基础算法

头文件

```cpp
#include<bits/stdc++.h>
using namespace std;
#define fir first
#define sec second
#define endl "\n"
typedef long long ll;
typedef unsigned long long ull;
typedef pair<int,int> pii;
typedef pair<ll, ll> pll;
const int mod = 1e9 + 7, inf = 0x3f3f3f3f, P = 131;
int read()  //快读
{
    int x = 0, w = 1;
    char ch = 0;
    while (ch < '0' || ch > '9')
    {
        if (ch == '-') w = -1;
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9')
    {
        x = x * 10 + (ch - '0');
        ch = getchar();
    }
    return x * w;
}
void solve()
{

}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    #ifndef ONLINE_JUDGE
        freopen("test.in", "r", stdin);
        freopen("test.out", "w", stdout);
    #endif
    int t = 1;
    //cin >> t;
    while(t--)
        solve();

    return 0;
}
```

整数域二分

- x 或 x 的前驱

```
int l = 0, r = 1E8, ans = l;
while (l <= r) {
    int mid = (l + r) / 2;
    if (judge(mid)) {
        l = mid + 1;
        ans = mid;
    } else {
        r = mid - 1;
    }
}
return ans;
```

实数域二分

目前主流的写法是限制二分次数。

```
for (int t = 1; t <= 100; t++) {
    ld mid = (l + r) / 2;
    if (judge(mid)) r = mid;
    else l = mid;
}
cout << l << endl;
```

整数域三分

```
while (l < r) {
    int mid = (l + r) / 2;
    if (check(mid) <= check(mid + 1)) r = mid;
    else l = mid + 1;
}
cout << check(l) << endl;
```

实数域三分

限制次数实现。

```
ld l = -1E9, r = 1E9;
for (int t = 1; t <= 100; t++) {
    ld mid1 = (l * 2 + r) / 3;
    ld mid2 = (l + r * 2) / 3;
    if (judge(mid1) < judge(mid2)) {
        r = mid2;
    } else {
        l = mid1;
    }
}
cout << l << endl;
```

最近公共祖先 LCA

树上倍增解法

预处理时间复杂度 $\mathcal{O}(N\log N)$ ；单次查询 $\mathcal{O}(\log N)$ ，但是常数比树链剖分解法更大。

封装一：基础封装，针对无权图。

```
struct Tree {
    int n;
    vector<vector<int>> ver, val;
    vector<int> lg, dep;
    Tree(int n) {
        this->n = n;
        ver.resize(n + 1);
        val.resize(n + 1, vector<int>(30));
        lg.resize(n + 1);
        dep.resize(n + 1);
        for (int i = 1; i <= n; i++) { //预处理 log
            lg[i] = lg[i - 1] + (1 << lg[i - 1] == i);
        }
    }
    void add(int x, int y) { // 建立双向边
        ver[x].push_back(y);
        ver[y].push_back(x);
    }
    void dfs(int x, int fa) {
        val[x][0] = fa; // 储存 x 的父节点
        dep[x] = dep[fa] + 1;
        for (int i = 1; i <= lg[dep[x]]; i++) {
            val[x][i] = val[val[x][i - 1]][i - 1];
        }
        for (auto y : ver[x]) {
            if (y == fa) continue;
            dfs(y, x);
        }
    }
    int lca(int x, int y) {
        if (dep[x] < dep[y]) swap(x, y);
        while (dep[x] > dep[y]) {
            x = val[x][lg[dep[x] - dep[y]] - 1];
        }
        if (x == y) return x;
        for (int k = lg[dep[x]] - 1; k >= 0; k--) {
            if (val[x][k] == val[y][k]) continue;
            x = val[x][k];
            y = val[y][k];
        }
        return val[x][0];
    }
    int clac(int x, int y) { // 倍增查询两点间距离
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
    void work(int root = 1) { // 在此初始化
        dfs(root, 0);
```

```
    }
};
```

图论

常见概念

> oriented graph：有向图
>
> bidirectional edges：双向边

平面图：若能将无向图 $G=(V,E)$ 画在平面上使得任意两条无重合顶点的边不相交，则称 G 是平面图。

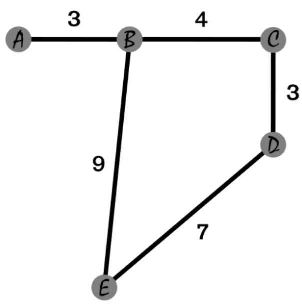
无向正权图上某一点的偏心距：记为 $ecc(u)=\max\left\{dist(u,v)\right\}$ ，即以这个点为源，到其他点的**所有最短路的最大值**。如下图 A 点，$ecc(A)$ 即为 12 。

图的直径：定义为 $d=\max\left\{ecc(u)\right\}$ ，即**最大的偏心距**，亦可以简化为图中最远的一对点的距离。

图的中心：定义为 $arg=\min\left\{ecc(u)\right\}$ ，即**偏心距最小的点**。如下图，图的中心即为 B 点。

图的绝对中心：可以定义在边上的图的中心。

图的半径：图的半径不同于圆的半径，其不等于直径的一半（但对于绝对中心定义上的直径而言是一半）。定义为 $r=\min\left\{ecc(u)\right\}$ ，即**中心的偏心距**。计算方式：使用全源最短路，计算出所有点的偏心距，再加以计算。



单源最短路径（SSSP问题）

（正权稀疏图）动态数组存图+Djikstra算法

使用优先队列优化，以 $\mathcal{O}(M\log N)$ 的复杂度计算。

```cpp
vector<int> dis(n + 1, 1E18);
auto djikstra = [&](int s = 1) -> void {
    using PII = pair<int, int>;
    priority_queue<PII, vector<PII>, greater<PII>> q;
    q.emplace(0, s);
    dis[s] = 0;
    vector<int> vis(n + 1);
    while (!q.empty()) {
        int x = q.top().second;
        q.pop();
        if (vis[x]) continue;
        vis[x] = 1;
```

```
        for (auto [y, w] : ver[x]) {
            if (dis[y] > dis[x] + w) {
                dis[y] = dis[x] + w;
                q.emplace(dis[y], y);
            }
        }
    }
};
```

(负权图、判负环）Bellman-ford 算法

使用结构体存边（该算法无需存图），以 $\mathcal{O}(NM)$ 的复杂度计算。

```
int n, m, s;
cin >> n >> m >> s;

vector<tuple<int, int, i64>> ver(m + 1);
for (int i = 1; i <= m; ++i) {
    int x, y;
    i64 w;
    cin >> x >> y >> w;
    ver[i] = {x, y, w};
}

vector<i64> dis(n + 1, inf), chk(n + 1);
dis[s] = 0;
for (int i = 1; i <= 2 * n; ++i) { // 双倍松弛，获取负环信息
    vector<i64> backup = dis;
    for (int j = 1; j <= m; ++j) {
        auto [x, y, w] = ver[j];
        chk[y] |= (i > n && backup[x] + w < dis[y]);
        dis[y] = min(dis[y], backup[x] + w);
    }
}

for (int i = 1; i <= n; ++i) {
    if (i == s) {
        cout << 0 << " ";
    } else if (dis[i] >= inf / 2) {
        cout << "no ";
    } else if (chk[i]) {
        cout << "inf ";
    } else {
        cout << dis[i] << " ";
    }
}
```

(负权图）SPFA 算法

以 $\mathcal{O}(KM)$ 的复杂度计算，其中 K 虽然为常数，但是可以通过特殊的构造退化成接近 N ，需要注意被卡。

```
const int N = 1e5 + 7, M = 1e6 + 7;
int n, m;
```

```
int ver[M], ne[M], h[N], edge[M], tot;
int d[N], v[N];

void add(int x, int y, int w) {
    ver[++ tot] = y, ne[tot] = h[x], h[x] = tot;
    edge[tot] = w;
}
void spfa() {
    ms(d, 0x3f); d[1] = 0;
    queue<int> q; q.push(1);
    v[1] = 1;
    while(!q.empty()) {
        int x = q.front(); q.pop(); v[x] = 0;
        for (int i = h[x]; i; i = ne[i]) {
            int y = ver[i];
            if(d[y] > d[x] + edge[i]) {
                d[y] = d[x] + edge[i];
                if(v[y] == 0) q.push(y), v[y] = 1;
            }
        }
    }
}
int main() {
    cin >> n >> m;
    for (int i = 1; i <= m; ++ i) {
        int x, y, w; cin >> x >> y >> w;
        add(x, y, w);
    }
    spfa();
    for (int i = 1; i <= n; ++ i) {
        if (d[i] == INF) cout << "N" << endl;
        else cout << d[n] << endl;
    }
}
```

最小生成树（MST问题）

（稀疏图）Prim算法

使用邻接矩阵存图，以 $\mathcal{O}(N^2 + M)$ 的复杂度计算，思想与 `djikstra` 基本一致。

```
const int N = 550, INF = 0x3f3f3f3f;
int n, m, g[N][N];
int d[N], v[N];
int prim() {
    ms(d, 0x3f); //这里的d表示到“最小生成树集合”的距离
    int ans = 0;
    for (int i = 0; i < n; ++ i) { //遍历 n 轮
        int t = -1;
        for (int j = 1; j <= n; ++ j)
            if (v[j] == 0 && (t == -1 || d[j] < d[t])) //如果这个点不在集合内且当前距离集合最近
                t = j;
        v[t] = 1; //将t加入“最小生成树集合”
        if (i && d[t] == INF) return INF; //如果发现不连通，直接返回
```

```
        if (i) ans += d[t];
        for (int j = 1; j <= n; ++ j) d[j] = min(d[j], g[t][j]); //用t更新其他点到集合的距离
    }
    return ans;
}
int main() {
    ms(g, 0x3f); cin >> n >> m;
    while (m -- ) {
        int x, y, w; cin >> x >> y >> w;
        g[x][y] = g[y][x] = min(g[x][y], w);
    }
    int t = prim();
    if (t == INF) cout << "impossible" << endl;
    else cout << t << endl;
} //22.03.19已测试
```

（稠密图）Kruskal算法

平均时间复杂度为 $\mathcal{O}(M\log M)$ ，简化了并查集。

```
struct DSU {
    vector<int> fa;
    DSU(int n) : fa(n + 1) {
        iota(fa.begin(), fa.end(), 0);
    }
    int get(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
    }
    bool merge(int x, int y) { // 设x是y的祖先
        x = get(x), y = get(y);
        if (x == y) return false;
        fa[y] = x;
        return true;
    }
    bool same(int x, int y) {
        return get(x) == get(y);
    }
};
struct Tree {
    using TII = tuple<int, int, int>;
    int n;
    priority_queue<TII, vector<TII>, greater<TII>> ver;

    Tree(int n) {
        this->n = n;
    }
    void add(int x, int y, int w) {
        ver.emplace(w, x, y); // 注意顺序
    }
    int kruskal() {
        DSU dsu(n);
```

```
        int ans = 0, cnt = 0;
        while (ver.size()) {
            auto [w, x, y] = ver.top();
            ver.pop();
            if (dsu.same(x, y)) continue;
            dsu.merge(x, y);
            ans += w;
            cnt++;
        }
        assert(cnt < n - 1); // 输入有误，建树失败
        return ans;
    }
};
```

数论

常见数列

调和级数

满足调和级数 $\mathcal{O}\left(\frac{N}{1}+\frac{N}{2}+\frac{N}{3}+\cdots+\frac{N}{N}\right)$，可以用 $\approx N\ln N$ 来拟合，但是会略小，误差量级在 10% 左右。本地可以在500ms内完成 10^8 量级的预处理计算。

N的量级	1	2	3	4	5	6	7	8	9
累加和	27	482	7'069	93'668	1'166'750	13'970'034	162'725'364	1'857'511'568	20'877'697'634

下方示例为求解 1 到 N 中各个数字的因数值。

```
const int N = 1E5;
vector<vector<int>> dic(N + 1);
for (int i = 1; i <= N; i++) {
    for (int j = i; j <= N; j += i) {
        dic[j].push_back(i);
    }
}
```

素数密度与分布

N的量级	1	2	3	4	5	6	7	8	9
素数数量	4	25	168	1'229	9'592	78'498	664'579	5'761'455	50'847'534

除此之外，对于任意两个相邻的素数 $p_1,p_2\le 10^9$ ，有 $|p_1-p_2|<300$ 成立，更具体的说，最大的差值为 282 。

因数最多数字与其因数数量

N的量级	1	2	3	4	5	6	7
因数最多数字的因数数量	4	25	32	64	128	240	448
因数最多的数字	-	-	-	7560, 9240	83160, 98280	720720, 831600, 942480, 982800, 997920	-

快速幂

常规

```
i64 pow(i64 a, int b, int m) { // 复杂度是 log N
    i64 r = 1 % m; /**! 这里的取模容易遗漏 */
    for (; b; b >>= 1, a = a * a % m) {
        if (b & 1) r = r * a % m;
    }
    return r;
}
```

长整型 with 防爆乘法

```
i64 mul(i64 a, i64 b, i64 m) { // 由于不取模，运行速度非常快
    a %= m, b %= m; /**! 这里的取模容易遗漏 */
    i64 r = a * b - m * i64(1.L / m * a * b);
    return r - m * (r >= m) + m * (r < 0);
}

i64 mul(i64 a, i64 b, i64 m) { // 比较慢
    return (__int128)a * b % m;
}

i64 pow(i64 a, i64 b, i64 m) {
    i64 res = 1 % m; /**! 这里的取模容易遗漏 */
    for (; b; b >>= 1, a = mul(a, a, m)) { // 配合下方手写乘法
        if (b & 1) res = mul(res, a, m);
    }
    return res;
}
```

质数判定

试除法

标准 $\mathcal{O}(\sqrt{N})$，有常数优化版本可达到 $\mathcal{O}(\frac{\sqrt{N}}{3})$ 。

```cpp
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;
}
```

筛法

使用欧拉筛（线性筛），预期时间复杂度为 $\mathcal{O}(N)$ 。

```cpp
vector<int> prime, minp;

void sieve(int n = 1e7) {
    minp.resize(n + 1);
    for (int i = 2; i <= n; i++) {
        if (!minp[i]) {
            minp[i] = i;
            prime.push_back(i);
        }
        for (auto j : prime) {
            if (j > minp[i] || j > n / i) break;
            minp[i * j] = j;
        }
    }
}

bool isprime(int n) {
    return minp[n] == n;
}
```

Miller-Rabin

随机化验证，非严谨计算的平均复杂度约为 $\mathcal{O}(3.5 \times \log X)$ 。对于某些强力质数，可能会退化至约 $\mathcal{O}(35 \times \log X)$ 。

```cpp
i64 mul(i64 a, i64 b, i64 m) { // 快速乘提速，约四倍效果
    i64 r = a * b - m * i64(1.L / m * a * b);
    return r - m * (r >= m) + m * (r < 0);
}

i64 pow(i64 a, i64 b, i64 m) {
    i64 res = 1 % m;
    for (; b; b >>= 1, a = mul(a, a, m)) {
        if (b & 1) res = mul(res, a, m);
    }
    return res;
}

bool isprime(i64 n) {
    if (n < 2 || n % 6 % 4 != 1) {
        return (n | 1) == 3;
    }
```

```
    i64 s = __builtin_ctzll(n - 1), d = n >> s;
    for (i64 a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
        i64 p = pow(a % n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--) {
            p = mul(p, p, n);
        }
        if (p != n - 1 && i != s) return false;
    }
    return true;
}
```

质因子分解

筛法

使用欧拉筛（线性筛），预处理时间复杂度 $\mathcal{O}(N)$ ，单次查询 $\mathcal{O}(\text{Prime Numer})$ 。

```
vector<int> prime, minp, maxp;

void sieve(int n = 1e7) {
    minp.resize(n + 1);
    maxp.resize(n + 1);
    for (int i = 2; i <= n; i++) {
        if (!minp[i]) {
            minp[i] = maxp[i] = i;
            prime.push_back(i);
        }
        for (auto j : prime) {
            if (j > minp[i] || j > n / i) break;
            minp[i * j] = j;
            maxp[i * j] = maxp[i];
        }
    }
}

vector<array<int, 2>> factorize(int n) {
    vector<array<int, 2>> ans;
    while (n > 1) {
        int now = minp[n], cnt = 0;
        while (n % now == 0) {
            n /= now;
            cnt++;
        }
        ans.push_back({now, cnt});
    }
    return ans;
}
```

Pollard-Rho

以单个因子 $\mathcal{O}(\log X)$ 的复杂度输出数字 X 的全部质因数，由于需要结合素数测试，总复杂度会略高一些。如果遇到超时的情况，可能需要考虑进一步优化，例如检查题目是否强制要求枚举全部质因数等等。有常数优化版本可以再快五倍。

```cpp
i64 rho(i64 n) {
    if (!(n & 1)) return 2;
    i64 x = 0, y = 0, prod = 1;
    auto f = [&](i64 x) -> i64 {
        return mul(x, x, n) + 5; // 这里的种子为 1 时能被 hack，取 5 到目前为止没有什么问题
    };
    for (int t = 30, z = 0; t % 64 || gcd(prod, n) == 1; ++t) {
        if (x == y) x = ++z, y = f(x);
        if (i64 q = mul(prod, x + n - y, n)) prod = q;
        x = f(x), y = f(f(y));
    }
    return gcd(prod, n);
}

vector<i64> factorize(i64 x) {
    vector<i64> res;
    auto f = [&](auto f, i64 x) {
        if (x == 1) return;
        if (isprime(x)) return res.push_back(x);
        i64 y = rho(x);
        f(f, y), f(f, x / y);
    };
    f(f, x), sort(res.begin(), res.end());
    return res;
}
```

裴蜀定理

> $ax + by = c\ (x \in Z^*, y \in Z^*)$ 成立的充要条件是 $gcd(a, b) \mid c$（Z^* 表示正整数集）。

例题：给定一个序列 a，找到一个序列 x，使得 $\sum_{i=1}^{n} a_i x_i$ 最小。

```cpp
LL n, a, ans;
LL gcd(LL a, LL b){
    return b ? gcd(b, a % b) : a;
}
int main(){
    cin >> n;
    for (int i = 0; i < n; i ++ ){
        cin >> a;
        if (a < 0) a = -a;
        ans = gcd(ans, a);
    }
    cout << ans << "\n";
    return 0;
}
```

逆元

费马小定理解（借助快速幂）

单次计算的复杂度即为快速幂的复杂度 $\mathcal{O}(\log X)$ 。限制：MOD 必须是质数，且需要满足 x 与 MOD 互质。

```
LL inv(LL x) { return mypow(x, mod - 2, mod);}
```

