基础算法

编程兔教育

主要内容

- 单调栈和单调队列
- 前缀和与差分
- STL
- 贪心
- 分治/二分
- 倍增

单调栈和单调队列

何为单调栈和单调队列

- 单调栈/单调队列，顾名思义，即保证内部元素单调（从大到下或者从小到大）的栈或者队列。我们只要在插入新元素的时候，将栈顶/队尾所有在插入新元素后不满足单调的元素依次弹出，再插入新元素即可。
- 单调队列和单调栈中的元素满足单调性。

- 可以这么理解：所有加入单调栈/单调队列的元素有一个『价值』和『时效性』。价值即为元素的值，时效性则为元素加入的时间。元素加入的时间越久远就越有可能失效。我们要查询在当前还没有失效的所有元素里面谁的价值最高。自然，如果一个元素加入的时间早（更有可能失效），其价值还赶不上后加入的元素，它便没有存在的必要。
- 所以，单调栈/单调队列的本质是，其内部的元素，由由栈底到栈顶/队首到队尾，价值越来越小，但时效性越来越强（越来越不容易失效）。因为每个元素只可能入栈/队列一次，出栈/队列一次，所以整体的时间复杂度是O(n)的（n为元素个数）。
- 通常用在DP的优化中。

单调栈

- 栈内元素单调按照递增（递减）顺序排列的栈。
- 分为单调递增栈和单调递减栈，通过单调栈可以访问到下一个比它大（小）的元素，也就是从数组中找到左右两边比它大的数或者比它小的数，而且时间复杂度是O(n)。
- 单调递增栈：从栈底到栈顶是递增的，栈中保留的都是比当前入栈元素小的值。
- 单调递减栈：从栈底到栈顶是递减的，栈中保留的都是比当前入栈元素大的值。
- 其实，递增递减傻傻分不清，有的定义是从从栈顶到栈底单调，有的定义是从从栈底到栈顶单调。
- 无需拘泥于这些细节，只要理解了相关的概念和应用即可，因此本课件全部使用从栈底到栈顶单调的概念来描述递增栈和递减栈的顺序。


需要入栈的元素
6
10
3
7
4
12
6
栈为空，6入栈
10
与栈顶元素6对比，10>6，6出栈，10入栈
10
3
与栈顶元素10对比，3<10，3入栈
10
7
与栈顶元素7对比，7>3，3出栈，7入栈
10
7
4
与栈顶元素6对比，4<7，4入栈
12
与栈顶元素6对比，12>4，4出栈，继续
与栈顶元素7对比，12>7，7出栈，继续
与栈顶元素10对比，12>10，10出栈，此
时栈为空，12入栈
单调递减栈
6
栈为空，6入栈
6
10
10>6，10入栈
3
3<10，10出栈，3<6，6出栈，3入栈
3
7
7>3，7入栈
3
4
4<7，7出栈，4>3，4入栈
3
4
12
12>4，12入栈
单调递增栈

具体操作

- 为了维护栈的单调性，在进栈过程中需要进行判断，具体进栈过程如下：假设当前进栈元素为e，
- 对于单调递减栈，从栈顶开始遍历元素，把小于e或者等于e的元素弹出栈，直至遇见一个大于e的元素或者栈为空为止，然后再把e压入栈中，这样就能满足从栈底到栈顶的元素是递减的。
- 对于单调递增栈，则每次弹出的是大于e或者等于e的元素，直至遇见一个小于e的元素或者栈为空为止。

应用

- 单调栈主要解决下面几种问题：
- 比当前元素更大的下一个元素
- 比当前元素更大的前一个元素
- 比当前元素更小的下一个元素
- 比当前元素更小的前一个元素

例1. P5788 单调栈

【题目描述】
给出项数为n的整数数列$a_{1...n}$，定义函数f(i)代表数列中第i个元素之后第一个大于a_i的元素的下标，即$f(i)=\min_{i<j\le n,a_j>a_i}\{j\}$。若不存在，则f(i)=0。试求出f(1...n)。
【输入格式】
第一行正整数n，
第二行n个正整数$a_{1...n}$。
【输出格式】
一行 n个整数f(1...n) 的值。
【输入样例1】
5
1 4 2 3 5
【输出样例1】
2 5 4 5 0
【数据规模与约定】
对于30%的数据，$n\le100$；对于60%的数据，$n\le5\times10^3$；对于100% 的数据，$1\le n\le3\times10^6$，$1\le a\le10^9$。

算法分析

- 单调栈模板题
- 从后往前扫，对于每个元素，弹出栈顶比它小的元素，此时栈顶就是答案，加入这个元素。

例2. P1901 发射站

【题目描述】
某地有N个能量发射站排成一行，每个发射站i都有不相同的高度H_i，并能向两边（两端的发射站只能向一边）同时发射能量值为V_i的能量，发出的能量只被两边最近的且比它高的发射站接收。显然，每个发射站发来的能量有可能被0或1或2个其他发射站所接受。请计算出接收最多能量的发射站接收的能量是多少。
【输入格式】
第1行一个整数N。
第2到N+1行，第i+1行有两个整数H_i和V_i，表示第i个人发射站的高度和发射的能量值。
【输出格式】
输出仅一行，表示接收最多能量的发射站接收到的能量值。答案不超过32位带符号整数的表示范围。
【输入样例1】
3
4 2
3 5
6 10
【输出样例1】
7

算法分析

- 问题可以转化成求一个元素右边第一个比他大的和左边第一个比它大的。而这个恰好一个栈就能实现。
- 如果即将入栈的元素大于栈顶元素，则栈顶元素的能量值累加到即将入栈的元素上去，接着弹栈，重复直到不满足条件为止。
- 入栈时栈顶元素一定大于即将入栈的元素，这时候只需要把能量累加到栈顶元素上去就行。
- 最后找出最大值。

例3.

【题目描述】

有一列n个数字a[1]...a[n]，对所有1<=L<=R<=n，求max(a[L],a[L+1],...,a[R])并求和，n<=1e6。

【输入样例】

6

5 9 3 2 4 7

【输出样例】

143

算法分析

- 这种题的一个常见套路就是考虑一个数字会在哪些区间中被算到。
- 先考虑所有数字互不相同的情况。
- 考虑数字a[p]会成为哪些区间的max，用上文方法求出左面和右面第一个比它大的位置l[p]和r[p]（认为a[0]和a[n+1]是正无穷即可避免一些特判），那么当l[p]<L<=p<=R<r[p]的时候，max(a[L],...,a[R])=a[p]。因此答案就是a[p]*(p-l[p])*(r[p]-p)的和。

- 注意序列中有相同数字的时候，要钦定（比如）左边的比右边的大。

例4.

【题目描述】

考虑有一列n个数字，求1<=L<=R<=n使得(R-L+1)*min(a[L],a[L+1],…,a[R])最大。n<=1000000

【输入样例】

5

3 6 2 9 7

【输出样例】

14

算法分析

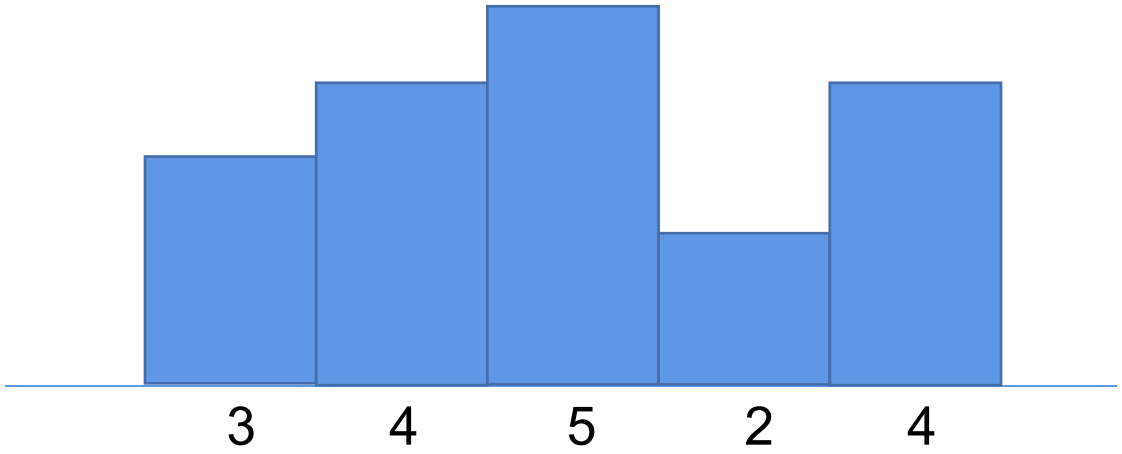
- 还是和刚才一样对每个数字维护其在哪个极大区间成为最小值，然后L,R取为这个区间算一算即可。

