A
C
M

算法模板

Code by yeates

目录

1 数据结构

1.1 二分

最简单的二分查找

```
while (low <= high)
{
    int mid = (low + high) / 2;
    if (dp[mid] >= tmp)   high = mid - 1;
    else low = mid + 1;
}
dp[low] = tmp;

//几种二分查找(O(logn))，区别在于是否数组中含有多个相同的查找值
//lower_bound返回等于item的第一个位置
template<typename T>
int mine_lower_bound(T *a, T item, int n) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (r - l) / 2 + l;
        if (a[m] >= item)  r = m;
        else    l = m + 1;
    }
    if (a[l] == item)    return l;
    else return -1;
}

//upper_bound返回等于item的最后一个位置的下一个位置。。
template<typename T>
int mine_upper_bound(T *a, T item, int n) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (r - l) / 2 + l;
        if (a[m] > item)   r = m;
        else l = m + 1;
    }
    if (a[l - 1] == item)   return l - 1;
    else return -1;
}

template<typename T>
int mine_binary_search(T *a, T item, int n) {
    int l = 0, r = n - 1;
    while (l < r) {
        int m = (r - l) / 2 + l;
        if (a[m] == item)  return m;
        else if (a[m] > item) r = m - 1;
        else l = m + 1;
    }
    return -1;
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int n;
    while (cin >> n) {
        int _array[MAX];
        REP(i, n)   cin >> _array[i];
        sort(_array, _array + n);
        REP(i, n)   fcout(_array[i], i, n);
        int item;
        cin >> item;
        cout << "upper_bound:" << endl;
        cout << mine_upper_bound(_array, item, n) << endl << endl;
        cout << "lower_bound:" << endl;
        cout << mine_lower_bound(_array, item, n) << endl << endl;
        cout << "binary_search:" << endl;
        cout << mine_binary_search(_array, item, n) << endl << endl;
    }
    return 0;
}
```

1.2 线段树

1.2.1 基础线段树(单点更新)

a[]是线段树；
build()是初始化线段树：在叶子结点输入值；
update()是从叶子结点开始改变线段树上的值，变化值即为 data；query()的代码类似于 build()，查询[l, r]段的值。

```
struct node {
    int l, r;
    int sum;
}a[4 * 50005];
void build(int i, int l, int r) {
    a[i].l = l;
    a[i].r = r;
    int mid = (l + r) / 2;
    if (l == r) {
        scanf("%d", &a[i].sum);
        return;
    }
    build(i * 2, l, mid);
    build(i * 2 + 1, mid + 1, r);
```

```
    a[i].sum = a[i * 2].sum + a[i * 2 + 1].sum;
}
void updata(int i, int index, int data) {
    if (a[i].l == a[i].r) {
        a[i].sum += data;
        return;
    }
    int mid = (a[i].l + a[i].r) / 2;
    if (index <= mid)  updata(i * 2, index, data);
    else    updata(i * 2 + 1, index, data);
    a[i].sum = a[i * 2].sum + a[i * 2 + 1].sum;
}
int query(int i, int l, int r) {
    int mid = (a[i].l + a[i].r) / 2;
    int sum = 0;
    if (a[i].l == l&&a[i].r == r)    return a[i].sum;
    if (r <= mid)    return query(i * 2, l, r);
    else if (l > mid)   return query(i * 2 + 1, l, r);
    else {
        sum += query(i * 2, l, mid);
        sum += query(i * 2 + 1, mid + 1, r);
        return sum;
    }
}
```

1.2.2优化线段树(成段更新)

暂时记忆以减少复杂度，可实现成段更新

```
typedef long long LL;
struct node {
    int l, r, m;
    LL sum, mark;
}T[maxn << 2];
int a[maxn];
void build(int id, int l, int r) {
    T[id].l = l;   T[id].r = r;  T[id].m = (l + r) >> 1;  T[id].mark = 0;
    if (l == r) { T[id].sum = a[l]; return; }
    int m = (l + r) >> 1;
    build(id << 1, l, m);  build((id << 1) + 1, m + 1, r);
    T[id].sum = (T[id << 1].sum + T[(id << 1) + 1].sum);
}
void update(int id, int l, int r, int val) {
    if (T[id].l == l&&T[id].r == r) {
        T[id].mark += val; return;
    }
```

```
    T[id].sum += (LL)val*(r - l + 1);
    if (T[id].m >= r)
        update(id << 1, l, r, val);
    else if (T[id].m<l)
        update((id << 1) + 1, l, r, val);
    else {
        update(id << 1, l, T[id].m, val);
        update((id << 1) + 1, T[id].m + 1, r, val);
    }
}
LL query(int id, int l, int r) {
    if (T[id].l == l&&T[id].r == r)  return T[id].sum + T[id].mark*(LL)(r - l + 1);
    if (T[id].mark != 0) {
        T[id << 1].mark += T[id].mark;
        T[(id << 1) + 1].mark += T[id].mark;
        T[id].sum += (LL)(T[id].r - T[id].l + 1)*T[id].mark;  T[id].mark = 0;
    }

    if (T[id].m >= r) {
        return query(id << 1, l, r);
    }
    else if (T[id].m<l) {
        return query((id << 1) + 1, l, r);
    }
    else {
        return query(id << 1, l, T[id].m) + query((id << 1) + 1, T[id].m + 1, r);
    }
}
```