扩展欧几里得解

此方法的 MOD 没有限制，复杂度为 $\mathcal{O}(\log X)$ ，但是比快速幂法常数大一些。

```
int x, y;
int exgcd(int a, int b, int &x, int &y) { //扩展欧几里得算法
    if (b == 0) {
        x = 1, y = 0;
        return a; //到达递归边界开始向上一层返回
    }
    int r = exgcd(b, a % b, x, y);
    int temp = y; //把x y变成上一层的
    y = x - (a / b) * y;
    x = temp;
    return r; //得到a b的最大公因数
}
LL getInv(int a, int mod) { //求a在mod下的逆元，不存在逆元返回-1
    LL x, y, d = exgcd(a, mod, x, y);
    return d == 1 ? (x % mod + mod) % mod : -1;
}
```

离线求解：线性递推解

以 $\mathcal{O}(N)$ 的复杂度完成 $1-N$ 中全部逆元的计算。

```
inv[1] = 1;
for (int i = 2; i <= n; i ++ )
    inv[i] = (p - p / i) * inv[p % i] % p;
```

扩展欧几里得 exgcd

求解形如 $a \cdot x + b \cdot y = \gcd(a, b)$ 的不定方程的任意一组解。

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

例题：求解二元一次不定方程 $A \cdot x + B \cdot y = C$ 。

```
auto clac = [&](int a, int b, int c) {
```

```cpp
    int u = 1, v = 1;
    if (a < 0) { // 负数特判，但是没用经过例题测试
        a = -a;
        u = -1;
    }
    if (b < 0) {
        b = -b;
        v = -1;
    }

    int x, y, d = exgcd(a, b, x, y), ans;
    if (c % d != 0) { // 无整数解
        cout << -1 << "\n";
        return;
    }
    a /= d, b /= d, c /= d;
    x *= c, y *= c; // 得到可行解

    ans = (x % b + b - 1) % b + 1;
    auto [A, B] = pair{u * ans, v * (c - ans * a) / b}; // x最小正整数 特解

    ans = (y % a + a - 1) % a + 1;
    auto [C, D] = pair{u * (c - ans * b) / a, v * ans}; // y最小正整数 特解

    int num = (C - A) / b + 1; // xy均为正整数 的 解的组数
};
```

离散对数 bsgs 与 exbsgs

以 $\mathcal{O}(\sqrt{P})$ 的复杂度求解 $a^x \equiv b(\bmod P)$ 。其中标准 BSGS 算法不能计算 a 与 MOD 互质的情况，而 exbsgs 则可以。

```cpp
namespace BSGS {
LL a, b, p;
map<LL, LL> f;
inline LL gcd(LL a, LL b) { return b > 0 ? gcd(b, a % b) : a; }
inline LL ps(LL n, LL k, int p) {
    LL r = 1;
    for (; k; k >>= 1) {
        if (k & 1) r = r * n % p;
        n = n * n % p;
    }
    return r;
}
void exgcd(LL a, LL b, LL &x, LL &y) {
    if (!b)
        x = 1, y = 0;
    } else {
        exgcd(b, a % b, x, y);
        LL t = x;
        x = y;
        y = t - a / b * y;
    }
}
LL inv(LL a, LL b) {
```

```
    LL x, y;
    exgcd(a, b, x, y);
    return (x % b + b) % b;
}
LL bsgs(LL a, LL b, LL p) {
    f.clear();
    int m = ceil(sqrt(p));
    b %= p;
    for (int i = 1; i <= m; i++) {
        b = b * a % p;
        f[b] = i;
    }
    LL tmp = ps(a, m, p);
    b = 1;
    for (int i = 1; i <= m; i++) {
        b = b * tmp % p;
        if (f.count(b) > 0) return (i * m - f[b] + p) % p;
    }
    return -1;
}
LL exbsgs(LL a, LL b, LL p) {
    if (b == 1 || p == 1) return 0;
    LL g = gcd(a, p), k = 0, na = 1;
    while (g > 1) {
        if (b % g != 0) return -1;
        k++;
        b /= g;
        p /= g;
        na = na * (a / g) % p;
        if (na == b) return k;
        g = gcd(a, p);
    }
    LL f = bsgs(a, b * inv(na, p) % p, p);
    if (f == -1) return -1;
    return f + k;
}
} // namespace BSGS

using namespace BSGS;

int main() {
    IOS;
    cin >> p >> a >> b;
    a %= p, b %= p;
    LL ans = exbsgs(a, b, p);
    if (ans == -1) cout << "no solution\n";
    else cout << ans << "\n";
    return 0;
}
```

欧拉函数

直接求解单个数的欧拉函数

1 到 N 中与 N 互质数的个数称为欧拉函数，记作 $\varphi(N)$ 。求解欧拉函数的过程即为分解质因数的过程，复杂度 $\mathcal{O}(\sqrt{n})$ 。

```cpp
int phi(int n) { //求解 phi(n)
    int ans = n;
    for(int i = 2; i <= n / i; i ++) { //注意，这里要写 n / i ，以防止 int 型溢出风险和 sqrt 超时风险
        if(n % i == 0) {
            ans = ans / i * (i - 1);
            while(n % i == 0) n /= i;
        }
    }
    if(n > 1) ans = ans / n * (n - 1); //特判 n 为质数的情况
    return ans;
}
```

求解 1 到 N 所有数的欧拉函数

利用上述第四条性质，我们可以快速递推出 $2-N$ 中每个数的欧拉函数，复杂度 $\mathcal{O}(N)$ ，而该算法**即是线性筛的算法**。

```cpp
const int N = 1e5 + 7;
int v[N], prime[N], phi[N];
void euler(int n) {
    ms(v, 0); //最小质因子
    int m = 0; //质数数量
    for (int i = 2; i <= n; ++ i) {
        if (v[i] == 0) { // i 是质数
            v[i] = i, prime[++ m] = i;
            phi[i] = i - 1;
        }
         //为当前的数 i 乘上一个质因子
        for (int j = 1; j <= m; ++ j) {
             //如 i 有比 prime[j] 更小的质因子，或超出 n ，停止
            if(prime[j] > v[i] || prime[j] > n / i) break;
             // prime[j] 是合数 i * prime[j] 的最小质因子
            v[i * prime[j]] = prime[j];
            phi[i * prime[j]] = phi[i] * (i % prime[j] ? prime[j] - 1 : prime[j]);
        }
    }
}
int main() {
    int n; cin >> n; euler(n);
    for (int i = 1; i <= n; ++ i) cout << phi[i] << endl;
    return 0;
}
```

使用莫比乌斯反演求解欧拉函数

```
int phi[N];
vector<int> fac[N];
void get_eulers() {
    for (int i = 1; i <= N - 10; i++) {
        for (int j = i; j <= N - 10; j += i) {
            fac[j].push_back(i);
        }
    }
    phi[1] = 1;
    for (int i = 2; i <= N - 10; i++) {
        phi[i] = i;
        for (auto j : fac[i]) {
            if (j == i) continue;
            phi[i] -= phi[j];
        }
    }
}
```

组合数

debug

提供一组测试数据：$\binom{132}{66}=$ 377'389'666'165'540'953'244'592'352'291'892'721'700，模数为 998244353 时为 $241'200'029$；10^9+7 时为 598375978。

逆元+卢卡斯定理（质数取模）

$\mathcal{O}(N)$，模数必须为质数。

```
struct Comb {
    int n;
    vector<Z> _fac, _inv;

    Comb() : _fac{1}, _inv{0} {}
    Comb(int n) : Comb() {
        init(n);
    }
    void init(int m) {
        if (m <= n) return;
        _fac.resize(m + 1);
        _inv.resize(m + 1);
        for (int i = n + 1; i <= m; i++) {
            _fac[i] = _fac[i - 1] * i;
        }
        _inv[m] = _fac[m].inv();
        for (int i = m; i > n; i--) {
            _inv[i - 1] = _inv[i] * i;
        }
        n = m;
    }
    Z fac(int x) {
        if (x > n) init(x);
```

```
        return _fac[x];
    }
    Z inv(int x) {
        if (x > n) init(x);
        return _inv[x];
    }
    Z C(int x, int y) {
        if (x < 0 || y < 0 || x < y) return 0;
        return fac(x) * inv(y) * inv(x - y);
    }
    Z P(int x, int y) {
        if (x < 0 || y < 0 || x < y) return 0;
        return fac(x) * inv(x - y);
    }
} comb(1 << 21);
```

质因数分解

此法适用于：$1 < n, m, MOD < 10^7$ 的情况。

```
int n,m,p,b[10000005],prime[1000005],t,min_prime[10000005];
void euler_Prime(int n){//用欧拉筛求出1~n中每个数的最小质因数的编号是多少，保存在min_prime中
    for(int i=2;i<=n;i++){
        if(b[i]==0){
            prime[++t]=i;
            min_prime[i]=t;
        }
        for(int j=1;j<=t&&i*prime[j]<=n;j++){
            b[prime[j]*i]=1;
            min_prime[prime[j]*i]=j;
            if(i%prime[j]==0) break;
        }
    }
}
long long c(int n,int m,int p){//计算C(n,m)%p的值
    euler_Prime(n);
    int a[t+5];//t代表1~n中质数的个数 ，a[i]代表编号为i的质数在答案中出现的次数
    for(int i=1;i<=t;i++) a[i]=0;//注意清0，一开始是随机数
    for(int i=n;i>=n-m+1;i--){//处理分子
        int x=i;
        while (x!=1){
            a[min_prime[x]]++;//注意min_prime中保存的是这个数的最小质因数的编号（1~t）
            x/=prime[min_prime[x]];
        }
    }
    for(int i=1;i<=m;i++){//处理分母
        int x=i;
        while (x!=1){
            a[min_prime[x]]--;
            x/=prime[min_prime[x]];
        }
    }
    long long ans=1;
```

```
    for(int i=1;i<=t;i++){//枚举质数的编号，看它出现了几次
        while(a[i]>0){
            ans=ans*prime[i]%p;
            a[i]--;
        }
    }
    return ans;
}
int main(){
    cin>>n>>m;
    m=min(m,n-m);//小优化
    cout<<C(n,m,MOD);
}
```

杨辉三角（精确计算）

60 以内 `long long` 可解，130 以内 `__int128` 可解。

```
vector C(n + 1, vector<int>(n + 1));
C[0][0] = 1;
for (int i = 1; i <= n; i++) {
    C[i][0] = 1;
    for (int j = 1; j <= n; j++) {
        C[i][j] = C[i - 1][j] + C[i - 1][j - 1];
    }
}
cout << C[n][m] << endl;
```

求解连续数字的正约数集合——倍数法

使用规律递推优化，时间复杂度为 $\mathcal{O}(N\log N)$ ，如果不需要详细的输出集合，则直接将 `vector` 换为普通数组即可（时间更快）。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 7;
vector<int> f[N];

void divide(int n) {
    for (int i = 1; i <= n; ++ i)
        for (int j = 1; j <= n / i; ++ j)
            f[i * j].push_back(i);
    for (int i = 1; i <= n; ++ i) {
        for (auto it : f[i]) cout << it << " ";
        cout << endl;
    }
}
int main() {
    int x; cin >> x; divide(x);
    return 0;
}
```

容斥原理

> 定义：
> $\left|S_1 \cup S_2 \cup S_3 \cup \ldots \cup S_n\right| = \sum_{i=1}^{N} |S_i| - \sum_{i,j=1}^{N} \left|S_i \cap S_j\right| + \sum_{i,j,k=1}^{N} \left|S_i \cap S_j \cap S_k\right| - \ldots$

例题：给定一个整数 n 和 m 个不同的质数 $p_1, p_2, \ldots, p_m$，请你求出 1 ~ n 中能被 $p_1, p_2, \ldots, p_m$ 中的至少一个数整除的整数有多少个。

二进制枚举解

```
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n, m;
    cin >> n >> m;
    vector <LL> p(m);
    for (int i = 0; i < m; i ++ )
        cin >> p[i];
    LL ans = 0;
    for (int i = 1; i < (1 << m); i ++ ){
        LL t = 1, cnt = 0;
        for (int j = 0; j < m; j ++ ){
            if (i >> j & 1){
                cnt ++ ;
                t *= p[j];
                if (t > n){
                    t = -1;
                    break;
                }
            }
        }
        if (t != -1){
            if (cnt & 1) ans += n / t;
            else ans -= n / t;
        }
    }
    cout << ans << "\n";
    return 0;
}
```

dfs 解

```
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n, m;
    cin >> n >> m;
    vector <LL> p(m);
    for (int i = 0; i < m; i ++ )
        cin >> p[i];
    LL ans = 0;
    function<void(LL, LL, LL)> dfs = [&](LL x, LL s, LL odd){
        if (x == m){
            if (s == 1) return;
            ans += odd * (n / s);
```

```
            return;
        }
        dfs(x + 1, s, odd);
        if (s <= n / p[x]) dfs(x + 1, s * p[x], -odd);
    };
    dfs(0, 1, -1);
    cout << ans << "\n";
    return 0;
}
```

同余方程组、拓展中国剩余定理 excrt

公式：$x \equiv b_i \pmod{a_i}$，即 $(x - b_i) \mid a_i$。

```
int n; LL ai[maxn], bi[maxn];
inline int mypow(int n, int k, int p) {
    int r = 1;
    for (; k; k >>= 1, n = n * n % p)
        if (k & 1) r = r * n % p;
    return r;
}
LL exgcd(LL a, LL b, LL &x, LL &y) {
    if (b == 0) { x = 1, y = 0; return a; }
    LL gcd = exgcd(b, a % b, x, y), tp = x;
    x = y, y = tp - a / b * y;
    return gcd;
}
LL excrt() {
    LL x, y, k;
    LL M = bi[1], ans = ai[1];
    for (int i = 2; i <= n; ++ i) {
        LL a = M, b = bi[i], c = (ai[i] - ans % b + b) % b;
        LL gcd = exgcd(a, b, x, y), bg = b / gcd;
        if (c % gcd != 0) return -1;
        x = mul(x, c / gcd, bg);
        ans += x * M;
        M *= bg;
        ans = (ans % M + M) % M;
    }
    return (ans % M + M) % M;
}
int main() {
    cin >> n;
    for (int i = 1; i <= n; ++ i) cin >> bi[i] >> ai[i];
    cout << excrt() << endl;
    return 0;
}
```

求解连续按位异或

以 $\mathcal{O}(1)$ 复杂度计算 $0 \oplus 1 \oplus \cdots \oplus n$。

```
unsigned xor_n(unsigned n) {
    unsigned t = n & 3;
    if (t & 1) return t / 2u ^ 1;
    return t / 2u ^ n;
}
```

```
i64 xor_n(i64 n) {
    if (n % 4 == 1) return 1;
    else if (n % 4 == 2) return n + 1;
    else if (n % 4 == 3) return 0;
    else return n;
}
```

高斯消元求解线性方程组

题目大意：输入一个包含 N 个方程 N 个未知数的线性方程组，系数与常数均为实数（两位小数）。求解这个方程组。如果存在唯一解，则输出所有 N 个未知数的解，结果保留两位小数。如果无数解，则输出 X，如果无解，则输出 N。

```
const int N = 110;
const double eps = 1e-8;
LL n;
double a[N][N];
LL gauss(){
    LL c, r;
    for (c = 0, r = 0; c < n; c ++ ){
        LL t = r;
        for (int i = r; i < n; i ++ )    //找到绝对值最大的行
            if (fabs(a[i][c]) > fabs(a[t][c]))
                t = i;
        if (fabs(a[t][c]) < eps) continue;
        for (int j = c; j < n + 1; j ++ ) swap(a[t][j], a[r][j]);    //将绝对值最大的一行换到最顶端
        for (int j = n; j >= c; j -- ) a[r][j] /= a[r][c];    //将当前行首位变成 1
        for (int i = r + 1; i < n; i ++ )    //将下面列消成 0
            if (fabs(a[i][c]) > eps)
                for (int j = n; j >= c; j -- )
                    a[i][j] -= a[r][j] * a[i][c];
        r ++ ;
    }
    if (r < n){
        for (int i = r; i < n; i ++ )
            if (fabs(a[i][n]) > eps)
                return 2;
        return 1;
    }
    for (int i = n - 1; i >= 0; i -- )
        for (int j = i + 1; j < n; j ++ )
            a[i][n] -= a[i][j] * a[j][n];
    return 0;
}
int main(){
```

```
    cin >> n;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < n + 1; j ++ )
            cin >> a[i][j];
    LL t = gauss();
    if (t == 0){
        for (int i = 0; i < n; i ++ ){
            if (fabs(a[i][n]) < eps) a[i][n] = abs(a[i][n]);
            printf("%.2lf\n", a[i][n]);
        }
    }
    else if (t == 1) cout << "Infinite group solutions\n";
    else cout << "No solution\n";
    return 0;
}

```

康拓展开

正向展开普通解法

将一个字典序排列转换成序号。例如：12345->1，12354->2。

```
int f[20];
void jie_cheng(int n) { // 打出1-n的阶乘表
    f[0] = f[1] = 1; // 0的阶乘为1
    for (int i = 2; i <= n; i++) f[i] = f[i - 1] * i;
}
string str;
int kangtuo() {
    int ans = 1; // 注意，因为 12345 是算作0开始计算的，最后结果要把12345看作是第一个
    int len = str.length();
    for (int i = 0; i < len; i++) {
        int tmp = 0; // 用来计数的
        // 计算str[i]是第几大的数，或者说计算有几个比他小的数
        for (int j = i + 1; j < len; j++)
            if (str[i] > str[j]) tmp++;
        ans += tmp * f[len - i - 1];
    }
    return ans;
}
int main() {
    jie_cheng(10);
    string str = "52413";
    cout << kangtuo() << endl;
}
```

正向展开树状数组解

给定一个全排列，求出它是 1 ~ n 所有全排列的第几个，答案对 998244353 取模。

答案就是 $\sum_{i=1}^{n} res_{a_i}(n-i)!$ 。res_x 表示剩下的比 x 小的数字的数量，通过**树状数组**处理。

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp
#define LL long long
const int mod = 998244353, N = 1e6 + 10;
LL fact[N];
struct fwt{
    LL n;
    vector <LL> a;
    fwt(LL n) : n(n), a(n + 1) {}
    LL sum(LL x){
        LL res = 0;
        for (; x; x -= x & -x)
            res += a[x];
        return res;
    }
    void add(LL x, LL k){
        for (; x <= n; x += x & -x)
            a[x] += k;
    }
    LL query(LL x, LL y){
        return sum(y) - sum(x - 1);
    }
};
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    LL n;
    cin >> n;
    fwt a(n);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++ ){
        fact[i] = fact[i - 1] * i % mod;
        a.add(i, 1);
    }
    LL ans = 0;
    for (int i = 1; i <= n; i ++ ){
        LL x;
        cin >> x;
        ans = (ans + a.query(1, x - 1) * fact[n - i] % mod ) % mod;
        a.add(x, -1);
    }
    cout << (ans + 1) % mod << "\n";
    return 0;
}
```