例5. poj3494 求最大全1子矩阵

【题目描述】
有一个n*m的矩阵，每个位置是0或者1。求最大的全1子矩阵。1<=n, m<=2000。
【输入样例】
2 2
0 0
0 0
4 4
0 0 0 0
0 1 1 0
0 1 1 0
0 0 0 0
【输出样例】
0
4

算法分析

- 对每个位置维护其向上极长的1的段的长度，然后每行转化为刚刚那个问题即可。
- 一行一行来，枚举以这一行为底的最长1的长度，就变成基本的求直方图的最大面积的情况了。
- 例如，如何求一个数组{3 4 5 2 4}围成的最大面积。
- 对任意一个位置i，我们需要找到其左右边第一个小于a[i]的数的位置p1,p2，则位置i对应的面积是(p2-p1-1)*a[i]。



例6. P1198 最大数

【题目描述】
现在请求你维护一个数列，要求提供以下两种操作：
1、 查询操作。
语法：Q L
功能：查询当前数列中末尾L个数中的最大的数，并输出这个数的值。
限制：L不超过当前数列的长度。(L>0)
2、 插入操作。
语法：A n
功能：将n加上t，其中t是最近一次查询操作的答案（如果还未执行过查询操作，则t=0），并将所得结果对一个固定的常数D取模，将所得答案插入到数列的末尾。
限制：n是整数（可能为负数）并且在长整范围内。
注意：初始时数列是空的，没有一个数。
【输入格式】
第一行两个整数，M和D，其中M表示操作的个数，D如上文中所述。
接下来的M行，每行一个字符串，描述一个具体的操作。语法如上文所述。
【输出格式】
对于每一个查询操作，你应该按照顺序依次输出结果，每个结果占一行。

【输入样例1】
5 100
A 96
Q 1
A 97
Q 1
Q 2
【输出样例1】
96
93
96

算法分析

- 看数据范围，插入相当于入栈，问题在于暴力查询O(n^2)超时。
- 考虑用单调栈去维护，将下标入栈，保证栈内代表的元素从栈底到栈顶逐渐减小。
- 因为只需要后面的最大值，所以去除前面小的毫无影响。
- 入栈时判断与栈顶元素大小关系，如果大于，则栈顶出栈，直到满足条件。
- 这样求最大值只需要找到栈内第一个大于等于n-L+1的下标，其对应序列值即为所求。
- 因为栈所存下标肯定单增，查找可采用二分查找，这样总的时间复杂度O(nlogn)。

算法分析

- 然后这道题也可以用并查集思想，栈顶元素出栈时可将该元素与即将入栈的元素合并。
- 这样要求后L项最大值则可以直接找第n-L+1所在集合的根节点。
- 时间复杂度为O(k*n),k为并查集相关的常数。

单调队列

- 单调队列只能解决一个叫滑动窗口的问题。
- 这个问题是这样的：有一列数{a_n}和m个区间[L(i),R(i)]，满足L(i)<=L(i+1)，R(i)<=R(i+1)，对每个区间求区间最大值。
- 由于这看上去像是有个大小不固定的窗口在移动，然后你透过窗口观察能看到的数字的最值，所以称为滑动窗口问题。
- 这个问题至少能用随便什么数据结构（线段树、RMQ算法）在O(nlogn)时间内完成，但是使用单调队列可以做到O(n)。

- 我们维护从左到右哪些数字有可能成为答案。
- 那么每次右边新增一个数字a[x]，若其比目前最右边的可能成为答案的数字a[y]还大，那么意味着之后无论怎么询问，a[y]都不可能是答案，删掉即可。然后接着看，直到a[x]比a[y]小为止。
- 发现这样维护出来的东西，从左到右有可能成为答案的数字是单调变小的。并且目前的区间的最大值就是最左面有可能成为答案的。
- 删除一个数字就看删掉的是不是目前最左面的那个数字即可。
- 单调队列能做的事情就这一个，所谓的单调队列优化dp等也不过是列完式子然后可以转化为一个左右端点递增的区间询问最值的问题罢了。

例7. P1886 滑动窗口

【题目描述】
现在有一堆数字共n个数字（n<=10^6），以及一个大小为k的窗口。现在这个窗口从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。
【输入格式】
输入文件名为window.in。
输入文件一共有两行，第一行为n,k。
第二行为n个数，在[2^-31，2^31)范围内。
【输出格式】
输出文件名为window.out。
输出文件名共两行，第一行为每次窗口滑动的最小值，
第二行为每次窗口滑动的最大值。
【输入样例】
8 3
1 3 -1 -3 5 3 6 7
【输出样例】
-1 -3 -3 -3 3 3
3 3 5 5 6 7

算法分析

- 朴素的算法：对于每个大小为k的区间，都要计算最大值和最小值，时间复杂度O(n*k)。
- 用单调队列分别求最大值和最小值。

位置	1	2	3	4	5	6	7	8
数值	1	3	-1	-3	5	3	6	7
min			-1	-3	-3	-3	3	3

队列： 1 3 -1 -3 5 3 6 7

- 以求最小值为例观察队列中元素离开队列的情况：

1. 元素Vi从队尾离开队列：

   i < j 且 Vi >= Vj，Vi没有参与求min的必要

2. 元素Vi从队首离开队列：

   Vi超出了当前窗口的范围。

例8. 一本通1598 最大连续和

【题目描述】
给你一个长度为n的整数序列{A1,A2,… ,An}，要求从中找出一段连续的长度不超过m的子序列，使得这个序列的和最大。
【输入】
第一行为两个整数n,m；
第二行为n个用空格分开的整数序列，每个数的绝对值都小于1000。
【输出】
仅一个整数，表示连续长度不超过 m 的最大子序列和。
【输入样例】
6 4
1 -3 5 1 -2 3
【输出样例】
7

算法分析

- 把前缀和当做值跑单调队列。
- 需要维护一个i-k ~ i+1的区间（长度为m），找到一个最小的前缀和（这样就可以保证这个区间内的和最大），用pre[i+1]-这个前缀和就是这个区间的最大值了，用ans来记录一下最大值即可。

例9. P2034 修剪草坪

【题目描述】
FJ让他的奶牛来修剪草坪。FJ有N只排成一排的奶牛，编号为1...N。每只奶牛的效率是不同的，奶牛i的效率为Ei。如果FJ安排超过K（1<=K<=N）只连续的奶牛，这些奶牛就会罢工。计算FJ可以得到的最大效率，并且该方案中没有连续的超过K只奶牛。
【输入格式】
第一行：空格隔开的两个整数N和K
第二到N+1行：第i+1行有一个整数Ei
【输出格式】
一个值，表示FJ可以得到的最大的效率值。

【样例输入】

```
5 2
1
2
3
4
5
```

【样例输出】

```
12
```

算法分析

- 设f[i]表示i不选且之前的选取都合法情况下答案损失的最小值。
- 则f[i]=f[j]+e[i]（i-j<=k+1），然后用单调队列来维护。
- 总效率-min(f[i])。

前缀和与差分

前缀和

- 对于一维数组a，前缀和S[i]表示的就是a[1]+a[2]+…+a[i]。

下标	1	2	3	4	5	6	7
a	2	3	1	4	5	2	6
S	2	5	6	10	15	17	23

前缀和的应用

- 前缀和有什么用?
- 如果想求a[i]~a[j]的和，普通做法是写一个循环把它们加起来，比较耗时间。
- 如果我们提前处理好了它的前缀和，我们只需要用S[j]-S[i-1]就能得到答案。

- 例如：有一列数字$\{a_n\}$，多次询问一个区间[L,R]的和。n,m<=1000000。
- 做法很简单，令s[p]=s[p-1]+a[p]=a[1]+a[2]+…+a[p]，那么：
- a[L]+a[L+1]+…+a[R]=s[R]-s[L-1]

例10.

