1.二分查找

①神奇的图书馆大爷：二分查找在生活中的应用

在正式学习二分搜索算法之前，我们先来分享一个现实生活中二分搜索的有趣案例：

有一天小明同学去图书馆借书。在看了一圈之后发现了好多喜欢的书，一口气借了20本书

在图书馆的门口，小明对图书进行消磁登记，但是在这个过程中，其中有一本书小明忘记了消磁，所以在过安检门的时候触发了警报

小明这个时候很慌，不得不将这20本书再重新一本一本的过一遍安检，用来判断到底是哪一本书没有消磁

这个时候图书馆看门的大爷看不下去了，直接接过小明手里的20本书，并进行了如下操作：

大爷首先将20本书分为两部分，每一部分10本书，并将两部分图书分别过一遍安检，

然后将触发警报的那一部分图书再次进行划分，同样划分成为两部分，每部分5本书，重复上面的操作

然后将触发警报的5本书再次划分成两部分，一部分2本书，另一部分3本书，然后再次将上述的两部分分别过安检

最终确定，两本书的那一部分中，有一本是没有消磁的，然后大爷直接将这两本书分别过安检，最终成功找到了那一本没有消磁的书……

小明被这一套神操作彻底惊呆了，因为他不知道，大爷使用的正是二分搜索的思想

根据上面的案例我们可以得到这样的一组数据：

如果按照小明的查找方式，相当于对整个20本书进行遍历，那么最坏情况需要查找20次才能够找到没有消磁的那一本书

但是按照看门大爷使用二分搜索的方式，即使是最坏情况下，也只需要查找4次，就能够找到未被消磁的那一本书

这，就是算法的力量！

②二分查找的基本理论

上面的案例，是二分搜索在显示生活中的一种应用，但是在真正的计算机编程中，二分搜索算法的实现还需要做如下的定义

1.二分搜索的定义

二分搜索算法的核心思想，就是对待搜索序列每一次查找后，如果没有找到查找目标，

那么就对待搜索序列进行折半，在其中的一半序列中继续查找，并根据查询结果继续搜索

直到找到目标元素在待搜索序列中的下标进行返回，或者确定待搜索序列中不存在目标元素为止

2.二分查找的查找内容

根据上面对二分搜索算法的定义我们可以知道：

在二分搜索算法中，待搜索序列和要查找的目标元素都是已经给定的，

所以二分搜索的查找目标是待搜索序列中目标元素所在的序列下标

也就是说：

　如果能够在待搜索序列中找到目标元素，那么二分搜索返回的就是这个元素在序列中的下标

　如果待搜索序列中不包含目标元素，也就是说在待搜索序列中查找不到目标元素，那么二分搜索返回-1

