整体二分

引入

在信息学竞赛中，有一部分题目可以使用二分的办法来解决。但是当这种题目有多次询问且我们每次查询都直接二分可能导致 TLE 时，就会用到整体二分。整体二分的主体思路就是把多个查询一起解决。（所以这是一个离线算法）

> 可以使用整体二分解决的题目需要满足以下性质：
>
> 1. 询问的答案具有可二分性
> 2. **修改对判定答案的贡献互相独立**，修改之间互不影响效果
> 3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
> 4. 贡献满足交换律，结合律，具有可加性
> 5. 题目允许使用离线算法
>
> ——许昊然《浅谈数据结构题几个非经典解法》

解释

记 $[l,r]$ 为答案的值域，$[L,R]$ 为答案的定义域。（也就是说求答案时仅考虑下标在区间 $[L,R]$ 内的操作和询问，这其中询问的答案在 $[l,r]$ 内）

- 我们首先把所有操作 **按时间顺序** 存入数组中，然后开始分治。
- 在每一层分治中，利用数据结构（常见的是树状数组）统计当前查询的答案和 mid 之间的关系。
- 根据查询出来的答案和 mid 间的关系（小于等于 mid 和大于 mid）将当前处理的操作序列分为 $q1$ 和 $q2$ 两份，并分别递归处理。
- 当 $l=r$ 时，找到答案，记录答案并返回即可。

需要注意的是，在整体二分过程中，若当前处理的值域为 $[l,r]$，则此时最终答案范围不在 $[l,r]$ 的询问会在其他时候处理。

过程

注：

1. 为可读性，文中代码或未采用实际竞赛中的常见写法。

2. 若觉得某段代码有难以理解之处，请先参考之前题目的解释， 因为节省篇幅解释过的内容不再赘述。

从普通二分说起：

查询全局第 k 小

> **题 1** 在一个数列中查询第 k 小的数。

当然可以直接排序。如果用二分法呢？可以用数据结构记录每个大小范围内有多少个数，然后用二分法猜测，利用数据结构检验。

> **题 2** 在一个数列中多次查询第 k 小的数。

可以对于每个询问进行一次二分；但是，也可以把所有的询问放在一起二分。

先考虑二分的本质：假设要猜一个 $[l,r]$ 之间的数，猜测之后会知道是猜大了，猜小了还是刚好。当然可以从 l 枚举到 r，但更优秀的方法是二分：猜测答案是 $m=\lfloor\frac{l+r}{2}\rfloor$，然后去验证 m 的正确性，再调整边界。这样做每次询问的复杂度为 $O(\log n)$，若询问次数为 q，则时间复杂度为 $O(q\log n)$。

回过头来，对于当前的所有询问，可以去猜测所有询问的答案都是 mid，然后去依次验证每个询问的答案应该是小于等于 mid 的还是大于 mid 的，并将询问分为两个部分（不大于/大于），对于每个部分继续二分。注意：如果一个询问的答案是大于 mid 的，则在将其划至右侧前需更新它的 k，即，如果当前数列中小于等于 mid 的数有 t 个，则将询问划分后实际是在右区间询问第 $k-t$ 小数。如果一个部分的 $l=r$ 了，则结束这个部分的二分。利用线段树的相关知识，我们每次将整个答案可能在的区间 $[1,n]$（假设已经离散化）划分成了若干个部分，这样的划分共进行了 $O(\log n)$ 次，一次划分会将整个操作序列操作一次。若对整个序列进行操作，并支持对应的查询的时间复杂度为 $O(T)$，则整体二分的时间复杂度为 $O(T\log n)$。

参考代码如下：

实现

```cpp
struct Query {
  int id, k;  // 这个询问的编号，这个询问的 k
};

int ans[N], a[N];  // ans[i] 表示编号为i的询问的答案，a 为原数列
int val[N], cnt[N];  // 离散化后，记录对应的值及其计数（假设已经处理好）

// 返回原数列中值域在 [l,r] 中的数的个数
int check(int l, int r) {
  int res = 0;
  for (int i = l; i <= r; i++) {
    res += cnt[i];
  }
  return res;
}

// 整体二分
void solve(int l, int r, vector<Query> q) {
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = val[l];
    return;
  }
  vector<Query> q1, q2;
  int t = check(l, m);
  for (unsigned i = 0; i < q.size(); i++) {
    if (q[i].k <= t)
      q1.push_back(q[i]);
    else
      q[i].k -= t, q2.push_back(q[i]);
  }
  solve(l, m, q1), solve(m + 1, r, q2);
  return;
}
```

