数据结构及算法应用

算法设计策略

1. 分治法

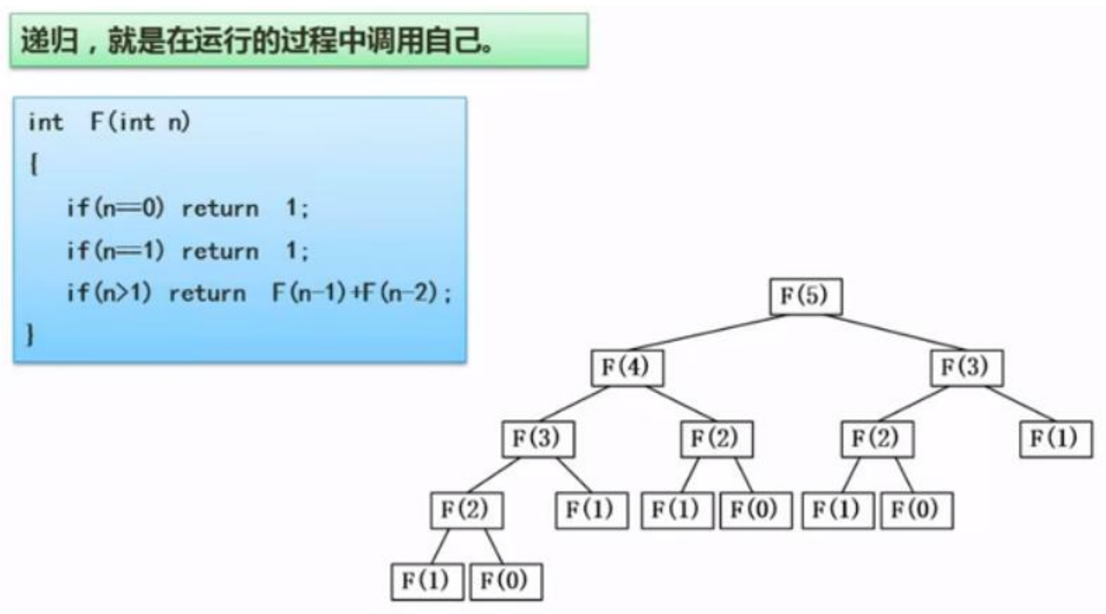
1.1. 基本思想

将一个规模为 n 的问题分解为 k 个规模的较小的子问题，
这些子问题相互独立且与原问题相同。
递归地解这些子问题，然后将各个子问题的解合并得到原问题的解。

问题的特点
- ✓ 该问题的规模缩小到一定程度就可以容易地解决（递归的出口）；
- ✓ 该问题可以分解为若干个相互独立且与原问题相同的子问题；
- ✓ 子问题的解可以合并为该问题的最终解。

