21级实验室暑假第八讲

目录

离线思想

算法可以从这样的一个维度分成两类：**在线算法和离线算法**。
什么叫在线算法？就是依次处理每一个 query，对每一个 query 的计算，和之后的 query 无关，也不会用到之后的 query 信息（但可能也可以使用之前的 query 信息）。

所以，在线算法，可以用来处理数据流。算法不需要一次性地把所有的 query 都收集到再处理。大家也可以想象成：把这个算法直接部署到线上，尽管在线上可能又产生了很多新的 query，也不影响，算法照常运行。

离线算法则不同。离线算法需要把所有的信息都收集到，才能运行。处理当前 query 的计算过程，可能需要使用之后 query 的信息。

离线思想

举两个最简单的例子来说明。

以排序算法为例，插入排序算法是一种**在线算法**。因为可以把插入排序算法的待排序数组看做是一个数据流。插入排序算法顺次**把每一个数据插入到当前排好序数组部分的正确位置**。在排序过程中，即使后面源源不断来新的数据也不怕，整个算法照常进行。

选择排序算法则是一种**离线算法**。因为选择排序算法一上来要**找到整个数组中最小(大)的元素；然后找第二小(大)元素；以此类推**。这就要求不能再有新的数据了。因为刚找到最小(大)元素，再来的新数据中有更小(大)的元素，之前的计算就不正确了。

离线思想

再举一个例子，对于 topK 问题（找前 k 小或者前 k 大的元素）。

使用**一个大小为 k 的优先队列是在线算法，虽然时间复杂度是 O(nlogk)**，但整个算法不需要一次性知道所有的数据，可以处理数据流；

使用快排的思想做改进，topK 问题可以在 O(n) 时间解决。但这是**一个离线的算法。初始必须知道所有的数据，才能完成**。

分块

回顾“区间”问题，前面给出了**暴力法、树状数组、线段树**等算法。给定一个保存n个数据的数列，做m次“区间修改”和“区间查询”，每次操作只涉及到部分区间。**暴力法只是简单地从整体上做修改和查询，复杂度O(mn)**，很低效。树状数组和线段树都用到了二分的思想，以O(logn)的复杂度组织数据结构，每次只处理涉及到的区间，从而**实现了O(mlogn)的高效的复杂度**。

虽然暴力法只能解决小规模的问题，但是它的代码非常简单。

分块

有一种代码比树状数组、线段树简单，效率比暴力法高的算法，称为“分块”，它能以$O(M\sqrt{N})$的复杂度解决“区间修改+区间查询”问题。简单地说，**分块是用线段树的“分区”思想改良的暴力法**；它把**数列分成很多“块”，对涉及到的块做整体性的维护操作（类似于线段树的lazy-tag）**，而不是像普通暴力法那样处理整个数列，从而提高了效率。

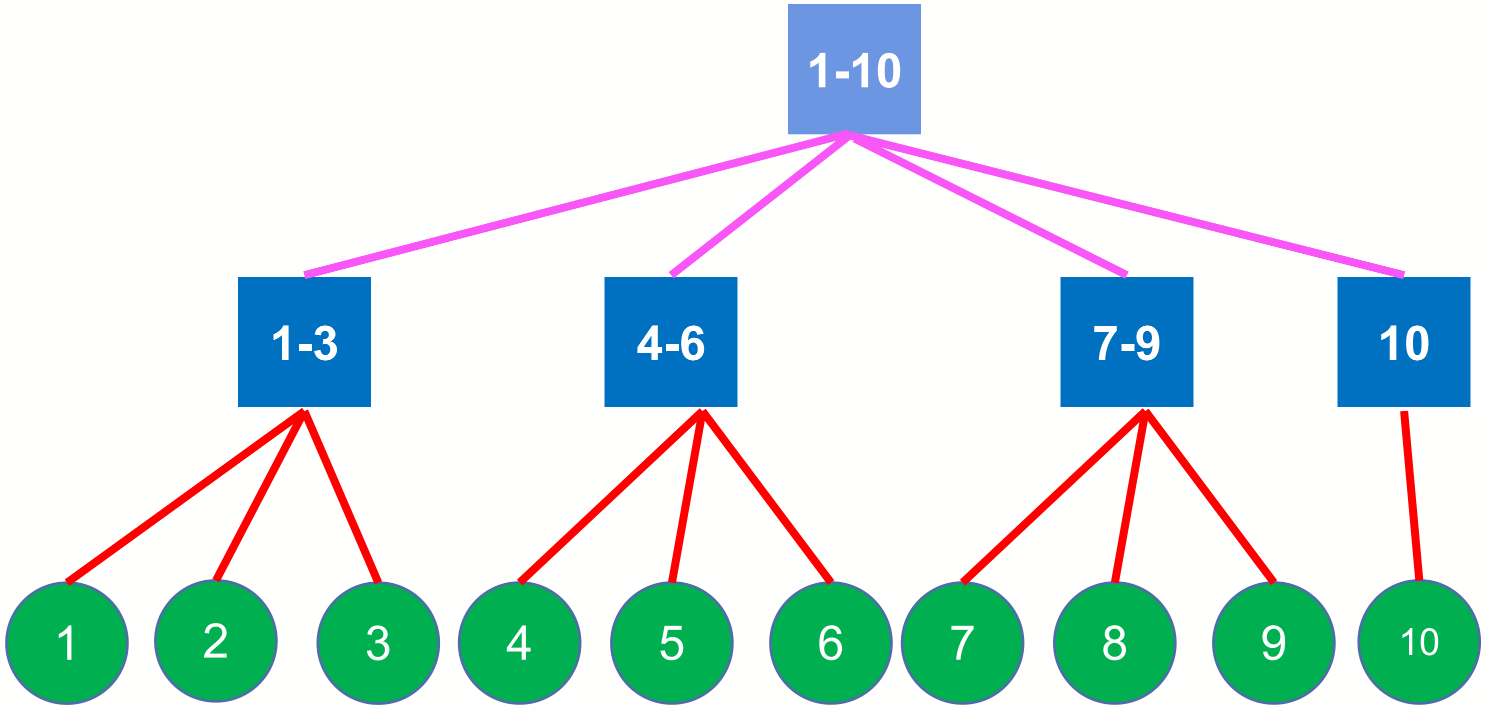
分块

用一个长度为n的数组来存储n个数据，把它分为t块，每块长度为n/t。下图是一个有10个元素的数组，共分成4块，前3块每块3个元素，最后一块1个元素。

数组	1	2	3	4	5	6	7	8	9	10
分块	1			2			3			4

分块

数组	1	2	3	4	5	6	7	8	9	10
分块	1			2			3			4



对比分块与线段树，线段树是一棵高度为logn的树，分块可以看成一棵高度为3的树。，在**线段树上做一次操作是O(logn)的**，因为它有logn层；**分块是O(n/t)的**，因为它把数据分成了t块，处理一块的时间是n/t的。下面介绍分块，并详细说明复杂度。

