XCPC 算法模板和结论 (补充)

August 8, 2025

Contents

Chapter 1

几何

1.1 极角排序

```cpp
#include <bits/stdc++.h>
using ld = long double;
const ld PI = acos(-1);
const ld EPS = 1e-7;
const ld INF = numeric_limits<ld>::max();
#define cc(x) cout << fixed << setprecision(x);

template <class T>
struct Point {  // 在 C++17 下使用 emplace_back 绑定可能会导致 CE!
  T x, y;
  Point(T x_ = 0, T y_ = 0) : x(x_), y(y_) {}  // 初始化
  template <class U>
  operator Point<U>() {  // 自动类型匹配
    return Point<U>(U(x), U(y));
  }
  Point &operator+=(Point p) & { return x += p.x, y += p.y, *this; }
  Point &operator+=(T t) & { return x += t, y += t, *this; }
  Point &operator-=(Point p) & { return x -= p.x, y -= p.y, *this; }
  Point &operator-=(T t) & { return x -= t, y -= t, *this; }
  Point &operator*=(T t) & { return x *= t, y *= t, *this; }
  Point &operator/=(T t) & { return x /= t, y /= t, *this; }
  Point operator-() const { return Point(-x, -y); }
  friend Point operator+(Point a, Point b) { return a += b; }
  friend Point operator+(Point a, T b) { return a += b; }
  friend Point operator-(Point a, Point b) { return a -= b; }
```

```cpp
  friend Point operator-(Point a, T b) { return a -= b; }
  friend Point operator*(Point a, T b) { return a *= b; }
  friend Point operator*(T a, Point b) { return b *= a; }
  friend Point operator/(Point a, T b) { return a /= b; }
  friend bool operator<(Point a, Point b) {
    return equal(a.x, b.x) ? a.y < b.y - EPS : a.x < b.x - EPS;
  }
  friend bool operator>(Point a, Point b) { return b < a; }
  friend bool operator==(Point a, Point b) { return !(a < b) && !(b < a); }
  friend bool operator!=(Point a, Point b) { return a < b || b < a; }
  friend auto &operator>>(istream &is, Point &p) { return is >> p.x >> p.y; }
  friend auto &operator<<(ostream &os, Point p) {
    return os << "(" << p.x << ", " << p.y << ")";
  }
};

using Points = vector<Point<int>>;

double theta(auto p) { return atan2(p.y, p.x); }
void psort(Points &ps, Point<int> c = {0, 0}) {
  sort(ps.begin(), ps.end(),
       [&](auto p1, auto p2) { return theta(p1 - c) < theta(p2 - c); });
}
```

1.2 旋转卡壳

```cpp
#include <bits/stdc++.h>

template <class T>
pair<vector<Point<T>>, T> maxInscribedQuadrilateral(vector<Point<T>>& p) {
  int n = (int)p.size();
  // 至少需要 4 个顶点
  if (n < 4) return {{}, 0};
  // 计算三角形的双倍面积（绝对值）
  auto area2 = [&](int i, int j, int k) {
    T dx1 = p[j].x - p[i].x;
    T dy1 = p[j].y - p[i].y;
    T dx2 = p[k].x - p[i].x;
    T dy2 = p[k].y - p[i].y;
```

```cpp
        return abs(dx1 * dy2 - dy1 * dx2);
    };

    // 找最低且最左顶点 (a0)，最高且最右顶点 (c0)
    int a0 = 0, c0 = 0;
    for (int i = 1; i < n; i++) {
        if (p[i].y < p[a0].y || (p[i].y == p[a0].y && p[i].x < p[a0].x)) a0 = i;
        if (p[i].y > p[c0].y || (p[i].y == p[c0].y && p[i].x > p[c0].x)) c0 = i;
    }


    int a = a0, b = a0, c = c0, d = c0;
    T maxArea = 0;
    int bestA = a, bestB = b, bestC = c, bestD = d;

    // 旋转卡尺主循环
    while (true) {
        // 移动 b 指针，使 A-B-C 三角形面积最大
        while (true) {
            int nb = (b + 1) % n;
            if (area2(a, nb, c) > area2(a, b, c))
                b = nb;
            else
                break;
        }
        // 移动 d 指针，使 C-D-A 三角形面积最大
        while (true) {
            int nd = (d + 1) % n;
            if (area2(c, nd, a) > area2(c, d, a))
                d = nd;
            else
                break;
        }
        // 计算四边形面积（注意这里 area2 返回双倍面积）
        T areaQuad = (area2(a, b, c) + area2(a, c, d)) * T(0.5);
        if (areaQuad > maxArea) {
            maxArea = areaQuad;
            bestA = a;
            bestB = b;
            bestC = c;
```

4

```
      bestD = d;
    }
    // 判断旋转方向：比较移动 C 后与移动 A 后的面积
    int a_next = (a + 1) % n;
    int c_next = (c + 1) % n;
    // 如果 area2(a,a_next,c_next) <= area2(a,a_next,c)，则移动 A，否则移动 C
    if (area2(a, a_next, c_next) <= area2(a, a_next, c)) {
      a = a_next;
    } else {
      c = c_next;
    }
    // 退出条件：回到初始对换位置
    if (a == c0 && c == a0) break;
  }


  vector<Point<T>> quad = {p[bestA], p[bestB], p[bestC], p[bestD]};
  return {quad, maxArea};
}
```