分解-->递归解决-->合并

1.2. 递归技术

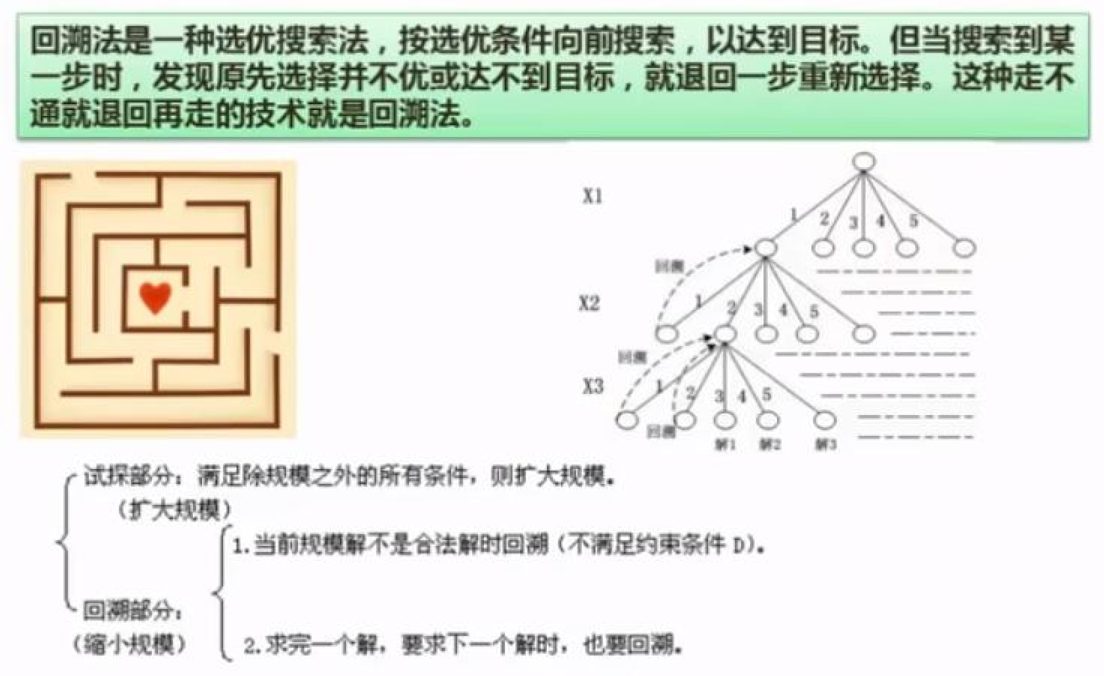


1.2.1. 典型案例

归并排序
二分查找

1.3. 回溯法

是一种**深度优先搜索**法。至少求得一个（最优）解。



回溯法解题步骤

(1) 针对所给问题，定义问题的解空间树；
(2) 确定易于搜索的解空间结构；
(3) 以深度优先的方式搜索解空间；
(4) 如果需要获取最优解，则需要对所有解进行对比，直至找到最优解。

0-1 背包问题的解空间树

一个（最优）解。例如，对于有 n 种可选择物品的 0-1 背包问题，其解空间由长度为 n 的 0-1 向量组成。该解空间包含了对变量的所有可能的 0-1 赋值。当 n=3 时，其解空间是{(0,0,0), (0,1,0), (0,0,1), (1,0,0), (0,1,1), (1,0,1),(1,1,0),(1,1,1)}。

定义了问题的解空间后，还应将解空间很好地组织起来，使得用回溯法能方便地搜索整个解空间。通常将解空间表示为树或图的形式。例如，对于 n=3 时的 0-1 背包问题，其解空间用一棵完全二叉树表示，如图 8-4 所示。

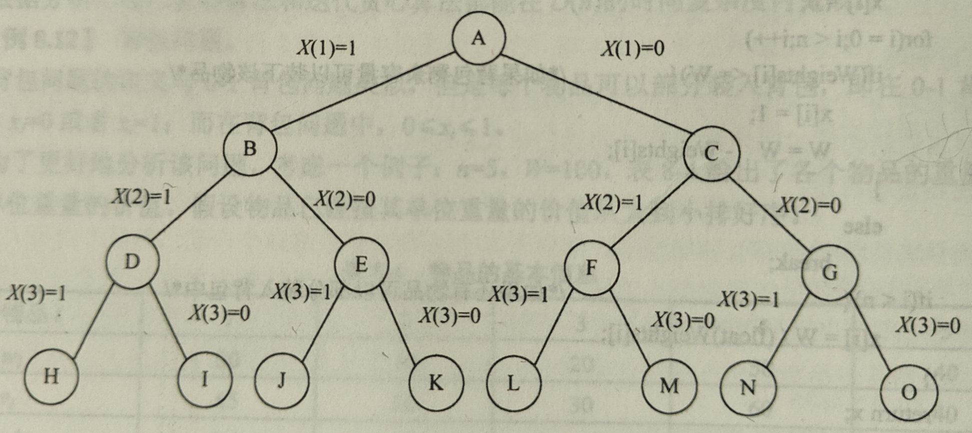


图 8-4　0-1 背包问题的解空间树

解空间树的第 i 层到第 i+1 层边上的标号给出了变量的值。从树根到叶子的任一路径表示解空间的一个元素。例如，从根结点到结点 H 的路径对应于解空间中的元素(1,1,1)。

