ACM/ICPC Template Manaual

Mango Gao

March 12, 2016

Contents

1 头文件模板

```
#include <bits/stdc++.h> // c++0x only
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <set>
#include <map>
#include <cmath>
#include <iomanip>
#include <functional>
#include <cstdlib>
#include <climits>
#include <cctype>
using namespace std;
#define REP(i,x) for(int i = 0; i < (x); i++)
#define DEP(i,x) for(int i = (x) - 1; i >= 0; i--)
#define FOR(i,x) for(__typeof(x.begin())i=x.begin(); i!=x.end(); i++)
#define CLR(a,x) memset(a, x, sizeof(a))
#define MO(a,b) (((a)%(b)+(b))%(b))
#define ALL(x) (x).begin(), (x).end()
#define SZ(v) ((int)v.size())
#define UNIQUE(v) sort(ALL(v)); v.erase(unique(ALL(v)), v.end())
#define out(x) cout << #x << ": " << x << endl;
#define fastcin ios_base::sync_with_stdio(0);cin.tie(0);
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int, int> PII;
typedef vector<int> VI;
#define INF 0x3f3f3f3f
#define MOD 1000000007
#define EPS 1e-8
#define MP(x,y) make_pair(x,y)
#define MT(x,y...) make_tuple(x,y) // c++0x only
#define PB(x) push_back(x)
#define IT iterator
#define X first
#define Y second
```

2 数学

2.1 素数

2.1.1 埃氏筛

$O(n \log \log n)$ 筛出 MAXN 内所有素数
$notprime[i] = 0/1$ 0 为素数 1 为非素数

```
const int MAXN = 1000100;
bool notprime[MAXN] = {1, 1}; // 0 && 1 为非素数
void GetPrime() {
   for (int i = 2; i < MAXN; i++)
      if (!notprime[i] && i <= MAXN / i) // 筛到√n为止
         for (int j = i * i; j < MAXN; j += i)
            notprime[j] = 1;
}
```

2.1.2 欧拉筛

$O(n)$ 得到欧拉函数 $phi[]$、素数表 $prime[]$、素数个数 tot
传入的 n 为函数定义域上界

```
const int MAXN = 100010;
bool vis[MAXN];
int tot, phi[MAXN], prime[MAXN];
void CalPhi(int n) {
   set(vis, 0); phi[1] = 1; tot = 0;
   for (int i = 2; i < n; i++) {
      if (!vis[i]) {
         prime[tot++] = i;
         phi[i] = i - 1;
      }
      for (int j = 0; j < tot; j++) {
         if (i * prime[j] > n) break;
         vis[i * prime[j]] = 1;
         if (i % prime[j] == 0) {
            phi[i * prime[j]] = phi[i] * prime[j];
            break;
         }
         else phi[i * prime[j]] = phi[i] * (prime[j] - 1);
      }
   }
}
```

2.1.3 随机素数判定

$O(s \log n)$ 内判定 2^{63} 内的数是不是素数，s 为测定次数

```
bool Miller_Rabin(ll n, int s) {
   if (n == 2) return 1;
   if (n < 2 || !(n & 1)) return 0;
   int t = 0; ll x, y, u = n - 1;
   while ((u & 1) == 0) t++, u >>= 1;
   for (int i = 0; i < s; i++) {
      ll a = rand() % (n - 1) + 1;
      ll x = Pow(a, u, n);
      for (int j = 0; j < t; j++) {
         ll y = Mul(x, x, n);
         if (y == 1 && x != 1 && x != n - 1) return 0;
         x = y;
      }
      if (x != 1) return 0;
   }
   return 1;
}
```

2.1.4 分解质因数

函数返回素因数个数
数组以 $fact[i][0]^{fact[i][1]}$ 的形式保存第 i 个素因数

```
ll fact[100][2];
int getFactors(ll x) {
   int cnt = 0;
   for (int i = 0; prime[i] <= x / prime[i]; i++) {
      fact[cnt][1] = 0;
      if (x % prime[i] == 0 ) {
         fact[cnt][0] = prime[i];
         while (x % prime[i] == 0) {
            fact[cnt][1]++;
            x /= prime[i];
         }
         cnt++;
      }
   }
   if (x != 1) {
      fact[cnt][0] = x;
      fact[cnt++][1] = 1;
   }
   return cnt;
}
```

