基础算法

快速排序

快速排序

【题目】对下标范围$[1,n]$的乱序数组进行升序排序

【时间复杂度】$O(nlogn)$

【划分区间问题】为了防止复杂度退化分界值最好取中值或者随机取值。另外要注意划分子区间时陷入死循环：

- $[l,j]$和$[j+1,r]$时只能取左边界或者$l+r>>1$
- $[l,i-1]$和$[i,r]$时只能取右边界

```
const int N = 1e6 + 5;
int num[N];

void quick_sort(int num[], int l, int r) {
    if (l >= r) return;
    int i = l - 1, j = r + 1, val = num[l + r >> 1];
    while (i < j) {
        while (num[++i] < val);
        while (num[--j] > val);
        // do ++i; while (num[i] < val);
        // do --j; while (num[j] > val);
        if (i < j) swap(num[i], num[j]);
    }
    quick_sort(num, l, j), quick_sort(num, j + 1, r);
}

void solve() {
    int n;
    cin >> n;
    rep(i, 1, n) cin >> num[i];
    quick_sort(num, 1, n);
    rep(i, 1, n) cout << num[i] << ' ';
    cout << endl;
}
```

查找第 K 大

【题目】在下标范围$[1,n]$的乱序数组中找出第K大的数

【时间复杂度】$O(n)$

【思想】利用快速排序思想只将第K大数字所在的子区间进一步排序

```
const int N = 1e6 + 5;
int num[N];

int findk(int num[], int l, int r, int tempk) {
    if (l >= r) return num[l];
    int i = l - 1, j = r + 1, val = num[l + r >> 1];
    while (i < j) {
        while (num[++i] < val)
            ;
        while (num[--j] > val)
            ;
        if (i < j) swap(num[i], num[j]);
    }
    int tempcnt = j - l + 1;  //划分左区间数量
    if (tempk <= tempcnt) return findk(num, l, j, tempk);  //
判断进一步递归区间
    return findk(num, j + 1, r, tempk - tempcnt);
}

void solve() {
    int n, k;
    cin >> n >> k;
    rep(i, 1, n) cin >> num[i];
    cout << findk(num, 1, n, k) << endl;
}
```

归并排序

归并排序

【题目】对乱序下标范围$[1,n]$的数组进行升序排序

【时间复杂度】$O(nlogn)$

```
const int N = 1e6 + 5;
int num[N], temp[N];  //临时数组放在外面防止爆栈

void merge_sort(int num[], int l, int r) {
    if (l >= r) return;
```

```
    int mid = l + r >> 1;
    merge_sort(num, l, mid), merge_sort(num, mid + 1, r);
    int i = l, j = mid + 1, k = 1;
    while (i <= mid && j <= r)
        if (num[i] <= num[j])
            temp[k++] = num[i++];
        else
            temp[k++] = num[j++];
    while (i <= mid) temp[k++] = num[i++];
    while (j <= r) temp[k++] = num[j++];
    for (int i = l, j = 1; i <= r; ++i, ++j) num[i] = temp[j]
;
}

void solve() {
    int n;
    cin >> n;
    rep(i, 1, n) cin >> num[i];
    merge_sort(num, 1, n);
    rep(i, 1, n) cout << num[i] << ' ';
    cout << endl;
}
```

求逆序对

【题目】求乱序下标范围$[1,n]$的数组的逆序对数量

【时间复杂度】$O(nlogn)$

【思想】利用归并排序拆分过程中求出各子区间内部逆序对数量，再在合并排序过程中求出左右子区间共同组成逆序对数量

```
const int N = 1e6 + 5;
int num[N], temp[N];  //临时数组放在外面防止爆栈

ll reversepair(int num[], int l, int r) {  //逆序对数量爆 int
    if (l >= r) return 0;
    ll res = 0;
    int mid = l + r >> 1;
    res += reversepair(num, l, mid), res += reversepair(num,
mid + 1, r);  //子区间内部逆序对
    int i = l, j = mid + 1, k = 1;
    while (i <= mid && j <= r)
        if (num[i] <= num[j])
            temp[k++] = num[i++];
        else
            temp[k++] = num[j++], res += mid - i + 1;  //归并
中求左右子区间共同组成的逆序对
```

```
    while (i <= mid) temp[k++] = num[i++];
    while (j <= r) temp[k++] = num[j++];
    for (int i = l, j = 1; i <= r; ++i, ++j) num[i] = temp[j]
;
    return res;
}

void solve() {
    int n;
    cin >> n;
    rep(i, 1, n) cin >> num[i];
    cout << reversepair(num, 1, n) << endl;
}
```

整数二分

寻找指定值

【题目】升序下标范围$[1,n]$的数组寻找指定值并返回任意下标

【时间复杂度】$O(logn)$

```
const int N = 1e5 + 5;
int num[N];

void solve() {
    int n, x;
    cin >> n >> x;
    rep(i, 1, n) cin >> num[i];
    int l = 1, r = n;   //全闭区间
    while (l <= r) {
        int mid = l + r >> 1;
        if (num[mid] == x) {  //判断是否找到
            cout << mid << endl;
            return;
        }
        if (num[mid] < x)
            l = mid + 1;
        else
            r = mid - 1;
    }
    cout << -1 << endl;
}
```

寻找指定值左边界

【题目】升序下标范围$[1,n]$的数组寻找指定值并返回最左侧下标

【时间复杂度】$O(logn)$

【思想】等同$\boldsymbol{lower_bound}$找到第一个大于等于$\boldsymbol{x}$的位置

```
const int N = 1e5 + 5;
int num[N];

void solve() {
    int n, x;
    cin >> n >> x;
    rep(i, 1, n) cin >> num[i];
    int l = 1, r = n + 1;  //左闭右开区间
    while (l < r) {
        int mid = l + r >> 1;
        if (num[mid] < x)
            l = mid + 1;  //在 x 值区域左侧缩小左边界
        else
            r = mid;  //在 x 值区域及右侧缩小右边界
    }
    if (l == n) cout << -1 << endl; //x 比所有数都大
    cout << (num[l] == x ? l : -1) << endl;
}
```

寻找指定值右边界

【题目】升序下标范围$[1,n]$的数组寻找指定值并返回最右侧下标

【时间复杂度】$O(logn)$

【思想】等同$\boldsymbol{upper_bound}$找到第一个大于$\boldsymbol{x}$的位置

```
const int N = 1e5 + 5;
int num[N];

void solve() {
    int n, x;
    cin >> n >> x;
    rep(i, 1, n) cin >> num[i];
    int l = 1, r = n + 1;  //左闭右开区间
    while (l < r) {
        int mid = l + r >> 1;
        if (num[mid] > x)
            r = mid;  //在 x 值区域右侧缩小右边界
        else
```

```
            l = mid + 1;  //在 x 值区域及左侧缩小左边界
    }
    if (l == 1) cout << -1 << endl;   //x 比所有数都小
    cout << (num[l - 1] == x ? l : -1) << endl;  //所要找的数在前面一个位置
}
```

浮点二分

循环终止

【题目】给定$n \in (-1000,10000)$求其三次方根

【时间复杂度】$O(logn)$

【循环设置】60 次循环达到$1e-18$精度，100 次循环达到$1e-30$精度

```
void solve() {
    double x;
    cin >> x;
    double l = -1000.0, r = 1000.0;
    rep(i, 1, 60) { //设置循环次数达到精度
        double mid = (l + r) / 2;
        if (mid * mid * mid >= x)
            r = mid;
        else
            l = mid;
    }
    cout << fixed << setprecision(6) << l << endl;
}
```

精度终止

【题目】给定$n \in (-1000,10000)$求其三次方根

【时间复杂度】$O(logn)$

```
const double eps = 1e-8;  //限制精度

void solve() {
    double x;
    cin >> x;
    double l = -1000.0, r = 1000.0;
    while (r - l > eps) {  //精度不够继续缩小边界
        double mid = (l + r) / 2;
        if (mid * mid * mid >= x)  //判断是否满足缩小边界方向
```

```
            r = mid;
        else
            l = mid;
    }
    cout << fixed << setprecision(6) << l << endl;
}
```

高精度运算

整数加法

【题目】两个不带前导零的高精度非负整数相加

【时间复杂度】$O(n)$

【存储】将低位到高位依次存数组低下标到高下标（即倒着存储）

```
vector<int> add(vector<int> &A, vector<int> &B) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size() || i < B.size(); ++i) {
        if (i < A.size()) t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }
    if (t) C.push_back(1);  //判断最高位是否进一
    return C;
}

void solve() {
    string a, b;
    cin >> a >> b;
    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; --i)
        A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; --i)
        B.push_back(b[i] - '0');
    vector<int> C = add(A, B);
    for (int i = C.size() - 1; i >= 0; --i)
        cout << C[i];
    cout << endl;
}
```

整数减法

【题目】两个不带前导零的高精度非负整数相减

【时间复杂度】$O(n)$

【思想】$A \geq B$情况结果为非负整数直接模拟，反之计算$B-A$然后再输出符号

【前导零】计算结果和被减数位数保持一致可能出现前导零

```
bool compare(vector<int> &A, vector<int> &B) {
    if (A.size() != B.size()) return A.size() > B.size();
    for (int i = A.size() - 1; i >= 0; --i)
        if (A[i] != B[i])
            return A[i] > B[i];
    return true;    //完全相同
}

vector<int> sub(vector<int> &A, vector<int> &B) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); ++i) {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        t = (t < 0 ? 1 : 0);
    }
    while (C.size() > 1 && C.back() == 0)  //去前导零
        C.pop_back();
    return C;
}

void solve() {
    string a, b;
    cin >> a >> b;
    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; --i)
        A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; --i)
        B.push_back(b[i] - '0');
    if (compare(A, B)) {  //判断是否 A>=B
        vector<int> C = sub(A, B);
        for (int i = C.size() - 1; i >= 0; --i)
            cout << C[i];
    } else {
        vector<int> C = sub(B, A);
        cout << '-';
        for (int i = C.size() - 1; i >= 0; --i)
            cout << C[i];
```

```
    }
    cout << endl;
}
```

整数乘法

【题目】不带前导零的高精度非负整数与int范围非负整数相乘

【时间复杂度】$O(n)$

【前导零】如果某个乘数为零很可能结果也有前导零

```
vector<int> mul(vector<int> &A, int b) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); ++i) {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }
    while (t) {
        C.push_back(t % 10);
        t /= 10;
    }
    while (C.size() > 1 && C.back() == 0)  //去前导零
        C.pop_back();
    return C;
}

void solve() {
    string a;
    int b;
    cin >> a >> b;
    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; --i)
        A.push_back(a[i] - '0');
    vector<int> C = mul(A, b);
    for (int i = C.size() - 1; i >= 0; --i)
        cout << C[i];
    cout << endl;
}
```

整数除法

【题目】不带前导零的高精度非负整数与int范围非负整数相除

【时间复杂度】$O(n)$

【注意】去掉商的前导零，余数需要传引用记录

```
vector<int> div(vector<int> &A, int b, int &r) {
    r = 0;
    vector<int> C;
    for (int i = A.size() - 1; i >= 0; --i) {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());  //结果是正向为了统一改为逆向
    while (C.size() > 1 && C.back() == 0)
        C.pop_back();  //去前导零
    return C;
}

void solve() {
    string a;
    int b, r;
    cin >> a >> b;
    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; --i)
        A.push_back(a[i] - '0');
    vector<int> C = div(A, b, r);  //最后传引用记录余数
    for (int i = C.size() - 1; i >= 0; --i)
        cout << C[i];
    cout << endl
         << r << endl;
}
```