1.2.3线段数区间合并

例子：给一个由 0,1 组成的序列，有两种操作，一种是翻转给定区间的数(0->1,1->0)，另一种是查询给定区间内由 1 组成的子串的最大长度。重点在**区间合并**和**延迟标记**。

```
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
#define lson l,m,rt<<1
#define rson m+1,r,rt<<1|1
const int maxn = 111111;
int sum[maxn << 2];
int lb[maxn << 2], rb[maxn << 2];
int col[maxn << 2];
int wm[maxn << 2];
```

```
int wl[maxn << 2];
int wr[maxn << 2];
int max(int a, int b) {
    return a>b ? a : b;
}
int Min(int a, int b) {
    return a<b ? a : b;
}
void pushup(int m, int rt) {
    wl[rt] = wl[rt << 1];
    lb[rt] = lb[rt << 1];
    wr[rt] = wr[rt << 1 | 1];
    rb[rt] = rb[rt << 1 | 1];

    if (wl[rt] == m - (m >> 1)) wl[rt] += wl[rt << 1 | 1];
    if (lb[rt] == m - (m >> 1)) lb[rt] += lb[rt << 1 | 1];

    if (wr[rt] == (m >> 1)) wr[rt] += wr[rt << 1];
    if (rb[rt] == (m >> 1))   rb[rt] += rb[rt << 1];

    wm[rt] = max(wm[rt << 1], wm[rt << 1 | 1]);
    sum[rt] = max(sum[rt << 1], sum[rt << 1 | 1]);
    wm[rt] = max(wm[rt], wr[rt << 1] + wl[rt << 1 | 1]);
    sum[rt] = max(sum[rt], rb[rt << 1] + lb[rt << 1 | 1]);
}
void make(int rt) {
    swap(sum[rt], wm[rt]);
    swap(wl[rt], lb[rt]);
    swap(wr[rt], rb[rt]);
}
void pushdown(int rt) {
    if (col[rt]) {
        col[rt << 1] ^= 1;
        col[rt << 1 | 1] ^= 1;
        col[rt] = 0;
        make(rt << 1);
        make(rt << 1 | 1);
    }
}
int num[100001];
void build(int l, int r, int rt) {
    col[rt] = lb[rt] = rb[rt] = wl[rt] = wr[rt] = sum[rt] = wm[rt] = 0;
    if (l == r) {
        if (num[l] == 1) sum[rt] = lb[rt] = rb[rt] = 1;
        else {
            wm[rt] = wl[rt] = wr[rt] = 1;
        }return;
```

```
    }
    int m = (l + r) >> 1;
    build(lson);
    build(rson);
    pushup(r - l + 1, rt);
}
void update(int L, int R, int l, int r, int rt) {
    if (L <= l&&r <= R) {
        col[rt] ^= 1;
        make(rt);
        return;
    }
    pushdown(rt);
    int m = (l + r) >> 1;
    if (L <= m) update(L, R, lson);
    if (R>m) update(L, R, rson);
    pushup(r - l + 1, rt);
}
int query(int L, int R, int l, int r, int rt) {
    if (L <= l&&r <= R) {
        return sum[rt];
    }
    pushdown(rt);
    int m = (l + r) >> 1;
    if (R <= m)
        return query(L, R, lson);
    if (L > m)
        return query(L, R, rson);
    int t1 = query(L, R, lson);
    int t2 = query(L, R, rson);
    int a = Min(m - L + 1, rb[rt << 1]);
    int b = Min(R - m, lb[rt << 1 | 1]);
    return max(max(t1, t2), a + b);
}
int main() {
    int n, m, i, j, k, a, b, op;
    while (scanf("%d", &n) != EOF) {
        for (i = 1; i <= n; i++) scanf("%d", &num[i]);
        build(1, n, 1);
        scanf("%d", &m);
        while (m--) {
            scanf("%d%d%d", &op, &a, &b);
            if (op == 1)
                update(a, b, 1, n, 1);
            else {
                int ans = query(a, b, 1, n, 1);
                printf("%d\n", ans);
```

```
            }
        }
    }
    return 0;
}
```

1.3 并查集

基础算法

```
void init(){
    for(int i=1;i<MAX;i++){
        bin[i]=i;
        crank[i]=1;
    }
}
int findr(int x){
    if(x!=bin[x]){
        bin[x]=findr(bin[x]);   //路径压缩
    }
    return bin[x];
}
void mergebin(int x,int y){
    int rx,ry;
    rx=findr(x);
    ry=findr(y);
    if(rx!=ry){
        bin[rx]=ry;
        crank[ry]+=crank[rx];
        ans=max(ans,crank[ry]);
    }
}
```

1.3.1 带权并查集

1.4 RMQ

已知数组 a[]，求这个数组 l、h 范围内（从 0 计数）的最小值

```
#include <bits/stdc++.h>
#define Max(a, b) a > b ? a: b
#define Min(a, b) a < b ? a: b
```

```
#define MAX 100050
using namespace std;

//若要求区间最大值，那么将Min改为Max即可
int a[MAX];
int dp[MAX][20];    //dp[i][j]表示a[]从下标i开始到i+2^j-1的范围中的最小值
void RMQ(int n) {    //Sparse Table算法,O(nlog(n))
    memset(dp, 0, sizeof(dp));
    for (int i = 0; i < n; i++)
        dp[i][0] = a[i];       //初始化

    int tmp = int(log(double(n)) / log(2.0));
    for (int j = 1; j <= tmp; j++) {
        for (int i = 0; i < n; i++) {
            if (i + (1 << (j - 1)) < n)
                dp[i][j] = Min(dp[i][j - 1], dp[i + (1 << (j - 1))][j - 1]);
        }
    }
}

int Minimum(int l, int h) {      //O(1)的查询
    int k = int(log(double(h - l + 1)) / log(2.0));
    return Min(dp[l][k], dp[h - (1 << k) + 1][k]);
}


int main() {
    int n;
    while (cin >> n) {
        for (int i = 0; i < n; i++)  cin >> a[i];
        RMQ(n);
        int x, y;
        while (cin >> x >> y)
            cout << Minimum(x, y) << endl;
    }
}
```

1.5 排序

1.5.1 计数排序

```
#include <iostream>
#include <string.h>
```

```
using namespace std;

void CountSort(int *a, int *b, int *rank, int n, int k) {
    int *c = new int[k + 1];
    memset(c, 0, sizeof(int)*(k + 1));
    for (int i = 1; i <= n; i++)
        c[a[i]]++;
    for (int i = 2; i <= k; i++)
        c[i] += c[i - 1];

    //important
    for (int i = n; i >= 1; i--) {          //如果从1-n遍历，排序的结果中，a数组值相同的元素从左到右为逆序排列，不符合要求。
        b[c[a[i]]] = a[i];    //c[a[i]]就是a[i]的排名，这行相当于将a[i]放到了根据他的排名应该放到的地方。
        rank[c[a[i]]] = i;    //rank[j]表示排名为j的数在a数组的位置
        c[a[i]]--;
    }
}

int main() {
    int n, k;                               //n为a数组元素个数，k为a数组元素最大值
    while (cin >> n >> k) {
        int *a = new int[n + 1];
        int *b = new int[n + 1];      //存放排序之后的数
        int *rank = new int[n + 1];       //存放排名为i的数
        for (int i = 1; i <= n; i++)
            cin >> a[i];
        CountSort(a, b, rank, n, k);      //计数排序
        cout << "sorted:" << endl;
        for (int i = 1; i <= n; i++)
            i == n ? cout << b[i] << endl : cout << b[i] << " ";
        cout << "rank:" << endl;
        for (int i = 1; i <= n; i++)
            i == n ? cout << rank[i] << endl : cout << rank[i] << " ";
    }
    return 0;
}
```