逆向还原

```cpp
string str;
int kangtuo(){
    int ans = 1;  //注意，因为 12345 是算作0开始计算的，最后结果要把12345看作是第一个
    int len = str.length();
    for(int i = 0; i < len; i++){
        int tmp = 0;//用来计数的
        for(int j = i + 1; j < len; j++){
            if(str[i] > str[j]) tmp++;
            //计算str[i]是第几大的数，或者说计算有几个比他小的数
        }
        ans += tmp * f[len - i - 1];
```

```
    }
    return ans;
}
int main(){
    jie_cheng(10);
    string str = "52413";
    cout<<kangtuo()<<endl;
}
```

Min25 筛

求解 $1-N$ 的质数和，其中 $N \leq 10^{10}$ 。

```
namespace min25{
    const int N = 1000000 + 10;
    int prime[N], id1[N], id2[N], flag[N], ncnt, m;
    LL g[N], sum[N], a[N], T;
    LL n;
    LL mod;
    inline LL ps(LL n,LL k) {LL r=1;for(;k;k>>=1){if(k&1)r=r*n%mod;n=n*n%mod;}return r;}
    void finit(){ // 最开始清0
        memset(g, 0, sizeof(g));
        memset(a, 0, sizeof(a));
        memset(sum, 0, sizeof(sum));
        memset(prime, 0, sizeof(prime));
        memset(id1, 0, sizeof(id1));
        memset(id2, 0, sizeof(id2));
        memset(flag, 0, sizeof(flag));
        ncnt = m = 0;
    }
    int ID(LL x) {
        return x <= T ? id1[x] : id2[n / x];
    }

    LL calc(LL x) {
        return x * (x + 1) / 2 - 1;
    }

    LL init(LL x) {
        T = sqrt(x + 0.5);
        for (int i = 2; i <= T; i++) {
            if (!flag[i]) prime[++ncnt] = i, sum[ncnt] = sum[ncnt - 1] + i;
            for (int j = 1; j <= ncnt && i * prime[j] <= T; j++) {
                flag[i * prime[j]] = 1;
                if (i % prime[j] == 0) break;
            }
        }
        for (LL l = 1; l <= x; l = x / (x / l) + 1) {
            a[++m] = x / l;
            if (a[m] <= T) id1[a[m]] = m; else id2[x / a[m]] = m;
            g[m] = calc(a[m]);
        }
        for (int i = 1; i <= ncnt; i++)
            for (int j = 1; j <= m && (LL) prime[i] * prime[i] <= a[j]; j++)
```

```
                g[j] = g[j] - (LL) prime[i] * (g[ID(a[j] / prime[i])] - sum[i - 1]);
    }
    LL solve(LL x) {
        if (x <= 1) return x;
        return n = x, init(n), g[ID(n)];
    }
}

using namespace min25;

int main() {
    // while (1) {
    int tt;
    scanf("%d",&tt);
    while(tt--){
        finit();
        scanf("%lld%lld", &n, &mod);
        LL ans = (n + 3) % mod * n % mod  * ps(2 , mod - 2) % mod + solve(n + 1) - 4;
        // cout << solve(n) << endl;
        // ans = (ans + mod) % mod;
        ans = (ans + mod) % mod;
        printf("%lld\n", ans);
    }

    // }
}
```

矩阵四则运算

封装来自。矩阵乘法复杂度 $\mathcal{O}(N^3)$。

```
const int SIZE = 2;
struct Matrix {
    ll M[SIZE + 5][SIZE + 5];
    void clear() { memset(M, 0, sizeof(M)); }
    void reset() { //初始化
        clear();
        for (int i = 1; i <= SIZE; ++i) M[i][i] = 1;
    }
    Matrix friend operator*(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 1; i <= SIZE; ++i)
            for (int j = 1; j <= SIZE; ++j)
                for (int k = 1; k <= SIZE; ++k)
                    Ans.M[i][j] = (Ans.M[i][j] + A.M[i][k] * B.M[k][j]) % mod;
        return Ans;
    }
    Matrix friend operator+(const Matrix &A, const Matrix &B) {
        Matrix Ans;
        Ans.clear();
        for (int i = 1; i <= SIZE; ++i)
```

```
            for (int j = 1; j <= SIZE; ++j)
                Ans.M[i][j] = (A.M[i][j] + B.M[i][j]) % mod;
        return Ans;
    }
};

inline int mypow(LL n, LL k, int p = MOD) {
    LL r = 1;
    for (; k; k >>= 1, n = n * n % p) {
        if (k & 1) r = r * n % p;
    }
    return r;
}
bool ok = 1;
Matrix getinv(Matrix a) { //矩阵求逆
    int n = SIZE, m = SIZE * 2;
    for (int i = 1; i <= n; i++) a.M[i][i + n] = 1;
    for (int i = 1; i <= n; i++) {
        int pos = i;
        for (int j = i + 1; j <= n; j++)
            if (abs(a.M[j][i]) > abs(a.M[pos][i])) pos = j;
        if (i != pos) swap(a.M[i], a.M[pos]);
        if (!a.M[i][i]) {
            puts("No Solution");
            ok = 0;
        }
        ll inv = q_pow(a.M[i][i], mod - 2);
        for (int j = 1; j <= n; j++)
            if (j != i) {
                ll mul = a.M[j][i] * inv % mod;
                for (int k = i; k <= m; k++)
                    a.M[j][k] = ((a.M[j][k] - a.M[i][k] * mul) % mod + mod) % mod;
            }
        for (int j = 1; j <= m; j++) a.M[i][j] = a.M[i][j] * inv % mod;
    }
    Matrix res;
    res.clear();
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            res.M[i][j] = a.M[i][n + j];
    return res;
}
```

矩阵快速幂

以 $\mathcal{O}(N^3 \log M)$ 的复杂度计算。

```
const int N = 110, mod = 1e9 + 7;
LL n, k, a[N][N], b[N][N], t[N][N];
void matrixQp(LL y){
    while (y){
        if (y & 1){
            memset(t, 0, sizeof t);
            for (int i = 1; i <= n; i ++ )
```

```
                for (int j = 1; j <= n; j ++ )
                    for (int k = 1; k <= n; k ++ )
                        t[i][j] = ( t[i][j] + (a[i][k] * b[k][j]) % mod ) % mod;
            memcpy(b, t, sizeof t);
        }
        y >>= 1;
        memset(t, 0, sizeof t);
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                for (int k = 1; k <= n; k ++ )
                    t[i][j] = ( t[i][j] + (a[i][k] * a[k][j]) % mod ) % mod;
        memcpy(a, t, sizeof t);
    }
}
int main(){
    cin >> n >> k;
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ ){
            cin >> b[i][j];
            a[i][j] = b[i][j];
        }
    matrixQp(k - 1);
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            cout << b[i][j] << " \n"[j == n];
    return 0;
}
```

矩阵加速

```
const int mod = 1e9 + 7;
LL T, n, t[5][5], a[5][5], b[5][5];
void matrixQp(LL y){
    while (y){
        if (y & 1){
            memset(t, 0, sizeof t);
            for (int i = 1; i <= 3; i ++ )
                for (int j = 1; j <= 1; j ++ )
                    for (int k = 1; k <= 3; k ++ )
                        t[i][j] = ( t[i][j] + (a[i][k] * b[k][j]) % mod ) % mod;
            memcpy(b, t, sizeof t);
        }
        y >>= 1;
        memset(t, 0, sizeof t);
        for (int i = 1; i <= 3; i ++ )
            for (int j = 1; j <= 3; j ++ )
                for (int k = 1; k <= 3; k ++ )
                    t[i][j] = ( t[i][j] + (a[i][k] * a[k][j]) % mod ) % mod;
        memcpy(a, t, sizeof t);
    }
}
void init(){
    b[1][1] = b[2][1] = b[3][1] = 1;
```

```
    memset(a, 0, sizeof a);
    a[1][1] = a[2][1] = a[1][3] = a[3][2] = 1;
}
void solve(){
    cin >> n;
    if (n <= 3) cout << "1\n";
    else{
        init();
        matrixQp(n - 3);
        cout << b[1][1] << "\n";
    }
}
int main(){
    cin >> T;
    while ( T -- )
        solve();
    return 0;
}

```

整除（数论）分块

$\left\lfloor\frac{n}{l}\right\rfloor=\left\lfloor\frac{n}{l+1}\right\rfloor=\ldots=\left\lfloor\frac{n}{r}\right\rfloor \Longleftrightarrow\left\lfloor\frac{n}{l}\right\rfloor \leq \frac{n}{r}<\left\lfloor\frac{n}{l}\right\rfloor+1$，根据不等式左侧，得到 $r \leq\left\lfloor\frac{n}{\left\lfloor\frac{n}{l}\right\rfloor}\right\rfloor$。

```
void solve() {
    LL n; cin >> n;
    LL ans = 0;
    for (LL i = 1, j; i <= n; i = j + 1) {
        j = n / (n / i);
        ans += (LL)(j - i + 1) * (n / i);
    }
    cout << ans << "\n";
}
int main() {
    int T; cin >> T;
    while (T--) solve();
}
```

常见结论

球盒模型

参考链接。给定 n 个小球 m 个盒子。

- 球同，盒不同、不能空

> 隔板法： N 个小球即一共 $N-1$ 个空，分成 M 堆即 $M-1$ 个隔板，答案为 $\binom{n-1}{m-1}$
>
> 。

- 球同，盒不同、能空

> 隔板法：多出 $M-1$ 个虚空球，答案为 $\binom{m-1+n}{n}$。

- 球同，盒同、能空

> $\frac{1}{(1-x)(1-x^2)\dots(1-x^m)}$ 的 x^n 项的系数。动态规划，答案为
>
> $$dp[i][j]=\begin{cases}dp[i][j-1]+dp[i-j][j] & i\ge j\\ dp[i][j-1] & i<j\\ 1 & j==1 \,||\, i\le 1\end{cases}$$

- 球同，盒同、不能空

> $\frac{x^m}{(1-x)(1-x^2)\dots(1-x^m)}$ 的 x^n 项的系数。动态规划，答案为
>
> $$dp[n][m]=\begin{cases}dp[n-m][m] & n\ge m\\ 0 & n<m\end{cases}$$

- 球不同，盒同、不能空

> 第二类斯特林数 $\texttt{Stirling2}(n,m)$ ，答案为
>
> $$dp[n][m]=\begin{cases}m*dp[n-1][m]+dp[n-1][m-1] & 1\le m<n\\ 1 & 0\le n==m\\ 0 & m==0\text{且}1\le n\end{cases}$$

- 球不同，盒同、能空

> 第二类斯特林数之和 $\sum_{i=1}^{m}\texttt{Stirling2}(n,m)$ ，答案为 $\sum_{i=0}^{m}dp[n][i]$ 。

- 球不同，盒不同、不能空

> 第二类斯特林数乘上 m 的阶乘 $m!\cdot\texttt{Stirling2}(n,m)$ ，答案为 $dp[n][m]*m!$ 。

- 球不同，盒不同、能空

> 答案为 m^n 。

```cpp
i64 mypow(i64 n, i64 k) { // 复杂度是 log N
    i64 r = 1;
    for (; k; k >>= 1, n *= n) {
        if (k & 1) r *= n;
    }
    return r;
}

vector<vector<i64>> comb;
void YangHuiTriangle(int n = 60) {
    comb.resize(n + 1, vector<i64>(n + 1));
    comb[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        comb[i][0] = 1;
        for (int j = 1; j <= n; j++) {
            comb[i][j] = comb[i - 1][j] + comb[i - 1][j - 1];
        }
    }
}
```

```
vector<vector<i64>> S;
void Stirling2(int n = 15) {
    S.resize(n + 1, vector<i64>(n + 1));
    S[1][1] = 1;
    for (int i = 2; i <= 15; i++) {
        for (int j = 1; j <= i; j++) {
            S[i][j] = S[i - 1][j - 1] + S[i - 1][j] * j;
        }
    }
}

vector<vector<i64>> dp;
void GeneratingFunction(int n = 15) {
    dp.resize(n + 1, vector<i64>(n + 1));
    for (int i = 0; i <= n; i++) {
        dp[i][1] = 1;
        for (int j = 2; j <= n; j++) {
            dp[i][j] = dp[i][j - 1];
            if (i >= j) dp[i][j] += dp[i - j][j];
        }
    }
}

vector<i64> fac;
void Fac(int n = 30) {
    fac.resize(n + 1);
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = fac[i - 1] * i;
    }
}

i64 A(int n, int m) {
    if (n < 0 || m < 0 || n < m) return 0;
    return fac[n] / fac[n - m];
}

i64 C(int n, int m) {
    if (n < 0 || m < 0 || n < m) return 0;
    return comb[n][m];
}

signed main() {
    int Task = 1;
    for (cin >> Task; Task; Task--) {
        int op, n, m;
        cin >> op >> n >> m;

        i64 ans = -1;
        if (op == 1) { // 球同，盒同、能空
            ans = dp[n][m];
        } else if (op == 2) { // 球同，盒同、至多放一个
            ans = (n <= m);
        } else if (op == 3) { // 球同，盒同、至少放一个
```

```
                ans = (n < m ? 0 : dp[n - m][m]);
            } else if (op == 4) { // 球同，盒不同、能空
                ans = C(m - 1 + n, n);
            } else if (op == 5) { // 球同，盒不同、至多放一个
                ans = C(m, n);
            } else if (op == 6) { // 球同，盒不同、至少放一个
                ans = C(n - 1, m - 1);
            } else if (op == 7) { // 球不同，盒同、能空
                ans = accumulate(S[n].begin() + 1, S[n].begin() + m + 1, 0LL);
            } else if (op == 8) { // 球不同，盒同、至多放一个
                ans = (n <= m);
            } else if (op == 9) { // 球不同，盒同、至少放一个
                ans = S[n][m];
            } else if (op == 10) { // 球不同，盒不同、能空
                ans = mypow(m, n);
            } else if (op == 11) { // 球不同，盒不同、至多放一个
                ans = A(m, n);
            } else if (op == 12) { // 球不同，盒不同、至少放一个
                ans = fac[m] * S[n][m];
            }
            cout << ans << "\n";
        }
}
```

麦乐鸡定理

给定两个互质的数 n, m，定义 $x = a * n + b * m$（$a \geq 0, b \geq 0$），当 $x > n * m - n - m$ 时，该式子恒成立。

抽屉原理（鸽巢原理）

将 $n+1$ 个物体，划分为 n 组，那么有至少一组有两个（或以上）的物体。

哥德巴赫猜想

任何一个大于 5 的整数都可写成三个质数之和；任何一个大于 2 的偶数都可写成两个素数之和。

除法、取模运算的本质

有公式：$x \nabla \cdot i = \left\lfloor \frac{x}{i} \right\rfloor + x - i \cdot \left\lfloor \frac{x}{i} \right\rfloor$，$x \mod i = x - i \cdot \left\lfloor \frac{x}{i} \right\rfloor$。

与、或、异或

运算	运算符、数学符号表示	解释
与	`&`、`and`	同1出1
或	`|`、`or`	有1出1
异或	`^`、$\bigoplus$、`xor`	不同出1

一些结论：

> 对于给定的 X 和序列 $[a_1, a_2, \dots, a_n]$ ，有：$\boldsymbol{X = (X\&a_1)or(X\&a_2)or\dots or(X\&a_n)}$ 。
> 原理是 and 意味着取交集，or 意味着取子集。[来源 - 牛客小白月赛49C]

调和级数近似公式

```
log(n) + 0.5772156649 + 1.0 / (2 * n)
```

欧拉函数常见性质

- $1-n$ 中与 n 互质的数之和为 $n*\varphi(n)/2$ 。
- 若 a，b 互质，则 $\varphi(a*b)=\varphi(a)*\varphi(b)$ 。实际上，所有满足这一条件的函数统称为积性函数。
- 若 f 是积性函数，且有 $n=\prod\limits_{i=1}^{m}p_i^{c_i}$ ，那么 $f(n)=\prod\limits_{i=1}^{m}f(p_i^{c_i})$ 。
- 若 p 为质数，且满足 $p\mid n$ ，
    - $p^2\mid n$ ，那么 $\varphi(n)=\varphi(n/p)*p$ 。
    - $p^2\nmid n$，那么 $\varphi(n)=\varphi(n/p)*(p-1)$ 。
- $\sum\limits_{d|n}\varphi(d)=n$ 。

> 如 $n=10$ ，则 $d=10/5/2/1$ ，那么 $10=\varphi(10)+\varphi(5)+\varphi(2)+\varphi(1)$ 。

- $\sum\limits_{i=1}^{n}\gcd(i,n)=\sum\limits_{d|n}\left\lfloor\frac{n}{d}\right\rfloor\varphi(d)$ （欧拉反演）。

组合数学常见性质

- $k*C_n^k=n*C_{n-1}^{k-1}$ ；
- $C_k^n*C_m^k=C_m^n*C_{m-n}^{m-k}$ ；
- $C_n^k+C_n^{k+1}=C_{n+1}^{k+1}$ ；
- $\sum_{i=0}^{n}C_n^i=2^n$ ；
- $\sum_{k=0}^{n}(-1)^k*C_n^k=0$ 。
- 二项式反演：$\begin{cases} f_n=\sum\limits_{i=0}^{n}\binom{n}{i}g_i\Leftrightarrow g_n=\sum\limits_{i=0}^{n}(-1)^{n-i}\binom{n}{i}f_i \\ f_k=\sum\limits_{i=k}^{n}\binom{i}{k}g_i\Leftrightarrow g_k=\sum\limits_{i=k}^{n}(-1)^{i-k}\binom{i}{k}f_i\end{cases}$ ；
- $\sum\limits_{i=1}^{n}i\binom{n}{i}=n*2^{n-1}$ ；
- $\sum\limits_{i=1}^{n}i^2\binom{n}{i}=n*(n+1)*2^{n-2}$ ；
- $\sum\limits_{i=1}^{n}\frac{1}{i}\binom{n}{i}=\sum\limits_{i=1}^{n}\frac{1}{i}$ ；
- $\sum\limits_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$ ；
- 拉格朗日恒等式：$\sum\limits_{i=1}^{n}\sum\limits_{j=i+1}^{n}(a_ib_j-a_jb_i)^2=(\sum\limits_{i=1}^{n}a_i)^2(\sum\limits_{i=1}^{n}b_i)^2-(\sum\limits_{i=1}^{n}a_ib_i)^2$ 。

范德蒙德卷积公式

在数量为 $n+m$ 的堆中选 k 个元素，和分别在数量为 n、m 的堆中选 i、$k-i$ 个元素的方案数是相同的，即$\sum_{i=0}^{k}\binom{n}{i}\binom{m}{k-i}=\binom{n+m}{k}$；

变体：

- $\sum_{i=0}^{k} C_{i+n}^{i}=C_{k+n+1}^{k}$；
- $\sum_{i=0}^{k} C_{n}^{i} * C_{m}^{i}=\sum_{i=0}^{k} C_{n}^{i} * C_{m}^{m-i}=C_{n+m}^{n}$。

