

# 算法设计与分析

## 代码背诵

李昊伦

2025.12

# 1 时间复杂度与渐近符号

## 1.1 时间复杂度的基本概念

设算法输入规模为  $n$ , 输入实例为  $I$ , 算法在输入  $I$  上的运行时间记为  $T(I)$ 。

**1. 最坏情况时间复杂度:**  $T_{\max}(n) = \max_{|I|=n} T(I)$ .

**解释:** 在所有规模为  $n$  的输入中, 选取运行时间最长的那个输入, 其运行时间作为算法在规模  $n$  下的时间复杂度。

**意义:** 最坏情况时间复杂度给出了算法性能的上界, 是算法分析中最常用、最保守、也是最安全的度量方式。

**2. 最好情况时间复杂度:**  $T_{\min}(n) = \min_{|I|=n} T(I)$ .

**解释:** 在所有规模为  $n$  的输入中, 选取运行时间最短的那个输入。

**说明:** 最好情况通常过于理想, 不能反映算法的真实性能, 因此在理论分析中参考价值较小。

**3. 平均情况时间复杂度:**  $T_{\text{avg}}(n) = \sum_{|I|=n} P(I) T(I)$ .

其中  $P(I)$  表示输入  $I$  出现的概率, 且满足  $\sum_{|I|=n} P(I) = 1$ .

**解释:** 平均情况时间复杂度是对所有输入运行时间的加权平均。

**说明:** 平均情况分析通常需要对输入分布作概率假设, 分析过程复杂, 因此在实际算法分析中使用较少。

## 1.2 渐近符号的定义

渐近符号用于刻画当  $n \rightarrow \infty$  时, 函数增长速度的数量级, 忽略常数因子和低阶项。

**1. 大  $O$  记号 (渐近上界):**  $f(n) = O(g(n))$  当且仅当存在正常数  $c > 0$  和自然数  $n_0$ , 使得

$$\forall n \geq n_0, \quad 0 \leq f(n) \leq c g(n).$$

**解释:** 当  $n$  足够大时, 函数  $f(n)$  的增长速度不会超过  $g(n)$  的某个常数倍。

**含义:** 大  $O$  记号给出了函数增长速度的上界。

**2.  $\Omega$  记号 (渐近下界):**  $f(n) = \Omega(g(n))$  当且仅当存在正常数  $c > 0$  和自然数  $n_0$ , 使得

$$\forall n \geq n_0, \quad f(n) \geq c g(n).$$

**解释:** 当  $n$  足够大时, 函数  $f(n)$  的增长速度至少不小于  $g(n)$  的某个常数倍。

**含义:**  $\Omega$  记号给出了函数增长速度的下界。

**3.  $\Theta$  记号 (精确渐近界):**  $f(n) = \Theta(g(n))$  当且仅当  $f(n) = O(g(n))$  且  $f(n) = \Omega(g(n))$ 。等价地, 存在正常数  $c_1, c_2 > 0$  和自然数  $n_0$ , 使得

$$\forall n \geq n_0, \quad c_1 g(n) \leq f(n) \leq c_2 g(n).$$

**解释:** 函数  $f(n)$  与  $g(n)$  具有相同的渐近增长阶, 二者在数量级上是等价的。

**4. 小  $o$  记号 (非紧上界):**  $f(n) = o(g(n))$  当且仅当

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

**解释:**  $f(n)$  的增长速度严格慢于  $g(n)$ , 即  $f(n)$  相对于  $g(n)$  可以忽略。

**5. 小  $\omega$  记号 (非紧下界):**  $f(n) = \omega(g(n))$  当且仅当

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty.$$

**解释:**  $f(n)$  的增长速度严格快于  $g(n)$ 。

### 1.3 渐近符号的运算性质

1. 加法法则:  $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$ .

解释: 多个子过程顺序执行时, 总时间复杂度由增长速度最快的一项决定。

2. 乘法法则:  $O(f(n)) \cdot O(g(n)) = O(f(n)g(n))$ .

解释: 当一个过程嵌套在另一个过程中执行时, 时间复杂度等于二者复杂度的乘积。

3. 常见等价关系:  $O(f(n)) = O(cf(n)), c > 0$ .

$\log_a n = \Theta(\log_b n), a, b > 1$ .

说明: 渐近分析中忽略常数因子与对数底数的差异。

### 1.4 上述定理证明

1. 证明:  $O(f) + O(g) = O(f + g)$

设  $F(n) = O(f(n))$ , 则存在自然数  $n_1$  与正常数  $c_1 > 0$ , 当  $n \geq n_1$  时有

$$F(n) \leq c_1 f(n).$$

同理, 若  $G(n) = O(g(n))$ , 则存在自然数  $n_2$  与正常数  $c_2 > 0$ , 当  $n \geq n_2$  时有

$$G(n) \leq c_2 g(n).$$

当  $n \geq n_0 := \max\{n_1, n_2\}$  时, 两式同时成立, 因此

$$F(n) + G(n) \leq c_1 f(n) + c_2 g(n).$$

令

$$c_3 = \max\{c_1, c_2\},$$

则有

$$c_1 f(n) + c_2 g(n) \leq c_3 f(n) + c_3 g(n) \leq c_3 (f(n) + g(n)).$$

于是当  $n \geq n_0$  时

$$F(n) + G(n) \leq c_3 (f(n) + g(n)),$$

从而当  $n \geq n_0$  时

$$F(n) + G(n) \leq c_3 (f(n) + g(n)),$$

因此

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

2. 由  $O(f(n)) + O(g(n)) = O(f(n) + g(n))$  推出  $O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\})$  已证

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

因此存在正常数  $C_3 > 0$  与自然数  $n_0$ , 使得当  $n \geq n_0$  时

$$O(f(n)) + O(g(n)) \leq C_3 (f(n) + g(n)).$$

又因为对任意  $n$  (默认  $f(n), g(n) \geq 0$ ) 都有

$$f(n) + g(n) \leq 2 \max\{f(n), g(n)\},$$

代入上式得当  $n \geq n_0$  时

$$O(f(n)) + O(g(n)) \leq 2C_3 \max\{f(n), g(n)\}.$$

由于大  $O$  记号忽略正常数倍, 故

$$O(f(n)) + O(g(n)) = O(\max\{f(n), g(n)\}).$$

**3. 证明:**  $O(f(n)) \cdot O(g(n)) = O(f(n)g(n))$

令  $f_1(n) = O(f(n))$ , 则存在自然数  $n_1$  与正常数  $c_1 > 0$ , 当  $n \geq n_1$  时有

$$f_1(n) \leq c_1 f(n).$$

同理, 若  $g_1(n) = O(g(n))$ , 则存在自然数  $n_2$  与正常数  $c_2 > 0$ , 当  $n \geq n_2$  时有

$$g_1(n) \leq c_2 g(n).$$

当  $n \geq n_0 := \max\{n_1, n_2\}$  时, 两式同时成立, 相乘得到

$$f_1(n) g_1(n) \leq (c_1 f(n))(c_2 g(n)) = c_3 f(n)g(n),$$

其中

$$c_3 = c_1 c_2.$$

因此当  $n \geq n_0$  时

$$f_1(n) g_1(n) \leq c_3 f(n)g(n),$$

从而

$$O(f(n)) \cdot O(g(n)) = O(f(n)g(n)).$$

## 2 递归与分治

### 2.1 阶乘函数 (Factorial)

**定义与递归式:** 阶乘用于表示从 1 到  $n$  的连乘积 (约定  $0! = 1$ ):  $n! = \prod_{k=1}^n k$ 。递归定义通常写成:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n > 0. \end{cases}$$

$$n! = 1 \cdot 2 \cdots \cdot (n-1) \cdot n = n \cdot (1 \cdot 2 \cdots \cdot (n-1)) = n \cdot (n-1)!.$$

**递归计算的时间复杂度:** 若用递归实现, 每次调用把  $n$  减 1:  $T(n) = T(n-1) + O(1)$ , 展开得  $T(n) = O(n)$ 。递归深度为  $n$ , 因此额外栈空间也为  $O(n)$ 。

**等价的迭代写法:** 循环从 1 乘到  $n$ , 时间仍为  $O(n)$ , 空间可做到  $O(1)$ 。

### 2.2 斐波那契数列

$$F(n) = \begin{cases} 1, & n = 0, \\ 1, & n = 1, \\ F(n-1) + F(n-2), & n > 1. \end{cases}$$

**代码:** if ( $n \leq 1$ ) return 1; else return  $F(n-1)+F(n-2)$ ;

**朴素递归时间复杂度:** 设递归实现的运行时间为  $T(n)$ , 则  $T(n) = T(n-1) + T(n-2) + O(1)$ 。它与斐波那契增长同阶, 解的数量级为指数级:  $T(n) = \Theta(\varphi^n)$ ,  $\varphi = \frac{1+\sqrt{5}}{2}$ 。因此朴素递归在  $n$  稍大时就会爆炸性变慢。

### 2.3 Ackermann 函数 (双递归函数)

$$\begin{aligned} A(1, 0) &= 2, \\ A(0, m) &= 1, \quad m \geq 0, \\ A(n, 0) &= n + 2, \quad n \geq 2, \\ A(n, m) &= A(A(n-1, m), m-1), \quad m \geq 1. \end{aligned}$$

### 2.4 排列问题

```
template <class Type>
void Perm(Type list[], int k, int m) {
    if (k == m) { // 递归终止条件: 只剩一个元素
        for (int i = 0; i <= m; i++) // 输出当前排列
            cout << list[i];
        cout << endl;
    }
    else { // 递归生成排列
        for (int i = k; i <= m; i++) {
```

```

        Swap(list[k], list[i]); // 交换元素到当前位置k
        Perm(list, k + 1, m); // 递归生成剩余元素的排列
        Swap(list[k], list[i]); // 恢复原始顺序（回溯）
    }
}
}

template <class Type>
inline void Swap(Type &a, Type &b) {
    Type temp = a;
    a = b;
    b = temp;
}

