动态求解区间最大数的问题

揭开它神秘的面纱，探寻真理的奥妙。——小志。

这偏文章主要总结划分树解决区间第 K 大数的问题。一些内容并非原创，全部内容只供参考交流！

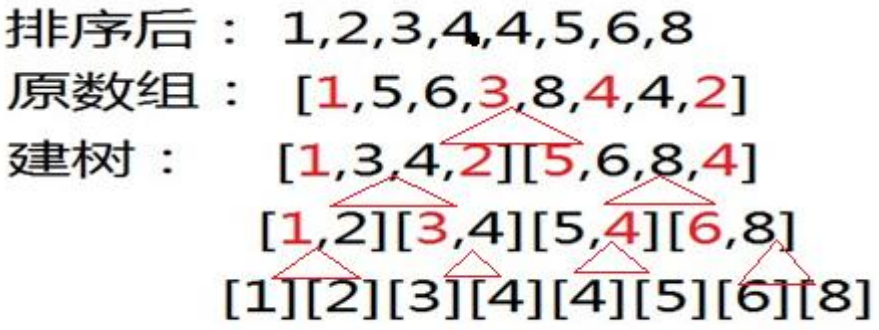
划分树

- **划分树的定义**

  划分树定义为，她的每一个节点保存区间 [lft, rht]所有元素，元素排列顺序与原数组（输入）相同，但是，两个子树的元素为该节点所有元素排序后$(rht - lft + 1) / 2$个进入左子树，其余的到右子树，同时维护一个 num 域，num[i] 表示$lft \rightarrow i$ 这些点有多少进入了左子树。（摘自某牛人的博客，为了和下面的算法统一，稍有变动）

- **划分树的 Sample.**

  如果由下而上看这个图，我们就会发现他和归并排序的（归并树）的过程很类似，或者说正好相反。归并树是由下而上的排序，而她确实由上而下的排序（观察‘4’的运动轨迹，我们可以猜到，划分树的排序也是一种稳定的排序方法，这里不是说明的重点，不予证明）。但这正是她可以用来解决第 k 大元素的理由所在。(具体的理由，写完再补)



- 划分树的存储结构（采用层次存储结构（由下而上，由左到右，每层两个孩子，见上图）

```
using namespace std;
#define M 100001
#define md(x, y) (((x)+(y))>>1)
int sorted[M];        // 对原来集合中的元素排序后的值。
struct node {
    int val[M];       // val 记录第 k 层当前位置的元素的值
    int num[M];       // num 记录元素所在区间的当前位置之前进入左孩子的个数
    double sum[M];    // sum 记录比当前元素小的元素的和。
}t[20];
```

- **划分树的建立 Build**

  划分树的建立和普通的二叉树的建立过程差不多，仍然采取中序的过程（先根节点，然后左右孩子）。

  树的建立相对比较简单，我们依据的是已经排好序的位置进行建树，所以先用快排将原集合牌还序。要维护每个节点的 num 域。

```
void build(int lft, int rht, int p) {
    if(lft == rht) return;
    int i, mid = md(lft, rht);
    int isame = mid - lft + 1, same = 0;
/* isame 用来标记和中间值 val_mid 相等的，且分到左孩子的数的个数。
```

```
    初始时，假定当前区间[lft,rht]有 mid-lft+1 个和 val_mid 相等。
    先踢掉比中间值小的,剩下的就是要插入到左边的*/
     for(i = lft; i <= rht; i++)
         if(t[p].val[i] < sorted[mid]) isame--;
     int ln = lft, rn = mid + 1;
     for(i = lft; i <= rht; i++) {
         if(i == lft) {            // 初始一个子树。
             t[p].num[i] = 0;
             t[p].sum[i] = 0;
         } else {                  // 初始区间下一个节点。
             t[p].num[i] = t[p].num[i-1];
             t[p].sum[i] = t[p].sum[i-1];
         }
  /* 如果大于，肯定进入右孩子，否则，判断是否还有相等的应该进入左孩子的，
    没有，就直接进入右孩子，否则进入左孩子，同时更新节点的 sum 域和 num 域*/
         if(t[p].val[i] < sorted[mid]) {
             t[p].num[i]++;
             t[p].sum[i] += t[p].val[i];
             t[p+1].val[ln++] = t[p].val[i];
         }else if(t[p].val[i] > sorted[mid]) {
             t[p+1].val[rn++] = t[p].val[i];
         }else {
             if(same < isame) {
                 same++;
                 t[p].num[i]++;
                 t[p].sum[i] += t[p].val[i];
                 t[p+1].val[ln++] = t[p].val[i];
             }else {
                 t[p+1].val[rn++] = t[p].val[i];
             }
         }
     }
     build(lft, mid, p+1);
     build(mid+1, rht, p+1);
```

- }划分树的查找

  在区间[a,b]上查找第 k 大的元素，同时返回她的位置和去见里面小于[a, b]的所有数的和。

  1. 如果t[p].num[b] − t[p]. num[a − 1] ≥ k, 即，进入 p 的左孩子的个数已经超过 k 个，那么就往左孩子里面查找，同时更新{a, b} ==> {$lft + t$[p].num[a − 1],, lft + t[p].num[b] − 1}
  2. 如果t[p].num[b] − t[p]. num[a − 1] < k, 即，进入 p 的左孩子的个数小于 k 个，那么就要往右孩子查找第 k – s (s 表示进入左孩子的个数)个元素。 同时要更新 sun 域。
  3. 详细的过程见代码和注释：

```
/* 在区间[a, b]上查找第 k 大元素，同时 Sum 返回区间[a, b]中小于第 k 大元素的和。*/
double Sum = 0;
int query(int a, int b, int k, int p, int lft, int rht) {
    if(lft == rht) return t[p].val[a];
/* 到达叶子就找到该元素，返回。
```

```
    s  记录区间[a, b]中进入左孩子的元素的个数。
    ss 记录区间[lft, a-1)中计入左孩子的元素的个数。
    sss 记录区间[a, b]中小于第 k 大元素的值的和。
    b2 表示[lft, a-1]中分到右孩子的个数
    bb 表示[a, b] 中分到右孩子的个数。*/
     int s, ss, b2, bb, mid = md(lft, rht);  double sss;
//区间端点点重合的情况, num[a] 不一定为，所以要单独考虑
    if(a == lft) {
        s = t[p].num[b];
        ss = 0;
        sss = t[p].sum[b];
    } else {
        s = t[p].num[b] - t[p].num[a-1];
        ss = t[p].num[a-1];
        sss = t[p].sum[b] - t[p].sum[a-1];
    }
    if(s >= k) {    // 进入左孩子，同时更新区间端点值。
        a = lft + ss;
        b = lft + ss + s - 1;
        return query(a, b, k, p+1, lft, mid);
    } else {
        bb = a - lft - ss;
        b2 = b - a - 1 - s;
        a = mid + bb + 1;
        b = mid + bb + b2;
        Sum += sss;
        return query(a, b, k-s,p+1, mid+1, lft);
     }
}
```

- 划分树的应用

pku_2401_K-th Number

hdu_2665_Kth number

前两个是基础，后一个是上面所得带 sum 域的加强版。

hdu_3743_Minium_Number