cldelve的算法笔记

算法目录

```
/*
==========知识点目录 =============

一、基础工具与宏定义
    - 调试与输入输出宏
    - 循环控制宏
    - 内存与容器操作宏
    - 数据类型别名
    - 基础主函数示例

二、前缀和与差分
    - 前缀和
    - 差分

三、查找算法
    - 二分查找

四、排序算法
    - 归并排序（含逆序对统计）
    - 计数排序
    - 快速排序
    - 快速幂

五、动态规划
    - 最长不下降子序列（LIS）
    - 最长公共子序列（LCS）

六、数据结构
    - 树状数组（BIT）
        - 单点修改 + 区间查询
        - 区间修改 + 单点查询
        - 区间修改 + 区间查询
    - 线段树
        - 单点修改 + 区间查询
        - 区间修改 + 区间查询（带延迟标记）

七、图论
    - 图的存储（邻接表）
    - 最短路算法
        - BFS最短路
        - Floyd算法
        - Dijkstra算法
        - SPFA算法
    - 最小生成树（Kruskal算法）

八、数论
    - 线性筛（欧拉筛）
    - 质因数分解
    - 中国剩余定理（CRT）
    - 最大公约数（GCD）
    - 欧拉函数（批量计算/单个计算）
```

```
    - 矩阵乘法与快速幂
    - 组合数计算
        - Lucas定理
        - 预处理阶乘+逆元

九、计算几何
    - 基础元素定义
    - 点（Point）结构体及运算（加减、点积、叉积、距离等）
    - 线段（Seg）结构体及运算（方向向量、长度、交点判断等）
    - 圆（Cir）结构体及运算（圆心、半径、圆上点计算等）
    - 点与基本图形关系
    - 点到点距离（含平方距离，不开方优化）
    - 点到直线 / 线段的距离计算
    - 点到直线的垂足（投影点）计算
    - 点是否在线段上的判断（含端点）
    - 线段与线段关系
    - 两线段是否相交的判断（含端点重合、点在线段上）
    - 两线段所在直线的交点计算
    - 圆相关计算
    - 两圆位置关系判断（返回公共切线数量：内含 / 内切 / 相交 / 外切 / 外离）
    - 两圆交点坐标计算
    - 直线与圆的交点计算及数量判断（相离 / 相切 / 相交）
    - 点到圆的切线切点计算（点在圆内 / 上 / 外的处理）
    - 两圆公切线的切点计算（针对第一个圆）
    - 三角形内切圆与外接圆计算（圆心与半径）
    - 平面点集操作
    - 平面最近点对距离（分治法实现）
    - 最小圆覆盖（随机增量法，覆盖所有点的最小圆）
    - 多边形计算
    - 多边形面积计算（凹凸多边形通用）
    - 多边形重心计算（凹凸多边形通用）
    - 点是否在多边形内的判断（含边界、内部、外部）
    - 凸包构造（Andrew 算法，返回凸包顶点）
    - 凸多边形的凸性判断
    - 凸多边形被直线切割的面积计算
    - 特殊算法与定理
    - 旋转卡壳求凸包直径（凸多边形最远点对距离）
    - Pick 定理（格点多边形面积与内点、边上点数量关系）

十、字符串
    - 马拉车算法（Manacher）：线性时间求解最长回文子串，统一处理奇/偶长度回文
    - Z函数：计算字符串每个位置与前缀的最长公共前缀长度，用于模式匹配
    - 后缀数组（SuffixArray）：构建字符串所有后缀的排序数组，支持（LCP）和（LCS）查询
    - 后缀自动机（SAM）：压缩存储字符串所有子串，支持子串存在性、出现次数等查询
    - 回文自动机（PAM）：高效处理字符串中所有回文子串，支持回文计数与最长回文查询
    - AC自动机（AhoCorasick）：多模式串匹配算法，快速在文本中查找多个模式串
    - 字符字典树（CharTrie）：基于字符的前缀树，用于前缀匹配、字符串计数等
    - 二进制字典树（BinTrie）：处理二进制整数的前缀树，用于异或最值、区间异或查询等
    - KMP算法（前缀函数）：单模式串匹配，通过前缀函数优化匹配效率
    - 字符串哈希：通过哈希值快速比较子串，支持随机底模降低冲突概率


*/

```

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

// 基础宏定义
#define dbg(x) cout << #x << ":" << x << " "
#define arrout(a,n) for(int i=1;i<=n;i++) cout << a[i] << " "
#define arrin(a,n) for(int i=1;i<=n;i++) cin >> a[i]
#define rep(i,x,n) for(int i=x;i<=n;i++)
#define dep(i,x,n) for(int i=x;i>=n;i--)
#define mem(a,x) memset(a,x,sizeof(a))
#define all(x) x.begin(),x.end()
#define arrall(a,n) a+1,a+1+n
#define PII pair<int,int>
#define m_p make_pair
#define ff first
#define ss second
#define CD const double
#define CI const int
#define int long long
#define itn int

// 基础模板主函数示例
signed main_basic() {
    int a[10];
    arrin(a,5);
    arrout(a,5);
    return 0;
}
```

```cpp
// ============== 二、前缀和与差分 ==============
void prefix_sum_demo() {
    const int N = 1e6 + 5;
    int n, m;
    int a[N], b[N];

    ios::sync_with_stdio(false);
    cin >> n >> m;
    rep(i,1,n) {
        cin >> a[i];
        b[i] = a[i] + b[i-1];
    }
    rep(i,1,m) {
        int x, y;
        cin >> x >> y;
        cout << b[y] - b[x-1] << endl;
    }
}

void difference_demo() {
    const int N = 1e6 + 5;
    int a[N], b[N];
    int n, m, l, r, c, x = 0;

    cin >> n >> m;
    rep(i,1,n) {
        cin >> a[i];
        b[i] = a[i] - a[i-1];
    }
    while (m--) {
        cin >> l >> r >> c;
        b[l] += c;
        b[r+1] -= c;
    }
    rep(i,1,n) {
        x += b[i];
        cout << x << " ";
    }
}
```

```cpp
// ========== 三、查找算法 ==========
void binary_search_demo() {
    const int N = 1e5 + 5;
    int n, a[N], x, ans = 0;

    ios::sync_with_stdio(0);
    cin >> n;
    rep(i,1,n) cin >> a[i];
    cin >> x;

    int l = 1, r = n;
    while (l <= r) {
        int mid = (l + r) >> 1;
        if (a[mid] == x) {
            ans = mid;
            r = mid - 1;
        } else if (a[mid] < x) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }

    cout << (ans ? ans : -1) << endl;
}
```

```cpp
// ============ 四、排序算法 ============
// 1. 归并排序（附带逆序对统计）
// 全局变量暂存归并所需数据（替代lambda捕获）
int *merge_a, *merge_r;
int merge_sum;

// 归并排序核心函数（替代lambda）
void msort_merge(int left, int right) {
    if (left == right) return;
    int mid = (left + right) / 2;
    msort_merge(left, mid);
    msort_merge(mid + 1, right);

    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (merge_a[i] <= merge_a[j]) {
            merge_r[k++] = merge_a[i++];
        } else {
            merge_r[k++] = merge_a[j++];
            merge_sum += mid - i + 1;
        }
    }
    while (i <= mid) merge_r[k++] = merge_a[i++];
    while (j <= right) merge_r[k++] = merge_a[j++];
    rep(i, left, right) merge_a[i] = merge_r[i];
}

void merge_sort_with_inversion() {
    const int N = 1e5 + 5;
    int a[N], r[N];
    int n;
    merge_sum = 0;  // 初始化逆序对计数

    cin >> n;
    rep(i,1,n) cin >> a[i];
    // 绑定全局指针到当前数组
    merge_a = a;
    merge_r = r;
    msort_merge(1, n);  // 调用普通函数而非lambda

    cout << "逆序对数量: " << merge_sum << endl;
}

// 2. 计数排序
void counting_sort_demo() {
    const int N = 1e5 + 5;
    int n, x, a[N] = {0};

    cin >> n;
    rep(i,1,n) {
        cin >> x;
        a[x] = x;
    }
    rep(i,0,x) {
```