前缀和

一维前缀和

【题目】对下标范围$[1,n]$的数组查询q次指定区间和

【时间复杂度】生成前缀和数组$O(n)$，每次查询$O(1)$

【注意】因为边界情况可能查询$[1,x]$区间所以前缀和数组从下标 1 开始

```
const int N = 1e5 + 5;
int num[N], pre[N];

void solve() {
    int n, q;
    cin >> n >> q;
    rep(i, 1, n) cin >> num[i], pre[i] = num[i] + pre[i - 1];
    while (q--) {
```

```
        int l, r;
        cin >> l >> r;
        cout << pre[r] - pre[l - 1] << endl;
    }
}
```

二维前缀和

【题目】对下标从 1 开始n行m列的二维数组查询q次指定矩阵和

【时间复杂度】生成前缀和数组$O(n^2)$，每次查询$O(1)$

【表示方法】(i,j)表示处于第i行第j列的数

【注意】下标同样从 1 开始，另外矩阵包含元素较多注意前缀和数组爆int

```
const int N = 1e3 + 5;
int num[N][N], pre[N][N];

void solve() {
    int n, m, q;
    cin >> n >> m >> q;
    rep(i, 1, n) rep(j, 1, m) {
        cin >> num[i][j];
        pre[i][j] = pre[i - 1][j] + pre[i][j - 1] + num[i][j]
 - pre[i - 1][j - 1];
    }
    while (q--) {
        int x1, y1, x2, y2;
        cin >> x1 >> y1 >> x2 >> y2;
        cout << pre[x2][y2] - pre[x1 - 1][y2] - pre[x2][y1 -
1] + pre[x1 - 1][y1 - 1] << endl;
    }
}
```

差分

一维差分

【题目】对下标范围$[1,n]$数组进行q次区间修改操作并查询所有位置数

【时间复杂度】每次修改$O(1)$，每次查询$O(n)$

【思想】通过记录相邻差值每次修改不需要操作整个区间了，但同样查询某个下标的数就导致每次需要做一遍前缀和了

【简化】起初存数可以看成做了n次单位区间修改的操作

【意义】$diff[i]$记录的是$num[i]$相较于$num[i-1]$的增量

【注意】下标还是从 1 开始并且差分数组还需要比数组多一个末尾空间

```
const int N = 1e5 + 5;
int num[N], diff[N];

void update(int l, int r, int x) {
    diff[l] += x;
    diff[r + 1] -= x;
}

void solve() {
    int n, m;
    cin >> n >> m;
    rep(i, 1, n) {
        int x;
        cin >> x;
        update(i, i, x);
    }
    while (m--) {
        int l, r, x;
        cin >> l >> r >> x;
        update(l, r, x);
    }
    rep(i, 1, n) num[i] = num[i - 1] + diff[i];
    rep(i, 1, n) cout << num[i] << ' ';
    cout << endl;
}
```

二维差分

【题目】对下标从1开始n行m列的二维数组进行q次修改操作并查询所有位置数

【时间复杂度】每次修改$O(1)$，每次查询$O(n^2)$

【意义】$diff[i][j]$记录的是$num[i][j]+num[i-1][j-1]$和$num[i-1][j]+num[i][j-1]$的增量，即对角线两点之和与反对角线两点之和的增量

【注意】二维差分不好想可以类比一维差分进行操作

```
const int N = 1e3 + 5;
int num[N][N], diff[N][N];

void update(int x1, int y1, int x2, int y2, int x) {
    diff[x1][y1] += x;
    diff[x2 + 1][y1] -= x;
    diff[x1][y2 + 1] -= x;
    diff[x2 + 1][y2 + 1] += x;
```

```
}

void solve() {
    int n, m, q;
    cin >> n >> m >> q;
    rep(i, 1, n) rep(j, 1, m) {
        int x;
        cin >> x;
        update(i, j, i, j, x);
    }
    while (q--) {
        int x1, y1, x2, y2, x;
        cin >> x1 >> y1 >> x2 >> y2 >> x;
        update(x1, y1, x2, y2, x);
    }
    rep(i, 1, n) rep(j, 1, m)
        num[i][j] = num[i - 1][j] + num[i][j - 1] - num[i - 1
][j - 1] + diff[i][j];
    rep(i, 1, n) {
        rep(j, 1, m)
                cout << num[i][j] << ' ';
        cout << endl;
    }
}
```

双指针

【思想】双指针是种思想，一般是先写出暴力的双重循环然后利用**单调性**将内部循环优化为和最外层循环一样只扫描一遍

【注意】限制双指针的移动边界以及判断合法情况

【分类】分为在同行扫描和不同行扫描的情况

最长连续不重复子序列

【题目】在整数序列中找出最长的不包含重复数字的连续区间长度

【做法】外层循环指针枚举区间右边界不同位置，内侧循环指针枚举区间右边界对应的最远左边界，右边界扩大过程出现重复元素则缩小左边界

【时间复杂度】$O(n+m)$

```
const int N = 1e5 + 5;
int num[N], cnt[N];

void solve() {
```

```
    int n, ans = 0;
    cin >> n;
    rep(i, 1, n) cin >> num[i];
    for (int i = 1, j = 1; i <= n; ++i) {  //考虑每个右边界
        ++cnt[num[i]];
        while (cnt[num[i]] > 1) {  //出现重复了缩小左边界
            --cnt[num[j]];
            ++j;
        }
        ans = max(ans, i - j + 1);
    }
    cout << ans << endl;
}
```

数组元素的目标和

【题目】两个升序排序的数组A和B中寻找唯一解(i,j)满足等式$A[i]+B[j]==x$

【做法】外层循环指针升序顺序枚举数组A，内层循环指针降序枚举数组B，数组A元素变大过程出现元素之和大于x则让B元素变小

【时间复杂度】$O(n+m)$

```
const int N = 1e5 + 5;
int a[N], b[N];

void solve() {
    int n, m, x;
    cin >> n >> m >> x;
    rep(i, 1, n) cin >> a[i];
    rep(i, 1, m) cin >> b[i];
    for (int i = 1, j = m; i <= n; ++i) {
        while (a[i] + b[j] > x) --j;
        if (a[i] + b[j] == x) {
            cout << i << ' ' << j << endl;
            return;
        }
    }
}
```

判断子序列

【题目】给定整数序列A判断是否为整数序列B的子序列

【做法】满足贪心性质双指针分别从两序列左端开始扫描，两指针不越界情况下不断增加j如果满足$b[j]==a[i]$则增加i，最后判断是否扫过整个序列A

【时间复杂度】$O(n+m)$

```
const int N = 1e5 + 5;
int a[N], b[N];

void solve() {
    int n, m;
    cin >> n >> m;
    rep(i, 1, n) cin >> a[i];
    rep(i, 1, m) cin >> b[i];
    int i = 1, j = 1;
    while (i <= n && j <= m) {
        if (a[i] == b[j]) ++i;
        ++j;
    }
    cout << (i == n + 1 ? "Yes" : "No") << endl;
}
```

位运算

二进制中 1 的个数

【题目】给出n个数，求每个数的二进制表示中 1 的个数

【做法】用$lowbit$求出数的二进制中最低 1 所对应的值，将数减去该值循环直至数为 0

【时间复杂度】每次处理为$O(M)$，M为数字中 1 的个数

```
int lowbit(int x) {
    return x & -x;
}

void solve() {
    int n;
    cin >> n;
    while (n--) {
        int x;
        cin >> x;
        int cnt = 0;
        while (x)
            x -= lowbit(x), ++cnt;
        cout << cnt << ' ';
    }
}
```

离散化

【题目】将给定的数组离散化为从 0 开始的自然数

【时间复杂度】$O(nlogn)$

【思想】本质是做映射，分三步走：排序、去重、索引

```
void solve() {
    int n;
    cin >> n;
    vector<int> num;
    rep(i, 1, n) {
        int x;
        cin >> x;
        num.push_back(x);
    }
    vector<int> temp(num);
    sort(temp.begin(), temp.end());
    temp.erase(unique(temp.begin(), temp.end()), temp.end());
    for (auto e : num)
        cout << lower_bound(temp.begin(), temp.end(), e) - te
mp.begin() << ' ';
}
```

区间合并

【题目】给定n个区间，求区间合并（区间相交或者对接即合并）后的个数

【时间复杂度】$O(n)$

【思想】按左端点排序然后扫一遍的过程中进行区间合并

```
const int inf = 0x3f3f3f3f;

void merge(vector<pii> &segs) {
    vector<pii> res;
    sort(segs.begin(), segs.end());
    int st = -inf, ed = -inf;  //设置初始值
    for (auto seg : segs) {
        if (ed < seg.first) {  //找到一段区间
            if (st != -inf)
                res.push_back(pii(st, ed));
            st = seg.first, ed = seg.second;
        } else
            ed = max(ed, seg.second);  //区间合并
    }
    if (st != -inf) res.push_back(pii(st, ed));
```

```
    segs = res;
}

void solve() {
    int n;
    cin >> n;
    vector<pii> segs;
    rep(i, 1, n) {
        int l, r;
        cin >> l >> r;
        segs.push_back(pii(l, r));
    }
    merge(segs);
    cout << segs.size() << endl;
}
```

数据结构

中缀表达式四则运算

【题目】求由非负整数、括号、四则运算组成的中缀表达式计算结果

```
struct Expression {
    string str;
    stack<int> num;
    stack<char> op;
    unordered_map<char, int> pr{{'+', 1}, {'-
', 1}, {'*', 2}, {'/', 2}};  //设定运算符优先级

    void eval() {   //处理两个数
        int b = num.top(); num.pop();
        int a = num.top(); num.pop();
        char c = op.top(); op.pop();
        int x;
        if (c == '+') x = a + b;
        else if (c == '-') x = a - b;
        else if (c == '*') x = a * b;
        else x = a / b;
        num.push(x);
    }

    int cal() {    //表达式计算
        for (int i = 0; i < str.size(); ++i) {
            char ch = str[i];
            if (isdigit(ch)) {  //读完一个数字
                int x = 0, j = i;
```

```
                while (j < str.size() && isdigit(str[j]))
                    x = x * 10 + str[j++] - '0';
                i = j - 1;
                num.push(x);
            } else if (ch == '(')
                op.push(ch);
            else if (ch == ')') {
                while (op.top() != '(') eval();  //处理找到括
号内的表达式
                op.pop();
            } else {  //低于前面运算符则先处理前面表达式
                while (op.size() && op.top() != '(' && pr[op.
top()] >= pr[ch]) eval();
                op.push(ch);
            }
        }
        while (op.size()) eval();  //处理剩余同级运算符表达式
        return num.top();
    }
} infix;

void solve() {
    cin >> infix.str;
    cout << infix.cal() << endl;
}
```

模拟栈

【题目】模拟栈的压入、弹出、判空、查询顶部、查询大小基本操作

【注意】元素从下标 1 开始存储这样让元素个数和栈顶指针对应

```
struct Stack {
#define N 100005  //静态栈空间
    int s[N], tt = 0;

    void push(int x) { s[++tt] = x; }
    void pop() { --tt; }
    bool empty() { return tt == 0; }
    int top() { return s[tt]; }
    int size() { return tt + 1; }
} stk;

void solve() {
    int m;
    cin >> m;
    while (m--) {
        string op;
```

```
        cin >> op;
        if (op == "push") {
            int x;
            cin >> x;
            stk.push(x);
        } else if (op == "pop")
            stk.pop();
        else if (op == "empty")
            cout << (stk.empty() ? "YES" : "NO") << endl;
        else if (op == "query")
            cout << stk.top() << endl;
        else
            cout << stk.size() << endl;
    }
}
```