查询区间第 k 小

> **题 3** 在一个数列中多次查询区间第 k 小的数。

涉及到给定区间的查询，再按之前的方法进行二分就会导致 `check` 函数的时间复杂度爆炸。仍然考虑询问与值域中点 m 的关系：若询问区间内小于等于 m 的数有 t 个，询问的是区间内的 k 小数，则当 $k \leq t$ 时，答案应小于等于 m；否则，答案应大于 m。（注意边界问题）此处需记录一个区间小于等于指定数的数的数量，即单点加，求区间和，可用树状数组快速处理。为提高效率，只对数列中值在值域区间 $[l,r]$ 的数进行统计，即，在进一步递归之前，不仅将询问划分，将当前处理的数按值域范围划为两半。

参考代码（关键部分）

实现

```
struct Num {
  int p, x;
};  // 位于数列中第 p 项的数的值为 x

struct Query {
  int l, r, k, id;
};  // 一个编号为 id, 询问 [l,r] 中第 k 小数的询问

int ans[N];
void add(int p, int x);  // 树状数组, 在 p 位置加上 x
int query(int p);        // 树状数组, 求 [1,p] 的和
void clear();            // 树状数组, 清空

void solve(int l, int r, vector<Num> a, vector<Query> q)
// a中为给定数列中值在值域区间 [l,r] 中的数
{
  int m = (l + r) / 2;
  if (l == r) {
    for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = l;
    return;
  }
  vector<Num> a1, a2;
  vector<Query> q1, q2;
  for (unsigned i = 0; i < a.size(); i++)
    if (a[i].x <= m)
      a1.push_back(a[i]), add(a[i].p, 1);
    else
      a2.push_back(a[i]);
  for (unsigned i = 0; i < q.size(); i++) {
    int t = query(q[i].r) - query(q[i].l - 1);
    if (q[i].k <= t)
      q1.push_back(q[i]);
    else
      q[i].k -= t, q2.push_back(q[i]);
  }
  clear();
  solve(l, m, a1, q1), solve(m + 1, r, a2, q2);
  return;
}
```

下面提供 【模板】可持久化线段树 2 一题使用整体二分的，偏向竞赛风格的写法。

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 200020;
int n, m;
int ans[N];
// BIT begin
int t[N];
int a[N];

int sum(int p) {
  int ans = 0;
  while (p) {
    ans += t[p];
    p -= p & (-p);
  }
  return ans;
}

void add(int p, int x) {
  while (p <= n) {
    t[p] += x;
    p += p & (-p);
  }
}

// BIT end
int tot = 0;

struct Query {
  int l, r, k, id, type;  // set values to -1 when they are not
used!
} q[N * 2], q1[N * 2], q2[N * 2];

void solve(int l, int r, int ql, int qr) {
  if (ql > qr) return;
  if (l == r) {
    for (int i = ql; i <= qr; i++)
      if (q[i].type == 2) ans[q[i].id] = l;
    return;
  }
  int mid = (l + r) / 2, cnt1 = 0, cnt2 = 0;
  for (int i = ql; i <= qr; i++) {
    if (q[i].type == 1) {
      if (q[i].l <= mid) {
        add(q[i].id, 1);
        q1[++cnt1] = q[i];
      } else
        q2[++cnt2] = q[i];
```

```
    } else {
      int x = sum(q[i].r) - sum(q[i].l - 1);
      if (q[i].k <= x)
        q1[++cnt1] = q[i];
      else {
        q[i].k -= x;
        q2[++cnt2] = q[i];
      }
    }
  }
  // rollback changes
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 1) add(q1[i].id, -1);
  // move them to the main array
  for (int i = 1; i <= cnt1; i++) q[i + ql - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++) q[i + cnt1 + ql - 1] = q2[i];
  solve(l, mid, ql, cnt1 + ql - 1);
  solve(mid + 1, r, cnt1 + ql, qr);
}

