# 算法设计与分析复习参考

沈容舟 anticlockwise5@gmail.com

2006年6月6日

# 前言

该文档仅供算法设计与分析考试参考，若有任何错误，请与本人联系，谢谢。

# Contents

# Chapter 1

# 基本概念

## 1.1 算法的特性

算法的特性如下:

1. 输入: 有零个或多个外部量作为算法的输入
2. 输出: 算法产生至少一个量作为输出
3. 确定性: 组成算法的每条指令是清晰的,无歧异的
4. 有限性: 算法中每条指令的执行次数有限,执行每条指令的时间也有限

## 1.2 算法复杂度

算法的复杂度是算法运行所需要的计算机资源的量，需要时间资源的量称为**时间复杂度**，需要的空间资源的量称为**空间复杂度**。

## 1.3 O, Ω, Θ 的具体定义

以下设f(N)和g(N)是定义在正数集上的正函数。

如果存在正的常数C和自然数$N_0$，使得当$N \geq N_0$时有$f(N) \leq Cg(N)$，则称函数f(N)当N充分大时上有界，且g(N)是它的一个上界，记为$f(N)=O(g(N))$。这时还说f(N)的阶不高于g(N)的阶。

如果存在正的常数C和自然数$N_0$，使得当$N \geq N_0$时，有$f(N) \geq Cg(N)$，则称函数f(N)当N充分大时下有界；且g(N)是它的一个下界，记为$f(N)=\Omega(g(N))$。

## 1.4 最坏情况下复杂度和平均情况下复杂度

1. 最坏情况下的时间复杂度: $T\max(n) = \max\{T(N,I)|size(I) = n\}$

## 1.5 递归

### 1.5.1 定义

直接或间接的调用自身的算法称为**递归算法**。用函数自身给出定义的函数称为**递归函数**。

### 1.5.2 递归算法二要素

1. 递归边界条件。
2. 递归定义：使问题向边界条件转化的规则。

# Chapter 2

# 算法

## 2.1 递归算法

### 2.1.1 多项式求值递归算法及时间复杂度

$$f(x) = a_n x^n + a_{n-1} x^{n-1} \ldots + a_2 x^2 + a_1 x^1 + a_0 \tag{2.1}$$

**算法描述**

```
public static int polynomial(int x, int n, int[] a) {
    int length = a.length;
    if(n == 0)
        return a[length - n - 1];
    else
        return x * f(n - 1) + a[length - n - 1];
}
```

**算法复杂度**

### 2.1.2 Hanoi塔算法步骤及时间复杂度

设a,b,c是3个塔座。开始时，在塔座a上有一叠共n个圆盘，这些原盘子下而上，由大到小的叠在一起。 个圆盘从小到大编号为1，2，...，n，现要求将塔座a上的这一叠圆盘移到塔座b上，并仍按同样顺序叠 置。

在移动圆盘时应 遵守以下移动规则:

- 规则（1）：每次只能移动1个圆盘;
- 规则（2）：任何时候都不允许将较大的圆盘压在较小的圆盘之上;
- 规则（3）：在满足移动规则（1）和规则（2）的前提下，可将圆盘移至a,b,c中任一塔座上。

**算法描述**

这个问题有一个简单的解法。假设塔座a,b,c排成一个三角形， a→b→c→a构成一顺时针循环。 在移动圆盘的过程中，若是奇数次移动，则保持最小的圆盘不动。 而在其它两个塔座之间，将较小的圆盘移到另一塔座上去。

上述算法简洁明确，可以证明它是正确的。但只看看法的计算步骤， 很难理解他的道理，也很难理解他的设计思想。下面用递归技术来解决统一问题。 当n=1时，问题比较简单，只要将编号为1的圆盘从塔座a直接移至塔座b上即可。 当n>1时，需要利用塔座c作为辅助塔座。 此时如能设法将n-1个较小的圆盘依照移动规则从塔座a移至塔座c，然后， 将剩下的最大圆盘从塔座a移至塔座b，最后， 在设法将n-1个较小的圆盘依照移动规则从塔座c移至塔座b。由此可见， n个圆盘的移动问题可分为两次n-1个圆盘的移动问题，这又可以递归的用上述方法来做。 由此可以 设计出解Hanoi塔问题的地柜算法如下:

```
public static void hanoi(int n, int a, int b, int c) {
    if(n > 0) {
        hanoi(n - 1, a, c, b);
        move(a, b);
        hanoi(n - 1, c, b, a);
    }
}
```