模拟队列

【题目】模拟队列的压入、弹出、判空、查询首部、查询大小基本操作

【注意】hh和tt分别控制队列的左右两个端点，元素从下标 0 开始存储

```
struct Queue {
#define N 100005  //静态队列空间
    int q[N], hh = 0, tt = -1;

    void push(int x) { q[++tt] = x; }
    void pop() { ++hh; }
    bool empty() { return tt < hh; }
    int front() { return q[hh]; }
    int size() { return tt - hh + 1; }
} que;

void solve() {
    int m;
    cin >> m;
    while (m--) {
        string op;
        cin >> op;
        if (op == "push") {
            int x;
            cin >> x;
            que.push(x);
        } else if (op == "pop")
            que.pop();
        else if (op == "empty")
            cout << (que.empty() ? "YES" : "NO") << endl;
        else if (op == "query")
```

```
            cout << que.front() << endl;
        else
            cout << que.size() << endl;
    }
}
```

单调栈

【题目】给定整数数列，求每个数右边第一个比它小的数，不存在输出-1

【时间复杂度】$O(n)$

【思想】栈保持单调每次用当前元素依次更新栈内潜力值小的数并用当前数去更新这些从栈中弹出的未求解数

【注意】栈数组存储的是下标，答案数组存储的是数值

【变式】单调栈还可以找每个数左边的答案只需要逆序枚举即可

```
struct Monostack {
#define N 100005
    int n;
    int num[N], s[N], ans[N], tt = 0;

    bool check(int x, int y) { return x < y; }  //满足小于
    void rfind() {                              //寻找右侧最值
        rep(i, 1, n) {
            while (tt && check(num[i], num[s[tt]]))
                ans[s[tt--]] = num[i];
            s[++tt] = i;
        }
    }
} mstk;

void solve() {
    cin >> mstk.n;
    rep(i, 1, mstk.n) cin >> mstk.num[i];
    memset(mstk.ans, -1, sizeof mstk.ans);
    mstk.rfind();
    rep(i, 1, mstk.n) cout << mstk.ans[i] << ' ';
}
```

单调队列

【题目】给定整数数列，找出滑动窗口处于不同位置时内部的数最小值

【时间复杂度】$O(n)$

【思想】队列保持单调每次用当前数去更新队尾潜力值小的数，队首及时去除不在窗口内的数，最后再从队首取出的数就是当前窗口的答案

【注意】队列数组存储的是下标，答案数组存储的是窗口在各位置答案

【变式】窗口如果必须为指定大小（不可小于）则需要将注释的代码加上

```
struct Monoqueue {
#define N 1000005
    int n, sz;
    int num[N], q[N], hh = 0, tt = -1;
    vector<int> ans;  //记录答案

    bool check(int x, int y) { return x < y; }  //满足小于
    void find() {                                //寻找窗口最值
        rep(i, 1, n) {
            while (hh <= tt && check(num[i], num[q[tt]]))
                --tt;
            q[++tt] = i;   //放入当前元素
            if (hh <= tt && q[hh] <= i - sz)
                ++hh;  //去除不在窗口内元素
            // if (i < sz) continue;     如果窗口必须为 sz 需要加上这句
            ans.push_back(num[q[hh]]);  //窗口首部元素作为答案
        }
    }
} mque;

void solve() {
    cin >> mque.n;
    rep(i, 1, mque.n) cin >> mque.num[i];
    mque.find();
    for (auto e : mque.ans) cout << e << ' ';
}
```

KMP

【题目】给定长度为n的模板串和长度为m的模式串，求模板串在模式串中匹配的初始下标位置

【时间复杂度】$O(n+m)$

【思想】求模板串的nex数组，然后利用nex数组进行kmp匹配减少无用匹配

【注意】下标从 1 开始，nex的意义为模板串最长前后缀长度

【变式】如果不允许重叠匹配则每次匹配成功应该让$j=0$重新开始

```
struct Match {
#define N 1000005
    int n, m;           //两个串的长度
    int nex[N];         //下标从 1 开始对应最长前后缀长度
    string patt, txt;  //模板串和模式串
    vector<int> ans;

    void getnex() {  //求模板串 nex 数组
        for (int i = 2, j = 0; i <= n; ++i) {
            while (j && patt[j + 1] != patt[i])
                j = nex[j];
            if (patt[j + 1] == patt[i]) ++j;
            nex[i] = j;
        }
    }

    void kmp() { //在模式串上匹配模板串
        getnex();
        for (int i = 1, j = 0; i <= m; ++i) {
            while (j && patt[j + 1] != txt[i])
                j = nex[j];
            if (patt[j + 1] == txt[i]) ++j;
            if (j == n) {
                ans.push_back(i - j + 1);
                j = nex[j];  //重叠匹配
                // j = 0; 不重叠匹配
            }
        }
    }
} sm;

void solve() {
    cin >> sm.n >> sm.patt >> sm.m >> sm.txt;
    sm.patt = " " + sm.patt, sm.txt = " " + sm.txt;
    sm.kmp();
    for (auto e : sm.ans)
        cout << e << ' ';
}
```

最短循环节

【最短循环节】对该字符串处理出nex数组，如果$n\%(n-nex[n])==0$则最短循环节就是记$len=n-nex[n]$的前缀，循环次数为$\frac{n}{n-nex[n]}$。不满足则需要添加$len-n\%len$。这里需要注意一点：

- 如果$nex[n] == 0$，说明前后缀没有重叠部分，也满足上述规律相当于自身就是最短循环节且循环 1 次
- 如果$nex[n]! = 0$，说明前后缀有重叠部分，最短循环节一定是该串的某个子串

【注意】有时题目认定循环两次以上才算是真正的循环节，那么我们就需要把第一种情况排除

【题目】给定字符串，求最少需要添加多少字符使其循环至少两次及以上

```
const int N = 1e5 + 5;
string s;
int nex[N];

void getnex() {
    for (int i = 2, j = 0; i <= s.size(); ++i) {
        while (j && s[j + 1] != s[i])
            j = nex[j];
        if (s[j + 1] == s[i]) ++j;
        nex[i] = j;
    }
}

void solve() {
    cin >> s;
    int n = s.size();
    s = " " + s;
    getnex();
    int len = n - (nex[n]);
    if (n % len == 0 && nex[n])  //自身为循环节不算
        cout << 0 << endl;
    else
        cout << len - n % len << endl;
}
```

Trie

【题目】两种操作：插入字符串或者查询字符串数量

【时间复杂度】每次插入和查询都是$O(n)$

【思想】边代表字符信息，结点代表编号。一般存储的字符串字符范围不大。

【注意】编号为 0 的既为空结点也是根结点

```
struct Trie {
```

```
#define N 100005
    int son[N][26], cnt[N], idx;

    int getval(char ch) {
        return ch - 'a';
    }

    void insert(string str) {
        int p = 0;
        for (int i = 0; i < str.size(); ++i) {
            int u = getval(str[i]);
            if (!son[p][u]) son[p][u] = ++idx;
            p = son[p][u];
        }
        ++cnt[p];  //当前结尾单词数量+1
    }

    int query(string str) {
        int p = 0;
        for (int i = 0; i < str.size(); ++i) {
            int u = getval(str[i]);
            if (!son[p][u]) return 0;
            p = son[p][u];
        }
        return cnt[p];  //为 0 说明是其他单词前缀
    }
} tree;

void solve() {
    int n;
    cin >> n;
    while (n--) {
        char opt;
        cin >> opt;
        if (opt == 'I') {
            string s;
            cin >> s;
            tree.insert(s);
        } else {
            string s;
            cin >> s;
            cout << tree.query(s) << endl;
        }
    }
}
```

最大异或对

【题目】给定n个数两两进行异或运算，得到的结果最大为多少

【做法】先都插入字典树，然后对于每个数处理找异或对最多且尽可能在高位的另一个数

```
struct Trie {
#define N 3000005   //用到的结点
    int son[N][2], cnt[N], idx;

    void insert(ll num) {
        int p = 0;
        dep(i, 30, 0) {  //从高向低位处理
            int u = (num >> i) & 1;
            if (!son[p][u]) son[p][u] = ++idx;
            p = son[p][u];
        }
    }

    ll query(ll num) {
        int p = 0;
        ll res = 0;  //记录异或最大答案
        dep(i, 30, 0) {
            int u = (num >> i) & 1;
            if (son[p][!u]) {  //该位可以异或得 1
                p = son[p][!u];
                res = res * 2 + 1;
            } else {  //否则只能异或得 0
                p = son[p][u];
                res = res * 2;
            }
        }
        return res;
    }
} tree;

ll num[100005];

void solve() {
    int n;
    cin >> n;
    rep(i, 1, n) {
        cin >> num[i];
        tree.insert(num[i]);
    }
    ll ans = 0;
```

```
    rep(i, 1, n)  //对于每个数尝试找异或对最多的另一个数
        ans = max(ans, tree.query(num[i]));
    cout << ans << endl;
}
```

并查集

裸并查集

【题目】两种操作：合并两个数的所在集合以及询问是否两个数属于同一集合

【时间复杂度】路径压缩优化下两种操作近乎$O(1)$

【注意】操作前需要初始化

【按秩合并】可以再开一个记录集合大小的数组每次合并让小的合并到大的上面，不过这样优化也没什么用

```
struct Collection {
#define N 100005
    int n;
    int fa[N];

    void init() { rep(i, 1, n) fa[i] = i; }
    int find(int x) { return x == fa[x] ? x : fa[x] = find(fa
[x]); }
    void merge(int x, int y) { fa[find(x)] = find(y); }
} col;

void solve() {
    int m;
    cin >> col.n >> m;
    col.init();
    while (m--) {
        char opt;
        int a, b;
        cin >> opt >> a >> b;
        if (opt == 'M')
            col.merge(a, b);
        else
            cout << (col.find(a) == col.find(b) ? "Yes" : "No
") << endl;
    }
}
```

维护集合大小并查集

【题目】在裸并查集基础上新添查询元素所在元素大小的操作

【思路】修改fa数组意义：集合大小由根节点用负数记录，父结点用正数记录

【变式】也可以再开一个数组记录，但是这样空间占用更多

【注意】路径压缩与合并操作注意变化

```
struct Collection {
#define N 100005
    int n;
    int fa[N];  //负数为根结点记录大小，正数记录父结点

    void init() { memset(fa, -1, sizeof fa); }
    int find(int x) { return fa[x] < 0 ? x : fa[x] = find(fa[
x]); }
    void merge(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;  //同一集合必须返回
        if (fa[fx] < fa[fy])
            fa[fx] += fa[fy], fa[fy] = fx;
        else
            fa[fy] += fa[fx], fa[fx] = fy;
    }
} col;

void solve() {
    int m;
    cin >> col.n >> m;
    col.init();
    while (m--) {
        string opt;
        int a, b;
        cin >> opt;
        if (opt == "C") {
            cin >> a >> b;
            col.merge(a, b);
        } else if (opt == "Q1") {
            cin >> a >> b;
            cout << (col.find(a) == col.find(b) ? "Yes" : "No
") << endl;
        } else {
            cin >> a;
            cout << -col.fa[col.find(a)] << endl;  //根结点的
权值取反
        }
    }
```

```
}
```

带权并查集

【思想】在题目限制条件下另开其他数组维护信息的并查集统称带权并查集

【题目】（食物链）存在三类动物食物链构成闭环，A吃B，B吃C，C吃A，现在有n个动物判断给出的k句话（$t=1$为a和b同类，$t=2$为a吃b）中多少与前面矛盾

