**二分查找（binary-search）**，主要是解决在“一堆数中找出指定的数”这类问题。在大类中属于“分治法”，分治法基本都可以用递归来实现。所有的递归都可以自行定义stack来解递归，所以二分查找可以不用递归实现，而且它的非递归实现甚至可以不用栈，因为二分的递归其实是尾递归，它不关心递归前的所有信息。

**时间复杂度为O(log2n)。**（n代表集合中元素的个数）

**空间复杂度为O(1)。**虽以递归形式定义，但是尾递归，可改写为循环。

简单查找每次猜测都只能排除一个数字。使用二分查找时，每次都排除一半的数字。一般而言，对于包含n个元素的列表，用二分查找最多需要log2n步，而简单查找最多需要n步。

说明：仅当列表是有序的时候，二分查找才管用。至于是顺序递增排列还是递减排列，数组中是否存在相同的元素都不要紧。不过一般情况，我们还是希望并假设数组是递增排列，数组中的元素互不相同。

#### 中值的计算

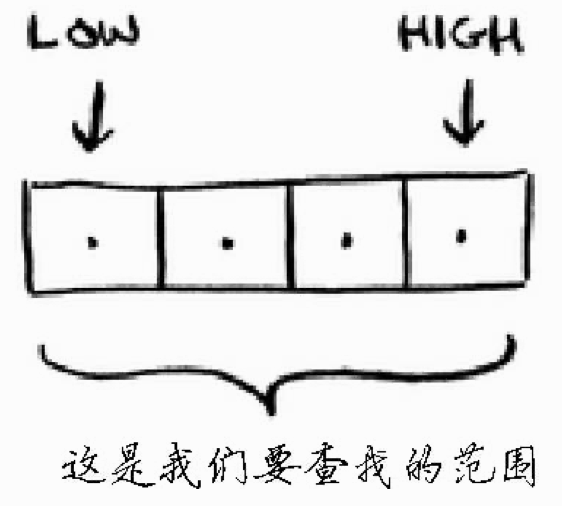
如何计算二分查找中的中值？

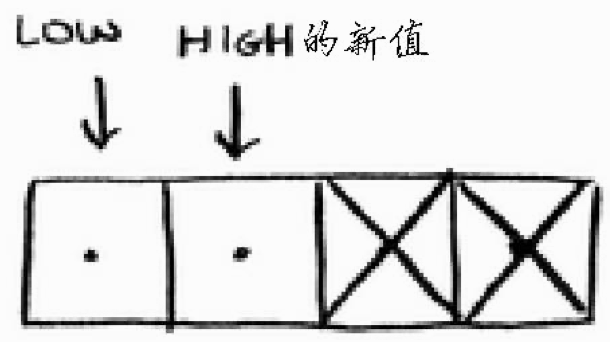
方法1：mid = (low + high) / 2

方法2：mid = low + (high - low) / 2

方法1的做法，在极端情况下，（low + high）存在着溢出的风险，进而得到错误的mid结果，导致程序错误。而方法2能够保证计算出来的mid，一定大于low，小于high，不存在溢出风险。

#### 代码实现





*/\*\**

*\* 递归二分查找  
 \*  
 \** ***@param*** *array 被查找的有序数组  
 \** ***@param*** *low 数组最低位的索引  
 \** ***@param*** *high 最高位的索引  
 \** ***@param*** *target 目标值  
 \** ***@return*** *值在数组中的位置，找不到返回-1  
 \*/*private static int binarySearch(int[] array, int low, int high, int target) {  
 if (low > high) {  
 return -1;  
 }  
 //中值的计算  
 int mid = low + (high - low) / 2;  
  
 //1.目标值小于中值，在中值前查找  
 if (array[mid] > target) {  
 return *binarySearch*(array, low, mid - 1, target);  
 }  
 //2.目标值大于中值，在中值后面查找  
 if (array[mid] < target) {  
 return *binarySearch*(array, mid + 1, high, target);  
 }  
 //3.目标值等于中值直接返回  
 return mid;  
}  
  
