二分搜索

例如，给定有 n 个元素的序列，这些元素是有序的（假定为升序），从序列中查找元素 x。

用一维数组 S[]存储该有序序列，设变量 low 和 high 表示查找范围的下界和上界，middle 表示查找范围的中间位置，x 表示特定的查找元素。

1. 算法步骤

（1）初始化。令 low=0，即指向有序数组 S[]的第 1 个元素；high=n−1，即指向有序数组 S[]的最后一个元素。

（2）判定 low≤high 是否成立，如果成立，则转向步骤 3，否则算法结束。

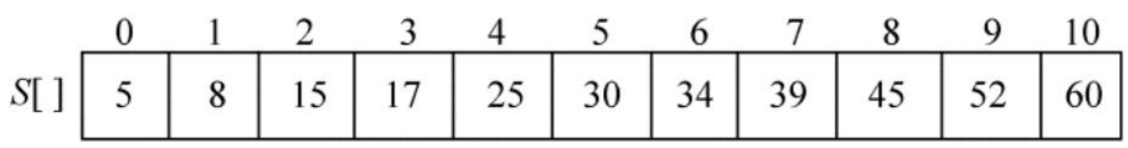
（3）middle=(low+high)/2，即指向查找范围的中间元素。如果数量较大，则为**避免 low+high 溢出**，可以采用 **middle=low+(high-low)/2。**

（4）判断 x 与 S[middle]的关系。如果 x=S[middle]，则搜索成功，算法结束；如果 x>S[middle]，则令 low=middle+1；否则令 high=middle−1，转向步骤 2。

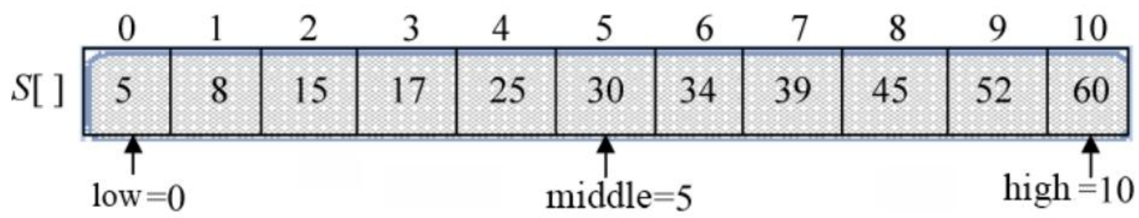
2. 图解

例如，在有序序列（5,8,15,17,25,30,34,39,45,52,60）中查找元素 17。

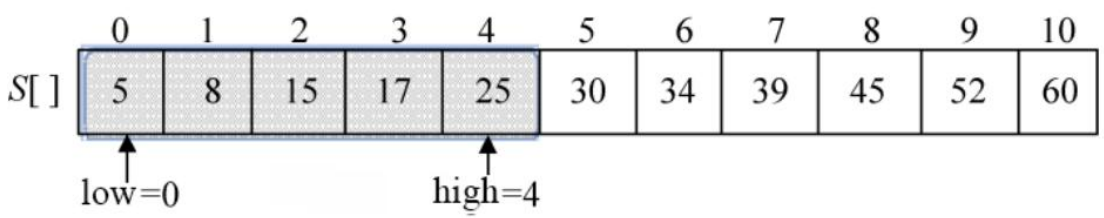
（1）数据结构。用一维数组 S[]存储该有序序列，x=17。



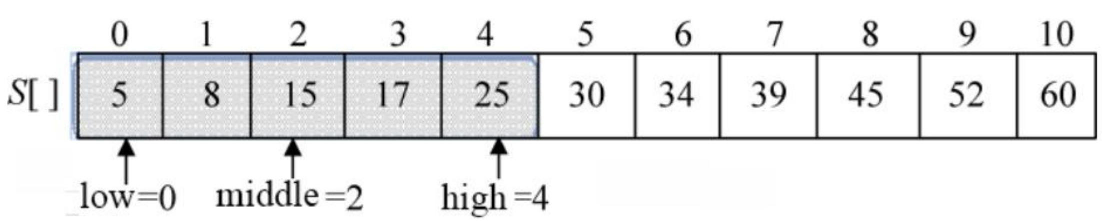
（2）初始化。low=0，high=10，计算 middle=(low+high)/2=5。



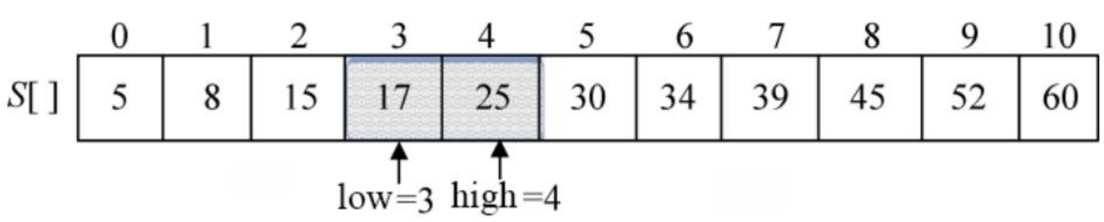
（3）将 x 与 S[middle]做比较。x=17，S[middle]=30，在序列的前半部分查找，令 high=middle−1，搜索的范围缩小到子问题 S[0…middle−1]。



