

# 代码模板

zdq

SCUT

March 26, 2021

# Contents

<b>开始</b>	<b>2</b>
宏定义 . . . . .	2
快读 . . . . .	2
对拍 . . . . .	2
<b>数学</b>	<b>3</b>
模乘模幂 . . . . .	3
GCD . . . . .	3
CRT . . . . .	4
线性筛 . . . . .	4
$\phi$ 单点欧拉函数 . . . . .	5
Miller-Rabin 素性测试 . . . . .	5
Pollard-Rho 分解质因数 . . . . .	5
组合数 . . . . .	6
exLucas . . . . .	6
<b>二维计算几何</b>	<b>7</b>
点向量基本运算 . . . . .	7
位置关系 . . . . .	8
多边形 . . . . .	8
求多边形面积 . . . . .	8
判断点是否在多边形内 . . . . .	8
凸包 . . . . .	9
凸包直径·平面最远点对 . . . . .	9
平面最近点对 . . . . .	10
圆 . . . . .	10
三点垂心 . . . . .	10
最小覆盖圆 . . . . .	11
<b>图论</b>	<b>11</b>
存图 . . . . .	11
最短路 . . . . .	11
Dijkstra . . . . .	11
LCA . . . . .	12
连通性 . . . . .	12
有向图强联通分量 . . . . .	12

# 开始

## 宏定义

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  typedef long long LL;
4  typedef __int128 LLL;
5  typedef unsigned u32;
6  typedef unsigned long long u64;
7  typedef long double LD;
8  #define il inline
9  #define pln putchar('\n')
10 #define For(i,a,b) for(int i=(a),(i##i)=(b);i<=(i##i);++i)
11 #define Rep(i,n) for(int i=0,(i##i)=(n);i<(i##i);++i)
12 #define Fodn(i,a,b) for(int i=(a),(i##i)=(b);i>=(i##i);--i)
13 const int M=1000000007,INF=0x3f3f3f3f;
14 const long long INFLL=0x3f3f3f3f3f3f3f3fLL;
15 const int N=1000010;
```

## 快读

```
1  ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);

1  template <typename T>
2  inline bool read(T &x) {
3      x = 0; char c = getchar(); int f = 1;
4      while (!isdigit(c) && (c != '-') && (c != EOF)) c = getchar();
5      if (c == EOF) return 0;
6      if (c == '-') f = -1, c = getchar();
7      while (isdigit(c)) { x = x * 10 + (c & 15); c = getchar();}
8      x *= f; return 1;
9  }

10
11 template <typename T, typename... Args>
12 inline bool read(T &x, Args &...args) {
13     bool res = 1;
14     res &= read(x);
15     res &= read(args...);
16     return res;
17 }
```

## 对拍

```
1  //in.txt
2  //AC.exe std.txt
3  //MY.exe my.txt
4
5  void init(){
6      FILE*F=fopen("int.txt","w");
7
8      //srand(time(0));
9      //int a=(long long)rand()*rand()%1001;
10     //fscanf(F,"%d",&a);fprintf(F,"%d\n",a);
11
12     fclose(F);
```

```

13 }
14
15 int main(){
16     init();
17     while(1){
18         system("AC.exe < in.txt > std.txt");
19
20         system("MY.exe < in.txt > my.txt");
21
22         if(system("fc std.txt my.txt")){
23             puts("WA");
24             return 0;
25         }else puts("AC\n\n");
26
27         init();
28     }
29 }

```

## 数学

### 模乘模幂

- longlong 范围用 fpl

```

1 inline LL mul(LL a, LL b, LL p) {
2     LL res = a * b - ((LL)((LD)a * b / p) * p);
3     return res < 0 ? res + p : (res < p ? res : res - p);
4 }
5
6 inline LL fp(LL a, LL b, LL Mod) {
7     LL res = (Mod != 1);
8     for (; b >>= 1, a = a * a % Mod)
9         if (b & 1)
10             res = res * a % Mod;
11     return res;
12 }
13
14 inline LL fpl(LL a, LL b, LL Mod) {
15     LL res = (Mod != 1);
16     for (; b >>= 1, a = mul(a, a, Mod))
17         if (b & 1)
18             res = mul(res, a, Mod);
19     return res;
20 }

```

### GCD