- 给你一个序列 a，每次给出 l,r，需要计算al+2al+1+3al+2+…+(r-l+1)ar
- 这东西怎么算?

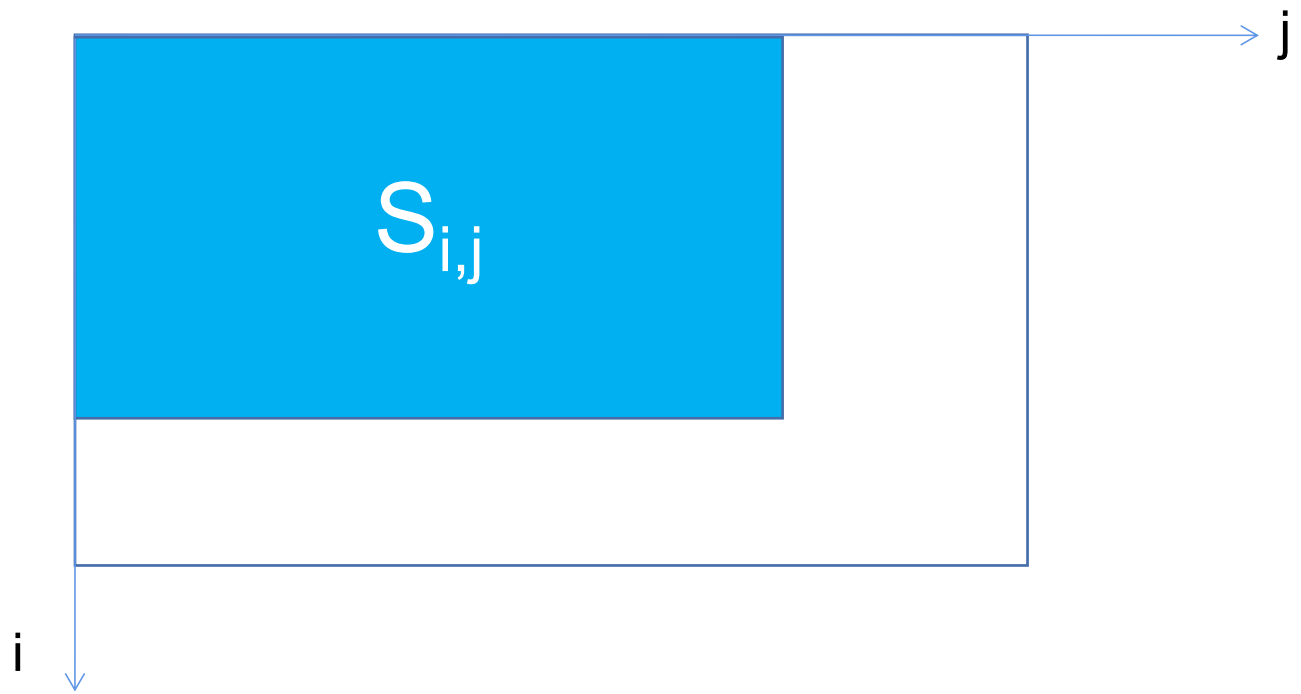
算法分析

- 可以发现
- al+2al+1+3al+2+…+(r-l+1)ar
- =(lal+(l+1)al+1+…+rar) – (l-1)(al+al+1+al+2+…+ar)
- 记 S 表示 a 的前缀和，S1 表示 i*ai 的前缀和就可以快速计算，答案就是
- (S1r-S1l-1)-(l-1)(Sr-Sl-1)

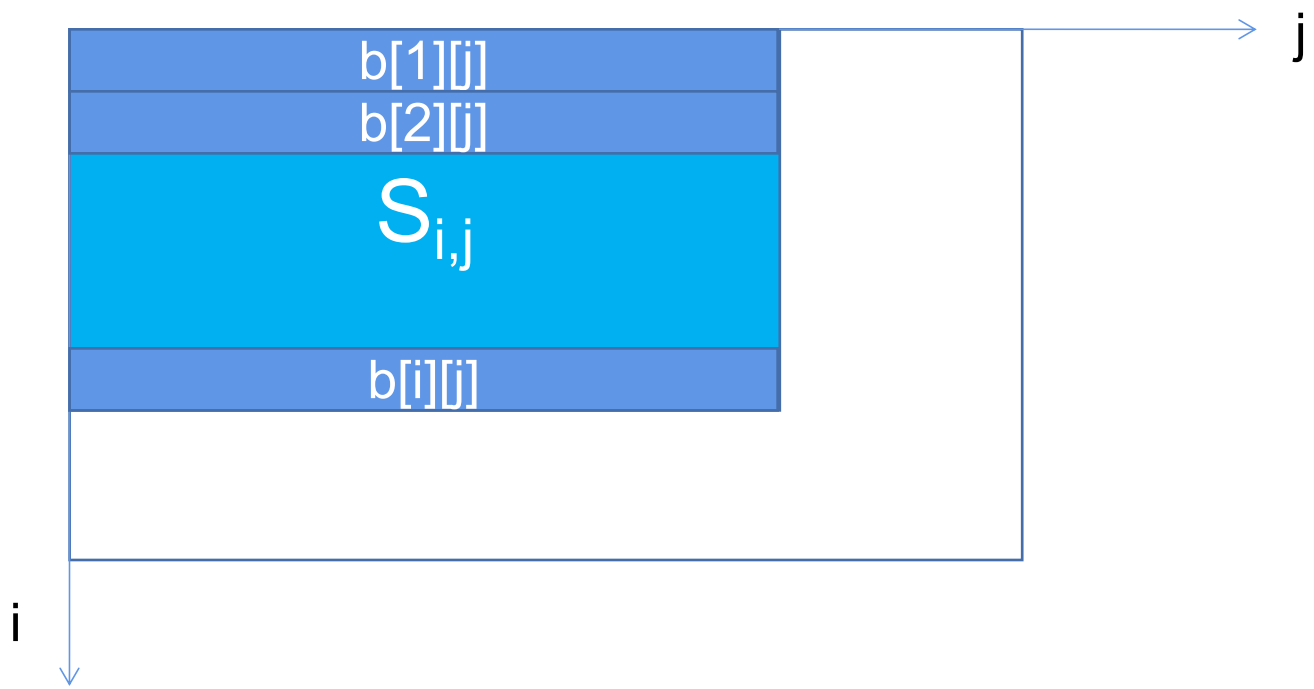
- 从刚才那个例子我们发现，可能有一些看上去不像是前缀和的东西可以通过拆成若干个可以计算前缀和的东西来算。
- 于是我们得到一个技巧：
- 把一个式子拆成若干个可以前缀和的东西，就能快速计算。