1.5.2 桶排序

1.5.3 基数排序

1.5.4 快速排序

1.5.5 归并排序

归并排序是稳定的排序。时间复杂度为：O(nlogn)。归并排序的思想除了用于排序，还只需要修改一点便能够实现逆序数个数的求解。
归并排序

归并排序（逆序数）

1.5.6 堆排序

1.6 树状数组

1.7 哈希表

1.8 霍夫曼树

1.9 平衡二叉树

1.9.1 Treap

1.9.2 Splay

1.9.3 树链剖分

2 图论

2.1 拓扑排序

2.2 最小生成树

2.2.1 普利姆算法

```
void prim(int v, int n) {
    int i, j, k, tmin;
    int ans = 0;
    for (i = 0; i<n; i++) {
        dis[i] = mpt[v][i];
        vis[i] = true;
    }
    vis[v] = false;
    for (j = 1; j<n; j++) {
        tmin = INF;
        k = 0;
        for (i = 0; i<n; i++) {
            if (vis[i] && dis[i] < tmin) {
                tmin = dis[i];
                k = i;
            }
        }
        if (tmin == INF)
            break;
        vis[k] = false;
        ans += tmin;
        for (i = 0; i<n; i++) {
            if (vis[i] && dis[i] > mpt[k][i]) {
                dis[i] = mpt[k][i];
            }
        }
    }
    cout << ans << endl;
}
```

2.2.2 克鲁斯卡尔算法

2.3 最短路

要注意

1、输入坑点，比如双向路径、相同路多次输入等。这种坑有的是读题不仔细，有的是题上既没有说而且不能用常识判断。所以要注意输入控制。

2、初始化

2.3.1 Dijkstra

```
int mpt[MAX][MAX];
int vis[MAX], dis[MAX];
int n;
int dijkstra(int s, int d) {
    init();    //初始化mpt二维数组、vis数组和dis数组
    dis[s] = 0;
    dis[0] = 0;     //这些初值要设置好
    for (int j = 0; j < n; j++) {
        int t = 0;
        for (int i = 1; i <= n; i++) {    //从1开始计数的dis数组
            if (vis[i])  continue;
            t = dis[t] > dis[i] ? i : t;
        }
        vis[t] = 1;
        for (int i = 0; i < n; i++) {
            if (vis[i] || mpt[t][i] == INF)  continue;
            dis[i] = Min(dis[i], dis[t] + mpt[t][i]);
        }
    }
    return dis[d];
}
```

2.3.2 Floyd:

```
for (k = 0; k<n; k++)
    for (i = 0; i<n; i++)
        for (j = 0; j<n; j++)
            dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
```

2.3.3 SPFA

```
int mpt[MAX][MAX];
int vis[MAX], dis[MAX];
int n;
int spfa(int s, int d) {
    init();
    queue<int>q;
    q.push(s);
    vis[s] = 1;    // 这里的vis数组是用来判断当前下标是否在队列中。
    dis[s] = 0;
    while (!q.empty()) {
        int e = q.front();
        q.pop();
        vis[e] = 0;
        for (int i = 0; i < n; i++) {
            if (mpt[e][i] + dis[e] >= dis[i])    continue;
            dis[i] = mpt[e][i] + dis[e];
            if (vis[i])  continue;
            q.push(i);
            vis[i] = 1;
        }
    }
    return dis[d];
}
```

2.3.4 优先队列对 dijkstra 算法的优化

```
/*
priority_queue实现的dijkstra最短路算法，时间复杂度O(mlogn)
相比原始dijkstra的O(n^2)复杂度，该算法不仅在稀疏图的情况下大大提高效率，且在稠密图下也常常不会慢于原始算法。
*/
#include <iostream>
#include <queue>
#include <string.h>
using namespace std;
#define Max a > b ? a: b
#define Min a < b ? a: b
#define MAX 1005
#define INF 0x3f3f3f3f

struct Edge {
    int from, to, dist; //起点，终点，边权
```

```
    Edge(int f, int t, int d) :from(f), to(t), dist(d) {};
};

int n, m;       //n是节点数，m是边数
vector<Edge>edges;  //边数组
vector<int>G[MAX];   //邻接图，存储边的序号，不过注意第二维下标不再表示边终点!
int dis[MAX];        //图中每个点距离单源起点s的最短距离

//添加边到edges中，并且用G保存边的编号
void AddEdge(int s, int d, int w) {
    edges.push_back(Edge(s, d, w));
    int no = edges.size() - 1;
    G[s].push_back(no);
}

void init() {
    for (int i = 0; i < MAX; i++) {
        dis[i] = INF;
        G[i].clear();
    }
    edges.clear();
}

struct Node {
    int d, u;
    bool operator < (const Node& tp) const {     //自定义优先级策略，在priority_queue会使用
        return d > tp.d;        //d即dis[u]
    }
};

//core code
int dijkstra(int s, int d) {
    int vis[MAX];
    priority_queue<Node>Q;
    //预处理部分
    memset(vis, 0, sizeof(vis));
    dis[s] = 0;
    Q.push((Node) { 0, s });
    while (!Q.empty()) {
        Node tp = Q.top();
        Q.pop();
        int u = tp.u;       //简化
        if (vis[u])  continue;
        vis[u] = 1;
        //松弛操作,和原版不一样了
        for (int i = 0; i < G[u].size(); i++) {
            Edge& e = edges[G[u][i]];   //简化
```

```
            if (vis[e.to] == 0 && dis[e.to] > dis[u] + e.dist) {
                dis[e.to] = dis[u] + e.dist;
                Q.push((Node) { dis[e.to], e.to });
            }
        }
    }
    return dis[d];
}

int main() {
    ios::sync_with_stdio(0);
    while (cin >> m >> n) {
        init();
        for (int i = 0; i < m; i++) {
            int s, d, w;
            cin >> s >> d >> w;
            AddEdge(s, d, w);
            AddEdge(d, s, w);
        }
        cout << dijkstra(1, n) << endl;
    }
    return 0;
}
```