1.3 切比雪夫距离

$$d(A, B) = \max(|x_a - x_b|, |y_a - y_b|)$$

1. 曼哈顿坐标系是通过切比雪夫坐标系旋转 45 度后，再缩小到原来的一半得到的。

2. 将一个点 (x, y) 的坐标变为 $(x + y, x - y)$ 后，原坐标系中的曼哈顿距离等于新坐标系中的切比雪夫距离。

3. 将一个点 (x, y) 的坐标变为 $(\frac{x + y}{2}, \frac{x - y}{2})$ 后，原坐标系中的切比雪夫距离等于新坐标系中的曼哈顿距离。

Chapter 2

树

2.1 树的重心

计算以无根树每个点为根节点时的最大子树大小，这个值最小的点称为无根树的重心。

一些性质:

1. 某个点是树的重心等价于它最大子树大小不大于整棵树大小的一半。

2. 树至多有两个重心。如果树有两个重心，那么它们相邻。此时树一定有偶数个节点，且可以被划分为两个大小相等的分支，每个分支各自包含一个重心。

3. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样。反过来，距离和最小的点一定是重心。

4. 往树上增加或减少一个叶子，如果原节点数是奇数，那么重心可能增加一个，原重心仍是重心；如果原节点数是偶数，重心可能减少一个，另一个重心仍是重心。

5. 把两棵树通过一条边相连得到一棵新的树，则新的重心在较大的一棵树一侧的连接点与原重心之间的简单路径上。如果两棵树大小一样，则重心就是两个连接点。

利用性质 1, 可以很快的找到重心.

2.2 树的直径

若树上所有边边权均为正，则树的所有直径中点重合

由此可以知道找出 (端点) 字典序最大的直径的方法: 从任意顶点出发: 找到离它最远其字典序最大的点。然后从这个点出发找离他最远且字典序最大的点。

这条直径就是答案。

Chapter 3

数学

3.1 多项式封装

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

typedef long long LL;
template <unsigned umod>
struct modint {
  static constexpr int mod = umod;
  unsigned v;
  modint() : v(0) {}
  template <class T, enable_if_t<is_integral<T>::value>* = nullptr>
  modint(T x) {
    x %= mod;
    if (x < 0) x += mod;
    v = x;
  }
  modint(const string& str) {
    v = 0;
    size_t i = 0;
    if (str.front() == '-') i += 1;
    while (i < str.size()) {
      assert(isdigit(str[i]));
      v = (v * 10ull % umod + str[i] - '0') % umod;
      i += 1;
    }
    if (str.front() == '-' && v) v = umod - v;
```

```cpp
    }
    modint operator+() const { return *this; }
    modint operator-() const { return modint() - *this; }
    friend int raw(const modint& self) { return self.v; }
    friend istream& operator>>(istream& is, modint& self) {
      string str;
      is >> str;
      self = str;
      return is;
    }
    friend ostream& operator<<(ostream& os, const modint& self) {
      return os << raw(self);
    }
    modint& operator+=(const modint& rhs) {
      v += rhs.v;
      if (v >= umod) v -= umod;
      return *this;
    }
    modint& operator-=(const modint& rhs) {
      v -= rhs.v;
      if (v >= umod) v += umod;
      return *this;
    }
    modint& operator*=(const modint& rhs) {
      v = static_cast<unsigned>(1ull * v * rhs.v % umod);
      return *this;
    }
    modint& operator/=(const modint& rhs) {
      static constexpr size_t ilim = 1 << 20;
      static modint inv[ilim + 10];
      static int sz = 0;
      assert(rhs.v);
      if (rhs.v > ilim) return *this *= qpow(rhs, mod - 2);
      if (!sz) inv[1] = sz = 1;
      while (sz < (int)rhs.v) {
        for (int i = sz + 1; i <= sz << 1; i++) inv[i] = -mod / i * inv[mod % i];
        sz <<= 1;
      }
      return *this *= inv[rhs.v];
```

8

```cpp
  }
  template <class T>
  friend modint qpow(modint a, T b) {
    modint r = 1;
    for (; b; b >>= 1, a *= a)
      if (b & 1) r *= a;
    return r;
  }
  friend modint operator+(modint lhs, const modint& rhs) { return lhs += rhs; }
  friend modint operator-(modint lhs, const modint& rhs) { return lhs -= rhs; }
  friend modint operator*(modint lhs, const modint& rhs) { return lhs *= rhs; }
  friend modint operator/(modint lhs, const modint& rhs) { return lhs /= rhs; }
  friend bool operator==(const modint& lhs, const modint& rhs) {
    return lhs.v == rhs.v;
  }
  friend bool operator!=(const modint& lhs, const modint& rhs) {
    return lhs.v != rhs.v;
  }
};