```

1 template <typename T>
2 inline T gcd(T a, T b) {
3     while (b){
4         T t = b;
5         b = a % b;
6         a = t;
7     }
8     return a;

```

```

9  }
10
11  template <typename T>
12  inline T lcm(T a, T b) { return a / gcd(a, b) * b; }
13
14  template <typename T>
15  T exgcd(T a, T b, T &x, T &y) {
16      if (!b) {
17          x = 1;
18          y = 0;
19          return a;
20      }
21      T res = exgcd(b, a % b, x, y);
22      T t = x;
23      x = y;
24      y = t - a / b * y;
25      return res;
26  }

```

## CRT

- 需要 GCD 64 位模乘
- 用来合并同余方程
- 返回最小正数解或最小非负解无解返回-1

```

1  inline LL Crt(LL a1, LL a2, LL mod1, LL mod2) {
2      LL u, v;
3      LL g = exgcd(mod1, mod2, u, v);
4      if ((a2 - a1) % g) return -1;
5      LL m12 = abs(lcm(mod1, mod2));
6      LL res = (mul(mod1, mul(u, ((a2 - a1) / g), m12), m12) + a1) % m12;
7      return res <= 0 ? res + m12 : res; /* 求最小正数解还是非负解 */
8  }

```

## 线性筛

```

1  struct primenumberlist{
2      #define MAXN (100000000)
3      int cnt, pri[100000000];
4      bool np[MAXN + 10];
5      primenumberlist(){
6          np[1] = 1; cnt = 0;
7          for (int i = 2; i <= MAXN; ++i) {
8              if (!np[i]) pri[++cnt] = i;
9              for (int j = 1; j <= cnt; ++j) {
10                 LL t = pri[j] * i;
11                 if (t > MAXN) break;
12                 np[t] = 1;
13                 if (!(i % pri[j])) break;
14             }
15         }
16     }
17 } prime;

```

## $\phi$ 单点欧拉函数

```
1 template <typename T>
2 inline T phi(T x) {
3     T res = x;
4     for (T i = 2; i * i <= x; ++i)
5         if ((x % i) == 0) {
6             res = res / i * (i - 1);
7             while ((x % i) == 0) x /= i;
8         }
9     if (x > 1) res = res / x * (x - 1);
10    return res;
11 }
```

## Miller-Rabin 素性测试

- $n \leq 10^{18}$
- 需要 64 位模乘 64 位模幂

```
1 inline bool MR(LL x, LL n, int t) {
2     LL las = x;
3     for (int i = 1; i <= t; ++i) {
4         x = mul(x, x, n);
5         if (x == 1 && las != 1 && las != (n - 1)) return 0;
6         las = x;
7     }
8     return x == 1;
9 }
10
11 inline bool isPrime(LL n) {
12     if (n == 4685624825598111 || n < 2) return 0;
13     if (n == 2 || n == 3 || n == 7 || n == 61 || n == 24251) return 1;
14     LL d = n - 1;
15     int t = 0;
16     while ((d & 1) == 0) d >>= 1, ++t;
17     return MR(fpl(2, d, n), n, t) && MR(fpl(61, d, n), n, t);
18 }
```

## Pollard-Rho 分解质因数

- 需要 64 位模乘 gcd
- 求  $n$  的一个大于 1 的因子可能返回  $n$  本身
- 调用 PR() 前务必判断  $n$  的素性检查  $n > 1$

```
1 mt19937 mt(time(0)); //随机化
2 inline LL PR(LL n) {
3     LL x = uniform_int_distribution<LL>(0, n - 1)(mt), s, t, c =
4     ↪ uniform_int_distribution<LL>(1, n - 1)(mt); //随机化
5     for (int gol = 1; 1; gol <= 1, s = t, x = 1) {
6         for (int stp = 1; stp <= gol; ++stp) {
7             t = (mul(t, t, n) + c) % n;
8             x = mul(x, abs(s - t), n);
9             if ((stp & 127) == 0) {
10                 LL d = gcd(x, n);
11             }
12         }
13     }
```