pair<int, int> b[N];
int toRaw[N];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  // read and discrete input data
  for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    b[i].first = x;
    b[i].second = i;
  }
  sort(b + 1, b + n + 1);
  int cnt = 0;
  for (int i = 1; i <= n; i++) {
    if (b[i].first != b[i - 1].first) cnt++;
    a[b[i].second] = cnt;
    toRaw[cnt] = b[i].first;
  }
  for (int i = 1; i <= n; i++) {
    q[++tot] = {a[i], -1, -1, i, 1};
  }
  for (int i = 1; i <= m; i++) {
    int l, r, k;
    cin >> l >> r >> k;
    q[++tot] = {l, r, k, i, 2};
  }
  solve(0, cnt + 1, 1, tot);
  for (int i = 1; i <= m; i++) cout << toRaw[ans[i]] << '\n';
}
```

带修区间第 k 小

> **题 4** Dynamic Rankings 给定一个数列，要支持单点修改，区间查第 k 小。

修改操作可以直接理解为从原数列中删去一个数再添加一个数，为方便起见，将询问和修改统称为「操作」。因后面的操作会依附于之前的操作，不能如题 3 一样将统计和处理询问分开，故将所有操作存于一个数组，用标识区分类型，依次处理每个操作。为便于处理树状数组，修改操作可分拆为擦除操作和插入操作。

优化

1. 注意到每次对于操作进行分类时，只会更改操作顺序，故可直接在原数组上操作。具体实现，在二分时将记录操作的 q, a 数组换为一个大的全局数组，二分时记录信息变为 L, R，即当前处理的操作是全局数组上的哪个区间。利用临时数组记录当前的分类情况，进一步递归前将临时数组信息写回原数组。
2. 树状数组每次清空会导致时间复杂度爆炸，可采用每次使用树状数组时记录当前修改位置（这已由 1 中提到的临时数组实现），本次操作结束后在原位置加 -1 的方法快速清零。
3. 一开始对于数列的初始化操作可简化为插入操作。

参考代码（关键部分）

实现

```cpp
struct Opt {
  int x, y, k, type, id;
  // 对于询问, type = 1, x, y 表示区间左右边界, k 表示询问第 k 小
  // 对于修改, type = 0, x 表示修改位置, y 表示修改后的值,
  // k 表示当前操作是插入(1)还是擦除(-1), 更新树状数组时使用.
  // id 记录每个操作原先的编号, 因二分过程中操作顺序会被打散
};

Opt q[N], q1[N], q2[N];
// q 为所有操作,
// 二分过程中, 分到左边的操作存到 q1 中, 分到右边的操作存到 q2 中.
int ans[N];
void add(int p, int x);
int query(int p);  // 树状数组函数, 含义见题3

