



---

# ACM-ICPC Template Libraries

---

合肥工业大学宣城校区



Author: Netcan

Blog: <http://www.netcan.xyz>

2015 年 12 月 7 日

# 目录

第一章 数学	3
1.1 $0 - 20$ 的阶乘	3
1.2 错排公式	3
1.3 最小公倍数 $lcm(a, b)$ && 最大公约数 $gcd(a, b)$	3
1.4 扩展欧几里得	3
1.5 母函数	4
1.6 动态规划	4
1.6.1 01 背包	4
1.6.2 多重背包	5
1.6.3 完全背包	5
1.6.4 最长公共子序列	6
1.6.5 最长上升子序列	6
1.6.6 多重部分和	6
1.6.7 多重集的组合数	7
1.6.8 划分数	7
1.7 高精度	8
1.7.1 加法 && 乘法	8
1.8 素数	9
1.8.1 埃式筛法 $O(n \log \log n)$	9
1.8.2 区间筛法	9
1.8.3 Miller_Rabin 素数判断 $O(\log N)$	10
1.8.4 Pollard_Rho 分解质因数 $O(\sqrt[4]{N})$	11
1.9 快速幂运算 $O(\log n)$	12

第二章 计算几何	13
2.1 点	13
第三章 组合博弈	14
3.1 SG 函数 && NIM 游戏	14
3.1.1 打表	14
3.1.2 递归	14
第四章 数据结构	16
4.1 二叉搜索树	16
4.2 并查集	17
第五章 字符串	19
5.1 KMP 算法 $O(M + N)$	19
5.2 字典树	20
5.3 AC 自动机	21
第六章 图	23
6.1 邻接表	23
6.1.1 样例 1	23
6.1.2 样例 2	23
6.2 拓扑排序	23
6.3 单源最短路	24
6.3.1 Bellman-Ford 算法 $O(V * E)$	24
6.3.2 SPFA 算法	24
6.3.3 Dijkstra 算法	25
6.3.4 差分约束系统	26
6.4 任意两点间的最短路	27
6.5 最小生成树	28
6.5.1 Prim 算法 $O(V^2)$	28
6.5.2 Kruskal 算法 $O(E \log(V))$	28

# 第一章 数学

## 1.1 $0 - 20$ 的阶乘

```
const long long fac[21]={1,1,2,6,24,120,720,5040,40320,362880,
3628800,39916800,479001600,6227020800,
87178291200,1307674368000,20922789888000,
355687428096000,6402373705728000,121645100408832000,
2432902008176640000};
```

## 1.2 错排公式

有  $n$  个元素的排列，若一个排列中所有的元素都不在自己原来的位置上，错排数记为  $D(n)$ ，则

$$D(n) = (n - 1)[D(n - 1) + D(n - 2)]$$

## 1.3 最小公倍数 $lcm(a, b)$ && 最大公约数 $gcd(a, b)$

```
inline int gcd(int a, int b) { // 如果a<b, 则递归得gcd(b,a%b)即gcd(b, a), 即交换了位置, 时间复杂度O(
    log max(a, b))
    return b==0?a:gcd(b,a%b);
}
inline int lcm(int a, int b) {
    return a/gcd(a,b)*b;
}
```

## 1.4 扩展欧几里得

求解  $ax + by = gcd(a, b)$  这里得到的是一组  $(x, y)$  的可行解,  $(x + kb, y - ka)$  为解集。

```
int extgcd(int a, int b, int &x, int &y) { // x, y为解, 返回gcd(a, b)
    int d = a;
    if(b!=0) {
        d = extgcd(b, a%b, y, x);
        y -= (a/b)*x;
    }
    else {
```

```

    x = 1; y = 0;
}
return d;
}

```

## 1.5 母函数

$G(x) = (1 + x + x^2 + \dots + x^N)(1 + x^2 + x^4 + \dots + x^N) \dots (1 + x^N)$  展开后  $x^N$  的系数(注意溢出)

```

int c1[MAX_N], c2[MAX_N]; // c1表示每一项的系数, c2表示每个表达式的临时系数
for(int i=0; i<=N; ++i) { // 每一项应该初始化为1,即1+x+x^2+...+x^N
    c1[i] = 1;
    c2[i] = 0;
}
for(int i=2; i<=N; ++i) { // 从第二个表达式开始
    for(int j=0; j<=N; ++j) // 表示第一个表达式的第j项
        for(int k=0; k+j<=N; k+=i) // k表示后一个表达式的第k项
            c2[j+k] += c1[j]; // 这里应该是相当于C1*x^j * x^k=C1*x^(j+k), 即c2[j+k]+=c1[j]
    for(int j=0; j<=N; ++j) {
        c1[j] = c2[j]; // 确定x^j的系数
        c2[j] = 0;
    }
}
}

```

若母函数形式为

$$G(x) = (1+x^{A_1}+x^{A_1*2}+\dots+x^{A_1*N_1})(1+x^{A_2}+x^{A_2*2}+\dots+x^{A_2*N_2})\dots(1+x^{A_n}+x^{A_n*2}+\dots+x^{A_n*N_n})$$