其中，hanoi(n, a, b, c)表示将塔座a上自下而上，由大到小叠在一起的n 个圆盘依照移动规则移至塔座b上并仍按同样顺序叠放。

**时间复杂度**

## 2.2 分治法

### 2.2.1 基本思想

分治法的基本思想是将一个规模为n的问题分解为k个规模较小的子问题， 这些子问题互相独立且与原问题相同。递归的解这些子问题， 然后将各自问题的解合并得到原问题的解。

### 2.2.2 棋盘覆盖算法步骤及时间复杂度

在一个$2^k \times 2^k$个方格组成的棋盘中， 恰有一个方格与其他方格不同，称该方格为一特殊方格， 切成该棋盘为一个特殊棋盘。显然特殊方格在棋盘上出现的位置有$4^k$种情形。 因而对任何k≥0，有$4^k$种不同的特殊棋盘。

**算法步骤**

当k>0时，将$2^k \times 2^k$棋盘分割为4个$2^{k-1} \times 2^{k-1}$子棋盘。

特殊方格必位于4个较小子棋盘指一种，其余3个子棋盘中无特殊方格。 为了将这3个特殊方格的子棋盘转化为特殊棋盘， 可以用一个L型骨牌覆盖这3个较小棋盘的会合处，这3个子棋盘上被L型骨牌覆盖的 方格就成为该棋盘上的特殊方格，从而将圆问题转化为4个较小规模的覆盖问题。 递归的使用这种分割，直至棋盘简化为1×1棋盘。

```
public void chessBoard(int tr, int tc, int dr, int dc, int size) {
    if(size == 1)
        return;
    int t = tile ++, // L型骨牌号
        s = size / 2; // 分割棋盘

    if(dr < tr + s && dc < tc + s)
        chessBoard(tr, tc, dr, dc, s);
    else {
        board[tr + s - 1][tc + s - 1] = t;
```

```
        chessBoard(tr, tc, tr + s - 1, tc + s - 1, s);
    }

    if(dr < tr + s && dc >= tc + s)
        chessBoard(tr, tc, + s, dr, dc, s);
    else {
        board[tr + s - 1][tc + s] = t;
        chessBoard(tr, tc + s, tr + s - 1, tc + s, s);
    }

    if(dr >= tr + s && dc < tc + s)
        chessBoard(tr + s, tc, dr, dc, s);
    else {
        board[tr + s][tc + s - 1] = t;
        chessBoard(tr + s, tc, tr + s, tc + s - 1, s);
    }

    if(dr >= tr + s && dc >= tc + s)
        chessBoard(tr + s, tc + s, dr, dc, s);
    else {
        board[tr + s][tc + s] = t;
        chessBoard(tr + s, tc + s, tr + s, tc + s, s);
    }
}
```

上述算法中，用整形数组board表示棋盘。board[0][0] 是棋盘的左上角方格。tile是算法中的一个全局整形变量， 用来表示L型骨牌的编号，其初始值为0。算法的输入参数是:

**tr: 棋盘左上角方格的行号;**

**tc: 棋盘左上角方格的列号;**

**dr: 特殊方格所在的行号;**

**dc: 特殊方格所在的行号;**

**size: $2^k$，棋盘规格为$2^k \times 2^k$。**

**时间复杂度**

设T(k)是算法chessBoard覆盖一个$2^k \times 2^k$棋盘所需的时间。从算法的 分割策略可知，T(k)满足如下递归方程:

$$T(k) = \begin{cases} O(1) & \text{k} = 0 \\ 4T(k-1) + O(1) & \text{k} > 0 \end{cases}$$