【做法】开一个距离数组意义为在模 3 下和根节点的关系：0 则当前动物与根结点动物同类，1 则当前动物吃根结点动物，2 则当前动物被根结点动物吃

【注意】只需要开一个权值数组维护在模 3 下的关系，具体值不重要。合并不同集合时最好画个图理清关系，权值数组具体情况具体分析。路径压缩时注意更新的先后顺序

```
const int N = 1e5 + 5;
int n, m;
int fa[N], d[N];

int find(int x) {
    if (x == fa[x]) return fa[x];
    int u = find(fa[x]);
    d[x] += d[fa[x]];
    fa[x] = u;
    return fa[x];
}

void solve() {
    cin >> n >> m;
    rep(i, 1, n) fa[i] = i;
    int ans = 0;
    while (m--) {
        int t, x, y;
        cin >> t >> x >> y;
        if (x > n || y > n)  //超过动物编号
            ++ans;
        else {
            int fx = find(x), fy = find(y);
            if (t == 1) {
                if (fx == fy && (d[x] - d[y]) % 3)  //同集合但是距离模 3 不等矛盾
                    ++ans;
                else if (fx != fy) {  //不同集合合并时让(d[fx]+d[x]-d[y])%3==0
```

```
                    fa[fx] = fy;
                    d[fx] = d[y] - d[x];
                }
            } else {
                if (fx == fy && (d[x] - d[y] - 1) % 3)  //同集合但是距离模 3 相差不为 1
                    ++ans;
                else if (fx != fy) {
                    fa[fx] = fy;  //不同集合合并让(d[fx]+d[x]-d[y]-1)%3==0
                    d[fx] = d[y] + 1 - d[x];
                }
            }
        }
    }
    cout << ans << endl;
}
```

堆

堆排序

【题目】输出长度为n的数列中前m小的数

【时间复杂度】建堆为$O(n)$，每次插入删除操作为$O(logn)$

【定义】只能确定父结点和子结点之间大小关系，非直系结点之间无法确定

【注意】数组实现堆根结点从 1 开始，操作的都是数组下标

```
struct Heap {  //小顶堆
#define N 100005
    int sz;
    int h[N];

    void up(int u) {  //上浮
        if (u / 2 >= 1 && h[u / 2] > h[u])
            swap(u, u / 2), up(u / 2);
    }

    void down(int u) {  //下沉
        int t = u;
        if (u * 2 <= sz && h[u * 2] < h[t]) t = u * 2;
        if (u * 2 + 1 <= sz && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
        if (t != u) swap(h[u], h[t]), down(t);
    }
```

```
    void delt(int u) {  //删除指定下标元素
        swap(h[u], h[sz]);
        --sz;
        up(u), down(u);  //交换过来的元素比之前大还是小所以两种都尝试,但肯定只执行一个
    }
} H;

void solve() {
    int n, m;
    cin >> n >> m;
    rep(i, 1, n) cin >> H.h[i];
    H.sz = n;
    dep(i, n / 2, 1) H.down(i);
    while (m--) {  //输出前 m 小
        cout << H.h[1] << ' ';
        H.delt(1);
    }
}
```

模拟堆

【题目】模拟空的小顶堆依次完成以下若干次操作：

- 插入一个数
- 输出当前集合最小值
- 删除当前集合中最小值（保证唯一）
- 删除第k个插入的数
- 修改第k个插入的数

【时间复杂度】每次插入删除操作为$O(logn)$

【思想】在堆操作基础上另开辟两个数组形成索引和数组下标的相互映射

【注意】最后两个操作为关于元素索引的操作

```
struct Heap {
#define N 100005
    int sz, idx;              //记录堆大小和插入新元素的索引值
    int h[N], ph[N], hp[N];  //下标对应权值、索引对应下标、下标对应索引

    void heap_swap(int u, int v) {  //三个都交换
        swap(h[u], h[v]);
        swap(hp[u], hp[v]);
        swap(ph[hp[u]], ph[hp[v]]);
```

```
    }

    void down(int u) {  //递归下沉
        int t = u;
        if (u * 2 <= sz && h[u * 2] < h[t]) t = u * 2;
        if (u * 2 + 1 <= sz && h[u * 2 + 1] < h[t]) t = u * 2
 + 1;
        if (u != t) heap_swap(u, t), down(t);
    }

    void up(int u) {  //递归上浮
        if (u / 2 >= 1 && h[u / 2] > h[u])
            heap_swap(u, u / 2), up(u / 2);
    }

    void insert(int x) {  //插入新元素
        ++sz, ++idx;       //堆大小+1,设立新元素索引
        h[sz] = x, ph[idx] = sz, hp[sz] = idx;
        up(sz);  //末尾插入新元素只能上浮
    }

    void delt(int u) {
        heap_swap(u, sz);
        --sz;
        up(u), down(u);  //交换删除并下沉当前位置元素
    }

    void deltk(int k) {
        int u = ph[k];  //根据索引找到下标
        heap_swap(u, sz);
        --sz;
        up(u), down(u);  //用 ph[k]就错了因为 swap 更新了三个数组
    }

    void updatek(int k, int x) {
        int u = ph[k];
        h[u] = x;
        up(u), down(u);  //更新然后调整
    }
} H;

void solve() {
    int n;
    cin >> n;
    while (n--) {
        string opt;
        cin >> opt;
        if (opt == "I") {
```

```
            int x;
            cin >> x;
            H.insert(x);
        } else if (opt == "PM")
            cout << H.h[1] << endl;
        else if (opt == "DM")
            H.delt(1);
        else if (opt == "D") {
            int k;
            cin >> k;
            H.deltk(k);
        } else {
            int k, x;
            cin >> k >> x;
            H.updatek(k, x);
        }
    }
}
```

哈希表

模拟散列表

【题目】完成两个操作：插入数和询问数是否在集合中出现过

【时间复杂度】近似$O(1)$

【思想】解决冲突两种方法：拉链法和开放寻址法，拉链法桶数组结合单链表，开放寻址法一般开发2~3倍空间

【哈希函数】比题目个数多一些，尽可能避开 2 的幂数并且取质数

【注意】本身哈希函数映射就应该尽可能取消映射冲突，操作前需要初始化

【删除操作】少见，真出现的话一般另开一个标记数组记录而不是真的删除

```
struct ListHash {  //拉链法
#define M 1000003  //多开一些容量且为质数

    int h[M], v[M], ne[M], idx;  //结点从 0 开始分配

    void init() {
        memset(h, -1, sizeof h);
        idx = 0;
    }

    void insert(int x) {
        int k = (x % M + M) % M;  //hash 函数
```

```
        v[idx] = x;
        ne[idx] = h[k];
        h[k] = idx++;
    }

    bool find(int x) {
        int k = (x % M + M) % M;
        for (int i = h[k]; i != -1; i = ne[i])
            if (v[i] == x)
                return true;
        return false;
    }
} H;

struct OpenHash {  //开放寻址法
#define M 100003
#define N 200005        //一般为 2 倍空间
#define inf 0x3f3f3f3f  //相当于 null

    int h[N];

    void init() {
        memset(h, 0x3f, sizeof h);
    }

    int find(int x) {  //找到了返回下标
        int p = (x % M + M) % M;
        while (h[p] != inf && h[p] != x)
            p = (p + 1) % N;
        return p;
    }

    void insert(int x) {
        int p = find(x);
        h[p] = x;
    }
} H;

void solve() {
    int n;
    cin >> n;
    H.init();
    while (n--) {
        string opt;
        int x;
        cin >> opt >> x;
        if (opt == "I")
            H.insert(x);
        else
```

```
            cout << (H.h[H.find(x)] != inf ? "Yes" : "No") <<
 endl;
    }
}
```

字符串哈希

【题目】给定字符串并进行n次询问：判断$[l_1,r_1]$和$[l_2,r_2]$形成的子串是否相同

【时间复杂度】$O(1)$

【思想】将各子串看做是一个P进制的整数，然后让他们映射到$[0,2^{64}-1]$区间

【注意】字符串下标从1开始，读入后先预处理h和p数组

【取值】进制取131或者13331，模数取2^{64}，99%几率不发生冲突

```
struct StrHash {
#define N 100005  //字符串最大长度
#define P 131     //进制一般取 131 或 13331

    string str;
    ull h[N], p[N];  //hash 前缀和,幂数预处理(自动模 2^64)

    void init() {
        p[0] = 1;
        rep(i, 1, str.size()) {
            h[i] = h[i - 1] * P + str[i];
            p[i] = p[i - 1] * P;
        }
    }

    ull hash(int l, int r) {  //取出[l,r]的 hash 值
        return h[r] - h[l - 1] * p[r - l + 1];
    }
} H;

void solve() {
    int n, m;
    cin >> n >> m >> H.str;
    H.str = " " + H.str;  //让下标从 1 开始
    H.init();
    while (m--) {
        int l1, r1, l2, r2;
        cin >> l1 >> r1 >> l2 >> r2;
        cout << (H.hash(l1, r1) == H.hash(l2, r2) ? "Yes" : "
No") << endl;
    }
}
```

图与搜索

树与图搜索

树的重心

【题目】给定n个形成的树，输出删除其重心后剩余各连通块中点数的最大值

【时间复杂度】$O(V+E)$

【思想】借助树的深度优先搜索可以求出每个结点的子树大小

【树的重心】重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心

```
const int N = 1e5 + 5;
int n, ans = 0x3f3f3f3f;
vector<int> G[N];
bool vis[N];

int dfs(int now) {
    vis[now] = true;
    int sz = 0, sum = 0;
    for (auto e : G[now]) {
        if (vis[e]) continue;
        int tmp = dfs(e);   //递归子结点
        sz = max(sz, tmp);  //求子结点连通块中最大
        sum += tmp;
    }
    sz = max(sz, n - sum - 1);  //求父节点连通块
    ans = min(ans, sz);
    return sum + 1;  //返回给父结点
}

void solve() {
    cin >> n;
    rep(i, 1, n - 1) {
        int x, y;
        cin >> x >> y;
        G[x].push_back(y), G[y].push_back(x);
    }
    dfs(1);
    cout << ans << endl;
}
```

树的直径

【题目】给定n个点形成的数，输出数的直径长度

【时间复杂度】$O(V)$

【思想】第一遍DFS找到距离最远的点作为直径一端，然后从该点再次DFS一遍找到直径另一端，中途记录从起点到达每个点的记录

```
const int N = 1e5 + 5;
int n, c;
int dis[N];
vector<int> G[N];

void dfs(int now, int from) {
    for (auto e : G[now]) {
        if (e == from) continue;
        dis[e] = dis[now] + 1;
        if (dis[e] > dis[c]) c = e;
        dfs(e, now);
    }
}

void solve() {
    cin >> n;
    rep(i, 1, n - 1) {
        int x, y;
        cin >> x >> y;
        G[x].push_back(y), G[y].push_back(x);
    }
    dfs(1, 0);
    dis[c] = 0, dfs(c, 0);
    cout << dis[c] << endl;
}
```

图中点的层次

【题目】给定n个点m条边形成有向图，各边权均为1，图中可能存在重边和自环。求从1号点到n号点的最短距离

【时间复杂度】$O(V+E)$

```
const int N = 1e5 + 5;
int n, m;
int dis[N];
vector<int> G[N];
```

```
void solve() {
    cin >> n >> m;
    memset(dis, -1, sizeof dis);
    rep(i, 1, m) {
        int x, y;
        cin >> x >> y;
        G[x].push_back(y);
    }
    queue<int> que;
    dis[1] = 0, que.push(1);

    while (que.size()) {
        int now = que.front();
        que.pop();
        for (auto e : G[now])
            if (dis[e] == -
1) dis[e] = dis[now] + 1, que.push(e);
    }
    cout << dis[n] << endl;
}
```