其中  $N_i$  为  $A_i$  的个数, 则注意初始化为 1 的时候, 第一项个数不要超过  $N_1$ , 且每隔  $A_1$  初始化为 1。

## 1.6 动态规划

### 1.6.1 01 背包

$N$  个物品重量和价值分别为  $w_i, v_i$ , 从这些物品中挑出总重量不超过  $W$  的物品, 求所有挑选方案中价值总和最大。设  $dp[i+1][j]$  为前  $i$  个物品总重量不超过  $j$  时最大价值

$$dp[i+1][j] = \begin{cases} dp[i][j] & j < w[i] \\ \max(dp[i][j-w[i]]+v[i], dp[i][j]) & j \geq w[i] \end{cases}$$

```

int dp[MAX_W + 1];
memset(dp, 0, sizeof(dp));
for(int i=0; i<N; ++i)
    for(int j=W; j>=w[i]; --j) // 注意逆序, 保证前面的是未使用的
        dp[j] = max(dp[j], dp[j-w[i]]+v[i]);

```

### 1.6.2 多重背包

$N$  种物品重量和价值个数分别为  $w_i, v_i, n_i$ ，从这些物品中挑出总重量不超过  $W$  的物品，求所有挑选方案中价值总和最大。

$$dp[i+1][j] = \max(dp[i][j - k \times w[i]] + k \times v[i] | 0 \leq k \leq n_i \&\& k \times w[i] \leq j)$$

相当于  $n_i$  个 01 背包。

```
int dp[MAX_W + 1];
memset(dp, 0, sizeof(dp));
for(int i=0; i<N; ++i)
    for(int j=0; j<=n[i]; ++j) // n[i]个01背包
        for(int k=W; k>=j*w[i]; --k)
            dp[k] = max(dp[k], dp[k-w[i]]+v[i]);
```

二进制优化下

```
int dp[MAX_W + 1];
memset(dp, 0, sizeof(dp));
for(int i=0; i<N; ++i) {
    int num = n[i];
    for(int j=1; j<=num; j<=1) {
        for(int k=W; k>=j*w[i]; --k)
            dp[k] = max(dp[k], dp[k-j*w[i]]+j*v[i]);
        num-=j;
    }
    if(num)
        for(int k=W; k>=num*w[i]; --k)
            dp[k] = max(dp[k], dp[k-num*w[i]]+num*v[i]);
}
```

### 1.6.3 完全背包

$N$  种物品重量和价值分别为  $w_i, v_i$ ，每种物品无限个，从这些物品中挑出总重量不超过  $W$  的物品，求所有挑选方案中价值总和最大。

$$\begin{aligned} dp[i+1][j] &= \max\{dp[i][j - k \times w[i]] + k \times v[i] | 0 \leq k\} \\ &= \max(dp[i][j], \max\{dp[i][j - k \times w[i]] + k \times v[i] | 1 \leq k\}) \\ &= \max(dp[i][j], \max\{dp[i][(j - w[i]) - k \times w[i]] + k \times v[i] | 0 \leq k\} + v[i]) \\ &= \max(dp[i][j], dp[i+1][j - w[i]] + v[i]) \end{aligned}$$

```
int dp[MAX_W + 1];
memset(dp, 0, sizeof(dp));
for(int i=0; i<N; ++i)
    for(int j=w[i]; j<=W; ++j) // 正序
        dp[j] = max(dp[j], dp[j-w[i]]+v[i]);
```

### 1.6.4 最长公共子序列

$$dp[i+1][j+1] = \begin{cases} dp[i][j] + 1 & s1[i] = s2[j] \\ \max(dp[i+1][j], dp[i][j+1]) & s1[i] \neq s2[j] \end{cases}$$

$$op[i+1][j+1] = \begin{cases} \nwarrow & s1[i] = s2[j] \\ \uparrow & dp[i][j+1] \geq dp[i+1][j] \\ \leftarrow & dp[i][j+1] < dp[i+1][j] \end{cases}$$

### 1.6.5 最长上升子序列

$$dp[i] = \max\{1, dp[j] + 1 | j < i \text{ \& \& } a_j < a_i\}$$

$O(N^2)$  算法

```
int a[MAX_N], dp[MAX_N]; // a为序列, dp[i]为以a[i]为结尾的最长上升子序列长度
int Maxlis = 0; // 最长上升子序列长度
int n; // a的有效长度
for(int i=0; i<n; ++i) {
    dp[i] = 1; // 记得初始化为1
    for(int j=0; j<i; ++j)
        if(a[i] > a[j])
            dp[i] = max(dp[i], dp[j] + 1);
    Maxlis = max(Maxlis, dp[i]); // 记得更新Maxlis
}
```

$O(n \log(n))$  算法

```
#define INF 0x3f3f3f3f
int a[MAX_N], dp[MAX_N]; // a为序列, dp[i]存放长度为i+1的上升序列中末尾元素的最小值
int n; // 序列长度
memset(dp, 0x3f, sizeof(dp)); // 初始化dp[i]值都为INF
for(int i=0; i<n; ++i)
    *lower_bound(dp, dp+n, a[i]) = a[i];
// cout << lower_bound(dp, dp+n, INF) - dp << endl; // 最长上升序列的长度
```

$lower\_bound(dp, dp+n, k)$  这个函数从已经排序好的序列  $dp$  中, 二分搜索找出满足  $dp_i \geq k$  的  $dp_i$  的最小指针。

$upper\_bound(dp, dp+n, k)$  这个函数则从已经排序好的序列  $dp$  中, 二分搜索找出满足  $dp_i > k$  的  $dp_i$  的最小指针。

例如求有序数组  $a$  中的  $k$  的个数, 可以利用下面的代码求出:

```
upper_bound(a, a+n, k) - lower_bound(a, a+n, k)
```