该算法的时间复杂度为: $\mathrm{T(k)} = \mathrm{O}(4^k)$。

### 2.2.3 合并排序算法及其时间复杂度

**算法描述**

合并排序算法使用分治策略实现对n个元素进行排序的算法。其基本思想是: 将待排序元素分成大小大相同

的2个子集合，分别对2个子集合进行排序， 最终将排好序的子集合合并成为所要求的排好序的集合。 合并排序算法可递归的描述如下:

```
public static void mergeSort(Comparable a[], int left, int right) {
    if(left < right) {
        int i = (left + right) / 2;
        mergeSort(a, left, i);
        mergeSort(a, i + 2, right);
        merge(a, b, left, i, right);
        copy(a, b, left, right);
    }
}
```

**时间复杂度**

合并排序算法对n各元素进行排序，在最坏情况下所需的时间T(n)满足:

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

解此方程可知: T(n) = O(nlog $n$)。由于排序问题的计算时间下界为$\Omega$(nlog $n$)， 故合并排序算法是渐进最优算法。

## 2.3 动态规划

### 2.3.1 基本步骤

动态规划算法适用于解最优化问题。通常可以按以下步骤 设计动态规划算法:

1. 找出最优解的性质，并刻画其结构特征;
2. 递归的定义最优值;
3. 以自底向上的方式计算出最优值;
4. 根据计算最优值时得到的信息，构造最优解。

### 2.3.2 货郎担问题

### 2.3.3 流水作业问题

n个作业 {1, 2, ..., n}要在两台机器$M_1$和$M_2$组成的流水线上完成加工。 每个作业加工的顺序都是先在$M_1$上加工，然后在$M_2$上加工。 $M_1$和$M_2$加工作业i所需的时间分别为$a_i$和$b_i$。 流水作业调度问题要求确定这n个作业的最优加工顺序使得从第一个作业在机器$M_1$上开始加工， 到最后一个作业在机器$M_2$上加工完成所需的时间最少。

**最优子结构**

流水作业调度问题具有最有字结构性质。

设$\pi$是所给n个流水作业的一个最优调度， 他所需的加工时间为$a_{\pi(1)} + T'$。 其中$T'$是在机器$M_2$的等待时间为$b_{\pi(1)}$时，安排作业$\pi(2)$，...， $\pi(n)$所需要的时间。

记$S = N - \{\pi(1)\}$, 则有$T' = T(S, b_{\pi(1)})$。

事实上，由T的定义知$T^{'} \geq T(S, b_{\pi(1)})$。若$T^{'} > T(S, b_{\pi(1)})$，设$\pi^{'}$是作业集S在机器$M_2$的等待时间为$b_{\pi(1)}$的情况下的一个最优调度。则$\pi(1)\pi^{'}(2)\ldots\pi^{n}$是N的一个调度，且该调度所需的时间为$a_{\pi(1)} + T(S, b_{\pi(1)}) < a_{\pi(1)} + T^{'}$。这与$\pi$是N的最优调度矛盾。故$T^{'} \leq T(S, b_{\pi(1)})$。从而 $T^{'} = T(S, b_{\pi(1)})$。这就证明了流水作业调度问题具有最优子结构的性质。

**递归计算最优值**

由流水作业调度问题的最优子结构性质可知，

$$T(N, 0) = \min_{1 \leq i \leq n} \{a_i + T(N - \{i\}, b_i)\}$$

推到一般情形下便有

$$T(S, t) = \min_{i \in S} \{a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\})\}$$

其中，$\max\{t - a_i, 0\}$这一项是由于在机器$M_2$上，作业i须在$\max\{t, a_i\}$时间之后才能开工。因此，在机器$M_i$上完成作业i之后，机器上还需

$$b_i + \max\{t, a_i\} - a_i = b_i + \max\{t - a_i, 0\}$$

时间才能完成对作业i的加工。

**流水作业调度的Johnson法则**

设$\pi$是作业集S在机器$M_2$的等待时间为t时的任意最优调度。若在这个调度中，安排在最前面的两个作业分别是i和j，即$\pi(1) = i, \pi(2) = j$。则由动态规划递归式可得

$$T(S, t) = a_i + T(S - \{i\}, b_i + \max\{t - a_i, 0\}) = a_i + a_j + T(S - \{i, j\}, t_{ij})$$