分块

块操作的基本要素有：

（1）块的大小。用block表示。

（2）块的数量。用t表示。

（3）块的左右边界。定义数组st[]、ed[]，用st[i]、ed[i]表示块i的第一个和最后一个元素的位置。

$$
\begin{aligned}
&st[1]=1, ed[1]=block;\\
&st[2]=block+1, ed[2]=2\times block;\\
&\cdots;\\
&st[i]=(i-1)\times block, ed[i]=i\times block;
\end{aligned}
$$

（4）每个元素所属的块。定义pos[]，pos[i]表示第i个元素所在的块。

$$pos[i]=(i-1)/block+1$$

分块

块操作的基本要素有：

（1）块的大小。用block表示。

（2）块的数量。用t表示。

（3）块的左右边界。定义数组st[]、ed[]，用st[i]、ed[i]表示块i的第一个和最后一个元素的位置。

$$
\begin{aligned}
& st[1]=1, ed[1]=block; \\
& st[2]=block+1, ed[2]=2\times block; \\
& \cdots; \\
& st[i]=(i-1)\times block, ed[i]=i\times block;
\end{aligned}
$$

（4）每个元素所属的块。定义pos[]，pos[i]表示第i个元素所在的块。

$$pos[i]=(i-1)/block+1$$

分块

其中**每块的大小block的值等于$\sqrt{n}$取整(有时候取$\sqrt{n}$ 不是最优值)**，后面的“复杂度分析”会说明原因。如果$\sqrt{n}$的结果不是整数，那么最后要加上一小块

```
int block=sqrt(n);//块的大小,每块有block个元素
int t=n/block;//块的数量,共分为t块
if(n%block!=0)//sqrt(n)的结果不是整数,最后加上一小块
    t++;
for(int i=1;i<=t;i++){//遍历块
    st[i]=(i-1)*block+1;
    ed[i]=i*block;
}
ed[t]=n;//sqrt(n)的结果不是整数,将右端点更新为n
for(int i=1;i<=n;i++)//遍历所有元素的位置
    pos[i]=(i-1)/block+1;
```

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

首先定义区间有关的辅助数组:

（1）定义数组w[]存储数据，共n个元素，读取初值，存储在w[1]、w[2]、…、w[n]中。

（2）定义sum[]，sum[i]为第i块的区间和，并预处理出初值。

```
//方法一:遍历n个元素
for(int i=1;i<=n;i++)
    sum[pos[i]]+=w[i];
////方法二:遍历所有的块
for(int i=1;i<=t;i++)
    for(int j=st[i];j<=ed[i];j++)//遍历块内所有元素
        sum[i]+=w[j];
```

（3）定义add[]，add[i]为第i块的增量标记(当整块修改才更新)，初始值为0。

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

1. 区间修改：将区间[l, r]内每个数加上num。

情况1: [l, r]在**某个i块之内**，即[l, r]是一个“碎片”。暴力修改[l,r]的值，**把a[l]、a[l+1]、…、a[r]逐个加上num，更新** $sum[i] = sum[i] + num \times (r - l + 1)$ 。计算次数约为n/t(即block)。

情况2: [l, r]跨越了多个块。在被[l, r]完全包含的那些整块内（设有k个块），更新 $add[i] = add[i] + num$ 。对于不能完全包含的那些碎片（它们在k个整块的两头），按情况1处理。情况2的计算次数约为n/t + k，1 ≤ k ≤ t。

例如修改[2,7]之间的值，即暴力修改区间[2,3]和[7,7]的值(情况1),而剩下的整块区间[4,6]只把懒标记更新(情况2)

数组	1	2	3	4	5	6	7	8	9	10
分块		1			2			3		4

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

1. 区间修改：将区间[l, r]内每个数加上num。

总结以上两种情况：

- 处理碎块时，只更新sum[i]和对应区间的a[]，不更新add[i]；
- 处理整块时，只更新add[i]，不更新sum[i]。

例如修改[2,7]之间的值，即暴力修改区间[2,3]和[7,7]的值(情况1),而剩下的整块区间[4,6]只把懒标记更新(情况2)

数组	1	2	3	4	5	6	7	8	9	10
分块	1			2			3			4

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

1. 区间修改：将区间[l, r]内每个数加上num。

- 处理碎块时，只更新sum[i]和对应区间的a[]，不更新add[i]；
- 处理整块时，只更新add[i]，不更新sum[i]。

```
void modify(int l,int r,int num){
    int lpos=pos[l],rpos=pos[r];//lpos对应l的块,rpos对应r的块
    if(lpos==rpos){//情况1:处理碎片块,l,r的块相同,暴力处理
        for(int i=l;i<=r;i++)//更新区间[l,r]的w[i]
            w[i]+=num;
        sum[lpos]+=num*(r-l+1);//更新块的sum
    }
```

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

1. 区间修改：将区间[l, r]内每个数加上num。

- 处理碎块时，只更新sum[i]和对应区间的a[]，不更新add[i]；
- 处理整块时，只更新add[i]，不更新sum[i]。

```
    else{
        for(int i=l;i<=ed[lpos];i++)
       //情况1:处理碎片块,l所属的块暴力处理,更新[l,ed[lpos]]块的w[i]
            w[i]+=num;
        sum[lpos]+=num*(ed[lpos]-l+1);//更新l所属块的sum
        for(int i=lpos+1;i<=rpos-1;i++)//情况2:处理整块,更新[lpos+1,rpos-1]整块的add
            add[i]+=num;
        for(int i=st[rpos];i<=r;i++)
       //情况1:处理碎片块,r所属的块暴力处理,更新[st[rpos],r]块的w[i]
            w[i]+=num;
        sum[rpos]+=num*(r-st[rpos]+1);//更新r所属块的sum
    }
}
```

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

2. 区间查询：输出区间[l, r]内每个数的和。

情况1: [l, r]在某个i块之内。**暴力加每个数，最后加上add[i]**，答案是

$$res = a[l] + a[l+1] + \cdots + a[r] + add[i] \times (r - l + 1)$$

，计算次数约为n/t。

情况2: [l, r]跨越了多个块。在被[l, r]完全包含的那些块内（设有k个块）

$$res += sum[i] + add[i] \times len[i]$$

，其中len[i]是第i段的长度，等于n/t。对于不能完全包含的那些碎片，按情况1处理，然后与res累加。计算次数约为n/t + k，1 ≤ k ≤ t。

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

2. 区间查询：输出区间[l, r]内每个数的和。

```
ll query(int l,int r){
    int lpos=pos[l],rpos=pos[r];//lpos对应l的块,rpos对应r的块
    ll res=0;
    if(lpos==rpos){//情况1:处理碎片块,l,r的块相同,暴力处理
        for(int i=l;i<=r;i++)//区间[l,r]求和
            res+=w[i];
        res+=add[lpos]*(r-l+1);//加上懒标记
    }
```