### 1.6.6 多重部分和

有  $n$  种大小不同的数字  $a_i$ , 每种  $m_i$  个, 判断是否可以从这些数字中选出若干个使他们的和恰好为  $K$ 。

设  $dp[i+1][j]$  为前  $i$  种数加和为  $j$  时第  $i$  种数最多能剩余多少个。(不能得到为-1)

$$dp[i+1][j] = \begin{cases} m_i & (dp[i][j] \geq 0) \\ -1 & (j < a_i \text{ Or } dp[i+1][j-a_i] \leq 0) \\ dp[i+1][j-a_i] - 1 & \text{Other} \end{cases}$$

```
int dp[MAX_K+1];
memset(dp, -1, sizeof(dp));
dp[0] = 0;
for(int i=0; i<n; ++i) {
    for(int j=0; j<=K; ++j) {
        if(dp[j] >= 0) dp[j] = m[i]; // 前i-1个数已经能凑成j了
        else if(j < a[i] || dp[j-a[i]] <= 0) dp[j] = -1; // 否则, 凑不成j或者a[i]已经用完, 则无法满足
        else dp[j] = dp[j-a[i]] - 1; // 否则可以凑成
    }
}
```

### 1.6.7 多重集的组合数

有  $n$  种物品, 第  $i$  种有  $a_i$  个。不同种类的物品可以相互区分但相同种类的物品无法区分。从这些物品中取出  $m$  个的话, 有多少种取法。结果取  $Mod$ 。

设  $dp[i+1][j]$  为从前  $i$  个物品取出  $j$  个的组合数

$$\begin{aligned} dp[i+1][j] &= \sum_{k=0}^{\min(j, a[i])} dp[i][j-k] \\ &= \sum_{k=0}^{\min(j-1, a[i])} dp[i][j-1-k] + dp[i][j] - dp[i][j-1-a[i]] \\ &= dp[i+1][j-1] + dp[i][j] - dp[i][j-1-a[i]] \end{aligned}$$

```
int dp[MAX_N+1][MAX_M+1];

for(int i=0; i<=n; ++i) dp[i][0] = 1; // 一个都不取
for(int i=0; i<n; ++i) {
    for(int j=1; j<=m; ++j) {
        if(j-1-a[i] >= 0)
            dp[i+1][j] = (dp[i+1][j-1] + dp[i][j] - dp[i][j-1-a[i]] + MOD) % MOD; // 避免减法出现负数
        else
            dp[i+1][j] = (dp[i+1][j-1] + dp[i][j]) % MOD;
    }
}
```

### 1.6.8 划分数

有  $n$  个无区别的物品, 将它们划分成不超过  $m$  组, 求出划分方法数, 取  $Mod$ 。

设  $dp[i][j]$  为  $j$  的  $i$  划分的总数。

$$dp[i][j] = dp[i][j-i] + dp[i-1][j]$$



## 1.7 高精度

### 1.7.1 加法 && 乘法

适合大数的加法和乘法

```
struct BigInt {
    const static int nlen = 4; // 控制每个数组数字长度，默认为4，计算乘法的时候每个数组相乘也不会溢出int范围
    const static int mod = 10000; // 值为10^nlen
    short n[1000], len; // 最多存4*1000位长度，可调，short占的内存小，但是速度慢
    BigInt() {
        memset(n, 0, sizeof(n));
        len = 1;
    }
    BigInt(int num) {
        len = 0;
        while(num > 0) {
            n[len++] = num % mod;
            num /= mod;
        }
    }
    BigInt(const char *s) {
        int l = strlen(s);
        len = l % nlen == 0 ? l/nlen : l/nlen+1;
        int index = 0;
        for(int i=l-1; i>=0; i -= nlen) {
            int tmp = 0;
            int j = i-nlen+1;
            if(j<0) j = 0;
            for(int k=j; k<=i; ++k)
                tmp = tmp*10+s[k]-'0';
            n[index++] = tmp;
        }
    }
    BigInt operator+(const BigInt &b) const { // 加法
        BigInt res;
        res.len = max(len, b.len);
        for(int i=0; i<res.len; ++i) {
            res.n[i] += (i<len ? n[i]:0) + (i<b.len ? b.n[i]:0);
            res.n[i+1] += res.n[i]/mod;
            res.n[i] = res.n[i]%mod;
        }
        if(res.n[res.len] > 0) ++res.len;
        return res;
    }
    BigInt operator*(const BigInt &b) const { // 乘法
        BigInt res;
        for(int i=0; i<len; ++i) { // 类似母函数，第一个数组
            int up = 0; // 进位
            for(int j=0; j<b.len; ++j) { // 第二个数组
                int tmp = n[i]*b.n[j] + up + res.n[i+j]; // 控制nlen=4是防止tmp溢出
                res.n[i+j] = tmp%mod;
                up = tmp/mod;
            }
            if(up!=0)
                res.n[i+b.len] = up;
        }
    }
};
```

```

    }
    res.len = len+b.len;
    while(res.n[res.len-1] == 0 && res.len>1 ) --res.len;
    return res;
}
void show() const {
    printf("%d", n[len-1]); // 先输出最高位, 后面可能需要前导0
    for(int i=len-2; i>=0; --i)
        printf("%04d", n[i]); // 前导0, %04d和nlen一致
    printf("\n");
}
};