2.2 欧拉函数

2.2.1 求一个数的欧拉函数

```
long long Euler(long long n) {
   long long rt = n;
   for (int i = 2; i * i <= n; i++)
      if (n % i == 0) {
         rt -= rt / i;
         while (n % i == 0) n /= i;
      }
   if (n > 1) rt -= rt / n;
   return rt;
}
```

2.2.2 筛法求欧拉函数

```
const int MAXN = 10001;
int phi[MAXN] = {0, 1};
void CalEuler() {
   for (int i = 2; i < MAXN; i++)
      if (!phi[i]) for (int j = i; j < MAXN; j += i) {
            if (!phi[j]) phi[j] = j;
            phi[j] = phi[j] / i * (i - 1);
         }
}
```

2.3 扩展欧几里得-乘法逆元

2.3.1 扩展欧几里得

```
void exgcd(ll a, ll b, ll &d, ll &x, ll &y) {
   if (!b) {d = a; x = 1; y = 0;}
   else {exgcd(b, a % b, d, y, x); y -= x * (a / b);}
}
```

2.3.2 求 ax+by=c 的解

```
// 引用返回通解: X = x + k * dx, Y = y - k * dy
// 引用返回的x是最小非负整数解, 方程无解函数返回0
#define Mod(a,b) (((a)%(b)+(b))%(b))
bool solve(ll a, ll b, ll c, ll &x, ll &y, ll &dx, ll &dy) {
    if (a == 0 && b == 0) return 0;
    ll d, x0, y0; exgcd(a, b, d, x0, y0);
    if (c % d != 0) return 0;
    dx = b / d; dy = a / d;
    x = Mod(x0 * c / d, dx); y = (c - a * x) / b;
//  y = Mod(y0 * c / d, dy); x = (c - b * y) / a;
    return 1;
}
```

2.3.3 乘法逆元

```
// 利用exgcd求a在模m下的逆元, 需要保证gcd(a, m) == 1.
ll inv(ll a, ll m) {
    ll x, y, d; exgcd(a, m, d, x, y);
    return d == 1 ? (x + m) % m : -1;
}
// a < m 且 m为素数时, 有以下两种求法
ll inv(ll a, ll m) {
    return a == 1 ? 1 : inv(m % a, m) * (m - m / a) % m;
}
ll inv(ll a, ll m) {
    return Pow(a, m - 2, m);
}
```

2.4 模线性方程组

2.4.1 中国剩余定理

```
// X = r[i] (mod m[i]); 要求m[i]两两互质
// 引用返回通解X = re + k * mo;
void crt(ll r[], ll m[], ll n, ll &re, ll &mo) {
    mo = 1, re = 0;
    for (int i = 0; i < n; i++) mo *= m[i];
    for (int i = 0; i < n; i++) {
        ll x, y, d, tm = mo / m[i];
        exgcd(tm, m[i], d, x, y);
        re = (re + tm * x * r[i]) % mo;
    } re = (re + mo) % mo;
}
```

2.4.2 一般模线性方程组

```
// X = r[i] (mod m[i]); m[i]可以不两两互质
// 引用返回通解X = re + k * mo; 函数返回是否有解
bool excrt(ll r[], ll m[], ll n, ll &re, ll &mo) {
    ll x, y, d; mo = m[0], re = r[0];
    for (int i = 1; i < n; i++) {
        exgcd(mo, m[i], d, x, y);
        if ((r[i] - re) % d != 0) return 0;
        x = (r[i] - re) / d * x % (m[i] / d);
        re += x * mo;
        mo = mo / d * m[i];
        re %= mo;
    } re = (re + mo) % mo;
    return 1;
}
```

2.5 组合数

2.5.1 一般组合数

```
// 0 <= m <= n <= 1000
const int maxn = 1010;
ll C[maxn][maxn];
void CalComb() {
   C[0][0] = 1;
   for (int i = 1; i < maxn; 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;
   }
}

// 0 <= m <= n <= 105, 模p为素数
const int maxn = 100010;
ll f[maxn];
void CalFact() {
   f[0] = 1;
   for (int i = 1; i < maxn; i++)
      f[i] = (f[i - 1] * i) % mod;
}
ll C(int n, int m) {
   return f[n] * inv(f[m] * f[n - m] % mod, mod) % mod;
}
```