typedef modint<998244353> mint;
// 返回大于等于 x 的最小 2 的幂
int glim(const int& x) { return 1 << (32 - __builtin_clz(x - 1)); }
// 返回 x 尾部连续 0 的个数。
int bitctz(const int& x) { return __builtin_ctz(x); }
struct poly : vector<mint> {
  poly() {}
  explicit poly(int n) : vector<mint>(n) {}
  poly(const vector<mint>& vec) : vector<mint>(vec) {}
  // 列表初始化
  poly(initializer_list<mint> il) : vector<mint>(il) {}
  mint operator()(const mint& x) const;
  poly& cut(int lim);
  void ntt(int op);
};
// 输入多项式系数
istream& operator>>(istream& is, poly& a) {
  for (auto& x : a) is >> x;
  return is;
```

```
}
// 输出一个多项式
ostream& operator<<(ostream& os, const poly& a) {
  bool flag = false;
  for (auto& x : a) {
    if (flag)
      os << " ";
    else
      flag = true;
    os << x;
  }
  return os;
}
// 单点求值
mint poly::operator()(const mint& x) const {
  const auto& a = *this;
  mint res = 0;
  for (int i = (int)a.size() - 1; i >= 0; i--) {
    res = res * x + a[i];
  }
  return res;
}
// 截断到 lim
poly& poly::cut(int lim) {
  resize(lim);
  return *this;
}

void poly::ntt(int op) {
  static bool wns_flag = false;
  static vector<mint> wns;
  if (!wns_flag) {
    wns_flag = true;
    for (int j = 1; j <= 23; j++) {
      wns.push_back(qpow(mint(3), raw(mint(-1)) >> j));
    }
  }
  vector<mint>& a = *this;
  int n = a.size();
```

```
    for (int i = 1, r = 0; i < n; i++) {
      r ^= n - (1 << (bitctz(n) - bitctz(i) - 1));
      if (i < r) std::swap(a[i], a[r]);
    }
    vector<mint> w(n);
    for (int k = 1, len = 2; len <= n; k <<= 1, len <<= 1) {
      mint wn = wns[bitctz(k)];
      for (int i = raw(w[0] = 1); i < k; i++) w[i] = w[i - 1] * wn;
      for (int i = 0; i < n; i += len) {
        for (int j = 0; j < k; j++) {
          mint x = a[i + j], y = a[i + j + k] * w[j];
          a[i + j] = x + y, a[i + j + k] = x - y;
        }
      }
    }
    if (op == -1) {
      mint iz = mint(1) / n;
      for (int i = 0; i < n; i++) a[i] *= iz;
      reverse(a.begin() + 1, a.end());
    }
  }
// 牛顿迭代，vec 是多项式，func 是计算的函数
poly concalc(int n, vector<poly> vec,
             const function<mint(vector<mint>)>& func) {
  int lim = glim(n);
  int m = vec.size();
  for (auto& f : vec) f.resize(lim), f.ntt(1);
  vector<mint> tmp(m);
  poly ret(lim);
  for (int i = 0; i < lim; i++) {
    for (int j = 0; j < m; j++) tmp[j] = vec[j][i];
    ret[i] = func(tmp);
  }
  ret.ntt(-1);
  return ret;
}

poly getInv(const poly& a, int lim) {
  poly b{1 / a[0]};
```

```cpp
  for (int len = 2; len <= glim(lim); len <<= 1) {
    poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
    b = concalc(len << 1, {b, c}, [](vector<mint> vec) {
          return vec[0] * (2 - vec[0] * vec[1]);
        }).cut(len);
  }
  return b.cut(lim);
}


poly operator+=(poly& a, const poly& b) {
  if (a.size() < b.size()) a.resize(b.size());
  for (size_t i = 0; i < b.size(); i++) a[i] += b[i];
  return a;
}


poly operator-=(poly& a, const poly& b) {
  if (a.size() < b.size()) a.resize(b.size());
  for (size_t i = 0; i < b.size(); i++) a[i] -= b[i];
  return a;
}


poly operator*=(poly& a, const mint& k) {
  if (k == 1) return a;
  for (size_t i = 0; i < a.size(); i++) a[i] *= k;
  return a;
}
poly operator/=(poly& a, const mint& k) { return a *= 1 / k; }

poly operator<<=(poly& a, const int& k) {
  // mnltiple by x^k
  a.insert(a.begin(), k, 0);
  return a;
}


poly operator>>=(poly& a, const int& k) {
  // divide by x^k
  a.erase(a.begin(), a.begin() + min(k, (int)a.size()));
  return a;
}
```