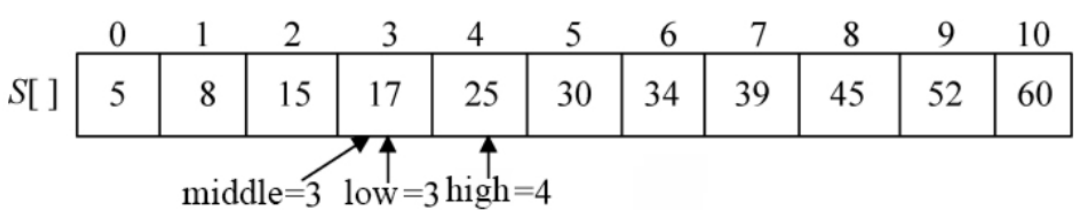
（4）计算 middle=(low+high)/2=2。



（5）将 x 与 S[middle]做比较。x=17，S[middle]=15，在序列的后半部分查找，令 low=middle+1，搜索的范围缩小到子问题 S[middle+1…high]。



（6）计算 middle=(low+high)/2=3。



（7）将 x 与 S[middle]做比较。x=S[middle]=17，查找成功，算法结束。

3. 算法实现

用 BinarySearch(int n, int s[], int x)函数实现二分查找算法，其中 n 为元素个数，s[]为有序数组，x 为待查找的元素。low 指向数组的第 1 个元素，high 指向数组的最后一个元素。如果 low≤high，middle=(low+high)/2，即指向查找范围的中间元素。如果 x=S[middle]，则搜索成功，算法结束；如果 x>S[middle]，则令 low=middle+1，在后半部分搜索；否则令 high=middle−1，在前半部分搜索。

（1）非递归算法。

```
int BinarySearch(int s[],int n,int x){//二分查找非递归算法
   int low=0,high=n-1;  //low 指向数组的第 1 个元素，high 指向数组的最后一个元素
   while(low<=high){
      int middle=(low+high)/2;  //middle 为查找范围的中间值
      if(x==s[middle])  //x 等于查找范围的中间值，算法结束
         return middle;
      else if(x>s[middle]) //x 大于查找范围的中间元素，在后半部分查找
            low=middle+1;
          else          //x 小于查找范围的中间元素，在前半部分查找
            high=middle-1;
    }
    return -1;
}
```

（2）递归算法。递归有自调用问题，增加两个参数 low 和 high 标记搜索范围的开始和结束。

```
int recursionBS(int s[],int x,int low,int high){ //二分查找递归算法
    //low 指向搜索区间的第 1 个元素，high 指向搜索区间的最后一个元素
    if(low>high)              //递归结束条件
       return -1;
    int middle=(low+high)/2; //计算 middle 值（查找范围的中间值）
    if(x==s[middle])        //x 等于 s[middle]，查找成功，算法结束
       return middle;
    else if(x<s[middle]) //x 小于 s[middle]，在前半部分查找
           return recursionBS(s,x,low,middle-1);
       else          //x 大于 s[middle]，在后半部分查找
           return recursionBS(s,x,middle+1,high);
}
```

4. 算法分析

1）时间复杂度

怎么计算二分查找算法的时间复杂度呢？如果用 T(n)来表示 n 个有序元素的二分查找算法的时间复杂度，那么结果如下。

- 当 n=1 时，需要一次做比较，T(n)=O(1)。
- 当 n>1 时，将待查找元素和中间位置元素做比较，需要 O(1)时间，如果比较不成功，那么需要在前半部分或后半部分搜索，问题的规模缩小了一半，时间复杂度变为 T(n/2)。

$$T(n)=\begin{cases} O(1), & n=1 \\ T(n/2)+O(1), & n>1 \end{cases}$$

- 当 n>1 时，可以递推求解如下：

$$\begin{aligned} T(n) &= T(n/2)+O(1) \\ &= T(n/2^2)+2O(1) \\ &= T(n/2^3)+3O(1) \\ &\cdots\cdots \\ &= T(n/2^x)+xO(1) \end{aligned}$$