void solve(int l, int r, int L, int R)
// 当前的值域范围为 [l,r], 处理的操作的区间为 [L,R]
{
  if (l > r || L > R) return;
  int cnt1 = 0, cnt2 = 0, m = (l + r) / 2;
  // cnt1, cnt2 分别为分到左边, 分到右边的操作数
  if (l == r) {
    for (int i = L; i <= R; i++)
      if (q[i].type == 1) ans[q[i].id] = l;
    return;
  }
  for (int i = L; i <= R; i++)
    if (q[i].type == 1) {  // 是询问: 进行分类
      int t = query(q[i].y) - query(q[i].x - 1);
      if (q[i].k <= t)
        q1[++cnt1] = q[i];
      else
        q[i].k -= t, q2[++cnt2] = q[i];
    } else
      // 是修改: 更新树状数组 & 分类
      if (q[i].y <= m)
        add(q[i].x, q[i].k), q1[++cnt1] = q[i];
      else
        q2[++cnt2] = q[i];
  for (int i = 1; i <= cnt1; i++)
    if (q1[i].type == 0) add(q1[i].x, -q1[i].k);  // 清空树状数组
  for (int i = 1; i <= cnt1; i++) q[L + i - 1] = q1[i];
  for (int i = 1; i <= cnt2; i++)
    q[L + cnt1 + i - 1] = q2[i];  // 将临时数组中的元素合并回原数组
  solve(l, m, L, L + cnt1 - 1), solve(m + 1, r, L + cnt1, R);
  return;
}
```

针对静态序列的优化

> **题 5** 【模板】可持久化线段树 2 给定一个序列，区间查询第 k 小。

树套树和整体二分实现带修区间第 k 小问题的复杂度都为 $O(n\log^2 n)$，但静态区间第 k 小问题可以使用可持久化线段树在 $O(n\log n)$ 时间复杂度内解决，而几乎所有整体二分实现的静态区间第 k 小问题代码时间复杂度都是 $O(n\log^2 n)$，面对大数据范围时存在 TLE 的风险。（这里默认值域与序列长度同阶，值域与序列长不同阶的情况可以通过离散化转化为同阶情况）

优化

1. 对于每一轮划分，如果当前数列中小于等于 mid 的数有 t 个，则将询问划分后实际是在右区间询问第 $k-t$ 小数，因此对划分到右区间的询问做出了修改。如果答案的原始值域为 $[L,R]$，某次划分的答案值域为 $[l,r]$，那么对于参与此次划分的询问，$[L,l)$ 中所有数值对它们的影响已经在之前被消除了。
2. 由于需要使每轮划分都仅和当前答案值域 $[l,r]$ 有关，树状数组需要多次载入和清空。

如果划分不仅仅和当前答案值域有关呢?

由此可以得到一个与全局序列有关的优化方法：维护一个指针 pos 追踪每轮划分的 mid（分治中心），将所有 $\le pos$ 的元素对应的下标在树状数组中置为 1，树状数组的其余位置置为 0。每次划分之前移动 pos 并更新树状数组。指针 pos 移动的次数与 $n\log n$ 同阶。划分时对每一个询问查询树状数组中对应区间的值，满足则划分至左区间，否则划分至右区间，**不需要对询问做出修改**。

由于要追踪分治中心，需要让 pos 准确地更新树状数组。在整体二分之前将序列按元素大小排序并记录元素对应下标，指针移动时在树状数组中对下标进行相应修改。对于绝大多数 **可以用整体二分解决并且不带修改的问题**，都可以应用此种优化以大幅降低数据结构的使用次数。

由于减少了很多树状数组的载入和清空操作，应用这种优化通常情况下会明显提升整体二分的效率（即使只是常数优化），对于静态区间第 k 小值问题而言效率完全不差于时间复杂度更优的可持久化线段树。值得注意的是，对于静态区间第 k 小值问题也存在时间复杂度 $O(n\log n)$ 的整体二分实现。

参考代码（关键部分）

实现

```cpp
struct Query {
  int i, l, r, k;
};  // 第 i 次询问查询区间 [l,r] 的第 k 小值

Query s[200005], t1[200005], t2[200005];
int n, m, cnt, pos, p[200005], ans[200005];
pair<int, int> a[200005];

void add(int x, int y);  // 树状数组 位置 x 加 y
int sum(int x);          // 树状数组 [1,x] 前缀和

// 当前处理的询问为 [l,r],答案值域为 [ql,qr]
void overall_binary(int l, int r, int ql, int qr) {
  if (l > r) return;
  if (ql == qr) {
    for (int i = l; i <= r; i++) ans[s[i].i] = ql;
    return;
  }
  int cnt1 = 0, cnt2 = 0, mid = (ql + qr) >> 1;
  // 追踪分治中心,认为 [1,pos] 的值已经载入树状数组
  while (pos <= n - 1 && a[pos + 1].first <= mid)
    add(a[pos + 1].second, 1), ++pos;
  while (pos >= 1 && a[pos].first > mid) add(a[pos].second, -1),
--pos;

  for (int i = l; i <= r; i++) {
    int now = sum(s[i].r) - sum(s[i].l - 1);
    if (s[i].k <= now)
      t1[++cnt1] = s[i];
    else
      t2[++cnt2] = s[i];  // 注意 不应修改询问信息
  }
  for (int i = 1; i <= cnt1; i++) s[l + i - 1] = t1[i];
  for (int i = 1; i <= cnt2; i++) s[l + cnt1 + i - 1] = t2[i];

  overall_binary(l, l + cnt1 - 1, ql, mid);
  overall_binary(l + cnt1, r, mid + 1, qr);
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    scanf("%d", &a[i].first);
    a[i].second = i;
    p[++cnt] = a[i].first;
  }
  sort(a + 1, a + n + 1);  // 对序列排序 离散化
  sort(p + 1, p + n + 1);
  cnt = unique(p + 1, p + n + 1) - p - 1;
```