有向图的拓扑序列

【拓扑序列】有向图中保证对于每一条边的两点起点在序列中永远在终点前面先出现，有向无环图成为拓扑图

【题目】输出给定n个点m条边形成有向图的拓扑序列，不存在输出-1

【时间复杂度】$O(V+E)$

```
const int N = 1e5 + 5;
int n, m;
int d[N];
vector<int> G[N];
queue<int> que;
vector<int> ans;

bool topsort() {
    rep(i, 1, n) if (!d[i]) que.push(i);
    while (que.size()) {
        int now = que.front();
        ans.push_back(now);
        que.pop();
        for (auto e : G[now])
            if (--d[e] == 0) que.push(e);
    }
    return ans.size() == n;
}
```

```
void solve() {
    cin >> n >> m;
    rep(i, 1, m) {
        int a, b;
        cin >> a >> b;
        G[a].push_back(b);
        ++d[b];
    }
    if (!topsort())
        cout << -1 << endl;
    else {
        for (auto e : ans)
            cout << e << ' ';
        cout << endl;
    }
}
```

Dijkstra

朴素*Dikstra*

【题目】给定n个点m条边形成有向图，边权均为正值，图中可能存在重边和自环。求从1号点到n号点的最短距离

【时间复杂度】$O(V^2)$

【思想】执行$n-1$次将每个距离集合最近的点扔进集合中并且用选中的点作为中间点去尝试更新相连的其他点的操作

【图分类】有很少条边或弧（边的条数$|E|$远小于$|V|^2$）的图称为稀疏图，反之边的条数$|E|$接近$|V|^2$，称为稠密图

【无向图】在有向图基础上加反向边即可

【存图方式】稠密图用邻接矩阵，稀疏图用邻接表。链式前向星同样适用于稀疏图，优点是是简短相对邻接表好写

【自环与重边】自环算法会自动解决，重边在存边的时候选择边权最小的

【适用范围】因为完全由点个数决定效率，所以适用于小于 1000 点的稠密图

```
const int N = 510;
int n, m;
int dis[N], graph[N][N];
bool vis[N];
```

```
void add(int u, int v, int w) {
    graph[u][v] = min(graph[u][v], w);
}

void dikstra() {
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    rep(i, 1, n) {
        int now = -1;
        rep(j, 1, n) if (!vis[j] && (now == -
1 || dis[now] > dis[j])) now = j;
        vis[now] = true;
        rep(j, 1, n) dis[j] = min(dis[j], dis[now] + graph[no
w][j]);
    }
}

void solve() {
    cin >> n >> m;
    memset(graph, 0x3f, sizeof graph);
    while (m--) {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
    }
    dikstra();
    cout << (dis[n] == 0x3f3f3f3f ? -1 : dis[n]) << endl;
}
```

堆优化$Dikstra$

【题目】给定n个点m条边形成有向图，边权均为正值，图中可能存在重边和自环。求从1号点到n号点的最短距离

【时间复杂度】$O(ElogV)$

【思想】朴素版选点每次为$O(V)$，更新其他点为$O(E)$，整体复杂度主要取决于选点和外层循环，对于稀疏图借助“堆”维护更新操作中的点如此下次选点可以直接取出来

```
const int V = 1e5 + 5, E = 2e5 + 5;
int n, m;
int dis[V];
int head[V], idx;  //链式前向星
int to[E], w[E], nex[E];

void add(int u, int v, int c) {
```

```
    to[idx] = v, w[idx] = c, nex[idx] = head[u], head[u] = id
x++;
}

void dikstra() {
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    priority_queue<pii, vector<pii>, greater<pii>> que;  //小
顶堆
    que.push({0, 1});
    while (que.size()) {
        auto now = que.top();
        que.pop();
        int d = now.first, v = now.second;
        if (dis[v] < d) continue;  //优化冗余
        for (int i = head[v]; ~i; i = nex[i])
            if (dis[to[i]] > dis[v] + w[i])
                dis[to[i]] = dis[v] + w[i], que.push({dis[to[
i]], to[i]});
    }
}

void solve() {
    cin >> n >> m;
    memset(head, -1, sizeof head);
    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;
        add(u, v, c);
    }
    dikstra();
    cout << (dis[n] == 0x3f3f3f3f ? -1 : dis[n]) << endl;
}
```

$Bellman-ford$

【题目】给定n个点m条边的有向图，图中可能存在重边和自环，边权可能为负数，求从1号点到n号点最多经过k条边的最短距离

【时间复杂度】$O(VE)$

【思想】执行n次操作：每次枚举图中所有边看能否用左端点距离+边权更新右端点距离，很简单利用的是三角不等式原理

【三角不等式】执行k次后保证经过边条数$\leq k$的点都已经是最短距离

【注意】整体性能不如$SPFA$但是上述问题只能借助$Bellman-ford$解决

【备份】为了防止在一次操作中用先前更新过的点又更新后面的点出现串联，可以先拷贝一份更新前的距离数组

【判负环】如果更新n次后还能继续缩短某点距离说明存在负环

```
const int V = 1e5 + 5, E = 2e5 + 5;
int n, m, k;
int dis[V], backup[V];
int u[E], v[E], w[E];

void bellman_ford() {
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    rep(i, 1, k) {
        memcpy(backup, dis, sizeof dis);
        rep(j, 1, m) dis[v[j]] = min(dis[v[j]], backup[u[j]]
+ w[j]);
    }
}

void solve() {
    cin >> n >> m >> k;
    rep(i, 1, m) cin >> u[i] >> v[i] >> w[i];
    bellman_ford();
    cout << (dis[n] == 0x3f3f3f3f/2 ? -1 : dis[n]) << endl;
}
```

$SPFA$

$SPFA$求最短路

【题目】给定一个n个点m条边的有向图，图中可能存在重环和自环，边权可能为负数。数据保证不存在负权回路，求出1号点到n号点的最短距离

【时间复杂度】时间复杂度最优$O(E)$，最坏$O(VE)$

【思想】在$Bellman-ford$基础上加上一个队列来优化松弛操作

【注意】全是正权的图还是用$Dikstra$，很可能良心出题人卡$SPFA$最坏

```
const int V = 1e5 + 5, E = 2e5 + 5;
int n, m;
int dis[V];
bool vis[V];
int head[V], idx;  //链式前向星
int to[E], w[E], nex[E];
```

```
void add(int u, int v, int c) {
    to[idx] = v, w[idx] = c, nex[idx] = head[u], head[u] = id
x++;
}

void spfa() {
    memset(dis, 0x3f, sizeof dis);
    queue<int> que;
    dis[1] = 0, que.push(1), vis[1] = true;
    while (que.size()) {
        auto now = que.front();
        que.pop();
        vis[now] = false;
        for (int i = head[now]; ~i; i = nex[i])
            if (dis[to[i]] > dis[now] + w[i]) {
                dis[to[i]] = dis[now] + w[i];
                if (!vis[to[i]]) vis[to[i]] = true, que.push(
to[i]);
            }
    }
}

void solve() {
    cin >> n >> m;
    memset(head, -1, sizeof head);
    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;
        add(u, v, c);
    }
    spfa();
    cout << (dis[n] > 0x3f3f3f3f/2 ? -1 : dis[n]) << endl;
}
```

Floyd

【题目】给定一个n个点m条边的有向图，图中可能存在重边和自环，边权可能为负数。数据保证不存在负权回路，询问q次指定两点之间的最短距离

【时间复杂度】$O(V^3)$

【思想】动态规划三重循环，滚动优化掉枚举中间结点的循环

【斜三角优化】无线图中邻接矩阵是对角线对称松弛操作可以只执行一侧

```
const int N = 505;
int n, m, q;
int dis[N][N];
```

```
void floyd() {
    rep(k, 1, n) rep(i, 1, n) rep(j, 1, n)
        dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
}

void solve() {
    cin >> n >> m >> q;
    rep(i, 1, n) rep(j, i, n) dis[j][i] = dis[i][j] = (i == j
 ? 0 : 0x3f3f3f3f);
    while (m--) {
        int u, v, w;
        cin >> u >> v >> w;
        if (u == v) continue;
        dis[u][v] = min(dis[u][v], w);
    }
    floyd();
    while (q--) {
        int x, y;
        cin >> x >> y;
        cout << (dis[x][y] > 0x3f3f3f3f / 2 ? -
1 : dis[x][y]) << endl;
    }
}
```

最小生成树

【题目】给定一个n个点m条边的无向图，图中可能存在重环和自环，边权可能为负数。求最小生成树的树边权重之和

$SPFA$求负环

【题目】给定一个n个点m条边的有向图，图中可能存在重边和自环，边权可能为负数。判断图中是否存在负权回路

【时间复杂度】几乎接近$O(VE)$

【思想】和$Bellman-ford$一样应用抽屉原理，在更新dis数组时另开一个cnt数组记录到该点距离经过的边数，某个点的$cnt \geq n$时则说明出现了负环。另外图可能不是连通的所以我们起点应该考虑所有点出发，我们不关心最短距离dis也不用初始化

```
const int V = 1e5 + 5, E = 2e5 + 5;
int n, m;
int dis[V], cnt[V];
```

```
bool vis[V];
int head[V], idx;  //链式前向星
int to[E], w[E], nex[E];

void add(int u, int v, int c) {
    to[idx] = v, w[idx] = c, nex[idx] = head[u], head[u] = id
x++;
}

bool spfa() {
    queue<int> que;
    rep(i, 1, n) vis[i] = true, que.push(i);  //所有点都放进去
    while (que.size()) {
        auto now = que.front();
        que.pop();
        vis[now] = false;
        for (int i = head[now]; ~i; i = nex[i])
            if (dis[to[i]] > dis[now] + w[i]) {
                dis[to[i]] = dis[now] + w[i], cnt[to[i]] = cn
t[now] + 1;
                if (cnt[to[i]] >= n) return true;
                if (!vis[to[i]]) vis[to[i]] = true, que.push(
to[i]);
            }
    }
    return false;
}

void solve() {
    cin >> n >> m;
    memset(head, -1, sizeof head);
    while (m--) {
        int u, v, c;
        cin >> u >> v >> c;
        add(u, v, c);
    }
    cout << (spfa() ? "Yes" : "No") << endl;
}
```

朴素$Prim$算法求解

【时间复杂度】$O(V^2)$

【思想】基于$Dikstra$算法实现修改了dis的定义，$Dikstra$中表示当前点距离源点最短距离，$Prim$中表示当前点距离集合最短距离

【堆优化】同样借助$Dikstra$的堆优化可以实现稀疏图复杂度进一步降低，或者

直接使用$Kruskal$算法进行边操作

```
const int N = 510;
int n, m;
int dis[N], graph[N][N];
bool vis[N];

void add(int u, int v, int w) {
    graph[v][u] = graph[u][v] = min(graph[u][v], w);
}

int prim() {
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    int res = 0;
    rep(i, 1, n) {
        int now = -1;
        rep(j, 1, n) if (!vis[j] && (now == -
1 || dis[now] > dis[j])) now = j;
        if (dis[now] == 0x3f3f3f3f) return 0x3f3f3f3f;
        res += dis[now];  //防止自环更新距离先累加
        vis[now] = true;
        rep(j, 1, n) dis[j] = min(dis[j], graph[now][j]);
    }
    return res;
}

void solve() {
    cin >> n >> m;
    memset(graph, 0x3f, sizeof graph);
    while (m--) {
        int u, v, w;
        cin >> u >> v >> w;
        add(u, v, w);
    }
    int ans = prim();
    cout << (ans == 0x3f3f3f3f ? -1 : ans) << endl;
}
```