2.5.2 Lucas 定理

```
// 1 <= n, m <= 1000000000, 1 < p < 100000, p是素数
const int maxp = 100010;
ll f[maxp];
void CalFact(ll p) {
   f[0] = 1;
   for (int i = 1; i <= p; i++)
      f[i] = (f[i - 1] * i) % p;
}
ll Lucas(ll n, ll m, ll p) {
   ll ret = 1;
   while (n && m) {
      ll a = n % p, b = m % p;
      if (a < b) return 0;
      ret = (ret * f[a] * Pow(f[b] * f[a - b] % p, p - 2, p)) % p;
      n /= p; m /= p;
   }
   return ret;
}
```

2.5.3 大组合数

```
// 0 <= n <= 109, 0 <= m <= 104, 1 <= k <= 109+7
vector<int> v;
int dp[110];
ll Cal(int l, int r, int k, int dis) {
   ll res = 1;
   for (int i = l; i <= r; i++) {
      int t = i;
      for (int j = 0; j < v.size(); j++) {
         int y = v[j];
         while (t % y == 0) {
            dp[j] += dis;
            t /= y;
         }
      }
      res = res * (ll)t % k;
   }
   return res;
}
ll Comb(int n, int m, int k) {
   set(dp, 0); v.clear(); int tmp = k;
   for (int i = 2; i * i <= tmp; i++) {
      if (tmp % i == 0) {
         int num = 0;
```

```
            while (tmp % i == 0) {
                tmp /= i;
                num++;
            }
            v.pb(i);
        }
    } if (tmp != 1) v.pb(tmp);
    ll ans = Cal(n - m + 1, n, k, 1);
    for (int j = 0; j < v.size(); j++) {
        ans = ans * Pow(v[j], dp[j], k) % k;
    }
    ans = ans * inv(Cal(2, m, k, -1), k) % k;
    return ans;
}
```

2.6 快速乘-快速幂

```
ll Mul(ll a, ll b, ll mod) {
    ll t = 0;
    for (; b; b >>= 1, a = (a << 1) % mod)
        if (b & 1) t = (t + a) % mod;
    return t;
}
ll Pow(ll a, ll n, ll mod) {
    ll t = 1;
    for (; n; n >>= 1, a = (a * a % mod))
        if (n & 1) t = (t * a % mod);
    return t;
}
```

3 字符串

3.1 KMP

```
// 返回y中x的个数
int next[10010];
void kmp_pre(char x[], int m, int next[]) {
   int i, j; j = next[0] = -1; i = 0;
   while (i < m) {
      while (j != -1 && x[i] != x[j])
         j = next[j];
      next[++i] = ++j;
   }
}
int KMP (char x[], int m, char y[], int n) {
   int i, j, ans; i = j = ans = 0;
   kmp_pre (x, m, next);
   while (i < n) {
      while (j != -1 && y[i] != x[j]) j = next[j];
      i++; j++;
      if (j >= m) {
         ans++; j = next[j];
      }
   }
   return ans;
}
```

3.2 Manacher 最长回文子串

```
const int MAXN = 110010;
char Ma[MAXN * 2];
int Mp[MAXN * 2];
void Manacher(char s[], int len) {
   int l = 0; Ma[l++] = '$'; Ma[l++] = '#';
   for (int i = 0; i < len; i++) {
      Ma[l++] = s[i]; Ma[l++] = '#';
   }
   Ma[l] = 0; int mx = 0, id = 0;
   for (int i = 0; i < l; i++) {
      Mp[i] = mx > i ? min(Mp[2 * i d - i], mx - i) : 1;
      while (Ma[i + Mp[i]] == Ma[i - Mp[i]]) Mp[i]++;
      if (i + Mp[i] > mx) {
         mx = i + Mp[i]; id = i;
      }
   }
}
```