```cpp
        if (a[i]) {
            cout << i << " ";
            a[i] = 0;
        }
    }
}

// 3. 快速排序（归并实现版）
int *qsort_a, *qsort_r;

void msort_quick(int left, int right) {
    if (left == right) return;
    int mid = (left + right) >> 1;
    msort_quick(left, mid);
    msort_quick(mid + 1, right);

    int i = left, j = mid + 1, k = left;
    while (i <= mid && j <= right) {
        if (qsort_a[i] <= qsort_a[j]) qsort_r[k++] = qsort_a[i++];
        else qsort_r[k++] = qsort_a[j++];
    }
    while (i <= mid) qsort_r[k++] = qsort_a[i++];
    while (j <= right) qsort_r[k++] = qsort_a[j++];
    rep(i, left, right) qsort_a[i] = qsort_r[i];
}

void quick_sort_demo() {
    const int N = 1e5 + 5;
    int n, a[N], r[N];

    cin >> n;
    rep(i,1,n) cin >> a[i];
    qsort_a = a;
    qsort_r = r;
    msort_quick(1, n);

    arrout(a, n);
}

// 4. 快速幂
int qpow(long long a, long long b, long long q) {
    if (b == 0) return 1 % q;
    long long ans = 1;
    while (b) {
        if (b & 1) ans = ans * a % q;
        a = a * a % q;
        b >>= 1;
    }
    return ans;
}
```

```cpp
// ==========五、动态规划 ===========
// 1. 最长不下降子序列（LIS）
void lis_demo() {
    const int N = 1e4 + 5;
    int n, a[N], dp[N], ans = 0;

    cin >> n;
    rep(i,1,n) {
        cin >> a[i];
        dp[i] = 1;
    }

    rep(i,1,n) {
        rep(j,1,i-1) {
            if (a[i] >= a[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        ans = max(ans, dp[i]);
    }

    cout << "LIS长度: " << ans << endl;
}

// 2. 最长公共子序列（LCS）
void lcs_demo() {
    const int N = 2e3 + 5;
    string a, b;
    int len1, len2, dp[N][N] = {0};

    cin >> a >> b;
    len1 = a.size(), len2 = b.size();

    rep(i,1,len1) {
        rep(j,1,len2) {
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            if (a[i-1] == b[j-1]) {
                dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
            }
        }
    }

    cout << "LCS长度: " << dp[len1][len2] << endl;
}
```

```cpp
// ============ 六、数据结构 ================
// 1. 树状数组（BIT）
// 1.1 单点修改 + 区间查询
int bit_n;  // 树状数组大小（全局暂存）
int *bit_c;

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

void bit_update(int x, int v) {
    for (int i = x; i <= bit_n; i += lowbit(i)) {
        bit_c[i] += v;
    }
}

int bit_get_sum(int l, int r) {
    int ans = 0;
    for (int i = r; i; i -= lowbit(i)) ans += bit_c[i];
    for (int i = l-1; i; i -= lowbit(i)) ans -= bit_c[i];
    return ans;
}

void bit_point_update_range_query() {
    const int N = 1e6 + 5;
    int a[N], c[N];
    int m, x;

    ios::sync_with_stdio(false);
    cin >> bit_n >> m;  // 赋值给全局变量
    bit_c = c;  // 绑定全局指针
    rep(i,1,bit_n) {
        cin >> a[i];
        bit_update(i, a[i]);
    }
    while (m--) {
        int op, k, v;
        cin >> op >> k >> v;
        if (op) bit_update(k, v);
        else cout << bit_get_sum(k, v) << endl;
    }
}

// 1.2 区间修改 + 单点查询
int *br_c, br_n;

void br_update(int k, int v) {
    for (int i = k; i <= br_n; i += lowbit(i)) {
        br_c[i] += v;
    }
}

int br_get_val(int x) {
    int ans = 0;
    for (int i = x; i; i -= lowbit(i)) ans += br_c[i];
    return ans;
```

```cpp
}

void bit_range_update_point_query() {
    const int N = 1e6 + 5;
    int a[N], b[N], c[N];
    int m;

    ios::sync_with_stdio(false);
    cin >> br_n >> m;
    br_c = c;
    rep(i,1,br_n) {
        cin >> a[i];
        b[i] = a[i] - a[i-1];
        br_update(i, b[i]);
    }
    while (m--) {
        int op, x, y, k;
        cin >> op;
        if (op == 1) {
            cin >> x >> y >> k;
            br_update(x, k);
            br_update(y+1, -k);
        } else {
            cin >> x;
            cout << br_get_val(x) << endl;
        }
    }
}

// 1.3 区间修改 + 区间查询
int *rr_b, *rr_c, rr_n;

void rr_update(int x, int y) {
    for (int i = x; i <= rr_n; i += lowbit(i)) {
        rr_b[i] += y;
        rr_c[i] += x * y;
    }
}

int rr_query(int x) {
    int ans = 0;
    for (int i = x; i; i -= lowbit(i)) {
        ans += (x + 1) * rr_b[i] - rr_c[i];
    }
    return ans;
}

int rr_range_query(int l, int r) {
    return rr_query(r) - rr_query(l-1);
}

void rr_range_update(int l, int r, int x) {
    rr_update(l, x);
    rr_update(r+1, -x);
```

```cpp
}

void bit_range_update_range_query() {
    const int N = 1e6 + 5;
    int a[N], b[N], c[N];
    int m;

    ios::sync_with_stdio(false);
    cin >> rr_n >> m;
    rr_b = b;
    rr_c = c;
    rep(i,1,rr_n) {
        cin >> a[i];
        rr_update(i, a[i] - a[i-1]);
    }
    while (m--) {
        int op, x, y, k;
        cin >> op;
        if (op == 1) {
            cin >> x >> y >> k;
            rr_range_update(x, y, k);
        } else {
            cin >> x >> y;
            cout << rr_range_query(x, y) << endl;
        }
    }
}

// 2. 线段树
// 2.1 单点修改 + 区间查询
int *seg_a, *seg_s, seg_n;

void seg_build(int k, int l, int r) {
    if (l == r) {
        seg_s[k] = seg_a[l];
        return;
    }
    int mid = (l + r) >> 1;
    seg_build(2*k, l, mid);
    seg_build(2*k+1, mid+1, r);
    seg_s[k] = seg_s[2*k] + seg_s[2*k+1];
}

void seg_update(int k, int l, int r, int x, int v) {
    if (x < l || x > r) return;
    if (l == r) {
        seg_s[k] += v;
        return;
    }
    int mid = (l + r) >> 1;
    seg_update(2*k, l, mid, x, v);
    seg_update(2*k+1, mid+1, r, x, v);
    seg_s[k] = seg_s[2*k] + seg_s[2*k+1];
}
```

```cpp

int seg_query(int k, int l, int r, int x, int y) {
    if (y < l || x > r) return 0;
    if (x <= l && r <= y) return seg_s[k];
    int mid = (l + r) >> 1;
    return seg_query(2*k, l, mid, x, y) + seg_query(2*k+1, mid+1, r, x, y);
}

void segment_tree_point_update_range_query() {
    const int N = 1e5 + 5;
    int a[N], s[4*N];
    int n;

    ios::sync_with_stdio(false);
    cin >> n;
    seg_n = n;
    seg_a = a;
    seg_s = s;
    rep(i,1,n) cin >> a[i];
    seg_build(1, 1, n);

    string op;
    int x, y;
    while (cin >> op) {
        cin >> x >> y;
        if (op == "Add") seg_update(1, 1, n, x, y);
        else if (op == "Query") cout << seg_query(1, 1, n, x, y) << endl;
    }
}

