。在后面的过程会把一些常见的板子列出来，一般来说这些板子 都已经经过验证，但是不好说对吧。简单题我们可以通过一些模板直接得出答案，但是怎么说，这些对于难题 估计只能算工具，重要的是如何转换。

4.2.1 素数

素数筛法

```
long long su[MAX],cnt;
bool isprime[MAX];
void prime()
{
    cnt=1;
    memset(isprime,1,sizeof(isprime));//初始化认为所有数都为素数
    isprime[0]=isprime[1]=0;//0和1不是素数
    for(long long i=2;i<=MAX;i++)
    {
        if(isprime[i])
            su[cnt++]=i;//保存素数i
        for(long long j=1;j<cnt&&su[j]*i<MAX;j++)
        {
            isprime[su[j]*i]=0;//筛掉小于等于i的素数和i的积构成的合数
        }
    }
}
```

4.2.2 梅森素数

一个知识点吧，m是一个正整数，且2^m-1为素数，那么m一定为素数。
如果m是一个素数，$M_p=2^p-1$是梅森数
如果p是一个素数，并且$M_p=2^p-1$也是素数，那么称M_p为梅森素数
对梅森素数的判定是一个算法：
Lucas-Lehmer: $r_k \equiv r_{k-1}-2 \mod M_p \quad r_1=4$ 当且仅当 $r_{p-1}\equiv 0 \mod M_p$

4.2.3 欧几里得

```
//欧几里得求最大公因数
int gcd(int a,int b)
{
   return b == 0?a:gcd(n,a%b);
}
//扩展欧几里得算法
```

```
// a*x + b*y = gcd(a,b) 这个是用于 x 和 y
// 不能肯定 x, y的正负
int exgcd(int a,int b,int &x,int &y)
{
   if(b == 0)
   {
     x = 1;
     y = 0;
     return a;
   }
   int r = exgcd(b,a%b,x,y);
   int t = y;
   y = x - (a/b)*y;
   x = t;
   return r;
}
```

然后是基于这个定理得出的一个定理，中国剩余定理

```
int n;
ll a[maxn],m[maxn];//a余数 m除数
ll gcd(ll a,ll b) {
    return !b?a:gcd(b,a%b);
}
ll exgcd(ll a,ll b,ll &x,ll &y) {
    if (!b) {
        x=1,y=0;
        return a;
    }
    ll d=exgcd(b,a%b,y,x);
    y-=(a/b)*x;
    return d;
}
ll inv(ll a,ll m) {
    ll x,y;
    ll d=exgcd(a,m,x,y);
    if (d==-1) return -1;
    return (x%m+m)%m;
}
bool merge(ll a1,ll m1,ll a2,ll m2,ll &a3,ll &m3) {
    ll d=gcd(m1,m2),c=a2-a1;
    if (c%d) return false;
    c=(c%m2+m2)%m2,
    c/=d,m1/=d,m2/=d,
    c*=inv(m1,m2),
```

```
    c=(c%m2+m2)%m2,
    c=(c*m1*d)+a1;
    m3=m1*m2*d;
    a3=(c%m3+m3)%m3;
    return true;
}
ll crt() {
    ll a1=a[1],m1=m[1];
    for (int i=2;i<=n;i++) {
        ll aa,mm;
        if (!merge(a1,m1,a[i],m[i],aa,mm)) return -1;
        a1=aa,m1=mm;
    }
    return (a1%m1+m1)%m1;
}
```

4.2.4 乘法逆元

思想是通过扩展欧几里得来得出，如果缘分到了，那么 还能用费马小定理来解

```
//扩展欧几里得
int extgcd(int a, int b, int& x, int& y)
{
    int d = a;
    if(b != 0){
        d = extgcd(b, a % b, y, x);
        y -= (a / b) * x;
    }else {
        x = 1;
        y = 0;
    }
    return d;
}
int mod_inverse(int a, int m)
{
    int x, y;
    extgcd(a, m, x, y);
    return (m + x % m) % m;
}
//费马小定理
//模p，p为素数
return quick(a,p-2);
```