2.4 二分图

2.5 网络流

3 组合游戏

3.1 Nim 博弈

取^（异或），结果为 0 则先手输。
上次的代码：

```
for(int i = 1; i < n; i++)
{
    scanf("%d", &a[i]);
```

```
    key = key ^ a[i];
}
if(key == 0)
{
    printf("No\n");
    continue;
}
else
{
    printf("Yes\n");
    for(int i = 0; i < n; i++)
    {
        int temp = key^a[i];
        if(temp<a[i]) printf("%d %d\n",a[i],temp);
    }
}
```

4 动态规划

4.1 记忆化搜索

```
/*
题目：POJ - 1088 滑雪
*/
int a[105][105], dp[105][105], n, m;
int ca[4][2] = { 1,0,0,-1,-1,0,0,1 };
int DFS(int x, int y) {
    if (dp[x][y] != 0)   return dp[x][y];
    int i, j, k, tmp, ans = 1;
    for (k = 0; k<4; k++) {
        i = x + ca[k][0];
        j = y + ca[k][1];
        if (i >= n || j >= m || i<0 || j<0)   continue;
        if (a[i][j] >= a[x][y]) continue;
        tmp = DFS(i, j) + 1;
        ans = max(tmp, ans);
    }
    dp[x][y] = ans;
    return ans;
}
int main() {
    while (cin >> n >> m) {
```

```
        int i, j, ans = 0;
        memset(dp, 0, sizeof(dp));
        for (i = 0; i<n; i++)
            for (j = 0; j<m; j++)
                scanf("%d", &a[i][j]);
        for (i = 0; i<n; i++)
            for (j = 0; j<m; j++) {
                ans = max(ans, DFS(i, j));
            }
        cout << ans << endl;
    }
    return 0;
}
```

4.2 背包问题

4.2.1 01 背包

```
/*
状态转移方程：F[i,k] = max{F[i-1, k], F[i-1, k-w[i]]+v[i]}
w[i]表示i物品重量，v[i]表示i物品价值，s表示背包容量
*/
int dp[10005];
memset(dp, 0, sizeof(dp));
for (int i = 0; i < n; i++) {
    for (int j = s; j >= w[i]; j--) {
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}

/*
01背包二维数组实现（最原始的版本）
*/
int dp[MAX][MAX];
for (int i = 1; i < n; i++) {
    for (int j = 0; j <= s; j++) {
        if (w[i] <= j)
            dp[i][j] = max(dp[i - 1][j - w[i]] + v[i], dp[i - 1][j]);
        else
            dp[i][j] = dp[i - 1][j];
    }
}
```

4.2.2 完全背包

```
/*
完全背包解法思路：
类比01背包，他的策略是取或者只取一件，而完全背包的策略是取0、1、2...⌊S/wi⌋件，故
状态转移方程：F[i,k] = max{F[i-1,k-a*w[i]]+a*v[i] | 0 <= a*w[i] <= S}
时间优化方法：
完全背包转化为01背包。即将第i种物品分解为⌊S/wi⌋件重量价值相同的物品。
不过这样时间仍比较复杂，更高效的方法是把第i种物品拆成重量w[i]*2^k，价值v[i]*2^k的若干物品，
这样便使分解的物品的个数变为了log(⌊S/wi⌋)件，能这样分解的原因是：
不管选几件第i种物品，这个件数总能表示成若干个2^k件的和。

w[i]表示i物品重量，v[i]表示i物品价值，s表示背包容量
*/
int dp[10005];
memset(dp, 0, sizeof(dp));
for (int i = 0; i < n; i++) {
    for (int j = w[i]; j <= s; j++) {  //次序和01背包颠倒：为了多次选择以前选过的物品
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

4.2.3 多重背包

```
/*
多重解法思路：
类比完全背包，多重背包的策略是取0、1、2...M件，故
状态转移方程：F[i,k] = max{F[i-1,k-a*w[i]]+a*v[i] | 0 <= a <= M}
时间优化方法：
和完全背包的二进制优化法思想完全相同。
若是M件物品，就可以分解为2^0,2^1,2^2...2^(k-1),M-(2^k-1)这些个物品，k是满足M-(2^k-1)>0的最大整数
例如，13件物品，算出k为3，按照这个方法就会分解为1,2,4,6

w[i]表示i物品重量，v[i]表示i物品价值，s表示背包容量
*/
//分解(ww：物品重量，vv：物品价值，num：物品件数)
int ic = 0;    //计数器
for (int j = 0; j < n; j++) {
    int i = 1;
    int ww, vv, num;
    cin >> ww >> vv >> num;
    while (i < num) {
        w[ic] = i * ww;
```

```
        v[ic] = i * vv;
        num -= i;
        i <<= 1;
        ic++;
    }
    //此时的i(k)已经不满足M-(2^k-1)>0了，而num正是M-(2^(k-1) - 1)
    w[ic] = num * ww;
    v[ic] = num * vv;
    ic++;
}

//分解后直接使用01背包
int dp[10005];
memset(dp, 0, sizeof(dp));
for (int i = 0; i < n; i++) {
    for (int j = s; j >= w[i]; j--) {    //只能选择一次
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
}
```

4.2.4 二维费用背包

```
/*
状态转移方程：F[i, s, u] = max{F[i-1, s, u], F[i-1, s-c[i], u-d[i]] + v[i]}
思路：类型为01背包或多重背包，s、u采用逆序遍历，类型为完全背包，s、u采用顺序遍历

c[i]、v[i]为物品的二维费用，最高容量分别是s和u，v[i]是物品价值
*/

int dp[MAX][MAX];
for (int i = 0; i < n; i++) {
    for (int j = s; j >= c[i]; j--) {
        for (int k = u; k >= v[i]; k--) {
            dp[j][k] = max(dp[j][k], dp[j-c[i]][k-d[i]] + v[i]);
        }
    }
}
```

4.2.5 分组背包

```
/*
分组背包：物品被分为k个组，选的时候要么从一个组中选一个，要么就不选
```

```
状态转移方程：F[k, u]=max{F[k-1, u-w[i]]+v[i], F[k-1, u]}
F[k, u]表示容量为u的背包，在前k组选最大能有多少重量
*/

int v[MAX][MAX];
int dp[MAX];
for (int i = 0; i < n; i++) {
    for (int j = s; j >= 0; j--) {
        for (int k = 1; k <= j; k++) {
            dp[j] = max(dp[j], dp[j - k] + v[i][k]);
        }
    }
}
```

4.3 区间 DP

dp[i][j] = max{dp[i][j], dp[i][k] + dp[k+1][j]}
石子归并

括号匹配

4.4 树形 DP

4.5 数位 DP