```
    for (int i = 1; i <= n; i++)
      a[i].first = lower_bound(p + 1, p + cnt + 1, a[i].first) - p;
    // 省略读入询问
    overall_binary(1, m, 1, cnt);
    for (int i = 1; i <= n; i++) printf("%d\n", p[ans[i]]);
    return 0;
}
```

区间前驱后继

> **题 6** 在一个数列中多次查询 k 在区间中的前驱（严格小于 k，且最大的数）或后继（严格大于 k，且最小的数），保证存在这样的数。

以前驱为例，使用数据结构解决此种问题的方法一般是先查询区间内有多少严格小于 k 的数（设它们的数量为 x），再查询区间第 x 小的数。后继则是查询区间内有多少不大于 k 的数（数量为 x），然后查询区间第 $x+1$ 小的数。

考虑使用整体二分解决这个问题：整体二分是一种高效求解区间第 k 小的离线算法，而 CDQ 分治 可以离线高效求解区间内的排名。先跑一遍 CDQ 分治求出排名就可以使用整体二分得到区间内部的前驱和后继了。

此问题还可以用 CDQ 分治套线段树离线一遍解决，但效率远低于跑两遍的 CDQ 分治 + 整体二分。

构造单调性序列

> **题 7** Sequence 给定一个序列，每次操作可以把某个数 $+1$ 或 -1。要求把序列变成单调不降的，并且修改后的数列只能出现修改前的数，输出最小操作次数。

此类题目也可以使用动态规划或反悔贪心解决。

在满足操作次数最小化的前提下，一定存在一种方案使得最后序列中的每个数都是序列修改前存在的，这个结论可以使用数学归纳法证明。由于题目并不需要最终序列的信息，问题转化为求出最小操作次数。

由于要求最终的序列单调不降，可以使用整体二分。每轮整体二分判定最终序列区间 $[l,r]$ 的值域，此时答案的值域为 $[ql,qr]$。令 $mid=\lfloor\frac{ql+qr}{2}\rfloor$，每轮二分开始时默认将所有数划分至 $[mid+1,qr]$（要划分到 $[ql,mid]$ 的数设为 0 个），初始代价设为将序列区间 $[l,r]$ 全部置为 $mid+1$ 的操作次数。依次枚举区间 $[l,r]$ 中的数 i 并且计算将 $[l,i]$ 置为 mid、将 $[i+1,r]$ 置为 $mid+1$ 的操作次数之和，如果优于之前的操作次数则更新最少操作次数和要划分到 $[ql,mid]$ 的数的个数。

划分时已经保证了最终序列的单调性不被破坏，同时因为每次都取最小操作次数，最终被划分至左区间的数取 mid 一定比取 $mid+1$ 更优，故整体二分得到的序列一定是单调不降且操作次数最小的。计算操作次数输出即可。

参考代码（关键部分）

实现

```cpp
int a[500005], ans[500005];  // a:原序列 ans:构造的序列

void overall_binary(int l, int r, int ql, int qr) {
  if (l > r) return;
  if (ql == qr) {
    for (int i = l; i <= r; i++) ans[i] = ql;
    return;
  }
  int cnt = 0,
      mid = ql + ((qr - ql) >> 1);  // 默认开始都填 mid+1 全部划分到右区间
  long long res = 0ll, sum = 0ll;
  for (int i = l; i <= r; i++) sum += abs(a[i] - (mid + 1));
  res = sum;
  for (int i = l; i <= r;
       i++) {  // 尝试把 [l,i] 从 mid+1 换成 mid 并且划分到左区间
    sum -= abs(a[i] - (mid + 1));
    sum += abs(a[i] - mid);
    if (sum < res) cnt = i - l + 1, res = sum;  // 发现 [l,i] 取 mid 更优,更新
  }
  overall_binary(l, l + cnt - 1, ql, mid);
  overall_binary(l + cnt, r, mid + 1, qr);
}
```

参考习题

「国家集训队」矩阵乘法

「POI2011 R3 Day2」流星 Meteors

二逼平衡树

[BalticOI 2004] Sequence 数字序列

参考资料

- 许昊然《浅谈数据结构题几个非经典解法》