```

**时间复杂度：**全排列的总数为  $n!$ , 任何生成全部排列的算法, 其时间复杂度至少为  $\Omega(n!)$ .

## 2.5 整数划分问题

**问题定义：**把正整数  $n$  表示成若干个正整数之和, 且加数不考虑顺序, 称为  $n$  的一个整数划分。

```

int q(int n, int m) {
    if ((n < 1) || (m < 1)) return 0; // 非法输入返回0
    if ((n == 1) || (m == 1)) return 1; // 基准情况
    if (n < m) return q(n, n); // 划分数不能超过n本身
    if (n == m) return 1 + q(n, n-1);
    if (n > m) return q(n, m-1) + q(n-m, m);
}

```

## 2.6 求解线性递推关系

**问题描述：**

给定递推关系

$$\begin{cases} X_{n+1} = X_n + 12X_{n-1}, & n \geq 1, \\ X_0 = 1, & X_1 = 0.5. \end{cases}$$

**判断递推类型**该递推关系只包含  $X_{n+1}, X_n, X_{n-1}$ , 系数为常数, 且右端无常数项, 因此是二阶常系数齐次线性递推关系。

(1) 写出特征方程

设  $X_n = r^n$ , 代入得  $r^{n+1} = r^n + 12r^{n-1}$ .

两边除以  $r^{n-1}$  得  $r^2 = r + 12$ , 即  $r^2 - r - 12 = 0$ .

(2) 求解特征方程

$(r - 4)(r + 3) = 0, r_1 = 4, r_2 = -3$ .

(3) 通解形式

$X_n = C_1(-3)^n + C_24^n$ .

(4) 利用初始条件求常数  $\begin{cases} C_1 + C_2 = 1, \\ -3C_1 + 4C_2 = 0.5. \end{cases}$

(5) 解方程组  $C_1 = 0.5, C_2 = 0.5$ .

## (6) 最终通项公式

$$X_n = 0.5(-3)^n + 0.5 \cdot 4^n$$

**理解要点总结：**假设指数解；构造特征方程；不同特征根对应指数项线性组合；初始条件用于唯一确定常数。

## 2.7 汉诺塔问题 (Tower of Hanoi)

**问题描述：**将  $n$  个大小不同的圆盘从起始柱  $a$  移动到目标柱  $b$ ，在移动过程中可以借助辅助柱  $c$ ，并且必须满足以下规则：每次只能移动一个圆盘；任意时刻，大圆盘不能放在小圆盘上面。

```
void hanoi(int n, int a, int b, int c) {
    if (n > 0) {
        hanoi(n - 1, a, c, b); // 将n-1个盘子从a移到c（借助b）
        move(a, b);           // 将第n个盘子从a直接移到b
        hanoi(n - 1, c, b, a); // 将n-1个盘子从c移到b（借助a）
    }
}
```

**时间复杂度分析：**设移动  $n$  个圆盘所需的时间为  $T(n)$ ，则有递推关系  $T(n) = 2T(n - 1) + 1$ 。  
解得： $T(n) = 2^n - 1$ . 因此，汉诺塔问题的时间复杂度为  $O(2^n)$

**空间复杂度分析：**递归调用的最大深度为  $n$ ，因此递归栈所需的空间复杂度为  $O(n)$ .

## 2.8 分治算法时间复杂度

$$T(n) = \begin{cases} O(1), & n = 1, \\ k T\left(\frac{n}{m}\right) + f(n), & n > 1. \end{cases}$$

其中： $k$  子问题个数； $n/m$  每个子问题的规模； $f(n)$  当前层的额外计算（分割或合并代价）。

## 重要结论

1. 递归深度为： $\log_m n$ .
2. 叶子结点（最底层子问题）总数量为： $k^{\log_m n} = n^{\log_m k}$ .
3. 分治算法的总时间复杂度可以表示为：

$$T(n) = n^{\log_m k} + \sum_{i=0}^{\log_m n-1} k^i f\left(\frac{n}{m^i}\right).$$

**记忆要点：**看子问题个数  $k$ ；看规模缩小倍数  $m$ ；先算  $n^{\log_m k}$ （叶子结点代价）；再看  $f(n)$  在各层累加后的大小；谁占主导，谁就是最终时间复杂度。

## 2.9 二分搜索 (Binary Search)

```
int BinarySearch(Type a[], const Type &x, int l, int r)
{
    while (l <= r) {
```

```

int m = (l + r) / 2;    // 计算中点下标
if (x == a[m])
    return m;           // 找到目标，返回位置
if (x < a[m])
    r = m - 1;         // 目标在左半区，缩小右边界
else
    l = m + 1;         // 目标在右半区，缩小左边界
}
return -1;               // 查找失败
}

```

**时间复杂度:** 设数组长度为  $n$ , 每次比较后查找区间规模减半:  $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \dots \rightarrow 1$ . 因此查找次数为  $\log_2 n$ , 二分搜索的时间复杂度为  $O(\log n)$ .

**空间复杂度:** 该实现为迭代版本, 只使用常数个辅助变量, 因此空间复杂度为  $O(1)$ .

## 2.10 大整数乘法 (Karatsuba 算法)

**问题背景:** 设  $X, Y$  为两个  $n$  位的大整数, 若直接相乘, 时间复杂度为  $O(n^2)$ .

**分治思想:** 将两个  $n$  位整数拆分为高位和低位 (假设  $n$  为偶数):  $X = a \cdot 2^{n/2} + b, Y = c \cdot 2^{n/2} + d$ .

直接展开乘积:  $XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$ .

**关键改进 (Karatsuba 思想):** 注意到  $ad + bc = (a + b)(c + d) - ac - bd$ . 因此只需计算以下三个乘法:  $ac, bd, (a + b)(c + d)$  其余部分只需要加减运算。

**递归时间复杂度:**

Karatsuba 算法的递归式为:

$$T(n) = \begin{cases} O(1), & n = 1, \\ 3T(n/2) + O(n), & n > 1. \end{cases}$$

由主定理可得:  $T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$ , 优于普通的大整数乘法  $O(n^2)$ .

## 2.11 Strassen 矩阵乘法

**问题背景:** 设  $A, B$  为  $n \times n$  矩阵, 普通矩阵乘法的时间复杂度为:  $O(n^3)$ .

**分块表示:**

将矩阵划分为四个子块

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

**Strassen 的 7 个中间矩阵**

$$\begin{aligned}
M_1 &= A_{11}(B_{12} - B_{22}), \\
M_2 &= (A_{11} + A_{12})B_{22}, \\
M_3 &= (A_{21} + A_{22})B_{11}, \\
M_4 &= A_{22}(B_{21} - B_{11}), \\
M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\
M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}), \\
M_7 &= (A_{11} - A_{21})(B_{11} + B_{12}).
\end{aligned}$$

### 结果矩阵的计算

$$\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6, \\
C_{12} &= M_1 + M_2, \\
C_{21} &= M_3 + M_4, \\
C_{22} &= M_5 + M_1 - M_3 - M_7.
\end{aligned}$$

**时间复杂度：**Strassen 矩阵乘法的递归式为  $T(n) = 7T(n/2) + O(n^2)$ . 由主定理可得： $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$ , 优于普通矩阵乘法的  $O(n^3)$ .

## 2.12 棋盘覆盖算法

要求：对一个  $2^k \times 2^k$  的棋盘进行覆盖，其中恰好有一个特殊方格（缺口），其余方格需用 L 型骨牌覆盖。算法采用分治与递归思想，始终保持：每一个递归子棋盘中恰好只有一个缺口。

```

#define N 8
int board[N][N];
int tile = 1; // 骨牌编号

// 参数说明：
// tr, tc: 当前子棋左上角位置
// dr, dc: 当前特殊方格坐标 (绝对坐标)
// size: 当前子棋盘规模 (边长)

void chessBoard(int tr, int tc, int dr, int dc, int size) {
    if (size == 1) return;
    int t = tile++;
    int s = size / 2;

    // 左上子棋盘
    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);
    }
}

```

```

// 右上子棋盘
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);
}

// 左下子棋盘
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);
}

// 右下子棋盘
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);
}
}

```

## 2.13 合并排序 (Merge Sort)

合并排序是一种典型的分治算法。其基本思想是：将待排序序列不断划分为两个子序列，分别排序后，再将两个有序子序列合并为一个有序序列。

### 1. 递归排序函数

```

void mergeSort(Type a[], int left, int right)
{
    if (left < right) {
        int i = (left + right) / 2;
        mergeSort(a, left, i);
        mergeSort(a, i + 1, right);
        merge(a, b, left, i, right);
        copy(a, b, left, right);
    }
}

```

### 2. 合并函数

```

void merge(Type a[], Type d[], int l, int m, int r)
{
    int i = l, j = m + 1, k = l;
    while (i <= m && j <= r) {
        if (a[i] <= a[j])
            d[k++] = a[i++];
        else
            d[k++] = a[j++];
    }

    if (i > m)
        for (int q = j; q <= r; q++)
            d[k++] = a[q];
    else
        for (int q = i; q <= m; q++)
            d[k++] = a[q];
}

```

**时间复杂度：**合并排序的递归关系为  $T(n) = 2T(n/2) + O(n)$ . 由主定理可得  $T(n) = O(n \log n)$ .

**空间复杂度：**由于合并过程需要额外的辅助数组，合并排序的空间复杂度为  $O(n)$ .

## 2.14 快速排序 (Quick Sort)

快速排序是一种典型的分治排序算法，其基本思想是：选取一个基准元素，将序列划分为左右两部分，使得左边元素不大于基准，右边元素不小于基准，然后分别对左右两部分递归排序。