// 2.2 区间修改 + 区间查询（延迟标记）
int *seg2_a, *seg2_s, *seg2_lazy, seg2_n;

void seg2_build(int k, int l, int r) {
    if (l == r) {
        seg2_s[k] = seg2_a[l];
        return;
    }
    int mid = (l + r) >> 1;
    seg2_build(2*k, l, mid);
    seg2_build(2*k+1, mid+1, r);
    seg2_s[k] = seg2_s[2*k] + seg2_s[2*k+1];
}

void seg2_pushdown(int k, int l, int r) {
    if (seg2_lazy[k] == 0) return;
    int mid = (l + r) >> 1;
    seg2_s[2*k] += (mid - l + 1) * seg2_lazy[k];
    seg2_lazy[2*k] += seg2_lazy[k];
    seg2_s[2*k+1] += (r - mid) * seg2_lazy[k];
    seg2_lazy[2*k+1] += seg2_lazy[k];
    seg2_lazy[k] = 0;
}

```

```cpp
void seg2_update(int k, int l, int r, int x, int y, int v) {
    if (y < l || x > r) return;
    if (x <= l && r <= y) {
        seg2_s[k] += (r - l + 1) * v;
        seg2_lazy[k] += v;
        return;
    }
    seg2_pushdown(k, l, r);
    int mid = (l + r) >> 1;
    seg2_update(2*k, l, mid, x, y, v);
    seg2_update(2*k+1, mid+1, r, x, y, v);
    seg2_s[k] = seg2_s[2*k] + seg2_s[2*k+1];
}

int seg2_query(int k, int l, int r, int x, int y) {
    if (y < l || x > r) return 0;
    if (x <= l && r <= y) return seg2_s[k];
    seg2_pushdown(k, l, r);
    int mid = (l + r) >> 1;
    return seg2_query(2*k, l, mid, x, y) + seg2_query(2*k+1, mid+1, r, x, y);
}

void segment_tree_range_update_range_query() {
    const int N = 1e6 + 5;
    int a[N], s[4*N], lazy[4*N] = {0};
    int n, m;

    ios::sync_with_stdio(false);
    cin >> n >> m;
    seg2_n = n;
    seg2_a = a;
    seg2_s = s;
    seg2_lazy = lazy;
    rep(i,1,n) cin >> a[i];
    seg2_build(1, 1, n);

    char op;
    int x, y, z;
    while (m--) {
        cin >> op;
        if (op == 'Q') {
            cin >> x >> y;
            cout << seg2_query(1, 1, n, x, y) << endl;
        } else {
            cin >> x >> y >> z;
            seg2_update(1, 1, n, x, y, z);
        }
    }
}
```

```cpp
// ==================================== 七、图论 ====================================
// 1. 图的存储（邻接表）
struct Graph {
    const static int N = 1e5 + 5;
    struct Edge { int to, nex; };
    Edge e[N];
    int head[N], tot = 0;

    void add_edge(int x, int y) {
        e[++tot] = {y, head[x]};
        head[x] = tot;
        e[++tot] = {x, head[y]};
        head[y] = tot;
    }

    void traverse(int u) {
        for (int i = head[u]; i; i = e[i].nex) {
            int v = e[i].to;
            cout << v << " ";
        }
    }
};

// 2. 最短路算法
// 2.1 BFS最短路
int bfs_n, bfs_st, bfs_ed;
int bfs_vis[40], bfs_dis[40];
int bfs_e[40][40];

void bfs() {
    mem(bfs_dis, 0x3f);
    queue<int> q;
    q.push(bfs_st);
    bfs_vis[bfs_st] = 1;
    bfs_dis[bfs_st] = 0;

    while (!q.empty()) {
        int x = q.front(); q.pop();
        rep(i,1,bfs_ed) {
            if (bfs_e[x][i] && !bfs_vis[i]) {
                bfs_vis[i] = 1;
                bfs_dis[i] = bfs_dis[x] + 1;
                if (i == bfs_ed) {
                    cout << bfs_dis[i] << endl;
                    return;
                }
                q.push(i);
            }
        }
    }
}

void bfs_shortest_path() {
    int n;
```

```cpp
    cin >> bfs_st >> bfs_ed >> n;
    mem(bfs_e, 0);
    mem(bfs_vis, 0);
    rep(i,1,n) {
        int x, y; cin >> x >> y;
        bfs_e[x][y] = bfs_e[y][x] = 1;
    }
    bfs();  // 调用普通函数而非lambda
}

// 2.2 Floyd算法
int floyd_n;
int floyd_dp[1003][1003];

void floyd() {
    rep(k,1,floyd_n) {
        rep(i,1,floyd_n) {
            rep(j,1,floyd_n) {
                floyd_dp[i][j] = min(floyd_dp[i][j], floyd_dp[i][k] + floyd_dp[k][j]);
            }
        }
    }
}

void floyd_shortest_path() {
    int m, st, ed;
    cin >> floyd_n >> m;
    mem(floyd_dp, 0x3f);
    rep(i,1,floyd_n) floyd_dp[i][i] = 0;

    rep(i,1,m) {
        int x, y; cin >> x >> y;
        floyd_dp[x][y] = floyd_dp[y][x] = 1;
    }
    cin >> st >> ed;
    floyd();
    cout << floyd_dp[st][ed] << endl;
}

// 2.3 Dijkstra算法
struct DijkstraEdge { int nex, to, data; };
DijkstraEdge dijkstra_a[1000005];
int dijkstra_head[1000005], dijkstra_tot;
int dijkstra_dis[1000005];
int dijkstra_n, dijkstra_m, dijkstra_st, dijkstra_ed;

void dijkstra_add(int x, int y, int z) {
    dijkstra_a[++dijkstra_tot] = {dijkstra_head[x], y, z};
    dijkstra_head[x] = dijkstra_tot;
    dijkstra_a[++dijkstra_tot] = {dijkstra_head[y], x, z};
    dijkstra_head[y] = dijkstra_tot;
}

void dijkstra() {
```

```cpp
    mem(dijkstra_dis, 0x3f);
    dijkstra_dis[dijkstra_st] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;
    q.push(m_p(0, dijkstra_st));

    while (!q.empty()) {
        pair<int, int> curr = q.top(); q.pop();
        int d = curr.first, u = curr.second;
        if (d > dijkstra_dis[u]) continue;
        for (int i = dijkstra_head[u]; i; i = dijkstra_a[i].nex) {
            int v = dijkstra_a[i].to, w = dijkstra_a[i].data;
            if (dijkstra_dis[v] > dijkstra_dis[u] + w) {
                dijkstra_dis[v] = dijkstra_dis[u] + w;
                q.push(m_p(dijkstra_dis[v], v));
            }
        }
    }
}

void dijkstra_shortest_path() {
    cin >> dijkstra_n >> dijkstra_m >> dijkstra_st >> dijkstra_ed;
    dijkstra_tot = 0;
    mem(dijkstra_head, 0);
    rep(i,1,dijkstra_m) {
        int x, y, z; cin >> x >> y >> z;
        dijkstra_add(x, y, z);
    }
    dijkstra();
    cout << (dijkstra_dis[dijkstra_ed] == 0x3f3f3f3f ? -1 : dijkstra_dis[dijkstra_ed]) <<
endl;
}

// 2.4 SPFA算法
struct SpfaEdge { int to, data, nex; };
SpfaEdge spfa_e[1000005];
int spfa_head[1000005], spfa_tot;
int spfa_dis[1000005], spfa_cnt[1000005], spfa_vis[1000005];
int spfa_n, spfa_m, spfa_st, spfa_ed;

void spfa_add(int x, int y, int z) {
    spfa_e[++spfa_tot] = {y, z, spfa_head[x]};
    spfa_head[x] = spfa_tot;
}

void spfa() {
    mem(spfa_dis, 0x3f);
    spfa_dis[spfa_st] = 0;
    queue<int> q;
    q.push(spfa_st);
    spfa_vis[spfa_st] = 1;

    while (!q.empty()) {
        int u = q.front(); q.pop();
        spfa_vis[u] = 0;
```