综上所述：二分查找的搜索目标并不是目标元素本身，而是目标元素在待搜索序列中的下标

3.二分查找的前提条件

计算机编程当中实现的二分搜索算法是具有一定前提条件的，二分搜索的前提条件就是：待搜索序列的所有元素必须是有序的

之所以要求整个序列是有序的，这和二分搜索的实现原理有关，我们将在后面讨论这个问题，现在请各位读者先记住这个结论

③二分查找的实现原理

二分搜索的实现原理和实现步骤可以描述如下（以升序有序序列为例）：

步骤1：首先根据搜索序列的起点下标和终点下标，计算出搜索序列的中间下标

步骤2：使用中间下标对应的序列元素与目标元素进行比较

步骤3：如果目标元素小于中间下标元素，则说明目标元素有可能存在于中间下标元素的左侧，将搜索序列结束下标重新标定义为中间下标-1

如果目标元素大于中间下标元素，则说明目标元素有可能存在于中间下标元素的右侧，将搜索序列开始下标重新定义为中间下标+1

步骤4：重复上述步骤1-3，直到中间下标元素等于目标元素时，说明目标元素在搜索序列中已找到，则返回此时的中间下标，就是目标元素在搜索序列中的下标

步骤5：如果在重复上述步骤的时候，出现搜索序列起点下标大于搜索序列终点下标的情况，则可以确定目标元素在搜索序列中不存在，此时返回-1

根据上面的步骤，就能够实现一个二分搜索算法。

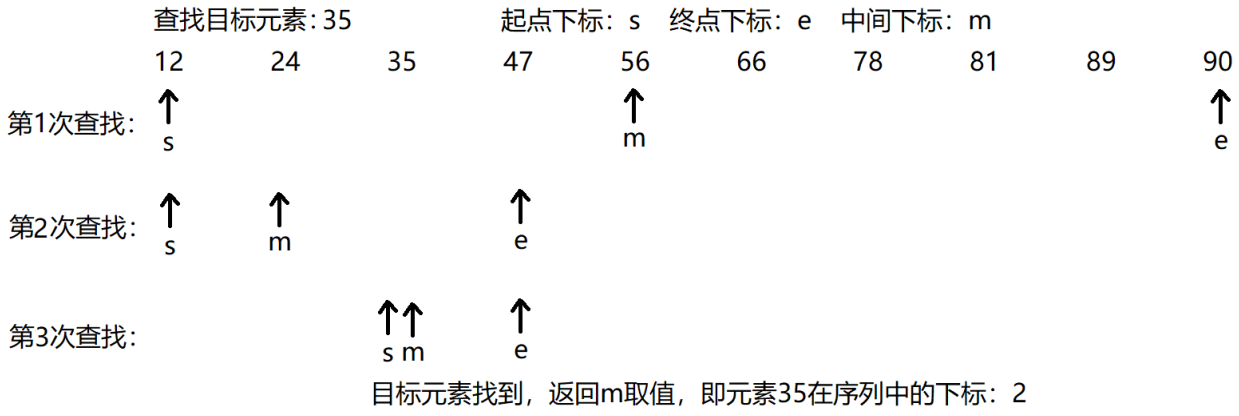
从上面的步骤描述中可以得知：我们在重新确定搜索序列的起点和终点的时候，是根据目标元素和中间下标元素的之间的大小关系确定的

并且我们根据这个大小关系能够推测出目标元素有可能存在于当前中间下标元素的左边还是右边。

这一推测的前提条件，就是整个搜索序列是有序的

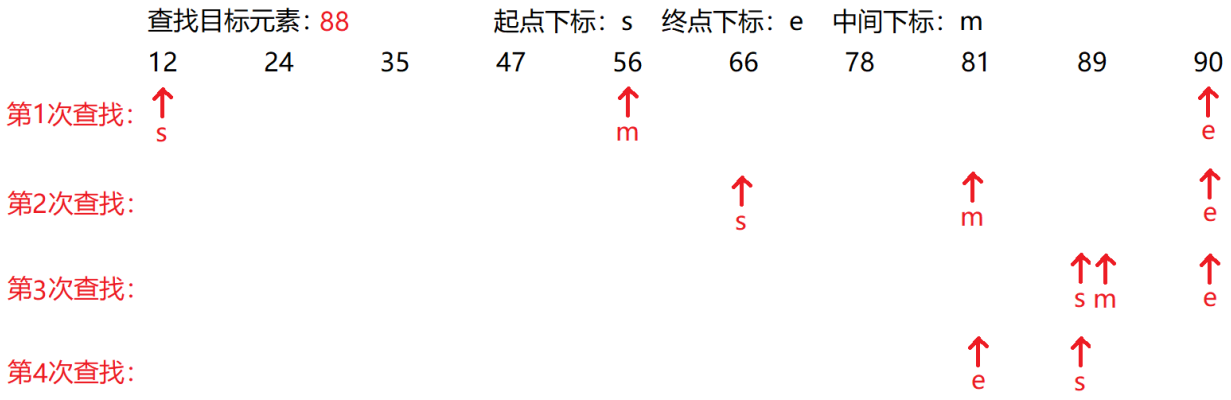
下面我们通过一组图片来观察和分析二分搜索算法的整体实现流程，并且从中找到代码层面的实现规律

1.能找到的情况



目标元素找到，返回m取值，即元素35在序列中的下标：2

2.找不到的情况



此时搜索序列的起点在终点的右侧说明我们查找了整个搜索序列也没有找到我们想要的元素，则此时我们返回-1，表示序列中没有这个目标元素

3.二分查找条件总结

通过上面的两张图示，我们可以发现在二分搜索中具有如下规律：

规律1：在搜索过程中，我们始终都是使用序列中间下标对应的元素和目标元素比较（array[m]和target比较），而在查找的的情况下，我们返回的是此时的中间下标（返回m的值）