### 1. 快速排序的递归框架

```

void QuickSort(Type a[], int p, int r)
{
    if (p < r) {
        int q = Partition(a, p, r);
        QuickSort(a, p, q - 1);
        QuickSort(a, q + 1, r);
    }
}

```

### 2. 划分函数 (Partition)

```

int Partition(Type a[], int p, int r)
{
    int i = p, j = r + 1;
    Type x = a[p];
    while (true) {
        while (a[++i] < x);
        while (a[--j] > x);
    }
}

```

```

    if (i >= j) break;
    Swap(a[i], a[j]);
}
a[p] = a[j];
a[j] = x;
return j;
}

```

### 3. 随机化划分

```

int RandomizedPartition(Type a[], int p, int r)
{
    int i = Random(p, r);
    Swap(a[i], a[p]);
    return Partition(a, p, r);
}

```

**时间复杂度：**在平均情况下，快速排序每次划分都能较均匀地分割数组，递归深度为  $O(\log n)$ ，每一层划分操作的代价为  $O(n)$ ，因此平均时间复杂度为： $O(n \log n)$ 。在最坏情况下（例如数组已经有序且基准选择不当），每次只能划分出一个规模为  $n - 1$  的子问题，时间复杂度退化为： $O(n^2)$ 。

**空间复杂度：**快速排序主要消耗递归栈空间，平均情况下递归深度为  $O(\log n)$ ，因此空间复杂度为： $O(\log n)$ 。

## 2.15 随机选择算法（Randomized Select）

随机选择算法用于在数组中查找第  $k$  小元素，其思想来源于快速排序中的划分操作。

```

RSelect(a, p, r, k)
{
    if (p == r)
        return a[p];
    mid = RandomizedPartition(a, p, r);
    if (mid == k)
        return a[mid];
    else if (mid > k)
        return RSelect(a, p, mid - 1, k);
    else
        return RSelect(a, mid + 1, r, k);
}

```

由于每一轮递归只进入一个子区间，该算法的平均时间复杂度为： $O(n)$ 。

## 2.16 寻找第 $k$ 小元素（Randomized-Select 算法）

随机划分函数 `RandomizedPartition`

```
template<class Type>
int RandomizedPartition(Type a[], int p, int r) {
    int i = Random(p, r);
    Swap(a[i], a[p]);
    return Partition(a, p, r);
}
```

### 划分函数 Partition

```
template<class Type>
int Partition(Type a[], int p, int r)
{
    int i = p, j = r + 1;
    Type x = a[p];

    while (true) {
        while (a[++i] < x && i < r);
        while (a[--j] > x);
        if (i >= j) break;
        Swap(a[i], a[j]);
    }

    a[p] = a[j];
    a[j] = x;
    return j;
}
```

### 随机化选择函数 RandomizedSelect

```
template<class Type>
Type RandomizedSelect(Type a[], int p, int r, int k) {
    if (p == r) return a[p];
    int i = RandomizedPartition(a, p, r);
    int j = i - p + 1;
    if (k <= j)
        return RandomizedSelect(a, p, i, k);
    else
        return RandomizedSelect(a, i + 1, r, k - j);
}
```

**算法复杂度分析：**在平均情况下，每一次划分都能较均匀地缩小问题规模，因此随机化选择算法的期望时间复杂度为： $O(n)$ 。相比完全排序所需的  $O(n \log n)$  时间复杂度，该算法在仅需寻找第  $k$  小元素时具有更高的效率。

## 2.17 循环赛问题

循环赛 (Round-Robin Tournament) 问题的目标是：给定  $n$  个选手 (或队伍)，安排比赛日程，使得每一对选手恰好比赛一次，并且每天 (每一轮) 每位选手只进行一场比赛。在经典模型中，若  $n = 2^k$ ，可以构造一个  $n \times n$  的表格  $a$  来描述日程安排，其中  $a[i][j]$  表示第  $i$  个选手在第  $j$  轮的对手编号 (不同教材可能从第 1 轮到第  $n - 1$  轮，这里代码按表格整体递推生成)。

```
void Table(int k, int **a)
{
    int n = 1;
    for (int i = 1; i <= k; i++)
        n *= 2;                                // 计算 n = 2^k

    for (int i = 1; i <= n; i++)
        a[1][i] = i;                          // 初始化第一行 (作为递推基底)

    int m = 1;                                // 当前已构造子表规模为 m
    for (int s = 1; s <= k; s++) {
        n /= 2;                                // 每次递推对应划分成若干块
        for (int t = 1; t <= n; t++) {
            for (int i = m + 1; i <= 2 * m; i++) {
                for (int j = m + 1; j <= 2 * m; j++) {
                    a[i][j + (t - 1) * m * 2 - m] =
                        a[i - m][j + (t - 1) * m * 2 - 2 * m];

                    a[i][j + (t - 1) * m * 2 - m] =
                        a[i - m][j + (t - 1) * m * 2 - 2 * m];
                }
            }
        }
        m *= 2;                                // 子表规模翻倍: m -> 2m
    }
}
```

**复杂度分析：**该算法需要填充  $n \times n$  的表格单元，每个单元最多被常数次赋值，因此时间复杂度为： $O(n^2)$ ；空间上需要保存整个赛程表，同样为： $O(n^2)$ .

### 3 动态规划

#### 3.1 矩阵连乘问题

**问题描述：**给定一系列矩阵  $A_1, A_2, \dots, A_n$ ，其中矩阵  $A_i$  的规模为  $p_{i-1} \times p_i$ 。由于矩阵乘法满足结合律，但不同的加括号方式会导致标量乘法次数不同，矩阵连乘问题的目标是：确定一种加括号方式，使得计算  $A_1 A_2 \cdots A_n$  所需的标量乘法次数最少。

状态转移方程为：

$$m[i, j] = \begin{cases} 0, & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\}, & i < j \end{cases}$$

##### 3.1.1 常规方法

```
void MatrixChain(int *p, int n, int **m, int **s)
// m[i][j]: 最少乘法次数
// s[i][j]: A_i...A_j 的最优断开位置
{
    for (int i = n; i >= 1; i--) {           // 区间左端点
        for (int j = i; j <= n; j++) {       // 区间右端点
            if (i == j)
                m[i][j] = 0;                  // 单个矩阵，代价为 0
            else {
                int temp = INT_MAX;
                for (int k = i; k < j; k++) {
                    int ans = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j];
                    // p[i-1] 是 k-1 的行数；p[k] 是列，也是 j 的行数；p[j] 是 j 的列数
                    if (ans < temp) {
                        temp = ans;
                        s[i][j] = k;          // 记录最优划分点
                    }
                }
                m[i][j] = temp;
            }
        }
    }
}
```

**算法复杂度分析：**MatrixChain 的主要计算量取决于算法中对  $i, j$  和  $k$  的 3 重循环。循环体内的计算量为  $O(1)$ ，而 3 重循环的总次数为  $O(n^3)$ 。因此算法的计算时间上界为  $O(n^3)$ 。算法所占用的空间显然为  $O(n^2)$ 。

##### 3.1.2 递归方法

```
int LookupChain(int i, int j)
```

```

{
    if (m[i][j] > 0) return m[i][j];      // 已算过：直接返回（记忆化）
    if (i == j) return 0;                  // 只有一个矩阵：代价为0
    int u = LookupChain(i, i) + LookupChain(i+1, j) + p[i-1] * p[i] * p[j]; // 先假设 k=i
    s[i][j] = i;
    for (int k = i+1; k < j; k++) {        // 枚举所有划分点 k
        int t = LookupChain(i, k) + LookupChain(k+1, j) + p[i-1] * p[k] * p[j];
        if (t < u) {                      // 取最小
            u = t;
            s[i][j] = k;                  // 记录最优划分点
        }
    }
    m[i][j] = u;                         // 记忆化保存结果
    return u;
}

```

**递归复杂度：**子问题总数为  $\frac{n(n+1)}{2} = O(n^2)$ . 每个子问题在求最小值时要枚举  $k$ , 最多  $O(n)$  次, 所以总时间复杂度为  $O(n^3)$ , 空间复杂度由  $m, s$  两张表决定, 为  $O(n^2)$ .

### 3.2 最长公共子序列 (LCS)

**问题描述：**给定两个序列  $X = \langle x_1, x_2, \dots, x_m \rangle$  与  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , 求它们的最长公共子序列 (LCS) 的长度, 并输出一个 LCS。

$$c[i][j] = \begin{cases} 0, & i = 0, j = 0 \\ c[i-1][j-1] + 1, & i, j > 0; x_i = y_j \\ \max\{c[i][j-1], c[i-1][j]\}, & i, j > 0; x_i \neq y_j \end{cases}$$

```

void LCSLength(int m, int n, char *x, char *y, int **c, int **b)
{
    int i, j;
    // 初始化边界：任意序列与空序列的LCS长度为0
    for (i = 0; i <= m; i++) c[i][0] = 0;
    for (j = 0; j <= n; j++) c[0][j] = 0;
    // 动态规划填表：从小规模子问题推到大规模子问题
    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i] == y[j]) {          // xi == yj
                c[i][j] = c[i-1][j-1] + 1;
                b[i][j] = 1;              // 记录来自左上
            } else if (c[i-1][j] >= c[i][j-1]) {
                c[i][j] = c[i-1][j];
                b[i][j] = 2;              // 记录来自上
            }
        }
    }
}

```

```

    } else {
        c[i][j] = c[i][j-1];
        b[i][j] = 3; // 记录来自左
    }
}

void LCS(int i, int j, char *x, int **b)
{
    if (i == 0 || j == 0) return; // 回溯到边界，结束
    if (b[i][j] == 1) { // 来自左上：说明匹配到一个字符
        LCS(i-1, j-1, x, b);
        printf("%c", x[i]); // 输出该字符（属于LCS）
    } else if (b[i][j] == 2) { // 来自上：丢弃 xi
        LCS(i-1, j, x, b);
    } else { // 来自左：丢弃 yj
        LCS(i, j-1, x, b);
    }
}

```

**复杂度：**表格大小为  $m \times n$ , 每个格子  $O(1)$  计算, 所以时间复杂度为  $O(mn)$ , 空间复杂度为  $O(mn)$ . 回溯输出最多走  $m + n$  步, 为  $O(m + n)$ .