4 数据结构

4.1 树状数组

$O(\log n)$ 查询和修改数组的前缀和

```
// 注意下标应从1开始
#define lowbit(i) (i&(-i))
int bit[maxn], n;
int query(int i){
   int s = 0;
   while(i){
      s += bit[i];
      i -= lowbit(i);
   }
   return s;
}
void add(int i, int x){
   while(i<=n){
      bit[i] += x;
      i += lowbit(i);
   }
}
```

4.2 线段树

4.2.1 声明

```
#define lson rt<<1   // 左儿子
#define rson rt<<1|1   // 右儿子
#define Lson l,m,lson  // 左子树
#define Rson m+1,r,rson  // 右子树
void PushUp(int rt);   // 用lson和rson更新rt
void PushDown(int rt[, int m]); // rt的标记下移，m为区间长度（若与标记有关）
void build(int l, int r, int rt); // 以rt为根节点，对区间[l, r]建立线段树
void update([...,] int l, int r, int rt) // rt[l, r]内寻找目标并更新
int query(int L, int R, int l, int r, int rt) // rt-[l, r]内查询[L, R]
```

4.2.2 单点更新-区间查询

```
const int maxn = 50010;
int sum[maxn << 2];
void PushUp(int rt) {
   sum[rt] = sum[lson] + sum[rson];
}
void build(int l, int r, int rt) {
   if (l == r) {scanf("%d", &sum[rt]); return;} // 建立的时候直接输入叶节点
   int m = (l + r) >> 1;
   build(Lson); build(Rson);
   PushUp(rt);
}
void update(int p, int add, int l, int r, int rt) {
   if (l == r) {sum[rt] += add; return;}
   int m = (l + r) >> 1;
   if (p <= m) update(p, add, Lson);
   else update(p, add, Rson);
   PushUp(rt);
}
int query(int L, int R, int l, int r, int rt) {
   if (L <= l && r <= R) {return sum[rt];}
   int m = (l + r) >> 1, s = 0;
   if (L <= m) s += query(L, R, Lson);
   if (m < R) s += query(L, R, Rson);
   return s;
}
```

4.2.3 区间更新-区间查询

```
// seg[rt]用于存放懒惰标记, 注意PushDown时标记的传递
const int maxn = 101010;
int seg[maxn << 2], sum[maxn << 2];
void PushUp(int rt) {
   sum[rt] = sum[lson] + sum[rson];
}
void PushDown(int rt, int m) {
   if (seg[rt] == 0) return;
   seg[lson] += seg[rt];
   seg[rson] += seg[rt];
   sum[lson] += seg[rt] * (m - (m >> 1));
   sum[rson] += seg[rt] * (m >> 1);
   seg[rt] = 0;
}
void build(int l, int r, int rt) {
   seg[rt] = 0;
   if (l == r) {scanf("%lld", &sum[rt]); return;}
   int m = (l + r) >> 1;
   build(Lson); build(Rson);
   PushUp(rt);
}
void update(int L, int R, int add, int l, int r, int rt) {
   if (L <= l && r <= R) {
      seg[rt] += add;
      sum[rt] += add * (r - l + 1);
      return;
   }
   PushDown(rt, r - l + 1);
   int m = (l + r) >> 1;
   if (L <= m) update(L, R, add, Lson);
   if (m < R) update(L, R, add, Rson);
   PushUp(rt);
}
int query(int L, int R, int l, int r, int rt) {
   if (L <= l && r <= R) return sum[rt];
   PushDown(rt, r - l + 1);
   int m = (l + r) >> 1, ret = 0;
   if (L <= m) ret += query(L, R, Lson);
   if (m < R) ret += query(L, R, Rson);
   return ret;
}
```

5 图论

5.1 并查集

```
const int MAXN = 128;
int fa[MAXN], ra[MAXN];
void init(int n) {
   for (int i = 0; i <= n; i++) {
      fa[i] = i; ra[i] = 0;
   }
}
int find(int x) {
   if (fa[x] != x) fa[x] = find(fa[x]);
   return fa[x];
}
void unite(int x, int y) {
   x = find(x); y = find(y); if (x == y) return;
   if (ra[x] < ra[y]) fa[x] = y;
   else {
      fa[y] = x; if (ra[x] == ra[y]) ra[x]++;
   }
}
bool same(int x, int y) {
   return find(x) == find(y);
}
```