```cpp
poly operator*(const poly& a, const poly& b) {
  if (a.empty() || b.empty()) return {};
  int rlen = a.size() + b.size() - 1;
  int len = glim(rlen);
  if (1ull * a.size() * b.size() <= 1ull * len * bitctz(len)) {
    poly ret(rlen);
    for (size_t i = 0; i < a.size(); i++)
      for (size_t j = 0; j < b.size(); j++) ret[i + j] += a[i] * b[j];
    return ret;
  } else {
    return concalc(len, {a, b},
                   [](vector<mint> vec) { return vec[0] * vec[1]; })
        .cut(rlen);
  }
}
poly operator/(poly a, poly b) {
  if (a.size() < b.size()) return {};
  int rlen = a.size() - b.size() + 1;
  reverse(a.begin(), a.end());
  reverse(b.begin(), b.end());
  a = (a * getInv(b, rlen)).cut(rlen);
  reverse(a.begin(), a.end());
  return a;
}
poly operator-(poly a, const poly& b) { return a -= b; }
poly operator%(const poly& a, const poly& b) {
  return (a - (a / b) * b).cut(b.size() - 1);
}
poly operator*=(poly& a, const poly& b) { return a = a * b; }
poly operator/=(poly& a, const poly& b) { return a = a / b; }
poly operator%=(poly& a, const poly& b) { return a = a % b; }
poly operator+(poly a, const poly& b) { return a += b; }
poly operator*(poly a, const mint& k) { return a *= k; }
poly operator*(const mint& k, poly a) { return a *= k; }
poly operator/(poly a, const mint& k) { return a /= k; }
poly operator<<(poly a, const int& k) { return a <<= k; }
poly operator>>(poly a, const int& k) { return a >>= k; }
// 形式导数
poly getDev(poly a) {
```

13

```cpp
    a >>= 1;
    for (size_t i = 1; i < a.size(); i++) a[i] *= i + 1;
    return a;
}
// 不定积分
poly getInt(poly a) {
    a <<= 1;
    for (size_t i = 1; i < a.size(); i++) a[i] /= i;
    return a;
}
// 对数函数
poly getLn(const poly& a, int lim) {
    assert(a[0] == 1);
    return getInt(getDev(a) * getInv(a, lim)).cut(lim);
}
// 指数函数
poly getExp(const poly& a, int lim) {
    assert(a[0] == 0);
    poly b{1};
    for (int len = 2; len <= glim(lim); len <<= 1) {
        poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
        b = concalc(len << 1, {b, getLn(b, len), c}, [](vector<mint> vec) {
            return vec[0] * (1 - vec[1] + vec[2]);
        }).cut(len);
    }
    return b.cut(lim);
}
// 快速幂
poly qpow(const poly& a, string k, int lim) {
    size_t i = 0;
    while (i < a.size() && a[i] == 0) i += 1;
    if (i == a.size() || (i > 0 && k.size() >= 9) ||
        1ull * i * raw(mint(k)) >= 1ull * lim)
        return poly(lim);
    lim -= i * raw(mint(k));
    return getExp(getLn(a / a[i] >> i, lim) * k, lim) *
                qpow(a[i], raw(modint<mint::mod - 1>(k)))
            << i * raw(mint(k));
}
```

```cpp
poly qpow(const poly& a, LL k, int lim) {
  size_t i = 0;
  while (i < a.size() && a[i] == 0) i += 1;
  if (i == a.size() || (i > 0 && k >= 1e9) || 1ull * i * k >= 1ull * lim)
    return poly(lim);
  lim -= i * k;
  return getExp(getLn(a / a[i] >> i, lim) * k, lim) *
              qpow(a[i], raw(modint<mint::mod - 1>(k)))
         << i * k;
}


mint sqrt(const mint& c) {
  static const auto check = [](mint c) {
    return qpow(c, (mint::mod - 1) >> 1) == 1;
  };
  if (raw(c) <= 1) return 1;
  if (!check(c)) throw "No solution!";
  static mt19937 rng{random_device{}()};
  mint a = rng();
  while (check(a * a - c)) a = rng();
  typedef pair<mint, mint> number;
  const auto mul = [=](number x, number y) {
    return make_pair(x.first * y.first + x.second * y.second * (a * a - c),
                     x.first * y.second + x.second * y.first);
  };
  const auto qpow = [=](number a, int b) {
    number r = {1, 0};
    for (; b; b >>= 1, a = mul(a, a))
      if (b & 1) r = mul(r, a);
    return r;
  };
  mint ret = qpow({a, 1}, (mint::mod + 1) >> 1).first;
  return min(raw(ret), raw(-ret));
}
// 开根号
poly getSqrt(const poly& a, int lim) {
  poly b{sqrt(a[0])};
  for (int len = 2; len <= glim(lim); len <<= 1) {
    poly c = vector<mint>(a.begin(), a.begin() + min(len, (int)a.size()));
```