```
861
862         for (int i = spfa_head[u]; i; i = spfa_e[i].nex) {
863             int v = spfa_e[i].to, w = spfa_e[i].data;
864             if (spfa_dis[v] > spfa_dis[u] + w) {
865                 spfa_dis[v] = spfa_dis[u] + w;
866                 spfa_cnt[v] = spfa_cnt[u] + 1;
867                 if (spfa_cnt[v] > spfa_n) {
868                     cout << "有负环" << endl;
869                     return;
870                 }
871                 if (!spfa_vis[v]) {
872                     spfa_vis[v] = 1;
873                     q.push(v);
874                 }
875             }
876         }
877     }
878 }
879
880 void spfa_shortest_path() {
881     cin >> spfa_n >> spfa_m >> spfa_st >> spfa_ed;
882     spfa_tot = 0;
883     mem(spfa_head, 0);
884     mem(spfa_cnt, 0);
885     mem(spfa_vis, 0);
886     rep(i,1,spfa_m) {
887         int x, y, z; cin >> x >> y >> z;
888         spfa_add(x, y, z);
889     }
890     spfa();
891     cout << (spfa_dis[spfa_ed] == 0x3f3f3f3f ? -1 : spfa_dis[spfa_ed]) << endl;
892 }
893
894 // 3. 最小生成树（Kruskal算法）
895 struct KruskalEdge {
896     int from, to, data;
897     bool operator<(const KruskalEdge& b) const {
898         return data < b.data;
899     }
900 } kruskal_e[100005];
901 int kruskal_f[100005];
902 int kruskal_n;
903
904 int kruskal_find(int x) {
905     return kruskal_f[x] == x ? x : kruskal_f[x] = kruskal_find(kruskal_f[x]);
906 }
907
908 void kruskal_mst() {
909     int ans = 0;
910     cin >> kruskal_n;
911     rep(i,1,kruskal_n) kruskal_f[i] = i;
912
913     rep(i,1,kruskal_n) {
914         rep(j,1,kruskal_n) {
```

```
            int x; cin >> x;
            kruskal_e[++kruskal_f[0]] = {i, j, x};
        }
    }

    sort(kruskal_e + 1, kruskal_e + 1 + kruskal_f[0]);
    int cnt = 0;
    rep(i,1,kruskal_f[0]) {
        int u = kruskal_e[i].from, v = kruskal_e[i].to;
        if (kruskal_find(u) != kruskal_find(v)) {
            kruskal_f[kruskal_find(u)] = kruskal_find(v);
            ans += kruskal_e[i].data;
            if (++cnt == kruskal_n-1) break;
        }
    }

    cout << "最小生成树总权值: " << ans << endl;
}
```

```cpp
// ============== 八、数学 ===============
// 1. 线性筛（欧拉筛）求素数
void linear_sieve() {
    const int N = 1e6 + 5;
    int p[N], vis[N] = {0};
    int n, cnt = 0;

    cin >> n;
    rep(i,2,n) {
        if (!vis[i]) p[++cnt] = i;
        rep(j,1,cnt) {
            if (p[j] * i > n) break;
            vis[p[j] * i] = 1;
            if (i % p[j] == 0) break;
        }
    }

    rep(i,1,cnt) cout << p[i] << " ";
}

// 2. 唯一分解定理（质因数分解）
void prime_factorization() {
    int x;
    cin >> x;
    vector<pair<int, int>> factors;

    for (int i = 2; i * i <= x; i++) {
        if (x % i == 0) {
            int cnt = 0;
            while (x % i == 0) {
                x /= i;
                cnt++;
            }
            factors.push_back(m_p(i, cnt));
        }
    }
    if (x > 1) factors.push_back(m_p(x, 1));

    rep(i,0,factors.size()-1) {
        PII p = factors[i];
        cout << p.first << "^" << p.second;
        if (i != factors.size()-1) cout << " * ";
    }
}

// 3. 中国剩余定理（CRT）
int crt_exgcd(int a, int b, int& x, int& y) {
    if (b == 0) { x = 1; y = 0; return a; }
    int ret = crt_exgcd(b, a%b, x, y);
    int t = x; x = y; y = t - (a/b)*y;
    return ret;
}

int crt_demo() {
```

```
    int n;
    cin >> n;
    vector<int> a(n+1), m(n+1);
    rep(i,1,n) cin >> m[i] >> a[i];

    int M = 1, ans = 0;
    rep(i,1,n) M *= m[i];
    rep(i,1,n) {
        int Mi = M / m[i];
        int x, y;
        crt_exgcd(Mi, m[i], x, y);
        ans = (ans + (long long)x * a[i] * Mi) % M;
    }
    if (ans < 0) ans += M;

    return ans;
}

// 4. 最大公约数（GCD）
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// 5. 欧拉函数
// 5.1 批量计算1~n的欧拉函数
void euler_phi_range() {
    int n;
    cin >> n;
    vector<int> phi(n+1);
    rep(i,1,n) phi[i] = i;
    phi[1] = 1;

    rep(i,2,n) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i) {
                phi[j] = phi[j] / i * (i - 1);
            }
        }
    }

    rep(i,1,n) cout << "ϕ(" << i << ")=" << phi[i] << endl;
}

// 5.2 计算单个数字的欧拉函数
int euler_phi_single(int n) {
    int ans = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
```

```cpp
}

// 6. 矩阵乘法与快速幂
const int MOD = 1e9 + 7;
const int MAT_SIZE = 105;

struct Matrix {
    int mat[MAT_SIZE][MAT_SIZE];
    Matrix() { mem(mat, 0); }
    Matrix operator*(const Matrix& b) const {
        Matrix ans;
        rep(i,1,MAT_SIZE) {
            rep(j,1,MAT_SIZE) {
                rep(k,1,MAT_SIZE) {
                    ans.mat[i][j] = (ans.mat[i][j] +
                        (long long)mat[i][k] * b.mat[k][j] % MOD) % MOD;
                }
            }
        }
        return ans;
    }
};

Matrix matrix_qpow(Matrix a, int b) {
    Matrix ans;
    rep(i,1,MAT_SIZE) ans.mat[i][i] = 1;
    while (b) {
        if (b & 1) ans = ans * a;
        a = a * a;
        b >>= 1;
    }
    return ans;
}

// 7. 组合数计算
// 7.1 Lucas定理
int lucas_MOD = 1e9 + 7;

int lucas_C(int a, int b) {
    if (b > a) return 0;
    if (b == 0) return 1;
    long long up = 1, down = 1;
    rep(i,0,b-1) {
        up = up * (a - i) % lucas_MOD;
        down = down * (i + 1) % lucas_MOD;
    }
    return up * qpow(down, lucas_MOD-2, lucas_MOD) % lucas_MOD;
}

int lucas(int a, int b) {
    if (b == 0) return 1;
    return (long long)lucas(a / lucas_MOD, b / lucas_MOD) * lucas_C(a % lucas_MOD, b % lucas_MOD) % lucas_MOD;
}
```

```cpp

int lucas_demo() {
    int n, m;
    cin >> n >> m;
    lucas_MOD = 1e9 + 7;
    return lucas(n, m);
}

// 7.2 预处理阶乘+逆元
// 7.2 预处理阶乘+逆元（移除lambda版本）
// 全局变量用于传递组合数计算所需的阶乘、逆元和模数
const int COMB_N = 1e6 + 5;
int comb_fac[COMB_N];  // 阶乘数组
int comb_inv[COMB_N];  // 逆元数组
int comb_MOD;          // 模数

