

# 五大算法思想：分治、动态规划、贪心、回溯和分支界定

2017 年 06 月 22 日 20:57:48 Zenhobby 阅读数：1614 标签： [算法](#) [更多](#)

个人分类： [数据结构和算法](#)

转载自： <http://blog.csdn.net/yapian8/article/details/28240973>

## 分治算法

### 一、基本概念

在计算机科学中，分治法是一种很重要的**算法**。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题.....直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换(快速傅立叶变换).....

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于  $n$  个元素的排序问题，当  $n=1$  时，不需任何计算。 $n=2$  时，只要作一次比较即可排好序。 $n=3$  时只要作 3 次比较即可，...。而当  $n$  较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。

---

### 二、基本思想及策略

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为  $n$  的问题，若该问题可以容易地解决（比如说规模  $n$  较小）则直接解决，否则将其分解为  $k$  个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

如果原问题可分割成  $k$  个子问题， $1 < k \leq n$ ，且这些子问题都可解并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

---

### 三、分治法适用的情况

分治法所能解决的问题一般具有以下几个特征：

- 1) 该问题的规模**缩小**到一定的程度**就可以容易地解决**
- 2) 该问题可以分解为若干个规模较小的相同问题，即该问题**具有最优子结构性**。
- 3) 利用该问题分解出的子问题的解**可以合并**为该问题的解；
- 4) 该问题所分解出的各个子问题是相互独立的，即**子问题之间不包含公共的子子问题**。

第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；

第二条特征是应用分治法的前提它也是大多数问题可以满足的，此特征反映了递归思想的应用；

**第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。**

第四条特征涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但**一般用动态规划法较好**。

---

#### 四、分治法的基本步骤

分治法在每一层递归上都有三个步骤：

- step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
- step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题
- step3 合并：将各个子问题的解合并为原问题的解。

它的一般的算法设计模式如下：

Divide-and-Conquer(P)

1. if  $|P| \leq n_0$
2. then return(ADHOC(P))
3. 将 P 分解为较小的子问题  $P_1, P_2, \dots, P_k$
4. for  $i \leftarrow 1$  to  $k$
5. do  $y_i \leftarrow \text{Divide-and-Conquer}(P_i)$   $\Delta$  递归解决  $P_i$
6.  $T \leftarrow \text{MERGE}(y_1, y_2, \dots, y_k)$   $\Delta$  合并子问题

## 7. return(T)

其中 $|P|$ 表示问题  $P$  的规模； $n_0$  为一阈值，表示当问题  $P$  的规模不超过  $n_0$  时，问题已容易直接解出，不必再继续分解。 $ADHOC(P)$  是该分治法中的基本子算法，用于直接解小规模的问题  $P$ 。因此，当  $P$  的规模不超过  $n_0$  时直接用算法  $ADHOC(P)$  求解。算法  $MERGE(y_1, y_2, \dots, y_k)$  是该分治法中的合并子算法，用于将  $P$  的子问题  $P_1, P_2, \dots, P_k$  的相应的解  $y_1, y_2, \dots, y_k$  合并为  $P$  的解。

---

## 五、分治法的复杂性分析

一个分治法将规模为  $n$  的问题分成  $k$  个规模为  $n/m$  的子问题去解。设分解阈值  $n_0=1$ ，且  $adhoc$  解规模为 1 的问题耗费 1 个单位时间。再设将原问题分解为  $k$  个子问题以及用  $merge$  将  $k$  个子问题的解合并为原问题的解需用  $f(n)$  个单位时间。用  $T(n)$  表示该分治法解规模为  $|P|=n$  的问题所需的计算时间，则有：

$$T(n) = k T(n/m) + f(n)$$

通过迭代法求得方程的解：

递归方程及其解只给出  $n$  等于  $m$  的方幂时  $T(n)$  的值，但是如果认为  $T(n)$  足够平滑，那么由  $n$  等于  $m$  的方幂时  $T(n)$  的值可以估计  $T(n)$  的增长速度。通常假定  $T(n)$  是单调上升的，从而当  $m_i \leq n < m_{i+1}$  时， $T(m_i) \leq T(n) < T(m_{i+1})$ 。