```cpp
      b = (c * getInv(b * 2, len) + b / 2).cut(len);
    }
    return b.cut(lim);
}
template <class T>
mint divide_at(poly f, poly g, T n) {
    for (; n; n >>= 1) {
        poly r = g;
        for (size_t i = 1; i < r.size(); i += 2) r[i] *= -1;
        f *= r;
        g *= r;
        int i;
        for (i = n & 1; i < (int)f.size(); i += 2) f[i >> 1] = f[i];
        f.resize(i >> 1);
        for (i = 0; i < (int)g.size(); i += 2) g[i >> 1] = g[i];
        g.resize(i >> 1);
    }
    return f.empty() ? 0 : f[0] / g[0];
}


template <class T>
mint linear_rec(poly a, poly f, T n) {
    // a[n] = sum_i f[i] * a[n - i]
    a.resize(f.size() - 1);
    f = poly{1} - f;
    poly g = a * f;
    g.resize(a.size());
    return divide_at(g, f, n);
}
poly BM(poly a) {
    poly ans, lst;
    int w = 0;
    mint delta = 0;
    for (size_t i = 0; i < a.size(); i++) {
        mint tmp = -a[i];
        for (size_t j = 0; j < ans.size(); j++) tmp += ans[j] * a[i - j - 1];
        if (tmp == 0) continue;
        if (ans.empty()) {
            w = i;
```

```cpp
        delta = tmp;
        ans = vector<mint>(i + 1, 0);
      } else {
        auto now = ans;
        mint mul = -tmp / delta;
        if (ans.size() < lst.size() + i - w) ans.resize(lst.size() + i - w);
        ans[i - w - 1] -= mul;
        for (size_t j = 0; j < lst.size(); j++) ans[i - w + j] += lst[j] * mul;
        if (now.size() <= lst.size() + i - w) {
          w = i;
          lst = now;
          delta = tmp;
        }
      }
    }
  }
  return ans << 1;
}


poly lagrange(const vector<pair<mint, mint>>& a) {
  poly ans(a.size()), product{1};
  for (size_t i = 0; i < a.size(); i++) {
    product *= poly{-a[i].first, 1};
  }
  auto divide2 = [&](poly a, mint b) {
    poly res(a.size() - 1);
    for (size_t i = (int)a.size() - 1; i >= 1; i--) {
      res[i - 1] = a[i];
      a[i - 1] -= a[i] * b;
    }
    return res;
  };
  for (size_t i = 0; i < a.size(); i++) {
    mint denos = 1;
    for (size_t j = 0; j < a.size(); j++) {
      if (i != j) denos *= a[i].first - a[j].first;
    }
    poly numes = divide2(product, -a[i].first);
    ans += a[i].second / denos * numes;
  }
```

```
    return ans;
}
```

3.2   异或线性基

时间复杂度 $O(n \log \max a)$ 异或问题, 同时又可以找到"子集""子序列"等字眼, 或者是图论中的某条路径的异或和时, 就可以往线性基方向想了。

我们可以利用异或线性基实现:

1. 判断一个数能否表示成某数集子集的异或和;

2. 求一个数表示成某数集子集异或和的方案数;

3. 求某数集子集的最大/最小/第 k 大/第 k 小异或和; (注意 01Tire 是求点对区间的第 k 大异或对)

4. 求一个数在某数集子集异或和中的排名。

```cpp
/* 异或线性基
 * 求数集子集异或和第 k 小 (k 从 0 开始计数)
 */
struct Basis {
    vector<u64> B;
    int cntz=0;//0 的个数
    bool ok=false;
    void insert(u64 x) {
      for (auto b:B) x=min(x,x^b);
      for (auto& b: B) b=min(b,b^x);
      if (x) B.push_back(x);
      else cntz++;
    }
    void _min() {
      sort(B.begin(),B.end());
      for (int i=1;i<B.size();++i) {
        for (int j=i-1;j>=0;--j) {
          B[i]=min(B[i],B[i]^B[j]);
        }
      }
    }
    //第 k 小的异或和
    u64 query(int k,bool overphi) {//第 k 小, 包含空集?(k 从 0 开始数)
      if (!ok)_min(),ok=true;
      if (!overphi and !cntz) k++;
      if (k>=(1ll<<(B.size()))) return -1;
```

```
    int res=0;
    for (int i=0;i<B.size();++i) {
      if ((k>>i)&1) res^=B[i];
    }
    return res;
  }


  u64 querymx() {
    return query((1ll<<B.size())-1,1);
  }


  void print() {
    for (int i=0;i<B.size();++i) cout<<B[i]<<" ";
    cout<<"\n";
  }


  //线性基的合并（双 log）
  void operator+=(Basis& _B) {
    for (auto &b:_B.B) this->insert(b);
  }
  friend Basis operator+(Basis& b1,Basis& b2) {
    Basis res=b1;
    for (auto& b:b2.B)res.insert(b);
    return res;
  }
};
```

模板题: 最大异或和:https://www.luogu.com.cn/record/204660302

Chapter 4

数据结构

4.1 线段树二分

```cpp
/*
线段树（LazySegmentTree）
左闭右开
*/
template <class Info, class Tag>
struct LazySegmentTree {
  int n;  // n+1
  vector<Info> info;
  vector<Tag> tag;
  // init begin
  LazySegmentTree() : n(0) {}
  LazySegmentTree(int n_, Info v_ = Info()) {
    init(n_ + 1, v_);  // 下标从 1 开始
  }
  template <class T>
  LazySegmentTree(vector<T> init_) {
    init(init_);
  }
  void init(int n_, Info v_ = Info()) { init(vector(n_, v_)); }
  template <class T>
  void init(vector<T> init_) {
    n = init_.size();
    info.assign(4 << __lg(n), Info());
    tag.assign(4 << __lg(n), Tag());
    std::function<void(int, int, int)> build = [&](int p, int l, int r) {
```