规律2：中间下标的计算公式如下：中间下标m = (搜索序列起点下标s + 搜索序列终点下标e) / 2，并向下取整

规律3：如果目标元素取值小于中间下标元素取值，那么在下一次查询的时候，将搜索序列终点下标改变为中间下标-1

规律4：如果目标元素取值大于中间下标元素取值，那么在下一次查询的时候，将搜索序列起点下标改变为中间下标+1

规律5：关于二分搜索的结束条件，一句话总结就是：活要见人 || 死要见尸

活要见人：指的是如果在搜索过程中出现中间下标元素等于目标元素的情况（array[m] == target），则说明目标元素已找到，此时返回中间下标即可

死要见尸：指的是如果在搜索过程中出现搜索序列的起点下标或者终点下标在经过变化之后，出现起点下标在终点下标右侧的情况（s > e），则表示确定序列中不存在目标元素，返回-1结束

所以综上所述，二分搜索运行下去的条件就是：活不见人 && 死不见尸（array[m] != target && s <= e）

规律6：值得注意的是，在某些极端条件下，搜索序列的起点下标、终点下标和中间下标是会重合的，那么此时依然是有可能找到目标元素的，所以这种情况下，还是要继续搜索一次的

根据上述规律总结，我们可以得出如下二分搜索算法的实现代码：

```
/**
 * 非递归版本实现的二分搜索算法
 * @param array 搜索序列数组
 * @param target 查找的目标元素
 * @return 如果搜索序列中存在目标元素，则返回目标元素在数组中的下标；
 *         否则返回-1表示目标序列中不存在目标元素
 */
public int binarySearch(int[] array, int target) {

    //[1]定义搜索序列的起点下标、终点下标和中间下标
    int s = 0;  //起点下标
    int e = array.length - 1;  //终点下标

    int m = (s + e) / 2;  //中间下标

    //[2]使用do-while循环实现一个二分搜索算法。为什么使用do-while循环呢？
    do {

        //[4]只要还能够走到这个位置，说明又进行了一次二分搜索，
        //此时搜索序列的起点下标和终点下标已经发生变化，中间下标需要重新计算
        m = (s + e) / 2;

        //[3]通过比较array[m]的取值和target之间的大小关系，
        //来确定是否进行下一次搜索，以及下一次搜索的起点和终点下标

        //[3.1]如果target < array[m]，
        //则表示目标元素有可能存在于中间下标元素的左边，下一次的搜索范围：[s, m-1]
        if(target < array[m]) {
            e = m - 1;
        }
```

```
31
32          //[3.2]如果target > array[m]，
33          //则表示目标元素有可能存在于中间下标元素的右边，下一次的搜索范围：[m+1, e]
34          if(target > array[m]) {
35              s = m + 1;
36          }
37
38          //[3.3]如果target == array[m]，
39          //则表示目标元素在搜索序列中已经找到，直接返回目标元素在序列中的下标即可
40          if(target == array[m]) {
41              return m;
42          }
43
44      }while(s <= e);
45
46      //[5]如果上述循环能够正常结束，则说明没有走到循环中的return语句，
47      //也就是说没有在序列中找到目标元素，此时返回-1，表示序列中不存在目标元素
48      return -1;
49
50  }
```

2.递归

递归一直以来都是数据结构和算法课程中的重点，但是不巧的是，递归同样是数据结构和算法中的难点

“这世上本没有递归，代码写的多了，也就遇见了递归”

虽然鲁迅先生从未如是说过，但是是否能够熟练的掌握递归结构，并且使用其解决一些编程问题，确实能够成为衡量一个程序员水平的标杆

实际上从编程的角度来讲，递归并不像我们在Java中常见的ArrayList、HashMap那样，是一种具体的代码，递归更多的是一种代码结构，

换句话说：递归结构可以使用各种编程语言来实现，不仅仅局限于Java

通过这种代码结构，我们能够更加简单的解决同一类问题，而这一类问题有一个统一的名字：分治算法问题

关于分治算法的问题和整体结构，我们会在后面的内容中进行讲解，现在我们首先来了解一下递归操作的结构：

简而言之：在一个方法内部，再次调用这个方法本身作为解决整体问题的一部分，这就是递归

但是根据一个方法内部调用自己的各种情况，我们又能够分析出递归的标准写法，避免一种非常严重的错误——死递归

下面就让我们一起通过几个小案例，一起学习递归结构的具体内容

①递归的简单案例

1.小学数学题：从1加到10是多少?