```

## 1.8 素数

### 1.8.1 埃式筛法 $O(n\log\log n)$

```

bool is_prime[MAX_N]; // 第i个素数
int prime[MAX_N+1]; // is_prime[i]为true表示i是素数

int sieve(int n) { // 返回n以内的素数个数
    int p=0;
    memset(is_prime, true, sizeof(is_prime));
    is_prime[0] = is_prime[1] = false;
    for(int i=2; i<=n; ++i) {
        if(is_prime[i]) {
            prime[p++] = i;
            for(int j=2*i; j<=n; j+=i) is_prime[j] = false;
        }
    }
    return p;
}

```

### 1.8.2 区间筛法

筛选出区间  $[a, b]$  间的素数,  $a \geq 1$

```

typedef long long ll;
bool is_prime[MAX_L]; // 对区间[a, b]内的整数筛选, is_prime[i-a] == true表示i是素数, MAX_L = B-A+1
bool is_prime_small[MAX_SQRT_B];

void segment_sieve(ll a, ll b) {
    memset(is_prime, true, sizeof(is_prime));
    memset(is_prime_small, true, sizeof(is_prime_small));
    if(a == 1) is_prime[0] = false; // 1不是素数

    for(int i=2; (ll)i*i <= b; ++i) {
        if(is_prime_small[i]) {
            for(int j=2*i; (ll)j*j <= b; j+=i) is_prime_small[j] = false; // 筛[2, √b]
            for(ll j = max(2LL, (a+i-1)/i)*i; j<=b; j+=i) is_prime[j-a] = false;
        }
    }
}

```

```

}

```

### 1.8.3 Miller\_Rabin 素数判断 $O(\log N)$

```

ll mod_mult(ll a, ll b, ll mod) { //return a*b%mod
    ll s = 0;
    while(b) {
        if(b&1) s=(s+a)%mod;
        a<<=1, b>>=1;
        if(a > mod) a %= mod;
    }
    return s;
}

ll mod_pow(ll x, ll n, ll mod) { // a^b%mod
    ll res = 1;
    while(n) {
        if(n&1) res = mod_mult(res, x, mod);
        x=mod_mult(x, x, mod);
        n>>=1;
    }
    return res;
}

//以a为基,n-1=u*2^t a^(n-1)=1(mod n) 验证n是不是合数
//一定是合数返回true,不一定返回false
bool witness(ll a, ll n, ll u, ll t) { // a随机数, u, t于外部计算
    ll ret = mod_pow(a, u, n);
    ll last = ret;
    for(int i=1; i<=t; ++i) {
        ret = mod_mult(ret, ret, n);
        if(ret == 1 && last != 1 && last != n-1) return true;
        last = ret;
    }
    if(ret!=1) return true; // a^(n-1)!=1(mod n)
    return false;
}

// Miller_Rabin()算法素数判定
//是素数返回true.(可能是伪素数, 但概率极小, P=(1/2)^s)
//合数返回false;
// s:挑选s个随机数
bool Miller_Rabin(ll n, int s) {
    if(n==2) return true;
    if(n < 2 || !(n&1)) return false;
    ll u=n-1;
    ll t=0;
    while(!(u&1)) {u>>=1; ++t;} // 2^t*u = n-1
    for(int i=0; i<s; ++i) {
        ll a=rand()%(n-1)+1; // 若n为合数, 证据为a的概率至少为1/2
        if(witness(a, n, u, t)) return false; // 合数
    }
    return true;
}

```

### 1.8.4 Pollard\_Rho 分解质因数 $O(\sqrt[4]{N})$

```
vector<ll> factors; // 质因数分解结果 (无序)
ll Pollard_rho(ll n, ll c) { // 伪随机函数 f(x)=x^2+c, c为随机数
    ll i=1, k=2;
    ll x=rand()%n;
    ll y = x;
    while(1) {
        ++i;
        x=(mod_mult(x, x, n)+c)%n; // 伪随机数
        ll d = gcd(y-x+n, n); // gcd注意负数
        if(d!=1 && d!=n) return d;
        if(y==x) return n; // 遇环退出
        if(i==k) {y=x; k<<=1;}
    }
}
// 对n进行递归素因子分解
void findfac(ll n) {
    if(Miller_Rabin(n, 20)) { // 素数
        factors.push_back(n); // 储存素因子
        return;
    }
    ll p = n;
    while(p>=n) p = Pollard_rho(p, rand()%(n-1)+1); // 找出当前合数的一个素因子
    findfac(p);
    findfac(n/p);
}
```

## 1.9 快速幂运算 $O(\log n)$

```
typedef long long ll;
ll mod_pow(ll x, ll n, ll mod) {
    ll res = 1;
    while(n > 0) {
        if(n&1) res = res * x % mod; // 如果最低位为1, 则乘上x^(2^i)
        x=x * x % mod; // x平方
        n >>= 1;
    }
    return res;
}
```

递归写法：

```
typedef long long ll;
ll mod_pow(ll x, ll n, ll mod) {
    if(n == 0) return 1;
    ll res = mod_pow(x * x % mod, n/2, mod);
    if(n & 1) res = res * x % mod;
    return res;
}
```

## 第二章 计算几何

### 2.1 点

## 第三章 组合博弈

### 3.1 SG 函数 && NIM 游戏

1. 可选步数为  $[1, m]$  的连续整数，直接取模即可， $SG(x) = x\%(m+1)$ ;
2. 可选步数为任意步， $SG(x) = x$ ;