$Kruskal$算法求解

【时间复杂度】$O(ElogE)$

【思想】将边按权值递增排序，枚举每个边将未放入集合点的放入点放入并查集

```
const int N = 1e5 + 5, E = 2e5 + 5;
int n, m;
```

```cpp
int fa[N];
struct Edge {
    int u, v, w;
    bool operator<(const Edge& e) const {
        return w < e.w;
    }
} edges[E];

int find(int x) {
    return x == fa[x] ? x : fa[x] = find(fa[x]);
}

int kruskal() {
    sort(edges + 1, edges + 1 + m);
    rep(i, 1, n) fa[i] = i;
    int res = 0, cnt = 0;
    rep(i, 1, m) {
        int fx = find(edges[i].u), fy = find(edges[i].v);
        if (fx != fy) fa[fx] = fy, ++cnt, res += edges[i].w;
        if (cnt == n - 1) break;
    }
    return cnt < n - 1 ? 0x3f3f3f3f : res;
}

void solve() {
    cin >> n >> m;
    rep(i, 1, m) {
        int u, v, w;
        cin >> u >> v >> w;
        edges[i] = {u, v, w};
    }
    int ans = kruskal();
    cout << (ans == 0x3f3f3f3f ? -1 : ans) << endl;
}
```

染色法判定二分图

【题目】给定一个n个点m条边的无向图，图中可能存在重边和自环请判断这个图是否为二分图

【时间复杂度】$O(V+E)$

【思想】搜索每个连通块尝试交错染色

【二分图】一个图是二分图当且仅当图中不含有奇数环

DFS版
`const int N = 1e5 + 5;`

```cpp
int n, m;
int color[N];
vector<int> G[N];

bool dfs(int now, int c) {
    color[now] = c;
    for (auto e : G[now])
        if (!color[e]) {
            if (!dfs(e, 3 - c))
                return false;
        } else if (color[e] == c)
            return false;
    return true;
}

void solve() {
    cin >> n >> m;
    while (m--) {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v), G[v].push_back(u);
    }
    bool flag = true;
    rep(i, 1, n) {
        if (!color[i])
            if (!dfs(i, 1)) {
                flag = false;
                break;
            }
    }
    cout << (flag ? "Yes" : "No") << endl;
}
```

BFS版

```cpp
const int N = 1e5 + 5;
int n, m;
int color[N];
vector<int> G[N];

bool bfs(int now) {
    color[now] = 1;
    queue<int> que;
    que.push(now);
    while (que.size()) {
        auto now = que.front();
        que.pop();
        for (auto e : G[now])
            if (!color[e])
```

```
                color[e] = 3 - color[now], que.push(e);
            else if (color[e] == color[now])
                return false;
    }
    return true;
}

void solve() {
    cin >> n >> m;
    while (m--) {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v), G[v].push_back(u);
    }
    bool flag = true;
    rep(i, 1, n) {
        if (!color[i])
            if (!bfs(i)) {
                flag = false;
                break;
            }
    }
    cout << (flag ? "Yes" : "No") << endl;
}
```

二分图最大匹配

【题目】给定一个二分图，左半部包含n个点，与右半部分点之间共m条边相连，求二分图最大匹配数量

匈牙利算法

【时间复杂度】远小于$O(VE)$

【思想】给左半部分点尝试匹配右半部分点，如果右半部分点没有匹配过则匹配，否则就尝试让和右半部分点已匹配的左半部分点另寻他点给当前左半部分点让位置，让不了就尝试其他点，否不行就孤立

```
const int V = 510, E = 1e3 + 5;

int n, m;
vector<int> G[V];
int match[V];
bool vis[V];
```

```
bool find(int now) {
    for (auto e : G[now]) {
        if (!vis[e]) {
            vis[e] = true;
            if (match[e] == 0 || find(match[e])) {
                match[e] = now;
                return true;
            }
        }
    }
    return false;
}

void solve() {
    cin >> n >> m;
    while (m--) {
        int u, v;
        cin >> u >> v;
        G[u].push_back(v);
    }
    int ans = 0;
    rep(i, 1, n) {
        memset(vis, 0, sizeof vis);
        if (find(i)) ++ans;
    }
    cout << ans << endl;
}
```

数学知识

质数

【质数】在大于 1 的整数中，只包含 1 和自身两个约数的数叫做质数

试除法判定质数

【题目】判定m个数是否为质数

【时间复杂度】每次判定为$O(\sqrt{n})$

```
bool isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n / i; ++i)
        if (n % i == 0) return false;
```

```
    return true;
}

void solve() {
    int m;
    cin >> m;
    rep(i, 1, m) {
        int n;
        cin >> n;
        cout << (isPrime(n) ? "Yes" : "No") << endl;
    }
}
```

分解质因数

【题目】给定m个正整数a_i，将每个数分解质因数，并按照质因数从小到大的顺序输出每个质因数的底数和指数

【时间复杂度】每次分解一般小于$O(\sqrt{n})$

【注意】n最多有一个大于$\sqrt{n}$的质因数

```
map<int, int> primes;

void divide(int n) {
    for (int i = 2; i <= n / i; ++i)
        while (n % i == 0)
            n /= i, ++primes[i];
    if (n > 1) ++primes[n];
}

void solve() {
    int m;
    cin >> m;
    rep(i, 1, m) {
        int n;
        cin >> n;
        primes.clear();
        divide(n);
        for (auto e : primes) cout << e.first << ' ' << e.sec
ond << endl;
        cout << endl;
    }
}
```

埃氏筛法筛质数

【题目】筛出1~n中所有质数

【时间复杂度】接近$O(nloglogn)$

【思想】由算术基本定理知我们每次枚举质数筛掉其后面倍数即可，如此每次遇到的数就是质数且就是下一次需要枚举的质数

【算术基本定理】任何一个合数都可以用质因数相乘唯一表示

【质数定理】$[1,n]$中大约有$\frac{n}{lnn}$

```
const int N = 1e6 + 5;
bool notprime[N];
vector<int> primes;

void getPrimes(int n) {
    rep(i, 2, n) {
        if (!notprime[i]) {
            primes.push_back(i);
            for (int j = i + i; j <= n; j += i) notprime[j] =
 true;
        }
    }
}

void solve() {
    int n;
    cin >> n;
    getPrimes(n);
    cout << primes.size() << endl;
}
```

线性筛法筛质数

【题目】筛出1~n中所有质数

【时间复杂度】$O(n)$

【思想】使合数仅被其最小的质因数筛掉。方法是枚举过程中是质数就筛掉其与$primes$中所有质数的乘积，否则筛掉其与$primes$中所有小于等于最小质因子的乘积

【注意】在$1e6$时效率与埃氏筛法差不多，在$1e7$时效率比埃氏筛法快接近1倍

```
const int N = 1e7 + 5;
```

```
bool notprime[N];
vector<int> primes;

void getPrimes(int n) {
    rep(i, 2, n) {
        if (!notprime[i])
            primes.push_back(i);
        for (int j = 0; primes[j] <= n / i; ++j) {
            notprime[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}

void solve() {
    int n;
    cin >> n;
    getPrimes(n);
    cout << primes.size() << endl;
}
```

约数

试除法求约数

【题目】给定m个正整数，对于每个整数a_i求按照从小到大的顺序输出所有约数

【时间复杂度】每次处理为$O\sqrt{n}$

```
vector<int> getDivisors(int n) {
    vector<int> res;
    for (int i = 1; i <= n / i; ++i)
        if (n % i == 0) {
            res.push_back(i);
            if (i != n / i) res.push_back(n / i);
        }
    sort(res.begin(), res.end());
    return res;
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        int x;
```

```
        cin >> x;
        auto res = getDivisors(x);
        for (auto e : res) cout << e << ' ';
        cout << endl;
    }
}
```

约数个数

【题目】给定m个正整数a_i，请你输出这些数的乘积的约数个数，答案取模

【时间复杂度】每次处理为$O\sqrt{n}$

【约数个数定理】对于一个大于1正整数n可以分解质因数$n = \prod_{i=1}^{k} p_i{}^{a_i}$，则$n$的正约数的个数就是$f(n) = \prod_{i=1}^{k}(a_i + 1)$

```
const int mod = 1e9 + 7;
unordered_map<int, int> primes;

void divide(int n) {
    for (int i = 2; i <= n / i; ++i)
        while (n % i == 0)
            n /= i, ++primes[i];
    if (n > 1) ++primes[n];
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        int n;
        cin >> n;
        divide(n);
    }
    ll ans = 1;
    for (auto e : primes) ans = ans * (e.second + 1) % mod;
    cout << ans << endl;
}
```

约数之和

【题目】给定m个正整数a_i请输出这些数的乘积的约数之和，答案取模

【时间复杂度】每次处理为$O\sqrt{n}$

【约束和定理】对于一个大于1正整数n可以分解质因数$n = \prod_{i=1}^{k} {p_i}^{a_i}$，则$n$的正约束的和就是$f(n) = \prod_{i=1}^{k}(\sum_{j=0}^{a_i} {p_i}^{j})$

【注意】运算符优先级依次为$*$、$/$、$\%$、$+$、$-$

```
const int mod = 1e9 + 7;
unordered_map<int, int> primes;

ll qpow(ll a, ll k) {
    ll res = 1;
    while (k) {
        if (k & 1) res = res * a % mod;
        k >>= 1;
        a = a * a % mod;  //注意计算过程中爆 int
    }
    return res;
}

void divide(int n) {
    for (int i = 2; i <= n / i; ++i)
        while (n % i == 0)
            n /= i, ++primes[i];
    if (n > 1) ++primes[n];
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        int n;
        cin >> n;
        divide(n);
    }
    ll ans = 1;
    for (auto e : primes) {
        ll tmp = 0;
        rep(i, 0, e.second) tmp = (tmp + qpow(e.first, i)) %
mod;
        ans = ans * tmp % mod;
    }
    cout << ans << endl;
}
```

最大公约数

【题目】给定m对正整数a_i和b_i，求出每对数的最大公约数

【时间复杂度】每次处理为 $O(logb)$

```
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        int a, b;
        cin >> a >> b;
        cout << gcd(a, b) << endl;
    }
}
```

欧拉函数

公式求欧拉函数

【题目】给定m个正整数a_i，请你求出每个数的欧拉函数

【时间复杂度】每次处理为$O\sqrt{n}$

【欧拉函数】对于整数n，在1~n中与n互质的数的个数成为欧拉函数。对于一个大于1正整数n可以分解质因数$n = \prod_{i=1}^{k} {p_i}^{a_i}$，则有引理$\phi(n) = n\prod_{i=1}^{k} 1 - \frac{1}{p_i}$，

利用容斥原理证明

```
void solve() {
    int m;
    cin >> m;
    while (m--) {
        int n;
        cin >> n;
        int res = n;
        for (int i = 2; i <= n / i; ++i) {
            if (n % i == 0) {
                res = res / i * (i - 1);
                while (n % i == 0) n /= i;
            }
        }
        if (n > 1) res = res / n * (n - 1);
        cout << res << endl;
    }
}
```