```cpp
    if (r - l == 1) {
      info[p] = init_[l];
      return;
    }
    int m = (l + r) / 2;
    build(2 * p, l, m);
    build(2 * p + 1, m, r);
    pull(p);
  };
  build(1, 1, n);
}
// init end
//  up
void pull(int p) { info[p] = info[2 * p] + info[2 * p + 1]; }
// 修改
void apply(int p, const Tag &v, int len) {
  info[p].apply(v, len);
  tag[p].apply(v);
}
// down
void push(int p, int len) {
  apply(2 * p, tag[p], len / 2);
  apply(2 * p + 1, tag[p], len - len / 2);
  tag[p] = Tag();
}
// 单点修改
void modify(int p, int l, int r, int x, const Info &v) {
  if (r - l == 1) {
    info[p] = v;
    return;
  }
  int m = (l + r) / 2;
  push(p, r - l);
  if (x < m) {
    modify(2 * p, l, m, x, v);
  } else {
    modify(2 * p + 1, m, r, x, v);
  }
  pull(p);
```

```cpp
  }
  void modify(int p, const Info &v) { modify(1, 1, n, p, v); }
  // 区间查询
  Info rangeQuery(int p, int l, int r, int x, int y) {
    if (l >= y || r <= x) {
      return Info();
    }
    if (l >= x && r <= y) {
      return info[p];
    }
    int m = (l + r) / 2;
    push(p, r - l);
    return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
  }
  Info rangeQuery(int l, int r) { return rangeQuery(1, 1, n, l, r); }

  // 区间修改
  void rangeApply(int p, int l, int r, int x, int y, const Tag &v) {
    if (l >= y || r <= x) {
      return;
    }
    if (l >= x && r <= y) {
      apply(p, v, r - l);
      return;
    }
    int m = (l + r) / 2;
    push(p, r - l);
    rangeApply(2 * p, l, m, x, y, v);
    rangeApply(2 * p + 1, m, r, x, y, v);
    pull(p);
  }
  void rangeApply(int l, int r, const Tag &v) {
    return rangeApply(1, 1, n, l, r, v);
  }
  //线段树二分
  template <class F>
  int findFirst(int p, int l, int r, int x, int y, F &&pred) {
    if (l >= y || r <= x) {
      return -1;
```

```
  }
  if (l >= x && r <= y && !pred(info[p])) {
    return -1;
  }
  if (r - l == 1) {
    return l;
  }
  int m = (l + r) / 2;
  push(p,r-l);
  int res = findFirst(2 * p, l, m, x, y, pred);
  if (res == -1) {
    res = findFirst(2 * p + 1, m, r, x, y, pred);
  }
  return res;
}
//第一个满足条件 F 的位置
template <class F>
int findFirst(int l, int r, F &&pred) {
  return findFirst(1, 1, n, l, r, pred);
}
template <class F>
int findLast(int p, int l, int r, int x, int y, F &&pred) {
  if (l >= y || r <= x) {
    return -1;
  }
  if (l >= x && r <= y && !pred(info[p])) {
    return -1;
  }
  if (r - l == 1) {
    return l;
  }
  int m = (l + r) / 2;
  push(p,r-l);
  int res = findLast(2 * p + 1, m, r, x, y, pred);
  if (res == -1) {
    res = findLast(2 * p, l, m, x, y, pred);
  }
  return res;
}
```

```cpp
    //最后一个满足条件 F 的位置
    template <class F>
    int findLast(int l, int r, F &&pred) {
      return findLast(1, 1, n, l, r, pred);
    }
};


struct Tag {
    int x = 0;
    void apply(const Tag &t) & { x += t.x; }
};


struct Info {
    int sum = 0,mx=-iinf,mi=iinf;
    void apply(const Tag &t, int len) & {
      sum += t.x * len;
      mx +=t.x;
      mi +=t.x;
    }
};
// merge
Info operator+(const Info &a, const Info &b) {
    Info res={};
    res.sum=a.sum+b.sum;
    res.mx=max(a.mx,b.mx);
    res.mi=min(a.mi,b.mi);
    return res;
}
```

4.2 线段树维护区间 gcd

$$\gcd_{i=l}^{r} = \gcd(a_l, \gcd_{i=l+1}^{r} (a[i] - a[i-1]))$$

这意味着我们无须维护区间加，只要做差分数组并维护单点加就可以了。


