基础Algorithm

2021年10月12日　　10:43

二分搜索， bitmap， 左右逼近搜索2值之和等于某个值， r-b tree, b tree, trie

二分搜索

最简单情况是从数组的给定范围内中找到指定数字，下面的情况都是 两边闭区间，即[low,high]
其他情况(左开右闭，左闭右开，双开) 都可以转化为 双闭。

当low==high时，区间中 只有一个点。 low>high时，空区间。
所以循环条件就是 while(low <= high)

当arr[mid] == num_to_find 的时候直接return
当arr[mid] < num_to_find 的时候，查找区间更新为 low=mid+1
当arr[mid] > num_to_find 的时候，查找区间更新为 high=mid-1

还有其他情况：当有多个满足条件的数时，返回最小的那个，或返回最大的那个(数组查找就是返回 数组中下标最小的，下标最大的)

返回下标最小：
循环条件不变，还是while(low <= high)
当找到一个满足条件的数时，如arr[mid]==num_to_find，不能直接return，因为可能存在比mid更小的下标，且元素值满足条件。
此时必须更新high，如果high更新为mid的话，会发生死循环。
　　假设不再存在满足条件的元素了，那么low一定会往上更新，直到和high一样大小，此时就会死循环
　　如果还存在的话，会被继续划分区间，直到上面的再次发生。
所以我们应该更新high=mid-1。　　。。。这种是 arr[mid]==num 的分支。
其他情况不变，依然是：low=mid+1，high=mid-1
假设我们找到arr[mid]==num_to_find，此时更新high=mid-1，如果mid就是下标最小的元素的话，那么之后的划分会不断地提升low，一直到high-1,发现arr[high-1]依然小于num_to_find，此时low就会变成high+1，就是最开始的mid。就找到了满足条件的下标(在low中)。

返回下标最大：
和上面同理。
arr[mid]==num时，low=mid+1.

如果没有找到元素的话，那么找最小下标的二分就会返回一个下标k，如果将查找数插入到下标k，那么此时已然是一个有序的数组。
找最大下标时返回的下标k，将查询数插入到下标k，也会得到一个有序的数组。
。。上面2个的区别是，最小下标时，原先下标k的值是后移， 最大下标时，原先下标k的值前移。。或者说，最小下标，插在k前面，最大下标，插在k后面。

==========

二分法有下面7种变式

1 是否存在字数t
2 找到大于t的第一个数
3 找到大于等于t的第一个数
4 找到小于t的最后一个数
5 找到小于等于t的最后一个数
6 是否存在数字t，返回第一个t
7 是否存在数字t，返回最后一个t

二分法的适用条件---P划分区间

要搞清什么是二分法及其适用条件，即什么样的问题可以用二分法解决，如果解决。答案是二分法的每一个变式，都是在P划分区间上查找划分点。

P划分区间：给定一个元素为s_i的闭区间S 和一个对s_i的判断条件P(即P(s_i)返回true或false)，如果S中存在某个元素s_x使得区间上在s_x之前的元素(不包括s_x)都不满足P，而s_x及其之后的元素都满足P，那么就称区间S是P划分的，且s_x是划分点，下面简写为划分点x。

划分点给予我们的信息：

对于S中任意一个元素s_i，如果s_i满足P，那么s_i后面的元素都满足P
如果s_i不满足P，则s_i前面的元素都不满足条件P

变式2,3，大于/大于等于t，第一个数

大于/大于等于t，就是一个条件P。二分查找中，区间S中元素是升序的，显然S满足P划分。所以我们的任务就是找到第一个满足P的元素，也就是划分点。

```
int findPartition(vector<int> nums,bool (*P)(int)) {
    if(P(nums[nums.size() - 1]) == false) return -1; //there no partition point
    int left = 0,right = nums.size() - 1,mid;
    while(left < right){
        mid = left + ((right - left) >> 1);
        if(P(nums[mid])){
            right = mid;
        }else{
            left = mid + 1;
        }
    }
    return left;
}
```