// 普通函数替代lambda，计算组合数C(n,m)
int comb_C(int n, int m) {
    if (m < 0 || m > n) return 0;
    return (long long)comb_fac[n] * comb_inv[m] % comb_MOD * comb_inv[n - m] % comb_MOD;
}

void comb_preprocess_demo() {
    // 初始化全局变量（替代lambda的[&]捕获）
    comb_MOD = 1e9 + 7;
    // 预处理阶乘
    comb_fac[0] = 1;
    for (int i = 1; i < COMB_N; i++) {
        comb_fac[i] = (long long)comb_fac[i-1] * i % comb_MOD;
    }
    // 预处理逆元（费马小定理）
    comb_inv[COMB_N - 1] = qpow(comb_fac[COMB_N - 1], comb_MOD - 2, comb_MOD);
    for (int i = COMB_N - 2; i >= 0; i--) {
        comb_inv[i] = (long long)comb_inv[i + 1] * (i + 1) % comb_MOD;
    }

    // 处理查询（直接调用普通函数comb_C）
    int q;
    cin >> q;
    while (q--) {
        int n, m;
        cin >> n >> m;
        cout << comb_C(n, m) << endl;
    }
}


/*
    |大数模拟加法|
    |用string模拟|
    |16/11/05ztx, thanks to caojiji|
*/

string add1(string s1, string s2)
```

```
{
    if (s1 == "" && s2 == "")   return "0";
    if (s1 == "")   return s2;
    if (s2 == "")   return s1;
    string maxx = s1, minn = s2;
    if (s1.length() < s2.length()){
        maxx = s2;
        minn = s1;
    }
    int a = maxx.length() - 1, b = minn.length() - 1;
    for (int i = b; i >= 0; --i){
        maxx[a--] += minn[i] - '0'; //  a一直在减 ，  额外还要减个'0'
    }
    for (int i = maxx.length()-1; i > 0;--i){
        if (maxx[i] > '9'){
            maxx[i] -= 10;//注意这个是减10
            maxx[i - 1]++;
        }
    }
    if (maxx[0] > '9'){
        maxx[0] -= 10;
        maxx = '1' + maxx;
    }
    return maxx;
}

/*
    |大数模拟阶乘|
    |用数组模拟|
    |16/12/02ztx|
*/


typedef long long LL;

const int maxn = 100010;

int num[maxn], len;

/*
    在mult函数中，形参部分：len每次调用函数都会发生改变，n表示每次要乘以的数，最终返回的是结果的长度
    tip: 阶乘都是先求之前的(n-1)!来求n!
    初始化Init函数很重要，不要落下
*/

void Init() {
    len = 1;
    num[0] = 1;
}

int mult(int num[], int len, int n) {
    LL tmp = 0;
    for(LL i = 0; i < len; ++i) {
```

```
1237            tmp = tmp + num[i] * n;     //从最低位开始，等号左边的tmp表示当前位，右边的tmp表示进位（之前进的位）
1238            num[i] = tmp % 10; //  保存在对应的数组位置，即去掉进位后的一位数
1239            tmp = tmp / 10;     //  取整用于再次循环,与n和下一个位置的乘积相加
1240        }
1241        while(tmp) {    //  之后的进位处理
1242            num[len++] = tmp % 10;
1243            tmp = tmp / 10;
1244        }
1245        return len;
1246    }
1247
1248    int main() {
1249        Init();
1250        int n;
1251        n = 1977; // 求的阶乘数
1252        for(int i = 2; i <= n; ++i) {
1253            len = mult(num, len, i);
1254        }
1255        for(int i = len - 1; i >= 0; --i)
1256            printf("%d",num[i]);    //  从最高位依次输出,数据比较多采用printf输出
1257        printf("\n");
1258        return 0;
1259    }
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
```

```cpp
// ============== 九、计算几何 ===============
#include <bits/stdc++.h>

const double EPS = 1e-8;

int sign(double val) {
    if (std::abs(val) < EPS) return 0;
    if (val > 0) return 1;
    return -1;
}

int dcmp(double val1, double val2) {
    return sign(val1 - val2);
}

struct Point {
    double x, y;

    Point(double a = 0, double b = 0) : x(a), y(b) {}

    friend Point operator+(Point a, Point b) {
        return Point(a.x + b.x, a.y + b.y);
    }

    friend Point operator-(Point a, Point b) {
        return Point(a.x - b.x, a.y - b.y);
    }

    friend Point operator/(Point a, double b) {
        return Point(a.x / b, a.y / b);
    }

    friend Point operator*(Point a, double b) {
        return Point(a.x * b, a.y * b);
    }

    friend Point operator*(double a, Point b) {
        return Point(b.x * a, b.y * a);
    }

    friend double operator*(Point a, Point b) {
        return a.x * b.x + a.y * b.y;
    }

    friend bool operator==(Point a, Point b) {
        return (!sign(a.x - b.x) && !sign(a.y - b.y));
    }

    friend bool operator!=(Point a, Point b) {
        return !(a == b);
    }

    // 用%代表叉乘,因为 ^ 优先级太低了
    friend double operator%(Point a, Point b) {
```

```
1344            return a.x * b.y - a.y * b.x;
1345        }
1346
1347        double ddis() { return x * x + y * y; }
1348
1349        double dis() { return sqrt(x * x + y * y); }
1350    } INFP(1e20, 1e20);
1351
1352    // 点到点的距离
1353    double dis(Point a, Point b) {
1354        return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
1355    }
1356
1357    // 点到点的距离（不开方）
1358    double ddis(Point a, Point b) {
1359        return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
1360    }
1361
1362    struct Seg {
1363        Point p1, p2;
1364
1365        Seg() {}
1366
1367        Seg(Point a, Point b) : p1(a), p2(b) {}
1368
1369        friend double operator*(Seg a, Seg b) {
1370            return (a.p2 - a.p1) * (b.p2 - b.p1);
1371        }
1372
1373        friend double operator%(Seg a, Seg b) {
1374            return (a.p2 - a.p1) % (b.p2 - b.p1);
1375        }
1376
1377        friend double operator%(Seg a, Point b) {
1378            return (a.p2 - a.p1) % (b - a.p1);
1379        }
1380
1381        friend double operator%(Point a, Seg b) {
1382            return (a - b.p1) % (b.p2 - b.p1);
1383        }
1384
1385        friend bool operator==(Seg a, Seg b) {
1386            return (a.p1 == b.p1 && a.p2 == b.p2) || (a.p1 == b.p2 && a.p2 == b.p1);
1387        }
1388
1389        void swap() {
1390            std::swap(p1, p2);
1391        }
1392
1393        double len() { return dis(p1, p2); }
1394    };
1395
1396    // 点到直线距离
1397    double calPSDis1(Seg a, Point b) {
```

```cpp
    return std::abs((a.p2 - a.p1) % (b - a.p1)) / a.len();
}

// 点到线段距离
double calPSDis2(Seg a, Point b) {
    int res1 = sign((b - a.p1) * (a.p2 - a.p1));
    int res2 = sign((b - a.p2) * (a.p1 - a.p2));
    if (res1 >= 0 && res2 >= 0) {
        return calPSDis1(a, b);
    }
    return std::min(dis(a.p1, b), dis(a.p2, b));
}

// 求点到线段/直线的垂足
Point calPSFeet(Seg a, Point b) {
    auto vec = a.p2 - a.p1;
    return a.p1 + vec * ((b - a.p1) * vec) / (vec.x * vec.x + vec.y * vec.y);
}

// 计算线段及其延长线/直线的交点
Point calSegCross(Seg a, Seg b) {
    auto res1 = (a.p2 - a.p1) % (b.p1 - a.p1);
    auto res2 = (a.p2 - a.p1) % (b.p2 - a.p1);
    auto x = (res2 * b.p1.x - res1 * b.p2.x) / (res2 - res1);
    auto y = (res2 * b.p1.y - res1 * b.p2.y) / (res2 - res1);
    return Point(x, y);
}

// 判断点是否在线段上（包括端点）
bool isPAtS(Seg a, Point b) {
    Point p1 = a.p1 - b, p2 = a.p2 - b;
    if (sign(p1 % p2) == 0 && sign(p1 * p2) <= 0) return true;
    return false;
}

