acm-template

langman

February 21, 2018

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 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]);
}
```

2.2 最短路

两种算法 但是要注意dijkstra无法处理负边的情况

2.2.1 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]);
    }
  }
}
```

2.2.2 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;
}
```

2.2.3 Flody

这个就不写了,一个小dp

2.3 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;
}
```

2.4 最大流

2.4.1 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 数学方面

3.1 三个特别的数

3.1.1 Fib 数列

$$f(x) = f(x-1) + f(x-2)$$
$$f(0) = 0, f(1) = 1$$

3.1.2 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。

3.1.3 斯大林公式

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

3.2 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。

3.2.1 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;
}
```

3.2.2 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);
```

3.2.3 欧拉函数

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;
      do n/=i;
      while(n%i == 0);
    }
    if(n>1)
    rea = rea - rea/n;
    return rea;
  }
}
//素数表
bool check[50000];
int p[20000];
void prim()//线性筛素数
{
  clr(check,0);


}
```

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。

3.2.4 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];
            }
        }
    }
}
```

3.3 组合数学

3.3.1 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;
    }
}
```

3.3.2 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;
}
```

3.4 线代

3.4.1 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;
}
```

3.4.2 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。

3.5 计算几何

3.5.1 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
$P_1Q_1 x P_1P_2 * P_1P_2 x P_1Q_2 >= 0$
$Q_1P_1 x Q_1Q_2 * Q_1Q_2 x Q_1P_2 >= 0$
$pick$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

3.5.2 判断点是否在多边形中

4 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

4.1 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。

4.2 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。

4.3 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。

4.4 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;
}
```

4.5 一些常见的dp

4.5.1 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 The end