其中

$$\begin{aligned} t_{ij} &= b_j + \max\{b_i + \max\{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\{\max\{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\{t - a_i, a_j - b_i, 0\} \\ &= b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_i, a_i\} \end{aligned}$$

如果作业i和j满足$\min\{b_i, a_i\} \geq \min\{b_j, a_i\}$，则称作业i和j满足Johnson不等式。

如果作业i和j不满足Johnson不等式，则交换作业i和作业j的加工顺序后，作业i和j满足Johnson不等式。

在作业集S当机器$M_2$的等待时间为t时的调度$\pi$中，交换作业i和作业j的加工顺序，得到作业集S的另一调度$\pi^{'}$，它所需的加工时间为

$$T^{'}(S, t) = a_i + a_j + T(S - \{i, j\}, t_{ji})$$

其中

$$t_{ji} = b_j + b_i - a_j - a_i + \max\{t, a_i + a_j - b_j, a_j\}$$

当作业i和j满足Johnson不等式$\min\{b_i, a_i\} \geq \min\{b_j, a_i\}$时，有

$$\max\{-b_i, -a_j \leq \max\{-b_j, -a_i\}$$

从而

$$a_i + a_j + \max\{-b_i, -a_j\} \leq a_i + a_j + \max\{-b_j, -a_i\}$$

由此可得

$$\max\{a_i + a_j - b_i, a_i\} \leq \max\{a_i + a_j - b_j, a_j\}$$

因此对任意t有

$$\max\{t, a_i + a_j - b_i, a_i\} \leq \max\{t, a_i + a_j - b_j, a_j\}$$

从而，$t_{ij} \leq t_{ji}$。由此可见$T(S,t) \leq T^{'}(S,t)$。

换句话说，当作业i和作业j不满足Johnson不等式时，交换他们的加工顺序后，作业i和j满足Johnson不等式，且不增加加工时间。由此可知，对于流水作业调度问题，必存在最优调度$\pi$，使得作业$\pi(i)$和$\pi(i+1)$满足Johnson不等式

$$\min\{b_{\pi(i)}, a_{\pi(i+1)}\} \geq min\{b_{\pi(i+1)}, a_{\pi(i)}\}, \quad 1 \leq i \leq n-1$$

这样的调度$\pi$称为满足Johnson法则的调度。

进一步还可以证明，调度$\pi$满足Johnson法则当且仅当对任意i<j有

$$\min\{b_{\pi(i)}, a_{\pi(j)}\} \geq min\{b_{\pi(j)}, a_{\pi(i)}\} \tag{2.2}$$

由此可知，任意两个满足Johnson法则的调度具有相同的加工时间。从而所有满足Johnson法则的调度均为最优调度。至此，将流水作业调度问题转化为求满足Johnson法则的调度问题。

**算法描述**

流水作业调度问题的Johnson算法：

**(1) 令$N_1 = \{i|a_i < b_i\}, N_2 = \{i|a_i \geq b_i\}$；**

**(2) 将$N_1$中作业依$a_i$的非减序排序；将$N_2$中作业依$b_i$的非增序排序；**

**(3) $N_1$中作业接$N_2$中作业构成满足Johnson法则的最优调度。**

**时间复杂度分析**

算法的主要计算时间花在对作业集的排序。因此，在最坏情况下算法所需的计算时间为$O(n\log n)$。

### 2.3.4 0-1 背包问题

0-1背包问题：给定n种物品和一个背包。物品i的重量是$w_i$，其价值为$v_i$，背包的容量为C。问：应该如何选择装入背包的物品，使得装入背包种物品的总价值最大？