卡特兰数

是一类奇特的组合数，前几项为 $1,1,2,5,14,42,132,429,1430,4862$。如遇到以下问题，则直接套用即可。

- 【括号匹配问题】n 个左括号和 n 个右括号组成的合法括号序列的数量，为 Cat_n。
- 【进出栈问题】$1,2,\ldots,n$ 经过一个栈，形成的合法出栈序列的数量，为 Cat_n。
- 【二叉树生成问题】n 个节点构成的不同二叉树的数量，为 Cat_n。
- 【路径数量问题】在平面直角坐标系上，每一步只能**向上**或**向右**走，从 $(0,0)$ 走到 (n,n)，并且除两个端点外不接触直线 $y=x$ 的路线数量，为 $2Cat_{n-1}$。

计算公式：$Cat_n=\frac{C_{2n}^{n}}{n+1}$，$C_n=\frac{C_{n-1}*(4n-2)}{n+1}$。

狄利克雷卷积

$\sum_{d|n}\varphi(d)=n$，$\sum_{d|n}\mu(d)\frac{n}{d}=\varphi(n)$。

斐波那契数列

通项公式：$F_n=\frac{1}{\sqrt{5}}*\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$。

直接结论：

- 卡西尼性质：$F_{n-1}*F_{n+1}-F_n^2=(-1)^n$；
- $F_n^2+F_{n+1}^2=F_{2n+1}$；
- $F_{n+1}^2-F_{n-1}^2=F_{2n}$（由上一条写两遍相减得到）；
- 若存在序列 $a_0=1,a_n=a_{n-1}+a_{n-3}+a_{n-5}+\ldots(n\geq 1)$ 则 $a_n=F_n(n\geq 1)$；
- 齐肯多夫定理：任何正整数都可以表示成若干个不连续的斐波那契数（F_2 开始）可以用贪心实现。

求和公式结论：

- 奇数项求和：$F_1+F_3+F_5+\ldots+F_{2n-1}=F_{2n}$；
- 偶数项求和：$F_2+F_4+F_6+\ldots+F_{2n}=F_{2n+1}-1$；
- 平方和：$F_1^2+F_2^2+F_3^2+\ldots+F_n^2=F_n*F_{n+1}$；
- $F_1+2F_2+3F_3+\ldots+nF_n=nF_{n+2}-F_{n+3}+2$；
- $-F_1+F_2-F_3+\ldots+(-1)^nF_n=(-1)^n(F_{n+1}-F_n)+1$；
- $F_{2n-2m-2}(F_{2n}+F_{2n+2})=F_{2m+2}+F_{4n-2m}$。

数论结论：

- $F_a \mid F_b \Leftrightarrow a \mid b$；
- $\gcd(F_a,F_b)=F_{\gcd(a,b)}$；

- 当 p 为 $5k \pm 1$ 型素数时，$\begin{cases} F_{p-1} \equiv 0 \pmod p \\ F_p \equiv 1 \pmod p \\ F_{p+1} \equiv 1 \pmod p \end{cases}$；
- 当 p 为 $5k \pm 2$ 型素数时，$\begin{cases} F_{p-1} \equiv 1 \pmod p \\ F_p \equiv -1 \pmod p \\ F_{p+1} \equiv 0 \pmod p \end{cases}$；
- $F(n)\%m$ 的周期 $\le 6m$ （$m = 2 \times 5^k$ 时取到等号）；
- 既是斐波那契数又是平方数的有且仅有 $1, 144$ 。

杂

- 负数取模得到的是负数，如果要用 $0/1$ 判断的话请取绝对值；
- 辗转相除法原式为 $\gcd(x, y) = \gcd(x, y - x)$ ，推广到 N 项为 $\gcd(a_1, a_2, \ldots, a_N) = \gcd(a_1, a_2 - a_1, \ldots, a_N - a_{N-1})$ ，
    - 该推论在"四则运算后 gcd "这类题中有特殊意义，如求解 $\gcd(a_1 + X, a_2 + X, \ldots, a_N + X)$ 时See；
- 以下式子成立： $\gcd(a, m) = \gcd(a + x, m) \Leftrightarrow \gcd(a, m) = \gcd(x, m)$ 。求解上式满足条件的 x 的数量即为求比 $\dfrac{m}{\gcd(a, m)}$ 小且与其互质的数的个数，即用欧拉函数求解 $\varphi\left(\dfrac{m}{\gcd(a, m)}\right)$ 。
- 已知序列 a ，定义集合 $S = \{a_i \cdot a_j \mid i < j\}$ ，现在要求解 $\gcd(S)$ ，即为求解 $\gcd(a_j, \gcd(a_i \mid i < j))$ ，换句话说，即为求解后缀 gcd 。
- 连续四个数互质的情况如下，当 n 为奇数时，$n, n-1, n-2$ 一定互质；而当 n 为偶数时，$\begin{cases} n, n-1, n-3\text{互质} & \gcd(n, n-3) = 1\text{时} \\ n-1, n-2, n-3\text{互质} & \gcd(n, n-3) \neq 1\text{时} \end{cases}$ See；
- 由 $a \mod b = (b + a) \mod b = (2 \cdot b + a) \mod b = \cdots = (K \cdot b + a) \mod b$ 可以推广得到 $(a \mod b) \mod c = ((K \cdot bc + a) \mod b) \mod c$ ，由此可以得到一个 bc 的答案周期See；
- 对于长度为 $2 \cdot N$ 的数列 a ，将其任意均分为两个长度为 N 的数列 p, q ，随后对 p 非递减排序、对 q 非递增排序，定义 $f(p, q) = \sum_{i=1}^{n} |p_i - q_i|$ ，那么答案为 a 数列前 N 大的数之和减去前 N 小的数之和See。
- 令 $\begin{cases} X = a + b \\ Y = a \oplus b \end{cases}$ ，**如果**该式子**有解**，那么存在前提条件 $\begin{cases} X \ge Y \\ X, Y\text{同奇偶} \end{cases}$ ；进一步，此时最小的 a 的取值为 $\dfrac{X - Y}{2}$ See。

  然而，上方方程并不总是有解的，只有当变量增加到三个时，才**一定有解**，即：**在保证上方前提条件成立的情况下**，求解 $\begin{cases} X = a + b + c \\ Y = a \oplus b \oplus c \end{cases}$ ，则一定存在一组解 $\{\dfrac{X - Y}{2}, \dfrac{X - Y}{2}, Y\}$ See。
- 已知序列 p 是由序列 a_1 、序列 a_2 、……、序列 a_n 合并而成，且合并过程中各序列内元素相对顺序不变，记 $T(p)$ 是 p 序列的最大前缀和，则 $T(p) = \sum_{i=1}^{n} T(a_i)$ See 。
- $x + y = x|y + x\&y$ ，对于两个数字 x 和 y ，如果将 x 变为 $x|y$ ，同时将 y 变为 $x\&y$ ，那么在本质上即将 x 二进制模式下的全部 1 移动到了 y 的对应的位置上 See 。
- 一个正整数 x 异或、加上另一个正整数 y 后奇偶性不发生变化：$a + b \equiv a \oplus b \pmod 2$ See 。

常见例题

题意：将 1 至 N 的每个数字分组，使得每一组的数字之和均为质数。输出每一个数字所在的组别，且要求分出的组数最少 See 。

考察哥德巴赫猜想，记全部数字之和为 S ，分类讨论如下：

- 为 S 质数时，只需要分入同一组；
- 当 S 为偶数时，由猜想可知一定能分成两个质数，可以证明其中较小的那个一定小于 N ，暴力枚举分组；
- 当 $S-2$ 为质数时，特殊判断出答案；
- 其余情况一定能被分成三组，其中 3 单独成组，$S-3$ 后成为偶数，重复讨论二的过程即可。

题意：给定一个长度为 n 的数组，定义这个数组是 BAD 的，当且仅当可以把数组分成两个子序列，这两个子序列的元素之和相等。现在你需要删除**最少的**元素，使得删除后的数组不是 BAD 的。

最少删除一个元素——如果原数组存在奇数，则直接删除这个奇数即可；反之，我们发现，对数列同除以一个数不影响计算，故我们只需要找到最大的满足 $2^k \mid a_i$ 成立的 2^k ，随后将全部的 a_i 变为 $\dfrac{a_i}{2^k}$ ，此时一定有一个奇数（换句话说，我们可以对原数列的每一个元素不断的除以 2 直到出现奇数为止），删除这个奇数即可 See 。

题意：设当前有一个数字为 x ，减去、加上最少的数字使得其能被 k 整除。

最少减去 $x \bmod k$ 这个很好想；最少加上 $\left(\left\lceil \dfrac{x}{k} \right\rceil * k\right) \bmod k$ 也比较好想，但是更简便的方法为加上 $k - x \bmod k$ ，这个式子等价于前面这一坨。

题意：给定一个整数 n ，用恰好 k 个 2 的幂次数之和表示它。例如：$n=9, k=4$ ，答案为 $1+2+2+4$ 。

结论1：k 合法当且仅当 `__builtin_popcountll(n) <= k && k <= n` ，显然。

结论2：$2^{k+1} = 2 \cdot 2^k$ ，所以我们可以将二进制位看作是数组，然后从高位向低位推，一个高位等于两个低位，直到数组之和恰好等于 k ，随后依次输出即可。举例说明，$\{1,0,0,1\} \rightarrow \{0,2,0,1\} \rightarrow \{0,1,2,1\}$ ，即答案为 0 个 2^3 、1 个 2^2 、……。

```
signed main() {
    int n, k;
    cin >> n >> k;

    int cnt = __builtin_popcountll(n);

    if (k < cnt || n < k) {
        cout << "NO\n";
        return 0;
    }
    cout << "YES\n";

    vector<int> num;
    while (n) {
        num.push_back(n % 2);
        n /= 2;
    }
```

```

    for (int i = num.size() - 1; i > 0; i--) {
        int p = min(k - cnt, num[i]);
        num[i] -= p;
        num[i - 1] += 2 * p;
        cnt += p;
    }

    for (int i = 0; i < num.size(); i++) {
        for (int j = 1; j <= num[i]; j++) {
            cout << (1LL << i) << " ";
        }
    }
}
```

题意：n 个取值在 $[0,k)$ 之间的数之和为 m 的方案数

答案为 $\sum_{i=0}^{n} -1^{i} \cdot \binom{n}{i} \cdot \binom{m-i \cdot k+n-1}{n-1}$ See1 See2。

```
Z clac(int n, int k, int m) {
    Z ans = 0;
        ans += C(n, i) * C(m - i * k + n - 1, n - 1) * pow(-1, i);
    }
    return ans;
}
```

[1] 先考虑没有 k 的限制，那么即球盒模型：m 个球放入 n 个盒子，球同、盒子不同、能空。使用隔板法得到公式：`C(m + n - 1, n - 1)`；[2] 下面加上取值范围后进一步考虑：假设现在 n 个数之和为 $m-k$，运用上述隔板法可得公式：`C(m - k + n - 1, n - 1)`；[3] 随后，选择任意一个数字，将其加上 k，这样，这个数字一定不满足条件，选法为：`C(n, 1)`；[4] 此时，至少有一个数字是不满足条件的，按照一般流程，到这里，`C(m + n - 1, n - 1) - C(n, 1) * C(m - k + n - 1, n - 1)` 即是答案；但是，这样的操作会导致重复的部分，所以这里要使用容斥原理将重复部分去除（关于为什么会重复，试比较概率论中的加法公式）。

/END/

数据结构

并查集（全功能）

```
struct DSU {
    vector<int> fa, p, e, f;

    DSU(int n) {
        fa.resize(n + 1);
        iota(fa.begin(), fa.end(), 0);
        p.resize(n + 1, 1);
        e.resize(n + 1);
        f.resize(n + 1);
    }
    int get(int x) {
        while (x != fa[x]) {
            x = fa[x] = fa[fa[x]];
        }
        return x;
    }
    bool merge(int x, int y) { // 设x是y的祖先
        if (x == y) f[get(x)] = 1;
        x = get(x), y = get(y);
        e[x]++;
        if (x == y) return false;
        if (x < y) swap(x, y); // 将编号小的合并到大的上
        fa[y] = x;
        f[x] |= f[y], p[x] += p[y], e[x] += e[y];
        return true;
    }
    bool same(int x, int y) {
        return get(x) == get(y);
    }
    bool F(int x) { // 判断连通块内是否存在自环
        return f[get(x)];
    }
    int size(int x) { // 输出连通块中点的数量
        return p[get(x)];
    }
    int E(int x) { // 输出连通块中边的数量
        return e[get(x)];
    }
};
```

线段树（需要处理区间加法与乘法修改）

```
#include<bits/stdc++.h>
using namespace std;
#define fir first
#define sec second
#define endl "\n"
typedef long long ll;
typedef unsigned long long ull;
```

```
typedef pair<int,int> pii;
typedef pair<ll, ll> pll;
const int mod = 1e9 + 7, inf = 0x3f3f3f3f, P = 131,maxn=1e5+5;
double v[maxn];
int n,m;
struct note
{
    double sum,sum_2,tag;
    note operator+ (const note &q)
    {
        note a;
        a.sum_2=sum_2+q.sum_2;
        a.sum=sum+q.sum;
        a.tag=0;
        return a;
    }
}tree[maxn*4];
int ls(int x)
{
    return x*2;
}
int rs(int x)
{
    return x*2+1;
}
void push_up(int x)
{
    tree[x]=tree[x*2]+tree[x*2+1];
    return;
}
void add_note(int root,int l,int r,double x)
{
    tree[root].sum_2=tree[root].sum_2+(r-l+1)*(x*x)+2*x*tree[root].sum;
    tree[root].sum=tree[root].sum+(r-l+1)*x;
    return;
}
void push_down(int root,int l,int r)
{
    if(tree[root].tag==0)
        return;
    int mid=(r+l)/2;
    double x=tree[root].tag;
    add_note(ls(root),l,mid,x);
    add_note(rs(root),mid+1,r,x);
    tree[ls(root)].tag+=x;
    tree[rs(root)].tag+=x;
    tree[root].tag=0;
}
void build(int root,int l,int r)
{
    if(l==r)
    {
        tree[root].sum=v[l];
        tree[root].sum_2=pow(v[l],2);
        tree[root].tag=0;
```

```
    }
    else
    {
        int mid=l+((r-l)>>1);
        build(root<<1,l,mid);
        build((root<<1)+1,mid+1,r);
        push_up(root);
    }
    return;
}
void add(int root,int l1,int r1,int l,int r,double x)//根，根包含的，包含的，要改的，+1，改变数值
{
    if(l1==l&&r1==r)
    {
        add_note(root,l,r,x);
        tree[root].tag+=x;
        return;
    }
    push_down(root,l1,r1);
    int mid=(l1+r1)/2;
    if(r<=mid)
    {
        add(root*2,l1,mid,l,r,x);
    }
    else
    {
        if(l>mid)
            add(root*2+1,mid+1,r1,l,r,x);
        else
        {
            add(root*2,l1,mid,l,mid,x);
            add((root*2)+1,mid+1,r1,mid+1,r,x);
        }
    }
    push_up(root);
    return;
}
double find_sum(int root, int l, int r, int l1, int r1)
{
    if (l == l1 && r == r1)
        return tree[root].sum;
    push_down(root, l1, r1);
    int mid = (l1 + r1) / 2;
    if (r <= mid)
        return find_sum(ls(root), l, r, l1, mid);
    else if (l > mid)
        return find_sum(rs(root), l, r, mid + 1, r1);
    else
        return find_sum(ls(root), l, mid, l1, mid) + find_sum(rs(root), mid + 1, r, mid + 1, r1);
}
double find_2(int root, int l, int r, int l1, int r1)
{
    if (l == l1 && r == r1)
```

```cpp
        return tree[root].sum_2;
    push_down(root, l1, r1);
    int mid = (l1 + r1) / 2;
    if (r <= mid)
        return find_2(ls(root), l, r, l1, mid);
    else if (l > mid)
        return find_2(rs(root), l, r, mid + 1, r1);
    else
        return find_2(ls(root), l, mid, l1, mid) + find_2(rs(root), mid + 1, r, mid + 1, r1);
}
void solve()
{
    cin>>n>>m;
    for (int i = 1; i <= n; i++)
        cin>>v[i];
    build(1,1,n);
    for (int i = 0; i < m; i++)
    {
        double x,y,k;
        int op;
        cin>>op;
        switch (op)
        {
        case 1:
            cin>>x>>y>>k;
            add(1,1,n,x,y,k);
            break;
        case 2:
            cin>>x>>y;
            printf("%.4f\n",find_sum(1,x,y,1,n)/(y-x+1));
            break;
        case 3:
            cin>>x>>y;
            double xb=find_sum(1,x,y,1,n)/(y-x+1),xf=find_2(1,x,y,1,n);
            double end=xf/(y-x+1)-pow(xb,2);
            printf("%.4f\n",end);
            break;
        }
    }


}
int main()
{
    solve();
    return 0;
}
```

jiangly

```cpp
template <class T> struct Segt_ {
    struct node {
        int l, r;
        T w, add, mul = 1; // 注意初始赋值
```

```cpp
    };
    vector<T> w;
    vector<node> t;

    Segt_(int n) {
        w.resize(n + 1);
        t.resize((n << 2) + 1);
        build(1, n);
    }
    Segt_(vector<int> in) {
        int n = in.size() - 1;
        w.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            w[i] = in[i];
        }
        t.resize((n << 2) + 1);
        build(1, n);
    }
    void pushdown(node &p, T add, T mul) { // 在此更新下递函数
        p.w = p.w * mul + (p.r - p.l + 1) * add;
        p.add = p.add * mul + add;
        p.mul *= mul;
    }
    void pushup(node &p, node &l, node &r) { // 在此更新上传函数
        p.w = l.w + r.w;
    }
#define GL (k << 1)
#define GR (k << 1 | 1)
    void pushdown(int k) { // 不需要动
        pushdown(t[GL], t[k].add, t[k].mul);
        pushdown(t[GR], t[k].add, t[k].mul);
        t[k].add = 0, t[k].mul = 1;
    }
    void pushup(int k) { // 不需要动
        pushup(t[k], t[GL], t[GR]);
    }
    void build(int l, int r, int k = 1) {
        if (l == r) {
            t[k] = {l, r, w[l]};
            return;
        }
        t[k] = {l, r};
        int mid = (l + r) / 2;
        build(l, mid, GL);
        build(mid + 1, r, GR);
        pushup(k);
    }
    void modify(int l, int r, T val, int k = 1) { // 区间修改
        if (l <= t[k].l && t[k].r <= r) {
            t[k].w += (t[k].r - t[k].l + 1) * val;
            t[k].add += val;
            return;
        }
        pushdown(k);
        int mid = (t[k].l + t[k].r) / 2;
```

```
            if (l <= mid) modify(l, r, val, GL);
            if (mid < r) modify(l, r, val, GR);
            pushup(k);
        }
        void modify2(int l, int r, T val, int k = 1) { // 区间修改
            if (l <= t[k].l && t[k].r <= r) {
                t[k].w *= val;
                t[k].add *= val;
                t[k].mul *= val;
                return;
            }
            pushdown(k);
            int mid = (t[k].l + t[k].r) / 2;
            if (l <= mid) modify2(l, r, val, GL);
            if (mid < r) modify2(l, r, val, GR);
            pushup(k);
        }
        T ask(int l, int r, int k = 1) { // 区间询问，不合并
            if (l <= t[k].l && t[k].r <= r) {
                return t[k].w;
            }
            pushdown(k);
            int mid = (t[k].l + t[k].r) / 2;
            T ans = 0;
            if (l <= mid) ans += ask(l, r, GL);
            if (mid < r) ans += ask(l, r, GR);
            return ans;
        }
        void debug(int k = 1) {
            cout << "[" << t[k].l << ", " << t[k].r << "]: ";
            cout << "w = " << t[k].w << ", ";
            cout << "add = " << t[k].add << ", ";
            cout << "mul = " << t[k].mul << ", ";
            cout << endl;
            if (t[k].l == t[k].r) return;
            debug(GL), debug(GR);
        }
    };
```