```cpp
int mygcd(int a,int b) {
    return __gcd(abs(a),abs(b));
}
```

```cpp
template<class T>
struct Segt {
  struct node {
    int l,r;
    T w;// gcd
    T sum;
  };
  vector<T> w;
  vector<node> t;
  Segt(){}
  Segt(int n) {init(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];
    }
    init(in.size()-1);
  }
#define GL k<<1
#define GR k<<1|1
  void init(int n) {
    t.resize(4*n +1);
    auto build=[&](auto self ,int l, int r,int k=1) {
      if (l==r) {
        t[k]={l,r,w[l],w[l]};
        return ;
      }
      t[k]={l,r,0,0};
      int mid=(l+r)/2;
      self(self,l,mid,GL);
      self(self,mid+1,r,GR);
      pushup(k);
    };
    build(build,1,n);
  }
  void pushup(int k) {
    auto pushup=[&](node& p,node& l, node &r) {
      p.w=mygcd(l.w,r.w);
```

```cpp
      p.sum=l.sum+r.sum;
    };
    pushup(t[k],t[GL],t[GR]);
  }
  void add(int pos,T val,int k=1) {
    if (t[k].l==t[k].r) {
      t[k].w+=val;
      t[k].sum+=val;
      return ;
    }
    int mid=(t[k].l+t[k].r)/2;
    if (pos<=mid) add(pos,val,GL);
    else add(pos,val,GR);
    pushup(k);
  }
  // 单点赋值，不用管 sum
  void upd(int pos,T val,int k=1) {
    if (t[k].l==t[k].r) {
      t[k].w=val;
      return ;
    }
    int mid=(t[k].l+t[k].r)/2;
    if (pos<=mid) upd(pos,val,GL);
    else upd(pos,val,GR);
    pushup(k);
  }
  T askgcd(int l,int r,int k=1) {
    if (l<=t[k].l and t[k].r<=r) return t[k].w;
    int mid=(t[k].l+t[k].r)/2;
    T ans=0;
    if (l<=mid) ans=mygcd(ans,askgcd(l,r,GL));
    if (mid<r)  ans=mygcd(ans,askgcd(l,r,GR));
    return ans;
  }
  T asksum(int l,int r,int k=1) {
    if (l<=t[k].l and t[k].r<=r) return t[k].sum;
    int mid=(t[k].l+t[k].r)/2;
    T ans=0;
    if (l<=mid) ans+=asksum(l,r,GL);
```

```cpp
        if (mid<r)  ans+=asksum(l,r,GR);
        return ans;
    }
};


void solve() {
    int n,m;cin>>n>>m;
    vector<int> a(n+1);
    for (int i=1;i<=n;++i) cin>>a[i];
    for (int i=n;i;--i) a[i]-=a[i-1];
    Segt<int> sgt(a);
    for (int i=1;i<=m;++i) {
        char op;cin>>op;
        if (op=='C') {// 区间修改
            int l,r,d;cin>>l>>r>>d;
            sgt.add(l,d);
            if (r<n) sgt.add(r+1,-d);
        }else {//区间查询
            int l,r;cin>>l>>r;
            cout<<mygcd(sgt.asksum(1,l),sgt.askgcd(l+1,r))<<"\n";
        }
    }
}
```

4.3   对顶堆

```cpp
using namespace std;
struct Maxheap {
    int n;
    vector<int> w;
    Maxheap(auto &_init):w(_init) {
        n=static_cast<int>(_init.size())-1;
        w.resize(n+1);
        for (int i=1;i<=n;++i) up(i);
    }
    void up(int x) {
        while (x>1 and w[x]>w[x/2]) swap(w[x],w[x/2]),x/=2;
    }
    void down(int x) {
```

```
    while (x*2<=n) {
      int t=x*2;
      if (t+1<=n and w[t+1]>w[t]) t++;
      if (w[x]>=w[t]) return ;
      swap(w[x],w[t]);
      x=t;
    }
  }
};
```

对顶堆可以动态维护一个序列上的中位数, 或者第 k 大的数,(k 的值可能变化).

对顶堆由一个大根堆与一个小根堆组成, 小根堆维护大值即前 k 大的值（包含第 k 个）, 大根堆维护小值即比第 k 大数小的其他数。

维护: 当小根堆的大小小于 k 时, 不断将大根堆堆顶元素取出并插入小根堆, 直到小根堆的大小等于 k;

当小根堆的大小大于 k 时, 不断将小根堆堆顶元素取出并插入大根堆, 直到小根堆的大小等于 k;

插入元素: 若插入的元素大于等于小根堆堆顶元素, 则将其插入小根堆, 否则将其插入大根堆, 然后维护对顶堆;

查询第 k 大元素: 小根堆堆顶元素即为所求;

删除第 k 大元素: 删除小根堆堆顶元素, 然后维护对顶堆;

变化 k: 根据新的 k 值直接维护对顶堆。

时间复杂度 $O(\log n)$

```
struct Heap {
  int k;
  int sum = 0;
  multiset<i64> mxq, miq;
  map<int, int> mxc, mic;
  void init(vector<int> _init) {
    sort(_init.begin() + 1, _init.end(), greater<int>());
    for (int i = 1; i <= k; ++i) miq.insert(_init[i]), mic[_init[i]]++;
    for (int i = k + 1; i < _init.size(); ++i)
      mxq.insert(-_init[i]), mxc[-_init[i]]++;
  }
  void adj() {
    while (miq.size() < k) {
      miq.insert(-*mxq.begin()), mic[-*mxq.begin()]++;
      mxc[*mxq.begin()]--;
      mxq.erase(mxq.begin());
    }
    while (miq.size() > k) {
```