在选择装入背包的物品时，对每种物品i只有两种选择，即装入背包或不装入背包。不能将物品i装入背包多次，也不能只装入部分的物品i。

**最优子结构性质**

0-1背包问题具有最优子结构性质。设$(y_1, y_2, \ldots, y_n)$是所给0-1背包问题的一个最优解，则$(y_2, \ldots, y_n)$是下面相应子问题的一个最优解：

$$\max \sum_{i=2}^{n} v_i x_i$$

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \leq C \\ x_i \in \{0,1\}, 2 \leq i \leq n \end{cases}$$

因若不然，设$(z_2,\ldots,z_n)$是上述子问题的一个最优解， 而$(y_2,\ldots,y_n)$不是它的最优解。 由此可知，$\sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i$， 且$w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le C$。因此

$$v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=2}^{n} v_i y_i$$
$$w_1 y_1 + \sum_{i=2}^{n} w_i z_i \le C$$

这说明$(z_1, z_2,\ldots,z_n)$时所给0-1背包问题的更优解， 从而$(y_1, y_2,\ldots,y_n)$不是所给0-1背包问题的最优解。 此为矛盾。

**递归关系**

设所给0-1背包问题的子问题

$$\max \sum_{k=i}^{n} v_k x_k$$
$$\begin{cases} \sum_{k=i}^{n} w_k x_k \le j \\ x_i \in \{0,1\}, i \le k \le n \end{cases}$$

的最优解为$m(i,j)$，即$m(i,j)$是背包容量为j, 可选择物品为$i, i+1,\ldots,n$时0-1背包问题的最优解。由0-1背包问题 的最优子结构性质，可以建立计算$m(i,j)$的递归式如下:

$$m(i,j) = \begin{cases} \max\{m(i+1,j), m(i+1,j-w_i)+v_i\} & j \ge w_i \\ m(i+1,j) & 0 \le j < w_i \end{cases}$$
$$m(n,j) = \begin{cases} v_n & j > w_n \\ 0 & 0 \le j < w_n \end{cases}$$

## 2.4 贪心算法

### 2.4.1 基本思想

顾名思义，贪心算法总是做出在当前看来最好的选择， 也就是说贪心算法并不从整体最优考虑， 它所做出的选择只是在某种意义上的局部最优选择。

### 2.4.2 贪心算法的基本要素

**贪心选择性质**

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择， 及贪心选择来达到。这是贪心算法可行的第一个基本要素， 也是贪心算法与动态规划算法的主要区别。

证明贪心选择性质：首先考察问题的一个整体最优解，并证明可以修改这个最优解， 使其以贪心选择开始。做出贪心选择后，原问题简化为规模更小的类似子问题。 然后，用数学归纳法证明，通过每一步做贪心选择，最终可得到问题的整体最优解。

**最优子结构性质**

当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。 通常用反证法证明。

### 2.4.3 背包问题

与0-1背包问题类似，所不同的是在选择物品i装入背包时，可以选择物品i的一部分， 而不一定要全部装入背包，$1 \leq i \leq n$。

此问题的形式化描述是，给定$C > 0, w_i > 0, v_i > 0, 1 \leq i \leq n$， 要求找出一个n元向量$(x_1, x_2, \ldots, x_n)$, $0 \leq x_i \leq 1, 1 \leq i \leq n$，使得$\sum_{i=1}^{n} w_i x_i \leq C$， 而且$\sum_{i=1}^{n} v_i x_i$，达到最大。

## 2.5 回溯法

### 2.5.1 基本步骤

用回溯法解题通常包含以下3个步骤:

1. 针对所给问题，定义问题的解空间
2. 确定易于搜索的解空间结构
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。

### 2.5.2 剪枝函数

回溯法搜索解空间树时，通常采用两种策略避免无效搜索，提高回溯法的搜索效率。 其一是用约束函数在扩展结点处剪去不满足约束的子树； 其二是用限界函数剪去得不到最优解的子树。这两类函数统称为剪枝函数。

### 2.5.3 子集树与排列树

当所给的问题是从n个元素的集合S中找出S满足某种性质的子集时， 相应的解空间树称为子集树。

当所给问题时确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。

### 2.5.4 旅行售货员问题

**解空间**

旅行售货员问题的解空间是一棵排列树。

**空间组织结构**