# 查找

一、综述

评价标准：

通常把查找过程中对关键字需要执行的平均比较次数（也成平均查找长度ASL）作为衡量查找算法效率优劣的标准。

ASL=∑PiCi (i=1,2,3,…,n)，

其中：Pi 为查找表中第i个数据元素的概率，Ci为找到第i个数据元素时已经比较过的次数。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类别 | 查找方法 | 平均查找长度ASL | | 适用场景 | 时间复杂度 |
| 成功 | 不成功 |
| 线性表的查找 | 顺序查找 | （n+1）/1 | n | 针对无序序列的一种最简单的查找方式。 | O(n) |
| 二分查找（有判定表） | ≈Log2(n+1)/2-1 |  | 有序。适合一经建立就很少改动，但需要经常查找的线性表.  并且只适用于顺序存储结构的序列。要求序列中的元素基本不变，在需要做删除和插入操作的时候，会影响检索效率。 | O(logN) |
| 分块查找 | 性能介于属于查找和二分查找之间。 | | 索引结构存储，分块有序 |  |
| 树表的查找 | 二叉排序树 | 最好log2n | (n+1)/2 |  | log2n |
| 平衡二叉树 | log2n | |  | log2n |
| B树 |  | |  | 在满足条件的情况下，m越大越好。 |
| B+树 |  |  | 数据库，数据量特别大 |  |
| 哈希查找 | 哈希构造和查找过程一样 |  |  |  | O（1） |

二、常用数据结构上的查找：

查找通常与遍历、增、删、改一起。

1、字符串查找、翻转、匹配

暴力法。KMP，trie，Manaches Algorithm，AC自动机。

2、数组查找（如二分查找、杨氏矩阵查找）

3、链表翻转、遍历、查找、删除、合并

多个指针。

4、Hash表查找

通常与其他情况结合考察。

5、树

遍历（前序、中序、后序）

set、map

高级树的查找（红黑树、B树、R树）

6、图

遍历

查找（DFS、BFS）

最短路径算法

**一、顺序查找**

　　条件：无序或有序队列。

　　原理：按顺序比较每个元素，直到找到关键字为止。

　　时间复杂度：O(n)

**二、二分查找（折半查找）**

　　条件：有序数组

　　原理：查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；

　　　　　如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。

　　　　　如果在某一步骤数组为空，则代表找不到。

　　　　　这种搜索算法每一次比较都使搜索范围缩小一半。

　　时间复杂度：O(logn)

**三、二叉排序树查找**

　　条件：先创建二叉排序树：

　　　　　 1. 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

　　　　　 2. 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

　　　　　 3. 它的左、右子树也分别为二叉排序树。

　　原理：

　　　　在二叉查找树b中查找x的过程为：

　　　　1. 若b是空树，则搜索失败，否则：

　　　　2. 若x等于b的根节点的数据域之值，则查找成功；否则：

　　　　3. 若x小于b的根节点的数据域之值，则搜索左子树；否则：

*4. 查找右子树。*

　　时间复杂度：



**四、哈希表法（散列表）**

　　条件：先创建哈希表（**散列表**）

　　原理：根据键值方式(Key value)进行查找，通过散列函数，定位数据元素。

　　时间复杂度：几乎是O(1)，取决于产生冲突的多少。

**五、分块查找**

　　原理：将n个数据元素"按块有序"划分为m块（m ≤ n）。

　　　　　每一块中的结点不必有序，但块与块之间必须"按块有序"；即第1块中任一元素的关键字都必须小于第2块中任一元素的关键字；

　　　　　而第2块中任一元素又都必须小于第3块中的任一元素，……。

　　然后使用二分查找及顺序查找。

【数组中查找】

* **求解多个有序数组的中位数**

[题目意思] http://blog.csdn.net/chen77716/article/details/7981307

如果多个有序数组能在一起排序，则取位置为中间的数字，如果有奇数个数字则中位数只有一个;若为偶数个则有两个，一般取第一个，也称下中位。但不能把数组合在一起做插入或快速排序，因为数据可能是海量的。