分块

用分块解决区间问题很方便，下面以“区间修改+区间查询”为例。

2. 区间查询：输出区间[l, r]内每个数的和。

```
    else{
        for(int i=l;i<=ed[lpos];i++)//情况1:处理碎片块,l所属的块暴力处理,区间[l,ed[lpos]]求和
            res+=w[i];
        res+=add[lpos]*(ed[lpos]-l+1);//加上懒标记

        for(int i=lpos+1;i<=rpos-1;i++)//情况2:处理整块
            res+=sum[i]+add[i]*(ed[i]-st[i]+1);

        for(int i=st[rpos];i<=r;i++)//情况1:处理碎片块,r所属的块暴力处理,区间[st[rpos],r]求和
            res+=w[i];
        res+=add[rpos]*(r-st[rpos]+1);//加上懒标记
    }
    return res;//返回结果
}
```

分块

分块算法的实现简单粗暴，没有复杂数据结构和复杂逻辑，很容易编码。

分块算法的思想，可以概况为“整块打包维护，碎片逐个枚举”。

把数列分为t块，t取何值时有最佳效果？

观察一次操作的计算次数n/t和n/t+k，其中1≤k≤t；当$t=\sqrt{n}$时，有较好的时间复杂度$O(\sqrt{n})$。m次操作的复杂度是$O(m\sqrt{n})$，适合求解m=n=10^5规模的问题，或$O(m\sqrt{n})\approx 10^7$的问题。

分块只能解决m = n = 10^5规模的问题，而线段树是10^6规模的，应用场景不同.

分块

四种方式对比:

	复杂度	时间	内存	代码	优劣
树状数组	$O((N+Q)\log N)$	1.0s	3MB	850B	效率高、代码短 不易扩展、不太直观
线段树	$O((N+Q)\log N)$	1.5s	7MB	1700B	效率较高、扩展性好 代码较长、直观性一般
分块	$O((N+Q)\sqrt{N})$	1.9s	1.5MB	1500B	通用、直观 效率偏低、码长一般
朴素(暴力)	$O((N+Q)\times N)$	TLE	1MB	500B	略

莫队算法

莫队算法是由莫涛(2010年信息学国家集训队队员)提出的算法。在莫涛提出莫队算法之前，莫队算法已经在 Codeforces 的高手圈里小范围流传，但是莫涛是第一个对莫队算法进行详细归纳总结的人。莫涛提出莫队算法时，只分析了**普通莫队算法(讲,其他版本不讲)**，但是经过 OIer 和 ACMer 的集体智慧改造，莫队有了多种扩展版本(带修改莫队，树上莫队，回滚莫队)。

莫队算法可以解决一类离线区间询问问题，适用性极为广泛。同时将其加以扩展，便能轻松处理树上路径询问以及支持修改操作。

莫队算法

“离线”和“在线”的概念。在线是交互式的，一问一答；如果前面的答案用于后面的提问，称为“强制在线”。离线是非交互的，一次性读取所有问题，然后一起回答，"记录所有步，回头再做”。

基础的莫队算法是一种离线算法，它通常用于**不修改只查询的一类区间**问题，复杂度为$O(n\sqrt{n})$，没有在线算法线段树或树状数组好，但是编码很简单。

莫队算法

HH的项链

题目描述：给定一个数量，询问某个区间内不同的数有多少个。

输入：第一行一个正整数 n，表示数列长度。第二行n个正整数 ai。第三行一个整数m，表示HH 询问的个数。接下来 m 行，每行两个整数 L,R，表示询问的区间。

输出：输出m行，每行一个整数，依次表示询问对应的答案。

题目询问区间内不同的数有多少个，即去重后数字的个数，本题的标准解法是**线段树或树状数组**。下面首先给出暴力法，然后再引导出莫队算法。

莫队算法

HH的项链

(洛谷上也有该题，但是在2021年被lxl加了数据卡了莫队，~~各种乱搞优化似乎都过不了~~子)

题目描述：给定一个数量，询问某个区间内不同的数有多少个。

1. 暴力法

可以用**STL的unique()函数去重，一次耗时O(n)，m次的总复杂度O(mn)**。或者自己编码， 用扫描法统计数字出现的次数，这是一种简单易行的暴力法。

莫队算法

（1）查询**一个区间有多少个不同的数字**

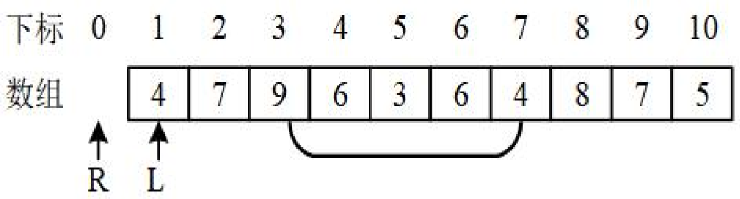
定义cnt[]，cnt[x]表示数字x出现的次数；定义答案为ans，即区间内不同的x有多少个。

用指针L、R单向扫描，从数列的头扫到尾，L最终落在查询区间的最左端，R落在区间最右端。L往右每扫一个数x，就把它出现的次数cnt[x]减去1；R往右每扫到一个数x，就把它出现的次数cnt[x]加上1。扫描完区间后，cnt[x]的值就是x在区间内出现的次数。若cnt[x]=1说明x第1次出现，ans加1；若cnt[x]变为0，说明它从区间里消失了，ans减1。