。。bool (*P)(int) 方法指针。。

1）若划分点不存在，函数返回 -1

如果不存在划分点，那说明区间中的元素都不满足条件P，此时只需要用区间最后一个元素对P进行检查即可。

2）若划分点存在，算法必然终止。

mid = (left + right) / 2，即left和right的中位数向下取整，可见mid >= left。又因为while循环条件为left不等于right，故而循环体中mid < right。所以每次迭代，区间长度至少减1。当区间长度为1时，算法终止，此时left 等于right。

3）若划分点存在，算法终止时，left所指元素即为划分点。

变式4,5 小于/小于等于t，最后一个满足条件的元素

```
int findPartition(vector<int> nums,bool (*P)(int)) {
    if(P(nums[0]) == false) return -1; //there no partition point
    int left = 0,right = nums.size() - 1,mid;
    while(left < right){
        mid = left + ((right - left + 1) >> 1); //attention,up[(left + right) / 2]
        if(P(nums[mid])){
            left = mid;
        }else{
            right = mid - 1;
        }
    }
    return left;
}
```

计算中位数时候，要取left和right的向上取整，这样就有left < mid，mid <=right，才能保证区间长度一直在减小，算法必然终止。

。。就是 能和mid 相等的 那边(left/right) 必须做出修改。 或者说 mid修改。。

变式6,7
要确定是否存在某数，并且返回其第一个或最后一个的位置，这可以当做之前情况的扩展。下面2点即可解决此种问题：
查找第一个t，即找到第一个大于等于t的元素，还需要专门考虑此数是否存在
查找最后一个t，就是查找最后一个小于等于t的元素，还需要专门考虑此数是否存在
在之前的基础上，return时增加判断条件nums[left]==target即可。

```
return nums[left] == target ? left : -1;
```

==========

bitmap， 压缩数据，比如有40亿个数，求一个数是否在里面。。。 不过40亿次 读写(修改)。。

==========

左右逼近，来搜索 2个值， 这2个值的和 等于 target。

LT的3sum，4sum，3sum closed

```
// int l = pos, r = size-1;
// while(l < r)
// {
//     int t = nums[l]+nums[r];
```

```
//     if(t > target) {do{r--;}while(l<r and nums[r]==nums[r+1]);}
//     else if(t < target) {do{l++;}while(l<r and nums[l]==nums[l-1]);}
//     else
//     {
//         v.push_back(nums[l++]);
//         v.push_back(nums[r--]);
//         vv.push_back(v);
//         v.pop_back(), v.pop_back();
//         while(l<r && nums[l]==nums[l-1]) l++;
//         while(l<r && nums[r]==nums[r+1]) r--;
//     }
// }



// for i := 0; i < l - 3; i++ {
//     if i > 0 && nums[i] == nums[i-1]{
//         continue
//     }
//     for j := i + 1; j < l - 2; j++ {
//         if j > i + 1 && nums[j] == nums[j-1] {
//             continue
//         }
//         lo := j + 1
//         hi := l - 1
//         for lo < hi {
//             sum := nums[i] + nums[j] + nums[lo] + nums[hi]
//             if(target > sum) {
//                 lo++
//             } else if (target < sum) {
//                 hi--
//             } else {
//                 ans = append(ans, []int{nums[i],nums[j],nums[lo],nums[hi]})
//                 lo++
//                 hi--
//                 for nums[lo] == nums[lo - 1] && lo < hi {
//                     lo++
//                 }
//                 for nums[hi] == nums[hi + 1] && lo < hi {
//                     hi--
//                 }
//             }
//         }
//     }
// }


============================
```

r-b tree

trie

b tree

==========

快慢指针找入口

假设非环部分的长度是x，从环起点到相遇点的长度是y。环的长度是c。
现在走的慢的那个指针走过的长度肯定是x+n1*c+y，走的快的那个指针的速度是走的慢的那个指针速度的两倍。这意味着走的快的那个指针走的长度是2(x+n1*c+y)。