4.2.5 欧拉函数

p为素数时$\phi(p) = p - 1$
a与n互质的时候 $a^{\phi(n)} \equiv 1 \bmod n$
m,n互质$\phi(mn) = \phi(m) * \phi(n)$

这个东西好呀，他求的是比n小的，并且和n互质的数的个数

```
//欧拉函数 求的是 1 -> n-1 中与n 互质的数的个数
ll phi(ll n) //直接实现
{
  ll rea = n;
  for(int i = 2;i*i<=n;i++)
  {
    if(n%i == 0)
    {
      rea = rea - rea/i;
      while(n%i == 0) n/=i;
    }
    if(n>1)
    rea = rea - rea/n;
    return rea;
  }
}
//欧拉打表
for(int i = 1;i<=maxn;i++) phi[i] = i;
for(int i = 2;i<=maxn;i+=2) phi[i]/=2;
for(int i = 3;i<=maxn;i+=2)
{
  if(phi[i] == i)
  {
    for(j = i ;j<=maxn;j+=i)
    {
      phi[j] = phi[j]/i*(i-1);
    }
  }
}
```

更多的来说我觉得这个东西是一个工具，他对解一些题有很重要的作用，起到一个工具的作用 我目前学的比较浅，对他的优化作用没有很深的了解。

4.2.6 莫比乌斯函数

$$F_n = \sum_{i=1}^{n} f_i$$

$$f_i = \sum_{d|n} u(d) * f(\frac{d}{n})$$

和欧拉函数一样很重要的一个函数他的定义我就不说了，毕竟我latex学的还不好，公式的 插入对我来说用处不大。

```
const int MAXN = 100005;
bool check[MAXN+10];
int prime[MAXN+10];
int mu[MAXN+10];
void Moblus()
{
    clr(check,0);
    mu[1] = 1;
    int tot = 0;
    for(int i = 2; i <= MAXN; i++)
    {
        if( !check[i] )
        {
            prime[tot++] = i;
            mu[i] = -1;
        }
        for(int j = 0; j < tot; j++)
        {
            if(i * prime[j] > MAXN) break;
            check[i * prime[j]] = true;
            if( i % prime[j] == 0)
            {
                mu[i * prime[j]] = 0;
                break;
            }
            else
            {
                mu[i * prime[j]] = -mu[i];
            }
        }
    }
}
```

4.3 博弈

4.3.1 主要的解题思想

官方说的是通过必败点和必胜点来判定 先通过必败点来推，直接来看必胜点，把问题抽象成图 把状态抽象成点，必败点就是先手必败点，然后通过必败点能走到的搞成必胜点，如过有一个状态没有走过 而且他后面的路都是必胜点那么他就是必败点。感觉就像dp一样，记忆化搜索。 当然题目不可能出的那么简单的。 不过根据雄爷定理，万事不离期宗，掌握基本，扩展自己去发掘。

4.3.2 题型

巴什博弈
这个是最简单的博弈，就是一堆东西，每个人自己能拿1-n件，谁最后一个拿完谁赢，这个是最简单的，不记录。
威佐夫博弈
有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。 这个的解题思路在于通过前面的那个np问题来解决，用局势来思考这些问题，前几个局势在于(0,0),(1,2),(3,5),(4,7).....然后一些大佬就总结出了一些牛逼的结论$(a_k, b_k), a_k = \frac{k*(\sqrt{5}+1)}{2}, b_k = a_k + k$人才。
Fibonacci
有一堆个数为n的石子，游戏双方轮流取石子，满足：
（1）先手不能在第一次把所有的石子取完；
（2）之后每次可以取的石子数介于1到对手刚取的石子数的2倍之间（包含1和对手刚取的石子数的2倍）。 约定取走最后一个石子的人为赢家。 结论是 当n为Fibonacci数时，先手必败
尼姆博弈
有三堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得胜。 这个博弈有点意思 他的必败点的局势在于$(a, b, c) a \wedge b \wedge c = 0$