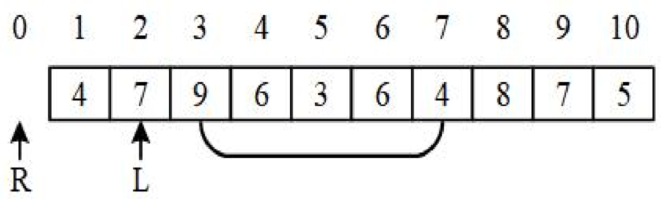
莫队算法

（1）查询**一个区间有多少个不同的数字**

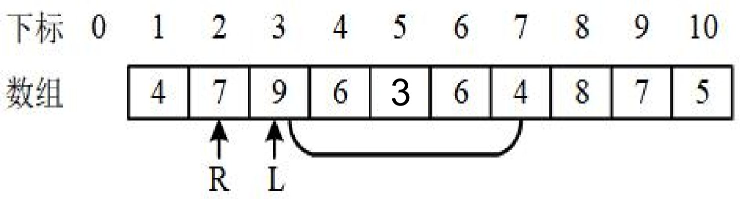
下面的例子是统计区间[3, 7]内有多少不同的数字，初始指针L=1，R=0。



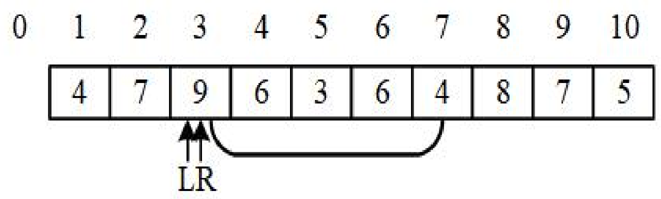
图(1) L=1, R=0



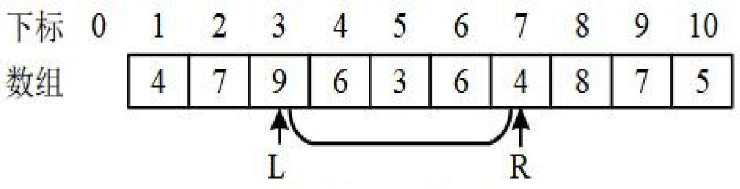
图(2) L=2, R=0



图(3) L=3, R=2



图(4) L=3, R=3



图(5) L=3, R=7

图(1)：L=1、R=0时，cnt[4]=0, cnt[7]=0, cnt[9]=0…答案ans=0。

图(2)：L=2、R=0时，cnt[4]=-1, cnt[7]=0。

图(3)：L=3、R=2时，cnt[4]=0, cnt[7]=0, cnt[9]=0…

图(4)：L=3、R=3时，cnt[4]=0, cnt[7]=0, cnt[9]=1。出现了一个等于1的cnt[9]，答案ans = 1。

图(5)：L=3、R=7时，cnt[4]=1, cnt[7]=0, cnt[9]=1, cnt[6]=2, cnt[3]=1,…其中cnt[4], cnt[9], cnt[6], cnt[3]都出现过等于1的情况，所以答案ans = 4。

莫队算法

（2）统计**多个区间**

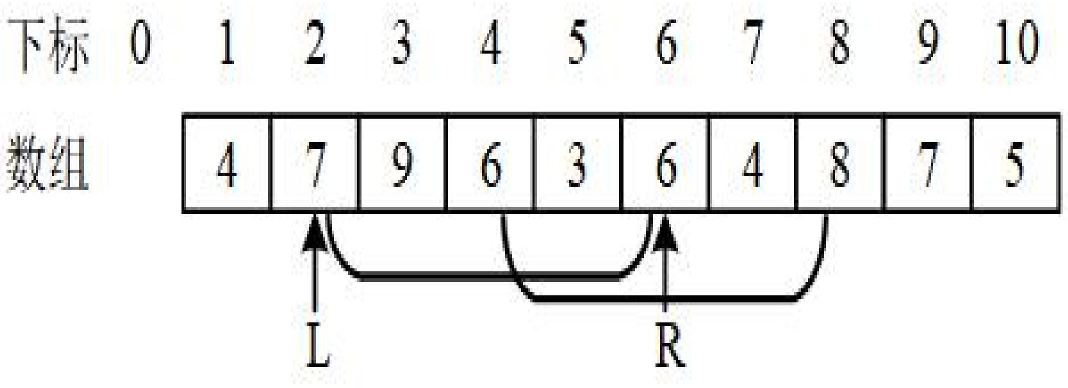
从上面查询一个区间的讨论可以知道，在L、R移动过程中，当它们停留在区间[L, R]时，就得到了这个区间的答案ans。那么对m个询问，只要不断移动L、R并与每个询问的区间匹配，就得到了m个区间询问的答案。

莫队算法

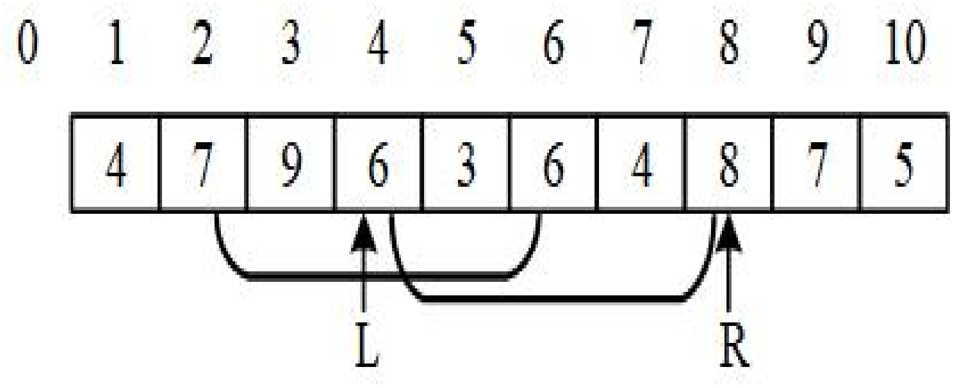
为了方便操作，可以把所有询问的区间**按左端点排序；如果左端点相同，再按右端点排序**。讨论以下情况：

- 简单情况，区间交错，区间[x1, y1]、[x2, y2]的关系是x1 ≤ x2，y1 ≤ y2。

例如下图中，查询两个区间[2, 6]、[4, 8]。



扫描第一个区间[2,6]

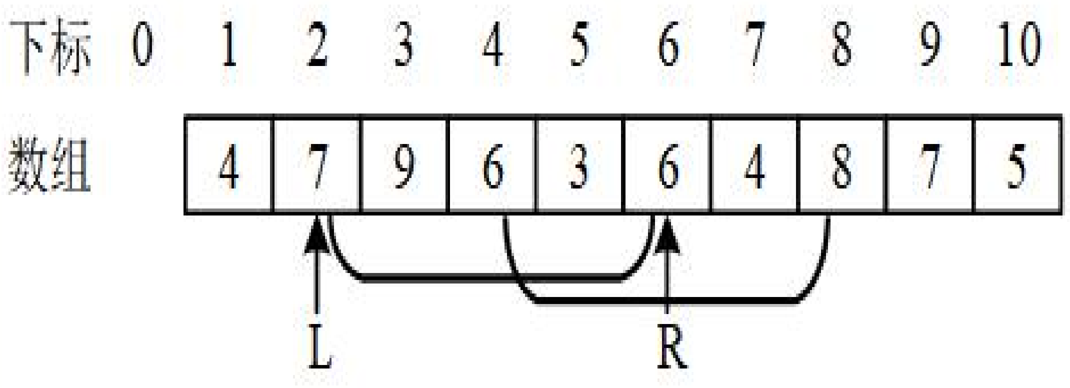


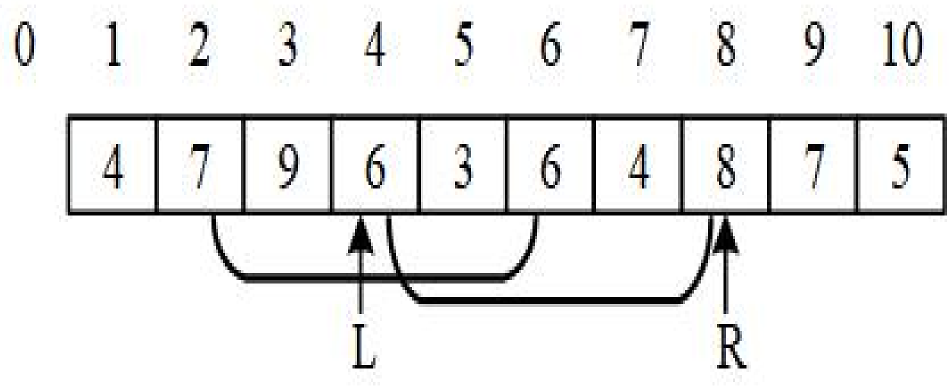
扫描第二个区间[4,8]