可能有的同学在看到这个问题的时候，直接就会得出结论：使用一个循环实现，最终运算的结果是55

但是今天我们要求大家不使用循环的方式，来进行解题。

那么我们实际上可以分析出这样一个拆分问题的思路：

从1加到10的结果，就是10加上从1加到9的结果

而从1加到9的结果，就是9加上从1加到8的结果

而从1加到8的结果，就是8加上从1加到7的结果

……

最终，当仅剩下一个1的时候，这个问题将无法继续进行拆分，我们可以认为，从1加到1的结果就是1自己本身

从上述案例，我们发现这样一个规律：当我们想要去计算一个大问题的答案的时候，例如计算从1加到10的答案

那么此时我们可以将这个大问题拆分成为一个具有固定取值的数字和一个更小问题答案的合并，

例如将从1加到10变成10这个固定值加上从1加到9这个小问题的解的加和，并且进一步将从1加到9这个大问题拆分成为9这个固定值和从1加到8的和的加法运算……

这个时候我们似乎发现：1.每个小问题的解决过程和前一个大问题的解决过程是一致的，只是每一次的临界条件不太相同，2.并且每一次的问题规模都在缩小

如果你能够理解上面这段话的含义，那么说明你已经“读懂了”递归

下面，我们就用我们刚刚读懂的递归思想，去解决这个小学数学题

2.用递归的思路去解题

在使用递归解决这道题的时候，有三个问题我们必须明确：

1.什么时候将问题过程继续拆分

2.拆到什么时候是头，也就是什么时候能够得到小问题的明确答案

3.大问题的解和小问题的解之间具有什么关系

上面这三个问题我们可以在本体中做如下解答：

1.如果计算的数据范围不是从1加到1，那么就将问题继续拆分

2.只有在计算“从1加到1”这个问题的时候，才不需要继续拆分，这个问题有明确的解，就是1

3.每一个大问题的答案，都是由小问题的答案“堆砌”出来的，比如从1加到10的和，就是10加上从1加到9的和而计算出来的

在解决了上述三个问题之后，我们可以对这道题得到如下伪代码描述：

一个计算从1加到n的和的方法 (int n)：

如果：n > 1

从1加到n的和 = n + 从1加到n-1的和

如果：n == 1

这个问题有直接的解，直接返回1

根据这个思路，我们可以轻松的得到如下解题代码：

```
/**
 * 计算从1加到n的结果的方法
 * @param n 一个大于1的正整数，表示加和运算的终点
 * @return 1 + 2 + 3 + ... + n-1 + n的和
 */
public int add(int n) {

    if(n == 1) {
        //如果n的取值就是1，那么这个问题不需要继续拆分了，
        //这个问题就是最小问题，他具有明确的答案，是1
        return 1;
    }else {  //等价于n > 1的条件
        return n + add(n-1);  //如果n的取值是大于1的，那么问题继续拆分
    }

}
```