4.3.3 SG函数

这个在看之前感觉很高级但是啊，好像也就是一个dp的过程，通过一个必败点，看成起点然后，那个方法看成通向下一个起点的路，然后找所有能直接到这个必败点的必胜点。好像也就那么回事。好像能解决的都是小数字题这是一个板子，f里面存的是方法，多堆问题可以转化成异或来解决。

```
void getSG(int n){
    int i,j;
    memset(SG,0,sizeof(SG));
    for(i = 1; i <= n; i++){
        memset(S,0,sizeof(S));
        for(j = 0; f[j] <= i && j <= N; j++)
```

```
            S[SG[i-f[j]]] = 1;
        for(j = 0;;j++) if(!S[j]){
            SG[i] = j;
            break;
        }
    }
}
```

4.3.4 解题策略

* 1：相信自己的第一感觉
2 ：博弈都会和一些特别的数搭边， 所以第一件事坑定是分析局势然后找找看是不是有特别的意义，像什么 卡特兰数， f i b 数列 ，幂次方，异或的值是否为 0 ;
3：不挂怎么说，记得打表。

4.4 组合数学

4.4.1 求组合数

第一个是求组合数,方法很多不去列举，注意的是一般来说，组合数都是需要去 模一个数，所以他的分母在计算的时候是需要去求逆元的

```
LL C[1005][1005];
void init()
{
    C[0][0] = 1;
    C[1][0] = C[1][1] = 1;
    for(int i = 2; i <= 1000; i++) {
        C[i][0] = 1;
        for(int j = 1; j <= i; j++)
            C[i][j] = (C[i-1][j-1] + C[i-1][j]) % mod;
    }
}
```

4.4.2 polay定理

设G=p1，p2，. . . ，pt是X=a1,a2,. . . ,an上一个置换群，用m种颜色对X中的元素进行涂色，那么不同的涂色方案数为

$$\frac{1}{G}\sum_{k=1}^{t} m^{Cyc(p_k)}$$

$Cyc(p_k)$是置换p_k的循环节个数

4.4.3 Pell方程

$$x^2 - dy^2 = 1$$

当d不为平方数时，有无穷多的解。接下来是通项的推导

$$x_{n+1} = x_1 * x_n + d * y_1 * y_n, y_{n+1} = x_1 * y_n + y_1 * x_n$$

暴力求x_1, y_1然后矩阵快速幂

4.4.4 lucas定理

当组合数的基数过大的时候进行这些操作但是注意，我们的操作也是要求那个模数为素数，且 模数要小的情况下，素数的情况我们可以用扩展lucas定理来解决。一个工具，一个数论上的分支。

```
//卢卡斯定理
//用于求组合数 当那两个玩意特别大的时候
//注意啊，我这里是用快速幂来求乘法逆元
//他的要求为 mod 必须为素数
//好像也没有如果，不然好像还真不知道
int mod;
ll dp[maxn+5];
void init()
{
  dp[0] = 1;
  for(int i = 1;i<=mod;i++)
  {
    dp[i] = dp[i-1]*i%mod;
  }
}
ll quick(ll a , ll n)
{
  ll res = 1;
  while(n)
  {
     if(n&1) res = res*a%mod;
     a = (a%mod)*(a%mod)%mod;
     n/=2;
  }
  return res;
}
ll lucas(ll n, ll m)
{
  ll ret = 1;
  while(n && m)
```

```
  {
    ll a = n%mod, b = m%mod;
    if(a<b) return 0;
    ret = ((ret * dp[a])%mod*quick(dp[b]*dp[a-b]%mod,mod-2))%mod;
    n/=mod;
    m/=mod;
  }
  return ret;
}
//扩展卢卡斯定理 及p不为素数  p<=1000000左右吧
//还利用了中国剩余定理
ll n,m,MOD,ans;

ll fast_pow(ll a,ll p,ll Mod)
{
    ll ans=1ll;
    for (;p;p>>=1,a=a*a%Mod)
        if (p&1)
            ans=ans*a%Mod;
    return ans;
}
void exgcd(ll a,ll b,ll &x,ll &y)
{
    if (!b) x=1ll,y=0ll;
    else exgcd(b,a%b,y,x),y-=a/b*x;
}
ll inv(ll A,ll Mod)
{
    if (!A) return 0ll;
    ll a=A,b=Mod,x=0ll,y=0ll;
    exgcd(a,b,x,y);
    x=((x%b)+b)%b;
    if (!x) x+=b;
    return x;
}
ll Mul(ll n,ll pi,ll pk)
{
    if (!n) return 1ll;
    ll ans=1ll;
    if (n/pk)
    {
        for (ll i=2;i<=pk;++i)
            if (i%pi) ans=ans*i%pk;
        ans=fast_pow(ans,n/pk,pk);
```

```
    }
    for (ll i=2;i<=n%pk;++i)
        if (i%pi) ans=ans*i%pk;
    return ans*Mul(n/pi,pi,pk)%pk;
}
ll C(ll n,ll m,ll Mod,ll pi,ll pk)
{
    if (m>n) return 0ll;
    ll a=Mul(n,pi,pk),b=Mul(m,pi,pk),c=Mul(n-m,pi,pk);
    ll k=0ll,ans;
    for (ll i=n;i;i/=pi) k+=i/pi;
    for (ll i=m;i;i/=pi) k-=i/pi;
    for (ll i=n-m;i;i/=pi) k-=i/pi;
    ans=a*inv(b,pk)%pk*inv(c,pk)%pk*fast_pow(pi,k,pk)%pk;
    return ans*(Mod/pk)%Mod*inv(Mod/pk,pk)%Mod;
}
ll slove(ll n,ll m,ll p) //求C(n,m) mod p;
{
   for(ll x=p,i=2;i<=p;i++)
   {
     if(x%i == 0)
     {
       ll pk = 1;
       while(x%i == 0) pk*=i,x/=i;
       ans = (ans + C(n,m,mod,i,pk))%mod;
     }
   }
   return ans;
}
```