该题目可能有很多种实现方法，而我们给出一种仅依赖中位数性质的算法。如果存在一个已经排好序的大数组（有序数列），则会发现几个性质：

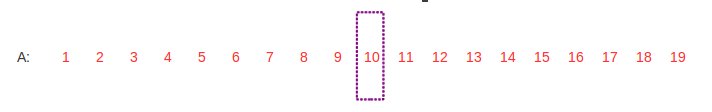
1. 对称性：中位数前面的数字与后面的数字一样多（在偶数元素情况下或相差1）
2. 确定性：若某个数字前后的数字数量相同（或相差1）则为该数列的中位数
3. 不变性：删除前N个元素与后N个元素后，中位数不变
4. 关联性：从大数组中按顺序任意取走n-1组元素，与剩下的元素共构成n个子列，每个子列存在中位数：m1,m1...mn,min，max分别为其中最小者与最大者，则原数列的中位数m满足：min<=m<=max  
   该性质可通过反证法获得

根据上述性质构造下面基于3分查找的算法：

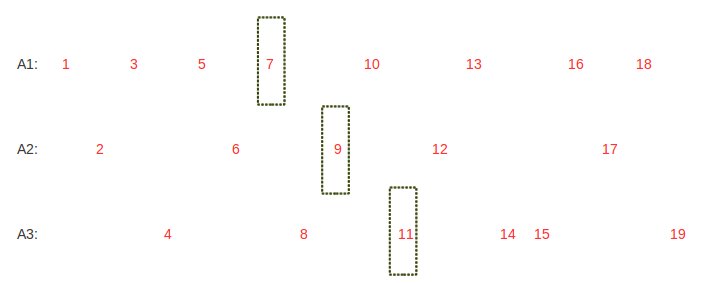
1. 输入多个有序数组查找其中位数m
2. 计算个数组的中位数，并计算其最小者min与最大者max，则m必满足min<=m<=max
3. 计算min之前所有的数字lTripCount（在所有数组）及max之后所有的数字rTripCount，如果二者相同，则min与max都为中位数，递归结束
4. 计算所有数组中属于区间[min,max]的中位数（小于min或大于max的全被舍弃）m1,如果m1满足lTripCount==rTripCount，则为中位数，递归结束
5. 如果lTripCount<rTripCount，则中位数m一定属于区间[min,m1]
6. 如果lTripCount>rTripCount，则中位数m一定属于区间(m1,max]
7. 递归上述过程
8. 若某个递归后发现各数组剩余的数据小于某个阀值，则考虑一起计算，从而终止递归

该算法的主要思想是利用子数组的中位数逼近全数组的中位数，具体过程如下：

要排序的全数组：

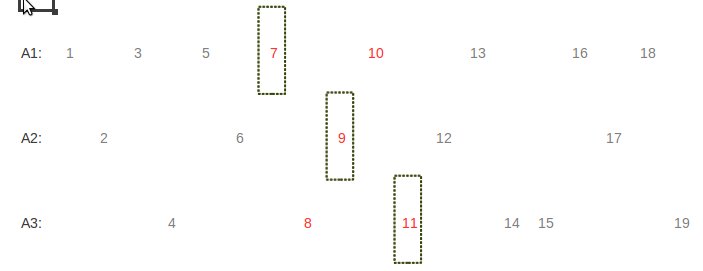


其中10便是要求解的中位数，数组A被划分为下面三个子数组：



计算步骤：

1. 分别计算A1,A2，A3的中位数,分别为7,9,11
2. 根据关联性，则要寻找的中位数（10）必定满足7<=m<=11
3. 从A1,A2，A3中去除小于7,大于11的数，此时各子数组的数据情况如下：



1. 继续计算各子数组的中位数，可的A1：10,A2：9,A3：11,因为此时各数组只剩一个元素，故根据假设可在一起计算其中位数10
2. 以10为中心计算lTripCount、rTripCount，经过计算二者相同，根据确定性可断定10为最终的中位数;如果lTreipCount与rTripCount不同，则可推导中位数会落在哪个区间（参考上面的算法描述），从而递归上面的过程

因为每次都根据最小中位数与最大中位数过滤，故每次计算可大约砍掉2/3的数据，算法大约以3n的速度收敛。

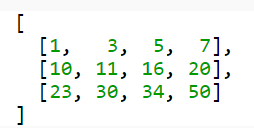
上述算法可能会产生一个无法收敛的特殊场景：如果某个计算过程出现最小中位数为当前有效数据的最小值，而最大中位数为最大值，则无法过滤掉更多的数据，解决该问题的办法是根据不变性，同时删除最小中位数与最大中位数。

总结排序（或部分排序）矩阵上的搜索问题

* **Search a 2D Matrix(LeetCode原题)**

[问题描述]

整个矩阵按行完全排好序，在矩阵中搜索是否含有某个值，含有返回true,否则返回false.