步数集合  $S$  需排序（升序）， $sg$  数组记得初始化，对于确定的步数，一系列  $sg(x)$  值也就确定了。

最终判断各个  $sg(x)$  的异或和，即可判断胜负。异或和为 0 先手必败，反之必胜。

#### 3.1.1 打表

```
int S[STEP_N], steps, sg[MAX_N]; // S集合存放走法, steps存放走法数, sg存放sg(x)的值
bool vis[MAX_N]; // 标记
void get_sg(int n) {
    memset(sg, 0, sizeof(sg)); // 初始化sg
    for(int i=1; i<=n; ++i) { // 从sg[1]开始计算
        memset(vis, 0, sizeof(vis)); // 每次计算完一个sg值需要归零
        for(int j=0; S[j] <= i && j < steps; ++j)
            vis[sg[i-S[j]]] = true; // 标记各个后继节点的sg值
        for(int j=0; j<=n; ++j)
            if(!vis[j]) { // 找出sg补集的最小值
                sg[i] = j;
                break;
            }
    }
}
```

#### 3.1.2 递归

```
#pragma comment(linker, "/STACK:1024000000,1024000000") // 防止爆栈
int S[STEP_N], sg[MAX_N], k; // 题目中的步数集合S, 以及sg(t)函数值sg(t), 步数集合大小k

int SG(int p) { // 求sg(t)值函数SG(t)
    bool vis[101] = {false}; // 标记各个sg(t)的值, 为了方便求补集最小值(sg(t)), 数组不宜开过大, 爆栈就扩栈
    for(int i=0; i<k; ++i) {
        int t = p - S[i];
        if(t < 0) // 小于0则退出循环, 求出该层的sg(t)值
            continue;
        vis[SG(t)] = true;
    }
    for(int i=0; i<101; ++i)
        if(!vis[i]) return i;
}
```

```
        break;
    if(sg[t] == -1) // 记得memset(sg, -1, sizeof(sg));
        sg[t] = SG(t); // 递归求sg(t)
    vis[sg[t]] = true; // 标记该层的sg(t)值
}
for(int i=0;; ++i) // 求出该层的sg(t)值，即补集的最小值
    if(!vis[i])
        return i;
}
```

## 第四章 数据结构

### 4.1 二叉搜索树

二叉搜索树是能够高效地进行如下操作的数据结构：

- 插入一个数值
- 查询是否包含某个数值
- 删除某个数值

时间复杂度： $O(\log(n))$

```
struct node { // 树节点
    int val;
    node *lch, *rch;
};

node *insert(node *p, int x) { //插入数值x
    if(p == NULL) { // 新建节点插入
        node *q = new node;
        q->val = x;
        q->lch = q->rch = NULL;
        return q;
    }
    else {
        if(x < p->val) p->lch = insert(p->lch, x); // 往左边搜索
        else p->rch = insert(p->rch, x); // 往右边搜索
        return p;
    }
}

bool find(node *p, int x) { // 查找数值x
    if(p == NULL) return false; // 找不到
    else if(p->val == x) return true; // 找到
    else if(x < p->val) return find(p->lch, x); // 往左边搜索
    else return find(p->rch, x); // 往右边搜索
}

node *remove(node *p, int x) { // 删除数值x
    if(p == NULL) return NULL; // 找不到数值
    else if(x < p->val) p->lch=remove(p->lch, x); // 往左边搜索
    else if(x > p->val) p->rch=remove(p->rch, x); // 往右边搜索
    else { // 找到
        if(p->lch == NULL) { // 如果删除的节点没有左儿子,将右儿子提上来
```



```

    node *q = p->rch;
    delete p; // 删除
    return q;
}
else if(p->lch->rch == NULL) { // 如果删除的节点左儿子没有右儿子,将左儿子提上来
    node *q = p->lch;
    q->rch = p->rch;
    delete p; // 删除
    return q;
}
else { // 否则, 将左儿子的子孙中最大的节点提上来
    node *q;
    for(q=p->lch; q->rch->rch; q=q->rch); // 往左儿子搜索最大节点
    node *r = q->rch; // r指向左儿子最大子孙节点,q指向最大儿子的父亲
    q->rch = r->lch; // 因为r为提上去的节点, 将r的左儿子(有的话,否则为NULL)挂到q的右边
    r->lch = p->lch;
    r->rch = p->rch;
    delete p; // 删除
    return r;
}
}
return p;
}
}
/*****Usage*****/
node *testbst=NULL; // 初始化
testbst = insert(testbst, x); // 插入数值x
if(find(testbst, x)) // 查找数值x
    // balabala
else
    // balabala
testbst = remove(testbst, x); // 删除数值x

```

## 4.2 并查集

并查集是一种用来管理元素分组情况的数据结构，可以高效地进行如下两种操作：

- 合并两个集合
- 查找某元素属于哪个集合

时间复杂度： $O(\alpha(n))$

```

int par[MAX_N];
// int height[MAX_N];
void init(int n) { // 初始化
    for (int i = 1; i <= n; ++i) {
        par[i] = i;
        // height[i] = 0;
    }
}
int find(int x) { // 查找根节点(集合)+路径压缩
    return x==par[x]?x:par[x]=find(par[x]);
}
void unite(int x, int y) { // 合并集合
    x = find(x);

```

```
y = find(y);
if(x!=y) {
    par[x] = y;
    // if(height[x] < height[y])
    // par[x] = y;
    // else
    // par[y] = x;
    // if(height[x] == height[y]) ++height[x];
}
}
bool same(int x,int y) { // 判断两个元素是否同集合
    return find(x) == find(y);
}
```