### 3.3 最优三角形划分 (Min-Weight Triangulation)

**问题描述：**给定一个凸多边形, 顶点按顺序编号为  $v_0, v_1, v_2, \dots, v_n$ , 其中共有  $n + 1$  个顶点。将多边形用不相交的对角线划分成若干三角形 (三角剖分)。设三角形  $(v_i, v_k, v_j)$  的代价为  $w(i, k, j)$ . 目标是找到一种三角剖分, 使总代价最小。

对所有  $i \leq k < j$  取最小即可。

$$t[i][j] = \begin{cases} 0, & i = j \\ \min_{i \leq k < j} \{t[i][k] + t[k+1][j] + w(v_{i-1}v_kv_j)\}, & i < j \end{cases}$$

```

void MinWeightTriangulation(int n, Type **t, int **s)
{
    for (int i = 1; i <= n; i++) t[i][i] = 0;
    for (int r = 2; r <= n; r++) { // 子问题长度 r: j = i + r - 1
        for (int i = 1; i <= n - r + 1; i++) {
            int j = i + r - 1;
            // 先令 k = i 作为初值 (即选三角形 (i-1, i, j))
            t[i][j] = t[i+1][j] + w(i-1, i, j);
            s[i][j] = i;
            // 枚举 i+1 ... j-1
        }
    }
}

```

```

        for (int k = i+1; k < j; k++) {
            int u = t[i][k] + t[k+1][j] + w(i-1, k, j);
            if (u < t[i][j]) {
                t[i][j] = u;
                s[i][j] = k;
            }
        }
    }
}

```

**复杂度分析：**状态数为  $O(n^2)$  (所有  $1 \leq i \leq j \leq n$ ), 每个状态需要枚举  $k$ , 最多  $O(n)$  次, 因此时间复杂度为  $O(n^3)$ . 数组  $t, s$  的规模为  $O(n^2)$ , 空间复杂度为  $O(n^2)$ .

### 3.4 多边形游戏 (Polygon Game)

**问题描述：**多边形游戏给定一个含  $n$  个顶点的环状表达式（可看作一个多边形），每条边上有一个运算符（如  $+$  或  $\times$ ），每个顶点上有一个数。通过选择删边/断开位置并确定计算顺序（等价于给表达式加括号），最终会得到一个结果值。目标通常是求最大结果（或同时求最大/最小以便处理乘法含负数的情况）。

$$\max f(i, j, s) = \begin{cases} b + d, & \text{op}[i + s] = '+' \\ \max\{ac, ad, bc, bd\}, & \text{op}[i + s] = '*' \end{cases}$$

由于最优断开位置  $s$  有  $1 \leq s \leq j-1$  的  $j-1$  种情况，由此可知

$$m[i, j, 0] = \min_{1 \leq s \leq j-1} \{ \min f(i, j, s) \}, \quad 1 \leq i, j \leq n$$

$$m[i, j, 1] = \max_{1 \leq s \leq j-1} \{ \max f(i, j, s) \}, \quad 1 \leq i, j \leq n$$

初始边界值:  $m[i, 1, 0] = v[i]$ ,  $m[i, 1, 1] = v[i]$

```
void PolyMax(int n)
{
    int minf, maxf;
    for (int j = 2; j <= n; j++) {           // 子链长度 j
        for (int i = 1; i <= n; i++) {       // 子链起点 i (环状)
            for (int s = 1; s < j; s++) {   // 断开位置 s (划分成两段)
                MinMax(n, i, s, j, minf, maxf);

                if (m[i][j][0] > minf)
```

```

m[i][j][0] = minf;           // 更新最小值

if (m[i][j][1] < maxf)
    m[i][j][1] = maxf;       // 更新最大值
}

}

int temp = m[1][n][1];
for (int i = 2; i <= n; i++) {           // 枚举起点，取全局最大
    if (temp < m[i][n][1])
        temp = m[i][n][1];
}
return temp;
}

```

**复杂度分析：**三重循环规模约为： $j : O(n)$ ,  $i : O(n)$ ,  $s : O(n)$ , 每次 MinMax 计算为常数时间，因此时间复杂度为  $O(n^3)$ , 状态数组  $m[i][j][0/1]$  的规模为  $O(n^2)$ , 空间复杂度为  $O(n^2)$ .

### 3.5 图像压缩

**问题描述：**给定灰度像素序列  $\{p_1, p_2, \dots, p_n\}$ , 其中  $0 \leq p_i \leq 255$ 。将序列划分为若干个连续段  $S_1, S_2, \dots, S_m$ 。第  $i$  段记为  $S_i = \{p_{t_{i-1}+1}, \dots, p_{t_i}\}$ , 并用同一位宽存储该段内所有像素。设该段所需位宽为  $b_i = \lceil \log_2 (\max_{t_{i-1}+1 \leq k \leq t_i} p_k + 1) \rceil$ , 且由题意有  $b_i \leq 8$ , 因此存储  $b_i$  需要 3 位；段长  $l_i = t_i - t_{i-1}$  满足  $1 \leq l_i \leq 255$ , 存储  $l_i$  需要 8 位。因此每段头部开销为  $3 + 8 = 11$  位。目标是：选择一种分段方式，使总存储位数最小，且每段长度不超过 256。

$$S[i] = \min_{1 \leq k \leq \min(i, 256)} \left\{ S[i-k] + k \cdot b_{\max}(i-k+1, i) \right\} + 11.$$

```

void Compress(int n, int p[], int S[], int L[], int b[])
{
    int Lmax = 256, header = 1;

    S[0] = 0;

    for (int i = 1; i <= n; i++) {           // 依次计算 S[1..n]
        b[i] = length(p[i]);                 // 第 i 个元素所需位数

        int bmax = b[i];
        S[i] = S[i-1] + bmax;                // 先假设最后一段只含 p[i]
        L[i] = i;

        for (int j = 2; j <= i && j <= Lmax; j++) { // j 表示最后一段长度
            if (bmax < b[i-j+1]) bmax = b[i-j+1]; // 更新该段最大位宽
        }
    }
}

```

```

    if (S[i] > S[i-j] + j * bmax) {           // 选择更优的分段
        S[i] = S[i-j] + j * bmax;
        L[i] = j;                            // 记录最后一段长度
    }
}

S[i] += header;                         // 加上该段头部开销
}
}

```

**复杂度分析：**对每个  $i$ , 内层最多枚举  $j = 1.. \min(i, L_{max})$ , 故时间复杂度为  $O(n \cdot L_{max})$ . 当  $L_{max} = 256$  为常数时, 可写为  $O(n)$ . 空间复杂度主要来自数组  $S, L, b$ , 为  $O(n)$ .

### 3.6 电路布线问题 (MNS)

**问题描述：**在电路布线问题中, 给定若干条连线, 每条连线用一对端点  $(i, \pi(i))$  表示, 其中  $i$  表示左侧端点编号,  $\pi(i)$  表示其在右侧对应的端点编号。若两条连线  $(i, \pi(i))$  与  $(t, \pi(t))$  满足  $i < t$  但  $\pi(i) > \pi(t)$ , 则称这两条连线相交。问题的目标是: 从所有连线中选取一个最大的不相交子集, 使得任意两条被选中的连线都不相交。

**边界条件：**当  $i = 1$  时

$$\text{Size}(1, j) = \begin{cases} 0, & j < \pi(1), \\ 1, & j \geq \pi(1). \end{cases}$$

**状态转移方程：**当  $i > 1$  时, 有如下递推关系:

$$\text{Size}(i, j) = \begin{cases} \text{Size}(i-1, j), & j < \pi(i), \\ \max\{\text{Size}(i-1, j), \text{Size}(i-1, \pi(i)-1) + 1\}, & j \geq \pi(i). \end{cases}$$

```

void MNS(int C[], int n, int **size)
{
    // 初始化 i=1 的情况
    for (int j = 0; j < C[1]; j++) size[1][j] = 0;
    for (int j = C[1]; j <= n; j++) size[1][j] = 1;
    // 填表
    for (int i = 2; i <= n; i++) {
        for (int j = 0; j < C[i]; j++)
            size[i][j] = size[i-1][j];
        for (int j = C[i]; j <= n; j++)
            size[i][j] = max(size[i-1][j], size[i-1][C[i]-1] + 1);
    }
    // 最终结果
    size[n][n] = max(size[n-1][n], size[n-1][C[n]-1] + 1);
}

```

textbf 最优解构造 (回溯):

```

void Traceback(int C[], int **size, int n, int Net[], int &m)
{
    int j = n;
    m = 0;
    for (int i = n; i > 1; i--) {
        if (size[i][j] != size[i-1][j]) {
            Net[m++] = i; // 选择连线 i
            j = C[i] - 1;
        }
    }
    if (j >= C[1])
        Net[m++] = 1;
}