普通莫队

以 $\mathcal{O}(N\sqrt{N})$ 的复杂度完成 Q 次询问的离线查询，其中每个分块的大小取 $\sqrt{N}=\sqrt{10^5}=317$ ，也可以使用 `n / min<int>(n, sqrt(q))` 、 `ceil((double)n / (int)sqrt(n))` 或者 `sqrt(n)` 划分。

```
signed main() {
    int n;
    cin >> n;
    vector<int> w(n + 1);
    for (int i = 1; i <= n; i++) {
        cin >> w[i];
    }

    int q;
```

```cpp
    cin >> q;
    vector<array<int, 3>> query(q + 1);
    for (int i = 1; i <= q; i++) {
        int l, r;
        cin >> l >> r;
        query[i] = {l, r, i};
    }

    int Knum = n / min<int>(n, sqrt(q)); // 计算块长
    vector<int> K(n + 1);
    for (int i = 1; i <= n; i++) { // 固定块长
        K[i] = (i - 1) / Knum + 1;
    }
    sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
        if (K[x[0]] != K[y[0]]) return x[0] < y[0];
        if (K[x[0]] & 1) return x[1] < y[1];
        return x[1] > y[1];
    });

    int l = 1, r = 0, val = 0;
    vector<int> ans(q + 1);
    for (int i = 1; i <= q; i++) {
        auto [ql, qr, id] = query[i];
        auto add = [&](int x) -> void {};
        auto del = [&](int x) -> void {};
        while (l > ql) add(w[--l]);
        while (r < qr) add(w[++r]);
        while (l < ql) del(w[l++]);
        while (r > qr) del(w[r--]);
        ans[id] = val;
    }
    for (int i = 1; i <= q; i++) {
        cout << ans[i] << endl;
    }
}
```

需要注意的是，在普通莫队中，`K` 数组的作用是根据左边界的值进行排序，当询问次数很少时（$q \ll n$），可以直接合并到 `query` 数组中。

```cpp
vector<array<int, 4>> query(q);
for (int i = 1; i <= q; i++) {
    int l, r;
    cin >> l >> r;
    query[i] = {l, r, i, (l - 1) / Knum + 1}; // 合并
}
sort(query.begin() + 1, query.end(), [&](auto x, auto y) {
    if (x[3] != y[3]) return x[3] < y[3];
    if (x[3] & 1) return x[1] < y[1];
    return x[1] > y[1];
});
```

分数运算类

定义了分数的四则运算，如果需要处理浮点数，那么需要将函数中的 `gcd` 运算替换为 `fgcd` 。

```
template<class T> struct Frac {
    T x, y;
    Frac() : Frac(0, 1) {}
    Frac(T x_) : Frac(x_, 1) {}
    Frac(T x_, T y_) : x(x_), y(y_) {
        if (y < 0) {
            y = -y;
            x = -x;
        }
    }

    constexpr double val() const {
        return 1. * x / y;
    }
    constexpr Frac norm() const { // 调整符号、转化为最简形式
        T p = gcd(x, y);
        return {x / p, y / p};
    }
    friend constexpr auto &operator<<(ostream &o, const Frac &j) {
        T p = gcd(j.x, j.y);
        if (j.y == p) {
            return o << j.x / p;
        } else {
            return o << j.x / p << "/" << j.y / p;
        }
    }
    constexpr Frac &operator/=(const Frac &i) {
        x *= i.y;
        y *= i.x;
        if (y < 0) {
            x = -x;
            y = -y;
        }
        return *this;
    }
    constexpr Frac &operator+=(const Frac &i) { return x = x * i.y + y * i.x, y *= i.y, *this; }
    constexpr Frac &operator-=(const Frac &i) { return x = x * i.y - y * i.x, y *= i.y, *this; }
    constexpr Frac &operator*=(const Frac &i) { return x *= i.x, y *= i.y, *this; }
    friend constexpr Frac operator+(const Frac i, const Frac j) { return i += j; }
    friend constexpr Frac operator-(const Frac i, const Frac j) { return i -= j; }
    friend constexpr Frac operator*(const Frac i, const Frac j) { return i *= j; }
    friend constexpr Frac operator/(const Frac i, const Frac j) { return i /= j; }
    friend constexpr Frac operator-(const Frac i) { return Frac(-i.x, i.y); }
```

```
    friend constexpr bool operator<(const Frac i, const Frac j) { return i.x * j.y < i.y * j.x; }
    friend constexpr bool operator>(const Frac i, const Frac j) { return i.x * j.y > i.y * j.x; }
    friend constexpr bool operator==(const Frac i, const Frac j) { return i.x * j.y == i.y * j.x; }
    friend constexpr bool operator!=(const Frac i, const Frac j) { return i.x * j.y != i.y * j.x; }
};
```

主席树（可持久化线段树）

以 $\mathcal{O}(N\log N)$ 的时间复杂度建树、查询、修改。

```
struct PresidentTree {
    static constexpr int N = 2e5 + 10;
    int cntNodes, root[N];
    struct node {
        int l, r;
        int cnt;
    } tr[4 * N + 17 * N];
    void modify(int &u, int v, int l, int r, int x) {
        u = ++cntNodes;
        tr[u] = tr[v];
        tr[u].cnt++;
        if (l == r) return;
        int mid = (l + r) / 2;
        if (x <= mid)
            modify(tr[u].l, tr[v].l, l, mid, x);
        else
            modify(tr[u].r, tr[v].r, mid + 1, r, x);
    }
    int kth(int u, int v, int l, int r, int k) {
        if (l == r) return l;
        int res = tr[tr[v].l].cnt - tr[tr[u].l].cnt;
        int mid = (l + r) / 2;
        if (k <= res)
            return kth(tr[u].l, tr[v].l, l, mid, k);
        else
            return kth(tr[u].r, tr[v].r, mid + 1, r, k - res);
    }
};
```

动态规划

01背包

有 n 件物品和一个容量为 W 的背包，第 i 件物品的体积为 $w[i]$，价值为 $v[i]$，求解将哪些物品装入背包中使总价值最大。

思路：

当放入一个体积为 $w[i]$ 的物品后，价值增加了 $v[i]$，于是我们可以构建一个二维的 $dp[i][j]$ 数组，装入第 i 件物品时，背包容量为 j 能实现的 **最大价值**，可以得到 **转移方程** $dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i])$。

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

我们可以发现，第 i 个物品的状态是由第 $i-1$ 个物品转移过来的，每次的 j 转移过来后，第 $i-1$ 个方程的 j 已经没用了，于是我们想到可以把二维方程压缩成 **一维** 的，用以 **优化空间复杂度。**

```
for (int i = 1; i <= n; i++)  //当前装第 i 件物品
    for (int j = W; j >= w[i]; j--)  //背包容量为 j
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);  //判断背包容量为 j 的情况下能是实现总价值最大是多少
```

完全背包

有 n 件物品和一个容量为 W 的背包，第 i 件物品的体积为 $w[i]$，价值为 $v[i]$，每件物品有**无限个**，求解将哪些物品装入背包中使总价值最大。

思路:

思路和**01背包**差不多，但是每一件物品有**无限个**，其实就是从每 **种** 物品中取 $0, 1, 2, \ldots$ 件物品加入背包中

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j <= W; j++)
        for (int k = 0; k * w[i] <= j; k++)     //选取几个物品
            dp[i][j] = max(dp[i][j], dp[i - 1][j - k * w[i]] + k * v[i]);
```

实际上，我们可以发现，取 k 件物品可以从取 $k-1$ 件转移过来，那么我们就可以将 k 的循环优化掉

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

和 01 背包 类似地压缩成一维

```
for (int i = 1; i <= n; i++)
    for (int j = w[i]; j <= W; j++)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

多重背包

有 n 种物品和一个容量为 W 的背包，第 i 种物品的体积为 $w[i]$，价值为 $v[i]$，数量为 $s[i]$，求解将哪些物品装入背包中使总价值最大。

思路：

对于每一种物品，都有 $s[i]$ 种取法，我们可以将其转化为**01背包**问题

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

上述方法的时间复杂度为 $O(n*m*s)$。

```
for (int i = 1; i <= n; i++){
    scanf("%lld%lld%lld", &x, &y, &s);  //x 为体积， y 为价值， s 为数量
    t = 1;
    while (s >= t){
        w[++num] = x * t;
        v[num] = y * t;
        s -= t;
        t *= 2;
    }
    w[++num] = x * s;
    v[num] = y * s;
}
for (int i = 1; i <= num; i++)
    for (int j = W; j >= w[i]; j--)
        dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
```

尽管采用了 **二进制优化**，时间复杂度还是太高，采用 **单调队列优化**，将时间复杂度优化至 $O(n*m)$

```
#include <bits/stdc++.h>
using namespace std;
const int N = 2e5 + 10;
int n, W, w, v, s, f[N], g[N], q[N];
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> W;
    for (int i = 0; i < n; i ++ ){
        memcpy ( g, f, sizeof f);
        cin >> w >> v >> s;
        for (int j = 0; j < w; j ++ ){
            int head = 0, tail = -1;
            for (int k = j; k <= W; k += w){
                if ( head <= tail && k - s * w > q[head] ) head ++ ;//保证队列长度 <= s
                while ( head <= tail && g[q[tail]] - (q[tail] - j) / w * v <= g[k] - (k - j) / w * v ) tail -- ;//保证队列单调递减
```

```
                q[ ++ tail] = k;
                f[k] = g[q[head]] + (k - q[head]) / w * v;
            }
        }
    }
    cout << f[W] << "\n";
    return 0;
}
```

混合背包

放入背包的物品可能只有 **1** 件（01背包），也可能有**无限**件（完全背包），也可能只有**可数的几件**（多重背包）。

思路：

分类讨论即可，哪一类就用哪种方法去 dp。

```
#include <bits/stdc++.h>
using namespace std;
int n, W, w, v, s;
int main(){
    cin >> n >> W;
    vector <int> f(W + 1);
    for (int i = 0; i < n; i ++ ){
        cin >> w >> v >> s;
        if (s == -1){
            for (int j = W; j >= w; j -- )
                f[j] = max(f[j], f[j - w] + v);
        }
        else if (s == 0){
            for (int j = w; j <= W; j ++ )
                f[j] = max(f[j], f[j - w] + v);
        }
        else {
            int t = 1, cnt = 0;
            vector <int> x(s + 1), y(s + 1);
            while (s >= t){
                x[++cnt] = w * t;
                y[cnt] = v * t;
                s -= t;
                t *= 2;
            }
            x[++cnt] = w * s;
            y[cnt] = v * s;
            for (int i = 1; i <= cnt; i ++ )
                for (int j = W; j >= x[i]; j -- )
                    f[j] = max(f[j], f[j - x[i]] + y[i]);
        }
    }
    cout << f[W] << "\n";
    return 0;
}
```

二维费用的背包

有 n 件物品和一个容量为 W 的背包，背包能承受的最大重量为 M，每件物品只能用一次，第 i 件物品的体积是 $w[i]$，重量为 $m[i]$，价值为 $v[i]$，求解将哪些物品放入背包中使总体积不超过背包容量，总重量不超过背包最大容量，且总价值最大。

思路：

背包的限制条件由一个变成两个，那么我们的循环再多一维即可。

```
for (int i = 1; i <= n; i++)
    for (int j = W; j >= w; j--)  //容量限制
        for (int k = M; k >= m; k--)  //重量限制
            dp[j][k] = max(dp[j][k], dp[j - w][k - m] + v);
```

分组背包

有 n **组**物品，一个容量为 W 的背包，每组物品有若干，同一组的物品最多选一个，第 i 组第 j 件物品的体积为 $w[i][j]$，价值为 $v[i][j]$，求解将哪些物品装入背包，可使物品总体积不超过背包容量，且使总价值最大。

思路：

考虑每**组**中的**某件**物品选不选，可以选的话，去下一组选下一个，否则在这组继续寻找可以选的物品，当这组遍历完后，去下一组寻找。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, s[N], w[N][N], v[N][N], dp[N];
int main(){
    cin >> n >> W;
    for (int i = 1; i <= n; i++){
        scanf("%d", &s[i]);
        for (int j = 1; j <= s[i]; j++)
            scanf("%d %d", &w[i][j], &v[i][j]);
    }
    for (int i = 1; i <= n; i++)
        for (int j = W; j >= 0; j--)
            for (int k = 1; k <= s[i]; k++)
                if (j - w[i][k] >= 0)
                    dp[j] = max(dp[j], dp[j - w[i][k]] + v[i][k]);
    cout << dp[W] << "\n";
    return 0;
}
```

有依赖的背包

有 n 个物品和一个容量为 W 的背包，物品之间有依赖关系，且之间的依赖关系组成一颗 **树** 的形状，如果选择一个物品，则必须选择它的 **父节点**，第 i 件物品的体积是 $w[i]$，价值为 $v[i]$，依赖的父节点的编号为 $p[i]$，若 $p[i]$ 等于 -1，则为 **根节点**。求将哪些物品装入背包中，使总体积不超过总容量，且总价值最大。

思路：

定义 $f[i][j]$ 为以第 i 个节点为根，容量为 j 的背包的最大价值。那么结果就是 $f[root][W]$，为了知道根节点的最大价值，得通过其子节点来更新。所以采用递归的方式。
对于每一个点，先将这个节点装入背包，然后找到剩余容量可以实现的最大价值，最后更新父节点的最大价值即可。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 110;
int n, W, w[N], v[N], p, f[N][N], root;
vector <int> g[N];
void dfs(int u){
    for (int i = w[u]; i <= W; i ++ )
        f[u][i] = v[u];
    for (auto v : g[u]){
        dfs(v);
        for (int j = W; j >= w[u]; j -- )
            for (int k = 0; k <= j - w[u]; k ++ )
                f[u][j] = max(f[u][j], f[u][j - k] + f[v][k]);
    }
}
int main(){
    cin >> n >> W;
    for (int i = 1; i <= n; i ++ ){
        cin >> w[i] >> v[i] >> p;
        if (p == -1) root = i;
        else g[p].push_back(i);
    }
    dfs(root);
    cout << f[root][W] << "\n";
    return 0;
}
```

背包问题求方案数

有 n 件物品和一个容量为 W 的背包，每件物品只能用一次，第 i 件物品的重量为 $w[i]$，价值为 $v[i]$，求解将哪些物品放入背包使总重量不超过背包容量，且总价值最大，输出 **最优选法的方案数**，答案可能很大，输出答案模 $10^9 + 7$ 的结果。

思路：

开一个储存方案数的数组 cnt，$cnt[i]$ 表示容量为 i 时的 **方案数**，先将 cnt 的每一个值都初始化为 1，因为 **不装任何东西就是一种方案**，如果装入这件物品使总的价值 **更大**，那么装入后的方案数 **等于** 装之前的方案数，如果装入后总价值 **相等**，那么方案数就是 **二者之和**

```cpp
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 1e9 + 7, N = 1010;
LL n, W, cnt[N], f[N], w, v;
int main(){
    cin >> n >> W;
    for (int i = 0; i <= W; i ++ )
        cnt[i] = 1;
    for (int i = 0; i < n; i ++ ){
```

```cpp
        cin >> w >> v;
        for (int j = W; j >= w; j -- )
            if (f[j] < f[j - w] + v){
                f[j] = f[j - w] + v;
                cnt[j] = cnt[j - w];
            }
            else if (f[j] == f[j - w] + v){
                cnt[j] = (cnt[j] + cnt[j - w]) % mod;
            }
    }
    cout << cnt[W] << "\n";
    return 0;
}
```

背包问题求具体方案

```cpp
signed main() {
    int Task = 1;
    for (cin >> Task; Task; Task--) {
        int n, m;
        cin >> n >> m;

        vector<int> w(n + 1), v(n + 1);
        vector<vector<int>> dp(n + 2, vector<int>(m + 2));
        for (int i = 1; i <= n; i++) {
            cin >> w[i] >> v[i];
        }

        for (int i = n; i >= 1; i--) {
            for (int j = 0; j <= m; j++) {
                dp[i][j] = dp[i + 1][j];
                if (j >= w[i]) {
                    dp[i][j] = max(dp[i][j], dp[i + 1][j - w[i]] + v[i]);
                }
            }
        }

        vector<int> ans;
        for (int i = 1; i <= n; i++) {
            if (m - w[i] >= 0 && dp[i][m] == dp[i + 1][m - w[i]] + v[i]) {
                ans.push_back(i);
                // cout << i << " ";
                m -= w[i];
            }
        }
        cout << ans.size() << "\n";
        for (auto i : ans) {
            cout << i << " ";
        }
        cout << "\n";
    }
}
```

数位 DP

```cpp
/* pos 表示当前枚举到第几位
sum 表示 d 出现的次数
limit 为 1 表示枚举的数字有限制
zero 为 1 表示有前导 0
d 表示要计算出现次数的数 */
const int N = 15;
LL dp[N][N];
int num[N];
LL dfs(int pos, LL sum, int limit, int zero, int d) {
    if (pos == 0) return sum;
    if (!limit && !zero && dp[pos][sum] != -1) return dp[pos][sum];
    LL ans = 0;
    int up = (limit ? num[pos] : 9);
    for (int i = 0; i <= up; i++) {
        ans += dfs(pos - 1, sum + ((!zero || i) && (i == d)), limit && (i == num[pos]),
                   zero && (i == 0), d);
    }
    if (!limit && !zero) dp[pos][sum] = ans;
    return ans;
}
LL solve(LL x, int d) {
    memset(dp, -1, sizeof dp);
    int len = 0;
    while (x) {
        num[++len] = x % 10;
        x /= 10;
    }
    return dfs(len, 0, 1, 1, d);
}
```

