数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Chapter 1 计算复杂性理论

1.1 时间复杂度

1.1.1 输入规模

算法的时间复杂度是针对指定基本运算，计算算法所做的运算次数。其中基本运算指的是比较、加法、乘法、置指针、交换等操作。

算法基本运算可以表示为跟输入规模相关的函数。常见的输入规模有数组的元素个数、调度问题的任务个数、图的顶点数和边数等。对于相同输入规模的不同实例，算法的基本运算次数有可能会不一样。

对于排序算法，输入规模为数组的元素个数，基本运算为元素之间的比较。

对于整数乘法，m 位整数与 n 位整数相乘需要做 $m \times n$ 次乘法。

对于矩阵乘法，$i \times j$ 矩阵与 $j \times k$ 矩阵相乘需要做 $i \times j \times k$ 次乘法。

1.1.2 时间复杂度

最好情况时间复杂度是指算法在最理想情况下的时间复杂度。例如在查找算法中，目标元素刚好在数组的第一个位置，那么只需要一次就能找到，时间复杂度是常量阶 $O(1)$。

最坏情况时间复杂度 $W(n)$ 是指算法在最坏情况下执行的时间复杂度。例如目标元素在数组最后一个位置或者不在数组中，那么需要遍历完整个数组才能得出结果，时间复杂度为 $O(n)$。

平均情况时间复杂度 $A(n)$ 是指用算法在所有情况下执行的次数的加权平均值表示，也就是算法在求解这类问题所需要的平均时间。

假设 S 是规模 n 为实例集，实例的 $i \in S$ 概率是 p_i，算法对实例 i 执行的基本运算次数是 t_i：

$$A(n) = \sum_{i \in S} p_i t_i$$

例如，利用顺序查找算法在一个长度为 n 的数组中查找元素 x。假设 x 在数组中的概率是 p（x 不在数组中的概率为 $1-p$），且每个位置概率相等：

$$\begin{aligned} A(n) &= \sum_{i=1}^{n} i\frac{p}{n} + (1-p)n \\ &= \frac{p(n+1)}{2} + (1-p)n \\ \text{当} p = \frac{1}{2}, & \\ &= \frac{n+1}{4} + \frac{n}{2} \\ &= \frac{3}{4}n \end{aligned}$$

1.1.3 大 O 符号

设 f 和 g 是定义域为自然数集 $\mathbb{N}$ 上的函数，若存在正数 c 和 n_0，使得一切 $n \geq n_0$ 满足

$$0 \leq f(n) \leq cg(n)$$

则称 $f(n)$ 的渐进上界是 $g(n)$，即 $f(n)$ 的阶不高于 $g(n)$ 的阶，记作：

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

大 O 符号

$$f(n) = n^2 + n$$

$$f(n) = O(n^2)$$

$$f(n) = O(n^3)$$

1.1.4 大 Ω 符号

设 f 和 g 是定义域为自然数集 $\mathbb{N}$ 上的函数，若存在正数 c 和 n_0，使得一切 $n \geq n_0$ 满足

$$0 \leq cg(n) \leq f(n)$$

则称 $f(n)$ 的渐进下界是 $g(n)$，即 $f(n)$ 的阶不低于 $g(n)$ 的阶，记作：

$$f(n) = \Omega(g(n))$$

大 Ω 符号

$$f(n) = n^2 + n$$
$$f(n) = \Omega(n^2)$$
$$f(n) = \Omega(100n)$$

1.1.5 小 o 符号

设 f 和 g 是定义域为自然数集 $\mathbb{N}$ 上的函数，若对于任意正数 c 都存在 n_0，使得一切 $n \geq n_0$ 满足

$$0 \leq f(n) < cg(n)$$

则称 $f(n)$ 的阶低于 $g(n)$ 的阶，记作：

$$f(n) = o(g(n))$$

小 o 符号

$$f(n) = n^2 + n$$
$$f(n) = o(n^3)$$

1.1.6 小 ω 符号

设 f 和 g 是定义域为自然数集 $\mathbb{N}$ 上的函数，若对于任意正数 c 都存在 n_0，使得一切 $n \geq n_0$ 满足

$$0 \leq cg(n) < f(n)$$

则称 $f(n)$ 的阶高于 $g(n)$ 的阶，记作：

$$f(n) = \omega(g(n))$$

小 ω 符号

$$f(n) = n^2 + n$$
$$f(n) = \omega(n)$$

1.1.7 Θ 符号

若 $f(n) = O(g(n))$ 且 $f(n) = \Omega(g(n))$，则称 $f(n)$ 的阶与 $g(n)$ 的阶相等，记作：

$$f(n) = \Theta(g(n))$$

Θ 符号

$$f(n) = n^2 + n$$
$$g(n) = 100n^2$$
$$f(n) = \Theta(g(n))$$

1.1.8 定理（Theorem）

1. 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)}$ 存在，并且等于某个常数 $c > 0$，那么 $f(n) = \Theta(g(n))$。
2. 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$，那么 $f(n) = o(g(n))$。
3. 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$，那么 $f(n) = \omega(g(n))$。
4. 多项式函数的阶低于指数函数的阶，即 $n^d = o(r^n)$，$r > 1$，$d > 0$。