```

**复杂度分析：**状态表规模为  $O(n^2)$ , 每个状态计算时间为  $O(1)$ , 因此时间复杂度为  $O(n^2)$ , 空间复杂度为  $O(n^2)$ . 回溯过程仅需一次线性扫描, 时间复杂度为  $O(n)$ .

### 3.7 流水作业调度（两台机器 $M_1, M_2$ ）

**问题描述：**有  $n$  个作业  $1, 2, \dots, n$ , 需要在两台机器  $M_1, M_2$  上依次加工。每个作业  $i$  必须先在  $M_1$  上加工  $a_i$  时间, 再在  $M_2$  上加工  $b_i$  时间 (不允许改变顺序)。要求安排一个作业加工顺序, 使得: 第一个作业在机器  $M_1$  上开始加工, 最后一个作业在机器  $M_2$  上完成加工的时间 (即**完工时间**) 最小。

**状态转移方程：**由流水作业调度问题的最优子结构性质可知,

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

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

其中:

- $a_j$ : 作业  $j$  在  $M_1$  的加工时间;
- $T(S - \{j\}, b_j)$ : 剩余作业子集的最优等待时间 (将  $b_j$  作为状态参数);
- $\max(t - a_j, 0)$ : 由于  $M_2$  的空闲/等待产生的额外影响项。

**结论：**流水作业调度问题在两台机器情形下具有最优子结构性质, 可用集合型动态规划描述, 其最优完成时间为  $T(N, 0)$ .

### 3.8 0-1 背包问题（特殊的整数规划问题）

**问题描述：**给定  $n$  种物品和一个背包。第  $i$  个物品的重量为  $w_i$ 、价值为  $v_i$ , 背包容量为  $C$ 。每个物品只能取 0 或 1 次 (不可分割)。目标是在不超过容量  $C$  的前提下, 使装入背包的物品总价值最大。

设决策变量

$$x_i = \begin{cases} 1, & \text{选取物品 } i, \\ 0, & \text{否则,} \end{cases} \quad i = 1, 2, \dots, n.$$

则模型为

$$\max \sum_{i=1}^n v_i x_i$$

满足约束

$$\sum_{i=1}^n w_i x_i \leq C, \quad x_i \in \{0, 1\}, \quad 1 \leq i \leq n.$$

**子问题与状态定义：**定义子问题：当可选物品为  $i, i+1, \dots, n$ , 背包容量为  $j$  ( $0 \leq j \leq C$ ) 时的最优值记为  $m(i, j)$ . 也就是说,  $m(i, j)$  表示在约束  $\sum_{k=i}^n w_k x_k \leq j$ ,  $x_k \in \{0, 1\}$  ( $i \leq k \leq n$ ) 下最大化  $\sum_{k=i}^n v_k x_k$  所能得到的最优目标值。

**边界条件：**当只剩最后一个物品  $n$  可选时,

$$m(n, j) = \begin{cases} v_n, & j \geq w_n, \\ 0, & 0 \leq j < w_n. \end{cases}$$

**状态转移方程（最优子结构）：**对  $i < n$ , 考虑物品  $i$  的“选/不选”两种情况:

$$m(i, j) = \begin{cases} \max\{m(i+1, j), m(i+1, j - w_i) + v_i\}, & j \geq w_i, \\ m(i+1, j), & 0 \leq j < w_i. \end{cases}$$

**解释：**

- 若  $j < w_i$ , 容量不足, 物品  $i$  不能选, 只能继承后续子问题:  $m(i, j) = m(i+1, j)$ ;
- 若  $j \geq w_i$ , 可选择: 不选  $i$  得  $m(i+1, j)$ ; 选  $i$  得  $m(i+1, j - w_i) + v_i$ ; 取两者最大即为最优。

**算法代码：**

```
template<class Type>
void knapsack(Type v[], int w[], int c, int n, Type **m)
{
    int jMax = min(w[n] - 1, c);

    // 初始化第 n 个物品这一行: m[n][j]
    for (int j = 0; j <= jMax; j++)
        m[n][j] = 0;
    for (int j = w[n]; j <= c; j++)
        m[n][j] = v[n];

    // 从 i=n-1 递推到 2 (最后单独处理 i=1)
    for (int i = n - 1; i > 1; i--) {
        jMax = min(w[i] - 1, c);

        // j < w[i] 时装不下物品 i, 只能继承 m[i+1][j]
        for (int j = 0; j <= jMax; j++)
            m[i][j] = m[i+1][j];
    }
}
```

```

// j >= w[i] 时, 取“选/不选”两种情况的最大值
for (int j = w[i]; j <= c; j++)
    m[i][j] = max(m[i+1][j], m[i+1][j - w[i]] + v[i]);
}

// 单独处理 i=1 (只计算 m[1][c])
m[1][c] = m[2][c];
if (c >= w[1])
    m[1][c] = max(m[1][c], m[2][c - w[1]] + v[1]);
}

```

**复杂度分析:** 需要填充大部分  $m[i][j]$  (约  $n \times c$  个状态), 每个状态  $O(1)$  转移, 因此时间复杂度为  $O(nc)$ , 空间复杂度为二维表  $O(nc)$ .

### 3.9 最优二叉搜索树 (Optimal Binary Search Tree)

**问题描述:** 给定有序关键字  $k_1 < k_2 < \dots < k_n$ 。查找过程中可能命中关键字, 也可能命中相邻关键字之间的“失败区间”。设成功查找  $k_i$  的概率 (或权值) 为  $a_i$ , 失败查找落在区间之间的概率 (或权值) 为  $b_i$  (通常  $b_0, \dots, b_n$ )。要求构造一棵二叉搜索树, 使得**平均查找代价** (期望比较次数) 最小。

```

void OptimalBinarySearchTree(int a[], int b[], int n, int **m, int **s, int **w)
{
    // 初始化空区间: m[i+1][i] = 0, 并给 w 的初值
    for (int i = 0; i <= n; i++) {
        w[i+1][i] = a[i]; // 以该课程版本的定义为准
        m[i+1][i] = 0;
    }

    // r 表示区间长度: j = i + r
    for (int r = 0; r < n; r++) {
        for (int i = 1; i <= n - r; i++) {
            int j = i + r;

            // 计算 w[i][j]
            w[i][j] = w[i][j-1] + a[j] + b[j];

            // 先令 k=i 作为初值
            m[i][j] = m[i+1][j];
            s[i][j] = i;

            // 枚举 k=i+1..j, 寻找最优根
            for (int k = i+1; k <= j; k++) {
                int t = m[i][k-1] + m[k+1][j];

```

```

        if (t < m[i][j]) {
            m[i][j] = t;
            s[i][j] = k;
        }
    }

    // 最后统一加上 w[i][j]
    m[i][j] += w[i][j];
}
}
}

```

**复杂度分析：**共有  $O(n^2)$  个区间状态  $(i, j)$ , 每个状态枚举  $k$  需要  $O(n)$ , 因此时间复杂度为  $O(n^3)$ , 空间复杂度为三张  $n \times n$  表, 为  $O(n^2)$ .

### 3.10 最大子段和 (Maximum Subarray Sum)

**问题描述：**给定长度为  $n$  的整数序列  $a[1..n]$ , 求其连续子段的最大和, 即  $\max_{1 \leq l \leq r \leq n} \sum_{k=l}^r a[k]$ .

**状态定义：**令  $b[j]$  表示以  $a[j]$  结尾的最大子段和, 则有递推:

$$b[j] = \max\{b[j-1] + a[j], a[j]\}, \quad 1 \leq j \leq n.$$

最终答案为  $\max_{1 \leq j \leq n} b[j]$ .

```

int MaxSum(int n, int *a)
{
    int sum = 0;    // 记录目前全局最大
    int b = 0;      // 记录以当前位置结尾的最大子段和

    for (int i = 1; i <= n; i++) {
        if (b > 0) b = b + a[i];
        else        b = a[i];

        if (sum < b) sum = b;
    }
    return sum;
}

```

**复杂度：**时间复杂度  $O(n)$ , 空间复杂度  $O(1)$ .

### 3.11 最大子矩阵和 (Maximum Submatrix Sum)

**问题描述：**给定一个  $m \times n$  的整数矩阵  $A$ , 求其元素和最大的子矩阵 (连续行 + 连续列)。

**核心思想：**固定子矩阵的上边界行  $i$  与下边界行  $j$  ( $i \leq j$ ), 把第  $i..j$  行之间的列元素累加成一个长度为  $n$  的数组  $b[k] = \sum_{t=i}^j A[t][k]$ . 则此时“最佳子矩阵”在列方向上等价于求数组  $b[1..n]$  的最大子段和。对所有  $(i, j)$  枚举即可求最优。

```

int MaxSum2(int m, int n, int **a)
{
    int sum = 0;
    int *b = new int[n+1];

    for (int i = 1; i <= m; i++)
        for (int k = 1; k <= n; k++)
            b[k] = 0;

    for (int j = i; j <= m; j++)
        for (int k = 1; k <= n; k++)
            b[k] += a[j][k];

    int max = MaxSum(n, b);
    if (max > sum) sum = max;
}

return sum;
}

```

**复杂度:** 共有  $O(m^2)$  组行边界  $(i,j)$ , 每次更新  $b$  需要  $O(n)$ , 求最大子段和也需要  $O(n)$ , 所以时间复杂度为  $O(m^2n)$ , 额外空间为数组  $b$ , 即  $O(n)$ .