状压 DP

题意： 在 $n * n$ 的棋盘里面放 k 个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共8个格子。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int N = 15, M = 150, K = 1500;
LL n, k;
LL cnt[K];    //每个状态的二进制中 1 的数量
LL tot;    //合法状态的数量
LL st[K];    //合法的状态
LL dp[N][M][K];    //第 i 行，放置了 j 个国王，状态为 k 的方案数
int main(){
    ios::sync_with_stdio(false);cin.tie(0);
    cin >> n >> k;
    for (int s = 0; s < (1 << n); s ++ ){  //找出合法状态
        LL sum = 0, t = s;
        while(t){  //计算 1 的数量
            sum += (t & 1);
```

```
            t >>= 1;
        }
        cnt[s] = sum;
        if ( (( (s << 1) | (s >> 1) ) & s) == 0 ){  //判断合法性
            st[ ++ tot] = s;
        }
    }
    dp[0][0][0] = 1;
    for (int i = 1; i <= n + 1; i ++ ){
        for (int j1 = 1; j1 <= tot; j1 ++ ){    //当前的状态
            LL s1 = st[j1];
            for (int j2 = 1; j2 <= tot; j2 ++ ){    //上一行的状态
                LL s2 = st[j2];
                if ( ( (s2 | (s2 << 1) | (s2 >> 1)) & s1 ) == 0 ){
                    for (int j = 0; j <= k; j ++ ){
                        if (j - cnt[s1] >= 0)
                            dp[i][j][s1] += dp[i - 1][j - cnt[s1]][s2];
                    }
                }
            }
        }
    }
    cout << dp[n + 1][k][0] << "\n";
    return 0;
}
```

常用例题

题意：在一篇文章（包含大小写英文字母、数字、和空白字符（制表/空格/回车））中寻找 `helloworld`（任意一个字母的大小写都行）的子序列出现了多少次，输出结果对 10^9+7 的余数。

字符串 DP，构建一个二维 DP 数组，$dp[i][j]$ 的 i 表示文章中的第几个字符，j 表示寻找的字符串的第几个字符，当字符串中的字符和文章中的字符相同时，即找到符合条件的字符， `dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1]` ，因为字符串中的每个字符不会对后面的结果产生影响，所以 DP 方程可以优化成一维的， 由于字符串中有重复的字符，所以比较时应该从后往前。

```
#include <bits/stdc++.h>
using namespace std;
#define LL long long
const int mod = 1e9 + 7;
char c, s[20] = "!helloworld";
LL dp[20];
int main(){
    dp[0] = 1;
    while ((c = getchar()) != EOF)
        for (int i = 10; i >= 1; i--)
            if (c == s[i] || c == s[i] - 32)
                dp[i] = (dp[i] + dp[i - 1]) % mod;
    cout << dp[10] << "\n";
    return 0;
}
```

题意：（最长括号匹配）给一个只包含‘(’，‘)’，‘[’，‘]’的非空字符串，“()”和“[]”是匹配的，寻找字符串中最长的括号匹配的子串，若有两串长度相同，输出靠前的一串。

设给定的字符串为 s ，可以定义数组 $dp[i], dp[i]$ 表示以 $s[i]$ 结尾的字符串里最长的括号匹配的字符。显然，从 $i - dp[i] + 1$ 到 i 的字符串是括号匹配的，当找到一个字符是‘)’或‘]’时，再去判断第 $i - 1 - dp[i - 1]$ 的字符和第 i 位的字符是否匹配，如果是，那么 `dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]]` 。

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e6 + 10;
string s;
int len, dp[maxn], ans, id;
int main(){
    cin >> s;
    len = s.length();
    for (int i = 1; i < len; i++){
        if ((s[i] == ')' && s[i - 1 - dp[i - 1]] == '(' ) || (s[i] == ']' && s[i - 1 - dp[i - 1]] == '[')){
            dp[i] = dp[i - 1] + 2 + dp[i - 2 - dp[i - 1]];
            if (dp[i] > ans) {
                ans = dp[i];  //记录长度
                id = i;  //记录位置
            }
        }
    }
    for (int i = id - ans + 1; i <= id; i++)
        cout << s[i];
    cout << "\n";
    return 0;
}
```

题意：去掉区间内包含“4”和“62”的数字，输出剩余的数字个数

```
int T,n,m,len,a[20];//a数组用于判断每一位能取到的最大值
ll l,r,dp[20][15];
ll dfs(int pos,int pre,int limit){//记搜
    //pos搜到的位置，pre前一位数
    //limit判断是否有最高位限制
    if(pos>len) return 1;//剪枝
    if(dp[pos][pre]!=-1 && !limit) return dp[pos][pre];//记录当前值
    ll ret=0;//暂时记录当前方案数
    int res=limit?a[len-pos+1]:9;//res当前位能取到的最大值
    for(int i=0;i<=res;i++)
        if(!(i==4 || (pre==6 && i==2)))
            ret+=dfs(pos+1,i,i==res&&limit);
    if(!limit) dp[pos][pre]=ret;//当前状态方案数记录
    return ret;
}
ll part(ll x){//把数按位拆分
    len=0;
    while(x) a[++len]=x%10,x/=10;
    memset(dp,-1,sizeof dp);//初始化-1（因为有可能某些情况下的方案数是0）
```

```
    return dfs(1,0,1);//进入记搜
}
int main(){
    cin>>n;
    while(n--){
        cin>>l>>r;
        if(l==0 && r==0)break;
        if(l) printf("%lld\n",part(r)-part(l-1));//[l,r](l!=0)
        else printf("%lld\n",part(r)-part(l));//从0开始要特判
    }
}
```

/END/

串

子串与子序列

中文名称	常见英文名称	解释
子串	`substring`	连续的选择一段字符（可以全选、可以不选）组成的新字符串
子序列	`subsequence`	从左到右取出若干个字符（可以不取、可以全取、可以不连续）组成的新字符串

字符串模式匹配算法 KMP

> 应用：
>
> 1. 在字符串中查找子串；
> 2. 最小周期：字符串长度-整个字符串的 `border`；
> 3. 最小循环节：区别于周期，当字符串长度 $n \bmod (n - nxt[n]) = 0$ 时，等于最小周期，否则为 n 。

以最坏 $\mathcal{O}(N+M)$ 的时间计算 t 在 s 中出现的全部位置。

```cpp
auto kmp = [&](string s, string t) {
    int n = s.size(), m = t.size();
    vector<int> kmp(m + 1), ans;
    s = "@" + s;
    t = "@" + t;
    for (int i = 2, j = 0; i <= m; i++) {
        while (j && t[i] != t[j + 1]) {
            j = kmp[j];
        }
        j += t[i] == t[j + 1];
        kmp[i] = j;
    }
    for (int i = 1, j = 0; i <= n; i++) {
        while (j && s[i] != t[j + 1]) {
            j = kmp[j];
        }
        if (s[i] == t[j + 1] && ++j == m) {
            ans.push_back(i - m + 1); // t 在 s 中出现的位置
        }
    }
    return ans;
};
```

Z函数（扩展 KMP）

获取字符串 s 和 $s[i, n-1]$ （即以 $s[i]$ 开头的后缀）的最长公共前缀（LCP）的长度，总复杂度 $\mathcal{O}(N)$。

```cpp
vector<int> zFunction(string s) {
```

```
    int n = s.size();
    vector<int> z(n);
    z[0] = n;
    for (int i = 1, j = 1; i < n; i++) {
        z[i] = max(0, min(j + z[j] - i, z[i - j]));
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if (i + z[i] > j + z[j]) {
            j = i;
        }
    }
    return z;
}
```

最长公共子序列 LCS

求解两个串的最长公共子序列的长度。

小数据解

针对 10^3 以内的数据。

```
const int N = 1e3 + 10;
char a[N], b[N];
int n, m, f[N][N];
void solve(){
    cin >> n >> m >> a + 1 >> b + 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++){
            f[i][j] = max(f[i - 1][j], f[i][j - 1]);
            if (a[i] == b[j]) f[i][j] = max(f[i][j], f[i - 1][j - 1] + 1);
        }
    cout << f[n][m] << "\n";
}
int main(){
    solve();
    return 0;
}
```

大数据解

针对 10^5 以内的数据。

```
const int INF = 0x7fffffff;
int n, a[maxn], b[maxn], f[maxn], p[maxn];
int main(){
    cin >> n;
    for (int i = 1; i <= n; i++){
        scanf("%d", &a[i]);
        p[a[i]] = i;  //将第二个序列中的元素映射到第一个中
    }
    for (int i = 1; i <= n; i++){
        scanf("%d", &b[i]);
        f[i] = INF;
```

```cpp
    }
    int len = 0;
    f[0] = 0;
    for (int i = 1; i <= n; i++){
        if (p[b[i]] > f[len]) f[++len] = p[b[i]];
        else {
            int l = 0, r = len;
            while (l < r){
                int mid = (l + r) >> 1;
                if (f[mid] > p[b[i]]) r = mid;
                else l = mid + 1;
            }
            f[l] = min(f[l], p[b[i]]);
        }
    }
    cout << len << "\n";
    return 0;
}
```

字符串哈希

双哈希封装

随机质数列表：1111111121、1211111123、1311111119。

```cpp
const int N = 1 << 21;
static const int mod1 = 1E9 + 7, base1 = 127;
static const int mod2 = 1E9 + 9, base2 = 131;
using U = Zmod<mod1>;
using V = Zmod<mod2>;
vector<U> val1;
vector<V> val2;
void init(int n = N) {
    val1.resize(n + 1), val2.resize(n + 2);
    val1[0] = 1, val2[0] = 1;
    for (int i = 1; i <= n; i++) {
        val1[i] = val1[i - 1] * base1;
        val2[i] = val2[i - 1] * base2;
    }
}
struct String {
    vector<U> hash1;
    vector<V> hash2;
    string s;

    String(string s_) : s(s_), hash1{1}, hash2{1} {
        for (auto it : s) {
            hash1.push_back(hash1.back() * base1 + it);
            hash2.push_back(hash2.back() * base2 + it);
        }
    }
    pair<U, V> get() { // 输出整串的哈希值
        return {hash1.back(), hash2.back()};
    }
    pair<U, V> substring(int l, int r) { // 输出子串的哈希值
```

```
        if (l > r) swap(l, r);
        U ans1 = hash1[r + 1] - hash1[l] * val1[r - l + 1];
        V ans2 = hash2[r + 1] - hash2[l] * val2[r - l + 1];
        return {ans1, ans2};
    }
    pair<U, V> modify(int idx, char x) { // 修改 idx 位为 x
        int n = s.size() - 1;
        U ans1 = hash1.back() + val1[n - idx] * (x - s[idx]);
        V ans2 = hash2.back() + val2[n - idx] * (x - s[idx]);
        return {ans1, ans2};
    }
};
```

前后缀去重

`sample please ease` 去重后得到 `samplease`。

```
string compress(vector<string> in) { // 前后缀压缩
    vector<U> hash1{1};
    vector<V> hash2{1};
    string ans = "#";
    for (auto s : in) {
        s = "#" + s;
        int st = 0;
        U chk1 = 0;
        V chk2 = 0;
        for (int j = 1; j < s.size() && j < ans.size(); j++) {
            chk1 = chk1 * base1 + s[j];
            chk2 = chk2 * base2 + s[j];
            if ((hash1.back() == hash1[ans.size() - 1 - j] * val1[j] + chk1) &&
                (hash2.back() == hash2[ans.size() - 1 - j] * val2[j] + chk2)) {
                st = j;
            }
        }
        for (int j = st + 1; j < s.size(); j++) {
            ans += s[j];
            hash1.push_back(hash1.back() * base1 + s[j]);
            hash2.push_back(hash2.back() * base2 + s[j]);
        }
    }
    return ans.substr(1);
}
```

马拉车

$\mathcal{O}(N)$ 时间求出字符串的最长回文子串。

```
string s;
cin >> s;
int n = s.length();
string t = "-#";
for (int i = 0; i < n; i++) {
```

```cpp
    t += s[i];
    t += '#';
}
int m = t.length();
t += '+';
int mid = 0, r = 0;
vector<int> p(m);
for (int i = 1; i < m; i++) {
    p[i] = i < r ? min(p[2 * mid - i], r - i) : 1;
    while (t[i - p[i]] == t[i + p[i]]) p[i]++;
    if (i + p[i] > r) {
        r = i + p[i];
        mid = i;
    }
}
```

博弈论

巴什博奕

> 有 N 个石子，两名玩家轮流行动，按以下规则取石子：
>
> 规定：每人每次可以取走 $X(1 \le X \le M)$ 个石子，拿到最后一颗石子的一方获胜。
>
> 双方均采用最优策略，询问谁会获胜。

> 两名玩家轮流报数。
>
> 规定：第一个报数的人可以报 $X(1 \le X \le M)$，后报数的人需要比前者所报数大 $Y(1 \le Y \le M)$，率先报到 N 的人获胜。
>
> 双方均采用最优策略，询问谁会获胜。

- $N = K \cdot (M+1)$（其中 $K \in \mathbb{N}^+$），后手必胜（后手可以控制每一回合结束时双方恰好取走 $M+1$ 个，重复 K 轮后即胜利）；
- $N = K \cdot (M+1) + R$（其中 $K \in \mathbb{N}^+, 0 < R < M+1$），先手必胜（先手先取走 R 个，之后控制每一回合结束时双方恰好取走 $M+1$ 个，重复 K 轮后即胜利）。

扩展巴什博弈

> 有 N 颗石子，两名玩家轮流行动，按以下规则取石子：。
>
> 规定：每人每次可以取走 $X(a \le X \le b)$ 个石子，如果最后剩余物品的数量小于 a 个，则不能再取，拿到最后一颗石子的一方获胜。
>
> 双方均采用最优策略，询问谁会获胜。

- $N = K \cdot (a+b)$ 时，后手必胜；
- $N = K \cdot (a+b) + R_1$（其中 $K \in \mathbb{N}^+, 0 < R_1 < a$）时，后手必胜（这些数量不够再取一次，先手无法逆转局面）；
- $N = K \cdot (a+b) + R_2$（其中 $K \in \mathbb{N}^+, a \le R_2 \le b$）时，先手必胜；
- $N = K \cdot (a+b) + R_3$（其中 $K \in \mathbb{N}^+, b < R_3 < a+b$）时，先手必胜（这些数量不够再取一次，后手无法逆转局面）；

Nim 博弈

> 有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：
>
> 规定：每人每次任选一堆，取走正整数颗石子，拿到最后一颗石子的一方获胜（注：几个特点是**不能跨堆**、**不能不拿**）。
>
> 双方均采用最优策略，询问谁会获胜。

记初始时各堆石子的数量 $(A_1, A_2, \dots, A_n)$，定义尼姆和 $Sum_N = A_1 \oplus A_2 \oplus \dots \oplus A_n$。

当 $Sum_N = 0$ 时先手必败，反之先手必胜。

Nim 游戏具体取法

先计算出尼姆和，再对每一堆石子计算 $A_i \oplus Sum_N$，记为 X_i。

若得到的值 $X_i < A_i$，X_i 即为一个可行解，即**剩下 X_i 颗石头，取走 $A_i - X_i$ 颗石头**（这里取小于号是因为至少要取走 1 颗石子）。

Moore's Nim 游戏（Nim-K 游戏）

> 有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：
>
> 规定：每人每次任选不超过 K 堆，对每堆都取走不同的正整数颗石子，拿到最后一颗石子的一方获胜。
>
> 双方均采用最优策略，询问谁会获胜。

把每一堆石子的石子数用二进制表示，定义 One_i 为二进制第 i 位上 1 的个数。

以下局面先手必胜：

对于每一位，$One_1, One_2, \dots, One_N$ 均不为 $K+1$ 的倍数。

Anti-Nim 游戏（反 Nim 游戏）

> 有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，按以下规则取石子：
>
> 规定：每人每次任选一堆，取走正整数颗石子，拿到最后一颗石子的一方**出局**。
>
> 双方均采用最优策略，询问谁会获胜。

- 所有堆的石头数量均不超过 1，且 $\boldsymbol{Sum_N = 0}$（也可看作“且有偶数堆”）；
- 至少有一堆的石头数量大于 1，且 $\boldsymbol{Sum_N \neq 0}$。

阶梯 - Nim 博弈

> 有 N 级台阶，每一级台阶上均有一定数量的石子，给出每一级石子的数量，两名玩家轮流行动，按以下规则操作石子：
>
> 规定：每人每次任选一级台阶，拿走正整数颗石子放到下一级台阶中，已经拿到地面上的石子不能再拿，拿到最后一颗石子的一方获胜。
>
> 双方均采用最优策略，询问谁会获胜。

对奇数台阶做传统 `Nim` 博弈，当 $Sum_N = 0$** 时先手必败，反之先手必胜。**

SG 游戏（有向图游戏）

我们使用以下几条规则来定义暴力求解的过程：

- 使用数字来表示输赢情况，0 代表局面必败，非 0 代表**存在必胜可能**，我们称这个数字为这个局面的SG值；
- 找到最终态，根据题意人为定义最终态的输赢情况；
- 对于非最终态的某个节点，其SG值为所有子节点的SG值取 mex ；
- 单个游戏的输赢态即对应根节点的SG值是否为 0 ，为 0 代表先手必败，非 0 代表先手必胜；
- 多个游戏的总SG值为单个游戏SG值的异或和。

使用哈希表，以 $\mathcal{O}(N+M)$ 的复杂度计算。

```cpp
int n, m, a[N], num[N];
int sg(int x) {
    if (num[x] != -1) return num[x];

    unordered_set<int> S;
    for (int i = 1; i <= m; ++ i)
        if(x >= a[i])
            S.insert(sg(x - a[i]));

    for (int i = 0; ; ++ i)
        if (S.count(i) == 0)
            return num[x] = i;
}
void Solve() {
    cin >> m;
    for (int i = 1; i <= m; ++ i) cin >> a[i];
    cin >> n;

    int ans = 0; memset(num, -1, sizeof num);
    for (int i = 1; i <= n; ++ i) {
        int x; cin >> x;
        ans ^= sg(x);
    }

    if (ans == 0) no;
    else yes;
}
```

Anti-SG 游戏（反 SG 游戏）

SG 游戏中最先不能行动的一方获胜。

以下局面先手必胜：

- **单局游戏的SG值均不超过 1 ，且总SG值为 0**；
- **至少有一局单局游戏的SG值大于 1 ，且总SG值不为 0** 。

在本质上，这与 Anti-Nim 游戏的结论一致。

Lasker's-Nim 游戏（ Multi-SG 游戏）

> 有 N 堆石子，给出每一堆的石子数量，两名玩家轮流行动，每人每次任选以下规定的一种操作石子：
>
> - 任选一堆，取走正整数颗石子；
> - 任选数量大于 2 的一堆，分成两堆非空石子。
>
> 拿到最后一颗石子的一方获胜。双方均采用最优策略，询问谁会获胜。

本题使用SG函数求解，SG值定义为：

$$SG(x) = \begin{cases} x-1 & ,x \mod 4 = 0 \\ x & ,x \mod 4 = 1 \\ x & ,x \mod 4 = 2 \\ x+1 & ,x \mod 4 = 3 \end{cases}$$