莫队算法

为了方便操作，可以把所有询问的区间**按左端点排序；如果左端点相同，再按右端点排序**。讨论以下情况：

- 简单情况，区间交错，区间[x1, y1]、[x2, y2]的关系是x1 ≤ x2，y1 ≤ y2。



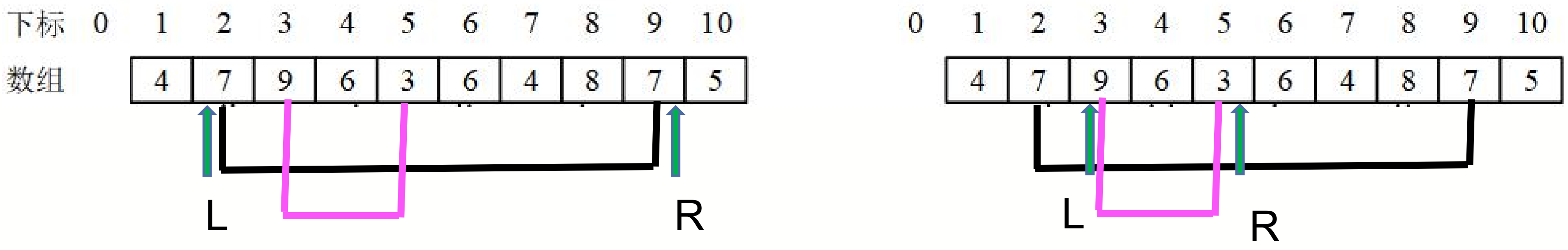


L、R停留在第1个区间上，得到了第1个区间的统计结果；L、R停留在第2个区间上，得到了第2个区间的结果。m次查询的m个区间，L、R指针只需要从左头到右（单向移动）扫描整个数组一次即可，总复杂度O(n)。

莫队算法

为了方便操作，可以把所有询问的区间**按左端点排序；如果左端点相同，再按右端点排序**。讨论以下情况：

- 复杂情况，既有**区间交错，又有区间包含**。区间[x1, y1]、[x2, y2]的包含关系是指x1 ≤ x2， y1 ≥ y2。例如下图中，区间[2, 9]包含了区间[3, 5]。此时L从头到尾单向扫描，而**R指针却需要来回往复扫描，每次扫描的复杂度是O(n)**。m次查询的总复杂度是O(mn)。



扫描第一个区间[2,9]　　　　扫描第二个区间[3,5]

- R往复移动的时候，R往左每扫一个数x，就把它出现的次数cnt[x]减去1。

莫队算法

区间查询问题，可以概况为这样一种离线的几何模型：

（1）m个询问对应m个区间，区间之间的转移，可以用L、R指针扫描，**能以O(1)的复杂度从区间[L,R]移动到[L±1, R±1]**。

（2）把一个区间[L, R]看成平面上的一个坐标点(x, y)，L对应x，R对应y，那么区间的转移等同于平面上坐标点的转移，计算量等于坐标点之间的曼哈顿距离。注意，所有的坐标点(x, y)都满足x ≤ y，即所有的点都分布在上半平面上。

莫队算法

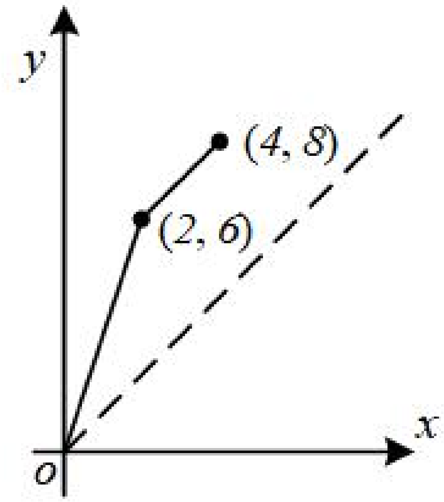
区间查询问题，可以概况为这样一种离线的几何模型：

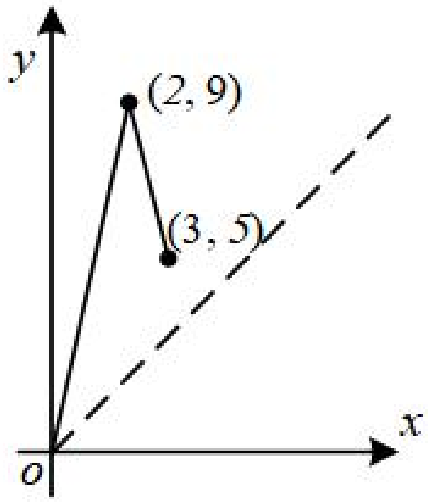
（3）完成m个询问，等于从原点出发，用直线连接这m个点，形成一条“Hamilton路径”(Hamilton 路径的定义是从 0 到 n-1 不重不漏地经过每个点恰好一次)，路径的长度就是计算量。若能找到一条最短的路径，计算量就最少。

Hamilton最短路径问题是NP难度的，没有多项式复杂度的解法。那么有没有一种较优的算法，能快速得到较好的结果呢？

莫队算法

暴力法是按照坐标点(x, y)的x值排序而生成的一条路径，它不是好算法。例如左图的简单情况，暴力法的顺序是好的；但是图(2)右图的复杂情况，暴力法的路径是(0, 0)--->(2, 9)--->(3, 5)，曼哈顿距离(2-0) + (9-0) + (3-2) + (9-5) = 16，不如另一条路径(0, 0)--->(3, 5)--->(2, 9)，曼哈顿距离 = 13。





莫队算法

下面介绍的莫队算法，提出了一种较好的排序方法。

莫队算法把排序做了简单的修改，就把暴力法的复杂度从O(mn)提高到$O(n\sqrt{n})$ 。

（1）暴力法的排序：把查询的区间按左端点排序，如果左端点相同，再按右端点排序。

（2）莫队算法的排序：把数组分块（分成 $\sqrt{n}$ 块），然后把查询的区间按左端点所在**块的序号**排序，如果左端点的块相同，再按**右端点**排序（注意**不是按右端点所在的块排序**，后面将说明原因）。