还有一个约束就是走的快的那个指针比走的慢的那个指针多走的路程一定是环长度的整数倍。
根据上面那个式子可以知道2(x+n1*c+y)-x+n1*c+y=x+n1*c+y=n2*c。

所以有x+y=(n2-n1)*c,这意味着什么？我们解读下这个数学公式：非环部分的长度+环起点到相遇点之间的长度就是环的整数倍。这在数据结构上的意义是什么？现在我们知道两个指针都在离环起点距离是y的那个相遇点，而现在x+y是环长度的整数倍，这意味着他们如果从相遇点再走x就到了起点。
那怎么才能再走x步呢？答：让一个指针从头部开始走，另一个指针从相遇点走，等这两个指针相遇那就走了x步此时就是环的起点。

。。n1,n2 应该是 圈数

==========

==========

二叉树的 先序，中序，后序

递归，stack/deque，Morris

后序的stack

```
// while (root || !todo.empty()) {
//     if (root) {
//         todo.push(root);
//         root = root -> left;
//     } else {
//         TreeNode* node = todo.top();
//         if (node -> right && last != node -> right) {
//             root = node -> right;
//         } else {
//             nodes.push_back(node -> val);
//             last = node;
//             todo.pop();
//         }
//     }
// }
```

先序的 stack，Morris

```
// Stack<TreeNode> rights = new Stack<TreeNode>();
// while(node != null) {
//     list.add(node.val);
//     if (node.right != null) {
//         rights.push(node.right);
//     }
//     node = node.left;
//     if (node == null && !rights.isEmpty()) {
//         node = rights.pop();
//     }
// }
//
// while (root) {
//     if (root -> left) {
//         TreeNode* pre = root -> left;
//         while (pre -> right && pre -> right != root) {
//             pre = pre -> right;
//         }
//         if (!pre -> right) {
//             pre -> right = root;
//             nodes.push_back(root -> val);
//             root = root -> left;
//         } else {
//             pre -> right = NULL;
//             root = root -> right;
//         }
```

```
//     } else {
//         nodes.push_back(root -> val);
//         root = root -> right;
//     }
// }
// Morris
```

后序：

```
    while(!stack.isEmpty() || p != null) {
        if(p != null) {
            stack.push(p);
            result.addFirst(p.val);  // Reverse the process of preorder
            p = p.right;             // Reverse the process of preorder
        } else {
            TreeNode node = stack.pop();
            p = node.left;           // Reverse the process of preorder
        }
    }
```

==========

==========

https://www.cnblogs.com/kkun/archive/2011/11/23/2260312.html

经典排序算法 - 快速排序Quick sort
经典排序算法 - 桶排序Bucket sort
经典排序算法 - 插入排序Insertion sort
经典排序算法 - 基数排序Radix sort
经典排序算法 - 鸽巢排序Pigeonhole sort
    计算数组的 min，max，申请[max-min+1]的空间，然后 [val-min]++，最后生成排序后数组。
经典排序算法 - 归并排序Merge sort
经典排序算法 - 冒泡排序Bubble sort
经典排序算法 - 选择排序Selection sort

经典排序算法 - 鸡尾酒排序Cocktail sort

又称双向冒泡排序，是冒泡排序的一种变形。不同之处在于 排序时 是以双向 在序列中进行排序。

先正向执行一次 冒泡，然后反向执行一次 冒泡， 然后继续正向 反向 交替。

冒泡是指： if([i] > [i+1]) swap([i], [i+1])

一些优化：

1. 没有交换，说明有序
2. for (0 .. sz/2) { for(i .. len-i-1){} for(len-1-i-1 .. i){}}

如果序列 一开始 就大部分有序，则 复杂度 可以 从 平时的O(n^2) 降到 O(n)

经典排序算法 - 希尔排序Shell sort

经典排序算法 - 堆排序Heap sort序

经典排序算法 - 地精排序Gnome Sort

冒泡，发生改动后，反向冒泡回去。

优化到最后 就是 选择排序。

经典排序算法 - 奇偶排序Odd-even sort

第0,2,4,..和 它的后面进行比较，然后发生可能的swap， 然后第1,3,5.. 和 它后面 进行比较，然后发生可能的swap， 然后 第0,2,4..， 第1,3,5.. 交替 直到 有序。