二维前缀和

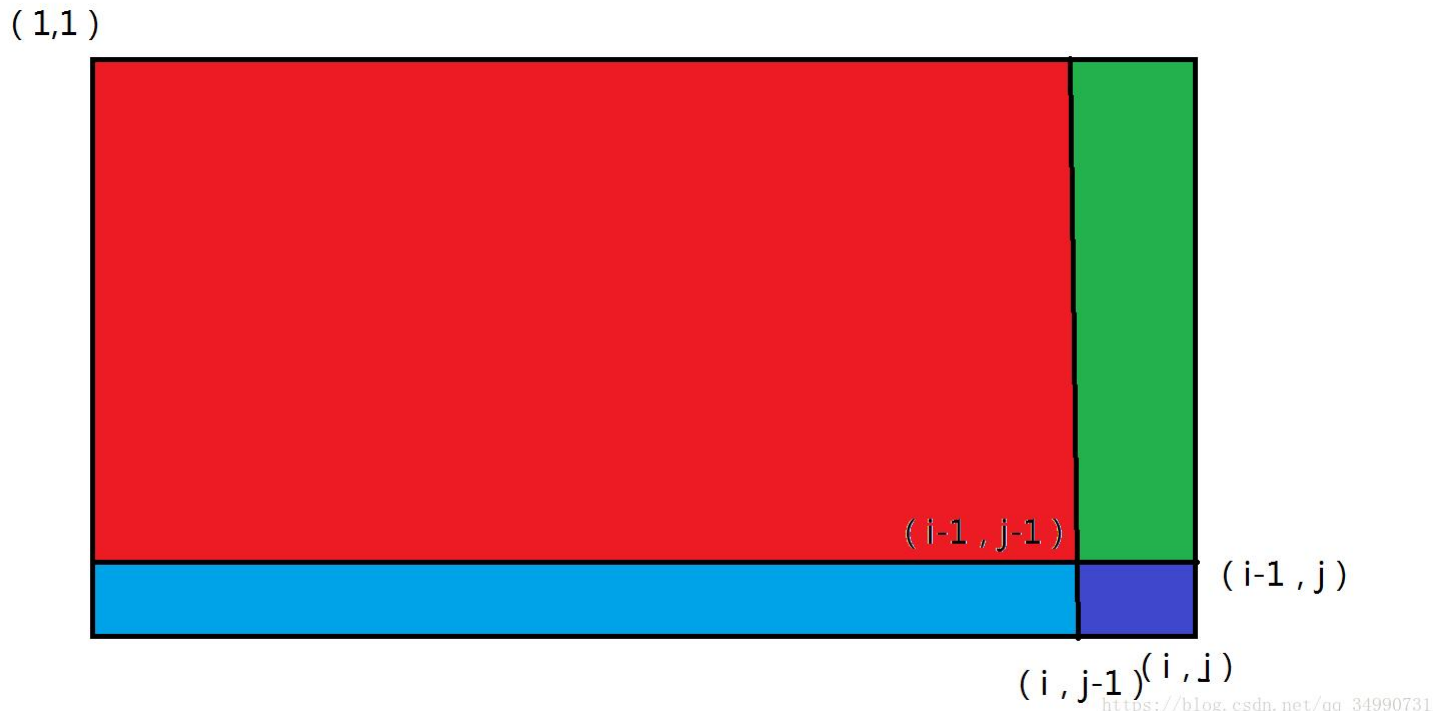
- 对于二维数组a，前缀和S[i][j]表示的是所有a[i’][j’]（1<=i’<=i，1<=j’<=j）的和。
- 考虑画到平面上，某个位置的前缀和就是它左上方所有数的和。



- 求这个二维前缀和的话，可以先对其中一维求前缀和，再对另一维求。
- 即令$b_{i,j}=a_{i,1}+a_{i,2}+\ldots+a_{i,j}$
- $S_{i,j}=b_{1,j}+b_{2,j}+\ldots+b_{i,j}$

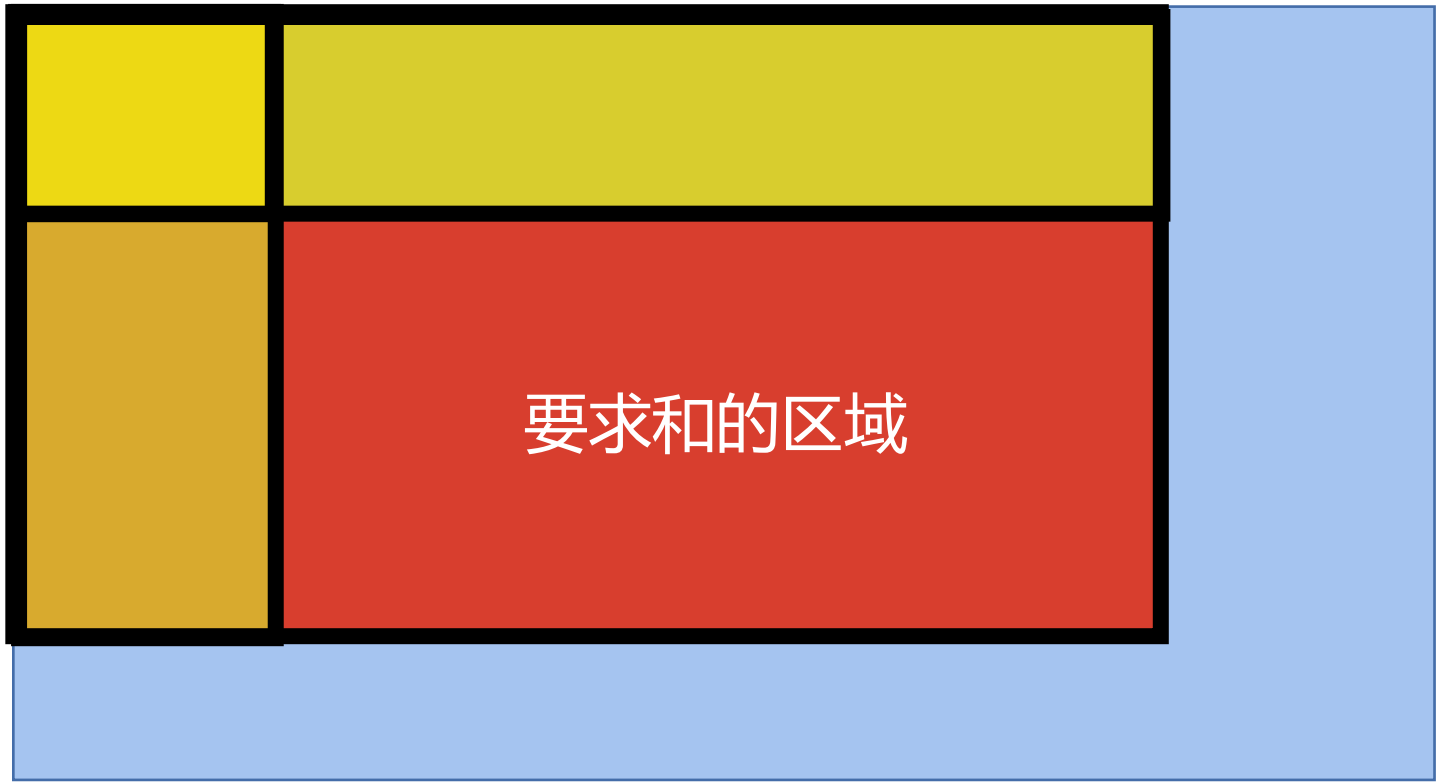


- 求二维前缀和也可以递推
- S[i][j]=S[i-1,j]+S[i,j-1]-S[i-1,j-1]+a[i,j]



二维前缀和的应用

- 假如我们处理出了前缀和S[i][j]，给定x1,y1,x2,y2,我们想求一下所有A[i’][j’](x1<=i’<=x2,y1<=j’<=y2)的和。
- ans=S[x2][y2]-S[x2][y1-1]-S[x1-1][y2]+S[x1-1][y1-1]



S[X2][Y2]

S[X1-1][Y2]

S[X2][Y1-1]

S[X1-1][Y1-1]

