算法的设计模式

算法，本质上是按照既定程序和步骤进行计算的流程。我们在这里对通用的算法设计的模式进行分类，每一种算法设计模式对应一种设计方法论，一个算法的实现，大多数情况下，需要集中若干种设计模式的结合。

常用的算法设计模式有：

1. 排序
2. 递归
3. 分而治之
4. 动态规划
5. 贪婪方法
6. 逐步改进
7. 排除法

下面，我们分别对每一种设计模式进行一个大略的说明。

**排序**

**不少问题，一旦使用了排序后，很可能答案就显现出来了。在算法面试中，如果题目中有数组，或者是一系列数据的集合，那么大家首先想到的是先把他们进行排序看看。举个例子，给定一个整形数组A，以及整数M, 要求判断A中是否存在这样的i , j 使得**

**M = A[i] + A[j].**

**对这个问题，大家如果先想到暴力枚举法的话，那表明前几节课大家能有所收获。如果我们使用暴力枚举法，遍历数组中所以可能的两两组合，看看是否存在满足条件的i, j, 那么时间复杂度是O(n^2). 如果我们先把A进行排序，假设是升序排序，一般来说，排序的时间复杂度是O(n\*lg n). 然后对于任意一个i, 是否存在一个j 使得 M = A[i] + A[j] 呢？既然已经排序了，那么，我们就可以在数组中进行折半查找，看看有没有j, 使得A[j] = M – A[i], 有的话，相应的i, j 就找到了。**

**假定数组A的内容为：**

**3， 1， 5， 6， 7 ， 4， 2， 8**

**M 取值为9**

**把A排序后的结果为：**

1. **2 3 4 5 6 7 8**

当 i 取 0 时 A[i] 等于 1， 然后我们在后面的元素中进行折半查找，最多查3次，我们可以找到元素A[7] = 8, 这样满足条件的i, j 就找到了。**由于折半查找的复杂度是lg n, i 的取值可以是从0到n, 因此总的时间复杂度就是O(n\*lgn), 这样，我们的算法效率就比使用暴力枚举法改进了一个层级。**

**递归**

递归就是一个函数在体内调用它自己，同时函数必须要有一个递归终结判断，要不然，递归会无止境的进行下去。当算法要处理的数据包含某种递归形式的时候，那么使用递归往往就是最合适的。例如字符串匹配是一个典型的递归问题。假设题目要求你写一个返回值为bool的函数，输入参数是一个字符串，以及一个匹配表达式，匹配表达式本身也是一个字符串，匹配表达式的形式如下：

1. x , x 是一个字符，字符串要匹配的话，他必须是”x”
2. . 只要字符串是1个字符长度就可以匹配
3. x\*, 输入字符串必须是字符x重复0次或多次才能匹配，如果x等于字符a, 那么输入字符串必须形如：a aa aaaa, aaaa…. 那么才能匹配
4. .\*, 无论输入字符串是任何形式，都可以匹配
5. r1r2 ,r1 r2 是正则表达式，字符串如果能分成两部分，第一部分匹配r1, 第二部分匹配r2, 那么字符串就满z匹配要求。

这个问题使用递归来处理就会很顺手，后面我们会对这个问题进行详细的分析。

**分而治之**

**分而治之是将一个问题分解成两个或多个规模更小的子问题，如果子问题容易解决，那么解决子问题后，把子问题的解决结果整合起来就得到原有问题的解。例如，如果要对数组进行排序，采用分而治之的办法，我们可以把数组分成两部分，然后分别将这两部分进行排序，最后把排序好的两部分合在一起就相当于把整个数组进行排序了。**

**我们看一个具体实例，假定数组内容为**

**3， 1， 5， 6， 7 ， 4， 2， 8**

**那么我们把数组分成两部分：**

**3 1 5 6 和 7 4 2 8**

**我们先看前半部分，前半部分又可以分成两部分**

**3 1 和 5 6**

**由于每部分只有两个元素，因此排序很简单，每部分排好序后，前半部变成 1 3， 后半部变成 5 6，然后把这两部分合并成一个有序部分 1 3 5 6，这样原数组的前半部分就排好序了，原数组后半部也可以依照相同的步骤进行排序，这样数组的前后两部分都已经排好序，最后把排好序的两部分合并就可以把整个数组排序了。**

**分而治之是一种很重要的算法设计模式，后面我们会有专门章节来进行详细的论述。**

**动态规划**

动态规划是算法面试中最常见，也是处理起来最棘手的问题。动态规划主要用来处理最优化的问题。最优化问题的一个特点是，如果要实现全局最优化，那么局部也必须是最优化的。动态规划和前面说的分而治之很类似，它也是讲一个大问题分解成若干个子问题，解决子问题，然后把子问题的解决结果整合起来得到大问题的解决方案。不同之处在于，动态规划分解子问题时，分解的力度没有分而治之那么大。举个例子，假定有两个字符串A(n) = a1 a2 a3 a4…. an. B(m) = b1 b2 b3 …. bm. 要求找到他们的最大共同子串(Longest common string)。假设如果A = “afghenj” B = “atfkuhndjop”, 那么A 和 B 的最大共同子串就是”afhnj”. 它的解法如下：

1. 如果 A 和 B 的最后一个字符是相同的，那么A 和 B 的最大共同子串是 A(n-1) 与 B(m-1)的最大共同子串加上最后一个字符, 如果A B 的最大共同子串用lcs(A, B) 表示， 那么lcs(A,B) = lcs(A(n-1), B(m-1)) + an
2. 如果最后一个字符不同，那么lcs(A, B) = max(lcs(A, B(m-1), lcs(A(n-1), B).