[思路]

由于整个矩阵完全排好序，其实可以直接把这个二维数组看成一个M\*N大小的一维数组，这样一次二分搜索就可以了。代码更加简洁。时间复杂度是log(M \* N) = logM + logN。

[代码实现]

public boolean find**(**int**[][]** matrix**,** int target**)** **{**

int numRows **=** matrix**.**length**;**

**if** **(**numRows **==** 0**)** **return** **false;**

int numCols **=** matrix**[**0**].**length**;**

int low **=** 0**,** high **=** numRows **\*** numCols **-** 1**;**

int mid**,** col**,** row**;**

**while** **(**low **<=** high**)** **{**

mid **=** low **+** **(**high **-** low**)** **/** 2**;**

col **=** mid **%** numCols**;**

row **=** mid **/** numCols**;**

**if** **(**matrix**[**row**][**col**]** **==** target**)** **{**

**return** **true;**

**}** **else** **if** **(**matrix**[**row**][**col**]** **>** target**)** **{**

high **=** mid **-** 1**;**

**}** **else** **{**

low **=** mid **+** 1**;**

**}**

**}**

**return** **false;**

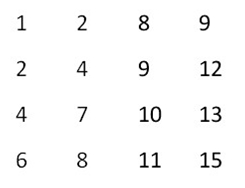
**}**

* **杨氏矩阵查找**

[问题描述]

在一个m行n列二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。（剑指offer3）

例如：下面的二维数组就是每行、每列都递增有序。如果在这个数组中查找数字6，则返回true；如果查找数字5，由于数组不含该数字，则返回false。



[解法一] ﻿﻿

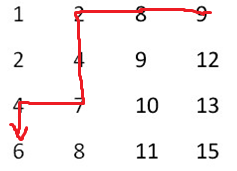
沿着对角线进行线性查询：因为每行和每列都排好序，以矩阵右上角为起点，沿着对角线扫描。

1）如果该元素等于目标元素，则找到目标；

2）如果该元素大于目标元素，则可以排除当前列，坐标往左移1个单位；

3）如果该元素小于目标元素，则可以排除当前行，坐标往下移1个单位。

例外一种查找策略是从左下角开始查找，与上述策略恰好相反。



算法实现——时间复杂度为O(m+n)，空间复杂度为O(1).