②递归算法的标准结构

如果你已经能够理解上面的代码，那么说明你已经在“读懂”递归算法的基础上，能够使用递归算法解决一些简单的问题了

下面我们就根据上面问题的解题思路，总结出递归的标准结构

1.递归调用

递归调用指的就是在同一个方法内部再次调用自己的过程

递归调用的作用实际上就是在将一个大问题进行解剖，将其划分成为诸多小问题，并将这些小问题一一解决

并且最终使用小问题的解，计算得到大问题的解

所以我们可以得到这样一条结论：在执行递归调用的时候，我们传递的参数的范围、规模等等，都比原来问题的范围、规模要小

就好比在上面的计算加和问题中，add(10) = 10 + add(9)的过程中，add(9)就是一次递归调用

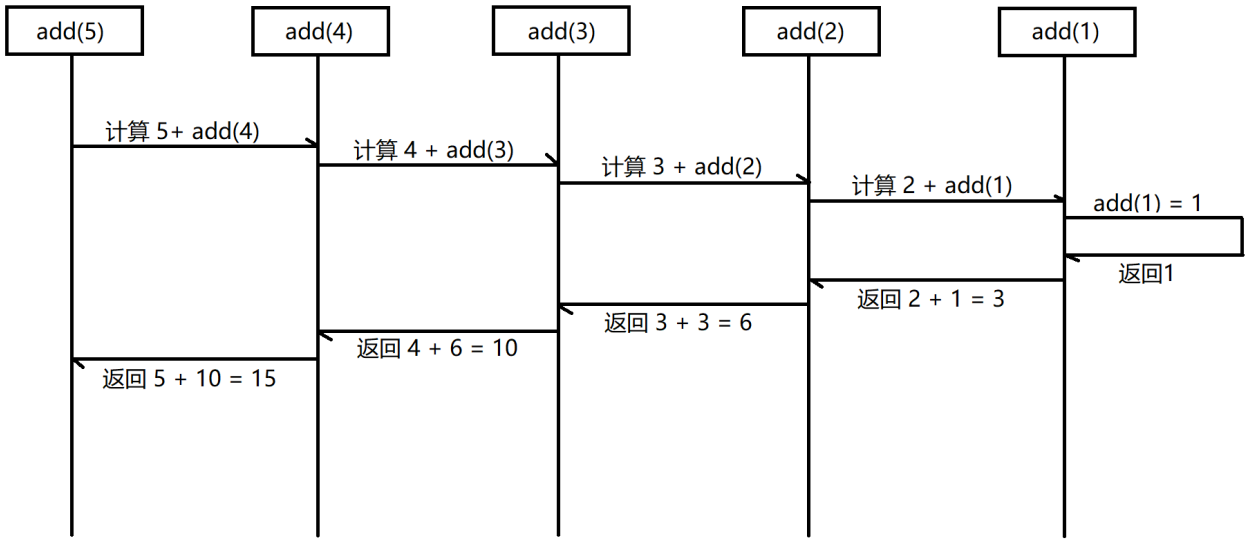
而add(9)和add(10)比起来，计算的范围就更小了

所以：递归调用的过程就是在将“大事化小”的过程

（如果你对JVM虚拟机的结构尚不熟悉，那么下面的内容我建议先不要看，否则可能会刷新你对递归的三观……）

在内存中，递归调用相当于在JVM的方法栈中重新调用了一个方法，但是在重新调用的方法尚未运行结束的时候，前面的方法是不会退出的

所以我们一从1加到5的过程为例，可以得到如下的方法调用时序图：



从上面的时序图中我们可以清楚的看到：

当计算add(4)的时候，内存add(5)方法并没有返回，而是在等待add(4)方法的运算结果

同理，在计算add(3)的时候，内存中的add(5)和add(4)方法同样都没有返回，因为add(4)方法在等待add(3)方法的返回值，而add(5)方法在等待add(4)方法的返回值……

这样一来，递归方法中的递归调用，就在内存中形成了一种层层调用，层层等待的结构

2.递归出口

通过观察之前的加和问题我们发现：问题不论如何拆分，当拆分到一定小的程度的时候，就不能够再向下拆分了

我们将着这个规模的问题，称之为“最小问题”，而在递归结构中的最小问题还具有另一重特性：具有明确答案的问题

从这个角度分析，当一个递归方法在遇见递归出口的时候，那么这个递归算法也就算是到头了，

剩下的事情就是在内存中，拿着最小问题的解，一层一层的滚雪球，逐步带入前面的问题中，计算并返回前面各个大问题的解

带入之前的案例，从1加到1就是一个具有明确答案的最小问题，它的答案就是1，然后在内存中，我们拿着这个1去前面的add(2)中计算2 + 1 = 3

再去前面的add(3)中计算3 + 3 = 6.....以此类推，直到算到10 + 45 = 55为止

所以，一切递归的终点，都在最小问题上，最小问题的解，是整个递归问题解的根源

所以：递归出口是递归中“小事化了”的部分

3.没有出口的递归：死递归

有一种程序员，天生脑洞清奇，代码不拘一格，甚至对自己电脑的用法都不拘一格，下面我们来看一个这样程序员写的案例：

```
/**
 * 咱也不知道这个方法到底要干啥，咱也不敢问
 * 但是在这个方法内部确实调用了自己
 * 那么我们认为这个方法确实是一个递归方法
 */
public void deathRecursion() {
    deathRecursion();
}
```

上面这个方法在运行之后，在卡顿几秒钟之后，会得到这样一个运行结果：

```
Exception in thread "main" java.lang.StackOverflowError
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
```

```
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    at com.oracle.recursion.DeathRecursion.deathRecursion(DeathRecursion.java
    然后是上面的重复，万里console一片红... ...
```

从输出内容上看来，这是一个“错误”，没错，这就是大名鼎鼎的栈内存溢出错误!