```
/*
统计数字：在1-n范围的数字，0~9这个10位数分别共出现了多少次（无前导零）
*/
#include <iostream>
#include <string.h>
#define LL long long
using namespace std;

int a[20];
int dp[20][20];

LL dfs(int pos, int num, bool limit, bool lead, int k) {
    if (pos == -1)   return num;
    if (!limit && !lead && dp[pos][num] != -1)    return dp[pos][num];
    int ans = 0;
```

```
    int up = limit ? a[pos] : 9;
    for (int i = 0; i <= up; i++) {
        int num1 = num + (i == k);
        if (i == 0 && lead)  num1 = num;
        ans += dfs(pos - 1, num1, limit && i == up, !i && lead, k);
    }
    if (!limit && !lead) dp[pos][num] = ans;
    return ans;
}

LL solve(int x, int i) {
    int ic = 0;
    while (x) {
        a[ic++] = x % 10;
        x /= 10;
    }
    return dfs(ic - 1, 0, true, true, i);
}

int main() {
    ios::sync_with_stdio(false);
    int n;
    while (cin >> n) {
        for (int i = 0; i < 10; i++) {
            memset(dp, -1, sizeof(dp));
            cout << i << " ";
            cout << solve(n, i) - solve(1 - 1, i) << endl;
        }
    }
    return 0;
}

/*
在1到r范围的数中，含2014数的个数
*/
#include <iostream>
#include <string.h>
#define LL long long
using namespace std;

int a[20];
int dp[20][2][2][2][2];

LL dfs(int pos, int n1, int n2, int n3, int n4, bool limit, bool lead) {
    if (pos == -1)   return n1&n2&n3&n4;
    if (!limit && !lead && dp[pos][n1][n2][n3][n4] != -1)    return dp[pos][n1][n2][n3][n4];
    int ans = 0;
```

```
    int up = limit ? a[pos] : 9;
    for (int i = 0; i <= up; i++) {
        int nn1 = i == 2 ? 1 : 0;
        int nn2 = i == 0 ? 1 : 0;
        int nn3 = i == 1 ? 1 : 0;
        int nn4 = i == 4 ? 1 : 0;
        if (lead)    nn2 = 0;
        ans += dfs(pos - 1, nn1 | n1, nn2 | n2, nn3 | n3, nn4 | n4, limit && i == up, lead && !i);
    }
    if (!limit && !lead) dp[pos][n1][n2][n3][n4] = ans;
    return ans;
}


LL solve(int x) {
    int ic = 0;
    while (x) {
        a[ic++] = x % 10;
        x /= 10;
    }
    return dfs(ic - 1, 0, 0, 0, 0, true, true);
}

int main() {
    ios::sync_with_stdio(false);
    int l, r;
    while (cin >> l >> r) {
        memset(dp, -1, sizeof(dp));
        cout << solve(r) - solve(l - 1) << endl;
    }
    return 0;
}
```

4.6 状态压缩 DP

5 数论

5.1 素数筛法

```
//素数筛法优化版本
memset(prime, 0, sizeof(prime));
prime[0] = prime[1] = 1;
for (int i = 2; i < MAX; i++) {
    if (prime[i] == 1)   continue;
    if (i > MAX / i) continue;           //点睛之笔
    for (int j = i * i; j < MAX; j += i)
        prime[j] = 1;
}
```

5.2 求质因子

```
int main()
{
    LL num;
    memset(prime, 0, sizeof(prime));
    prime[0] = prime[1] = 1;
    //筛法
    for (LL i = 2; i < MAX; i++)
    {
        if (prime[i] == 1)   continue;
        if (i * i >= MAX) continue;
        for (LL j = i * i; j < MAX; j += i)
        {
            prime[j] = 1;
        }
    }
    //输入一个数num，求其质因子(no是输出格式开关变量)
    while (~scanf("%lld", &num))
    {
        int no = 0;
        for (LL j = 2; j <= num; j++)
        {
            if (prime[j] == 1)   continue;
            if (num % j == 0)
            {
                if (no == 0)
                {
                    printf("%lld", j);
                    no = 1;
```

```
                }
                else printf("*%lld", j);
                num /= j;
                j = 1;
            }
        }
        cout << endl;
    }
    return 0;
}
```

5.3 逆元求法

5.3.1 费马小定理

```
/*
逆元：a*x  ≡ 1 (mod p)，x就是a关于p的逆元
逆元的作用：(a / b) % p = (a * inv(a)) % p
前提条件：a和p互质，且gcd(a, p) = 1
*/
LL POW(LL a, LL n, LL mod) {
    LL ans = 1, tmp = a;
    while (n) {
        if (n % 2 == 1)  ans = (ans * tmp) % mod;
        tmp = (tmp * tmp) % mod;
        n >>= 1;
    }
    return ans;
}

LL INV(LL a, LL p) {
    return POW(a, p - 2, p);
}
```

5.3.2 扩展欧几里德

```
/*
前提条件：a和b互质
*/

void ex_gcd(LL a, LL b, LL &x, LL &y, LL &d) {
    if (!b)  d = a, x = 1, y = 0;
```

```
    else {
        ex_gcd(b, a % b, y, x, d);
        y -= x * (a / b);
    }
}

LL INV(LL a, LL p) { //a、p代表扩展欧几里德中的a、b。
    LL x, y, d;
    ex_gcd(a, p, x, y, d);
    return d == 1 ? (x % p + p) % p : -1; //返回-1表示无解
}
```

5.4 中国剩余定理

```
void ex_gcd(LL a, LL b, LL &x, LL &y, LL &d) {   //扩展欧几里德
    if (!b)  d = a, x = 1, y = 0;
    else {
        ex_gcd(b, a % b, y, x, d);
        y -= x * (a / b);
    }
}

//X表示被取模的数，M表示原本模的数，M=m1*m2*m3...*mn，m[]数组元素两两互素
LL CRT(LL n, LL X, LL M, LL *m) {    //中国剩余定理
    LL ans = 0, d, x, y;
    for (int i = 0; i < n; i++) {
        ex_gcd(M / m[i], m[i], x, y, d);    //扩欧求逆元x
        ans += ((X % m[i]) * (M / m[i]) * x);
    }
    if (ans < 0) ans += M;
    return ans;
}
```

5.5 Lucas 定理