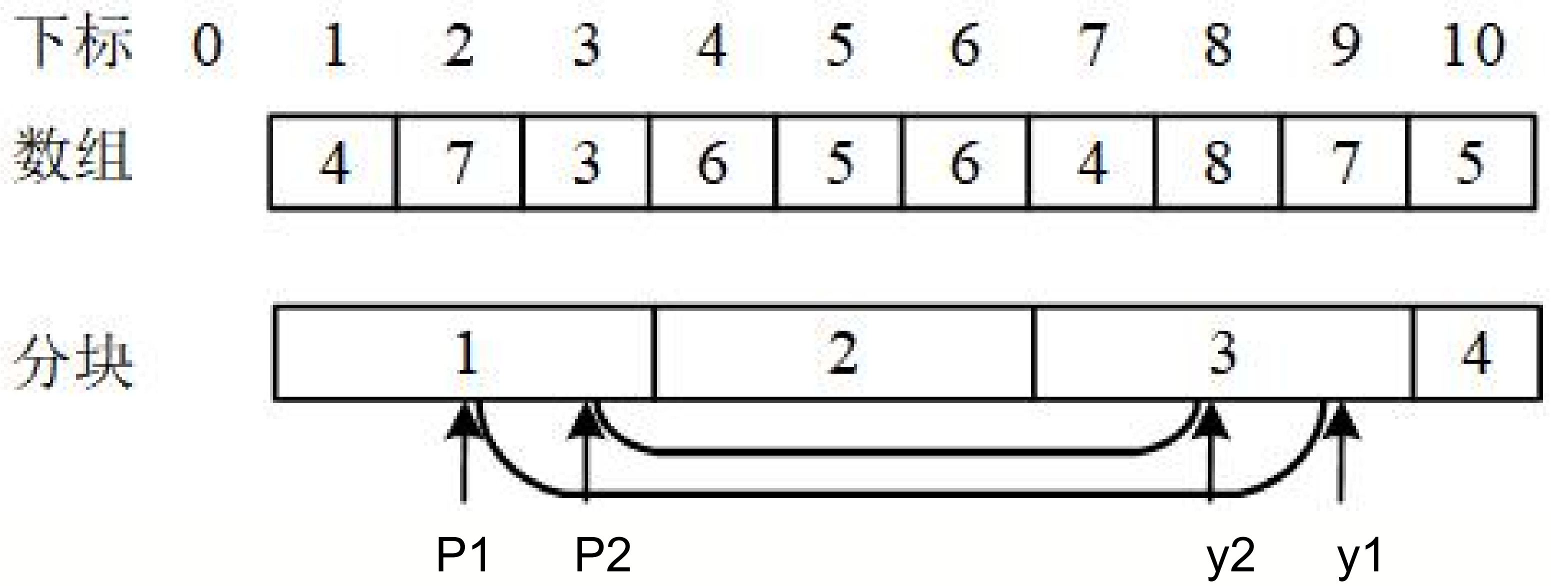
除了排序不一样，莫队算法和暴力法的其他步骤完全一样。

这个简单的修改是否真能提高效率？下面分析多种情况下莫队算法的复杂度。

莫队算法

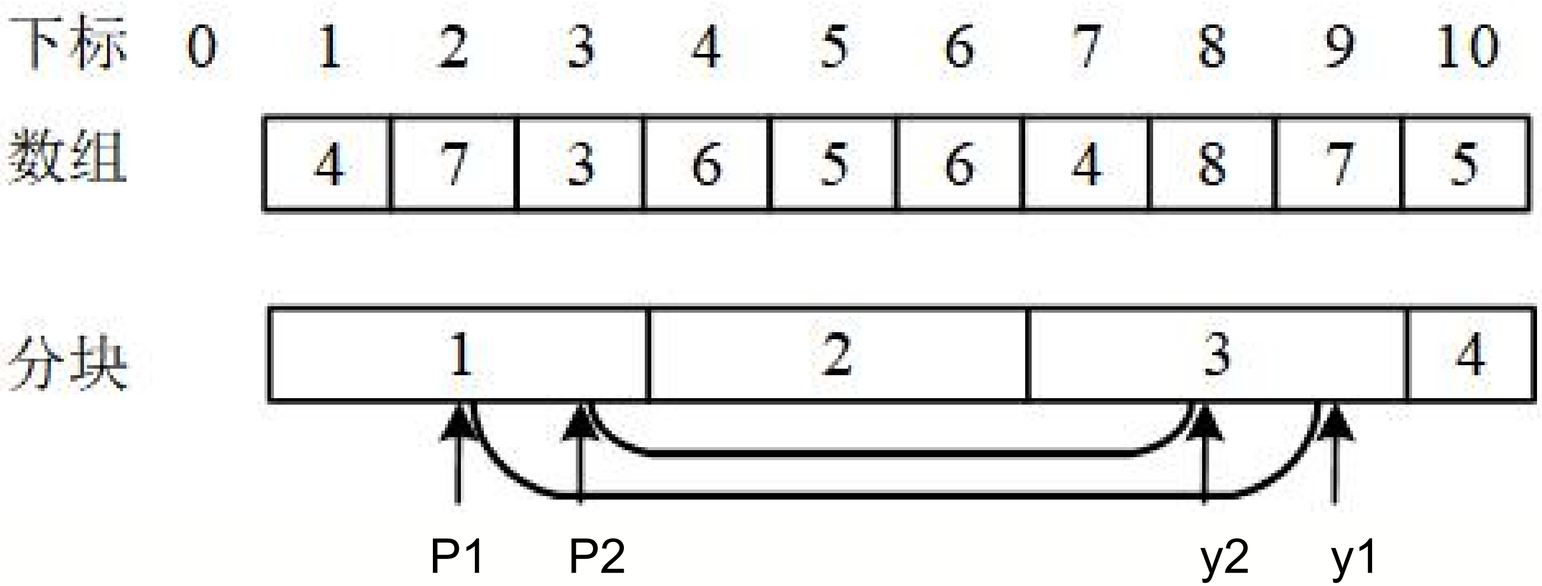
- 简单情况。区间交错，设区间[P1,y1]、[P2,y2]的关系是P1<P2，y1≤y2，其中**P1、P2是左端点所在的块**。L、R只需要从左到右扫描一次，m次查询的总复杂度是O(n)。

- 复杂情况。区间包含，设两个区间查询[P1,y1]、[P2,y2]的关系是P1=P2，y2≤y1。如下图所示。



莫队算法

- 复杂情况。区间包含，设两个区间查询[P1,y1]、[P2,y2]的关系是P1=P2，y2≤y1。如下图所示。



此时**小区间[P2,y2]排在大区间[P1,y1]的前面，与暴力法正好相反**。在区间内，**右指针R从左到右单向移动，不再往复移动**。而左指针L发生了回退移动，但是被限制在一个长为$\sqrt{n}$ 的块内，每次移动的复杂度是$O(\sqrt{n})$ 的。m次查询，每次查询左端点只需要移动$O(\sqrt{n})$ 次，右端点R共单向移动 $O(n)$ 次，总复杂度 $O(n)$ 。