Every-SG 游戏

> 给出一个有向无环图，其中 K 个顶点上放置了石子，两名玩家轮流行动，按以下规则操作石子：
>
> 移动图上所有还能够移动的石子；
>
> 无法移动石子的一方出局。双方均采用最优策略，询问谁会获胜。

定义 $step$ 为某一局游戏至多需要经过的回合数。

以下局面先手必胜：$step$ 为奇数 。

威佐夫博弈

> 有两堆石子，给出每一堆的石子数量，两名玩家轮流行动，每人每次任选以下规定的一种操作石子：
>
> - 任选一堆，取走正整数颗石子；
> - 从两队中同时取走正整数颗石子。
>
> 拿到最后一颗石子的一方获胜。双方均采用最优策略，询问谁会获胜。

以下局面先手必败：

$(1,2),(3,5),(4,7),(6,10),\dots$ 具体而言，每一对的第一个数为此前没出现过的最小整数，第二个数为第一个数加上 $1,2,3,4,\dots$ 。

更一般地，对于第 k 对数，第一个数为 $First_k = \left\lfloor \frac{k*(1+\sqrt{5})}{2} \right\rfloor$ ，第二个数为 $Second_k = First_k + k$ 。

其中，在两堆石子的数量均大于 10^9 次时，由于需要使用高精度计算，我们需要人为定义 $\frac{1+\sqrt{5}}{2}$ 的取值为 $lorry = 1.618033988749894848204586834$ 。

```
const double lorry = (sqrt(5.0) + 1.0) / 2.0;
//const double lorry = 1.618033988749894848204586834;
void Solve() {
    int n, m; cin >> n >> m;
    if (n < m) swap(n, m);
    double x = n - m;
    if ((int)(lorry * x) == m) cout << "lose\n";
    else cout << "win\n";
}
```

斐波那契博弈

> 有一堆石子，数量为 N ，两名玩家轮流行动，按以下规则取石子：
>
> 先手第1次可以取任意多颗，但不能全部取完，此后每人取的石子数不能超过上个人的两倍，拿到最后一颗石子的一方获胜。
>
> 双方均采用最优策略，询问谁会获胜。

当且仅当 N 为斐波那契数时先手必败。

```
int fib[100] = {1, 2};
map<int, bool> mp;
void Force() {
  for (int i = 2; i <= 86; ++ i) fib[i] = fib[i - 1] + fib[i - 2];
    for (int i = 0; i <= 86; ++ i) mp[fib[i]] = 1;
}
void Solve() {
    int n; cin >> n;
    if (mp[n] == 1) cout << "lose\n";
    else cout << "win\n";
}
```

树上删边游戏

> 给出一棵 N 个节点的有根树，两名玩家轮流行动，按以下规则操作：
>
> 选择任意一棵子树并删除（即删去任意一条边，不与根相连的部分会同步被删去）；
>
> 删掉最后一棵子树的一方获胜（换句话说，删去根节点的一方失败）。双方均采用最优策略，询问谁会获胜。

结论：当根节点SG值非 1 时先手必胜。

相较于传统SG值的定义，本题的SG函数值定义为：

- 叶子节点的SG值为 $\mathbf{0}$ 。
- 非叶子节点的SG值为其所有孩子节点SG值 $+1$ 的异或和。

```cpp
auto dfs = [&](auto self, int x, int fa) -> int {
    int x = 0;
    for (auto y : ver[x]) {
        if (y == fa) continue;
        x ^= self(self, y, x);
    }
    return x + 1;
};
cout << (dfs(dfs, 1, 0) == 1 ? "Bob\n" : "Alice\n");
```

无向图删边游戏（Fusion Principle 定理）

> 给出一张 N 个节点的无向联通图，有一个点作为图的根，两名玩家轮流行动，按以下规则操作：
>
> 选择任意一条边删除，不与根相连的部分会同步被删去；
>
> 删掉最后一条边的一方获胜。双方均采用最优策略，询问谁会获胜。

- **对于奇环，我们将其缩成一个新点+一条新边；**
- **对于偶环，我们将其缩成一个新点；**
- **所有连接到原来环上的边全部与新点相连。**

此时，本模型转化为“树上删边游戏”。

/END/

STL

库函数

pb_ds 库

其中 `gp_hash_table` 使用的最多，其等价于 `unordered_map` ，内部是无序的。

```
#include <bits/extc++.h>
#include <ext/pb_ds/assoc_container.hpp>
template<class S, class T> using omap = __gnu_pbds::gp_hash_table<S, T, myhash>;
```

查找后继 lower_bound、upper_bound

`lower` 表示 $\geq$ ， `upper` 表示 $>$ 。使用前记得**先进行排序**。

```
//返回a数组[start,end)区间中第一个>=x的地址【地址！！！】
cout << lower_bound(a + start, a + end, x);

cout << lower_bound(a, a + n, x) - a; //在a数组中查找第一个>=x的元素下标
upper_bound(a, a + n, k) - lower_bound(a, a + n, k) //查找k在a中出现了几次
```

数组打乱 shuffle

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
shuffle(ver.begin(), ver.end(), rnd);
```

二分搜索 binary_search

用于查找某一元素是否在容器中，相当于 find 函数。在使用前需要**先进行排序**。

```
//在a数组[start,end)区间中查找x是否存在，返回bool型
cout << binary_search(a + start, a + end, x);
```

批量递增赋值函数 iota

对容器递增初始化。

```
//将a数组[start,end)区间复制成“x，x+1，x+2，…”
iota(a + start, a + end, x);
```

数组去重函数 unique

在使用前需要**先进行排序**。

其作用是，对于区间 `[开始位置, 结束位置)` ，**不停的把后面不重复的元素移到前面来**，也可以说是**用不重复的元素占领重复元素的位置**。并且返回**去重后容器中不重复序列的最后一个元素的下一个元素**。所以在进行操作后，数组、容器的大小并**没有发生改变**。

```cpp
//将a数组[start,end)区间去重，返回迭代器
unique(a + start, a + end);

//与earse函数结合，达到去重+删除的目的
a.erase(unique(ALL(a)), a.end());
```

bit 库与位运算函数 __builtin_

```cpp
__builtin_popcount(x) // 返回x二进制下含1的数量，例如x=15=(1111)时答案为4

__builtin_ffs(x) // 返回x右数第一个1的位置(1-idx)，1(1) 返回 1，8(1000) 返回 4，26(11010) 返回 2

__builtin_ctz(x) // 返回x二进制下后导0的个数，1(1) 返回 0，8(1000) 返回 3

bit_width(x) // 返回x二进制下的位数，9(1001) 返回 4，26(11010) 返回 5
```

注：以上函数的 `long long` 版本只需要在函数后面加上 `ll` 即可（例如 `__builtin_popcountll(x)`)， `unsigned long long` 加上 `ull` 。

数字转字符串函数

`itoa` 虽然能将整数转换成任意进制的字符串，但是其不是标准的C函数，且为Windows独有，且不支持 `long long` ，建议手写。

```cpp
// to_string函数会直接将你的各种类型的数字转换为字符串。
// string to_string(T val);
double val = 12.12;
cout << to_string(val);
```

```cpp
// 【不建议使用】itoa允许你将整数转换成任意进制的字符串，参数为待转换整数、目标字符数组、进制。
// char* itoa(int value, char* string, int radix);
char ans[10] = {};
itoa(12, ans, 2);
cout << ans << endl; /*1100*/

// 长整型函数名ltoa，最高支持到int型上限2^31。ultoa同理。
```

字符串转数字

```cpp
// stoi直接使用
cout << stoi("12") << endl;

// 【不建议使用】stoi转换进制，参数为待转换字符串、起始位置、进制。
// int stoi(string value, int st, int radix);
cout << stoi("1010", 0, 2) << endl; /*10*/
cout << stoi("c", 0, 16) << endl; /*12*/
cout << stoi("0x3f3f3f3f", 0, 0) << endl; /*1061109567*/

// 长整型函数名stoll，最高支持到long long型上限2^63。stoull、stod、stold同理。
```

```cpp
// atoi直接使用，空字符返回0，允许正负符号，数字字符前有其他字符返回0，数字字符前有空白字符自动去除
cout << atoi("12") << endl;
cout << atoi("   12") << endl; /*12*/
cout << atoi("-12abc") << endl; /*-12*/
cout << atoi("abc12") << endl; /*0*/

// 长整型函数名atoll，最高支持到long long型上限2^63。
```

全排列算法 next_permutation、prev_permutation

在提及这个函数时，我们先需要补充几点字典序相关的知识。

> 对于三个字符所组成的序列`{a,b,c}`，其按照字典序的6种排列分别为：
> `{abc}`，`{acb}`，`{bac}`，`{bca}`，`{cab}`，`{cba}`
> 其排序原理是：先固定 `a` (序列内最小元素)，再对之后的元素排列。而 `b` < `c` ，所以 `abc` < `acb` 。同理，先固定 `b` (序列内次小元素)，再对之后的元素排列。即可得出以上序列。

`next_permutation` 算法，即是按照**字典序顺序**输出的全排列；相对应的，`prev_permutation` 则是按照**逆字典序顺序**输出的全排列。可以是数字，亦可以是其他类型元素。其直接在序列上进行更新，故直接输出序列即可。

```cpp
int n;
cin >> n;
vector<int> a(n);
// iota(a.begin(), a.end(), 1);
for (auto &it : a) cin >> it;
sort(a.begin(), a.end());

do {
    for (auto it : a) cout << it << " ";
    cout << endl;
} while (next_permutation(a.begin(), a.end()));
```

字符串转换为数值函数 sto

可以快捷的将**一串字符串**转换为**指定进制的数字**。

使用方法

- `stoi(字符串, 0, x进制)` ：将一串 `x` 进制的字符串转换为 `int` 型数字。

```
void Solve() {
    cout << stoi("1010", 0, 2) << endl;
    cout << stoi("c", 0, 16) << endl;
    cout << stoi("0x3f3f3f3f", 0, 0) << endl;
    cout << stoi("10", 0, 8) << endl;
    cout << stoll("aaaaaaaaaaa", 0, 16) << endl;
}
```

C:\Users\26099\Desktop\万能头文件.exe

```
10
12
1061109567
8
11728124029610
```

- `stoll(字符串, 0, x进制)`：将一串 x 进制的字符串转换为 long long 型数字。
- stoull，stod，stold 同理。

数值转换为字符串函数 to_string

允许将**各种数值类型**转换为字符串类型。

```
//将数值num转换为字符串s
string s = to_string(num);
```

判断非递减 is_sorted

```
//a数组[start,end)区间是否是非递减的，返回bool型
cout << is_sorted(a + start, a + end);
```

累加 accumulate

```
//将a数组[start,end)区间的元素进行累加，并输出累加和+x的值
cout << accumulate(a + start, a + end, x);
```

迭代器 iterator

```
//构建一个UUU容器的正向迭代器，名字叫it
UUU::iterator it;

vector<int>::iterator it; //创建一个正向迭代器，++ 操作时指向下一个
vector<int>::reverse_iterator it; //创建一个反向迭代器，++ 操作时指向上一个
```

其他函数

`exp2(x)`：返回 2^x

`log2(x)`：返回 $\log_2(x)$

`gcd(x, y) / lcm(x, y)`：以 log 的复杂度返回 $\gcd(|x|,|y|)$ 与 $\mathtt{lcm}(|x|,|y|)$，且返回值符号也为正数。

容器与成员函数

元组 tuple

```
//获取obj对象中的第index个元素——get<index>(obj)
//需要注意的是这里的index只能手动输入，使用for循环这样的自动输入是不可以的
tuple<string, int, int> Student = {"Wida", 23, 45000);
cout << get<0>(Student) << endl; //获取Student对象中的第一个元素，这里的输出结果应为“Wida”
```

数组 array

```
array<int, 3> x; // 建立一个包含三个元素的数组x

[] // 跟正常数组一样，可以使用随机访问
cout << x[0]; // 获取数组重的第一个元素
```

变长数组 vector

```
resize(n) // 重设容器大小，但是不改变已有元素的值
assign(n, 0) // 重设容器大小为n，且替换容器内的内容为0

// 尽量不要使用[]的形式声明多维变长数组，而是使用嵌套的方式替代
vector<int> ver[n + 1]; // 不好的声明方式
vector<vector<int>> ver(n + 1);

// 嵌套时只需要在最后一个注明变量类型
vector dis(n + 1, vector<int>(m + 1));
vector dis(m + 1, vector(n + 1, vector<int>(n + 1)));
```

栈 stack

栈顶入，栈顶出。先进后出。

```
//没有clear函数
size() / empty()
push(x) //向栈顶插入x
top() //获取栈顶元素
pop() //弹出栈顶元素
```

队列 queue

队尾进，队头出。先进先出。

```
//没有clear函数
size() / empty()
push(x) //向队尾插入x
front() / back() //获取队头、队尾元素
pop() //弹出队头元素
```

```
//没有clear函数，但是可以用重新构造替代
queue<int> q;
q = queue<int>();
```

双向队列 deque

```
size() / empty() / clear()
push_front(x) / push_back(x)
pop_front(x) / pop_back(x)
front() / back()
begin() / end()
[]
```

优先队列 priority_queue

默认升序（大根堆），自定义排序需要重载 `<` 。

```
//没有clear函数
priority_queue<int, vector<int>, greater<int> > p; //重定义为降序（小根堆）
push(x); //向栈顶插入x
top(); //获取栈顶元素
pop(); //弹出栈顶元素
```

```
//重载运算符【注意，符号相反！！！】
struct Node {
    int x; string s;
    friend bool operator < (const Node &a, const Node &b) {
        if (a.x != b.x) return a.x > b.x;
        return a.s > b.s;
    }
};
```

字符串 string

```
size() / empty() / clear()
```

```
//从字符串S的S[start]开始，取出长度为len的子串——S.substr(start, len)
//len省略时默认取到结尾，超过字符串长度时也默认取到结尾
cout << S.substr(1, 12);

find(x) / rfind(x); //顺序、逆序查找x，返回下标，没找到时返回一个极大值【！建议与 size() 比较，而不要和 -1 比较，后者可能出错】
//注意，没有count函数
```

有序、多重有序集合 set、multiset

默认升序（大根堆），`set` 去重，`multiset` 不去重，$\mathcal{O}(\log N)$ 。

```
set<int, greater<> > s; //重定义为降序（小根堆）
size() / empty() / clear()
begin() / end()
++ / -- //返回前驱、后继

insert(x); //插入x
find(x) / rfind(x); //顺序、逆序查找x，返回迭代器【迭代器！！！】，没找到时返回end()
count(x); //返回x的个数
lower_cound(x); //返回第一个>=x的迭代器【迭代器！！！】
upper_cound(x); //返回第一个>x的迭代器【迭代器！！！】
```

特殊函数 `next` 和 `prev` 详解：

```
auto it = s.find(x); // 建立一个迭代器
prev(it) / next(it); // 默认返回迭代器it的前/后一个迭代器
prev(it, 2) / next(it, 2); // 可选参数可以控制返回前/后任意个迭代器

/* 以下是一些应用 */
auto pre = prev(s.lower_bound(x)); // 返回第一个<x的迭代器
int ed = *prev(S.end(), 1); // 返回最后一个元素
```

`erase(x);` 有两种删除方式：

- 当x为某一元素时，删除**所有**这个数，复杂度为 $\mathcal{O}(num_x+logN)$；
- 当x为迭代器时，删除这个迭代器。

```
//连续头部删除
set<int> S = {0, 9, 98, 1087, 894, 34, 756};
auto it = S.begin();
int len = S.size();
for (int i = 0; i < len; ++ i) {
    if (*it >= 500) continue;
    it = S.erase(it); //删除所有小于500的元素
}
//错误用法如下【千万不能这样用！！！】
//for (auto it : S) {
//    if (it >= 500) continue;
//    S.erase(it); //删除所有小于500的元素
//}
```

map、multimap

默认升序（大根堆），`map` 去重，`mulitmap` 不去重，$\mathcal{O}(logS)$，其中 S 为元素数量。

```
map<int, int, greater<> > mp; //重定义为降序（小根堆）
size() / empty() / clear()
begin() / end()
++ / -- //返回前驱、后继

insert({x, y}); //插入二元组
[] //随机访问，multimap不支持
count(x); //返回x为下标的个数
lower_cound(x); //返回第一个下标>=x的迭代器
upper_cound(x); //返回第一个下标>x的迭代器
```

`erase(x);` 有两种删除方式：

- 当x为某一元素时，删除所有**以这个元素为下标的二元组**，复杂度为 $\mathcal{O}(num_x + logN)$ ；
- 当x为迭代器时，删除这个迭代器。

慎用随机访问！——当不确定某次查询是否存在于容器中时，不要直接使用下标查询，而是先使用 `count()` 或者 `find()` 方法检查key值，防止不必要的零值二元组被构造。

```
int q = 0;
if (mp.count(i)) q = mp[i];
```

慎用自带的 pair、tuple 作为key值类型！使用自定义结构体！

```
struct fff {
    LL x, y;
    friend bool operator < (const fff &a, const fff &b) {
        if (a.x != b.x) return a.x < b.x;
        return a.y < b.y;
    }
};
map<fff, int> mp;
```

bitset

将数据转换为二进制，从高位到低位排序，以 0 为最低位。当位数相同时支持全部的位运算。

```
// 如果输入的是01字符串，可以直接使用">>"读入
bitset<10> s;
cin >> s;

//使用只含01的字符串构造——bitset<容器长度>B (字符串)
string S; cin >> S;
bitset<32> B (S);

//使用整数构造（两种方式）
int x; cin >> x;
bitset<32> B1 (x);
bitset<32> B2 = x;

// 构造时，尖括号里的数字不能是变量
int x; cin >> x;
bitset<x> ans; // 错误构造

```

```
[] //随机访问
set(x) //将第x位置1，x省略时默认全部位置1
reset(x) //将第x位置0，x省略时默认全部位置0
flip(x) //将第x位取反，x省略时默认全部位取反
to_ullong() //重转换为ULL类型
to_string() //重转换为ULL类型
count() //返回1的个数
any() //判断是否至少有一个1
none() //判断是否全为0

_Find_fisrt() // 找到从低位到高位第一个1的位置
_Find_next(x) // 找到当前位置x的下一个1的位置，复杂度 O(n/w + count)

bitset<23> B1("11101001"), B2("11101000");
cout << (B1 ^ B2) << "\n";  //按位异或
cout << (B1 | B2) << "\n";  //按位或
cout << (B1 & B2) << "\n";  //按位与
cout << (B1 == B2) << "\n"; //比较是否相等
cout << B1 << " " << B2 << "\n"; //你可以直接使用cout输出
```

哈希系列 unordered

通常指代 unordered_map、unordered_set、unordered_multimap、unordered_multiset，与原版相比不进行排序。

如果将不支持哈希的类型作为 `key` 值代入，编译器就无法正常运行，这时需要我们为其手写哈希函数。而我们写的这个哈希函数的正确性其实并不是特别重要（但是不可以没有），当发生冲突时编译器会调用 `key` 的 `operator ==` 函数进行进一步判断。参考

对 pair、tuple 定义哈希

