XCPC 算法模板和结论 (补充)

July 16, 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; }
```

```
  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 lt(theta(p1 - c), theta(p2 - c)); });
}
```

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 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。

```
/* 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

```
/*
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) {
```

```
    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);
```

```
}
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;
}
```

```
  //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。

```
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);
```

```
    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);
```

```
    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

```
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

```
struct Bitset {
  Bitset() : Bitset(0) {}
  Bitset(int _sz) {
    sz = _sz;
    ptr = 0;
    vec.resize((_sz + 63) >> 6);
  }
  void Add(int cnt, uLL val) {
    if (cnt <= 64 - (ptr & 63))
      vec[ptr >> 6] |= val << (ptr & 63);
    else {
      uLL mask = BitBetween(0, 64 - (ptr & 63) - 1);
      vec[ptr >> 6] |= (val & mask) << (ptr & 63);
      vec[(ptr >> 6) + 1] = val >> (64 - (ptr & 63));
    }
```

```
    ptr += cnt;
  }
  void GetSame(const Bitset& rhs) {
    while (sz != ptr || rhs.sz != rhs.ptr);
    for (int i = 0; i < vec.size(); i++) vec[i] ^= ~rhs.vec[i];
    int mn_sz = min(sz, rhs.sz);
    for (int i = mn_sz >> 6; i < vec.size(); i++) {
      int l_bit = max(0, mn_sz - (i << 6));
      int r_bit = 63;
      vec[i] &= ~BitBetween(l_bit, r_bit);
    }
  }
  uLL Calc() const {
    uLL ret = 0;
    for (int i = 0, ri = 0; i < vec.size(); i++) {
      Update(vec[i] & (S - 1), ret, ri);
      Update(vec[i] >> 16 & (S - 1), ret, ri);
      Update(vec[i] >> 32 & (S - 1), ret, ri);
      Update(vec[i] >> 48 & (S - 1), ret, ri);
    }
    return ret;
  }
  void out() const {
    fprintf(stderr, "sz = %d : ", sz);
    for (int i = 0; i < sz; i++)
      fprintf(stderr, "%llu", (vec[i >> 6] >> (i & 63)) & 1);
    fprintf(stderr, "\n");
  }

  int sz, ptr;
  vector<uLL> vec;
};
```

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

```
#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。