5.2 最小生成树

5.2.1 Kruskal

```
vector<pair<int, PII> > G;
void add_edge(int u, int v, int d) {
   G.push_back(make_pair(d, make_pair(u, v)));
}
int Kruskal(int n) {
   init(n);
   sort(G.begin(), G.end());
   int m = G.size();
   int num = 0, ret = 0;
   for (int i = 0; i < m; i++) {
      pair<int, PII> p = G[i];
      int x = p.Y.X;
      int y = p.Y.Y;
      int d = p.X;
      if (!same(x, y)) {
         unite(x, y);
         num++;
         ret += d;
         printf("(%d, %d) %d\n", x, y, d);
      }
      if (num == n - 1) break;
   }
   return ret;
}
```

5.3 最短路

5.3.1 Dijkstra-邻接矩阵

```
// MAXN为点数最大值 求s到所有点的最短路
// 要求边权值为非负数 模板为有向边
// dis[x]为起点到点x的最短路 inf表示无法走到
// 记得初始化
const int MAXN = 100; // 点数最大值
const int INF = 0x3F3F3F3F;
int G[MAXN][MAXN], dis[MAXN];
bool vis[MAXN];
void init(int n) {
   memset(G, 0x3F, sizeof(G));
```

```
}
void add_edge(int u, int v, int w) {
   G[u][v] = min(G[u][v], w);
}
void Dijkstra(int s, int n) {
   memset(vis, 0, sizeof(vis));
   memset(dis, 0x3F, sizeof(dis));
   dis[s] = 0;
   for (int i = 0; i < n; i++) {
      int x, min_dis = INF;
      for (int j = 0; j < n; j++) {
         if (!vis[j] && dis[j] <= min_dis) {
            x = j;
            min_dis = dis[j];
         }
      }
      vis[x] = 1;
      for (int j = 0; j < n; j++)
         dis[j] = min(dis[j], dis[x] + G[x][j]);
   }
}
```

5.3.2 Dijkstra-邻接表数组

```
// 点最大值: MAX_N 边最大值: MAX_E
// 求起点s到每个点x的最短路dis[x]
const int MAX_N = "Edit"; // 点数最大值
const int MAX_E = "Edit";
const int INF = 0x3F3F3F3F;
int tot;
int Head[MAX_N], vis[MAX_N], dis[MAX_N];
int Next[MAX_E], To[MAX_E], W[MAX_E];
void init() {
   tot = 0;
   memset(Head, -1, sizeof(Head));
}
void add_edge(int u, int v, int d) {
   W[tot] = d;
   To[tot] = v;
   Next[tot] = Head[u];
   Head[u] = tot++;
}
void Dijkstra(int s, int n) {
   memset(vis, 0, sizeof(vis));
   memset(dis, 0x3F, sizeof(dis));
   dis[s] = 0;
   for (int i = 0; i < n; i++) {
      int x, min_dis = INF;
      for (int j = 0; j < n; j++) {
         if (!vis[j] && dis[j] <= min_dis) {
            x = j;
            min_dis = dis[j];
         }
      }
      vis[x] = 1;
      for (int j = Head[x]; j != -1; j = Next[j]) {
         int y = To[j];
         dis[y] = min(dis[y], dis[x] + W[j]);
      }
   }
}
```

5.3.3 Dijkstra-邻接表向量

```
// MAXN: 点数最大值
// 求起点s到所有点x的最短路dis[x]
// 记得初始化
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
vector<int> G[MAXN];
```

```
vector<int> GW[MAXN];
bool vis[MAXN];
int dis[MAXN];
void init(int n) {
   for (int i = 0; i < n; i++) {
      G[i].clear();
      GW[i].clear();
   }
}
void add_edge(int u, int v, int w) {
   G[u].push_back(v);
   GW[u].push_back(w);
}
void Dijkstra(int s, int n) {
   memset(vis, false, sizeof(vis));
   memset(dis, 0x3F, sizeof(dis));
   dis[s] = 0;
   for (int i = 0; i < n; i++) {
      int x;
      int min_dis = INF;
      for (int j = 0; j < n; j++) {
         if (!vis[j] && dis[j] <= min_dis) {
            x = j;
            min_dis = dis[j];
         }
      }
      vis[x] = true;
      for (int j = 0; j < (int)G[x].size(); j++) {
         int y = G[x][j];
         int w = GW[x][j];
         dis[y] = min(dis[y], dis[x] + w);
      }
   }
}
```