// 判断线段是否相交
bool isSegCross(Seg a, Seg b) {
    double c1 = (a.p2 - a.p1) % (b.p1 - a.p1), c2 = (a.p2 - a.p1) % (b.p2 - a.p1);
    double c3 = (b.p2 - b.p1) % (a.p1 - b.p1), c4 = (b.p2 - b.p1) % (a.p2 - b.p1);

    // 允许线段在端点处相交则添加
    if (!sign(c1) || !sign(c2) || !sign(c3) || !sign(c4)) {
        bool f1 = isPAtS(a, b.p1);
        bool f2 = isPAtS(a, b.p2);
        bool f3 = isPAtS(b, a.p1);
        bool f4 = isPAtS(b, a.p2);
        bool f = (f1 | f2 | f3 | f4);
        return f;
    }
    return (sign(c1) * sign(c2) < 0 && sign(c3) * sign(c4) < 0);
}

// 计算平面最近点对欧式距离
double calPointMinDis(int l, int r, Point point[], int t[]) {
```

```
1452        /*
1453        开始递归前先按照 x 排序
1454        std::sort(point, point + n, [&](const Point &p1, const Point &p2) {
1455            return p1.x < p2.x;
1456        });
1457        */
1458        if (r - l == 0)
1459            return 1e15;
1460        if (r - l == 1)  // 如果递归完后直接输出距离
1461            return dis(point[l], point[r]);
1462        int mid = (l + r) >> 1;
1463        double ans = std::min(calPointMinDis(l, mid, point, t),
1464                              calPointMinDis(mid + 1, r, point, t));
1465        int cnt = 0;
1466        for (int i = l; i <= r; i++)
1467            // 还有一种情况是距离最小的两点刚好分在 mid 两端 ans 距离内的点
1468            if (point[i].x >= point[mid].x - ans && point[i].x <= point[mid].x + ans)
1469                t[++cnt] = i;
1470        std::sort(t + 1, t + cnt + 1, [&](int i, int j) {
1471            return point[i].y < point[j].y;
1472        }); // 以 y 坐标大小排序
1473        for (int i = 1; i <= cnt; i++)
1474            for (int j = i + 1; j <= cnt; j++) {
1475                if (point[t[j]].y >= point[t[i]].y + ans) break;
1476                // 两个点的垂直距离超过 ans 就不必计算了，显然不可能会成为新的 ans
1477                ans = std::min(ans, dis(point[t[i]], point[t[j]]));
1478            }
1479        return ans;
1480    }
1481    struct Cir {
1482        Point o;
1483        double r;
1484
1485        Cir(Point oo, double rr) : o(oo), r(rr) {}
1486
1487        Cir(double x = 0, double y = 0, double rr = 0) { o.x = x, o.y = y, r = rr; }
1488
1489        Point getPoint(double theta) { // 根据极角返回圆上一点的坐标
1490            double co = cos(theta) * r;
1491            double si = sin(theta) * r;
1492            if (sign(co) == 0) co = 0.0;
1493            if (sign(si) == 0) si = 0.0;
1494            return Point(o.x + co, o.y + si);
1495        }
1496
1497        Point getPoint(Point p) { // 根据一点返回圆上一点的坐标
1498            auto vec = p - o;
1499            return o + vec * r / vec.dis();
1500        }
1501    };
1502
1503    // 判断圆相交情况
1504    int isCirCross(Cir a, Cir b) {
1505        // 返回值恰好为两个圆的公共切线数量
```

```cpp
    int res = dcmp(dis(a.o, b.o), a.r + b.r);
    if (res > 0) return 4; // 两个圆不相交
    if (res == 0) return 3; // 两个圆外接
    res = dcmp(dis(a.o, b.o), std::abs(a.r - b.r));
    if (res > 0) return 2; // 两个圆相交
    if (res == 0) return 1; // 有一个圆内切另外一个圆
    return 0; // 一个圆在另一个圆内部
}

// 计算两个圆的交点
std::array<Point, 2> calCirCross(Cir c1, Cir c2) {
    Point vec12 = c2.o - c1.o; // 两圆圆心的向量
    double d = vec12.dis();    // 圆心距
    double a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2.0 * c1.r * d));
    // vec12 与（c1 与一个交点）的夹角
    double t = atan2(vec12.y, vec12.x); // vec12 与x轴的夹角
    return {c1.getPoint(t + a), c1.getPoint(t - a)};
}

// 三角形内切圆
Cir calTrangleInnerCir(Point p1, Point p2, Point p3) {
    Cir c;
    auto d1 = dis(p1, p2), d2 = dis(p1, p3), d3 = dis(p2, p3);
    c.r = std::abs((p2.x - p1.x) * (p3.y - p1.y) -
            (p2.y - p1.y) * (p3.x - p1.x)) / (d1 + d2 + d3);
    c.o.x = (d3 * p1.x + d1 * p3.x + d2 * p2.x) / (d1 + d2 + d3);
    c.o.y = (d3 * p1.y + d1 * p3.y + d2 * p2.y) / (d1 + d2 + d3);
    return c;
}

// 三角形外接圆
Cir calTrangleOuterCir(Point p1, Point p2, Point p3) {
    Cir c;
    c.r = dis(p1, p2) * dis(p2, p3) * dis(p1, p3) / std::abs((p2 - p1) % (p3 - p1)) /
2.0;
    auto a11 = (p2.x - p1.x);
    auto a12 = (p2.y - p1.y);
    auto b1 = (p1.x * (p2.x - p1.x) + p1.y * (p2.y - p1.y) + 0.5 * ddis(p1, p2));
    auto a21 = (p3.x - p1.x);
    auto a22 = (p3.y - p1.y);
    auto b2 = (p1.x * (p3.x - p1.x) + p1.y * (p3.y - p1.y) + 0.5 * ddis(p1, p3));
    auto down = (a11 * a22 - a21 * a12);
    c.o.x = (b1 * a22 - b2 * a12) / down;
    c.o.y = (a11 * b2 - b1 * a21) / down;
    return c;
}

// 直线和圆交点个数
int isSegCrossCir(Seg s, Cir c) {
    int res = dcmp(calPSDis1(s, c.o), c.r);
    if (res == 0) return 1; // 外切
    if (res == 1) return 0; // 不相交
    return 2;
}
```

```cpp

// 计算直线与圆交点
std::array<Point, 2> calSegCrossCir(Cir c, Seg s) {
    auto feet = calPSFeet(s, c.o);
    int sta = isSegCrossCir(s, c);
    if (sta == 0) return {INFP, INFP};
    if (sta == 1) return {feet, feet};
    double base = sqrt(c.r * c.r - (feet - c.o).ddis());
    auto vec = s.p2 - s.p1;
    Point e = vec / vec.dis();
    return {feet + e * base, feet - e * base};
}

std::array<Point, 2> calPointCirTangent(Cir C, Point p) {
    auto d = dis(p, C.o);
    int aa = dcmp(d, C.r);
    if (aa < 0) return {INFP, INFP};  // 点在圆内
    else if (aa == 0) return {p, p}; // 点在圆上，该点就是切点

    // 点在圆外，有两个切点
    double base = atan2(p.y - C.o.y, p.x - C.o.x);
    double ang = acos(C.r / d);
    return {C.getPoint(base - ang), C.getPoint(base + ang)};
}

// 计算 c1 与 c2 的所有切线中 c1 的所有切点
std::vector<Point> calCirTangents(Cir c1, Cir c2) {
    int CrossRes = isCirCross(c1, c2);
    std::vector<Point> res;
    auto d = dis(c1.o, c2.o);
    if (CrossRes == 0) return res;
    double base = atan2(c2.o.y - c1.o.y, c2.o.x - c1.o.x);
    // AB 向量的极角，c1 为大圆
    if (CrossRes == 1) { // 内切
        res.push_back(c1.getPoint(base));
        return res;
    }
    double ang1 = acos((c1.r - c2.r) / d); // 外公切线的图
    res.push_back(c1.getPoint(base + ang1));
    res.push_back(c1.getPoint(base - ang1));
    if (CrossRes == 3)
        res.push_back(c1.getPoint(base));
        // 一条内公切线，此时c2上的点为 c2.getPoint(base + pi)
    else if (CrossRes == 4) { // 两条内公切线，对应内公切线的图
        double ang2 = acos((c1.r + c2.r) / d);
        res.push_back(c1.getPoint(base + ang2));
        res.push_back(c1.getPoint(base - ang2));
    }
    return res;
}