```

10         if (d > 1) return d;
11     }
12 }
13 LL d = gcd(x, n);
14 if (d > 1) return d;
15 }
16 }

```

## 组合数

- 数较小模数为较大质数求逆元
  - 如果模数固定可以  $O(n)$  预处理阶乘的逆元
- 数较大模数为较小质数用 *Lucas* 定理
- –

$$C_n^m \equiv C_{\lfloor \frac{n}{p} \rfloor}^{\lfloor \frac{m}{p} \rfloor} * C_{n \bmod p}^{m \bmod p} (\bmod p)$$

- 数较大模数较小用 *exLucas* 定理求  $C_n^m \bmod P$

## exLucas

- 需要模乘 CRT
- $O(P \log P)$
- 不要求  $P$  为质数

```

1 namespace EXLUCAS {
2     inline LL idxp(LL n, LL p) {
3         LL nn = n;
4         while (n > 0) nn -= (n % p), n /= p;
5         return nn / (p - 1);
6     }
7
8     LL facp(LL n, LL p, LL pk) {
9         if (n == 0) return 1;
10        LL res = 1;
11        if (n >= pk) {
12            LL t = n / pk, k = 1, els = n - t * pk;
13            for (LL i = 1; i <= els; ++i) if (i % p) k = k * i % pk;
14            res = k;
15            for (LL i = els + 1; i < pk; ++i) if (i % p) k = k * i % pk;
16            res = res * fp(k, n / pk, pk) % pk;
17        }
18        else for (LL i = 1; i <= n; ++i) if (i % p) res = res * i % pk;
19        return res * facp(n / p, p, pk) % pk;
20    }
21
22    inline LL exlucas(LL n, LL m, LL p, LL pk, LL k) {
23        LL a = facp(n, p, pk) * fp(facp(n - m, p, pk) * facp(m, p, pk) % pk, pk / p * (p - 1)
24        ↪ - 1, pk) % pk;
25        LL b = idxp(n, p) - idxp(m, p) - idxp(n - m, p);
26        if (b >= k) return 0;
27        while (b-- > 0) a *= p;
28    }
29 }

```

```

27     return a % pk;
28 }
29
30 /* 接口 */ inline LL exlucas(LL n, LL m, LL p) {
31     LL a = 0, b = 1;
32     for (LL i = 2; i * i <= p; ++i) {
33         if (p % i) continue;
34         LL t = 0, pk = 1;
35         while (p % i == 0) ++t, p /= i, pk *= i;
36         a = Crt(a, exlucas(n, m, i, pk, t), b, pk);
37         b *= pk;
38     }
39     return (p > 1) ? Crt(a, exlucas(n, m, p, p, 1), b, p) : a;
40 }
41 }

```

## 二维计算几何

- Point 直接支持整型和浮点型
- 部分函数可以对整型改写
- 多边形 (凸包) 按逆时针存在下标 1..n

## 点向量基本运算

```

1  template <typename T>
2  struct Point {
3      T x, y;
4      Point() {}
5      Point(T u, T v) : x(u), y(v) {}
6      Point operator+(const Point &a) const { return Point(x + a.x, y + a.y); }
7      Point operator-(const Point &a) const { return Point(x - a.x, y - a.y); }
8      Point operator*(const T &a) const { return Point(x * a, y * a); }
9      T operator*(const Point &a) const { return x * a.x + y * a.y; }
10     T operator%(const Point &a) const { return x * a.y - y * a.x; }
11     double len() const { return hypot(x, y); }
12     double operator^(const Point &a) const { return (a - (*this)).len(); }
13     double angle() const { return atan2(y, x); }
14     bool id() const { return y < 0 || (y == 0 && x < 0); }
15     bool operator<(const Point &a) const { return id() == a.id() ? (*this) % a > 0 : id() <
        ↪ a.id(); }
16 };
17 typedef Point<double> point;
18
19 #define sqr(x) ((x) * (x))
20 const point O(0, 0);
21 const double PI(acos(-1.0)), EPS(1e-8);
22 inline bool dcmp(const double &x, const double &y) { return fabs(x - y) < EPS; }
23 inline int sgn(const double &x) { return fabs(x) < EPS ? 0 : (x < 0) ? -1 : 1; }
24 inline double mul(point p1, point p2, point p0) { return (p1 - p0) % (p2 - p0); }

```



## 位置关系