从上面的方法我们看出：这个递归方法并没有定义递归出口，也就是说在内存中，前面的方法在调用并等待后面的方法结果的过程中，永远得不到结果

得不到结果，那么前面的方法就永远不会结束，也就不会释放占用的方法栈内存空间

要知道，JVM中方法栈的内存空间是有限的，JVM不会无限制的为方法分配方法栈内存，那么当方法栈的内存空间不足，其他方法无法开辟空间运行的时候

JVM就会宕机（停止运行的意思），并且抛出这样一个错误——JVM不想和你说话，并向你抛出一个StackOverflowError

我们称这种没有递归出口，或者递归出口定义不正确、永远执行不到，并且会导致栈内存溢出错误的递归方法，为死递归——这个递归方法死在了JVM的栈内存当中……

那么从这一点看来，上面的案例方法唯一的作用就是：把JVM的栈内存调大，然后吃死CPU和计算机内存，让风扇满负荷运转，然后用电脑的出风口当电吹风使……

③递归算法的经典应用

1.求n的阶乘

n的阶乘是数学中一个非常经典的计算方式，在数学公式中，n的阶乘被表示为：n!

n的阶乘的运算方式和加和差不多，只不过是把加法换成了乘法，也就是：

n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1

并且在数学定义中，1! = 1

有了上面的加和运算的案例，那么再去写这个案例的代码，简直小菜一碟!

也就是说：

递归调用是：当n的取值大于1的时候，n的阶乘等于n乘以n-1的阶乘

递归出口是：当n的取值等于1的时候，1的阶乘就是1

实现代码如下：

```
/**
 * 计算n的阶乘的方法
 * @param n 一个大于等于1的正整数
```

```
 * @return n的阶乘
 */
public int factorial(int n) {

    if(n == 1) {
        //递归出口：当n等于1的时候，1的阶乘就是1
        return 1;
    }else {
        //递归调用：当n大于1的时候，n的阶乘等于n * (n-1)的阶乘
        return n * factorial(n-1);
    }

}
```

2.能生的兔子：斐波那契数列问题

在很久很久以前有这样一个问题：给你一对小兔子，这对小兔子从第3个月开始，每个月会再生一对兔子，

然后按照这个规律，生下来的小兔子从第三个月开始，同样每个月都会生一对小兔子，并且这些兔子的存活率相当高，可以达到100%，而且还不死

在不考虑近亲繁殖问题的情况下，问当第n个月的时候，你总共有多少对兔子

实际上这个诡异的问题，我们可以列一张数据表进行解决：

首先我们假设，每一对兔子，最开始的时候都是小兔子，过一个月之后变成大兔子并开始繁殖，并在一个月之后生产

而大兔子才会再去生小兔子，并且在中兔子变成大兔子的那个月就会再生一对小兔子，那么数据表格如下：

兔子类型\月份	1个月	2个月	3个月	4个月	5个月	6个月	7个月	8个月
小兔子	1	0	1	1	2	3	5	8
大兔子	0	1	1	2	3	5	8	13
兔子总对	1	1	2	3	5	8	13	21

实际上这个上生小兔子的问题，早在将近一千年前的意大利，就有一个闲极无聊的数学家进行了计算，这个人就是斐波那契

他根据上面的数据，最终推导出了这个数列的规律，并且将这个带有规律的数列命名为——斐波那契数列

斐波那契数列的规律就是：整个数列的前两个数是定死的就是1，并且从数列的第3位开始，每一位的取值都等于前两位的和

那么斐波那契数列的取值就是：1,1,2,3,5,8,13,21,34,55,89,144...正好是上面问题的解

根据斐波那契数列的规律，我们就可以入手去整理递归的思路了：

将斐波那契数列记做f(n)：

递归出口：当n等于2或者n等于1的时候，直接返回1，即f(1)或f(2)都等于1

递归调用：当n的取值大于2的时候，f(n) = f(n-1) + f(n-2)

好了，根据上述思路，我们可以得到如下实现代码：

```
/**
 * 计算斐波那契数列第n项取值的方法
 * @param n 斐波那契数列的第n项
 * @return 斐波那契数列第n项的取值
 */
public int fibonacci(int n) {

    if(n == 1 || n == 2) {
        //递归出口：如果所取项数为1或者2，那么取一个确定的值1
        return 1;
    }else {
        //递归调用：从第3项开始，等于前两项的和
        return fibonacci(n-1) + fibonacci(n-2);
    }

}
```

