数据结构与算法（Python版）

二分查找算法及分析

陈斌 北京大学 gischen@pku.edu.cn

二分查找

- 那么对于有序表，有没有更好更快的查找算法?
- 在顺序查找中，如果第1个数据项不匹配查找项的话，那最多还有n-1个待比对的数据项
- 那么，有没有方法能利用有序表的特性，迅速缩小待比对数据项的范围呢?

#

二分查找

- **我们从列表中间开始比对！**

  如果列表中间的项匹配查找项，则查找结束

  如果不匹配，那么就有两种情况：

  - 列表中间项比查找项大，那么查找项只可能出现在前半部分
  - 列表中间项比查找项小，那么查找项只可能出现在后半部分

  无论如何，我们都会将比对范围缩小到原来的一半：n/2

- **继续采用上面的方法查找**

  每次都会将比对范围缩小一半

二分查找：代码

中间项比对

缩小比对范围

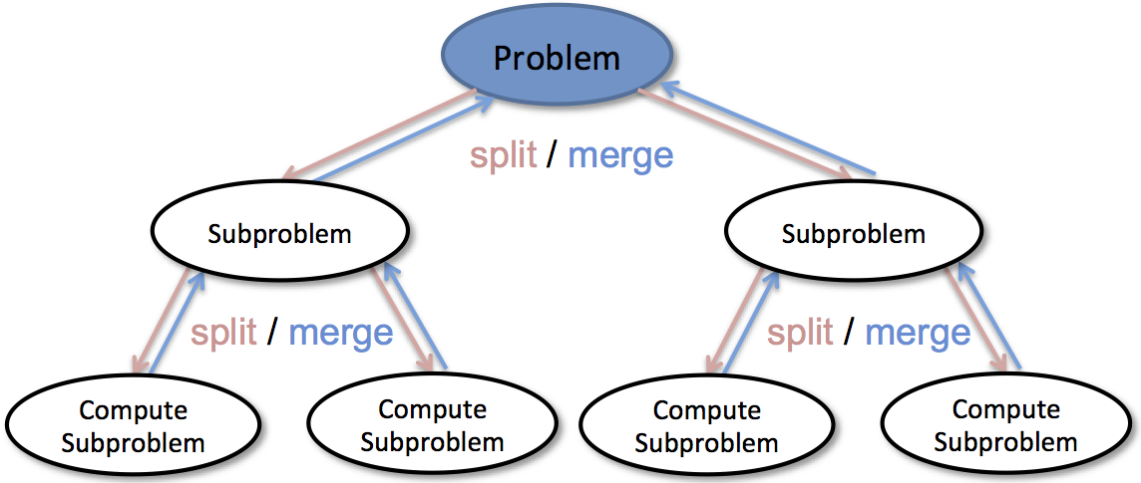
```
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

二分查找：分而治之

- 二分查找算法实际上体现了解决问题的典型策略：**分而治之**

  将问题分为若干更小规模的部分

  通过解决每一个小规模部分问题，并将结果汇总得到原问题的解



二分查找：分而治之

- 显然，**递归算法就是一种典型的分治策略**
  算法，**二分法也适合用递归算法来实现**

```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint]==item:
          return True
        else:
          if item<alist[midpoint]:
            return binarySearch(alist[:midpoint],item)
          else:
            return binarySearch(alist[midpoint+1:],item)
```

结束条件

调用自身

缩小规模

二分查找：算法分析

- 由于二分查找，每次比对都将下一步的比对范围缩小一半
- 每次比对后剩余数据项如下表所示：

Comparisons	Approximate Number of Items Left
1	n/2
2	n/4
3	n/8
…	
i	$n/2^i$

二分查找：算法分析

- 当比对次数足够多以后，比对范围内就会仅剩余1个数据项
- 无论这个数据项是否匹配查找项，比对最终都会结束，解下列方程：

  得到：$i=\log_2(n)$

$$\frac{n}{2^i}=1$$

- 所以二分法查找的算法复杂度是O(log n)

二分查找：进一步的考虑

- 虽然我们根据比对的次数，得出二分查找的复杂度O(log n)
- 但本算法中除了比对，还有一个因素需要注意到：

```
binarySearch(alist[:midpoint],item)
```

这个递归调用使用了列表切片，而切片操作的复杂度是O(k)，这样会使整个算法的时间复杂度稍有增加；

当然，我们采用切片是为了程序可读性更好，实际上也可以不切片，而只是传入起始和结束的索引值即可，这样就不会有切片的时间开销了。

二分查找：进一步的考虑

- 另外，虽然二分查找在时间复杂度上优于顺序查找
- 但也要考虑到**对数据项进行排序**的开销

  如果一次排序后可以进行多次查找，那么排序的开销就可以摊薄

  但如果数据集经常变动，查找次数相对较少，那么可能还是直接用无序表加上顺序查找来得经济

- 所以，在算法选择的问题上，光看时间复杂度的优劣是不够的，还需要考虑到**实际应用**的情况。