```
1 inline bool in_same_seg(point p, point a, point b) {
2     if (fabs(mul(p, a, b)) < EPS) {
3         if (a.x > b.x) swap(a, b);
4         return (a.x <= p.x && p.x <= b.x && ((a.y <= p.y && p.y <= b.y) || (a.y >= p.y && p.y
           ↪ >= b.y)));
5     } else return 0;
6 }
7
8 inline bool is_right(point st, point ed, point a) {
9     return ((ed - st) % (a - st)) < 0;
10 }
11
12 inline point intersection(point s1, point t1, point s2, point t2) {
13     return s1 + (t1 - s1) * (((s1 - s2) % (t2 - s2)) / ((t2 - s2) % (t1 - s1)));
14 }
15
16 inline bool parallel(point a, point b, point c, point d) {
17     return dcmp((b - a) % (d - c), 0);
18 }
19
20 inline double point2line(point p, point s, point t) {
21     return fabs(mul(p, s, t) / (t - s).len());
22 }
23
24 inline double point2seg(point p, point s, point t) {
25     return sgn((t - s) * (p - s)) * sgn((s - t) * (p - t)) > 0 ? point2line(p, s, t) : min((p
           ↪ ^ s), (p ^ t));
26 }
```

## 多边形

### 求多边形面积

```
1 inline double area(int n, point s[]) {
2     double res = 0;
3     s[n + 1] = s[1];
4     for (int i = 1; i <= n; ++i)
5         res += s[i] % s[i + 1];
6     return fabs(res / 2);
7 }
```

### 判断点是否在多边形内

- 特判边上的点
- 使用了  $a[1] \dots a[n+1]$  的数组

```
1 inline bool in_the_area(point p, int n, point area[]) {
2     bool ans = 0; double x;
3     area[n + 1] = area[1];
4     for (int i = 1; i <= n; ++i) {
5         point p1 = area[i], p2 = area[i + 1];
6         if (in_same_seg(p, p1, p2)) return 1; //特判边上的点
7         if (p1.y == p2.y) continue;
8         if (p.y < min(p1.y, p2.y)) continue;
```

```

9         if (p.y >= max(p1.y, p2.y)) continue;
10        ans ^= (((p.y - p1.y) * (p2.x - p1.x) / (p2.y - p1.y) + p1.x) > p.x);
11    }
12    return ans;
13 }

```

## 凸包

- Andrew 算法
- $O(n \log n)$
- 可以应对凸包退化成直线/单点的情况但后续旋转卡壳时应注意特判
- 注意是否应该统计凸包边上的点

```

1 inline bool pcmp1(const point &a, const point &b) { return a.x == b.x ? a.y < b.y : a.x < b.x;
↪ }
2
3 inline int Andrew(int n, point p[], point ans[]) { //ans[] 逆时针存凸包
4     sort(p + 1, p + 1 + n, pcmp1);
5     int m = 0;
6     for (int i = 1; i <= n; ++i) {
7         while (m > 1 && mul(ans[m - 1], ans[m], p[i]) < 0) --m; //特判凸包边上的点
8         ans[++m] = p[i];
9     }
10    int k = m;
11    for (int i = n - 1; i >= 1; --i) {
12        while (m > k && mul(ans[m - 1], ans[m], p[i]) < 0) --m; //特判凸包边上的点
13        ans[++m] = p[i];
14    }
15    return m - (n > 1); //返回凸包有多少个点
16 }

```

## 凸包直径·平面最远点对

- 旋转卡壳算法
- $O(n)$
- 凸包的边上只能有端点，否则不满足严格单峰
- 凸包不能退化成直线，调用前务必检查  $n \geq 3$
- 使用了  $a[1] \dots a[n+1]$  的数组

```

1 inline double Rotating_Caliper(int n, point a[]) {
2     a[n + 1] = a[1];
3     double ans = 0;
4     int j = 2;
5     for (int i = 1; i <= n; ++i) {
6         while (fabs(mul(a[i], a[i + 1], a[j])) < fabs(mul(a[i], a[i + 1], a[j + 1]))) j = (j %
↪ n + 1);
7         ans = max(ans, max((a[j] ^ a[i]), (a[j] ^ a[i + 1])));
8     }
9     return ans;
10 }

```

## 平面最近点对

- 分治 + 归并
- $O(n \log n)$

