以ZJOI2013K大数查询为例。

我们需要动态维护的区间第k大，传统做法下，区间内部的第k大我们需要额外的高级数据结构进行维护，因此产生了很多优秀的树套树算法。是否我们存在优秀的算法可以避免维护这k个数。

复杂度的分析我们在最后进行，首先我们仅从思维的进化程度来讨论算法过程。

一、针对任意一个询问，我们取二分答案m（二分的是询问中第k大那个数），那么显然针对操作序列：

[a, b] 添加 c

如果c >= m 显然添加的c会影响询问，故该操作应该生效，反之则没有影响，此处我们可以使用传统的区间线段树加以统计，[a, b]线段+1（为什么是+1而不是c）

[a, b] 询问第k大

和传统的二分答案一样，此处已经变成一个判定行问题，针对询问[a, b]，我们只需要看该线段的计数，如果>=k，则意味着答案应该比m大，此时往右二分，否则往左。

二、针对多个询问，显然每个询问重新额二分一次是会超时的，此时我们思考可否在二分时候带着询问操作一起做一些事情。于是容易想到：

[a, b] 添加 c

C >= m 操作生效，加进线段树，将该操作靠右（可通过赋1方便实现）

C < m 操作在小的m时才会生效，直接将该操作靠左（赋0）

[a, b]询问k大

返回当前线段树中a, b的计数sum，

K >= sum 询问在>=m的答案下可能生效，故该询问靠右

K < sum 询问在<m的答案下方能生效，询问靠左

按照左右分组排序，左右分段分开，左右半段分治下去即可。

代码框架如下

Solve(int opt\_f, int opt\_l, int l, int r)

If l == r

Each opt in opt\_f to opt\_l | type is queue

Ans[opt] = l

m = (l + r) >> 1

Each opt in opt\_f to opt\_l

…

Sort(opt\_f, opt\_l)

Solve(opt\_f, opt\_f + leftnum, l, m -1)

Solve(opt\_f+leftnum+1, opt\_l, m, r)

复杂度分析：

通过对操作序列的划分

M logn logn

M为操作数，第一个log为线段树，第二个log为二分答案 应该是这样的！恩！