4.5 线代

4.5.1 fft优化多项式乘法法

这个是一个工具，他的作用是加速多项式得乘法。这个点得难处还是在于多项式的构建，其实也不是 非常复杂得东西，当然他的原理，不敢去碰。关键词：多项式乘法，并且复杂度为1e5左右。

```
#include<bits/stdc++11>
using namespace std;
//版子
struct complex
{
    double r,i;
    complex(double _r = 0.0,double _i = 0.0)
```

```
    {
        r = _r; i = _i;
    }
    complex operator +(const complex &b)
    {
        return complex(r+b.r,i+b.i);
    }
    complex operator -(const complex &b)
    {
        return complex(r-b.r,i-b.i);
    }
    complex operator *(const complex &b)
    {
        return complex(r*b.r-i*b.i,r*b.i+i*b.r);
    }
};
const int MAXN = 200010;
complex x1[MAXN],x2[MAXN];//这一个是第一个多项式的系数，第二个是第二个多项式的系数
char str1[MAXN/2],str2[MAXN/2];//这是未处理的输入字符
int sum[MAXN];//这是答案所放的位置
/*
 * 进行FFT和IFFT前的反转变换。
 * 位置i和 （i二进制反转后位置）互换
 * len必须去2的幂
 */
void change(complex y[],int len)
{
    int i,j,k;
    for(i = 1, j = len/2;i < len-1; i++)
    {
        if(i < j)swap(y[i],y[j]);
        //交换互为小标反转的元素，i<j保证交换一次
        //i做正常的+1，j左反转类型的+1,始终保持i和j是反转的
        k = len/2;
        while( j >= k)
        {
            j -= k;
            k /= 2;
        }
        if(j < k) j += k;
    }
}
/*
 * 做FFT
```

```
 * len必须为2^k形式，
 * on==1时是DFT，on==-1时是IDFT
 */
void fft(complex y[],int len,int on)
{
    change(y,len);
    for(int h = 2; h <= len; h <<= 1)
    {
        complex wn(cos(-on*2*PI/h),sin(-on*2*PI/h));
        for(int j = 0;j < len;j+=h)
        {
            complex w(1,0);
            for(int k = j;k < j+h/2;k++)
            {
                complex u = y[k];
                complex t = w*y[k+h/2];
                y[k] = u+t;
                y[k+h/2] = u-t;
                w = w*wn;
            }
        }
    }
    if(on == -1)
        for(int i = 0;i < len;i++)
            y[i].r /= len;
}
void ans()
{
        //step 1
        int len = 1;
        int len1 = strlen(str1),len2 = strlen(str2);
        while(len<2*len1 || len<2*len2)       len<<=1;//第一步确定len相当于在确定那个
        //step 2
  for(int i = 0;i<len1;i++)          //第二膊就是把系数表示用点来表示，挺复杂的，首先就是
    x1[i] = complex(str1[len1-1-i]-'0',0);
 for(int i = len1;i<len;i++)
    x1[i] = complex(0,0);
 for(int i = 0;i<len2;i++)
    x2[i] = complex(str2[len2-1-i]-'0',0);
 for(int i = len2;i<len;i++)
    x2[i] = complex(0,0);
        //step 3
        fft(x1,len,1);fft(x2,len,1);//第三步反正就是这意思，我也不懂这个是啥原理
        for(int i = 0;i<len;i++)
```

```
        {
                x1[i] = x1[i]*x2[i];
        }
        fft(x1,len,-1);
        for(int i = 0;i<len;i++)
        {
                sum[i] = (int)(x1[i].r+0.5);
        }
        //step 4
        //这就是自己加工了，这里面存的是答案，不过这也是倒着存的，注意去前导0，因为这里


}
```