```
/*
结论1. Lucas(n,m,p) = C(n, m)%p = C(n%p, m%p)*Lucas(n/p, m/p, p)
结论2. 把n写成p进制a[n]a[n-1]a[n-2]...a[0]，把m写成p进制b[n]b[n-1]b[n-2]...b[0]，则
C(n,m)与C(a[n],b[n])*C(a[n-1],b[n-1])*C(a[n-2],b[-2])*....*C(a[0],b[0])模p同余。
前提条件：p为素数，n、m为非负整数
前置工具代码：阶乘数组，快速幂取模，费马小定理求逆元
注意事项：n,m不能大于10^5，不大于情况下用逆元的方法可以解决，如果大了就不能解决。
```

```
*/
LL fac[MAX];    //阶乘数组
void init_jc(LL p) { //初始化阶乘数组
    fac[0] = 1;
    for (int i = 1; i < MAX; i++)
        fac[i] = fac[i - 1] * i % p;
}

LL POW(LL a, LL n, LL mod) { //快速幂取模
    LL ans = 1, tmp = a % mod;
    while (n) {
        if (n % 2 == 1)  ans = (ans * tmp) % mod;
        tmp = (tmp * tmp) % mod;
        n >>= 1;
    }
    return ans;
}

LL INV(LL a, LL p) { //求逆元（费马小定理）
    return POW(a, p - 2, p);
}

LL C(LL n, LL m, LL mod) {   //组合公式（利用逆元）
    if (m > n)   return 0;
    return fac[n] * INV((fac[n - m] * fac[m]) % mod, mod) % mod;
}

LL Lucas(LL n, LL m, LL p) {
    if (m == 0)
        return 1;
    return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}

int main() {
    LL n, m, mod;
    while (cin >> n >> m >> mod) {
        init_jc(mod);
        cout << Lucas(n, m, mod) << endl;   //别忘了mod需要是素数
    }
    return 0;
}
```

6 字符串

6.1 KMP

s 是正文串，t 是模式串

```
void makeNext(int m) {
    int i = 0, j = -1;
    Next[0] = -1;
    while (i < m) {
        if (j == -1 || t[i] == t[j]) {
            i++, j++;
            Next[i] = j;
        }
        else j = Next[j];
    }
}

int kmp(int n, int m) {
    int i = 0, j = -1;
    makeNext(m);
    while (i < n && j < m) {
        if (j == -1 || t[j] == s[i]) {
            i++, j++;
        }
        else    j = Next[j];
    }
    if (i >= n && j < m) return -1;
    else return i - m;
}
```

next 数组的使用(字符串 str，当有 n 个相同子串 x 整合到一起满足为 str 字符串时，求 x 串的长度)：

```
void makeNext(int n) {
    int i = 0, j = -1;
    Next[0] = -1;
    while (i < n) {
        if (j == -1 || s[i] == s[j]) {
            Next[++i] = ++j;
            if (j == 1) {
                ans = i - 1;
            }
        }
        else j = Next[j];
    }
}
```

```
int main() {
    while (cin >> s && s[0] != '.') {
        int n = strlen(s);
        ans = 1;
        makeNext(n);
        if (Next[n] >= ans && Next[n] % ans == 0)    cout << n / ans << endl;
        else cout << "1" << endl;
    }
    return 0;
}
```

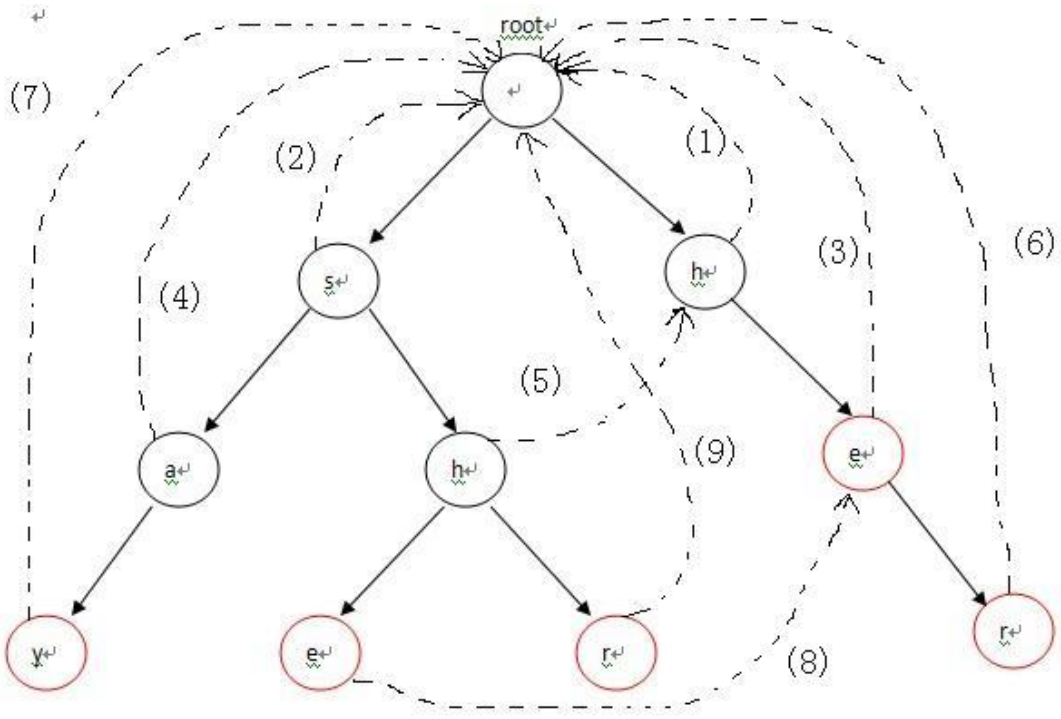
6.2 AC 自动机

判断多个字符串在一段长文本中的存在情况。
举例：给出 n 个字符串，在这 n 个中有多少个存在于 str[]中？

```
struct node {
    int cnt;
    struct node *next[CAP];
    struct node *fail;
    void init() {
        cnt = 0;
        for (int i = 0; i < CAP; i++)    next[i] = NULL;
        fail = NULL;
    }
};
void build(char str[MAX], struct node *&root) {
    struct node *tmp = root;
    int j = 0;
    int m = strlen(str);
    while (j < m) {
        int i = str[j] - 'a';
        if (tmp->next[i] == NULL) {
            tmp->next[i] = new struct node;
            tmp->next[i]->init();          //important
        }
        tmp = tmp->next[i];
        j++;
    }
    tmp->cnt++;
}
void setFail(struct node *&root) {
    root->fail = NULL;
    queue<struct node *>que;
    struct node *son, *p, *tmp;
    que.push(root);
```