---

## 六、可使用分治法求解的一些经典问题

(1) 二分搜索

(2) 大整数乘法

(3) Strassen 矩阵乘法

(4) 棋盘覆盖

(5) 合并排序

(6) 快速排序

(7) 线性时间选择

(8) 最接近点对问题

(9) 循环赛日程表

(10) 汉诺塔

---

## 七、依据分治法设计程序时的思维过程

实际上就是类似于数学归纳法，找到解决本问题的求解方程式，然后根据方程式设计递归程序。

- 1、一定是先找到最小问题规模时的求解方法
- 2、然后考虑随着问题规模增大时的求解方法
- 3、找到求解的递归函数式后（各种规模或因子），设计递归程序即可。

## 动态规划算法

### 一、基本概念

动态规划过程是：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

### 二、基本思想与策略

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

---

### 三、适用的情况

能采用动态规划求解的问题的一般要具有 3 个性质：

(1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有**最优子结构**，即满足最优化原理。

(2) **无后效性**：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

(3) 有**重叠子问题**：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

---

### 四、求解的基本步骤

动态规划所处理的问题是一个**多阶段决策问题**，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一一定的模式，一般要经历以下几个步骤。

初始状态 → | 决策 1 | → | 决策 2 | → ... → | 决策 n | → 结束状态

图 1 动态规划决策过程示意图

(1)**划分阶段**：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)**确定状态和状态变量**：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3)**确定决策并写出状态转移方程**：因为决策和状态转移有着天然的联系，**状态转移就是根据上一阶段的状态和决策来导出本阶段的状态**。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，**根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程**。

(4)**寻找边界条件**：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的**阶段、状态和状态转移决策**确定了，就可以写出**状态转移方程（包括边界条件）**。

实际应用中可以按以下几个简化的步骤进行设计：

- (1) 分析最优解的性质，并刻画其结构特征。
- (2) 递归的定义最优解。
- (3) 以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值
- (4) 根据计算最优值时得到的信息，构造问题的最优解

---

## 五、算法实现的说明

动态规划的主要难点在于理论上的设计，也就是上面 4 个步骤的确定，一旦设计完成，实现部分就会非常简单。

使用动态规划求解问题，**最重要的就是确定动态规划三要素**：

- (1) **问题的阶段**
- (2) **每个阶段的状态**
- (3) **从前一个阶段转化到后一个阶段之间的递推关系**。



递推关系必须是从次小的问题开始到较大的问题之间的转化，从这个角度来说，动态规划往往可以用递归程序来实现，不过**因为递推可以充分利用前面保存的子问题的解来减少重**

复计算，所以对于大规模问题来说，有递归不可比拟的优势，这也是动态规划算法的核心之处。

确定了动态规划的这三要素，整个求解过程就可以用一个最优决策表来描述，最优决策表是一个二维表，其中行表示决策的阶段，列表示问题状态，表格需要填写的数据一般对应此问题的在某个阶段某个状态下的最优值（如最短路径，最长公共子序列，最大价值等），填表的过程就是根据递推关系，从 1 行 1 列开始，以行或者列优先的顺序，依次填写表格，最后根据整个表格的数据通过简单的取舍或者运算求得问题的最优解。

$$f(n,m)=\max\{f(n-1,m), f(n-1,m-w[n])+P(n,m)\}$$

## 六、动态规划算法基本框架

  
 代码  

```
1 for(j=1; j<=m; j=j+1) // 第一个阶段 2 xn[j] = 初始值; 3 4 for(i=n-1; i>=1; i=i-1) // 其他 n-1 个阶段 5 for(j=1; j>=f(i); j=j+1) // f(i) 与 i 有关的表达式 6 xi[j]=j=max ( 或 min ) {g(xi-1[j1:j2]), ....., g(xi-1[jk:j1+1])}; 8 9 t = g(x1[j1:j2]); // 由子问题的最优解求解整个问题的最优解的方案 10 11 print(x1[j1]); 12 13 for(i=2; i<=n-1; i=i+1) 15 { 17 t = t-xi-1[ji]; 18 19 for(j=1; j>=f(i); j=j+1) 21 if(t=xi[ji]) 23 break; 25 }
```