差分

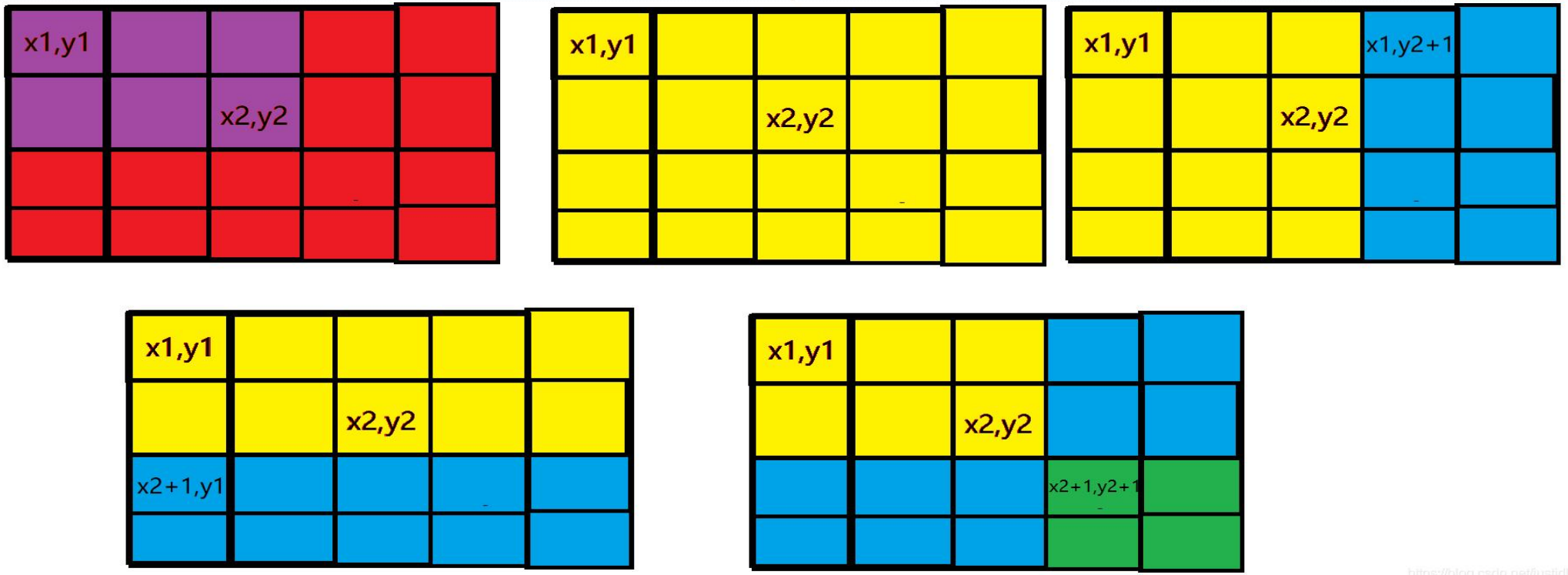
- 用差分实现区间操作
- 例如：给定一个长度为n的序列a，要求多次做区间加操作，即对a[L]~a[R]的每个数都加上v。问最后序列a是什么样的。
- n和操作数都是10^6级别。
- 考虑前缀和的逆变换，a[p]=S[p]-S[p-1]，S是a的前缀和。
- 差分的思想是前缀和思想的逆运算，通过构造一个新的数组，使原来的数组的每一个元素是新数组的前缀和。
- b[1]=a[1]; b[2]=a[2]-a[1]; b[3]=a[3]-a[2]; .....

下标	1	2	3	4	5	6	7
a	2	3	1	4	5	2	6
b	2	1	-2	3	1	-3	4
a’	2	7	5	8	9	2	6
b’	2	5	-2	3	1	-7	4

- a’为修改后的数组，b’为修改后数组的差分数组。
- 例如：区间[2,5]的元素都加上4
- 每次a的一个区间[L,R]+=v，等价于b[L]+=v,b[R+1]-=v。
- 最后再做一遍前缀和还原回来即可。

二维差分

- 二维差分： b[x][y]=a[x][y]+a[x-1][y-1]-a[x-1][y]-a[x][y-1]
- 修改矩形[x1,y1,x2,y2]等价于
- b[x1][y1]+=v, b[x2+1][y2+1]+=v, b[x1][y2+1]-=v, b[x2+1][y1]-=v。



差分的用途

- 很多东西直接统计起来比较复杂
- 或者得不到好的复杂度
- 然而如果具有可减性
- 即可以通过差分来解决问题或者降低复杂度

例11. P2879 Tallest Cow S

【题目描述】

有头牛从1到n线性排列，每头牛的高度为h[i]，现在告诉你这里面的牛的最大高度为maxH，而且有R组关系，每组关系输入两个数字，假设为a和b表示第a头牛能看到第b头牛，能看到的条件是a, b之间的其它牛的高度都严格小于min(h[a], h[b])，而h[b] >= h[a]。最后求所有牛的可能最高身高并输出。

【输入格式】

第1行：四个以空格分隔的整数：n，i，h和R（n和R意义见题面; i 和 h 表示第 i 头牛的高度为 h ，他是最高的奶牛）

接下来R行：两个不同的整数a和b（1 ≤ a，b ≤ n）

输出格式：

一共n行，表示每头奶牛的最大可能高度。

【输入 #1】

```
9 3 5 5
1 3
5 3
4 3
3 7
9 8
```

【输出 #1】

```
5
5
3
4
4
5
5
5
```

算法分析

- 先将身高全部假设为最高身高。
- 把给的信息(a,b)去重之后，为了使[a+1,b-1]内所有数都小于h[a],h[b]只需要区间减1即可。
- 维护一个差分数组d，d[a+1]--,d[b]++，然后扫一遍求出前缀和就可以得到每个位置的值。

STL

关于c++中的STL

- 在OI里面最常用的几个容器：
- stack，queue，deque，priority_queue，vector，set，map。

stack, queue, deque

- 分别是栈、队列、双端队列。
- 双端队列的入队和出队操作在两端都可进行。
- 据说双端队列常数不小，建议少用。

priority_queue

- 优先队列，也就是堆。

- 默认是大根堆。

```
priority_queue<int> pq;//默认大根堆
pq.push(x);
pq.top();
pq.pop();
pq.empty();
```

若想小根堆：
若只是int类型，可以每次直接pq.push(-x)，然后-pq.top()取出。

对于自定义类型，必须提供重载过的小于运算符（或者比较函数）。

```
struct node{
    int x;
    node(int _x=0) { x=_x; }
    bool operator<(const node &n)const
    {
        return x<n.x;//这样做会使堆是个大根堆。若想小根堆需要return x>n.x;
    }
}
priority_queue<node> pq_node;
```

- 在c++11中还有个函数是pq.swap(pq2)。

- 在c++11以前进行swap(pq,pq2)复杂度不是O(1)的！慎用！

vector

- 几乎是stl里面用的最多的容器了。

- 相当于是个动态数组，每次可以往末端插入一个元素，下标从0开始。

```cpp
vector<int> v;

v.push_back(x);
v.pop_back();
v.resize(size);
v.clear();
v.size();
v.begin(),v.end();//sort(v.begin(),v.end()),lower_bound(v.begin(),v.end(),x),etc;
v.erase(position),v.erase(first,last);
v[x]=y;
v.swap(another_vector)
vector<int>().swap(v);
```

关于v.size()

- 这个一个unsigned int类型。也就是说对空的vector的size()-1会得到一个INF。因此写代码的时候应带尽量避免这种写法。（或者强制类型转化成int）

关于v.resize()

- 其复杂度是O(max(1, resize()中的参数-原来的size()))的。

v.clear()和vector<>().swap(v)的区别。

- 前者是假装清空了，实际内存没有被回收。
- 后者是真的回收了，不过需要和v.size()的大小成正比的时间。

set/map

- 分别是集合/映射。
- 内部使用红黑树实现。
- 同样的当set<>和map<,>中
- 的第一个参数是自定义类型的时候
- 需要重新定于小于号。
- 复杂度基本上是O(log(当前大小))。

```
set<int> s;
s.insert(x);
s.count();
s.erase(x);
s.lower_bound(x);
s.upper_bound(x);
s.swap(s2);

map<int,int> m;
m.insert(make_pair(x,y));
m.count(x);
m[x]=y;
m.swap(m2);
```

关于m[x]

- 哪怕你什么也不干只写一个m[x];也会新建一个点。
- 因此当你想知道map中是否存在这个映射的时候最好使用m.count(x)。
- 很多时候可以有效卡常。

还有multiset和multimap

- 是可重集合和可重映射。
- 有两个注意的：第一个是count函数复杂度变成了O(log+答案)的，也就是如果有很多相同元素，那么count函数代价很大。
- 第二个是删除x的话，使用s.erase(x)会把所有权值为x的删除。
- 如果只想删掉一个需要s.erase(s.find(x))。