public boolean find**(**int**[][]** matrix**,** int elem**)** **{**

// start from the top right element

int row **=** 0**,** col **=** matrix**[**0**].**length **-** 1**;**

**while** **(**row **<** matrix**.**length **&&** col **>=** 0**)** **{**

**if** **(**matrix**[**row**][**col**]** **==** elem**)** **{**

**return** **true;**

**}** **else** **if** **(**matrix**[**row**][**col**]** **>** elem**)** **{**

col**--;**

**}** **else** **{**

row**++;**

**}**

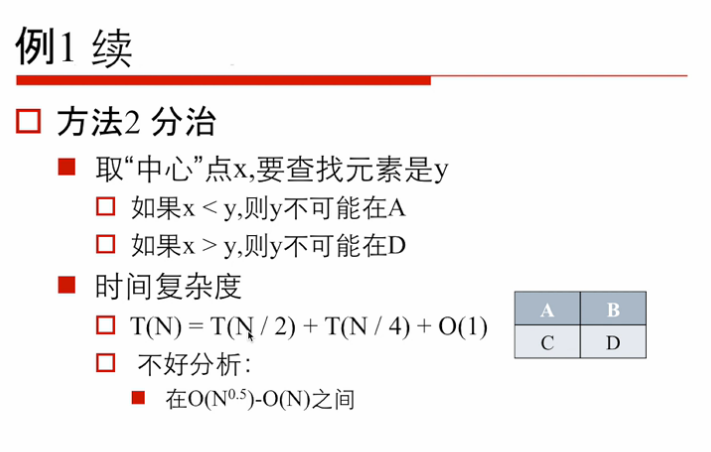
**}**

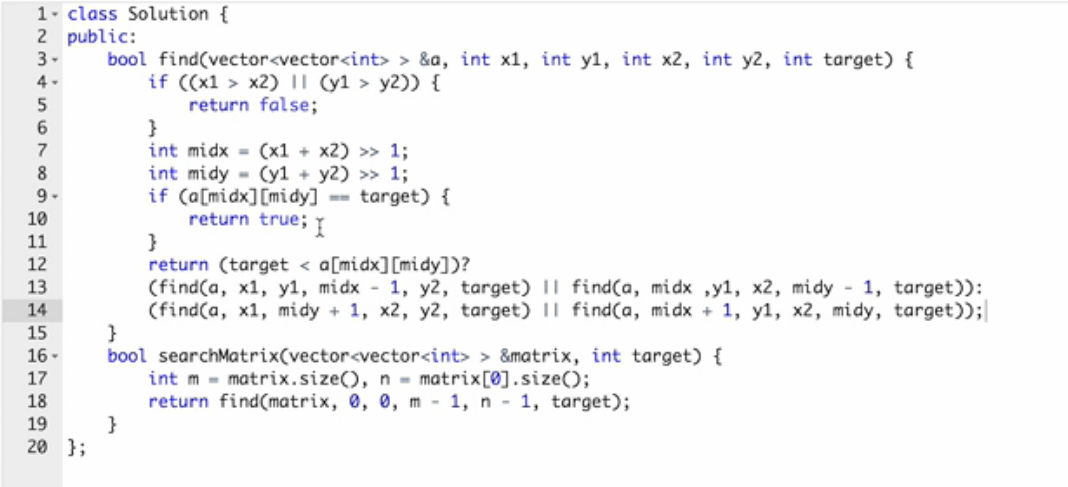
**return** **false;**

**}**

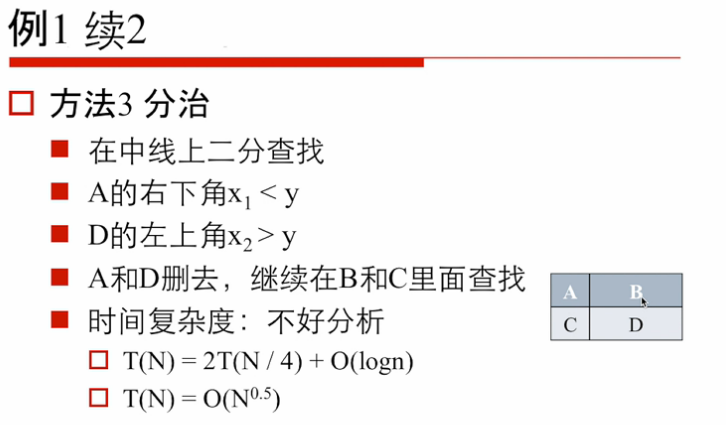
**[解法二]分治1：**

**将矩阵分为四块，midx = (x1+x2)>>2; midy = (y1+y2)>>2; y = a[midx][midy],判断中心点的值与target的大小。**





**[法三]分治法2：**



|  |  |
| --- | --- |
| **A** | **B** |
| **C** | **D** |

**找临界点indy的值，**

**若target=indy,直接返回；**

**若indy<target,扔掉以indy为右下角的矩阵A和以indy+1为左上角的矩阵D。**



* **杨氏矩阵查找一题的扩展**（Google面试题）

[问题描述]

给定 n×n 的实数矩阵，每行和每列都是递增的，求这 n^2 个数的中位数.

[思路1]

利用Partition算法。经典的Partition算法可以在线性时间内找到一维数组内的第K个元素。如果这里忽略已有的排序性质，把这个矩阵当成一个乱序的1维数组，那么直接利用Partition算法可以得到O(M\*N)的解。

[思路2]：利用部分排序的性质和最小堆

如果使用最小堆，如何使得插入的次数最少呢？换句话说，我们应该只插入需要插入的元素，或者说”刚好“只比当前元素大一点的元素。按照这个思路，我们仍然应该以第一个元素作为起点。

那么对于任意一个元素，“刚好”比它大一点的元素有哪些呢？

1.右边邻居；

2.下边邻居；

3.下边邻居的某些左侧元素。

前两种情况比较直观，第三种情况就只有靠标记了。具体做法如下：

初始化将第一个元素放入最小堆中。

1）每次弹出当前堆中的最小值，并将其右邻居和下邻居插入堆中，并标额外记右邻居和下邻居已经在堆中。如果之前某邻居已经被标记，则不再处理；（注意下邻居很可能之前是某元素的右邻居，要避免重复处理）