*/\*\*  
 \* 循环二分查找  
 \*  
 \** ***@param*** *array 被查找的有序数组  
 \** ***@param*** *target 目标值  
 \** ***@return*** *\*/*public static int binarySearchWithoutRecursion(int[] array, int target) {  
 int low = 0;  
 int high = array.length - 1;  
 while (low <= high) {  
 //中值的计算  
 int mid = low + (high - low) / 2;  
 //1.目标值小于中值，在中值前面查找  
 if (array[mid] > target) {  
 high = mid - 1;  
 } else if (array[mid] < target) { //2.目标值大于中值，在中值后面查找  
 low = mid + 1;  
 } else { //3.目标值等于中值  
 return mid;  
 }  
 }  
 return -1;  
}

#### 二分查找的缺陷

O(log2n)的时间复杂度让它成为十分高效的算法。不过缺点也很明显，在它的限定之上：必须有序，我们很难保证数组都是有序的。当然可以在构建数组的时候进行排序，可是又落到了第二个瓶颈上：它必须是数组。

数组读取效率是O(1)，可是它的插入和删除某个元素的效率确实O(n)。因而导致构建有序数组变成低效的事情。

解决这些问题的方法应该是使用二叉查找树，最好自然是自平衡二叉查找树。既能高效地（O(nlog2n)）构建有序元素集合，又能如同二分查找一样快速地（O(log2n)）搜寻目标数。

#### 用二分查找法找寻边界值

#### 用二分查找法找寻区域

#### 二分查找法在数据库内核中的实现

在数据库内核实现中，二分查找时一个非常重要的逻辑，几乎99%以上的SQL语句（所有索引上的范围扫描、等值查询、Unique查询等），都会使用到二分查找进行数据的定位。

考虑一个数据库表t1(a int primary key, b int)，表上的b字段有一个B+树索引，表中记录的b字段取值，就是[1,2,2,3,4,4,4,5,6,7,7]序列。此时，给定以下的两条查询语句，就是使用到了不同的二分查找逻辑：

SQL1：select \* from t1 where b>4;

SQL2：select \* from t1 where b>=4;

针对SQL1，索引的二分查找，就需要跳过所有的4，从最后一个4之后开始返回所有记录；针对SQL2，二分查找就需要定位到一个4，然后顺序读取所有记录。

除此之外，针对数据库中其他的查询逻辑，二分查找还需要附带更多的功能，例如：

SQL3：select \* from t1 where b<2;

SQL4：select \* from t1 where b<=2;

由于数据库索引同时支持反向扫描，因此SQL3、SQL4的语句，都可以使用索引反向扫描。反向扫描时，SQL3需要定位到索引中的第一个2；而SQL4，则需要定位到索引的最后一个2，然后开始反向满足查询条件的索引记录。

InnoDB针对不同的SQL语句，总结出四种不同的Search Mode，分别为：

#define PAGE\_CUR\_G 1 >查询

#define PAGE\_CUR\_GE 2 >=，=查询

#define PAGE\_CUR\_L 3 <查询

#define PAGE\_CUR\_LE 4 <=查询

然后根据这四种不同的Search Mode，在二分查找碰到相同键值时进行调整。例如，若Search Mode为PAGE\_CUR\_G或PAGE\_CUR\_GE，则移动low至mid，继续进行二分查找；若Search Mode为PAGE\_CUR\_LE或PAGE\_CUR\_L，则移动high至mid。继续进行二分查找。

TNT引擎，采用了与InnoDB不同的方案，但是也实现了相同的功能。TNT引擎针对相同键值的调整总结为下图：

/\* 操作符 includeKey forward compare result: 1 0 -1 \*/

=============================================================================

>= 1 1 | 1 -1 -1

= 1 1 | 1 -1 -1

> 0 1 | 1 1 -1

< 0 0 | 1 -1 -1

<= 1 0 | 1 1 -1

=============================================================================

参考：

<https://www.cnblogs.com/ider/archive/2012/04/01/binary_search.html>

<http://hedengcheng.com/?p=595>