迭代器

- 只介绍set/map的迭代器。

```
set<int>::iterator it;
for(it=s.begin();it!=s.end();it++)//从小到大遍历集合
//一般情况下一边删除一边遍历会有问题，请不要这么干。
s.erase(it)//删除
s.insert(it)//插入
it=s.lower_bound(x);
*it就是x在s中的后继（即用*来获取权值）

map<int,int>::iterator it;
基本上和set一样，不过*it是一个pair<int,int>类型。
```

bitset

- 高精度压位二进制。
- 所有时间复杂度是线性的操作，常数都是1/32大概。
- 下标从0开始。

```
bitset<3> b;//这里大小必须是常量。
b.set();//b=111
b.reset();//b=000
b.set(1);//b=010
b.reset(1);//b=000
cout<<b<<endl;//没错可以直接cout输出
b[x]=y;//可以像数组一样直接访问x的第y位。
还支持& | ^ &= |= ^= ~ >> << >>= <<=这几种位运算符
b.any(),b.count(),b.none()获取是否b中有1，1的个数，是否全0。
其中set/reset返回其本身。
比如你可以b.set(1).reset(2)之类的，也可以cout<<b.set(0)<<endl;
还可以什么b2=b.set(1)之类的。
```

algorithm

- algorithm 库封装了许多函数。
- copy(A + l, A + r, B) 把 A 数组里 [l, r) 开区间拷贝到 B 数组开头。A+l, A+r 可以换成 vector 的 v.begin(), v.end(), 或者 v.begin()+i。下同
- fill(A + l, A + r, v) 把 A 数组里 [l, r) 都赋值为 v。
- swap(x, y) 交换两个元素。注意由于大部分 stl 容器只包含一个指针，交换是 O(1) 的。
- reverse(A + l, A + r) 翻转区间 [l, r)。
- sort(A + l, A + r) 排序区间 [l, r)。

- unique(A + l, A + r) 把区间里连续的相同的元素移至末尾，返回剩下的元素结尾位置。

```
int m = unique(A, A + n) – A;
```

- nth_element(A + l, A + k, A + r) 把 [l, r) 部分排序，使得 A[k] 这个位置的元素是正确的，比它小的在左边而比它大的在右边。
- 比如 nth_element(A, A + 5, A + 10) 会把第 6 小的（注意从 0 开始）放到 A[5]，前 5 小以任意顺序放到 A[0…4]，其他以任意顺序放到 A[6…9]。

- lower_bound(A + l, A + r, x) 返回第一个大于等于 x 的位置（要求序列有序）。取其下标可以 lower_bound(A + l, A + r, x) – A;
- upper_bound 同理，返回第一个大于 x 的位置。
- binary_search(A + l, A + r, x) 二分查找判断 x 是否在这个区间里出现，只返回true/false。

贪心

- 所谓贪心，就是说每次做出当前看来最好的选择。大多数时候这是错的，但是有些问题来说确实是对的。
- 贪心的优缺点：
- 优点：一般更容易实现，速度更快。
- 缺点：更难想猜出，更难证明。

例12. noip2012 国王游戏

【题目描述】
国王有 n 个大臣，他们每人（包括国王）左右手各有一个数 a_i, b_i；国王要让他们站成一排（国王自己必须站在最前面），随后给每个大臣奖赏。
对于一个大臣，如果他前面的人（不包括自己）左手上的数乘积是 A，他自己右手上的数是 b_i，那么他会获得 A / b_i 的奖赏（向下取整）。
国王想要让获得奖赏最多的大臣获得的尽量少，请你安排一种顺序满足他的要求，并输出奖赏最多的大臣最少获得多少奖赏。n≤1000。
【输入】
第一行包含一个整数 n，表示大臣的人数。
第二行包含两个整数 a 和 b，之间用一个空格隔开，分别表示国王左手和右手上的整数。
接下来 n 行，每行包含两个整数 a 和 b，之间用一个空格隔开，分别表示每个大臣左手和右手上的整数。
【输出】
输出只有一行，包含一个整数，表示重新排列后的队伍中获奖赏最多的大臣所获得的金币数。
【输入样例】　　【输出样例】
3　　2
1 1
2 3
7 4
4 6

算法分析

- 首先我们先思考一下当只有两个大臣的时候怎么做（当题目没有思路的时候，先考虑数据范围小的情况往往是有帮助的）。
- 如果国王左手上的数是 a_0，两个大臣左右手上的数是a_1,b_1,a_2,b_2；那么有两种排法：
- 第一个大臣排在前面，那么第一个大臣获得 a_0 / b_1 奖赏，第二个获得 a_0a_1 / b_2 奖赏，奖赏最多的大臣获得的奖赏就是 $\max(a_0 / b_1, a_0a_1 / b_2)$。
- 同样，如果第二个大臣排在前面，答案就是$\max(a_0 / b_2, a_0a_2 / b_1)$。

算法分析

- 比较 $ans_1=max(a_0 / b_1, a_0a_1 / b_2)$ 和 $ans2=max(a_0 / b_2, a_0a_2 / b_1)$，可以写成
- $ans_1=a_0\ max(b_2,a_1b_1) / b_1b_2$
- $ans_2=a_0\ max(b_1,a_2b_2) / b_1b_2$
- 可以想象，如果 $a_1b_1 \le a_2b_2$，那么 ans_1 肯定更小，因为这种情况下 $a_1b_1 \le a_2b_2$ 而 $b_2 \le a_2b_2$，所以 $max(b_2,a_1b_1) \le a_2b_2 \le max(b_1,a_2b_2)$。
- 同理，如果 $a_1b_1 > a_2b_2$，那么 ans_2 更小。
- 于是我们得到一个结论：只有两个人时，把 a_ib_i 更小那个放到前面一定更优。

算法分析

- 当有更多大臣的时候呢？
- 考虑最优解满足什么性质。如果我们交换最优解中两个相邻的大臣，那么他们前面的大臣得到的奖赏显然**不受影响**；而他们后面的大臣**也不受影响**。（想一想，为什么）
- 那么把这两个大臣单独拎出来，根据之前的结论，最优解中一定是 a*b 较小的放到前面。
- 这样的话，我们得出结论：最优解中相邻两个大臣，前面的 a*b 一定更小。否则可以交换相邻的大臣使得答案更小。

算法分析

- 于是最优解肯定是按大臣的 a*b 排序后的结果。所以将大臣按 a*b 排好序，计算每个大臣的奖赏就可以了（这题代码难点在于高精度）
- 通过这道题我们看出一种贪心的分析方式，它尤其适用于“将若干个物品重新排列使得____最小/最大”的问题：
- 假设我们有一组解，考虑如何**调整**能使它更优（对于重新排列，一般是交换**相邻物品**）；
- 最优解一定**不能**调整。