```
struct hash_pair {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2> &p) const {
        return hash<T1>()(p.fi) ^ hash<T2>()(p.se);
    }
};
unordered_set<pair<int, int>, int, hash_pair> S;
unordered_map<tuple<int, int, int>, int, hash_pair> M;
```

对结构体定义哈希

需要两个条件，一个是在结构体中重载等于号（区别于非哈希容器需要重载小于号，如上所述，当冲突时编译器需要根据重载的等于号判断），第二是写一个哈希函数。注意 `hash<>()` 的尖括号中的类型匹配。

```
struct fff {
    string x, y;
    int z;
    friend bool operator == (const fff &a, const fff &b) {
        return a.x == b.x || a.y == b.y || a.z == b.z;
    }
};
struct hash_fff {
    size_t operator()(const fff &p) const {
        return hash<string>()(p.x) ^ hash<string>()(p.y) ^ hash<int>()(p.z);
    }
};
unordered_map<fff, int, hash_fff> mp;
```

对 vector 定义哈希

以下两个方法均可。注意 `hash<>()` 的尖括号中的类型匹配。

```
struct hash_vector {
    size_t operator()(const vector<int> &p) const {
        size_t seed = 0;
        for (auto it : p) {
            seed ^= hash<int>()(it);
        }
        return seed;
    }
};
unordered_map<vector<int>, int, hash_vector> mp;
```

```
namespace std {
    template<> struct hash<vector<int>> {
        size_t operator()(const vector<int> &p) const {
            size_t seed = 0;
            for (int i : p) {
                seed ^= hash<int>()(i) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
            }
            return seed;
        }
    };
}
unordered_set<vector<int> > S;
```

程序标准化

使用 Lambda 函数

- `function` 统一写法

需要注意的是，虽然 `function` 定义时已经声明了返回值类型了，但是有的时候会出错（例如，声明返回 `long long` 但是返回 `int`，原因没去了解），所以推荐在后面使用 `->` 再行声明一遍。

```cpp
function<void(int, int)> clac = [&](int x, int y) -> void {
};
clac(1, 2);

function<bool(int)> dfs = [&](int x) -> bool {
    return dfs(x + 1);
};
dfs(1);
```

- `auto` 非递归写法

不需要使用递归函数时，直接用 `auto` 替换 `function` 即可。

```cpp
auto clac = [&](int x, int y) -> void {
};
```

- `auto` 递归写法

相较于 `function` 写法，需要额外引用一遍自身。

```cpp
auto dfs = [&](auto self, int x) -> bool {
    return self(self, x + 1);
};
dfs(dfs, 1);
```

使用构造函数

可以将一些必要的声明和预处理放在构造函数，在编译时，无论放置在程序的哪个位置，都会先于主函数进行。下方是我将输入流控制声明的过程。

```cpp
int __FAST_IO__ = []() { // 函数名称可以随意修改
    ios::sync_with_stdio(0), cin.tie(0);
    cout.tie(0);
    cout << fixed << setprecision(12);
    freopen("out.txt", "r", stdin);
    freopen("in.txt", "w", stdout);
    return 0;
}();
```

/END/

卡常

基础算法 | 最大公约数 `gcd` | 位运算加速

略快于内置函数。

```
LL gcd(LL a, LL b) {
    #define tz __builtin_ctzll
    if (!a || !b) return a | b;
    int t = tz(a | b);
    a >>= tz(a);
    while (b) {
        b >>= tz(b);
        if (a > b) swap(a, b);
        b -= a;
    }
    return a << t;
    #undef tz
}
```

数论 | 质数判定 | 预分类讨论加速

常数优化，达到 $\mathcal{O}(\frac{\sqrt{N}}{3})$ 。

```
bool is_prime(int n) {
    if (n < 2) return false;
    if (n == 2 || n == 3) return true;
    if (n % 6 != 1 && n % 6 != 5) return false;
    for (int i = 5, j = n / i; i <= j; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}
```

数论 | 质数判定 | Miller-Rabin

借助蒙哥马利模乘加速取模运算。

```
using u64 = uint64_t;
using u128 = __uint128_t;

struct Montgomery {
    u64 m, m2, im, l1, l2;
    Montgomery() {}
    Montgomery(u64 m) : m(m) {
        l1 = -(u64)m % m, l2 = -(u128)m % m;
        m2 = m << 1, im = m;
        for (int i = 0; i < 5; i++) {
            im *= 2 - m * im;
        }
```

```
    }
    inline u64 operator()(i64 a, i64 b) const {
        u128 c = (u128)a * b;
        return u64(c >> 64) + m - u64((u64)c * im * (u128)m >> 64);
    }
    inline u64 reduce(u64 a) const {
        a = m - u64(a * im * (u128)m >> 64);
        return a >= m ? a - m : a;
    }
    inline u64 trans(i64 a) const {
        return (*this)(a, r2);
    }

    inline u64 mul(i64 a, i64 b) const {
        u64 r = (*this)(trans(a), trans(b));
        return reduce(r);
    }
    u64 pow(u64 a, u64 n) {
        u64 r = r1;
        a = trans(a);
        for (; n; n >>= 1, a = (*this)(a, a)) {
            if (n & 1) r = (*this)(r, a);
        }
        return reduce(r);
    }
};

bool isprime(i64 n) {
    if (n < 2 || n % 6 % 4 != 1) {
        return (n | 1) == 3;
    }
    u64 s = __builtin_ctzll(n - 1), d = n >> s;
    Montgomery M(n);
    for (i64 a : {2, 325, 9375, 28178, 450775, 9780504, 1795265022}) {
        u64 p = M.pow(a, d), i = s;
        while (p != 1 && p != n - 1 && a % n && i--) {
            p = M.mul(p, p);
        }
        if (p != n - 1 && i != s) return false;
    }
    return true;
}
```

数论 | 质因数分解 | Pollard-Rho

```
struct Montgomery {} M(10); // 注意预赋值
bool isprime(i64 n) {}

i64 rho(i64 n) {
    if (!(n & 1)) return 2;
    i64 x = 0, y = 0, prod = 1;
    auto f = [&](i64 x) -> i64 {
        return M.mul(x, x) + 5; // 这里的种子能被 hack ，如果是在线比赛，请务必 rand 生成
    };
```

```
    for (int t = 30, z = 0; t % 64 || gcd(prod, n) == 1; ++t) {
        if (x == y) x = ++z, y = f(x);
        if (i64 q = M.mul(prod, x + n - y)) prod = q;
        x = f(x), y = f(f(y));
    }
    return gcd(prod, n);
}

vector<i64> factorize(i64 x) {
    vector<i64> res;
    auto f = [&](auto f, i64 x) {
        if (x == 1) return;
        M = Montgomery(x); // 重设模数
        if (isprime(x)) return res.push_back(x);
        i64 y = rho(x);
        f(f, y), f(f, x / y);
    };
    f(f, x), sort(res.begin(), res.end());
    return res;
}
```

数论 | 取模运算类 | 蒙哥马利模乘

```
using u64 = uint64_t;
using u128 = __uint128_t;

struct Montgomery {
    u64 m, m2, im, l1, l2;
    Montgomery() {}
    Montgomery(u64 m) : m(m) {
        l1 = -(u64)m % m, l2 = -(u128)m % m;
        m2 = m << 1, im = m;
        for (int i = 0; i < 5; i++) im *= 2 - m * im;
    }
    inline u64 operator()(i64 a, i64 b) const {
        u128 c = (u128)a * b;
        return u64(c >> 64) + m - u64((u64)c * im * (u128)m >> 64);
    }
    inline u64 reduce(u64 a) const {
        a = m - u64(a * im * (u128)m >> 64);
        return a >= m ? a - m : a;
    }
    inline u64 trans(i64 a) const {
        return (*this)(a, l2);
    }

    inline u64 add(i64 a, i64 b) const {
        u64 c = trans(a) + trans(b);
        if (c >= m2) c -= m2;
        return reduce(c);
    }
    inline u64 sub(i64 a, i64 b) const {
        u64 c = trans(a) - trans(b);
        if (c >= m2) c += m2;
        return reduce(c);
```

```
    }
    inline u64 mul(i64 a, i64 b) const {
        return reduce((*this)(trans(a), trans(b)));
    }
    inline u64 div(i64 a, i64 b) const {
        a = trans(a), b = trans(b);
        u64 n = m - 2, inv = 1l;
        for (; n; n >>= 1, b = (*this)(b, b))
            if (n & 1) inv = (*this)(inv, b);
        return reduce((*this)(a, inv));
    }
    u64 pow(u64 a, u64 n) {
        u64 r = 1l;
        a = trans(a);
        for (; n; n >>= 1, a = (*this)(a, a))
            if (n & 1) r = (*this)(r, a);
        return reduce(r);
    }
};
```

杂项

最长严格/非严格递增子序列 (LIS)

一维

注意子序列是不连续的。使用二分搜索，以 $\mathcal{O}(N\log N)$ 复杂度通过，另也有 $\mathcal{O}(N^2)$ 的 dp 解法。dis dis

```
vector<int> val; // 堆数
for (int i = 1, x; i <= n; i++) {
    cin >> x;
    int it = upper_bound(val.begin(), val.end(), x) - val.begin(); // low/upp: 严格/非严格递增
    if (it >= val.size()) { // 新增一堆
        val.push_back(x);
    } else { // 更新对应位置元素
        val[it] = x;
    }
}
cout << val.size() << endl;
```

二维+输出方案

```
vector<array<int, 3>> in(n + 1);
for (int i = 1; i <= n; i++) {
    cin >> in[i][0] >> in[i][1];
    in[i][2] = i;
}
sort(in.begin() + 1, in.end(), [&](auto x, auto y) {
    if (x[0] != y[0]) return x[0] < y[0];
    return x[1] > y[1];
});

```

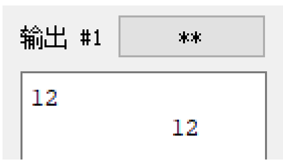
```cpp
vector<int> val{0}, idx{0}, pre(n + 1);
for (int i = 1; i <= n; i++) {
    auto [x, y, z] = in[i];
    int it = lower_bound(val.begin(), val.end(), y) - val.begin(); // low/upp: 严格/非严格递增
    if (it >= val.size()) { // 新增一堆
        pre[z] = idx.back();
        val.push_back(y);
        idx.push_back(z);
    } else { // 更新对应位置元素
        pre[z] = idx[it - 1];
        val[it] = y;
        idx[it] = z;
    }
}

vector<int> ans;
for (int i = idx.back(); i != 0; i = pre[i]) {
    ans.push_back(i);
}
reverse(ans.begin(), ans.end());
cout << ans.size() << "\n";
for (auto it : ans) {
    cout << it << " ";
}
```

cout 输出流控制

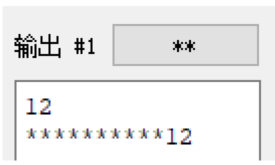
设置字段宽度：`setw(x)` ，该函数可以使得补全 x 位输出，默认用空格补全。

```cpp
bool Solve() {
    cout << 12 << endl;
    cout << setw(12) << 12 << endl;
    return 0;
}
```



设置填充字符：`setfill(x)` ，该函数可以设定补全类型，注意这里的 x 只能为 `char` 类型。

```cpp
bool Solve() {
    cout << 12 << endl;
    cout << setw(12) << setfill('*') << 12 << endl;
    return 0;
}
```



读取一行数字，个数未知

```
string s;
getline(cin, s);
stringstream ss;
ss << s;
while (ss >> s) {
    auto res = stoi(s);
    cout << res * 100 << endl;
}
```

约瑟夫问题

n 个人编号 $0, 1, 2 \ldots, n-1$ ，每次数到 k 出局，求最后剩下的人的编号。

$\mathcal{O}(N)$ 。

```
int jos(int n,int k){
    int res=0;
    repeat(i,1,n+1)res=(res+k)%i;
    return res; // res+1，如果编号从1开始
}
```

$\mathcal{O}(K \log N)$ ，适用于 K 较小的情况。

```
int jos(int n,int k){
    if(n==1 || k==1)return n-1;
    if(k>n)return (jos(n-1,k)+k)%n; // 线性算法
    int res=jos(n-n/k,k)-n%k;
    if(res<0)res+=n; // mod n
    else res+=res/(k-1); // 还原位置
    return res; // res+1，如果编号从1开始
}
```

日期换算（基姆拉尔森公式）

已知年月日，求星期数。

```
int week(int y,int m,int d){
    if(m<=2)m+=12,y--;
    return (d+2*m+3*(m+1)/5+y+y/4-y/100+y/400)%7+1;
}
```

单调队列

查询区间 k 的最大最小值。

```
deque<int> D;
int n,k,x,a[MAX];
int main(){
    IOS();
    cin>>n>>k;
```

```
    for(int i=1;i<=n;i++) cin>>a[i];
    for(int i=1;i<=n;i++){
        while(!D.empty() && a[D.back()]<=a[i]) D.pop_back();
        D.emplace_back(i);
        if(!D.empty()) if(i-D.front()>=k) D.pop_front();
        if(i>=k)cout<<a[D.front()]<<endl;
    }
    return 0;
}
```

扩展欧拉定理（欧拉降幂公式）

$$n^k \equiv \begin{cases} n^{k \bmod \varphi(p)} & \gcd(n,p)=1 \\ n^{k \bmod \varphi(p)+\varphi(p)} & \gcd(n,p)\neq 1 \land k \geq \varphi(p) \\ n^k & \gcd(n,p)\neq 1 \land k < \varphi(p) \end{cases}$$

最终我们可以将幂降到 $\varphi(p)$ 的级别，使得能够直接使用快速幂解题，复杂度瓶颈在求解欧拉函数 $\mathcal{O}(\sqrt{p})$ 。

```
int phi(int n) { //求解 phi(n)
    int ans = n;
    for (int i = 2; i <= n / i; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
    if (n > 1) { //特判 n 为质数的情况
        ans = ans / n * (n - 1);
    }
    return ans;
}
signed main() {
    string n_, k_;
    int p;
    cin >> n_ >> k_ >> p;

    int n = 0; // 转化并计算 n % p
    for (auto it : n_) {
        n = n * 10 + it - '0';
        n %= p;
    }
    int mul = phi(p), type = 0, k = 0; // 转化 k
    for (auto it : k_) {
        k = k * 10 + it - '0';
        type |= (k >= mul);
        k %= mul;
    }
    if (type) {
        k += mul;
    }
    cout << mypow(n, k, p) << endl;
}
```

int128 输入输出流控制

int128 只在基于 `Lumix` 系统的环境下可用，需要 C++20 。38位精度，除输入输出外与普通数据类型无差别。该封装支持负数读入，需要注意 `write` 函数结尾不输出多余空格与换行。

```
namespace my128 { // 读入优化封装，支持__int128
    using i64 = __int128_t;
    i64 abs(const i64 &x) {
        return x > 0 ? x : -x;
    }
    auto &operator>>(istream &it, i64 &j) {
        string val;
        it >> val;
        reverse(val.begin(), val.end());
        i64 ans = 0;
        bool f = 0;
        char c = val.back();
        val.pop_back();
        for (; c < '0' || c > '9'; c = val.back(), val.pop_back()) {
            if (c == '-') {
                f = 1;
            }
        }
        for (; c >= '0' && c <= '9'; c = val.back(), val.pop_back()) {
            ans = ans * 10 + c - '0';
        }
        j = f ? -ans : ans;
        return it;
    }
    auto &operator<<(ostream &os, const i64 &j) {
        string ans;
        function<void(i64)> write = [&](i64 x) {
            if (x < 0) ans += '-', x = -x;
            if (x > 9) write(x / 10);
            ans += x % 10 + '0';
        };
        write(j);
        return os << ans;
    }
} // namespace my128
```

对拍版子

- 文件控制

```cpp
// BAD.cpp, 存放待寻找错误的代码
freopen("A.txt","r",stdin);
freopen("BAD.out","w",stdout);

// 1.cpp, 存放暴力或正确的代码
freopen("A.txt","r",stdin);
freopen("1.out","w",stdout);

// Ask.cpp
freopen("A.txt", "w", stdout);
```

- C++版 bat

```cpp
int main() {
    int T = 1E5;
    while(T--) {
        system("BAD.exe");
        system("1.exe");
        system("A.exe");
        if (system("fc BAD.out 1.out")) {
            puts("WA");
            return 0;
        }
    }
}
```

随机数生成与样例构造

```cpp
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
int r(int a, int b) {
    return rnd() % (b - a + 1) + a;
}

void graph(int n, int root = -1, int m = -1) {
    vector<pair<int, int>> t;
    for (int i = 1; i < n; i++) { // 先建立一棵以0为根节点的树
        t.emplace_back(i, r(0, i - 1));
    }

    vector<pair<int, int>> edge;
    set<pair<int, int>> uni;
    if (root == -1) root = r(0, n - 1); // 确定根节点
    for (auto [x, y] : t) { // 偏移建树
        x = (x + root) % n + 1;
        y = (y + root) % n + 1;
        edge.emplace_back(x, y);
        uni.emplace(x, y);
    }

    if (m != -1) { // 如果是图，则在树的基础上继续加边
        for (int i = n; i <= m; i++) {
            while (true) {
                int x = r(1, n), y = r(1, n);
```

```
                if (x == y) continue; // 拒绝自环
                if (uni.count({x, y})) continue; // 拒绝重边
                edge.emplace_back(x, y);
                uni.emplace(x, y);
            }
        }
    }

    random_shuffle(edge.begin(), edge.end()); // 打乱节点
    for (auto [x, y] : edge) {
        cout << x << " " << y << endl;
    }
}
```

OJ测试

对于一个未知属性的OJ，应当在正式赛前进行以下全部测试：

GNU C++ 版本测试

```
for (int i : {1, 2}) {} // GNU C++11 支持范围表达式

auto cc = [&](int x) { x++; }; // GNU C++11 支持 auto 与 lambda 表达式
cc(2);

tuple<string, int, int> V; // GNU C++11 引入
array<int, 3> C; // GNU C++11 引入

auto dfs = [&](auto self, int x) -> void { // GNU C++14 支持 auto 自递归
    if (x > 10) return;
    self(self, x + 1);
};
dfs(dfs, 1);

vector in(1, vector<int>(1)); // GNU C++17 支持 vector 模板类型缺失

map<int, int> dic;
for (auto [u, v] : dic) {} // GNU C++17 支持 auto 解绑
dic.contains(12); // GNU C++20 支持 contains 函数

constexpr double Pi = numbers::pi; // C++20 支持
```

编译器位数测试

```
using i64 = __int128; // 64 位 GNU C++11 支持
```

评测器环境测试

Windows 系统输出 -1；反之则为一个随机数。

```cpp
#define int long long
map<int, int> dic;
int x = dic.size() - 1;
cout << x << endl;
```