```
1 namespace find_the_closest_pair_of_points {
2     const int N = 200010; //maxn
3     inline bool cmp1(const point &a, const point &b) { return a.x < b.x || (a.x == b.x && a.y
4         ↪ < b.y); }
5     inline bool operator>(const point &a, const point &b) { return a.y > b.y || (a.y == b.y &&
6         ↪ a.x > b.x); }
7
8     point a[N], b[N];
9     double ans;
10    inline void upd(const point &i, const point &j) { ans = min(ans, i ^ j); }
11
12    void find(int l, int r) {
13        if (l == r) return;
14        if (l + 1 == r) {
15            if (a[l] > a[r]) swap(a[l], a[r]);
16            upd(a[l], a[r]); return;
17        }
18        int mid = (l + r) >> 1;
19        double mx = (a[mid + 1].x + a[mid].x) / 2;
20        find(l, mid); find(mid + 1, r);
21        int i = l, j = mid + 1;
22        for (int k = l; k <= r; ++k) b[k] = a[((j > r) || (i <= mid && a[j] > a[i])) ? (i++) :
23            ↪ (j++)];
24        for (int k = l; k <= r; ++k) a[k] = b[k];
25        int tot = 0;
26        for (int k = l; k <= r; ++k) if (fabs(a[k].x - mx) <= ans) {
27            for (int j = tot; j >= 1 && (a[k].y - b[j].y <= ans); --j) upd(a[k], b[j]);
28            b[++tot] = a[k];
29        }
30    }
31
32    //接口
33    inline double solve(int n, point ipt[]){
34        ans = 0x3f3f3f3f3f3f3fll; //max distance
35        for (int i = 1; i <= n; ++i) a[i] = ipt[i];
36        sort(a + 1, a + 1 + n, cmp1);
37        find(1, n);
38        return ans;
39    }
40 }
```

## 圆

### 三点垂心

```
1 inline point geto(point p1, point p2, point p3) {
2     double a = p2.x - p1.x;
3     double b = p2.y - p1.y;
4     double c = p3.x - p2.x;
5     double d = p3.y - p2.y;
6     double e = sqr(p2.x) + sqr(p2.y) - sqr(p1.x) - sqr(p1.y);
```

```

7     double f = sqr(p3.x) + sqr(p3.y) - sqr(p2.x) - sqr(p2.y);
8     return {(f * b - e * d) / (c * b - a * d) / 2, (a * f - e * c) / (a * d - b * c) / 2};
9 }

```

## 最小覆盖圆

- 随机增量  $O(n)$

```

1 inline void min_circlefill(point &o, double &r, int n, point a[]) {
2     mt19937 myrand(20011224); shuffle(a + 1, a + 1 + n, myrand); //越随机越难 hack
3     o = a[1];
4     r = 0;
5     for (int i = 1; i <= n; ++i) if ((a[i] ^ o) > r + EPS) {
6         o = a[i];
7         r = 0;
8         for (int j = 1; j < i; ++j) if ((o ^ a[j]) > r + EPS) {
9             o = (a[i] + a[j]) * 0.5;
10            r = (a[i] ^ a[j]) * 0.5;
11            for (int k = 1; k < j; ++k) if ((o ^ a[k]) > r + EPS) {
12                o = geto(a[i], a[j], a[k]);
13                r = (o ^ a[i]);
14            }
15        }
16    }
17 }

```

## 图论

### 存图

- 前向星
- 注意边数开够

```

1 int Head[N], Ver[N*2], Next[N*2], Ew[N*2], Gtot=1;
2 inline void graphinit(int n) {Gtot=1; for(int i=1; i<=n; ++i) Head[i]=0;}
3 inline void edge(int u, int v, int w=1) {Ver[++Gtot]=v; Next[Gtot]=Head[u]; Ew[Gtot]=w;
4     ↪ Head[u]=Gtot;}
5 #define go(i,st,to) for (int i=Head[st], to=Ver[i]; i; i=Next[i], to=Ver[i])

```

### 最短路

#### Dijkstra

- 非负权图