分治/二分

分治/二分

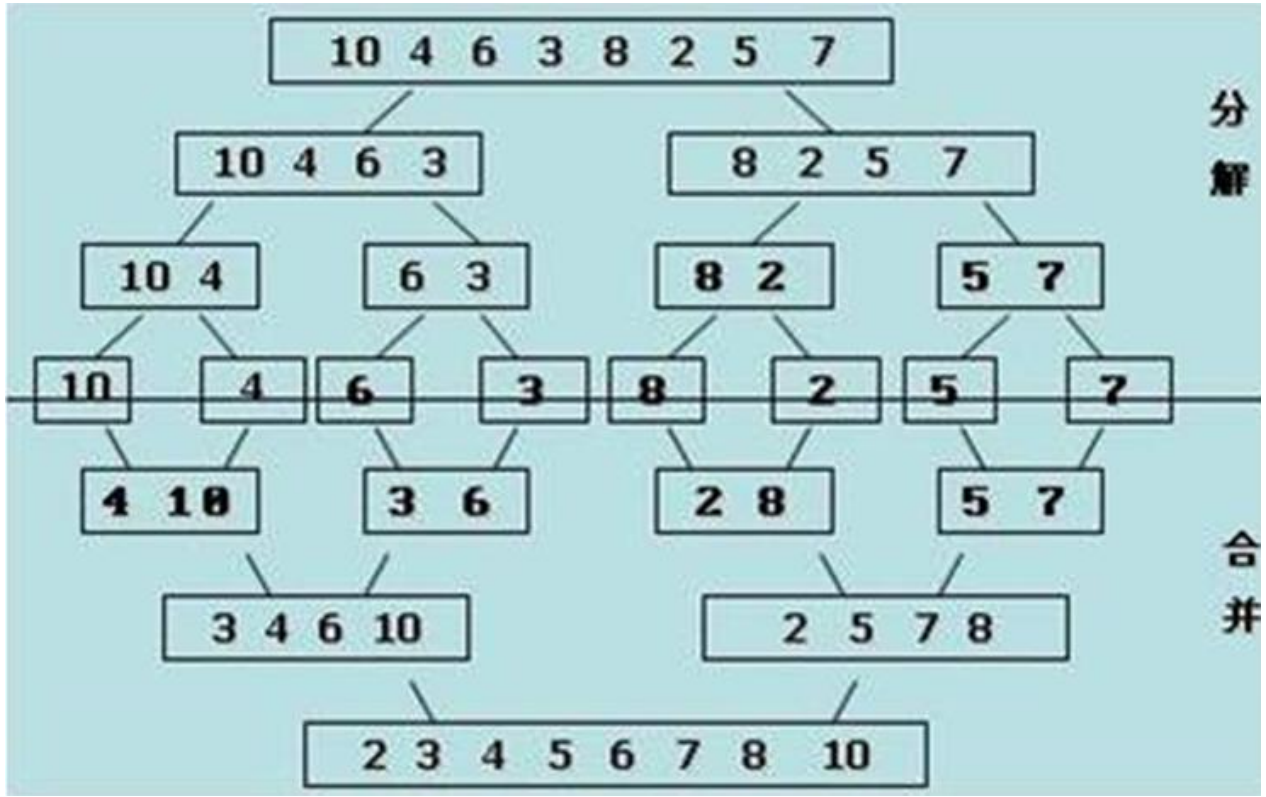
- 分治/二分算法本质上都是采用了“分而治之”的思想，因此放在一起来讲。
- 所谓分而治之，即把一个大的问题转化成一个、两个或多个小的问题之和。

例13. P1908 逆序对

【题目描述】
给定一个序列，逆序对是i < j 且 a[i] > a[j]的有序对。求其逆序对个数。n ≤ 500000。
【输入格式】
第一行，一个数n表示序列中有n个数。
第二行n个数，表示给定的序列。序列中每个数字不超过 10^9。
【输出格式】
输出序列中逆序对的数目。
【输入 #1】
6
5 4 2 6 3 1
【输出 #1】
11

算法分析

- 归并排序就可以帮我们来解决这个问题。
- 归并排序主要分两大步：分解、合并。



算法分析

- 在合并操作中，我们假设左右两个区间元素为：
- 左边：{3 4 7 9}　　右边：{1 5 8 10}
- 那么合并操作的第一步就是比较3和1，然后将1取出来，放到辅助数组中，这个时候我们发现，右边的区间如果是当前比较的较小值，那么其会与左边剩余的数字产生逆序关系，也就是说1和3、4、7、9都产生了逆序关系，我们可以一下子统计出有4对逆序对。接下来3，4取下来放到辅助数组后，5与左边剩下的7、9产生了逆序关系，我们可以统计出2对。依此类推，8与9产生1对，那么总共有4+2+1对。这样统计的效率就会大大提高，便可较好的解决逆序对问题。

例14. P4141消失之物

【题目描述】
ftiasch 有n个物品, 体积分别是 w_1 ,w_2,…,w_n。由于她的疏忽，第i个物品丢失了。要使用剩下的n-1物品装满容积为x的背包，有几种方法呢？她把答案记为 cnt(i,x) 。
【输入格式】
第一行两个整数n,m，表示物品的数量和最大的容积。 第二行n 个整数w1,w2,…,wn，表示每个物品的体积。
【输出格式】
输出一个n×m的矩阵，表示cnt(i,x)的末位数字。
【输入 #1】
3 2
1 1 2
【输出 #1】
11
11
21

算法分析

- 众所周知的 01 背包，但是这次你需要求出所有的 f[i][j] 表示去掉第 i 个元素后剩下的元素拼出体积为 j 的方案数。 $n,m \le 2000$。
- 跑 n 遍 dp 大家应该都会，复杂度是 $O(n^2m)$。
- 考虑分治。我们把物品分成两半，求出左边一半的背包，把它扔给右边递归；求出右边一半的背包扔给左边递归。

例15. P1429平面最近点对

【题目描述】
平面上有 n 个点，求距离最短的一对点的距离。n ≤ 200000。
【输入格式】
第一行：n，保证 2≤n≤200000 。
接下来n行：每行两个实数：x y，表示一个点的行坐标和列坐标，中间用一个空格隔开。
【输出格式】
仅一行，一个实数，表示最短距离，精确到小数点后面4位。
【输入 #1】
3
1 1
1 2
2 2
【输出 #1】
1.0000

算法分析

- 继续考虑分治。从某个位置画一条竖线把点分成两半，两半分别求答案，记现在的答案是 ans’ 。考虑两边之间求最短距离，那么离竖线超过ans’的点可以忽略掉，并且只需要算纵坐标差不超过ans’的点。
- 可以发现这种情况下需要算的情况数很少。

二分

- 一般来说，所谓二分，多指二分一个答案（至少目前如此）。
- 二分关键词：最大值最小/最小值最大。
- 另外还有套路算法 01 分数规划。

例16. P1083借教室

【题目描述】
第 i 天有 r[i] 个教室可以租借，一共有 m 份订单，第 j 个订单需要在第 s[j] 到 t[j] 天每天租借 d[j] 个教室。
如果我们按订单编号依次处理，到哪个订单的时候会出现无法满足要求的情况？n,m≤100000。
【输入格式】
第一行包含两个正整数n,m，表示天数和订单的数量。
第二行包含n个正整数r_i，表示第i天可用于租借的教室数量。
接下来有m行，每行包含三个正整数d_j,s_j,t_j，表示租借的数量，租借开始、结束分别在第几天。
【输出格式】
如果所有订单均可满足，则输出只有一行，包含一个整数0。否则（订单无法完全满足）输出两行，第一行输出一个负整数-1，第二行输出需要修改订单的申请人编号。

【输入 #1复制】	【输出 #1复制】
4 3 2 5 4 3 2 1 3 3 2 4 4 2 4	-1 2

算法分析

- 二分一个 mid，如果在 mid 之前就不满足条件了，mid 之后肯定也不满足；反之亦然，因此我们可以得知真实的答案是否小于等于 mid。
- 至于怎么知道前 mid 个订单能否全部满足...
- 差分！

01分数规划

- n 个物品，每个物品有一个价格 w[i] 和一个价值 v[i]。要求选出恰好 k 个使得总价值除以总价格尽量大。
- $0 < k \le n \le 100000$; $v[i], w[i] \le 10^9$。
- 类似的题目如poj2976、poj2728、poj3621

算法分析

- 考虑二分一个答案 mid，我们现在不关心答案具体有多大，只关心答案是否能大于等于 mid。
- sum(v[i]) / sum(w[i]) ≥ mid 等价于sum(v[i]) ≥ mid × sum(w[i]);
- 从而等价于sum(v[i] - mid × w[i]) ≥ 0。
- 那么只需要把每个物品的 v[i] - mid × w[i] 算出来并取前 k 大之和即可（nth_element！）
- 总的来说，01分数规划就是一些最大化 sum(w) / sum(v) 的问题，可以通过二分答案转化成最大化一个和。