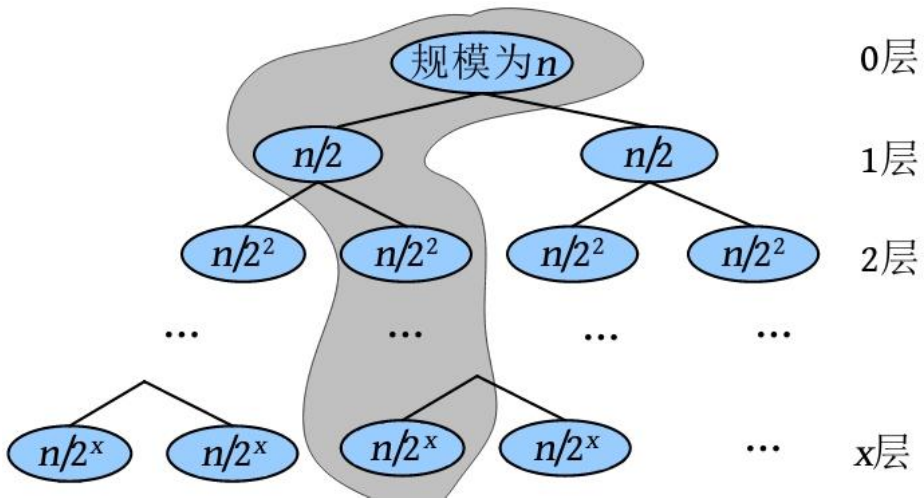
递推最终的规模为 1，令 **$n=2^x$，则 $x=\log n$。**

二分查找的非递归算法和递归算法查找的方法是一样的，时间复杂度相同，均为 O(logn)。

2）空间复杂度

在二分查找的**非递归算法**中，变量占用了一些辅助空间，这些辅助空间都是常数阶的，因此空间复杂度为 **O(1)。**

二分查找的**递归算法**，除了使用一些变量，还需要使**用栈来实现递归调用。**在递归算法中，每一次递归调用都需要一个栈空间存储，我们只需看看有多少次调用即可。假设原问题的规模为 n，首先第 1 次递归就分为两个规模为 n/2 的子问题，这两个子问题并不是每个都执行，只会执行其中之一，因为与中间值做比较后，要么在前半部分查找，要么在后半部分查找；然后把规模为 n/2 的子问题继续划分为两个规模为 n/4 的子问题，选择其一；继续分治下去，在最坏情况会分治到只剩下一个数值，那么算法执行的节点数就是从树根到叶子所经过的节点，每一层执行一个，直到最后一层，如下图所示。



递归调用最终的规模为 1，即 $n/2^x=1$，则 x=logn。假设阴影部分是搜索经过的路径，一共经过了 logn 个节点，也就是说递归调用了 logn 次。递归算法使用的栈空间为递归树的深度，因此二分查找**递归算法的空间复杂度为 O(logn)。**

在二分搜索中需要注意以下几个问题。

（1）**必须满足有序性。**

（2）**搜索范围。**初始时，需要指定搜索范围，如果不知道具体范围，则对**正数可以采用范围[0,inf]，对负数可以采用范围[-inf,inf]，**inf 为无穷大，**通常设定为 0x3f3f3f3f。**

（3）**二分搜索。**在一般情况下，mid=(l+r)/2 或 mid=(l+r)>>1。如果 l 和 r 特别大，则为了避免 l+r 溢出，可以采用 **mid=l+(r-l)/2。**对判断二分搜索结束的条件，以及判断 mid 可行时是在前半部分搜索，还是在后半部分搜索，需要具体问题具体分析。

（4）**答案是什么。**在减少搜索范围时，要特别**注意是否漏掉了 mid 点上的答案。**

二分搜索分为整数上的二分搜索和实数上的二分搜索，大致过程如下。

1. 整数上的二分搜索

整数上的二分搜索，因为缩小搜索范围时，**有可能 r=mid-1 或 l=mid+1，因此可以用 ans 记录可行解。**对是否需要减 1 或加 1，要根据具体问题来分析。

```
l=a; r=b; //初始搜索范围
while(l<=r){
     int mid=(l+r)/2;
     if(judge(mid)){
         ans=mid; //记录可行解
         r=mid-1;
     }
    else
        l=mid+1;
}
return ans;
```

2. 实数上的二分搜索

实数上的二分搜索**不可以直接比较大小**，可以**将 r-l>eps 作为循环条件**，eps 为一个较小的数，例如 1e-7 等。为避免丢失可能解，缩小范围时 r=mid 或 l=mid，在循环结束时返回最后一个可行解。

```
l=a; r=b; //初始搜索范围
while(r-l>eps){//判断差值
    double mid=(l+r)/2;
    if(judge(mid))
        l=mid;  //l 记录了可行解，在循环结束时返回答案 l
    else
        r=mid;
}
return l;
```

还**可以运行固定的次数**，例如运行 100 次，可达 10^{-30} 精度，在一般情况下都可以解决问题。

```
l=a; r=b;
for(int i=0;i<100;i++){//运行 100 次
    double mid=(l+r)/2;
    if(judge(mid))
        l=mid;
    else
        r=mid;
}
return l;
```