### 3.12 最大 $m$ 子段和 (Maximum $m$ Disjoint Subarrays Sum)

**问题描述:** 给定序列  $a[1..n]$ , 要求选取恰好  $m$  段互不重叠的连续子段, 使得这些子段元素和的总和最大。

```

int MaxSum(int m, int n, int *a)
{
    if (n < m || m < 1) return 0;

    int *b = new int[n+1];
    int *c = new int[n+1];
    b[0] = 0;
    c[1] = 0;

    for (int i = 1; i <= m; i++) {
        b[i] = b[i-1] + a[i];
        c[i-1] = b[i];

        int maxv = b[i];
        for (int j = i+1; j <= n-m+i; j++) {
            b[j] = (b[j-1] > c[j-1] ? b[j-1] + a[j] : c[j-1] + a[j]);
            c[j] = b[j];
        }
    }

    return b[m];
}

```

```
c[j-1] = maxv;
if (maxv < b[j]) maxv = b[j];
}
c[i+n-m] = maxv;
}

int sum = 0;
for (int j = m; j <= n; j++)
if (sum < b[j]) sum = b[j];

return sum;
}
```

**复杂度：**两重循环规模约为  $m \times n$ , 因此时间复杂度为  $O(mn)$ , 额外空间为  $b, c$  两个数组, 为  $O(n)$ .

## 4 贪心算法

贪心算法在求解优化问题时，每一步都作出一个当前看来“最优”的选择，期望通过一系列局部最优决策得到全局最优解。贪心算法的关键在于：问题必须具有贪心选择性质和最优子结构。

### 4.1 活动选择问题

**问题描述：**设有  $n$  个活动  $A_1, A_2, \dots, A_n$ ，每个活动  $A_i$  有开始时间  $s_i$  和结束时间  $f_i$ ，同一时间只能进行一个活动。目标是从中选择尽可能多的互不冲突的活动。

**贪心策略：**按活动的结束时间从小到大排序；每次选择当前结束时间最早且与已选活动不冲突的活动。该策略可以保证给后续活动留下尽可能多的时间，因此是全局最优的。

```
void GreedySelector(int n, int s[], int f[], bool A[])
{
    A[1] = true; // 选择第一个活动
    int j = 1;

    for (int i = 2; i <= n; i++) {
        if (s[i] >= f[j]) {
            A[i] = true; // 选择活动 i
            j = i;
        } else {
            A[i] = false;
        }
    }
}
```

**时间复杂度：**若活动已排序，则算法时间复杂度为  $O(n)$ 。若需要先排序，则总体复杂度为  $O(n \log n)$ 。

### 4.2 分数背包问题 (Fractional Knapsack)

**问题描述：**给定  $n$  个物品，第  $i$  个物品重量为  $w_i$ ，价值为  $v_i$ ，背包容量为  $M$ 。与 0-1 背包不同，每个物品允许取任意比例。目标是在不超过容量  $M$  的前提下，使装入背包的总价值最大。

**贪心策略：**按物品的单位重量价值  $\frac{v_i}{w_i}$  从大到小排序；依次尽可能多地装入单位价值最高的物品；若剩余容量不足以装入整个物品，则装入该物品的一部分并结束。该策略能保证每一步都优先获得最大“单位收益”，因此可以得到全局最优解。

```
void Knapsack(int n, float M, float v[], float w[], float x[])
{
    Sort(n, v, w); // 按 v[i]/w[i] 从大到小排序

    for (int i = 1; i <= n; i++)
        x[i] = 0;

    float c = M;
```

```

int i;

for (i = 1; i <= n; i++) {
    if (w[i] > c) break; // 剩余容量不足, 无法装入整个物品 i
    x[i] = 1; // 装入整个物品 i
    c -= w[i];
}

if (i <= n) x[i] = c / w[i]; // 装入物品 i 的一部分
}

```

**时间复杂度:** 排序耗时  $O(n \log n)$ , 装包过程为  $O(n)$ , 因此总体时间复杂度为  $O(n \log n)$ .

### 4.3 最优装载问题 (Optimal Loading)

**问题描述:** 有  $n$  个货箱, 第  $i$  个货箱的重量为  $w_i$ , 现有一艘载重能力为  $C$  的船。要求选择若干个货箱装入船中, 使得装入货箱的数量尽可能多, 且总重量不超过船的载重能力  $C$ 。

**贪心策略:** 优先装载重量较小的货箱; 每次选择当前剩余货箱中重量最小者装入船中; 当剩余载重不足以装下下一个货箱时, 算法结束。该策略保证在相同载重条件下, 能够装入尽可能多的货箱, 因此是最优的。

```

void Loading(int x[], int w[], int C, int n)
{
    int *t = new int[n+1]; // t[i] 存放排序后的货箱下标

    Sort(w, t, n); // 按重量 w 从小到大排序, 结果存入 t

    for (int i = 1; i <= n; i++)
        x[i] = 0; // 初始化: 不装任何货箱

    for (int i = 1; i <= n && w[t[i]] <= C; i++) {
        x[t[i]] = 1; // 装入编号为 t[i] 的货箱
        C -= w[t[i]]; // 更新剩余载重
    }
}

```

**时间复杂度:** 排序过程需要  $O(n \log n)$  时间, 装载过程最多遍历一次货箱, 为  $O(n)$ , 因此算法的总体时间复杂度为  $O(n \log n)$ .

**空间复杂度:** 额外使用了数组  $t$  和  $x$ , 空间复杂度为  $O(n)$ .

### 4.4 哈夫曼编码 (Huffman Coding)

**问题描述:** 给定一组字符及其出现频率 (或权值)  $\{c_1, c_2, \dots, c_n\}$ , 要求为每个字符设计一个二进制编码, 使得: 编码满足前缀码性质 (任一字符的编码不是另一个字符编码的前缀); 所有字符编码后的加权路径长度 (**WPL**) 最小。

其中，加权路径长度定义为  $WPL = \sum_{i=1}^n w_i \cdot l_i$ ,  $w_i$  为字符  $i$  的频率（权值）， $l_i$  为其编码长度。

**贪心策略：**在当前所有结点中，反复选取权值最小的两个结点；将这两个结点合并为一个新结点，新结点的权值为二者权值之和；将新结点重新插入集合中，重复上述过程，直到只剩一个结点。该策略保证每一步的局部最优选择（合并最小权值结点）最终得到全局最优的哈夫曼树。

```
void Huffman(int w[], int n)
{
    MinHeap<HuffmanNode> Q;

    Initialize(Q, w, n); // 将 n 个权值初始化到最小堆中

    HuffmanNode x, y, z;

    for (int i = 1; i < n; i++) {
        Q.deleteMin(x); // 取出权值最小的结点 x
        Q.deleteMin(y); // 取出权值次小的结点 y

        z.MakeTree(x, y); // 合并 x 和 y 为一棵新树
        z.weight = x.weight + y.weight;

        Q.insert(z); // 将新结点插回最小堆
    }
}
```

**时间复杂度：**最小堆初始化需要  $O(n)$  时间，合并过程中共进行  $n - 1$  次循环，每次包含两次删除最小值和一次插入操作，每个堆操作时间为  $O(\log n)$ ，因此总时间复杂度为  $O(n \log n)$ .

## 4.5 单源最短路径问题 (Dijkstra 算法)

**问题描述：**给定一个带非负权值的有向图（或无向图） $G = (V, E)$ ，其中每条边  $(u, v)$  的权值为  $c(u, v) \geq 0$ 。指定一个源点  $v \in V$ ，要求计算从源点  $v$  到图中其余各顶点的最短路径长度，并可同时记录对应的最短路径。

```
void Dijkstra(int n, int v, Type c[][] [MAXN], Type dist[], int prev[])
{
    bool S[MAXN];

    for (int i = 1; i <= n; i++) {
        dist[i] = c[v][i]; // 初始化距离
        S[i] = false; // 初始时所有顶点均未加入 S
        if (dist[i] < maxint) prev[i] = v;
        else prev[i] = -1;
    }

    dist[v] = 0;
```

```

S[v] = true;           // 源点加入 S

for (int i = 1; i < n; i++) {
    Type temp = maxint;
    int u = v;

    // 在 V-S 中寻找 dist 最小的顶点 u
    for (int j = 1; j <= n; j++) {
        if (!S[j] && dist[j] < temp) {
            u = j;
            temp = dist[j];
        }
    }

    S[u] = true;           // 将 u 加入 S

    // 用 u 松弛其邻接点
    for (int j = 1; j <= n; j++) {
        if (!S[j] && c[u][j] < maxint) {
            Type newdist = dist[u] + c[u][j];
            if (newdist < dist[j]) {
                dist[j] = newdist;
                prev[j] = u;
            }
        }
    }
}
}

```

**时间复杂度：**在该实现中每一轮需在  $O(n)$  时间内寻找最小的  $dist$ ; 共进行  $n - 1$  轮。因此总时间复杂度为  $O(n^2)$ .

## 4.6 最小生成树 (Minimum Spanning Tree)

给定一个连通无向带权图  $G = (V, E)$ , 其中每条边  $(u, v)$  具有权值  $w(u, v)$ 。最小生成树 (MST) 是指一棵：包含图中所有顶点；边数为  $|V| - 1$ ；总权值之和最小的生成树。

下面介绍两种经典的贪心算法：**Prim 算法**和**Kruskal 算法**。

### 4.6.1 Prim 算法

**基本思想：**Prim 算法从某个起始顶点出发，逐步扩展一棵生成树。在每一步中，选择一条连接当前生成树与外部顶点的最小权值边，将该边及其对应的顶点加入生成树。

```

void Prim(int n, Type c[][] [MAXN])
{

```

```

bool S[MAXN];
Type lowcost[MAXN];
int closest[MAXN];

// 初始化: 从顶点 1 开始
for (int i = 1; i <= n; i++) {
    S[i] = false;
    lowcost[i] = c[1][i];
    closest[i] = 1;
}
S[1] = true;

for (int i = 1; i < n; i++) {
    Type min = maxint;
    int j = 1;

    // 选择距离生成树最近的顶点 j
    for (int k = 2; k <= n; k++) {
        if (!S[k] && lowcost[k] < min) {
            min = lowcost[k];
            j = k;
        }
    }
    S[j] = true; // 将顶点 j 加入生成树

    // 更新其余顶点到生成树的最小距离
    for (int k = 2; k <= n; k++) {
        if (!S[k] && c[j][k] < lowcost[k]) {
            lowcost[k] = c[j][k];
            closest[k] = j;
        }
    }
}
}