```
      mxq.insert(-*miq.begin());
      mxc[-*miq.begin()]++;
      mic[*miq.begin()]--;
      miq.erase(miq.begin());
    }
  }
  void insert(i64 x) {
    if (x >= *miq.begin())
      miq.insert(x), mic[x]++;
    else
      mxq.insert(-x), mxc[-x]++;
    adj();
  }
  void erase(i64 x) {
    if (x >= *miq.begin())
      miq.erase(miq.find(x)), mic[x]--;
    else
      mxq.erase(mxq.find(-x)), mxc[-x]--;
  }
  i64 queryk() { return *miq.begin(); }
};
```

4.4 手写 Bitset

```
u64 mi[200];
// for (int i = 0; i < 64; i++) mi[i] = (1ULL << i);
struct Bit {
  vector<u64> bit;
  int len;

  Bit(int sz = 0) {
    len = (sz + 63) >> 6;
    bit.assign(len, 0);
  }

#define I inline
  I void reset() { fill(bit.begin(), bit.end(), 0); }
  Bit() { fill(bit.begin(), bit.end(), 0); }
  I void set1(int x) { bit[x >> 6] |= mi[x & 63]; }
```

```cpp
  I void set0(int x) { bit[x >> 6] &= ~mi[x & 63]; }
  I void flip(int x) { bit[x >> 6] ^= mi[x & 63]; }
  bool operator[](int x) { return (bit[x >> 6] >> (x & 63)) & 1; }
#define re register
  Bit operator~(void) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = ~bit[i];
    return res;
  }
  Bit operator&(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] & b.bit[i];
    return res;
  }
  Bit operator|(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] | b.bit[i];
    return res;
  }
  Bit operator^(const Bit &b) const {
    Bit res;
    for (re int i = 0; i < len; i++) res.bit[i] = bit[i] ^ b.bit[i];
    return res;
  }
  void operator&=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] &= b.bit[i];
  }
  void operator|=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] |= b.bit[i];
  }
  void operator^=(const Bit &b) {
    for (re int i = 0; i < len; i++) bit[i] ^= b.bit[i];
  }
  Bit operator<<(const int t) const {
    Bit res;
    int high = t >> 6, low = t & 63;
    u64 last = 0;
    for (register int i = 0; i + high < len; i++) {
      res.bit[i + high] = (last | (bit[i] << low));
```

```cpp
      if (low) last = (bit[i] >> (64 - low));
    }
    return res;
  }
  Bit operator>>(const int t) const {
    Bit res;
    int high = t >> 6, low = t & 63;
    u64 last = 0;
    for (register int i = len - 1; i >= high; i--) {
      res.bit[i - high] = last | (bit[i] >> low);
      if (low) last = bit[i] << (64 - low);
    }
    return res;
  }
  void operator<<=(const int t) {
    int high = t >> 6, low = t & 63;
    for (register int i = len - high - 1; ~i; i--) {
      bit[i + high] = (bit[i] << low);
      if (low && i) bit[i + high] |= bit[i - 1] >> (64 - low);
    }
    for (register int i = 0; i < high; i++) bit[i] = 0;
  }
};
```

Chapter 5

trick

5.0.1　枚举子集

用于循环枚举子集，**注意枚举不了空集**

```
for(int j=st;j;j=(j-1)&st)
```

st，为要枚举子集的集合，j 为子集
本质是将每一位设为 0，1 后枚举后面的。
时间复杂度 $O(3^n)$

5.0.2　求所有因数

利用类似筛法的原理

```
for (int i = 1; i <= MX; ++i) {
  for (int j = i; j <= MX; j += i) {
    d[j].push_back(i);
  }
}
```

时间复杂度 $O(n \log n)$

Chapter 6

杂项

6.1　格雷码

构造格雷码

$$G(n) = n \oplus \left\lfloor \frac{n}{2} \right\rfloor$$

格雷码构造原数

$$n_{k-i} = \oplus_{0 \le j \le i} g_{k-j}$$

6.2　pbds

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>//用 tree
#include<ext/pb_ds/hash_policy.hpp>//用 hash
#include<ext/pb_ds/trie_policy.hpp>//用 trie
#include<ext/pb_ds/priority_queue.hpp>//用 priority_queue
```

6.2.1　哈希表

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/hash_policy.hpp>
const int RANDOM = time(NULL);
struct MyHash {int operator() (int x) const {return x ^ RANDOM;}};
template <class T1, class T2>
struct std::tr1::hash <std::pair <T1, T2> > {
  size_t operator() (std::pair <T1, T2> x) const {
    std::tr1::hash <T1> H1; std::tr1::hash <T2> H2;
    return H1(x.first) ^ H2(x.second); // 你自定义的 hash 函数。
```

```
  }
};
```

直接当没有 emplace(),cbegin(),cend()unordered_map 用就好了。