不得不说……如果这些兔子一只不死的话，不出几年功夫，就能生的全宇宙都是兔子了……

上述案例中最为重点的内容，并不是计算兔子的过程，而是从代码中体现出的一个信息：

当项数为1或者项数为2的时候，我们都可以返回一个固定的值为1；

当从第3项开始，我们需要通过2次递归，分别计算n-1和n-2项的和，

那么也就是说，在上述方法中，递归出口和递归调用都有两个！

所以：在同一个递归方法中，递归调用和递归出口的数量不一定只有一个

3.使用递归实现的二分搜索

通过上述两个问题我们发现：递归的思路，无非就是大事化小小事化了，将大问题逐层拆分，不断逼近具有直接答案的小问题

并且最重要的是，处理小问题的过程和处理大问题的过程是一致的，只不过问题的规模越来越小

那么，实际上之前我们学习的二分搜索算法，也是这样的一个思路：将搜索范围逐渐折半，直到找到目标元素在序列中的下标，或者确定序列中确实不存在这个目标元素为止

那么根据这个思路，我们可以整理出使用递归实现二分搜索算法的过程：

递归调用：

1.当目标元素小于中间下标元素的时候，将搜索范围的终点改为中间下标-1，重新在这个更小的范围内搜索

2.当目标元素大于中间下标元素的时候，将搜索范围的起点改为中间下标+1，重新在这个更小的范围内搜索

递归出口：

1.当找到目标元素在序列中的下标时，返回目标元素在序列中的下标，此时在小序列中找到的目标元素下标，就是目标元素在整个序列中的下标

也就是说，小问题的解，直接就是整个大问题的解

2.当起点下标大于终点下标的取值的时候，表示序列中没有目标元素，返回-1，此时小问题的解也同样直接就是整个大问题的解

根据上述思路，我们可以将二分搜索改变为如下结构：

```
/**
 * 使用递归实现的二分搜索算法
 * @param array 待搜索序列的数组
 * @param target 目标元素
 * @param start 搜索范围的起点
 * @param end 搜索范围的终点
 * @return 如果目标元素再搜索序列中存在，则返回目标元素在搜索序列中的下标；
 *         否则返回-1
 */
public int binarySearchRecursion(int[] array, int target, int start, int end)

    //[3]如果发现起点下标大于终点下标，说明目标元素不存在，直接返回-1
    if(start > end) {
        //递归出口2：当搜索范围起点大于终点的时候，返回-1表示目标元素不存在
        return -1;
```

```
    }

    //[1]通过搜索序列的起点下标和终点下标，计算中间下标
    int m = (start + end) / 2;

    //[2]将中间下标元素与目标元素进行大小比较，
    //并根据比较结果确定是继续搜索还是返回目标元素下标
    if(target < array[m]) {
        //递归调用1：在左半个区间内继续搜索
        return binarySearchRecursion(array, target, start, m-1);
    }else if(target > array[m]) {
        //递归调用2：在右半个区间内继续搜索
        return binarySearchRecursion(array, target, m+1, end);
    }else {  //等价于target == array[m]的情况
        //递归出口1：当在序列中查找到目标元素的情况下，
        //直接返回目标元素在序列中的下标
        return m;
    }

}
```