5.3.4 Dijkstra-优先队列

```
// pair<权值, 点>
// 记得初始化
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
typedef pair<int, int> PII;
typedef vector<PII> VII;
VII G[MAXN];
int vis[MAXN], dis[MAXN];
void init(int n) {
   for (int i = 0; i < n; i++)
      G[i].clear();
}
void add_edge(int u, int v, int w) {
   G[u].push_back(make_pair(w, v));
}
void Dijkstra(int s, int n) {
   memset(vis, 0, sizeof(vis));
   memset(dis, 0x3F, sizeof(dis));
   dis[s] = 0;
   priority_queue<PII, VII, greater<PII> > PQ;
   PQ.push(make_pair(dis[s], s));
   while (!PQ.empty()) {
      PII t = PQ.top();
      int x = t.second;
      PQ.pop();
      if (vis[x]) continue;
      vis[x] = 1;
      for (int i = 0; i < (int)G[x].size(); i++) {
         int y = G[x][i].second;
         int w = G[x][i].first;
         if (!vis[y] && dis[y] > dis[x] + w) {
            dis[y] = dis[x] + w;
            PQ.push(make_pair(dis[y], y));
         }
      }
```

```
    }
}
```

5.3.5 Bellman-Ford(可判负环)

```
// 求出起点s到每个点x的最短路dis[x]
// 存在负环返回1 否则返回0
// 记得初始化
const int MAX_N = "Edit"; // 点数最大值
const int MAX_E = "Edit"; // 边数最大值
const int INF = 0x3F3F3F3F;
int From[MAX_E], To[MAX_E], W[MAX_E];
int dis[MAX_N], tot;
void init() {tot = 0;}
void add_edge(int u, int v, int d) {
   From[tot] = u;
   To[tot] = v;
   W[tot++] = d;
}
bool Bellman_Ford(int s, int n) {
   memset(dis, 0x3F, sizeof(dis));
   dis[s] = 0;
   for (int k = 0; k < n - 1; k++) {
      bool relaxed = 0;
      for (int i = 0; i < tot; i++) {
         int x = From[i], y = To[i];
         if (dis[y] > dis[x] + W[i]) {
            dis[y] = dis[x] + W[i];
            relaxed = 1;
         }
      }
      if (!relaxed) break;
   }
   for (int i = 0; i < tot; i++)
      if (dis[To[i]] > dis[From[i]] + W[i])
         return 1;
   return 0;
}
```

5.3.6 SPFA

```
// G[u] = mp(v, w)
// SPFA()返回0表示存在负环
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
vector<pair<int, int> > G[MAXN];
bool vis[MAXN];
int dis[MAXN];
int inqueue[MAXN];
void init(int n) {
   for (int i = 0; i < n; i++)
      G[i].clear();
}
void add_edge(int u, int v, int w) {
   G[u].push_back(make_pair(v, w));
}
bool SPFA(int s, int n) {
   memset(vis, 0, sizeof(vis));
   memset(dis, 0x3F, sizeof(dis));
   memset(inqueue, 0, sizeof(inqueue));
   dis[s] = 0;
   queue<int> q; // 待优化的节点入队
   q.push(s);
   while (!q.empty()) {
      int x = q.front();
      q.pop();
      vis[x] = false;
      for (int i = 0; i < G[x].size(); i++) {
         int y = G[x][i].first;
         int w = G[x][i].second;
```

```
            if (dis[y] > dis[x] + w) {
               dis[y] = dis[x] + w;
               if (!vis[y]) {
                  q.push(y);
                  vis[y] = true;
                  if (++inqueue[y] >= n) return 0;
               }
            }
         }
      }
   return 1;
}
```