4.5.2 矩阵快速幂

这类方法，很多是用在递推关系式的时候，像什么fib数列什么的。

```
struct node
{
  ll p[2][2];
};
node mut(node a,node b)
{
  node o;
  clr(o.p,0);
  for(int i = 0;i<2;i++)
  {
    for(int j = 0;j<2;j++)
    {
      for(int k = 0;k<2;k++)
      {
        o.p[i][j] =( a.p[i][k] * b.p[k][j] + o.p[i][j])%mod;
      }
    }
  }
  return o;
}
node quick(node a,ll l)
{
  node origin;
  clr(origin.p,0);
  origin.p[1][1] = origin.p[0][0] = 1;
  while(l)
  {
```

```
    if(l&1) origin = mut(a,origin);
    a = mut(a,a);
    l/=2;
  }
  return origin;
}
```

4.5.3 矩阵方面知识

就是用高斯消元法去解决一些问题，像什么秩和方阵的值。

4.6 计算几何

4.6.1 几何基本知识

矢量
矢量的乘积有很多的作用，注意定义。 适用点在于：1:面积 2：位置
跨立实验与判断两线段是否相交
线段P_1P_2, Q_1Q_2,相交的条件为
$P_1Q_1xP_1P_2 * P_1P_2xP_1Q_2 >= 0$
$Q_1P_1xQ_1Q_2 * Q_1Q_2xQ_1P_2 >= 0$
$pick$定理
线段上的是整数点的数的个数 求gcd;
PICK定理 设以整数点为顶点的多边形的面积为S，多边形内部的整数点数为N，多边形边界上的整数点数为L，则 S=L/2 + N-1

4.6.2 判断点是否在多边形中

```
const double eps=1e-8;//解析几何中有时并不能保证等于0，在误差范围就行
struct CPoint//点的存法
{
    double x,y;
}point[103];
int dcmp(double x)//不晓得干啥
{
    if(x<-eps) return -1;
    else   return (x>eps);
}
double cross(CPoint p0,CPoint p1,CPoint p2)//点乘
{
    return (p1.x-p0.x)*(p2.y-p0.y)-(p2.x-p0.x)*(p1.y-p0.y);
}
double dot(CPoint p0,CPoint p1,CPoint p2)//叉乘
{
    return (p1.x-p0.x)*(p2.x-p0.x)+(p1.y-p0.y)*(p2.y-p0.y);
```

```
}
int PointOnSegment(CPoint p0,CPoint p1,CPoint p2)//判断点是否在线段上
{
    return dcmp(cross(p0,p1,p2))==0&&dcmp(dot(p0,p1,p2))<=0;
}
int PointInPolygon(CPoint cp,CPoint p[],int n)//判断点是否在多边形中
{
    int i,k,d1,d2,wn=0;
  //  double sum=0;
    p[n]=p[0];
    for( i=0;i<n;i++)
    {
        if(PointOnSegment(cp,p[i],p[i+1])) return 2;
        k=dcmp(cross(p[i],p[i+1],cp));
        d1=dcmp(p[i+0].y-cp.y);
        d2=dcmp(p[i+1].y-cp.y);
        if(k>0&&d1<=0&&d2>0)wn++;
        if(k<0&&d2<=0&&d1>0)wn--;
    }
    return wn!=0;
}
```