**④分治算法和递归

1.大事化小小事化了：什么是分治算法

分治算法所解决的问题，都有相同的特征：整个大问题的解是通过分别计算多个小问题的解，并通过将这些小问题的解进行合并而得到的

举个例子：菜市场做活动打折，拉哥要去把这一个月的要吃的菜都买回来，那么我就需要知道我这个月都要吃什么；

我想要知道这个月吃什么，那么就要知道这个月的每个星期我想要吃什么；

我想要知道这个月的每个星期吃什么，我就要知道每一个星期的每一天吃什么；

我想要知道每个星期的每一天吃什么，我就要知道每天的三顿饭分别吃什么；

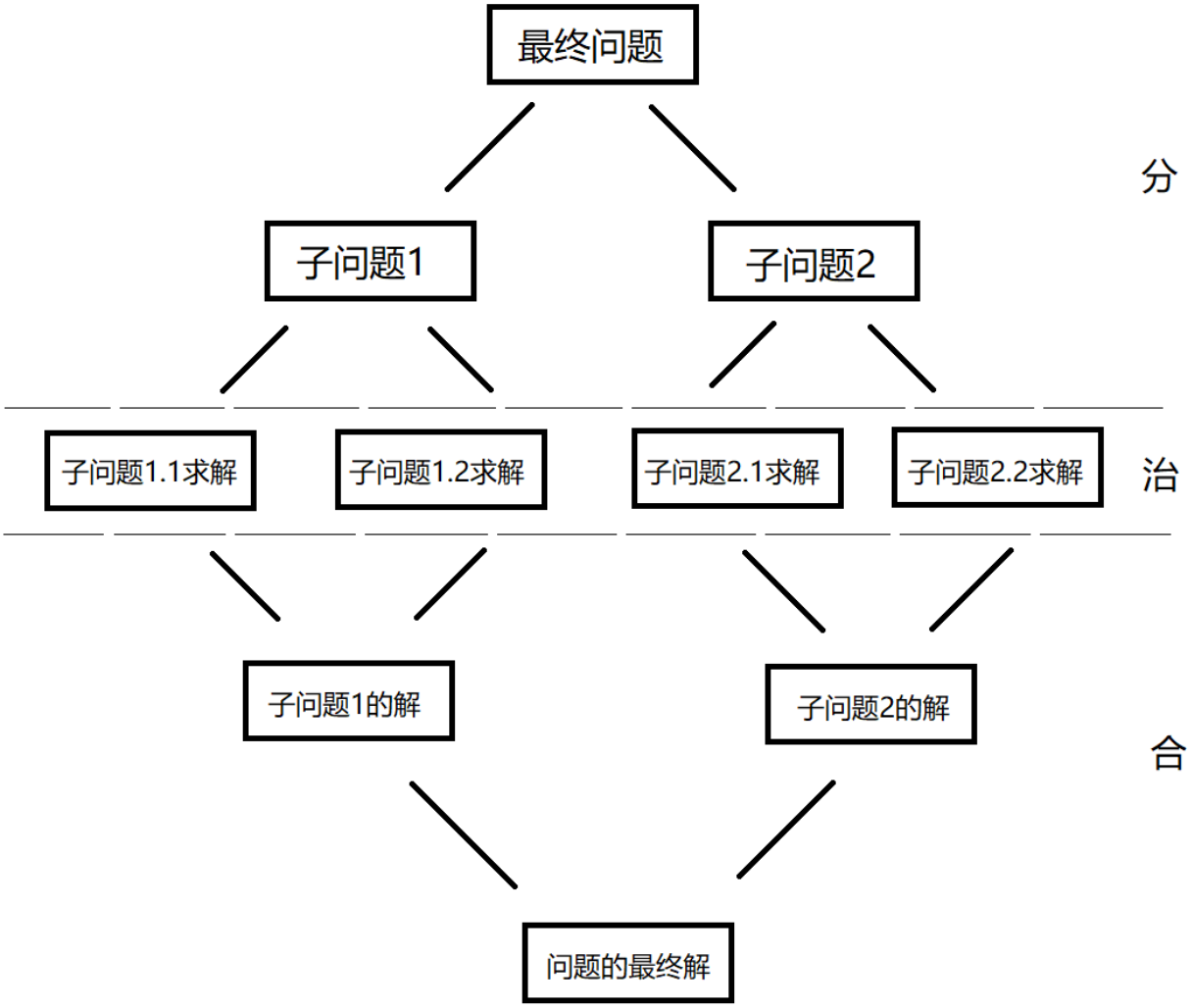
每天的每一餐就已经是最小单位了，不能再拆分了，所以我只要能够确定每一餐吃什么，我就能够不断向上递推

得到每一天、每一个星期、进而这个月我都要吃什么

所以：分治算法就是一种大事化小，小事化了的算法——用小问题的解递推合并计算出大问题的解

2.分治算法的经典步骤：分、治、合

分治算法也有其规范的实现步骤，按照这个步骤，我们可以得到如下图所示的流程：



通过上图，我们能够看到，分治算法的经典解题步骤就是：分、治、合

所谓分治合，解释如下：

分：将大问题逐渐拆分成为小问题的过程；只要一个小问题没有小到能够直接求解的程度，就要继续分解，这个过程称之为分

治：当一个问题被拆分的足够小的时候，这个小问题的解能够直接计算得到，那么计算最小问题解的过程，称之为治

合：求取最小问题解并不是我们的最终目标，我们的最终目标是要通过最小问题的解，逐渐合并并推导出整个大问题的解，这个过程称之为合

实际上，我们后面所学的归并排序、快速排序等等算法，都会用到这个思路，所以他们也都属于分治算法的实现

3.分治算法和递归的关系

递归结构是解决分治算法问题的常用工具，实际上几乎所有的分治算法问题都可以套用递归结构来实现

其中：

分治算法分的阶段，可以使用递归结构中的递归调用来实现，每一次的递归调用，都在将问题的规模不断变小

分治算法治的阶段，可以看做是递归结构中递归出口，递归出口的取值就是最小问题的直接解

分治算法合的阶段，需要在递归中使用各个递归调用结果值之间的关系来表示，因为递归方法的回退在内存中时自动的，

所以我们只要表示清楚小问题的解之间，是如何递推出大问题的解的关系即可

所以综上所述：递归结构是实现分治算法问题的有力工具，绝大部分分治算法问题都可以使用递归结构进行实现