5.3.7 Floyd 算法

$O(n^3)$ 求出任意两点间最短路

```
const int MAXN = "Edit";
const int INF = 0x3F3F3F3F;
int G[MAXN][MAXN];
void init(int n) {
   memset(G, 0x3F, sizeof(G));
   for (int i = 0; i < n; i++)
      G[i][i] = 0;
}
void add_edge(int u, int v, int w) {
   G[u][v] = min(G[u][v], w);
}
void Floyd(int n) {
   for (int k = 0; k < n; k++)
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
}
```

5.4 拓扑排序

5.4.1 邻接矩阵

```
// 存图前记得初始化
// Ans存放拓排结果, G为邻接矩阵, deg为入度信息
// 排序成功返回1, 存在环返回0
const int MAXN = "Edit";
int Ans[MAXN];  // 存放拓扑排序结果
int G[MAXN][MAXN]; // 存放图信息
int deg[MAXN];  // 存放点入度信息
void init() {
   memset(G, 0, sizeof(G));
   memset(deg, 0, sizeof(deg));
   memset(Ans, 0, sizeof(Ans));
}
void add_edge(int u, int v) {
   if (G[u][v]) return;
   G[u][v] = 1;
   deg[v]++;
}
bool Toposort(int n) {
   int tot = 0;
   queue<int> que;
   for (int i = 0; i < n; ++i)
      if (deg[i] == 0) que.push(i);
   while (!que.empty()) {
      int v = que.front(); que.pop();
      Ans[tot++] = v;
      for (int i = 0; i < n; ++i)
         if (G[v][i] == 1)
            if (--deg[t] == 0) que.push(t);
   }
   if (tot < n) return false;
   return true;
}
```

5.4.2 邻接表

```
// 存图前记得初始化
// Ans排序结果, G邻接表, deg入度, map用于判断重边
// 排序成功返回1, 存在环返回0
const int MAXN = "Edit";
typedef pair<int, int> PII;
int Ans[MAXN];
vector<int> G[MAXN];
int deg[MAXN];
map<PII, bool> S;
void init(int n) {
   S.clear();
   for (int i = 0; i < n; i++)G[i].clear();
   memset(deg, 0, sizeof(deg));
   memset(Ans, 0, sizeof(Ans));
}
void add_edge(int u, int v) {
   if (S[make_pair(u, v)]) return;
   G[u].push_back(v);
   S[make_pair(u, v)] = 1;
   deg[v]++;
}
bool Toposort(int n) {
   int tot = 0; queue<int> que;
   for (int i = 0; i < n; ++i)
      if (deg[i] == 0) que.push(i);
   while (!que.empty()) {
      int v = que.front(); que.pop();
      Ans[tot++] = v;
      for (int i = 0; i < G[v].size(); ++i) {
         int t = G[v][i];
         if (--deg[t] == 0) que.push(t);
      }
   }
   if (tot < n) return false;
   return true;
}
```

6 其他

6.1 子序列

6.1.1 最大子序列和

```
// 传入序列a和长度n，返回最大子序列和
// 限制最短长度：用cnt记录长度，rt更新时判断
int MaxSeqSum(int a[], int n) {
   int rt = 0, cur = 0;
   for (int i = 0; i < n; i++) {
      cur += a[i];
      rt = rt < cur ? cur : rt;
      cur = cur < 0 ? 0 : cur;
   }
   return rt;
}
```

6.1.2 最长上升子序列 LIS

```
// 序列下标从1开始，LIS()返回长度，序列存在lis[]中
#define N 100100
int n, len, a[N], b[N], f[N];
int Find(int p, int l, int r) {
   int mid;
   while (l <= r) {
      mid = (l + r) >> 1;
      if (a[p] > b[mid]) l = mid + 1;
      else r = mid - 1;
   }
   return f[p] = l;
}
int LIS(int lis[]) {
int len = 1;
f[1] = 1;
b[1] = a[1];
   for (int i = 2; i <= n; i++) {
      if (a[i] > b[len]) b[++len] = a[i], f[i] = len;
      else b[Find(i, 1, len)] = a[i];
   }
   for (int i = n, t = len; i >= 1 && t >= 1; i--)
      if (f[i] == t)
         lis[--t] = a[i];
   return len;
}
```