```
    p = root;
    while (!que.empty()) {
        tmp = que.front();
        que.pop();
        for (int i = 0; i < CAP; i++) {
            if (tmp->next[i] == NULL)   continue;
            son = tmp->next[i];
            if (tmp == root)    son->fail = root;
            else {
                p = tmp->fail;
                while (p) {
                    if (p->next[i]) {
                        son->fail = p->next[i];
                        break;
                    }
                    else p = p->fail;
                }
                if (p == NULL)  son->fail = root;
            }
            que.push(son);
        }
    }
}
void query(char str[MAX], struct node *root) {
    struct node *p, *tmp;
    int ans = 0;
    p = root;
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        int pos = str[i] - 'a';
        while (p->next[pos] == NULL && p != root)   p = p->fail;
        p = p->next[pos];
        if (p == NULL)  p = root;
        tmp = p;
        while (tmp != NULL)
        {
            if (tmp->cnt >= 0) {
                ans += tmp->cnt;
                tmp->cnt = -1;
            }
            else break;
            tmp = tmp->fail;
        }
    }
    cout << ans << endl;
}
int main() {
```

```
    int T;
    cin >> T;
    while (T--) {
        int n;
        struct node *root = new struct node;
        root->init();
        scanf("%d", &n);
        for (int i = 0; i < n; i++) {
            char str[MAX];
            scanf("%s", str);
            build(str, root);
        }
        setFail(root);
        char str[MAX];
        scanf("%s", str);
        query(str, root);
    }
    return 0;
}
```



（失败指针指法如图，理解:当模式匹配过程中**失配时**，指针则指向**失败指针**指向的内存）

AC 自动机变形：(不仅要知道哪些字符串存在，还要分别其求出在长文本中的个数)

```
struct node {
    int no;
    int cnt;
    struct node *next[CAP];
    struct node *fail;
    void init() {
```

```
        for (int i = 0; i < CAP; i++)    next[i] = NULL;
        fail = NULL;
        no = -1;
        cnt = 0;
    }
};
char s[1000][55];
void build(char str[MAX], struct node *&root, int no) {
    struct node *tmp = root;
    int j = 0;
    int m = strlen(str);
    while (j < m) {
        int i = str[j] - 'A';
        if (tmp->next[i] == NULL) {
            tmp->next[i] = new struct node;
            tmp->next[i]->init();          //important
        }
        tmp = tmp->next[i];
        j++;
    }
    tmp->no = no;
    tmp->cnt++;
}
void setFail(struct node *&root) {
    root->fail = NULL;
    queue<struct node *>que;
    struct node *son, *p, *tmp;
    que.push(root);
    p = root;
    while (!que.empty()) {
        tmp = que.front();
        que.pop();
        for (int i = 0; i < CAP; i++) {
            if (tmp->next[i] == NULL)   continue;
            son = tmp->next[i];
            if (tmp == root)    son->fail = root;
            else {
                p = tmp->fail;
                while (p) {
                    if (p->next[i]) {
                        son->fail = p->next[i];
                        break;
                    }
                    else p = p->fail;
                }
                if (p == NULL)  son->fail = root;
            }
```

```
            que.push(son);
        }
    }
}
void query(char str[MAX], struct node *root) {
    struct node *p, *tmp;
    int vis[1001];
    int ans[1001];
    memset(vis, 0, sizeof(vis));
    memset(ans, 0, sizeof(ans));
    p = root;
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        int pos = str[i] - 'A';
        if (pos < 0 || pos > 25) { p = root; continue; }
        while (p->next[pos] == NULL && p != root)   p = p->fail;
        p = p->next[pos];
        if (p == NULL)  p = root;
        tmp = p;
        while (tmp != NULL)
        {
            vis[tmp->no] = 1;
            ans[tmp->no] += tmp->cnt;
            tmp = tmp->fail;
        }
    }
    for (int i = 0; i < 1000; i++) {
        if (vis[i]) {
            printf("%s: %d\n", s[i], ans[i]);
        }
    }
}
int main() {
    int n;
    while (~scanf("%d", &n)) {
        struct node *root = new struct node;
        root->init();
        for (int i = 0; i < n; i++) {
            scanf("%s", s[i]);
            build(s[i], root, i);
        }
        setFail(root);
        char str[MAX];
        scanf("%s", str);
        query(str, root);
    }
    return 0;
```

```
}
```

6.3 后缀数组

```
int wa[maxn], wb[maxn], wv[maxn], ws[maxn];
int cmp(int *r, int a, int b, int l)
{
    return r[a] == r[b] && r[a + l] == r[b + l];
}

void da(int *r, int *sa, int n, int m)
{
    int i, j, p, *x = wa, *y = wb, *t;
    for (i = 0; i<m; i++) ws[i] = 0;
    for (i = 0; i<n; i++) ws[x[i] = r[i]]++;
    for (i = 1; i<m; i++) ws[i] += ws[i - 1];
    for (i = n - 1; i >= 0; i--) sa[--ws[x[i]]] = i;

    for (j = 1, p = 1; p<n; j *= 2, m = p)
    {

        for (p = 0, i = n - j; i<n; i++) y[p++] = i;
        for (i = 0; i<n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;

        for (i = 0; i<n; i++) wv[i] = x[y[i]];
        for (i = 0; i<m; i++) ws[i] = 0;
        for (i = 0; i<n; i++) ws[wv[i]]++;
        for (i = 1; i<m; i++) ws[i] += ws[i - 1];
        for (i = n - 1; i >= 0; i--) sa[--ws[wv[i]]] = y[i];

        for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i<n; i++)
            x[sa[i]] = cmp(y, sa[i - 1], sa[i], j) ? p - 1 : p++;
    }
    return;
}

int rank[maxn], height[maxn];
void calheight(int *r, int *sa, int n)
{
    int i, j, k = 0;
    for (i = 1; i <= n; i++) rank[sa[i]] = i;
    for (i = 0; i<n; height[rank[i++]] = k)
        for (k ? k-- : 0, j = sa[rank[i] - 1]; r[i + k] == r[j + k]; k++);
    return;
}
```