// 最小圆覆盖
Cir calMinCoverCir(Point p[], int n) {
    std::random_device rd;
```

```cpp
    std::mt19937 g(rd());
    std::shuffle(p, p + 1 + n, g);
    Cir c(p[0], 0);
    for (int i = 1; i < n; i++) {
        if (dcmp(dis(c.o, p[i]), c.r) > 0) {
            c.r = 0;
            c.o = p[i];
            for (int j = 0; j < i; j++) {
                if (dcmp(dis(c.o, p[j]), c.r) > 0) {
                    c.r = dis(p[j], p[i]) / 2;
                    c.o = {(p[j].x + p[i].x) / 2, (p[j].y + p[i].y) / 2};
                    for (int k = 0; k < j; k++) {
                        if (dcmp(dis(c.o, p[k]), c.r) > 0) {
                            c = calTrangleOuterCir(p[i], p[j], p[k]);
                        }
                    }
                }
            }
        }
    }
    return c;
}

// 多边形的重心(凹凸都可以)
Point calPolygonCentre(Point point[], int n) {
    double sum = 0.0, sumx = 0, sumy = 0;
    Point p1 = point[0], p2 = point[1], p3;
    for (int i = 2; i <= n - 1; i++) {
        p3 = point[i];
        double area = (p2 - p1) % (p3 - p2) / 2.0;
        sum += area;
        sumx += (p1.x + p2.x + p3.x) * area;
        sumy += (p1.y + p2.y + p3.y) * area;
        p2 = p3;
    }
    return Point(sumx / (3.0 * sum), sumy / (3.0 * sum));
}

// 判断点是否在多边形内部(凹凸都可以)
int isPointInPolygon(int n, Point point[], Point P) {
    int cnt = 0;
    Point P1, P2;
    for (int i = 0; i < n; i++) {
        P1 = point[i];
        P2 = point[(i + 1) % n];
        if (isPAtS(Seg(P1, P2), P)) return 1;  // 如果点在边上返回 1
        if ((sign(P1.y - P.y) > 0 != sign(P2.y - P.y) > 0) &&
            sign(P.x - (P.y - P1.y) * (P1.x - P2.x) / (P1.y - P2.y) - P1.x) < 0)
            cnt++;
    }
    if (cnt & 1) return 2; // 在内部返回 2
    else return 0; // 在外部返回 0
}

```

```cpp
// 多边形面积(凹凸都可以)
double calPolygonArea(Point point[], int n) {
    double area = 0;
    for (int i = 0; i < n; i++)
        area += point[i] % point[(i + 1) % n];
    return std::abs(area) / 2.0;
}

// 计算凸多边形被直线 s 切割的面积
double calPolygonAreaCut(Point point[], int n, Seg s) {
    Point dir = s.p2 - s.p1, tmp[110];
    // 默认点的顺序为为逆时针
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        int res1 = sign((point[i] - s.p1) % dir);
        int res2 = sign((point[(i + 1) % n] - s.p1) % dir);
        if (res1 <= 0) tmp[cnt++] = point[i];
        // 求左侧面积，如果求右侧改成 >= 0
        if (res1 * res2 == -1) {
            tmp[cnt++] = calSegCross(s, Seg(point[i], point[(i + 1) % n]));
        }
    }
    return calPolygonArea(tmp, cnt);
}

// Pick 定理：给定顶点均为整点的简单多边形，皮克定理说明了
// 其面积 A 和内部格点数目 i,边上格点数目 b 的关系：
// A = i + b / 2 - 1（没有取整,可能是小数）
// 取格点的组成图形的面积为一单位.在平行四边形格点,皮克定理依然成立。
// 套用于任意三角形格点，皮克定理则是 A = 2 * i + b - 2
std::array<int, 2> pick(int n, Point point[]) {

    int sum = 0, num = 0;
    for (int i = 0; i < n; i++) {
        int x = std::abs(point[(i + 1) % n].x - point[i].x) + 0.5;
        int y = std::abs(point[(i + 1) % n].y - point[i].y) + 0.5;
        num += std::__gcd(abs(x), abs(y));
        sum += point[(i + 1) % n] % point[i];
    }
    sum = abs(sum);// sum 为两倍的面积
    return {(sum - num) / 2 + 1, num};
}

// andrew 找凸包
int andrew(int n, Point point[], Point res[]) {
    // n >= 2
    // point 和 res 下标从 0 开始,方便取模
    std::sort(point, point + n, [&](const Point &a, const Point &b) {
        if (dcmp(a.x, b.x) == 0) return a.y < b.y;
        return a.x < b.x;
    });
    int m = 0;
    for (int i = 0; i < n; i++) {
        while (m > 1 && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;
```

```cpp
        // 如果想保留凸包边上的点,可以把 <= 换成 <
        // 凸包为逆时针方向
        res[m++] = point[i];
    }
    int k = m;
    for (int i = n - 2; i >= 0; i--) {
        while (m > k && (res[m - 1] - res[m - 2]) % (point[i] - res[m - 2]) <= 0) --m;
        res[m++] = point[i];
    }
    --m;
    // point[0] 被重复记录了,删去
    return m;
}

// 点c到线段ab的距离
double segdis(Point a, Point b, Point c) {
    return std::abs((b - a) % (c - a)) / dis(a, b);
}

// 旋转卡壳求凸包直径
double rotate(int n, Point res[]) {
    // 虽然 n >= 2 时即认为凸包存在，但是旋转卡壳的时候要求 n >= 3
    if (n == 2) return dis(res[1], res[0]);
    int cur = 0;
    double ans = 0;
    for (int i = 0; i < n; i++) {
        while (segdis(res[i], res[(i + 1) % n], res[cur]) <=
               segdis(res[i], res[(i + 1) % n], res[(cur + 1) % n]))
            cur = (cur + 1) % n;
        ans = std::max(ans, dis(res[i], res[cur]));
        ans = std::max(ans, dis(res[(i + 1) % n], res[cur]));
    }
    return ans;
}

// 判断凸包
bool isConvex(int n, Point point[]) {
    // 默认为顺时针
    // 如果输入点的方式为逆时针输入,则将 < 改成 >
    // 认为凸包边上可以有 点 则加上 =
    for (int i = 0; i < n; i++) {
        if ((point[i] - point[(i + 1) % n]) % (point[(i + 2) % n] - point[(i + 1) % n]) > 0)
            return false;
    }
    return true;
}
```

```cpp
// ============== 十、字符串 ===============
using i64 = long long;
constexpr int INF = 1e9;
constexpr int ALPHABET = 26;

// 1. 马拉车算法：求最长回文子串
vector<int> manacher(string s) {
    string t = "#";
    for (char c : s) { t += c; t += '#'; }
    int n = t.size(), j = 0;
    vector<int> r(n);
    for (int i = 0; i < n; i++) {
        if (2 * j - i >= 0 && j + r[j] > i)
            r[i] = min(r[2 * j - i], j + r[j] - i);
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]])
            r[i]++;
        if (i + r[i] > j + r[j]) j = i;
    }
    return r;
}

// 2. Z函数：求每个位置最长前缀匹配
vector<int> z_function(string s) {
    int n = s.size(), j = 0;
    vector<int> z(n); z[0] = n;
    for (int i = 1; i < n; i++) {
        z[i] = max((long long)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;
}