## 第五章 字符串

### 5.1 KMP 算法 $O(M + N)$

<http://www.cnblogs.com/goagent/archive/2013/05/16/3068442.html>

求出模式串  $P$  在字符串  $S$  中的位置当失配前 ( $P_j \neq S_i$ ), 满足

$$P_0 P_1 \cdots P_{k-1} = S_{i-k} S_{i-k+1} \cdots S_{i-1}$$

已有部分匹配结果,

$$P_{j-k} P_{j-k+1} \cdots P_{j-1} = S_{i-k} S_{i-k+1} \cdots S_{i-1}$$

由上两式子可得:

$$P_0 P_1 \cdots P_{k-1} = P_{j-k} P_{j-k+1} \cdots P_{j-1}$$

定义  $next[j] = k$ , 有

$$next[j] = \begin{cases} -1 & j = 0 \\ \max\{k | 0 < k < j \ \&\& \ P_0 P_1 \cdots P_{k-1} = P_{j-k} P_{j-k+1} \cdots P_{j-1}\} & \\ 0 & Other \ Conditions \end{cases}$$

```
int Next[MAX_PLEN];
string s, p; // 求出模式串p在字符串s中的位置
void getNext() {
    Next[0] = -1;
    int i=0, j=-1;
    while(i!=p.length()-1) {
        if(j == -1 || p[j] == p[i]) {
            ++i;
            ++j;
            Next[i] = p[i] != p[j]?j:Next[j];
        }
        else
            j = Next[j];
    }
}

int kmp(const int sIndex = 0) // 返回模式串位置
{
    getNext();
    int i = sIndex, j = 0;
    while(i != s.length() && j != p.length()) {
        if (j == -1 || s[i] == p[j]) {
```

```

        ++i;
        ++j;
    }
    else
        j = next[j];
}
return j == p.length() ? i - j: -1;
}

```

## 5.2 字典树

[http://www.cnblogs.com/tanky\\_woo/archive/2010/09/24/1833717.html](http://www.cnblogs.com/tanky_woo/archive/2010/09/24/1833717.html)

```

const int MAX_C = 26; // 假设全为小写字母
struct Trie {
    Trie *next[MAX_C];
    int v; // 存放以此为前缀的个数
    Trie() {
        for(int i=0; i<MAX_C; ++i) next[i] = NULL;
    }
};
Trie *root = new Trie;

void createTrie(const string &str) { // 建立字典树
    Trie *p = root, *q;
    for(int i=0; i<str.length(); ++i) {
        int id = str[i] - 'a';
        if(p->next[id] == NULL) {
            q = new Trie;
            q->v = 1;
            p->next[id] = q;
            p = p->next[id];
        }
        else {
            ++(p->next[id]->v);
            p = p->next[id];
        }
    }
    // p->v = -1; // 结尾
}

int findTrie(const string &w) { // 查找字典树
    Trie *p = root;
    for(int i=0; i<w.length(); ++i) {
        int id = w[i] - 'a';
        p = p->next[id];
        if(p == NULL) return 0; // 空集, 则不存在以此为前缀的串
    }
    return p->v; // 此字符串是字符集某串的前缀
}

void freeTrie(Trie *p) { // 释放内存
    if(p == NULL) return;
}

```

```

for(int i=0; i<MAX_C; ++i) freeTrie(p->next[i]);
delete p;
}

```

## 5.3 AC 自动机

参考 kuangbin 模板。静态版。

```

struct Trie {
    static const int max_L = 500010;
    static const int max_c = 26;
    int cnt[max_L]; // 单词结尾节点统计该单词个数
    int fail[max_L]; // fail指针
    int next[max_L][max_c];
    int root, L; // 根指针, 当前最大有效节点指针
    int newnode() {
        for(int i=0; i<max_c; ++i)
            next[L][i] = -1;
        cnt[L++] = 0;
        return L-1;
    }
    void init() {
        L = 0;
        root = newnode();
    }

    void insert(const char *s) { // 建立Trie树
        int len = strlen(s);
        int p = root;
        for(int i=0; i<len; ++i) {
            int id = s[i] - 'a';
            if(next[p][id] == -1) next[p][id] = newnode();
            p = next[p][id];
        }
        ++cnt[p];
    }

    void build() {
        queue<int> que;
        fail[root] = root; // 根fail指针指向根, 避免多余的判断
        for(int i=0; i<max_c; ++i) // 处理root的后继节点, 都指向root
            if(next[root][i] == -1) next[root][i] = root;
            else {
                fail[next[root][i]] = root;
                que.push(next[root][i]);
            }

        while(!que.empty()) {
            int p = que.front(); que.pop();

            for(int i=0; i<max_c; ++i)
                if(next[p][i] == -1) next[p][i] = next[fail[p]][i]; // 方便后面match失配的fail转移
                else {
                    fail[next[p][i]] = next[fail[p]][i];
                    que.push(next[p][i]);
                }
        }
    }
}

```

```
        }  
    }  
}  
  
int match(const char *s) {  
    int len = strlen(s);  
    int p = root;  
    int res = 0;  
    for(int i=0; i<len; ++i) {  
        int id = s[i] - 'a';  
        p = next[p][id];  
        int tmp = p;  
        while(tmp != root) {  
            res += cnt[tmp];  
            cnt[tmp] = 0;  
            tmp = fail[tmp];  
        }  
    }  
    return res;  
}  
};
```