### 贪心算法

#### 一、基本概念：

所谓贪心算法是指，在对问题求解时，总是做出在**当前看来是最好的选择**。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的**局部最优解**。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，**贪心算法不是对所有问题都能得到整体最优解**，选择的贪心策略必须具备**无后效性**，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。

所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。

#### 二、贪心算法的基本思路：

1. 建立数学模型来描述问题。

- 2.把求解的问题分成若干个子问题。
- 3.对每一子问题求解，得到子问题的局部最优解。
- 4.把子问题的解局部最优解合成原来解问题的一个解。

### 三、贪心算法适用的问题

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

实际上，**贪心算法适用的情况很少**。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

### 四、贪心算法的实现框架

从问题的某一初始解出发；

while （能朝给定总目标前进一步）

{

    利用可行的决策，求出可行解的一个解元素；

}

由所有解元素组合成问题的一个可行解；

### 五、贪心策略的选择

因为用贪心算法只能通过解局部最优解的策略来达到全局最优解，因此，一定要注意判断问题是否适合采用贪心算法策略，找到的解是否一定是问题的最优解。

### 六、例题分析

下面是一个可以试用贪心算法解的题目，贪心解的确不错，可惜不是最优解。

[背包问题]有一个背包，背包容量是  $M=150$ 。有 7 个物品，物品可以分割成任意大小。

要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

物品 A B C D E F G

重量 35 30 60 50 40 10 25

价值 10 40 30 50 35 40 30

分析：

目标函数：  $\sum p_i$  最大

约束条件是装入的物品总重量不超过背包容量：  $\sum w_i \leq M$  ( $M=150$ )

（1）根据贪心的策略，每次挑选价值最大的物品装入背包，得到的结果是否最优？

（2）每次挑选所占重量最小的物品装入是否能得到最优解？

（3）每次选取单位重量价值最大的物品，成为解本题的策略。

值得注意的是，贪心算法并不是完全不可以使用，贪心策略一旦经过证明成立后，它就是一种高效的算法。

贪心算法还是很常见的算法之一，这是由于它简单易行，构造贪心策略不是很困难。

可惜的是，它需要证明后才能真正运用到题目的算法中。

一般来说，**贪心算法的证明围绕着：整个问题的最优解一定由在贪心策略中存在的子问题的最优解得来的。**

对于例题中的 3 种贪心策略，都是无法成立（无法被证明）的，解释如下：

（1）贪心策略：选取价值最大者。反例：

$W=30$



物品：A B C

重量：28 12 12

价值：30 20 20

根据策略，首先选取物品 A，接下来就无法再选取了，可是，选取 B、C 则更好。

（2）贪心策略：选取重量最小。它的反例与第一种策略的反例差不多。

（3）贪心策略：选取单位重量价值最大的物品。反例：

$W=30$

物品：A B C

重量：28 20 10

价值：28 20 10

根据策略，三种物品单位重量价值一样，程序无法依据现有策略作出判断，如果选择 A，则答案错误。

## 回溯法

### 1、概念

回溯算法实际上一个类似**枚举**的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“**回溯**”返回，尝试别的路径。

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。

许多**复杂的**，**规模较大**的问题都可以使用回溯法，有“通用解题方法”的美称。

### 2、基本思想

在包含问题的所有解的解空间树中，按照**深度优先搜索的策略**，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实**回溯法就是对隐式图的深度优先搜索算法**）。

若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。

而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

### 3、用回溯法解题的一般步骤：

（1）针对所给问题，确定问题的解空间：

首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。

(2) 确定结点的扩展搜索规则

(3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

#### 4、算法框架

(1) 问题框架

设问题的解是一个  $n$  维向量  $(a_1, a_2, \dots, a_n)$ , 约束条件是  $a_i (i=1, 2, 3, \dots, n)$  之间满足某种条件, 记为  $f(a_i)$ 。