```

**时间复杂度:** 该实现中每次选择最小边需  $O(n)$ , 共进行  $n - 1$  次, 因此时间复杂度为  $O(n^2)$ 。

#### 4.6.2 Kruskal 算法

**基本思想:** Kruskal 算法从边的角度构造最小生成树: 将所有边按权值从小到大排序; 依次选择当前权值最小且不会形成回路的边; 直到选取  $|V| - 1$  条边为止。

```

void Kruskal(int n, int m, Edge edges[])
{

```

```

MinHeap<Edge> H;
UnionFind U(n);

for (int i = 1; i <= m; i++)
    H.insert(edges[i]); // 将所有边加入最小堆

int k = 0; // 已选边数
while (k < n-1 && !H.empty()) {
    Edge x;
    H.deleteMin(x); // 取出权值最小的边

    int a = U.Find(x.u);
    int b = U.Find(x.v);

    if (a != b) { // 不形成回路
        k++;
        U.Union(a, b); // 合并两个连通分量
    }
}
}

```

**时间复杂度：**边排序（或最小堆）需要  $O(m \log m)$ ；并查集操作近似为  $O(1)$ ；因此总体时间复杂度为  $O(m \log m)$ 。

**算法比较：**Prim 算法适合稠密图；Kruskal 算法适合稀疏图；二者均为贪心算法，且都能正确求得最小生成树。

## 4.7 多机调度问题 (Multi-machine Scheduling)

**问题描述：**设有  $n$  个相互独立的作业  $J_1, J_2, \dots, J_n$ ，以及  $m$  台相同的机器  $M_1, M_2, \dots, M_m$ 。第  $i$  个作业的处理时间为  $p_i$ 。每个作业可以在任意一台机器上加工，但同一时刻一台机器只能加工一个作业，且作业一旦开始加工不能中断。目标是：合理安排作业顺序与分配方案，使所有作业完成的总时间（完工时间，makespan）最小。

该问题是一个经典的 NP-完全问题，目前不存在多项式时间的精确算法，通常采用贪心策略设计近似算法。

**贪心策略（最长作业优先，LPT）：**将所有作业按处理时间  $p_i$  从大到小排序；依次将当前作业分配给当前负载最小（最早空闲）的机器。该策略的直观思想是：优先处理耗时最长的作业，避免其被推迟到后期造成整体完成时间过大。

**算法步骤说明：**

- 当  $n \leq m$  时，可直接将每个作业分配给一台机器，总完成时间为  $\max\{p_1, p_2, \dots, p_n\}$ ；
- 当  $n > m$  时：
  1. 按作业处理时间从大到小排序；
  2. 维护每台机器的当前完成时间；

3. 每次选择当前完成时间最小的机器分配下一个作业。

**时间复杂度：**

- 作业排序需要  $O(n \log n)$ ;
- 作业分配过程可在  $O(n \log m)$  (使用优先队列) 或  $O(nm)$  时间内完成;

因此整体时间复杂度为  $O(n \log n)$ .

## 5 回溯法

### 5.1 回溯法的基本框架

#### 5.1.1 递归回溯

**思想说明：**回溯法对解空间树进行深度优先搜索。在一般情况下，回溯法采用递归方式实现。搜索过程中，逐层构造解向量，当到达叶结点时输出一个解；若当前部分解违反约束或不可能得到更优解，则进行剪枝。

```
void backtrack(int t)
{
    if (t > n) output(x);
    else
        for (int i = f(n,t); i <= g(n,t); i++) {
            x[t] = h(i);
            if (constraint(t) && bound(t))
                backtrack(t + 1);
        }
}
```

**复杂度分析：**回溯法在最坏情况下需要遍历整个解空间树。若每一层有  $b$  个分支、深度为  $n$ ，则最坏时间复杂度为  $O(b^n)$ 。实际运行中，由于约束函数和限界函数的剪枝作用，搜索规模通常会显著减小。

#### 5.1.2 迭代回溯

**思想说明：**递归回溯可以用非递归方式实现，即采用树的非递归深度优先遍历方法。迭代回溯通过显式维护层号  $t$ ，模拟递归调用与回退过程。

```
void iterativeBacktrack()
{
    int t = 1;
    while (t > 0) {
        if (f(n,t) <= g(n,t))
            for (int i = f(n,t); i <= g(n,t); i++) {
                x[t] = h(i);
                if (constraint(t) && bound(t)) {
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else t--;
    }
}
```

**复杂度分析：**迭代回溯与递归回溯本质上遍历的是同一棵解空间树，因此在最坏情况下时间复杂度相同，仍为指数级  $O(b^n)$ ，仅实现方式不同。

### 5.1.3 子集树与排列树

**子集树：**子集树用于描述每个元素只有“取 / 不取”两种选择的问题，例如 0-1 背包、装载问题、最大团问题等。解空间是一棵二叉树。

```
void backtrack(int t)
{
    if (t > n) output(x);
    else
        for (int i = 0; i <= 1; i++) {
            x[t] = i;
            if (legal(t)) backtrack(t + 1);
        }
}
```

**复杂度分析：**子集树共有  $2^n$  个叶结点，因此遍历子集树的时间复杂度为  $O(2^n)$ .

**排列树：**排列树用于描述需要枚举元素排列顺序的问题，例如旅行售货员问题、批处理作业调度问题等。解空间是一棵多叉树。

```
void backtrack(int t)
{
    if (t > n) output(x);
    else
        for (int i = t; i <= n; i++) {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t + 1);
            swap(x[t], x[i]);
        }
}
```

**复杂度分析：**排列树共有  $n!$  个叶结点，因此遍历排列树的时间复杂度为  $O(n!)$ .

## 5.2 装载问题（回溯法）

**问题描述：**有一批共  $n$  个集装箱，需要装载到两艘轮船上，两艘轮船的载重量分别为  $c_1$  和  $c_2$ 。第  $i$  个集装箱的重量为  $w_i$ ，且满足  $\sum_{i=1}^n w_i \leq c_1 + c_2$ . 装载问题要求判断是否存在一种合理的装载方案，使得所有集装箱均能被装入这两艘轮船中。

**问题转化：**若第一艘轮船尽可能多地装载集装箱，剩余集装箱自然装入第二艘轮船。因此，该问题可转化为：从  $n$  个集装箱中选择一个子集，使其总重量不超过  $c_1$ ，且尽可能接近  $c_1$ 。这等价于一个特殊的 0-1 背包问题，可采用回溯法求解。

```
void backtrack(int i)
{
```

```

if (i > n) {           // 到达叶结点
    bestw = cw;         // 更新最优解
    return;
}

r -= w[i];             // 更新剩余重量

if (cw + w[i] <= c) { // 搜索左子树 (装第 i 个)
    x[i] = 1;
    cw += w[i];
    backtrack(i + 1);
    cw -= w[i];
}

if (cw + r > bestw) { // 搜索右子树 (不装第 i 个)
    x[i] = 0;
    backtrack(i + 1);
}

r += w[i];             // 回退到上一层
}

```

**复杂度分析：**装载问题的解空间为一棵子集树，共有  $2^n$  个叶结点。在最坏情况下，回溯算法需要遍历整个解空间，其时间复杂度为  $O(2^n)$ 。但通过可行性约束和上界函数进行剪枝，实际搜索规模通常远小于  $2^n$ 。在一般情况下，若  $n$  较大且要求多次求解，可采用动态规划算法求解。

### 5.3 批处理作业调度问题（回溯法）

**问题描述：**给定  $n$  个作业的集合  $\{J_1, J_2, \dots, J_n\}$ ，每个作业必须先在机器 1 上加工，然后在机器 2 上加工。作业  $J_i$  在机器  $k$  上的处理时间记为  $M[i][k]$ 。对于一个确定的作业调度顺序，设  $f_2$  为所有作业在机器 2 上完成加工的总时间，则该调度的完成时间为  $f_2$ 。

**目标：**确定一种作业加工顺序，使得所有作业在机器 2 上完成的总时间最小。

```

void Flowshop::Backtrack(int i)
{
    if (i > n) {
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestf = f;
    }
    else
        for (int j = i; j <= n; j++) {
            f1 += M[x[j]][1];
            f2[i] = (f2[i-1] > f1 ? f2[i-1] : f1) + M[x[j]][2];
        }
}

```

```

f += f2[i];

if (f < bestf) {
    swap(x[i], x[j]);
    Backtrack(i + 1);
    swap(x[i], x[j]);
}

f1 -= M[x[j]][1];
f -= f2[i];
}
}

```

**复杂度分析：**批处理作业调度问题的解空间是一棵排列树，最多包含  $n!$  个叶结点。在最坏情况下，回溯算法需要遍历整个排列空间，时间复杂度为  $O(n!)$ 。通过限界函数进行剪枝，可以在实际运行中大幅减少搜索结点数，但该问题本质上仍是一个 **NP-难问题**。

#### 5.4 符号三角形问题（回溯法）

**问题描述：**符号三角形由符号“+”和“-”组成。设第一行有  $n$  个符号，其余各行由上一行相邻两个符号决定：若两个符号相同，则下一行对应位置为“+”；若两个符号不同，则下一行对应位置为“-”。整个三角形共有  $\frac{n(n+1)}{2}$  个符号。符号三角形问题要求：对于给定的  $n$ ，计算有多少种不同的符号三角形，使得其中“+”和“-”的个数相同。

```

void Triangle::Backtrack(int t)
{
    if ((count > half) || (t*(t+1)/2 - count > half))
        return;

    if (t > n)
        sum++;
    else
        for (int i = 0; i < 2; i++) { // {0,1} -> {+, -}
            p[1][t] = i;
            count += i;

            for (int j = 2; j <= t; j++) {
                p[j][t-j+1] = p[j-1][t-j+1] + p[j-1][t-j+2];
                count += p[j][t-j+1];
            }

            Backtrack(t + 1);

            for (int j = 2; j <= t; j++)