# 第六章 图

## 6.1 邻接表

### 6.1.1 样例 1

```
#define MAX_V 100
vector<int> G[MAX_V];

/* 边上有属性
 * struct edge { int to, cost; };
 * vector<edge> G[MAX_V];
 */

int main()
{
    int V, E;
    cin >> V >> E;
    for(int i=0; i<E; ++i) {
        int s, t;
        cin >> s >> t;
        G[s].push_back(t); // s->t
        // G[s].push_back(edge(t, c));
        // G[t].push_back(s); // 无向图
    }
    // balabala...
}
```

### 6.1.2 样例 2

## 6.2 拓扑排序

首先将入度为 0 的顶点入队，然后在队列中依次取出顶点并删除出边（将指向的顶点入度-1），若指向节点入度为 0，则入队。直到队列为空。

若输出结果小于顶点数，则说明有圈。

```
int V, E;
vector<int> G[MAXV]; // 邻接表
vector<int> order; // 拓扑排序结果
int indegree[MAXV]; // 入度，注意重边

void top_sort() {
```

```

order.clear();
priority_queue<int, vector<int>, greater<int> > que; // 最小堆, 按照从小到大的顺序输出
for(int u=1; u<=N; ++u) if(indegree[u] == 0) que.push(u); // 将入度为0的顶点入队
int u;
while(!que.empty()) { // 直到队列为空
    u = que.top(); que.pop(); // 出队
    order.push_back(u); // 记录拓扑排序结果
    for(int i = 0; i<G[u].size(); ++i) { // 注意重边
        --indegree[G[u][i]]; // 删除出边
        if(indegree[G[u][i]] == 0) que.push(G[u][i]); // 入度为0, 入队
    }
}
}

```

## 6.3 单源最短路

### 6.3.1 Bellman-Ford 算法 $O(V * E)$

可以处理负圈。

```

struct Edge{ // 边
    int from, to, cost; // 顶点from指向to权值为cost
} edge[MAX_E];
int d[MAX_V];
int V, E; // 节点数量V, 边的数量E

bool bellman_ford(int s) { // 求解顶点s出发到所有节点的最短距离
    memset(d, 0x3f, sizeof(d)); // 初始化到INF
    d[s] = 0;
    for(int i=1; i<=V-1; ++i) // 图的顶点编号从1开始计算
        for(int j=1; j<=E; ++j) {
            Edge e=edge[j];
            if(d[e.to] > d[e.from] + e.cost) // 松弛计算
                d[e.to] = d[e.from] + e.cost;
        }

    int flag = true; // 判断有没有负圈
    for(int j=1; j<=E; ++j)
        if(d[edge[j].to] > d[edge[j].from] + edge[j].cost) {
            flag = false;
            break;
        }
    return flag;
}

```

### 6.3.2 SPFA 算法

可以判断负圈。据说最坏复杂度  $O(VE)$

<http://www.nocow.cn/index.php/SPFA%E7%AE%97%E6%B3%95>

```

struct edge {
    int to, cost;
}

```



```

    edge(int to, int cost) : to(to), cost(cost) {}
};

int V, E; // 节点数, 边数
int d[MAXV]; // 单源最短距离
vector<edge> G[MAXV]; // 图, 邻接表
bool vique[MAXV]; // 判断节点是否已经在队列中
int cnt[MAXV]; // 记录每个节点入队次数, 超过V则退出 (有负圈)。

bool SPFA(int s) {
    memset(d, 0x3f, sizeof(d));
    memset(vique, 0, sizeof(vique));
    memset(cnt, 0, sizeof(cnt));
    d[s] = 0;
    queue<int> que; // 入队, 存储SPFA需要松弛计算的节点
    que.push(s);
    vique[s] = true;
    cnt[s] = 1;
    while(!que.empty()) {
        int from = que.front(); que.pop();
        vique[from] = false;
        for(int i=0; i<G[from].size(); ++i) {
            edge &t = G[from][i]; // 据说用引用/指针可以提高寻址速度。。
            if(d[t.to] > d[from] + t.cost) {
                d[t.to] = d[from] + t.cost; // 松弛计算
                if(!vique[t.to]) { // 该节点未入队, 将其入队
                    que.push(t.to);
                    vique[t.to] = true;
                    ++cnt[t.to]; // 入队次数加一
                    if(cnt[t.to] > V) { // 该节点松弛计算次数大于总节点数, 有负边
                        // while(!que.empty()) que.pop();
                        return false;
                    }
                }
            }
        }
    }
    return true;
}

```

### 6.3.3 Dijkstra 算法

$$O(V^2)$$

```

int cost[MAXV][MAXV]; // cost[u][v]表示e={u,v}的权值 (不存在则INF)
int d[MAXV]; // 顶点s出发的最短距离
bool used[MAXV]; // 标记已经使用过的顶点
int V, E; // 顶点数V, 边数E

void dijkstra(int s) { // 源点s
    memset(d, 0x3f, sizeof(d)); // 初始化至INF
    memset(used, 0, sizeof(used)); // 初始化至INF
    d[s] = 0;

    while(true) {
        int v = -1;

```

```

    for(int u=1; u<=V; ++u) // 从未使用过的节点中选择一个距离最小的顶点, 编号从1开始
        if(!used[u] && (v==-1 || d[u] < d[v])) v = u;
    if(v == -1) break; // 已经用完所有顶点了
    used[v] = true; // 标记顶点
    for(int u=1; u<=V; ++u) // 顶点编号从1开始计算
        d[u] = min(d[u], d[v]+cost[v][u]);
}
}