(2) 非递归回溯框架

```
1: int a[n], i;
```

```
2: 初始化数组 a[];
```

```
3: i = 1;
```

```
4: while (i>0 (有路可走) and (未达到目标)) // 还未回溯到头
```

```
5: {
```

```
6:     if (i > n) // 搜索到叶结点
```

```
7:     {
```

```
8:         搜索到一个解, 输出;
```

```
9:     }
```

```
10:    else // 处理第 i 个元素
```

```
11:    {
```

```
12:        a[i] 第一个可能的值;
```

```
13:        while (a[i] 在 不满足约束条件 且在 搜索空间内)
```

```
14:        {
```

```
15:            a[i] 下一个可能的值;
```

```

16:         }

17:         if (a[i]在搜索空间内)

18:         {

19:             标识占用的资源;

20:             i = i+1;                // 扩展下一个结点

21:         }

22:     else

23:     {

24:         清理所占的状态空间;        // 回溯

25:         i = i -1;

26:     }

27: }

```

### （3）递归的算法框架

回溯法是对解空间的深度优先搜索，在一般情况下使用递归函数来实现回溯法比较简单，其中*i*为搜索的深度，框架如下：

```

1: int a[n];

2: try(int i)

3: {

4:     if(i>n)

5:         输出结果;

6:     else

```

```

7:     {

8:         for(j = 下界; j <= 上界; j=j+1) // 枚举 i 所有可能的路径

9:         {

10:            if(fun(j)) // 满足限界函数和约束条件

11:            {

12:                a[i] = j;

13:                ... // 其他操作

14:                try(i+1);

15:                回溯前的清理工作（如 a[i]置空值等）;

16:            }

17:        }

18:    }

19: }

```

## 分支限界法

### 一、基本描述

类似于回溯法，也是一种在问题的解空间树  $T$  上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。**回溯法**的求解目标是找出  $T$  中满足约束条件的**所有解**，而**分支限界法**的求解目标则是找出**满足约束条件的一个解**，或是在满足约束条件的解中找出使某一目标函数值达到**极大或极小的解**，即在某种意义下的**最优解**。

#### （1）分支搜索算法

所谓“分支”就是采用**广度优先**的策略，依次搜索 E-结点的所有分支，也就是所有相邻结点，抛弃不满足约束条件的结点，其余结点加入活结点表。然后从表中选择一个结点作为下一个 E-结点，继续搜索。

选择下一个 E-结点的方式不同，则会有几种不同的分支搜索方式。

1) FIFO 搜索

2) LIFO 搜索

3) 优先队列式搜索

## (2) 分支限界搜索算法

### 二、分支限界法的一般过程

由于求解目标不同，导致分支限界法与回溯法在解空间树 T 上的搜索方式也不相同。回溯法以深度优先的方式搜索解空间树 T，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树 T。

分支限界法的**搜索策略是**：在扩展结点处，先生成其所有的儿子结点(分支)，然后再从当前的活结点表中选择下一个扩展对点。为了有效地选择下一扩展结点，以加速搜索的进程，在每一活结点处，计算一个函数值(限界)，并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。

分支限界法常以广度优先或以最小耗费(最大效益)优先的方式搜索问题的解空间树。问题的**解空间树是表示问题解空间的一棵有序树，常见的有子集树和排列树**。在搜索问题的解空间树时，分支限界法与回溯法对当前扩展结点所使用的扩展方式不同。在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，那些导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被子加入活结点表中。此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所求的解或活结点表为空时为止。

### 三、回溯法和分支限界法的一些区别

有一些问题其实无论用回溯法还是分支限界法都可以得到很好的解决，但是另外一些则不然。也许我们需要具体一些的分析——到底何时使用分支限界而何时使用回溯呢？

回溯法和分支限界法的一些区别：

1. 方法对解空间树的搜索方式
2. 存储结点的常用**数据结构**
3. 结点存储特性常用应用
4. 回溯法深度优先搜索堆栈活结点的所有可行子结点被遍历后才被从栈中弹出找出满足约束条件的所分支，分支限界法广度优先或最小消耗优先搜索队列、优先队列每个结点只有一次成为活结点的机会找出满足约束条件的一个解或特定意义下的最优解