ACM-ICPC Template Libraries

合肥工业大学宣城校区

Author: Netcan

Blog: http://www.netcan.xyz

2015 年 8 月 5 日

目录

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) {
   return b==0?a:gcd(b,a%b)
}
inline int lcm(int a, int b) {
   return a/gcd(a,b)*b;
}
```

1.4 母函数

$G(x) = (1 + x + x^2 + \cdots + x^N)(1 + x^2 + x^4 + \cdots + x^N)\ldots(1 + x^N)$ 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[k+j]+=C1[j]
    for(int j=0; j<=N; ++j) {
        c1[j] = c2[j]; // 确定x^j的系数
        c2[j] = 0;
    }
}
```

1.5 动态规划

1.5.1 最长公共子序列

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

1.6 高精度

1.6.1 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");
    }
};
```

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
         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); // 往左边搜索
}

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　图

5.1　邻接表

5.1.1　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...
}
```

5.1.2　样例 2