筛法求欧拉函数

【题目】给定正整数n，求$1\sim n$中每个数的欧拉函数之和

【时间复杂度】$O(n)$

【思想】质数i的欧拉函数即为$i-1$。对于$euler[primes[j]*i]$分为两种情况：

- $i\%primes[j]==0$时，$primes[j]$是i的最小质因子同时也是$primes[j]*i$的最小质因子。因此$1-\frac{1}{prims[j]}$这一项在$euler[i]$中计算过了，只需要将基数N修正$primes[j]$倍，因此最终结果为$euler[i]*primes[j]$
- $i\%primes[j]!=0$时，$primes[j]$不是i的质因子，只是$primes[j]*i$的最小质因子，因此不仅需要将基数N修正为$primes[j]$倍，还需要补上$\frac{i-1}{primes[j]}$这一项，因此最终结果$euler[i]*(primes[j]-1)$

```
const int N = 1e7 + 5;
vector<int> primes;
int euler[N];
bool notprime[N];

void getEulers(int n) {
    euler[1] = 1;
    rep(i, 2, n) {
        if (!notprime[i])
            primes.push_back(i), euler[i] = i - 1;
        for (int j = 0; primes[j] <= n / i; ++j) {
            int tmp = primes[j] * i;
            notprime[tmp] = true;
            if (i % primes[j] == 0) {
                euler[tmp] = euler[i] * primes[j];
                break;
            }
            euler[tmp] = euler[i] * (primes[j] - 1);
        }
    }
}

void solve() {
    int n;
    cin >> n;
    getEulers(n);
    ll res = 0;
    rep(i, 1, n) res += euler[i];
    cout << res << endl;
}
```

快速乘

【题目】给定m对两个数a_i和b_i，求它们相乘取模后结果，已知直接乘法爆int

【时间复杂度】$O(logk)$

【思想】将第二个数拆分成多个二进制数然后乘法分配律和a_i相乘最后求和

```
ll qmul(ll x, ll k, ll mod) {
    ll res = 0;
    while (k) {
        if (k & 1) res = (res + x) % mod;
        x = (x + x) % mod;
        k >>= 1;
    }
    return res;
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        ll a, b, p;
        cin >> a >> b >> p;
        cout << qmul(a, b, p) << endl;
    }
}
```

快速幂

普通快速幂

【题目】给定 m 对两个数a_i和b_i，求${a_i}^{b_i}$对p_i取模后结果

【时间复杂度】每次处理为$O(logk)$

【思想】将第二个数拆分成多个二进制数，然后将这些a_i的二进制数幂数相乘

```
ll qpow(ll x, ll k, ll mod) {
    ll res = 1;
    while (k) {
        if (k & 1) res = res * x % mod;
        x = x * x % mod;
        k >>= 1;
    }
    return res;
}
```

```
void solve() {
    int m;
    cin >> m;
    while (m--) {
        ll a, b, p;
        cin >> a >> b >> p;
        cout << qpow(a, b, p) << endl;
    }
}
```

矩阵快速幂

【题目】给定$n*n$的矩阵A，求A^k结果取模

【时间复杂度】$O(logk)$

【思想】重载乘法符号为矩阵乘，思想不变。递推式中常常使用矩阵快速幂来加速优化

【常用公式】$f(n)=a*f(n-1)+b*f(n-2)+c(a,b,c是常数)$

$$\begin{pmatrix} a & b & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} f_{n-1} \\ f_{n-2} \\ C \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n-1} \\ C \end{pmatrix}$$

$$f(n)=c^n-f(n-1)(c是常数)$$

$$\begin{pmatrix} -1 & c \\ 0 & c \end{pmatrix} = \begin{pmatrix} f_{n-1} \\ c^{n-1} \end{pmatrix} = \begin{pmatrix} f_n \\ c^n \end{pmatrix}$$

```
const int mod = 1e9 + 7;
ll n, k;  //设定大小和次幂
struct Matrix {
#define N 105
    ll mat[N][N];

    Matrix() { memset(mat, 0, sizeof mat); }  //不设置栈区随机

    void build() {  //构造单位矩阵
        rep(i, 1, n) mat[i][i] = 1;
    }

    Matrix operator*(const Matrix &B) const {  //重载
        Matrix res;
        rep(k, 1, n) rep(i, 1, n) rep(j, 1, n)
            res.mat[i][j] = (res.mat[i][j] + mat[i][k] * B.ma
t[k][j] % mod) % mod;
```

```
        return res;
    }
} A;

Matrix qpow(Matrix a, ll k) {
    Matrix res;
    res.build();
    while (k) {
        if (k & 1) res = res * a;
        a = a * a;
        k >>= 1;
    }
    return res;
}

void solve() {
    cin >> n >> k;
    rep(i, 1, n) rep(j, 1, n) cin >> A.mat[i][j];
    auto ans = qpow(A, k);
    rep(i, 1, n) rep(j, 1, n) cout << ans.mat[i][j] << (j ==
n ? '\n' : ' ');
}
```

逆元

费马小定理求逆元

【题目】给定m组a_i，p_i是质数，如果存在求a_i模p_i的乘法逆元，可能存在多个请返还$0\sim p-1$之间的逆元

【时间复杂度】每次处理为$O(logp)$

【乘法逆元】若a和m互质，并且对于任意的整数b满足$a|b$则存在一个整数x使得$\frac{b}{a}\equiv b*x(mod\ m)$，称$x$为$a$在模$m$情况下的逆元，注意$a$存在乘法逆元充要条件是$a$与模数$m$互质

【费马小定理】如果p是质数且a不是p的倍数则有$a^{p-1}\equiv 1(mod\ p)$

【思想】此题模数恰好为质数所以利用费马小定理得$a*a^{p-2}\equiv 1(mod\ p)$

```
ll qpow(ll x, ll k, ll mod) {
    ll res = 1;
    while (k) {
        if (k & 1) res = res * x % mod;
        x = x * x % mod;
        k >>= 1;
```

```
    }
    return res;
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        ll a, p;
        cin >> a >> p;
        if (a % p == 0)
            cout << "impossible" << endl;
        else
            cout << qpow(a, p - 2, p) << endl;
    }
}
```

扩展欧几里得求逆元

【题目】给定m组a_i，p_i，如果存在求a_i模p_i的乘法逆元， 可能存在多个请返还$0\sim p-1$之间的逆元

【时间复杂度】$O(logp)$

【思想】等价于$ax \equiv 1(mod\ p)$求特解x，利用扩展欧几里得求解线性同余方程

【注意】适用范围比费马小定理要广，只要a和p满足互质即可

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

void solve() {
    int k;
    cin >> k;
    while (k--) {
        int a, p;
        ll x, y;
        cin >> a >> p;
        int gcd = exgcd(a, p, x, y);
        if (gcd != 1)
            cout << "impossible" << endl;
        else
```

```
            cout << (x % p + p) % p << endl;
    }
}
```

扩展欧几里得

扩展欧几里得算法

【题目】对于给定m对正整数a_i, b_i，对于每对数求出一组x_i，y_i使其满足$a_i x_i + b_i y_i = gcd(x_i, y_i)$

【时间复杂度】每次处理为$O(logb)$

【裴蜀定理】对于任何正整数a、b一定存在线性关系$ax + by = gcd(a, b)$且$gcd(a, b)$是它们能凑出来的最小数，它的重要推论是a、b互质则等价于存在整数$ax + by = 1$

【思想】扩展欧几里得算法就是在欧几里得算法基础上在求解gcd递归过程中记录下系数并回代从而求解裴蜀定理的x和y：

- 当$b = 0$时，$ax + by = a$故$x = 1$，$y = 0$
- 当$b \neq 0$时，欧几里得算法有$gcd(a, b) = gcd(b, a\%b)$，又存在

$$bx' + (a\%b)y' = gcd(b, a\%b)$$

$$bx' + (a - \lfloor a/b \rfloor * b)y' = gcd(b, a\%b)$$

$$ay' + b(x' - \lfloor a/b \rfloor * y') = gcd(b, a\%b) = gcd(b, a\%b)$$

故而$x = y', y = x' - \lfloor a/b \rfloor * y'$

```
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    int gcd = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return gcd;
}

void solve() {
    int m;
    cin >> m;
    while (m--) {
        int a, b, x, y;
```

```
        cin >> a >> b;
        exgcd(a, b, x, y);
        cout << x << ' ' << y << endl;
    }
}
```

线性同余方程

【题目】给定k组数据a_i, b_i, m_i，对于每组数求出一个x_i，使其满足$a_i x_i \equiv b_i (mod\ m_i)$，输出答案可能不唯一但必须在$int$范围内

【扩展】对于方程$ax + by = c$有解当且仅当$gcd|c$。扩展欧几里得算法求出的$ax_0 + by_0 = d$的解转换一下就有$a\left(x_0 * \frac{c}{gcd}\right) + b\left(y_0 * \frac{c}{gcd}\right) = c$，故该方程特解$x' = x_0 * \frac{c}{gcd}, y' = y_0 * \frac{c}{gcd}$。

又因为通解=特解+齐次解，齐次方程$ax + by = 0$的解为$\left(\frac{kb}{gcd}, -\frac{ka}{gcd}\right), k \in Z$，所以通解为$x = x' + \frac{kb}{gcd}, y = y' - \frac{ka}{gcd}, k \in Z$

【思想】题目$ax \equiv b(mod\ m) \Leftrightarrow ax + my = b$，利用上述扩展知识有解的条件就是$gcd(a, m)|b$，然后利用扩展欧几里得算法求特解即可。

【逆元】特别的当b=1且a与m互质时所求的x即为a的乘法逆元

```
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    int gcd = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return gcd;
}

void solve() {
    int k;
    cin >> k;
    while (k--) {
        int a, b, m, x, y;
        cin >> a >> b >> m;
        int gcd = exgcd(a, m, x, y);
        if (b % gcd)
            cout << "impossible" << endl;
        else
            cout << ((ll)x * b / gcd) % m << endl;
```

```
    }
}
```

中国剩余定理

中国剩余定理

【题目】给定整数$m_1, m_2, \dots, m_n$和$a_1, a_2, \dots, a_n$，求一个最小的非负整数x满足$\forall i \in [1,n], x \equiv m_i (mod\ a_i)$，其中保证$m_1, m_2, \dots, m_n$两两互质

【时间复杂度】$O(nlogm)$

【定理】设$M = m_1 \times m_2 \times \cdots \times m_n = \prod_{i=1}^{n} m_i$是整数$m_1, m_2, \dots, m_n$乘积，并设$M_i = M/m_i, \forall i \in 1,2,\cdots,n$是除$m_i$以外的$n-1$个乘数的积。设$t_i = M_i^{-1}$为$M_i$模$m_i$下的逆元，则在方程组通解为$kM + \sum_{i=1}^{n} a_i t_i M_i, k \in Z$，在模$M$的意义下方程组只有一个解$x = (\sum_{i=1}^{n} a_i t_i M_i) mod M$

```
const int N = 2e5 + 5;
ll m[N], a[N], M[N], t[N];

ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    ll gcd = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return gcd;
}

void solve() {
    int n;
    ll x, y, ans = 0, mlcm = 1;
    cin >> n;
    rep(i, 1, n) cin >> m[i] >> a[i], mlcm *= m[i];
    rep(i, 1, n) M[i] = mlcm / m[i];
    rep(i, 1, n) exgcd(M[i], m[i], x, y), t[i] = (x % m[i] +
m[i]) % m[i], ans = (ans + M[i] * t[i] * a[i]) % mlcm;
    cout << ans << endl;
}
```

扩展中国剩余定理

【题目】给定整数$m_1, m_2, \ldots, m_n$和$a_1, a_2, \ldots, a_n$，求一个最小的非负整数x满足$\forall i \in [1,n], x \equiv m_i (mod\ a_i)$，其中$m_1, m_2, \ldots, m_n$不一定两两互质

【时间复杂度】$O(nlogm)$

【定理】假设已经求出前$n-1$个方程组成的同余方程组的一个解为x且有$M = LCM_{i=1}^{n-1} m_i$，则前$n-1$个方程的方程组通解为$x + i * M (i \in Z)$，那么对于加入的第k个方程后的方程组我们要求一个正整数t使得$x + t * M \equiv a_n (mod\ m_n)$

$x + t * M \equiv a_n (mod\ m_n)$

【证明】对于前两个式子我们考虑合并移项得$k_1 * m_1 - k_2 * m_2 = a_2 - a_1$则有

$$k_1 * m_1 + k_2 * (-m_2) = a_2 - a_1 \tag{1}$$

根据题目我们需要找到最小的k_1, k_2使得等式成立，在有解情况下利用扩展欧几里得算法求得特解后利用通解放缩即

$$k_1 = k_1 + k * \frac{m_2}{d} \tag{2}$$

将（2）式代入（1）式得$x = \left(k_1 + k * \frac{m_2}{d}\right) * m_1 + a_1$，拆分化解后得到最终解

$$x = k * lcm(m_1, m_2) + k_1 * m_1 + a_1 \tag{3}$$

另记$m_1 = lcm(m_1, m_2), a_1 = k_1 * m_1 + a_1$，当有第3个方程时继续联立求解

【注意】$\%\frac{m_2}{d}$时需要取绝对值放缩，C语言中模负数不会得到正值

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

ll adjust(ll a, ll b) {
    return (a % b + b) % b;
}
```

```
void solve() {
    bool flag = true;
    int n;
    cin >> n;
    ll m1, a1;
    cin >> m1 >> a1;
    rep(i, 1, n - 1) {
        ll m2, a2, k1, k2;
        cin >> m2 >> a2;
        if (!flag) continue;
        ll gcd = exgcd(m1, -m2, k1, k2);
        if ((a2 - a1) % gcd)
            flag = false;
        else {
            k1 = adjust(k1 * (a2 - a1) / gcd, abs(m2 / gcd));
  //放缩求 min
            a1 = k1 * m1 + a1;
            m1 = abs(m1 / gcd * m2);
        }
    }
    if (!flag)
        cout << -1 << endl;
    else
        cout << adjust(a1, m1) << endl;  //放缩求 min
}
```

高斯消元

解实数方程组

【题目】给定包含n个方程n个未知数的线性方程组。方程组中的系数均为实数求解这个方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

【初等行变换】

（1）用非零数乘上某一方程

（2）交换两个方程的位置

（3）把一个方程的倍数加到另一个方程

【思想】利用初等行变换模拟上三角求解过程：枚举每一列c，找到当前列绝对值最大的一行→初等行变换（2）将这一行提到未确定阶梯性行中最上面→初等行变换（1）将该行的第一个数变为1 →初等行变换（3）将下面所有行的当前列变为0

【注意】存的是增广矩阵，最终最右侧列等式记录答案

```
const int N = 110;
const double eps = 1e-6;
int n;
double a[N][N];

int gauss() {
    int c = 1, r = 1;
    for (; c <= n; ++c) {  //枚举列
        int tmp = r;
        rep(i, r, n) if (fabs(a[i][c]) > fabs(a[tmp][c])) tmp
 = i;  //寻找列中最大
        if (fabs(a[tmp][c]) < eps) continue;
        rep(i, c, n + 1) swap(a[tmp][i], a[r][i]);  //提到最
高行
        dep(i, n + 1, c) a[r][i] /= a[r][c];
        rep(i, r + 1, n) {  //用当前行将下面所有列消 0
            if (fabs(a[i][c]) > eps)
                dep(j, n + 1, c) a[i][j] -
= a[r][j] * a[i][c];
        }
        ++r;
    }
    if (r < n + 1) {
        rep(i, r, n) if (fabs(a[i][n + 1] > eps)) return 2;
        return 1;
    }
    dep(i, n, 1) rep(j, i + 1, n) a[i][n + 1] -
= a[i][j] * a[j][n + 1];
    return 0;
}

void solve() {
    cin >> n;
    rep(i, 1, n) rep(j, 1, n + 1) cin >> a[i][j];
    int flag = gauss();
    if (flag == 2)
        cout << "No solution" << endl;
    else if (flag == 1)
        cout << "Infinite group solutions" << endl;
    else
```

```
        rep(i, 1, n) cout << fixed << setprecision(2) << (fab
s(a[i][n + 1]) < eps ? 0 : a[i][n + 1]) << endl;
}
```

解异或方程组

【题目】给定包含n个方程n个未知数的异或线性方程组，方程组中的系数和常数为0或1，每个未知数的取值也为0或1，求解这个方程组

【思想】异或本质是不进位的加法。我们仍然可以借助高斯消元并且更加简单

```
const int N = 110;
const double eps = 1e-6;
int n;
int a[N][N];

int gauss() {
    int c = 1, r = 1;
    for (; c <= n; ++c) {  //枚举列
        int tmp = r;
        rep(i, r, n) if (a[i][c]) {
            tmp = i;  //寻找列中最
            break;
        }
        if (!a[tmp][c]) continue;
        rep(i, c, n + 1) swap(a[tmp][i], a[r][i]);  //提到最
高行
        rep(i, r + 1, n) {                          //用当前
行将下面所有列消 0
            if (a[i][c])
                dep(j, n + 1, c) a[i][j] ^= a[r][j];
        }
        ++r;
    }
    if (r < n + 1) {
        rep(i, r, n) if (a[i][n + 1]) return 2;
        return 1;
    }
    dep(i, n, 1) rep(j, i + 1, n) a[i][n + 1] ^= a[i][j] & a[
j][n + 1];
    return 0;
}

void solve() {
    cin >> n;
    rep(i, 1, n) rep(j, 1, n + 1) cin >> a[i][j];
    int flag = gauss();
```

```
    if (flag == 2)
        cout << "No solution" << endl;
    else if (flag == 1)
        cout << "Multiple sets of solutions" << endl;
    else
        rep(i, 1, n) cout << a[i][n + 1] << endl;
}
```

求组合数

组合数递推公式打表

【题目】最大2000次询问，每次询问 $C_a^b \bmod(10^9+7)$ 的值 $(1 \le b \le a \le 2000)$

【时间复杂度】预处理 $O(n^2)$

【思想】利用组合数递推公式打表预处理

【递推公式】$C_a^b = C_{a-1}^b + C_{a-1}^{b-1}$

```
const int N = 2010;
const int mod = 1e9 + 7;

int C[N][N];
void table() {
    rep(i, 0, 2000) rep(j, 0, i) {
        if (!j)
            C[i][j] = 1;
        else
            C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
    }
}

void solve() {
    int n;
    cin >> n;
    table();
    while (n--) {
        int a, b;
        cin >> a >> b;
        cout << C[a][b] << endl;
    }
}
```

组合数定义递推打表

【题目】最大100000次询问，每次询问 $C_a^b \bmod(10^9+7)$ 的值 $(1 \le b \le a \le 10^5)$

【时间复杂度】预处理 $O(n \log N)$

【思想】利用定义求将可能用到的数在模下的阶乘以及阶乘逆元预处理出来

【递推公式】$C_a^b = \dfrac{a!}{b!(a-b)!}$

```
const int N = 100005;
const int mod = 1e9 + 7;

int fact[N], infact[N];
int qpow(int x, int k) {
    int res = 1;
    while (k) {
        if (k & 1) res = (ll)res * x % mod;
        x = (ll)x * x % mod;
        k >>= 1;
    }
    return res;
}

void initFact() {
    fact[0] = infact[0] = 1;
    rep(i, 1, 100000) {
        fact[i] = (ll)fact[i - 1] * i % mod;
        infact[i] = (ll)infact[i - 1] * qpow(i, mod - 2) % mo
d;
    }
}

int C(int a, int b) {
    return ((ll)fact[a] * infact[b] % mod) * infact[a - b] %
mod;
}

void solve() {
    initFact();
    int n;
    cin >> n;
    while (n--) {
        int a, b;
        cin >> a >> b;
        cout << C(a, b) << endl;
    }
```

```
}
```

卢卡斯定理求大组合数

【题目】每次询问 $C_a^b \bmod p (1 \le a,b \le 10^{18}, p$为质数）

【时间复杂度】$O(plogNlogp)$

【卢卡斯定理】$C_a^b = C_{a\%p}^{b\%p} \bullet C_{a/p}^{b/p} (\bmod p)$

【思想】利用卢卡斯定理不断递归求，查询次数很大时组合数阶乘也可预处理

【组合数求法】求组合数用快速幂并简化循环次数为b次

$$C_a^b = \frac{a*(a-1)*(a-2)*\ldots*(a-b+1)*(a-b)*\ldots*1}{(a-b)*(a-b-1)*\ldots*1*b!} = \frac{a*(a-1)*(a-2)*\ldots(a-b+1)}{b!}$$

```
ll qpow(ll x, ll k, ll p) {
    ll res = 1;
    while (k) {
      if (k & 1) res = res * x % p;
        x = x * x % p;
        k >>= 1;
    }
    return res;
}

ll C(ll a, ll b, ll p) {
    ll res = 1;
    for (int i = 1, j = a; i <= b; ++i, --j) {
        res = res * j % p;
        res = res * qpow(i, p - 2, p) % p;
    }
    return res;
}

ll lucas(ll a, ll b, ll p) {
    if (a < p && b < p) return C(a, b, p);
    return C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}

void solve() {
    int n;
    cin >> n;
    while (n--) {
        ll a, b, p;
        cin >> a >> b >> p;
        cout << lucas(a, b, p) << endl;
```

```
    }
}
```

高精度+质因数拆解求超大质因数

【题目】求$C_a^b(1\le a,b\le 5000)$结果非常大

【时间复杂度】主要取决于高精度计算和筛法

【思想】根据定义$C_a^b=\frac{a!}{b!(a-b)!}$，我们将分子分母上的阶乘分别拆解成质因数相乘形式，那么结果可以通过用$a!$的质因子减去$b!$和$(a-b)!$的质因子后相乘求得。阶数质因子拆解部分可以将质数通过筛法提前预处理出来，最终结果可能会很大可以用高精度乘法

```
const int N = 5005;

bool notprime[N];
int sum[N];
vector<int> primes;

void getPrimes(int n) {
    rep(i, 2, n) {
        if (!notprime[i])
            primes.push_back(i);
        for (int j = 0; primes[j] <= n / i; ++j) {
            notprime[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}

int get(int n, int p) {  //获取 n!的质因数为 p 的个数
    int cnt = 0;
    while (n) {
        cnt += n / p;
        n /= p;
    }
    return cnt;
}

vector<int> mul(vector<int> &A, int b) {
    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); ++i) {
```

```
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }
    while (t) {
        C.push_back(t % 10);
        t /= 10;
    }
    while (C.size() > 1 && C.back() == 0)  //去前导零
        C.pop_back();
    return C;
}

void solve() {
    int a, b;
    cin >> a >> b;
    getPrimes(a);
    for (int i = 0; i < primes.size(); ++i) {
        int p = primes[i];
        sum[i] = get(a, p) - get(b, p) - get(a - b, p);
    }
    vector<int> res;
    res.push_back(1);
    for (int i = 0; i < primes.size(); ++i)
        for (int j = 0; j < sum[i]; ++j)
            res = mul(res, primes[i]);
    for (int i = res.size() - 1; i >= 0; --i) cout << res[i];
    cout << endl;
}
```