```
da(r, sa, n + 1, 128);
calheight(r, sa, n);
```

7 C++ STL

7.1 map：

声明语法（例子），map<string, int>mpt;
然后就可以把 mpt 当成一个一维数组来使用，只不过数组的下标变成了 string 类型的值。

7.2 vector：

添加，push_back()
删除 vec 的第 i+1 个元素：vec.erase(vec.begin()+i);
删除 vec 的[i,j-1]区间的所有元素：vec.erase(vec.begin()+i,vec.begin()+j);
其余和数组差不多

7.3 queue(先进先出)：

添加，push()
删除，pop()

7.4 stack(先进后出)

添加，push()
删除，pop()

7.5 重载运算符

```
/*重载运算符让自定义变量使用sort函数*/
template <typename T>
struct NODE {
    T x, y;
    NODE(T a = 0, T b = 0) : x(a), y(b) {};       //构造函数
```

```
};

template<typename T>
//运算符<的返回值是布尔变量
bool operator < (const NODE<T>& A, const NODE<T>& B) {
    if (A.x > B.x)   return false;
    else    return true;
}

template<typename T>
ostream& operator << (ostream& out, const NODE<T>& A) {
    out << "x=" << A.x << endl;
    out << "y=" << A.y << endl;
    return out;
}

int main() {
    int n;
    while (cin >> n) {
        NODE<double> tp[MAX];
        REP(i, n)   cin >> tp[i].x >> tp[i].y;
        sort(tp, tp + n);       //若是自定义变量需要重载<运算符才能使用sort函数!
        REP(i, n)   cout << tp[i];
    }
    return 0;
}
```

7.6 模版函数

```
//模版函数的使用
template <typename T>
struct NODE {
    T x, y;
    NODE(T a, T b) : x(a), y(b) {};       //构造函数
};

/*
重载+运算符
注意每个使用到模版函数的都要修改保留字格式:后面加上<typename>
*/
template <typename T>       //每段使用模版函数的函数前都应该加上模版函数的声明
NODE<T> operator + (const NODE<T>& A, const NODE<T>& B) {
    return NODE<T>(A.x + B.x, A.y + B.y);
}

//重载<<运算符,ostream是<<运算符的数据类型
```

```
template <typename T>
ostream& operator << (ostream& out, const NODE<T>& A) {
    out << "x=" << A.x << endl;
    out << "y=" << A.y << endl;
    return out;
}

int main() {
    double _1, _2, _3, _4;      //下划线_不能作为变量的后缀
    while (cin >> _1 >> _2 >> _3 >> _4) {
        NODE<double> m(_1, _2), n(_3, _4);
        cout << m << endl;
        cout << n << endl;
        cout << m + n << endl;
    }
    return 0;
}
```

8 数学公式

8.1 求三角形面积

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

① 海伦公式描述：公式中 a，b，c 分别为三角形三边长，p 为半周长，S 为三角形的面积。

② 行列式求三角形面积：

S=（1/2)*(下面行列式）= (1/2)*(x1y2+x2y3+x3y1-x1y3-x2y1-x3y2)

|x1 y1 1|

|x2 y2 1|

|x3 y3 1|

8.2 点到直线的距离

方法一：距离公式直接求

$$d = \left| \frac{Ax_0 + By_0 + C}{\sqrt{A^2 + B^2}} \right|$$

公式描述：公式中的直线方程为 Ax+By+C=0，点 P 的坐标为(x0,y0)。但是直线方程不是够直接。推荐使用方法二。

方法二：先用海伦公式求面积然后求三角形高

8.3 卡特兰数

令 h(1)=1,h(0)=1，catalan 数满足递归式： h(n)= h(0)*h(n-1)+h(1)*h(n-2) + ... + h(n-1)h(0) (其中 n>=2)
另类递归式： $h(n) = h(n-1) * (4 * n - 2)/(n + 1)$
该递推关系的解为：h(n)=C(2n,n)/(n+1) (n=1,2,3,...)

8.4 判断一个点是否在已知三点坐标的三角形内

```
//判断一个点是否在已知三点坐标的三角形内（向量法）
struct node {
    double x, y;
};
double dot(node a, node b) {
    return a.x * b.x + a.y * b.y;
}
bool JudgePointInTriangle(node v0, node v1, node v2) {
    double fenzi = dot(v0, v0) * dot(v1, v1) - dot(v0, v1) * dot(v1, v0);
    double u = dot(v1, v1) * dot(v2, v0) - dot(v1, v0) * dot(v2, v1);
    u /= fenzi;
    if (u < 0 || u > 1)  return false;
    double v = dot(v0, v0) * dot(v2, v1) - dot(v0, v1) * dot(v2, v0);
    v /= fenzi;
    if (v < 0 || v > 1)  return false;
    return u + v <= 1;
}
```

9 杂项

9.1 快速幂取模

```
//求 a^n 的前log10 MOD位数
LL POW(LL a, LL n, LL MOD) {
    LL ans = 1, tmp = a;
    while (n != 0) {
        if (n % 2 == 1) {
            ans = (ans * tmp) % MOD;
        }
        tmp = (tmp * tmp) % MOD;
        n /= 2;
    }
    return ans;
```

```
}
```

9.2 大整数取模

```
#include<cstdio>
char a[10105];
int main(void) {
    int i, sum;
    while (scanf("%s", a) != EOF) {//a为大整数
        if (a[0] == '0') break;
        for (i = sum = 0; a[i] != '\0'; i++) {
            sum *= 10;
            sum += a[i] - '0';
            sum %= 17;    //模17
        }
        if (sum == 0) printf("1\n");    //能整除
        else printf("0\n");             //不能
    }
}
```

9.3 约瑟夫环

```
int jsf(int n) {
    if (n == 3) return 3;
    if (n == 2) return 1;
    if (n == 1) return 1;
    bool isOdd;
    if (n % 2 == 1) isOdd = true;
    else isOdd = false;
    int ans;
    if (isOdd) {
        ans = jsf((n - 1) / 2);
        return ans * 2 + 1;
    }
    else {
        ans = jsf(n / 2);
        return ans * 2 - 1;
    }
}
```

9.4 求字典序值

```
// 求字典序值 ans
for(int i = 0; i < n; i++)
{
    ans += beishu(a, i, n) * jiecheng(n- i - 1);
}
int jiecheng(int x)
{
    int ans = 1;
    for(int i = 2; i <= x; i++)
    {
        ans *= i;
    }
    return ans;
}

int beishu(int a[], int x, int n)
{
    int ans = 0;
    for(int i = x + 1; i < n; i++)
        if(a[i] < a[x])
            ans++;
    return ans;
}
```

9.5 输入输出流重定向

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```