```

1 namespace DIJK{//适用非负权图 满足当前 dist 最小的点一定不会再被松弛
2     typedef pair<long long,int> pii;
3     long long dist[N]; //存最短路长度
4     bool vis[N]; //记录每个点是否被从队列中取出 每个点只需第一次取出时扩展
5     priority_queue<pii,vector<pii>,greater<pii> > pq; //维护当前 dist[] 最小值及对应下标 小根堆
6
7     inline void dijk(int s,int n){ //s 是源点 n 是点数
8         while(pq.size())pq.pop(); for(int i=1;i<=n;++i)dist[i]=INFL,vis[i]=0; //所有变量初始化
9         dist[s]=0;pq.push(make_pair(0,s));
10        while(pq.size()){

```

```

11         int now=pq.top().second;pq.pop();
12         if(vis[now])continue;vis[now]=1;
13         go(i,now,to){
14             const long long relx(dist[now]+Ew[i]);
15             if(dist[to]>relx){dist[to]=relx;pq.push(make_pair(dist[to],to));} //松弛
16         }
17     }
18 }
19 }

```

## LCA

- 倍增求 lca
- 数组开够

```

1 namespace LCA_Log{
2     int fa[N][22],dep[N];
3     int t,now;
4     void dfs(int x){
5         dep[x]=dep[fa[x][0]]+1;
6         go(i,x,to){
7             if(dep[to])continue;
8             fa[to][0]=x;for(int j=1;j<=t;++j)fa[to][j]=fa[fa[to][j-1]][j-1];
9             dfs(to);
10        }
11    }
12
13    //初始化接口
14    inline void lcainit(int n,int rt){ //记得初始化全部变量
15        now=1;t=0;while(now<n)++t,now<=1;
16        for(int i=1;i<=n;++i)dep[i]=0,fa[i][0]=0;
17        for(int i=1;i<=t;++i)fa[rt][i]=0;
18        dfs(rt);
19    }
20
21    //求 lca 接口
22    inline int lca(int u,int v){
23        if(dep[u]>dep[v])swap(u,v);
24        for(int i=t;~i;--i)if(dep[fa[v][i]]>=dep[u])v=fa[v][i];
25        if(u==v)return u;
26        for(int i=t;~i;--i)if(fa[u][i]!=fa[v][i])u=fa[u][i],v=fa[v][i];
27        return fa[u][0];
28    }
29 }

```

## 连通性

### 有向图强联通分量

- tarjan  $O(n)$

```

1 namespace SCC{
2     int dfn[N],clk,low[N];
3     bool ins[N];int sta[N],tot; //栈 存正在构建的强连通块
4     vector<int>scc[N];int c[N],cnt; //cnt 为强联通块数 scc[i] 存放每个块内点 c[i] 为原图每个结点属于的块
5     void dfs(int x){

```

```

6      dfn[x]=low[x]=(++clk); //low[] 在这里初始化
7      ins[x]=1; sta[++tot]=x;
8      go(i,x,to){
9          if(!dfn[to]){dfs(to); low[x]=min(low[x], low[to]);} //走树边
10         else if(ins[to]) low[x]=min(low[x], dfn[to]); //走返祖边
11     }
12     if(dfn[x]==low[x]){ //该结点为块的代表元
13         ++cnt; int u;
14         do{u=sta[tot--]; ins[u]=0; c[u]=cnt; scc[cnt].push_back(u);} while(x!=u);
15     }
16 }
17 inline void tarjan(int n){ //n 是点数
18     for(int i=1; i<=cnt; ++i) scc[i].clear(); //清除上次的 scc 防止被卡 MLE
19     for(int i=1; i<=n; ++i) dfn[i]=ins[i]=0; tot=clk=cnt=0; //全部变量初始化
20     for(int i=1; i<=n; ++i) if(!dfn[i]) dfs(i);
21     for(int i=1; i<=n; ++i) c[i]+=n; //此行 (可以省略) 便于原图上加点建新图 加新点前要初始化 Head[]=0
22 }
23 }

```