经典排序算法 - 梳排序Comb sort

改良自 冒泡排序 和 快速排序， 旨在 消除乌龟，即在 数组尾部 的小数值，这些数值是 造成冒泡排序 缓慢的 主因。相对的，兔子，即在 数组前端的 大数值，不影响冒泡排序的 性能。

冒泡排序中，只比较数组中 相邻的 两项，即 比较的两项的 间距(Gap) 是1， 梳排序提出 这个间距可以大于1， 希尔排序(改良自 插入排序) 也有同样的观点。

梳排序中，开始时的间隔 设置为数组长度，并在 循环中 以 固定的比率 降低，通常 递减率设置为 1.3 。 在一次循环中， 梳排序 和冒泡排序一样 把数组 从头到尾 扫描一次，比较及swap，不同的是 两项的间距不固定于 1 。 如果 间距递减至1，梳排序 假定输入数组 大致排序好，并以 冒泡排序 做最后的 检查 和修正。

经典排序算法 - 耐心排序Patience Sorting

将数组元素 分类成 很多堆 再 串接回数组的 一种排序算法。受到纸牌游戏 耐心 的启发和命名。 算法的变体 有效地计算 给定 阵列中 最长的 增加子序列的长度。

1. 创建堆数组
2. 比较当前指向的元素 和 每个堆的 第一个元素，计算出 比当前元素小的 堆数量。
3. 如果当前元素 比 所有堆的 第一个元素大，创建 新堆 并加入到 堆数组中，否则将 当前元素 加入到 第"比当前元素小的 堆数量" 个堆
4. 分类完成后 将 每个堆 反序， 然后 对每个堆 再做 耐心排序(。。应该不需要了，只需要 反序，然后 第五步 merge 就可以了。)
5. 最后将 每个 堆串接 并 存储回 原本的数组

耐心分类 与 被称为 弗洛伊德游戏 的 纸牌游戏 密切相关：

1. 发出的 第一张牌 形成 一个 由 单张牌 组成的 牌堆
2. 后续的 每张牌 放置在 一些 现有的 堆上，其 顶部的值 不大于 新卡的值，或者放在 所有 现有 牌堆的 右侧，形成 新的牌堆。

。。这个 和 算法 是反的，这里是 堆顶 要小于 牌， 算法是 堆顶要大于当前元素。

用于找到 最长 增加 子序列 的算法

首先，执行上述的 排序算法。堆的数量 就是 最长子序列的长度。

经典排序算法 - 珠排序Bead Sort

珠排序是一种自然排序算法。无论是 电子 还是 实物上的实现，珠排序都能在 O(n) 时间内完成；然而，该算法 在 电子上的 实现 明显 要比 实物 慢很多，并且 只能对 正

整数 序列 进行排序。并且，在最好的情况下，依然需要 O(n^2) 的空间。
珠排序可以类比于 珠子 在平行的 竖直杆上滑动，就像算盘一样，然而，每一竖直杆 都有 珠子数目的限制。
。。。
。。就是 最大值 mx，申请 mx列 * sz行 的空间，每行是一个 元素， 这个元素是多大，那么 就有 多少列 上放上 珠子， 然后 通过 重力 下降， 最终形成 一个 非升的xx，然后 统计 每 行 的 珠子个数 就得到了 排序后的 结果。
。。。