莫队算法

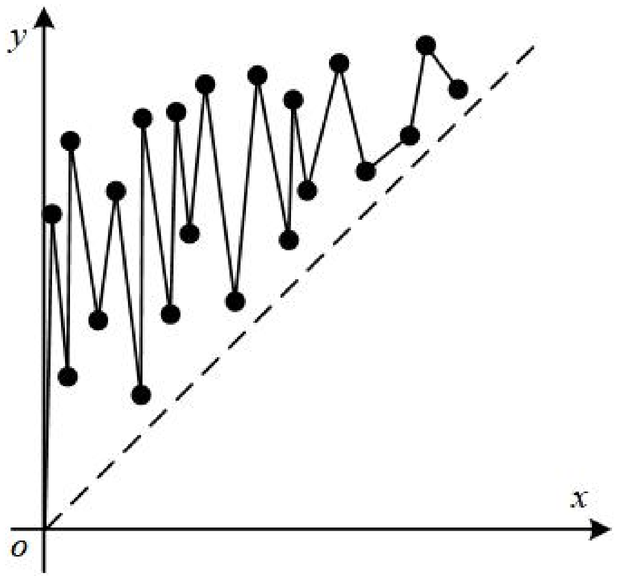
- 特殊情况：m个询问，端点都在不同的块上，此时莫队算法和暴力法是一样的。但此时m小于 $\sqrt{n}$，总复杂度$O(mn)=O(n\sqrt{n})$ 。

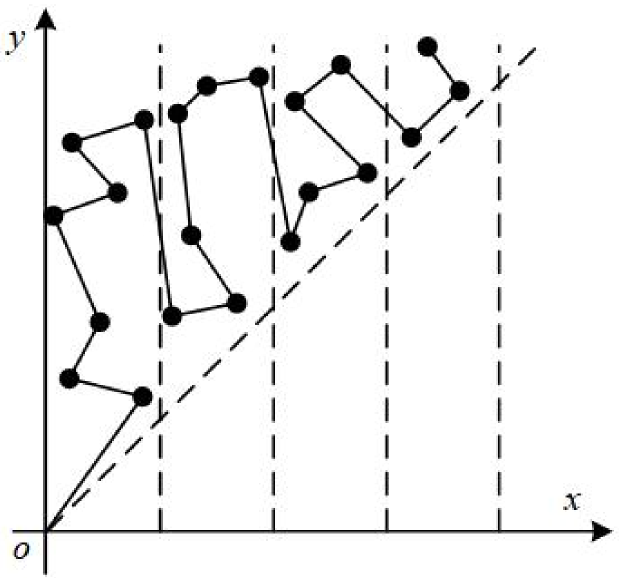
莫队算法

莫队算法的几何意义见下图，这张图透彻说明了莫队算法的原理。图中的每个黑点是一个查询。

左图是暴力法排序后的路径，所有的点按x坐标排序，在复杂情况下，路径沿y方向来回往复，震荡幅度可能非常大（**纵向震荡，幅度O(n)**），导致路径很长。

右图是莫队算法排序后的路径，它把x轴分成多个区（分块），每个区内的点按y坐标排序，在区内沿x方向来回往复，此时震荡幅度被限制在区内（**横向震荡，幅度O(√n)**），形成了一条比较短的路径，从而实现了较好的复杂度。





莫队算法

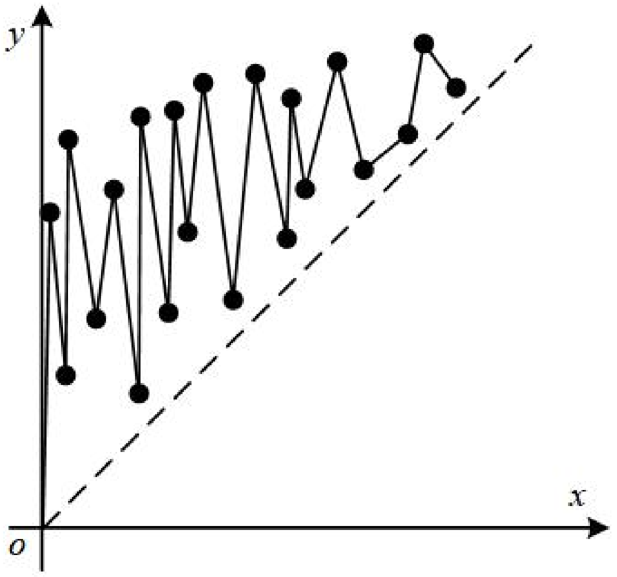
通过右图可以更清晰地计算莫队算法的复杂度：

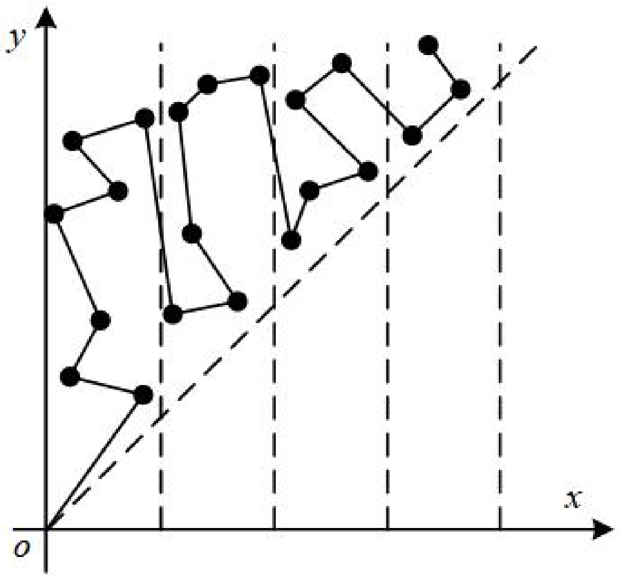
（1）x方向的复杂度。在一个区块内，沿着x方向一次移动最多$\sqrt{n}$，所有区块共有m次移动，总复杂度 $O(m\sqrt{n})$ 。