6.1.3 最长公共上升子序列 LCIS

```
// 序列下标从1开始
int LCIS(int a[], int b[], int n, int m) {
   set(dp, 0);
   for (int i = 1; i <= n; i++) {
      int ma = 0;
      for (int j = 1; j <= m; j++) {
         dp[i][j] = dp[i - 1][j];
         if (a[i] > b[j]) ma = max(ma, dp[i - 1][j]);
         if (a[i] == b[j]) dp[i][j] = ma + 1;
      }
   }
   return *max_element(dp[n] + 1, dp[n] + 1 + m);
}
```

6.2 矩阵

```
typedef long long ll;
#define REP(i,n) for(int i=0;i<n;i++)
const ll mod = 1000000007;
class Matrix {
```

```
private:
   int r, c;
   ll **m;
public:
   Matrix(int R, int C): r(R), c(C) {
      m = new ll*[r];
      REP(i, r) m[i] = new ll[c];
      REP(i, r) REP(j, c) m[i][j] = 0;
   }
   Matrix(const Matrix& B) {
      r = B.r; c = B.c; m = new ll*[r];
      REP(i, r) m[i] = new ll[c];
      REP(i, r) REP(j, c) m[i][j] = B.m[i][j];
   }
   ~Matrix() {REP(i, r) delete[] m[i]; delete[] m;}
   void e() {REP(i, r) REP(j, c) m[i][j] = !(i ^ j);}
   ll* operator [] (int p) {return m[p];}
   ll* operator [] (int p) const {return m[p];}
   Matrix& operator = (const Matrix& B) {
      r = B.r; c = B.c; this->~Matrix(); m = new ll*[r];
      REP(i, r) m[i] = new ll[c];
      REP(i, r) REP(j, c) m[i][j] = B[i][j];
      return *this;
   }
   Matrix operator * (const Matrix& B) const {
      Matrix rt(r, B.c);
      REP(i, r) REP(k, c) if (m[i][k] != 0) REP(j, B.c)
         rt[i][j] = (rt[i][j] + m[i][k] * B[k][j] % mod) % mod;
      return rt;
   }
   Matrix operator ^ (ll n) {
      Matrix rt(r, c); rt.e();
      Matrix ba(*this);
      while (n) {
         if (n & 1) rt = rt * ba;
         ba = ba * ba;
         n >>= 1;
      }
      return rt;
   }
   void out() {
      REP(i, r) REP(j, c)
      cout << m[i][j] << (j == c - 1 ? '\n' : ' ');
      cout << endl;
   }
};
```

6.3 高精度

```
// 加法 乘法 小于号 输出
struct bint {
   int l; short int w[100];
   bint(int x = 0) {
      l = x == 0; memset(w, 0, sizeof(w));
      while (x != 0) {w[l++] = x % 10; x /= 10;}
   }
   bool operator < (const bint& x) const {
      if (l != x.l) return l < x.l;
      int i = l - 1;
      while (i >= 0 && w[i] == x.w[i]) i--;
      return (i >= 0 && w[i] < x.w[i]);
   }
   bint operator + (const bint& x) const {
      bint ans; ans.l = l > x.l ? l : x.l;
      for (int i = 0; i < ans.l; i++) {
         ans.w[i] += w[i] + x.w[i];
         ans.w[i + 1] += ans.w[i] / 10;
         ans.w[i] = ans.w[i] % 10;
      }
      if (ans.w[ans.l] != 0) ans.l++;
      return ans;
   }
```

```
    bint operator * (const bint& x) const {
        bint res; int up, tmp;
        for (int i = 0; i < l; i++) {
            up = 0;
            for (int j = 0; j < x.l; j++) {
                tmp = w[i] * x.w[j] + res.w[i + j] + up;
                res.w[i + j] = tmp % 10;
                up = tmp / 10;
            }
            if (up != 0) res.w[i + x.l] = up;
        } res.l = l + x.l;
        while (res.w[res.l - 1] == 0 && res.l > 1) res.l--;
        return res;
    }
    void print() {
        for (int i = l - 1; i >= 0 ; i--)
printf("%d", w[i]);
        printf("\n");
    }
};
```