经典排序算法 - 计数排序Counting sort

经典排序算法 - Proxmap Sort
基数排序 和 桶排序的 混合

经典排序算法 - Flash Sort
号称O(n) 时间复杂度。需要额外内存O(n)，不是 stable(稳定) 的算法。
类似于桶排序，但是在 桶排序的 基础上增加了一些 对数据分布的 猜测，减少了 桶的用量。
对于待排序数组 [6,3,2,5,1,4,7,9,8]
首先遍历数组，找到 max9，min1，我们设置 m 个桶，对于 数组中的 每个数字，我们计算它 属于 哪个桶： K(ai) = 1 + int((m - 1)*(ai - amin)/(amax - amin))。
假设 m = 6，对于上述数组计算得出： [4,2,1,3,1,2,4,6,5]，这里的时间复杂度是O(n)。
此时，我们并不直接存 这组数据，这只是中间值。真正存的是 另一个数组 L，L的长度是 6 (也就是 m 的值)， L 如下： [2,2,1,2,1,1]
L[i] 表示 第 i 个桶里面有 几个数，然后对 L[i] 做前缀和得到： [2,4,5,7,8,9]
L[i] 表示 第 i 个桶 index 的上限，这里 时间复杂度 O(m)
对于 原数组的 每一个数，我们将其放置到 L[i] 的位置，然后 --L[i]，再将 L[i] 位置的 数放到 其对应的位置上，然后依次类推 知道 将 每个数 放到 对应位置。

经典排序算法 - Strand Sort

经典排序算法 - 圈排序Cycle Sort
交换次数最少的排序
效率并不高
这个 cycle 在于 需要交换的数据 形成圈

如:
Array 4 3 2 5 5 6 要处理的数组
Result 2 3 4 5 5 6 结果
pos 0 1 2 3 4 5 下标
从下标0的元素开始观察。4 需要到下标 2 而下标2的元素为 2 需要到下标0 。刚好可以回到4。 也就是形成了 4-2 这样的圈
接下来是3 需要到下标1 而3本身就在1上。那么3自成一圈
再接下来是2。这个2在圈1中所以跳过。
然后是5 5需要在3上（也许你认为4也是可以的。但是实际上我们是按照类似计数排序的手法。算法是稳定的） 5自成一圈
下一个5 也是一样的。
而下一个6同样如此。

实际操作中：
我们从4开始进行一次类似计数排序一样的位置。多加一步。判断当前位置上的元素

是否应该放在4这个位置上。如果不是。就对这个元素所该放的元素继续访问是否应该放在4这个位置上。直到可以。

经典排序算法 - 图书馆排序(Library Sort)

最好时间复杂度	O(nlogn)
平均时间复杂度	O(nlogn)
最坏时间复杂度	O(n^2)
空间复杂度	O(n)
是否稳定	是

基于折半查找(二分搜索) 的插入排序，插入时在元素附近空出一定位置，这样插入后 移动元素 的复杂度 由原来的 O(n) 下降到 平均 O(1)，于是整个算法的复杂度 达到 O(nlogn)。
当输入正序 或 倒序时，插入点都在同一位置，"留空位" 的策略失效，这时就是 最坏情况。

排序算法之 Smooth Sort

最好时间复杂度	O(n)
平均时间复杂度	O(nlogn)
最坏时间复杂度	O(nlogn)
空间复杂度	O(1)
是否稳定	否

Smooth Sort基本思想和Heap Sort相同，但Smooth Sort使用的是一种由多个堆组成的优先队列，这种优先队列在取出最大元素后剩余元素可以就地调整成优先队列，所以Smooth Sort不用像Heap Sort那样反向地构建堆，在数据基本有序时可以达到O(n)复杂度。Smooth Sort算法在维基百科上有详细介绍。

Smooth Sort是所有算法中时间复杂度理论值最好的，但由于Smooth Sort所用的优先队列是基于一种不平衡的结构，复杂度因子很大，所以该算法的实际效率并不是很好。

TimSort
稳定的算法， 是 归并排序 和 二分插入排序 的混合。
号称世界上最快的算法。

指数搜索
也称为 加倍搜索， 用于 大型数组 中 搜索元素。
分为2步：

1. 试图 找出 目标元素 所在的范围
2. 在这个范围中 使用 二叉搜索 来寻找准确位置。

二分插入排序

插入排序很简单，从第二个元素开始，依次向前移动 交换元素，直到找到 合适的位置。
可以使用 二分查找 来减少 插入时 元素的比较次数。(不能减少swap次数)
二分插入排序 在 小数据集 场景下 排序效率非常高。