主定理

- 主定理 (master theorem)，是一个用来分析分治时间复杂度的有效方法。它大概是这么说的：
- 如果有一个递归式 $T(n) = a\ T(n/b) + f(n)$，也就是说分治成 a 个规模为原来 1/b 的问题，f(n)是分解和合并的时间，那么记 $t = \log_b a$:
- 1. 如果 $f(n) = o(n^t)$（即，比nt要小），那么 $T(n) = O(n^t)$。这种情况下，递归树上的复杂度主要在于叶节点。
- 2. 如果 $f(n) = O(n^t \log^k n)$，那么 $T(n) = O(n^t \log^{k+1} n)$。这种情况下递归树每一层有一样的贡献。
- 3. 如果 $f(n) = \Omega(n^q)$ 而 $q > t$，存在c<1满足af(n/b)<=cf(n)，那么 $T(n) = O(f(n))$。这种情况下复杂度主要在递归树的根节点。

倍増

倍增

- 所谓倍增，就是把一个数据规模为n的问题分解成若干个2^ai的和，预处理数据范围内所有2^ai 的情况，再将这些规模为2^ai 的问题通过一定的方法合并，得出原问题的解。
- 分治是把整个问题分成几个互不重复的子问题，合并求解；倍增是找互为倍数关系的子问题之间的联系，再合并求解。
- 可以认为，倍增也体现了分治思想。
- 适用题型：问题规模大，且成倍数据规模的问题之间存在简单的递推关系，可以轻易地由小范围求出大范围。

RMQ问题

- RMQ（Range Minimum/Maximum Query），即区间最值查询，是指这样一个问题：对于长度为n的数列A，回答若干询问RMQ(A,i,j)，返回数列A中下标在i，j之间的最小/大值。
- 如何求最小/大值？
- 暴力扫描，单次查询的时间复杂度是O(n)。如果开一个二维数组ans[i][j]预存答案，预处理时间复杂度是$O(n^2)$，单次查询是O(1)。

ST算法

- ST算法是一种用于解决RMQ问题的离线算法，类似于线段树和树状数组，其功能特性差不多，当实现起来的话，显然是ST算法更为简便。
- ST算法的时间复杂度：预处理是O(nlogn)，查询是O(1)。
- 该算法是在倍增的思想基础上实现的。
- 应用：无修改时求区间最小/大值，特别是需要优化常数，或者询问次数大于序列长度的时候。

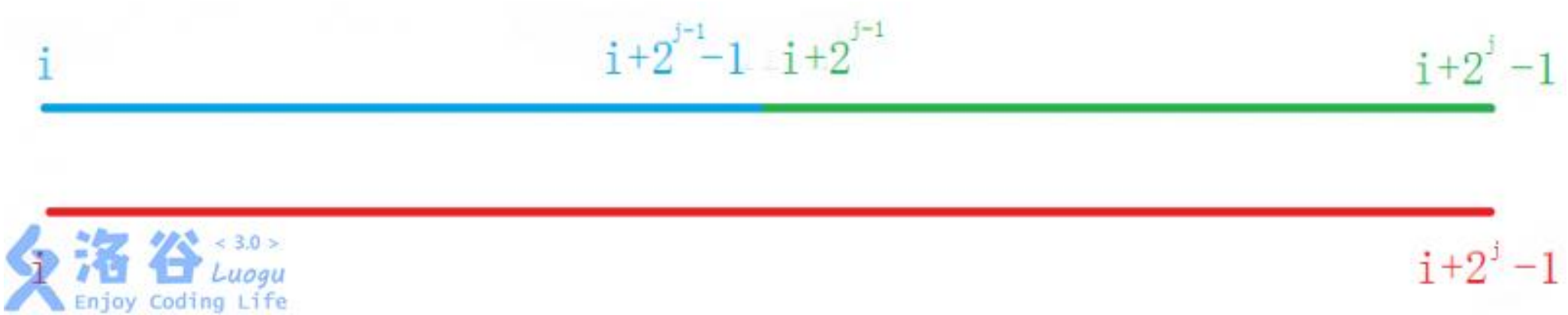
ST算法

- 以求最大值为例，f[i][j]表示从第i个位置开始，往后2^j个数字中的最大值，区间[i,i+2^j-1]的最大值。
- 比如数列：3，5，2，4，7，6

F	0(1)	1(2)	2(4)
1	3	5	5
2	5	5	7
3	2	4	7
4	4	7	-
5	7	7	-
6	6	-	-

预处理

- 显然，区间[i,i+2^{j-1}-1]和[i+2^{j-1},i+2^{j}-1]一定覆盖了区间[i,i+2^{j}-1]。
- 区间[i,i+2^{j}-1]内的最大值就是区间[i,i+2^{j-1}-1]和[i+2^{j-1},i+2^{j}-1]内的最大值中更大的一个。可以得出递推式：f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1]);（j从小到大枚举），f[i][0]=a[i]。
- 预处理的时间复杂度O(nlogn)。

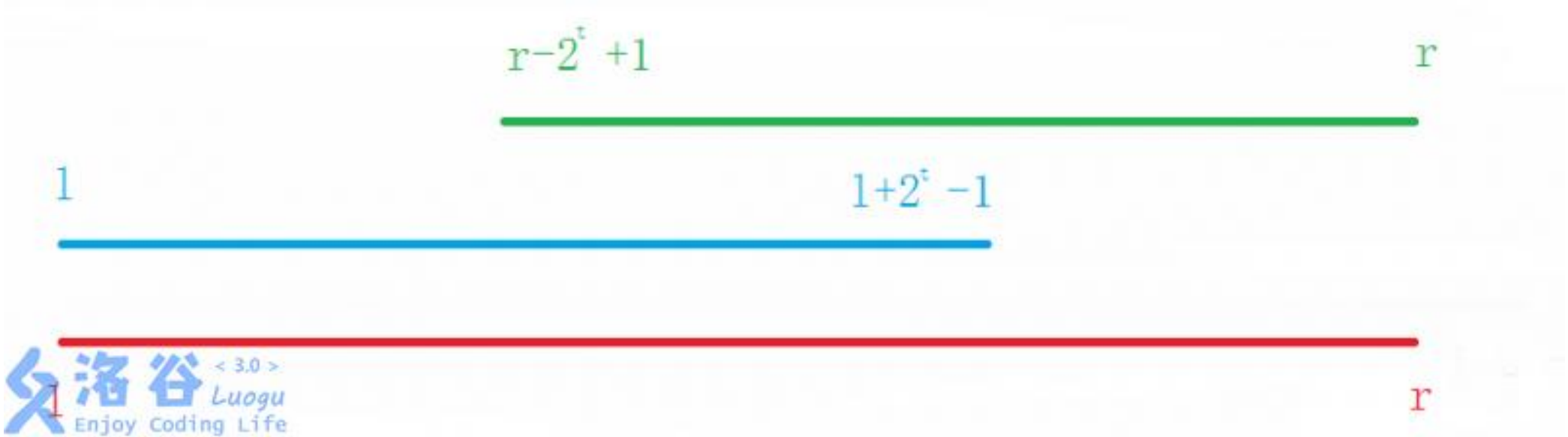


预处理

```
void ST(int n)//预处理
{
    for(int i=1;i<=n;i++)
            f[i][0]=a[i];//初始化以自己为起点2的0次方长的区间就是自己
    for(int j=1;(1<<j)<=n;j++)///枚举区间长度
       for(int i=1;i+(1<<j)-1<=n;i++)///枚举起点
          f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1]);
}
```

查询

- 先计算出一个满足$2^k<r-l+1<2^k+1$的t值，即小于区间长度的2的最高次幂。
- 显然，区间$[l,l+2^k-1]$和$[r-2^k+1,r]$一定覆盖了区间[l,r]。
- 区间[l,r]内的最大值就是区间$[r-2^k+1,r]$和$[l,l+2^k-1]$内的最大值中更大的一个。可以得出递推式：ans[l][r]=max(f[l][k],f[r-(1<<k)+1][k]);



查询

```
int RMQ(int l,int r)
{
    int k=trunc(log2( r-l+1 ));//trunc函数向下取整
     //log2函数求以2为底的对数，log2(x)=log(x)/log(2);
     //log函数求以e为底的对数，头文件cmath
    printf("k=%d\n",k);
    return max(f[l][k], f[r-(1<<k)+1][k]);
}
```

Thanks