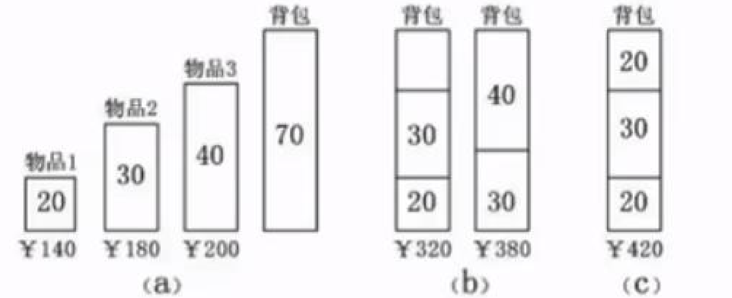
回溯法的限界函数

“剪枝”，尽可能早点地“杀掉”不可能产生最优解的活结点，从而**提高搜索效率**。

1.4. 贪心法

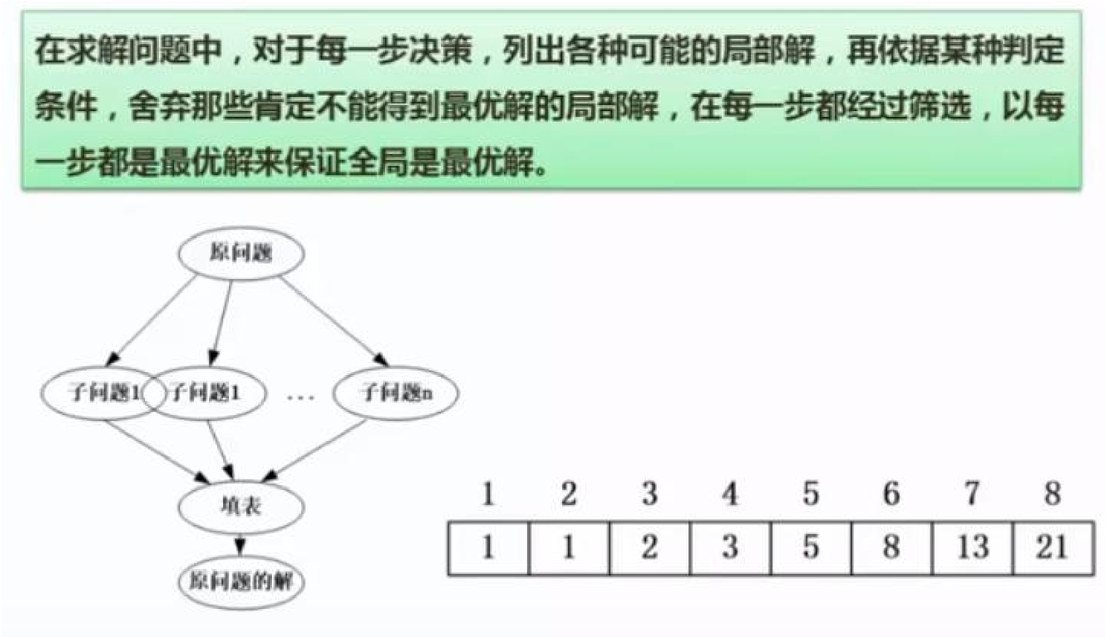
贪心法仅根据当前状态下局部最好来做出选择，而且一旦做出了选择，不管将来有什么结果，这个选择都不会改变。不从整体考虑最优，**属于局部最优解算法**，典型的背包问题。

总是做出在当前来说是最好的选择，而并不从整体上加以考虑，它所做的每步选择只是当前步骤的局部最优选择，但从整体来说不一定是最优的选择。由于它不必为了寻找最优解而穷尽所有可能解，因此其耗费时间少，一般可以快速得到满意的解，但得不到最优解。



1.5. 动态规划法

拆分成子问题，解决子问题，比较获取最优解并记录到二维数组表格中（不管这个记录后续是否用到都要记录），计算父问题最优解时，查表获取子问题最优解来求解父问题最优解，避免了重复计算子问题。



算法的基本要素：最优子结构性质和重叠子问题性质；

算法设计 4 个步骤：

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

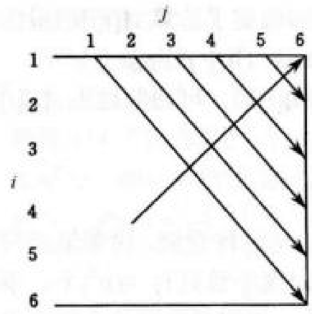
分割思路是：
对 r 个矩阵，例如 r=4，ABCD 四个矩阵；(A(BCD)),((AB)(CD)),((ABC)D);
通过移动切割位置来遍历所有情况，比较获取最小值，其中 BCD 和 ABC 这种连乘的值是取已保存好的最优值，所以无需再计算譬如(A(B(CD)))这些情况。因此需要先计算三个矩阵连乘的最优值、三个矩阵连乘要用到两个矩阵连乘的最优值（两个矩阵连乘只有一种情况，无最优值可言），单个矩阵就没有计算意义，这就是自底向上的计算方式。

1.5.1. 矩阵连乘问题

例如，要计算矩阵连乘积 $A_1A_2A_3A_4A_5A_6$，其中各矩阵的维数分别为：

A_1	A_2	A_3	A_4	A_5	A_6
30×35	35×15	15×5	5×10	10×20	20×25

动态规划算法 MatrixChain 计算 m[i][j]先后次序如图 3-1(a)所示；计算结果 m[i][j]和 s[i][j]（$1\le i\le j\le n$），分别如图 3-1(b)和(c)所示。



(a) 计算次序

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

(b) m[i][j]

i \ j	1	2	3	4	5	6
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

(c) s[i,j]