由于动态规划是面试算法题中出现最多的情况，我们在后面必将集中一大部分精力在这部分上。

**贪婪算法**

贪婪算法也是解决最优化问题的一种算法模式，它的思想是，如果要达到全局最优，那么我们就尝试使得每一个局部都最优，然后看看全局是否能达到最优状态。显然，很多情况下，即使是所以局部都达到了最优，全局也不可能达到最优。还有一种情况是，要使得每个局部达到最优的方法有若干种，其中只有某几种才可能使得局部最优的同时，达到全局最优。

我们看看运用贪婪算法的一个例子，假设有2n个城市排列在一条直线上。其中有一半的城市是黑色的，一半是白色的。我们希望找到一种把黑色城市和白色城市进行配对方法，如果把配对的城市进行连接的话，使得连接所需要的长度最短。例如如果我们把（0，4）进行连接，那么（1，2）自然就被连接了，因为1，2处于0和4之间，因此连接0和4就自动使得1和2进行连接。最简单的解决这个问题的贪婪算法是扫描每一个白城市，然后找到一个离该白城市最近的但却还没有匹配过的黑城市进行匹配。但是这种贪婪算法无法实现全局最优，例如，如果白色城市位于点0，3。黑色城市位于点2和5. 那么按照上面的贪婪算法，如果第一个被扫描到的白城市是3，那么配对的黑城市就是2，于是使得0就只能和5配对，于是总连接长度是5（0到5的距离）,然而最后的配对方式是0和2，3和5，这样总配对长度是4.

然而存在另一种贪婪算法能实现全局最优，假定W表示白城市的坐标数组，B表示黑城市的坐标数组，先将W和B进行排序，然后配对W[i]和B[i],可以证明，这种配对方法可以达到全局最优，在后续章节中，我们也会对贪婪算法进行详细的分析。

**逐步改进**

**逐步改进的思想是，当我们要寻找一个全局最优的方案时，我们先找到一个次优的方案，然后不断的去优化改进这个方案，使得它最终达到最优。**

**举个例子，有n个学生和n个教授，每个学生根据自身对教授的好感度进行排序，同理，每个教授分别对n个学生有自己的好感度排序。你需要实现一个算法，输入就是学生和教授的好感度列表，输出是一种一一配对，要求，输出的配对中不能出现下面的情况：**

**(s0, a0), 和 (s1,a1), 但是 学生s0对教授a1的好感度强于a0,并且教授a1对学生s0的好感度也强于学生s1.**

**如果运用逐步改进的思想来解决这个问题，我们可以这么做，每个学生根据自己心中对教授的好感度排名，对教授请求配对。接到请求的教授，根据他心中对学生的好感度排名，看看当前请求的学生，他的好感度是否高于上一位向他请求的学生，如果高于，那么就拒绝上一个学生，接受这个学生，如果低于，那么就拒绝当前的学生。如果学生被当前教授拒绝，那么他根据教授好感度排名，向下一位教授发出请求。如果一轮下来，有学生被拒绝，或有学生没有被接受，那么以相同的方式进行下一轮。反复进行，直到所有学生和教授都配对后，满足条件的匹配就生成了。要证明该算法的正确性并不容易，后面我们再对此问题做详细的分析。**

**逐步改进方法适用于探索性的算法，也就是当寻找最优的算法太难实现，或是时间复杂度过高，在工程上不现实。那么利用逐步改进法去获取次一级的优化结果，虽然逐步改进法得到的结果不一定是最优的，但它很接近全局最优，同时在工程上又容易实现。**

**排除法**

排除法的思想是，算法设计的核心不是直接去获取符合条件的结果，而是不断的去排除那些不满足条件的结果，当所有不能满足条件的结果都排除后，剩下的就是可以满足条件的结果了。

一个典型的使用排除法的算法是二分查找法，假定有一个已经排序好的数组:

1 2 3 4 5 6 7 8 9 10

如果要查询数字8是否在数组中，二分查找法的做法是先看数组中间元素，假定是5，由于数组是升序排序的，此时算法可以确切的排除，要查找的数字8不可能在5的左边，于是下一次查找就将5已经5以前的元素全部排除，只要把精力集中在5以后的元素即可。

以上几种算法设计模式几乎可满足于应对算法面试，当我们拿到题目，然后把上面的几种模式一一套入题目，看哪种模式能够产生突破口，甚至是直接得到解决方案。算法面试最棘手的情况是，面对题目没有一点思路或是觉得无处着手，如果把面试题比作一只狡猾的狐狸，那么上面这7种设计模式相当于一个笼子，把狐狸罩尽笼子里，即使它再狡猾，捉住它也不是什么难事了。