为1的时候，则在内部。2，应该是边上。

4.6.3 凸包问题

```
struct node
{
    int x,y;
} a[105],p[105];
int top,n;
double cross(node p0,node p1,node p2)//计算叉乘，注意p0,p1,p2的位置，这个决定了方向
{
    return (p1.x-p0.x)*(p2.y-p0.y)-(p1.y-p0.y)*(p2.x-p0.x);
}
double dis(node a,node b)//计算距离，这个用在了当两个点在一条直线上
{
    return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}
bool cmp(node p1,node p2)//极角排序
{
    double z=cross(a[0],p1,p2);
    if(z>0||(z==0&&dis(a[0],p1)<dis(a[0],p2)))
        return 1;
```

```
    return 0;
}
void Graham()//p是凸包的点
{
    int k=0;
    for(int i=0; i<n; i++)
        if(a[i].y<a[k].y||(a[i].y==a[k].y&&a[i].x<a[k].x))
            k=i;
        swap(a[0],a[k]);//找p[0]
        sort(a+1,a+n,cmp);
        top=1;
        p[0]=a[0];
        p[1]=a[1];
        for(int i=2; i<n; i++)//控制进栈出栈
        {
            while(cross(p[top-1],p[top],a[i])<0&&top)
                top--;
            top++;
            p[top]=a[i];
        }
}
```

5 状态转移 dp

dp的定义： 1：记忆化搜索；2：状态转移 所以我们的解决方案总是跟着这个来走，从定义出发。 难点集中于两个方面，状态式的确定和状态转移方程的确定

5.1 背包

背包的问题主要以下几种： 01背包，部分背包，完全背包;[相对来说比较简单]，分组背包[个人感觉较难] 背包难在如何， 确定维数，确定背包的容量是什么以及背包的价值是什么，还有背包的dp关系转移式。

5.2 一些常见的dp

5.2.1 LIS 最长上升子序列

```
LIS(LDS)
template<class Cmp>
int LIS (Cmp cmp)(nlogn)
{
    static int m, end[N];
    m = 0;
```

```
    for (int i=0;i<n;i++)
    {
        int pos = lower_bound(end, end+m, a[i], cmp)-end;
        end[pos] = a[i], m += pos==m;
    }
    return m;
}
    cout << LIS(less<int>()) << endl;          //严格上升
    cout << LIS(less_equal<int>()) << endl;    //非严格上升
    cout << LIS(greater<int>()) << endl;       //严格下降
    cout << LIS(greater_equal<int>()) << endl;//非严格下降
```

‘‘‘

5.3 树形dp

关键点在于找状态点间的关系，他一般只有三个关系，父亲节点 ，儿子节点，还有兄弟节点，去找他们之间的关系，所以一般是两遍dfs 找父亲与儿子的关系，找儿子与父亲的关系。

5.4 数位dp

这个dp的精髓在于记忆化搜索，也就是在最高位不是被限定的情况下进行记录，这样的话省掉很多多余的步骤。 所有的出发点都处于这个目的。

5.5 状压dp

这个dp的精髓在于状态转移，不过能压缩的情况也是很限定的。 像什么每个点的状态在于都是能用两个状态来描述，且这些点不多，但是组合的方式很多。 一些状压dp经常用的上的公式。

```
//1 获得当前行的数
int getnum(int x)
{
  int ret = 0;
  while(x)
  {
    x &= x-1;
    ret++;
  }
  return ret;
}
//2 看当前行左右是不是满足题设
bool check(int x)
{
  if(x & x<<1) return 0;
```

```
  return 1;
}
// 看是不是可以满足条件,和题目给的图一样,是可以放的,并且和上一个是不是会冲突
bool suit(int x,int y)
{
  if(x&y) return 0;
  return 1;
}
```

5.6 the end