归并排序
使用分治策略。 递归 将数组不断 二分，直到无法分割(即 空数组 或 只有一个元素)， 然后进行合并排序。 合并操作 简单来说，就是将 2个 较小的有序数组 合并成 一个更大的 有序数组。
归并排序 主要为 大数据集 场景设计

TimSort流程
如果数组长度小于 指定 阈值(MIN_MERGE)， 直接使用 二分插入算法完成排序
否则：
1. 从这个数组 左侧开始，执行 升序运行 得到 一个子序列
2. 将 这个 子序列 放入 运行堆栈中， 等待 执行合并
3. 检查 运行堆栈中的 子序列，如果满足 合并条件 则 执行合并
4. 回到步骤1

升序运行
从数组中查找 一个 连续 递增 或 递减 的子序列的过程，如果 子序列 为 降序 则 反转为升序。
。。反转有什么用，又没有办法合并，除非 是多个 降序合并，但是 这种不需要 反转啊。

MIN_MERGE 在 Tim Peter 实现的 C 版本中 为 64， 但是 实际经验中 设置 32更好，所以 java中是 32 。

合并条件
为了 执行 平衡合并 (让合并的序列 大小尽可能相同)， 制定了一个 合并规则， 对于在 栈顶的 3个 子序列，分别用 x,y,z 表示 它们的长度， 其中 x 在栈顶， 它们必须 维持 2个规则：
1. Z > X+Y
2. Y > X

一旦不能全部满足，则 将 y 与 min(x, z) 进行合并。 合并后，如果还是不满足，那就继续合并。
如果只有2个 子序列，那么满足 Y > X 即可。

合并内存开销
原始归并排序 空间复杂度是 O(n)。
为了实现中间型，TimSort 进行了一次归并排序，时间，空间 都比 O(n) 小。

优化是为了 尽可能减少 数据移动，占用 更少的 临时内存， 先找到 需要移动的元素，然后将 较小的序列 复制到 临时内存，在 按最终顺序排序 并 填充到 组合序列中。

比如，我们要 合并 X[1,2,3,6,10], Y[4,5,7,9,12,14,17]，X中最大为 10， 我们可以通过二分搜索，确定，它需要插入到 Y 的 第5 个位置， 而Y 中最小是4，需要插入到 X 的第四个位

置 才能保证顺序，那么 就知道 [1,2,3],[12,14,17] 是不需要移动的， 我们只需要 移动 [6,10] [4,5,7,9]，只需要分配一个 大小为 2 的 临时存储 即可。

合并优化
在合并2个数组时，为了 优化合并的过程，设定了一个 阈值 MIN_GALLOP， 当 B 中元素 向 A 合并时， 如果 A 中连续 MIN_GALLOP 个 元素 比 B 中 某个元素小，那么 就进行 GALLOP 模式。
根据基准测试，比如当A 中连续 7 个以上元素 比 B 中 某个元素 小时 进入该模式的 效果最好，所以 初始值 为 7。

进入 gallop 模式后，搜索 算法 变成了 指数搜索，分为 2个步骤，比如 想 确定 A 中元素 x 在B 中 准确的位置：

1. 首先在 B 中找到 合适的 索引区间 (2^(k-1), 2^(k+1) - 1)， 使得 x 落在 这个范围内。
2. 然后在 第一步 的范围中 通过 二分搜索 来找到准确的 位置。

双支点快速排序

Java 对于 基本数据类型 的数组 使用 双支点快速排序， 对 ref类型 采用 mergeSort。

因为 基本类型 无法区分 2个相同的数， 所以 可以 快速排序。
而 ref类型 是可以 区分 两个 equal 的 值得，所以 需要 稳定性 的 mergeSort。
。。但是 TimSort 小范围的时候 使用 快速排序，就不可能保证 稳定性的。 我觉得 Java 就是 用 TimSort ，而不是 mergeSort。