```

```

    count -= p[j][t-j+1];

    count -= i;
}
}

```

**复杂度分析：**该问题的解空间为一棵子集树，规模为  $2^n$ ；每一次可行性判断与符号生成的代价为  $O(n)$ ；因此在最坏情况下，算法时间复杂度为  $O(n2^n)$ 。通过可行性约束进行剪枝，可以显著减少实际搜索规模，但该问题本质上仍属于指数级复杂度问题。

## 5.5 n 后问题（回溯法）

**问题描述：**在一个  $n \times n$  的棋盘上放置  $n$  个皇后，使得任意两个皇后之间都不能互相攻击。按照国际象棋规则，皇后可以攻击与其处在同一行、同一列或同一条对角线上的棋子。 $n$  后问题要求计算在棋盘上放置  $n$  个皇后的所有可行方案数。

可行性检测函数：

```

bool Queen::Place(int k)
{
    for (int j = 1; j < k; j++)
        if ((abs(x[j] - x[k]) == abs(j - k)) || (x[j] == x[k]))
            return false;
    return true;
}

```

回溯算法代码：

```

void Queen::Backtrack(int t)
{
    if (t > n)
        sum++;
    else
        for (int i = 1; i <= n; i++) {
            x[t] = i;
            if (Place(t))
                Backtrack(t + 1);
        }
}

```

**复杂度分析：** $n$  后问题的解空间是一棵排列树，最多包含  $n!$  个叶结点。在最坏情况下，回溯算法需要遍历整个排列空间，其时间复杂度为  $O(n!)$ 。通过可行性检测函数进行剪枝，可以显著减少实际搜索规模，但该问题仍属于指数时间复杂度问题。

## 5.6 0-1 背包问题（回溯法）

**问题描述：**给定  $n$  个物品，第  $i$  个物品的重量为  $w_i$ ，价值为  $p_i$ ，背包容量为  $c_1$ 。每个物品只能选择“装入”或“不装入”一次。目标是在不超过背包容量的前提下，使装入物品的总价值最大。

回溯算法代码:

```
template<class Typew, class Typep>
void Knap<Typew, Typep>::Backtrack(int i)
{
    if (i > n) {
        bestp = cp;
        return;
    }

    if (cw + w[i] <= c) {
        cw += w[i];
        cp += p[i];
        Backtrack(i + 1);
        cw -= w[i];
        cp -= p[i];
    }

    if (Bound(i + 1) > bestp)
        Backtrack(i + 1);
}
```

上界函数代码:

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i)
{
    Typew cleft = c - cw;
    Typep b = cp;

    while (i <= n && w[i] <= cleft) {
        cleft -= w[i];
        b += p[i];
        i++;
    }

    if (i <= n)
        b += p[i] / w[i] * cleft;

    return b;
}
```

**复杂度分析:** 0-1 背包问题的解空间为一棵子集树, 最坏情况下共有  $2^n$  个叶结点。回溯算法在最坏情况下需要遍历整个解空间, 时间复杂度为  $O(2^n)$ . 通过上界函数进行剪枝, 可以显著减少实际搜索规模, 但该问题本质上仍属于指数时间复杂度问题。

## 5.7 最大团问题（回溯法）

**问题描述：**给定无向图  $G = (V, E)$ 。若  $U \subseteq V$ , 且对任意  $u, v \in U$ , 都有  $(u, v) \in E$ , 则称  $U$  为图  $G$  的一个团。若  $U$  不包含于任何更大的团中, 则称其为极大团。图  $G$  中顶点数最多的团称为最大团。最大团问题要求找出图  $G$  的一个最大团。

```
void Clique::Backtrack(int i)
{
    if (i > n) {
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        return;
    }

    bool OK = true;
    for (int j = 1; j < i; j++)
        if (x[j] && !a[i][j]) {
            OK = false;
            break;
        }

    if (OK) {
        x[i] = 1;
        cn++;
        Backtrack(i + 1);
        cn--;
    }

    if (cn + n - i > bestn) {
        x[i] = 0;
        Backtrack(i + 1);
    }
}
```

**复杂度分析：**最大团问题的解空间规模为  $2^n$ 。在最坏情况下, 回溯算法需要遍历整个子集树, 其时间复杂度为  $O(2^n)$ 。该问题是典型的 NP-难问题。

## 5.8 图的 $m$ 着色问题（回溯法）

**问题描述：**给定无向连通图  $G = (V, E)$  和  $m$  种不同的颜色, 要求用这  $m$  种颜色对图中每个顶点着色, 使得任意一条边的两个端点颜色不同。判断图是否存在一种合法的  $m$  着色方案。

回溯算法代码:

```
void Color::Backtrack(int t)
{
```

```

if (t > n)
    sum++;
else
    for (int i = 1; i <= m; i++) {
        x[t] = i;
        if (OK(t))
            Backtrack(t + 1);
    }
}

```

可行性检测函数：

```

bool Color::OK(int k)
{
    for (int j = 1; j < k; j++)
        if (a[k][j] && x[j] == x[k])
            return false;
    return true;
}

```

**复杂度分析：**图的  $m$  着色问题的解空间为  $m^n$ 。在最坏情况下，回溯算法需要遍历整个解空间，时间复杂度为  $O(m^n)$ 。该问题是 NP-完全问题。

## 5.9 旅行售货员问题（回溯法）

**问题描述：**设有  $n$  个城市，已知任意两个城市之间的距离。旅行售货员从某一城市出发，需要恰好访问每个城市一次，并最终返回出发城市，要求使得总旅行距离最短。

```

template<class Type>
void Traveling<Type>::Backtrack(int i)
{
    if (i == n) {
        if (a[x[n-1]][x[n]] != NoEdge &&
            a[x[n]][x[1]] != NoEdge &&
            (cc + a[x[n-1]][x[n]] + a[x[n]][x[1]] < bestc
             || bestc == NoEdge)) {
            bestc = cc + a[x[n-1]][x[n]] + a[x[n]][x[1]];
        }
    }
    else
        for (int j = i; j <= n; j++) {
            if (a[x[i-1]][x[j]] != NoEdge &&
                (cc + a[x[i-1]][x[j]] < bestc
                 || bestc == NoEdge)) {
                swap(x[i], x[j]);
                cc += a[x[i-1]][x[i]];
            }
        }
}

```

```

        Backtrack(i + 1);
        cc -= a[x[i-1]][x[i]];
        swap(x[i], x[j]);
    }
}
}
}

```

**复杂度分析：**旅行售货员问题的解空间规模为  $(n - 1)!$ 。在最坏情况下，回溯算法需要遍历所有排列，时间复杂度为  $O(n!)$ 。该问题是经典的 NP-难问题。

## 5.10 圆排列问题（回溯法）

**问题描述：**给定  $n$  个大小不等的圆，其半径分别为  $c_1, c_2, \dots, c_n$ 。现要求将这  $n$  个圆排列在一条水平直线上，使所有圆都与水平线相切，且整体排列所占的长度最小。圆排列问题要求确定一种排列顺序，使得最左端到最右端的距离最小。

回溯算法代码：

```

void Circle::Backtrack(int t)
{
    if (t > n)
        Compute();
    else
        for (int j = t; j <= n; j++) {
            swap(r[t], r[j]);
            if (Center(t) < best)
                Backtrack(t + 1);
            swap(r[t], r[j]);
        }
}

```

圆心计算函数：

```

float Circle::Center(int t)
{
    float temp = 0;
    for (int j = 1; j < t; j++) {
        float value = x[j] + 2.0 * sqrt(r[j] * r[t]);
        if (value > temp)
            temp = value;
    }
    x[t] = temp;

    float low = 0, high = 0;
    for (int i = 1; i <= t; i++) {
        if (x[i] - r[i] < low)
            low = x[i] - r[i];
    }
}

```

```

    if (x[i] + r[i] > high)
        high = x[i] + r[i];
    }
    if (high - low < best)
        best = high - low;

    return x[t];
}

```

**复杂度分析：**圆排列问题的解空间为一棵排列树，最坏情况下需要枚举  $n!$  种排列。在每个结点需要  $O(n)$  时间计算排列长度，因此算法在最坏情况下的时间复杂度为  $O(n \cdot n!)$ .

## 5.11 连续邮资问题（回溯法）

**问题描述：**假设国家发行了  $n$  种不同面值的邮票，并规定每封信封上最多允许贴  $m$  张邮票。连续邮资问题要求确定这  $n$  种邮票的面值，使得在最多贴  $m$  张邮票的条件下，从 1 开始可以连续表示的邮资区间尽可能大。

```

void Stamp::Backtrack(int t)
{
    if (t > n) {
        if (r[n] > maxvalue) {
            maxvalue = r[n];
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
        }
        return;
    }

    int *z = new int[maxx + 1];
    for (int i = 1; i <= maxx; i++)
        z[i] = y[i];

    for (int i = x[t-1] + 1; i <= r[t-1] + 1; i++) {
        x[t] = i;
        for (int k = 1; k <= maxx; k++)
            y[k] = z[k];

        Update(t);
        Backtrack(t + 1);
    }

    for (int k = 1; k <= maxx; k++)
        y[k] = z[k];
}

```

```
    delete[] z;  
}
```

**复杂度分析：**连续邮资问题的解空间规模随  $n$  和  $m$  急剧增长，在最坏情况下，回溯算法需要枚举大量组合，时间复杂度为指数级。