2）如果弹出的元素总数为K，则停止。

按照这个思路，处理K个元素最多也就将2K个元素插入堆中，堆的大小不超过2K，，所以算法复杂度是O(K\*logK)。

下面代码实现是google 一道面试题“Given a N\*N Matrix. All rows are sorted, and all columns are sorted. Find the Kth Largest element of the matrix.”。与这道题本质上一样的。解法没有进行额外标记已经在堆中的数据。

[代码实现]

#include <vector>

#include <algorithm>

#include <functional>

**using** **namespace** std**;**

struct Entry **{**

int value**;**

int x**;**

int y**;**

bool **operator** **<** **(**const Entry**&** other**)** **{**

**return** **this->**value **>** other**.**value**;**

**}**

**};**

bool getKthNumber**(**int**\*** matrix**,** int row**,** int col**,** int k**,** int**\*** result**){**

**if(**matrix **==** **NULL** **||** row **<=** 0 **||** col **<=** 0 **||** result **==** **NULL)**

**return** **false;**

**if(**k **<=** 0 **||** k **>** row **\*** col**)**

**return** **false;**

vector**<**Entry**>** minHeap**;**

Entry first **=** **{**matrix**[**0**],** 0**,** 0**};**

minHeap**.**push\_back**(**first**);**

make\_heap**(**minHeap**.**begin**(),** minHeap**.**end**());**

**for(**int i **=** 0**;** i **<** k**;** **++**i**){**

first **=** minHeap**[**0**];**

int x **=** first**.**x**;**

int y **=** first**.**y**;**

**if(**first**.**y **==** 0 **&&** first**.**x **<** row **-** 1**){**

Entry next **=** **{**matrix**[(**x **+** 1**)** **\*** col**],** x **+** 1**,** y**};**

minHeap**.**push\_back**(**next**);**

push\_heap**(**minHeap**.**begin**(),** minHeap**.**end**());**

**}**

**if(**first**.**y **<** col **-** 1**){**

Entry next **=** **{**matrix**[**x **\*** col **+** y **+** 1**],** x**,** y **+** 1**};**

minHeap**.**push\_back**(**next**);**

push\_heap**(**minHeap**.**begin**(),** minHeap**.**end**());**

**}**

pop\_heap**(**minHeap**.**begin**(),** minHeap**.**end**());**

minHeap**.**pop\_back**();**

**}**

**\***result **=** first**.**value**;**

**return** **true;**

**}**

[思路3]：O(M+N)算法

<http://www.cse.yorku.ca/~andy/pubs/X+Y.pdf>

[思路4]：Frederickson和Johnson实现的O(K)算法

Greg N. Frederickson and Donald B. Johnson. Generalized Selection and Ranking: Sorted Matrices. SIAM J. Comput. 13, pp. 14-30.

<http://epubs.siam.org/sicomp/resource/1/smjcat/v13/i1/p14_s1?isAuthorized=no>

算法选择需要看实际情况，取决于K和M，N之间的大小关系。后两种思路太复杂，我觉得前两种思路作为面试题的解法足够了。

另外可以看看这个网页，<http://zhiqiang.org/blog/science/computer-science/median-algorithm-of-ordered-matrix.html>。

* **冰山查询Iceberg query**

在数据仓库领域有一个概念叫Iceberg query，中文一般翻译为“冰山查询”。冰山查询在一个属性或属性集上计算一个聚集函数，以找出大于某个指定阈值的聚集值。

以销售数据为例，你想产生这样的一个顾客－商品对的列表，这些顾客购买商品的数量达到3件或更多。这可以用下面的冰山查询表示：

Select P.cust\_ID, P.item\_ID, SUM(P.qty)

From Purchase P

Group by P.cust\_ID, P.item\_ID

Having SUM(P.qty)>=3

这种在给出大量输入数据元组的情况下，使用having字句中的阈值来进行过滤的查询方法就叫做冰山查询。输出结果可以看作“冰山顶”，而“冰山”是输入数据。

这种冰山查询在数据仓库的数据概况分析阶段、数据质量检查阶段和数据挖掘的购物篮分析中都经常使用。而且，冰山查询也是面试中出现频率非常高的一道题，经常用来检测SQL能力。