哨兵插入排序

指数搜索

二分查找 是一种分治算法，不断 缩小有序数组中 要查询的值 的可能存在的 区间，每次缩小

一半。
指数搜索 和 二分查找类似，也是一种 搜索 固定值的 分治算法，只不过 它的空间范围 是以 指数形式 缩小的。

例子
在[1,3,4,5,6,7,9,10,11,13] 中搜索 9：

1. 从下标0 开始，由于 1!=9，所以 不在下标0处， 向后依次判断 下标1,2,4,8 的值 是否小于等于 9
2. 发现 下标8 的值 不小于等于 9，就停止判断， 此时知道 要查找的值 肯定在 [4,8) 中。
3. 对 [4,8) 是用 二分搜索。

学习型索引(。。机器学习) alex 采用了一种 叫 gapped array 的数组， 这种数组 在 每个元素之间 留了一些空隙， 这样在插入元素的时候， 如果要插入的位置 正好是空隙的话，就可以直接插入， 如果不是的话，左移 或 右移 的个数 也不会很多
查找这种 洗漱的数组， 指数搜索 明显 比 二分查找 要快。

==========

==========

https://www.cnblogs.com/LacLic/p/14916279.html
ACM算法模板（里面很多很多很多模板）

```
struct DSU {
    std::vector<int> f, siz;
    DSU(int n) : f(n), siz(n, 1) { std::iota(f.begin(), f.end(), 0); }
    int leader(int x) {
        while (x != f[x]) x = f[x] = f[f[x]];
        return x;
    }
    bool same(int x, int y) { return leader(x) == leader(y); }
```

```
    bool merge(int x, int y) {
        x = leader(x);
        y = leader(y);
        if (x == y) return false;
        siz[x] += siz[y];
        f[y] = x;
        return true;
    }
    int size(int x) { return siz[leader(x)]; }
};



template<class T>
struct Fenwick{
    int n;
    std::vector<T> t;
    Fenwick(int n) : n(n + 1), t(n + 1) {}

    void isr(int p,T v) {
        for(++p;p<n;p+=p&-p) t[p] += v;
    }

    T qry(int p) {
        T res = 0;
        for(++p;p;p-=p&-p) res += t[p];
        return res;
    }
};
```

==========

==========

Sieve of Eratosthenes
埃拉托斯特尼筛法，简称埃氏筛

寻找质数，每次将 质数的 所有倍数 设置为false。
i += 2

==========

lee215:
When use binary search,
I suggest you using my template

```
while (left < right) {
    int mid = (left + right) / 2;
    if (condition)
        right = mid;
    else
        left = mid + 1;
}
return left;
```

。。不过我感觉还是会有原先的，原先的简单。这个的边界条件实在是太恐怖了。

lee215

```
        int left = 1, right = 1e9, n = A.size();
        while (left < right) {
            int mid = (left + right) / 2, take = 0;
            for (int i = 0; i < n; ++i)
                if (A[i] <= mid) {
                    take += 1;
                    i++;
                }
            if (take >= k)
                right = mid;
            else
                left = mid + 1;
        }
        return left; //left == right
```

votrubac

```
    int l = 1, r = 1000000000;
    while (l < r) {
        int m = (l + r) / 2, take = 0;
        for (int i = 0; i < nums.size() && take < k; ++i) {
            take += nums[i] <= m;
```

```
            i += nums[i] <= m;
        }
        if (take < k)
            l = m + 1;
        else
            r = m;
    }
    return l;
```

。。最重要的 condition ，是 >= ? >? 。。 所以还是 用原先的 带 ans 的吧。

==========

求100万内任意数的 质因子分解时：

只需要预先算出 1000 以内的 质数，如果 除以1000以内的质数后，还有剩余，那么 剩余的这个数 也是 质数。

特别是如果 给你 99999997 这种，如果缓存 100万内的 质数(有7万多个)，你需要 遍历7万多。如果 给你 几千个99999997， 就tle 了。

==========