折半查找：

给定一个整数 X 和整数 $A_0, A_1, \cdots, A_{N-1}$，后者已经预先排序并在内存中，求下标 i 使得 $A_i = X$，如果 X 不在数据中，则返回 $i = -1$。

明显的解法是从左到右扫描数据，其运行花费线性时间。然而，这个算法没有用到该表已经排序的事实，这就使得算法很可能不是最好的。一个好的策略是验证 X 是否是居中的元素。如果是，则答案就找到了。如果 X 小于居中元素，那么我们可以应用同样的策略于居中元素左边已排序的子序列；同理，如果 X 大于居中元素，那么我们检查数据的右半部分。(同样，也存在可能会终止的情况。)图 2-9 列出了折半查找的程序(其答案为 mid)。图中的程序同样也反映了 Java 语言数组下标从 0 开始的惯例。

```
1       /**
2        * Performs the standard binary search.
3        * @return index where item is found, or -1 if not found.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       int binarySearch( AnyType [ ] a, AnyType x )
7       {
8           int low = 0, high = a.length - 1;
9
10          while( low <= high )
11          {
12              int mid = ( low + high ) / 2;
13
14              if( a[ mid ].compareTo( x ) < 0 )
15                  low = mid + 1;
16              else if( a[ mid ].compareTo( x ) > 0 )
17                  high = mid - 1;
18              else
19                  return mid;   // Found
20          }
21          return NOT_FOUND;     // NOT_FOUND is defined as -1
22      }
```

图 2-9 折半查找

显然，每次迭代在循环内的所有工作花费 $O(1)$，因此分析需要确定循环的次数。循环从 high − low = $N-1$ 开始并在 high − low ⩽ −1 结束。每次循环后 high − low 的值至少将该次循环前的值折半；于是，循环的次数最多为$\lceil \log(N-1) \rceil + 2$。(例如，若 high − low = 128，则在各次迭代后 high − low 的最大值是 64,32,16,8,4,2,1,0,−1。)因此，运行时间是 $O(\log N)$。与此等价，我们也可以写出运行时间的递推公式，不过，当我们理解实际在做什么以及为什么的原理时，这种强行写公式的做法通常没有必要。

折半查找可以看做是我们的第一个数据结构实现方法，它提供了在 $O(\log N)$时间内的 contains 操作，但是所有其他操作(特别是 insert 操作)均需要 $O(N)$时间。在数据是稳定(即不允许插入操作和删除操作)的应用中，这种操作可能是非常有用的。此时输入数据需要一次排序，但是此后的访问会很快。有个例子是一个程序，它需要保留(产生于化学和物理领域的)元素周期表的信息。这个表是相对稳定的，因为很少会加进新的元素。元素名可以始终是排序的。由于只有大约 110 种元素，因此找出一个元素最多需要访问 8 次。要是执行顺序查找就会需要多得