（2）y方向的复杂度。在每个区块内，沿着y方向单向移动，整个区块的y方向长度是n，有$\sqrt{n}$ 个区块，总复杂度 $O(n\sqrt{n})$ 。

两者相加，总复杂度 $O(m\sqrt{n}+n\sqrt{n})$ ，一般情况下题目会给出n = m。

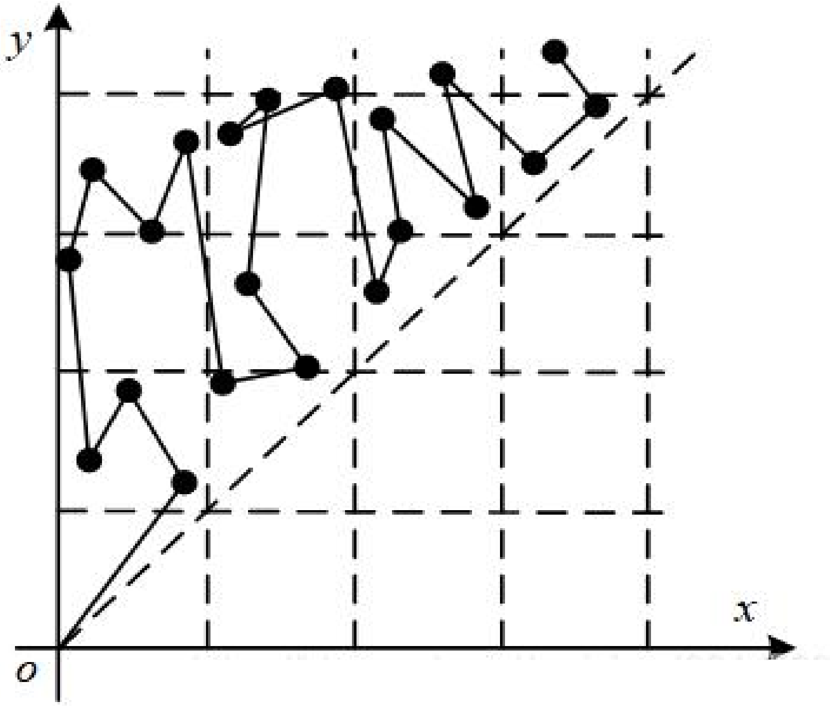
根据下图总结出莫队算法的核心思想：把暴力法的y方向的O(n) 幅度的震荡，改为x方向的受限于区间的$O(\sqrt{n})$ 幅度的震荡，从而减少了路径的长度，提高了效率。





莫队算法

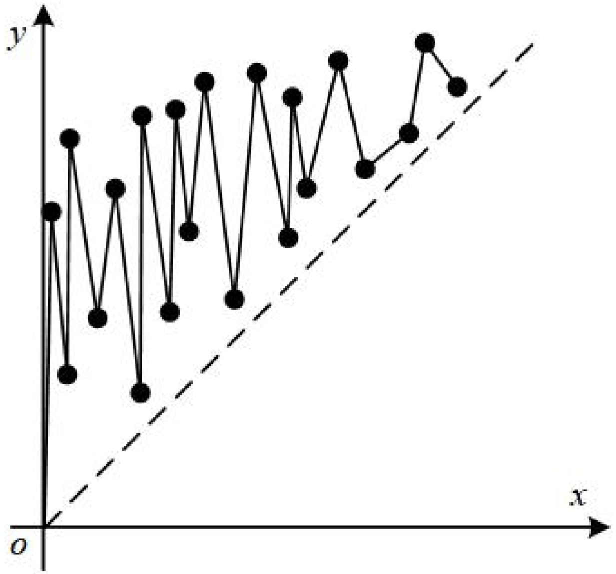
前面曾提到排序问题，对区间排序是先按左端点所在块排序，再按右端点排序，不是按右端点所在的块排序。原因解释如下：如果右端点也按块排序，几何图就需要画成一个方格图，方格中的点无法排序，实际的结果就是乱序。那么同一个方格内的点，在y方向上就不再是一直往上的复杂度为O(n)的单向移动，而是**忽上忽下的往复移动**，导致路径更长，复杂度变差。

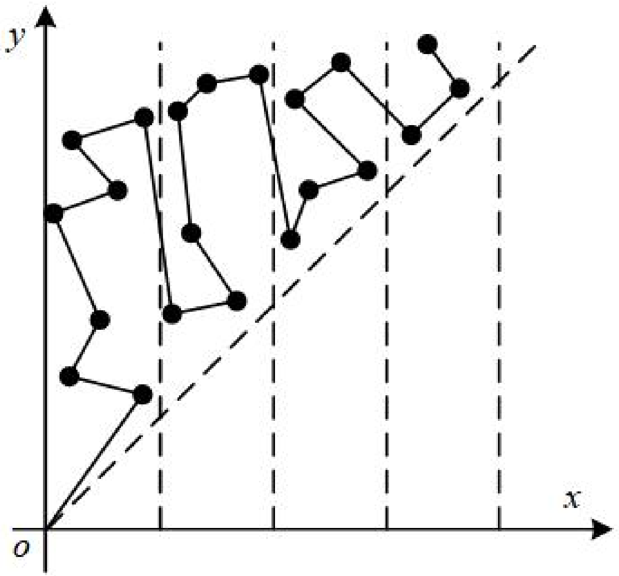


莫队算法

编码时，还可以对排序做一个小优化：**奇偶性排序**，让**奇数块和偶数块的排序相反**。例如左端点L都在奇数块，则对R从大到小排序；若L在偶数块，则对R从小到大排序（反过来也可以：奇数块从小到大，偶数块从大到小）。

这个小优化对照右图很容易理解，图中路径在两个区块之间移动时，是**从左边区块的最大y值点移动到右边区块的最小y值点，跨度很大**。用奇偶性排序后，奇数块的点从最大y值到最小y值点排序，偶数块从最小y值点到最大y值点排序；那么奇数块最后遍历的点是最小y值点，然后右移到偶数块的最小y值点，这样移动的距离是最小的。从偶数块右移到奇数块的情况类似。





关于莫队区间端点移动的顺序

莫队算法

莫队完整代码

那么怎么移动L、R并与每个询问的区间匹配?

L往右每扫一个数x，就把它出现的次数cnt[x]减去1；R往右每扫到一个数x，就把它出现的次数cnt[x]加上1。若cnt[x]=1说明x第1次出现，ans加1；若cnt[x]变为0，说明它从区间里消失了，ans减1。

一般采用前两步先扩大区间(l--,r++)，后两步再缩小区间（l++ , r--），这样写，前两步由于是扩大区间，因此区间一定不会变非法；执行完前两步后， $l \le l' \le r' \le r$ 一定成立，此时执行后两步只会把区间缩小到 $[l',r']$ ，不会造成区间不合法。

```
void add(int x) {
    cnt[a[x]]++;
    if (cnt[a[x]] == 1)
        ans++;
}

void del(int x) {
    cnt[a[x]]--;
    if (cnt[a[x]] == 0)
        ans--;
}
```

```
while(l>s[i].l)
    add(--l);
while(r<s[i].r)
    add(++r);
while(l<s[i].l)
    del(l++);
while(r>s[i].r)
    del(r--);
```

注意:add 是前++或前--,而del是后++或后--