// 3. 后缀数组：含LCP/LCS查询
struct SuffixArray {
    int n;
    vector<int> sa, rk, lc;
    SuffixArray(string s) : n(s.size()), sa(n), rk(n), lc(n-1) {
        iota(sa.begin(), sa.end(), 0);
        sort(sa.begin(), sa.end(), [&](int a, int b) { return s[a] < s[b]; });
        rk[sa[0]] = 0;
        for (int i = 1; i < n; i++)
            rk[sa[i]] = rk[sa[i-1]] + (s[sa[i]] != s[sa[i-1]]);
        int k = 1; vector<int> tmp, cnt(n); tmp.reserve(n);
        while (rk[sa[n-1]] < n-1) {
            tmp.clear();
            for (int i = 0; i < k; i++) tmp.push_back(n - k + i);
            for (int i : sa) if (i >= k) tmp.push_back(i - k);
            fill(cnt.begin(), cnt.end(), 0);
            for (int i = 0; i < n; i++) cnt[rk[i]]++;
            for (int i = 1; i < n; i++) cnt[i] += cnt[i-1];
            for (int i = n-1; i >= 0; i--) sa[--cnt[rk[tmp[i]]]] = tmp[i];
            swap(rk, tmp); rk[sa[0]] = 0;
```

```
            for (int i = 1; i < n; i++) {
                bool diff = (tmp[sa[i-1]] != tmp[sa[i]]) || (sa[i-1]+k >= n) || (sa[i]+k >= n) || (tmp[sa[i-1]+k] != tmp[sa[i]+k]);
                rk[sa[i]] = rk[sa[i-1]] + diff;
            }
            k *= 2;
        }
        for (int i = 0, j = 0; i < n; i++) {
            if (rk[i] == 0) j = 0;
            else {
                j -= j > 0;
                while (i + j < n && sa[rk[i]-1] + j < n && s[i+j] == s[sa[rk[i]-1]+j]) j++;
                lc[rk[i]-1] = j;
            }
        }
    }
};

// 4. 后缀自动机：压缩存储子串
struct SAM {
    struct Node { int len, link; array<int, ALPHABET> next; Node() : len(0), link(0), next{} {} };
    vector<Node> t;
    SAM() { t.assign(2, Node()); t[0].next.fill(1); t[0].len = -1; }
    int newNode() { t.emplace_back(); return t.size()-1; }
    int extend(int p, int c) {
        if (t[p].next[c]) {
            int q = t[p].next[c];
            if (t[q].len == t[p].len + 1) return q;
            int r = newNode(); t[r].len = t[p].len + 1;
            t[r].link = t[q].link; t[r].next = t[q].next;
            t[q].link = r;
            while (t[p].next[c] == q) { t[p].next[c] = r; p = t[p].link; }
            return r;
        }
        int cur = newNode(); t[cur].len = t[p].len + 1;
        while (!t[p].next[c]) { t[p].next[c] = cur; p = t[p].link; }
        t[cur].link = extend(p, c);
        return cur;
    }
    int extend(int p, char c) { return extend(p, (long long)(c - 'a')); }
};

// 5. 回文自动机：处理回文子串
struct PAM {
    struct Node { int len, link, cnt; array<int, ALPHABET> next; Node() : len(0), link(0), cnt(0), next{} {} };
    vector<Node> t; int suff; string s;
    PAM() { t.assign(2, Node()); t[0].link = 1; t[1].len = -1; suff = 1; s.clear(); }
    int newNode() { t.emplace_back(); return t.size()-1; }
    bool add(char c) {
        int pos = s.size(); s += c; int let = c - 'a', cur = suff;
        while (true) {
```

```
            int curlen = t[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos-1-curlen] == s[pos]) break;
            cur = t[cur].link;
        }
        if (t[cur].next[let]) { suff = t[cur].next[let]; return false; }
        int num = newNode(); suff = num; t[num].len = t[cur].len + 2; t[cur].next[let] = num;
        if (t[num].len == 1) { t[num].link = 1; t[num].cnt = 1; return true; }
        while (true) {
            cur = t[cur].link; int curlen = t[cur].len;
            if (pos - 1 - curlen >= 0 && s[pos-1-curlen] == s[pos]) {
                t[num].link = t[cur].next[let]; break;
            }
        }
        t[num].cnt = 1 + t[t[num].link].cnt; return true;
    }
};

// 6. AC自动机：多模式匹配
struct AhoCorasick {
    struct Node { int len, link; array<int, ALPHABET> next; Node() : len(0), link(0), next{} {} };
    vector<Node> t;
    AhoCorasick() { t.assign(2, Node()); t[0].next.fill(1); t[0].len = -1; }
    int newNode() { t.emplace_back(); return t.size()-1; }
    int add(const string& a) {
        int p = 1;
        for (char c : a) {
            int x = c - 'a';
            if (!t[p].next[x]) { t[p].next[x] = newNode(); t[t[p].next[x]].len = t[p].len + 1; }
            p = t[p].next[x];
        }
        return p;
    }
    void work() {
        queue<int> q; q.push(1);
        while (!q.empty()) {
            int x = q.front(); q.pop();
            for (int i = 0; i < ALPHABET; i++) {
                if (!t[x].next[i]) t[x].next[i] = t[t[x].link].next[i];
                else { t[t[x].next[i]].link = t[t[x].link].next[i]; q.push(t[x].next[i]); }
            }
        }
    }
};

// 7. 字符字典树：前缀匹配
struct CharTrie {
    static constexpr int MAX = 1e6 + 10;
    int trie[MAX][ALPHABET], val[MAX], tot;
    CharTrie() { tot = 0; newNode(); }
```

```
1927     int newNode() { tot++; memset(trie[tot], 0, sizeof(trie[tot])); val[tot] = 0; return
tot; }
1928     void insert(const string& s) {
1929         int p = 1;
1930         for (char c : s) {
1931             int x = c - 'a';
1932             if (!trie[p][x]) trie[p][x] = newNode();
1933             p = trie[p][x]; val[p]++;
1934         }
1935     }
1936     int query(const string& s) {
1937         int p = 1;
1938         for (char c : s) {
1939             int x = c - 'a';
1940             if (!trie[p][x]) return 0;
1941             p = trie[p][x];
1942         }
1943         return val[p];
1944     }
1945 };
1946
1947 // 8. 二进制字典树：异或最值
1948 struct BinTrie {
1949     static constexpr int MAX = 1e7 + 10, BITS = 30;
1950     int trie[MAX][2], val[MAX], tot;
1951     BinTrie() { tot = 0; newNode(); }
1952     int newNode() { tot++; memset(trie[tot], 0, sizeof(trie[tot])); val[tot] = INF;
return tot; }
1953     void insert(int x, int i) {
1954         int p = 1;
1955         for (int j = BITS; j >= 0; j--) {
1956             int b = (x >> j) & 1;
1957             if (!trie[p][b]) trie[p][b] = newNode();
1958             p = trie[p][b]; val[p] = min(val[p], i);
1959         }
1960     }
1961     int query_max_xor(int x) {
1962         int p = 1, res = INF;
1963         for (int j = BITS; j >= 0; j--) {
1964             int b = (x >> j) & 1;
1965             p = trie[p][1 - b] ? trie[p][1 - b] : trie[p][b];
1966             res = min(res, val[p]);
1967         }
1968         return res == INF ? -1 : res;
1969     }
1970 };
1971
1972
1973 // 9. KMP前缀函数：单模式匹配
1974 vector<int> kmp_prefix(const string& s) {
1975     int n = s.size(), j = 0;
1976     vector<int> f(n + 1);
1977     for (int i = 1; i < n; i++) {
1978         while (j && s[i] != s[j]) j = f[j];
```

```
        if (s[i] == s[j]) j++;
        f[i + 1] = j;
    }
    return f;
}

vector<int> kmp_match(const string& text, const string& pat) {
    vector<int> f = kmp_prefix(pat), res;
    int j = 0;
    for (int i = 0; i < text.size(); i++) {
        while (j && text[i] != pat[j]) j = f[j];
        if (text[i] == pat[j]) j++;
        if (j == pat.size()) { res.push_back(i - j + 1); j = f[j]; }
    }
    return res;
}
```