```

$O(E \log V)$

```

struct edge { // 顶点属性
    int to, val;
    edge(int t, int v): to(t), val(v){}
    bool operator<(const edge &b) const {
        return val > b.val;
    }
};

vector<edge> G[MAX_V]; // 邻接链表图
int d[MAX_V];
int V, E; // 顶点数V, 边数E

void dijkstra(int s) {
    priority_queue<edge> que;
    memset(d, 0x3f, sizeof(d));
    d[s] = 0;
    que.push(edge(s, 0)); // 源点入队

    while(!que.empty()) {
        edge p = que.top(); que.pop();
        int v = p.to;
        if(d[v] < p.val) continue; // 当前最小值不是最短距离的话, 丢弃
        for(int i=0; i<G[v].size(); ++i) {
            edge e = G[v][i];
            if(d[e.to] > d[v] + e.val) {
                d[e.to] = d[v] + e.val;
                que.push(edge(e.to, d[e.to]));
            }
        }
    }
}
}

```

#### 6.3.4 差分约束系统

如果一个系统由  $n$  个变量和  $m$  个约束条件组成, 其中每个约束条件形如  $x_j - x_i \leq b_k$  ( $i, j \in [1, n], k \in [1, m]$ ), 则称其为差分约束系统 (system of difference constraints)。亦即, 差分约束系统是求解关于一组变量的特殊不等式组的方法。

求解差分约束系统, 可以转化成图论的单源最短路径 (或最长路径) 问题。

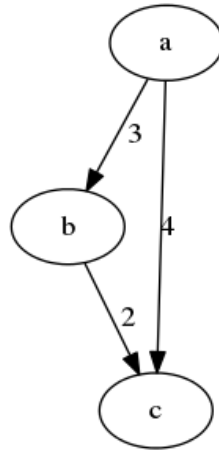
观察  $x_j - x_i \leq b_k$ , 会发现它类似最短路中的三角不等式  $d[v] \leq d[u] + w[u, v]$ , 即  $d[v] - d[u] \leq w[u, v]$ 。因此, 以每个变量  $x_i$  为结点, 对于约束条件  $x_j - x_i \leq b_k$ , 连接一条边  $(i, j)$ , 边权为  $b_k$ 。我们再增加一个源点  $s$ ,  $s$  与所有定点相连, 边权均为  $0$ 。对这个图, 以  $s$  为源点运行 Bellman-ford 算法 (或 SPFA 算法), 最终  $d[i]$  即为一组可行解。

例如：

$$\begin{cases} b - a \leq 3 & (1) \\ c - b \leq 2 & (2) \\ c - a \leq 4 & (3) \end{cases}$$

求出  $c-a$  的最大值。

建立如下有向图



根据条件有

$$\begin{cases} c - a \leq 4 & (4) \\ c - a \leq 3 + 2 = 5 & (5) \end{cases}$$

最短路即为  $c-a$  的最大值，即 4。

## 6.4 任意两点间的最短路

Floyd  $O(V^3)$

$$d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$$

和 Bellman-Ford 算法一样可以处理负圈，只需检查  $d[i][i]$  是否负数的顶点即可。

```

int V, E; // 顶点数V, 边数E
int d[102][102]; // d[u][v]表示边e={u, v}的权值 (不存在则设为INF, d[u][u]=0)

void floyd() {
    for(int k=0; k<V; ++k) // 顶点依次从0-V-1开始
        for(int i=0; i<V; ++i)
            for(int j=0; j<V; ++j)
                d[i][j] = min(d[i][j], d[i][k]+d[k][j]); // i到j的最短距离等于i到j的距离与i到k和k到j的距
                离的最小值
    return;
}
  
```

## 6.5 最小生成树

### 6.5.1 Prim 算法 $O(V^2)$

```
int cost[MAX_V][MAX_V]; // cost[u][v]表示边e=(u, v)的权值(不存在则INF)
int mincost[MAX_V]; // 从集合出发的边到每个顶点的最小权值
bool used[MAX_V]; // 顶点u是否在集合中
int V; // 顶点数

int prim() {
    int res = 0;
    memset(used, 0, sizeof(used));
    memset(mincost, 0x3f, sizeof(mincost));
    mincost[0] = 0;
    while(true) {
        int v = -1;
        for(int u=0; u<V; ++u)
            if(!used[u] && (v==-1 || mincost[u] < mincost[v])) v = u;
        if(v == -1) break;
        used[v] = true; // 标记顶点到集合中
        res += mincost[v];
        for(int u=0; u<V; ++u)
            mincost[u] = min(mincost[u], cost[v][u]);
    }
    return res;
}
```

### 6.5.2 Kruskal 算法 $O(E\log(V))$

```
int V, E; // 边数, 顶点数
struct edge { // 边
    int u, v, cost;
    bool operator<(const edge &b) const {
        return cost < b.cost; // 需要按照边的权值从小到大的顺序排序
    }
} es[MAX_E];

int kruskal() {
    sort(es, es+E); // 排序
    init_union_find(V); // 初始化并查集
    int res = 0;
    for(int i=0; i<E; ++i) {
        edge e = es[i];
        if(find(e.u) != find(e.v)) { // 判断是否产生圈(重边也算在内)
            unite(e.u, e.v);
            res+=e.cost;
        }
    }
    return res;
}
```