图 3-1 计算 m[i][j]的次序

4 矩阵连乘：$A_1A_2A_3A_4$

计算结果分为 4-1 种情况：

两连乘：A_1A_2、A_2A_3、A_3A_4

三连乘：$A_1A_2A_3$、$A_2A_3A_4$

四连乘：$A_1A_2A_3A_4$

用三连乘表示四连乘：$((A_1A_2A_3)A_4)$、$(A_1(A_2A_3A_4))$

用两连乘表示三连乘 $A_1A_2A_3$：$((A_1A_2)A_3)$、$(A_1(A_2A_3))$

用两连乘表示三连乘 $A_2A_3A_4$：$((A_2A_3)A_4)$、$(A_2(A_3A_4))$

java 源码

```
public class MatrixChain {


    /**
     * 矩阵连乘法
     *
     * @param p 按顺序记录矩阵的行和列
     * @param m 最优值数组 m，记录计算位置和最小计算次数
     * @param s 最优断开位置数组 s，即求解出最小计算次数需要在哪里断开加括号
```

```
     */
    public static void matrixchain(int[] p, int[][] m, int[][] s) {
        int n = p.length - 1;// 矩阵个数
        int tempMin = -1;
        int tempIndex = 0;
        // 初始化数组，使 m[][]对角线上的元素为 0
        for (int i = 1; i <= n; i++)
            m[i][i] = 0; // 在 i=j 时，就是同一个矩阵连乘，无连乘的意义
        // r 为问题规模，处理不同规模的子问题，即多少个矩阵连乘（一个矩阵无
连乘意义，所以从 2 开始）
        for (int r = 2; r <= n; r++) {
            // 从第 i 个开始
            for (int i = 1; i <= n - r + 1; i++) {
                // 到第 j 个结束
                int j = i + r - 1;
                tempMin = -1;
                // i<j 对从 i 到 j 的所有决策，求最优值
                for (int k = i; k < j; k++) {
                    int temp = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] *
p[j];
                    if (tempMin == -1 || temp < tempMin) {
                        tempMin = temp;
                        tempIndex = k;
                    }
                }
                m[i][j] = tempMin; // 子问题的最优计算次数
                s[i][j] = tempIndex; // 子问题的最优分割位置
            }
        }
    }

    // 递归构造最优解
    public static void print(int[][] s, int i, int j) {
        // 当 i==j 时，表示只有一个矩阵,即矩阵号
        if (i == j) {
            System.out.print("A" + i);
            return;
        }
        System.out.print("(");
        // s[i][j]为 i 到 j 的最优切割位置
        print(s, i, s[i][j]); // 左边 i 到 s[i][j]
        print(s, s[i][j] + 1, j);// 右边 s[i][j]+1 到 j
        System.out.print(")");
    }
```

```java
    // 格式化输出
    public static void printArr(int[][] c) {
        for (int i = 1; i < c.length; i++) {
            for (int j = 1; j < c[i].length; j++) {
                System.out.print(c[i][j] + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        // 按顺序记录矩阵的行和列，第 i 个矩阵的行数存储在数组 p 的第 i-1 个位置，列数存储在数组 p 的第 i 个位置
        int[] p = { 30, 35, 15, 5, 10, 20, 25 };
        int n = p.length - 1; // n 个矩阵
        int N = p.length; //为了忽略数组下标从 0 开始影响对算法的理解
        int[][] m = new int[N][N];// 记录矩阵连乘的最优值
        int[][] s = new int[N][N];// 存放各个子问题的最优决策

        matrixchain(p, m, s);
        System.out.println("所有阶乘的最优值：");
        printArr(m);
        System.out.println("所有阶乘最优值的加括号位置：");
        printArr(s);
        System.out.println("计算最优值为： " + m[1][n]);
        System.out.print("最优解为： ");
        print(s, 1, n);
    }

}
```

执行结果：

```
所有阶乘的最优值：
0	15750	7875	9375	11875	15125
0	0	2625	4375	7125	10500
0	0	0	750	2500	5375
0	0	0	0	1000	3500
0	0	0	0	0	5000
0	0	0	0	0	0
所有阶乘最优值的加括号位置：
0	1	1	3	3	3
0	0	2	3	3	3
0	0	0	3	3	3
0	0	0	0	4	5
```

```
0   0   0   0   0   5
0   0   0   0   0   0
计算最优值为：15125
最优解为：((A1(A2A3))((A4A5)A6))
```

```
所有阶乘的最优值：
0       15750   7875    9375    11875   15125
0       0       2625    4375    7125    10500
0       0       0       750     2500    5375
0       0       0       0       1000    3500
0       0       0       0       0       5000
0       0       0       0       0       0
所有阶乘最优值的加括号位置：
0       1       1       3       3       3
0       0       2       3       3       3
0       0       0       3       3       3
0       0       0       0       4       5
0       0       0       0       0       5
0       0       0       0       0       0
计算最优值为：15125
最优解为：((A1(A2A3))((A4A5)A6))
```