5. 对数函数的阶低于幂函数的阶，即 $ln(n) = o(n^d),\ d > 0$。

6. 函数的阶之间的关系具有传递性：

   - 如果 $f = O(g),\ g = O(h)$，那么 $f = O(h)$。
   - 如果 $f = \Omega(g),\ g = \Omega(h)$，那么 $f = \Omega(h)$。
   - 如果 $f = \Theta(g),\ g = \Theta(h)$，那么 $f = \Theta(h)$。

证明

$f(n) = \frac{1}{2}n^2 - 3n$，证明 $f(n) = \Theta(n^2)$

$$
\begin{aligned}
&\lim_{n\to\infty} \frac{f(n)}{n^2} \\
&= \lim_{n\to\infty} \frac{\frac{1}{2}n^2 - 3n}{n^2} \\
&= \frac{1}{2}
\end{aligned}
$$

证明

证明多项式函数的阶低于指数函数的阶。

$$
\begin{aligned}
&\lim_{n\to\infty} \frac{n^d}{r^n} \\
&= \lim_{n\to\infty} \frac{dn^{d-1}}{r^n ln(r)} \\
&= \lim_{n\to\infty} \frac{d(d-1)n^{d-2}}{r^n ln(r)^2} \\
&= \dots \\
&= \lim_{n\to\infty} \frac{d!}{r^n ln(r)^d} \\
&= 0
\end{aligned}
$$

证明

证明对数函数的阶低于幂函数的阶。

$$
\begin{aligned}
&\lim_{n\to\infty} \frac{ln(n)}{n^d} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n}}{dn^{d-1}} \\
&= \lim_{n\to\infty} \frac{1}{dn^d} \\
&= 0
\end{aligned}
$$

排序

$$
\begin{aligned}
f(n) &= (n^2+n)/2 \\
g(n) &= 10n \\
h(n) &= 1.5^n \\
t(n) &= n^{\frac{1}{2}}
\end{aligned}
$$

按照阶从高到低排序。

$$
\begin{aligned}
h(n) &= \omega(f(n)) \\
f(n) &= \omega(g(n)) \\
g(n) &= \omega(t(n)) \\
h(n) &< f(n) < g(n) < t(n)
\end{aligned}
$$

1.2 均摊时间复杂度

1.2.1 均摊时间复杂度（Amortized Time Complexity）

均摊时间复杂度也称摊还分析或分摊分析，均摊复杂度是一个更加高级的概念，它是一种特殊的情况，应用的场景也更加特殊和有限。

```
1  void insert(int val) {
2      if(cnt == arr.length) {
3          int sum = 0;
4          for(int i = 0; i < arr.length; i++) {
5              sum += arr[i];
6          }
7          arr[0] = sum;
8          cnt = 1;
9      }
10     arr[cnt++] = val;
11 }
```

这段代码实现了一个往数组中插入数据的功能。当数组元素满时，就遍历数组求和，将元素和保存到数组的第 0 个位置，并清空数组，然后再将新的数据插入。但如果数组一开始就有空闲空间，则直接将数据插入数组。

最理想的情况下，数组中有空闲空间，最好情况时间复杂度为 $O(1)$；最坏的情况下，数组中没有空闲空间了，需要先做一次遍历求和，然后再将数据插入，所以最坏情况时间复杂度为 $O(n)$。

假设数组的长度是 n，根据数据插入的位置的不同，可以分为 n 种情况，每种情况的时间复杂度是 $O(1)$。除此之外，还有一种情况，就是在数组没有空闲空间时插入一个数据，这个时候的时间复杂度是 $O(n)$。这 n + 1 种情况发生的概率一样，都是 $\frac{1}{n+1}$。

根据加权平均的计算方法，求得的平均时间复杂度：

$$
\begin{aligned}
&1 \times \frac{1}{n+1} + 1 \times \frac{1}{n+1} + \cdots + 1 \times \frac{1}{n+1} + n \times \frac{1}{n+1} \\
&= \frac{2n}{n+1} \\
&= O(1)
\end{aligned}
$$

对一个数据结构进行一组连续操作中，大部分情况下时间复杂度都很低，只有个别情况下时间复杂度比较高，而且这些操作之间存在前后连贯的时序关系，这个时候就可以将这一组操作放在一块分析，看是否能将较高时间复杂度那次操作的耗时，平摊到其它那些时间复杂度比较低的操作上。而且，在能够应用均摊时间复杂度分析的场合，一般均摊时间复杂度就等于最好情况时间复杂度。

1.3 递推方程

1.3.1 递推（Recurrence）

如果数列 $\{a_n\}$ 的第 n 项与它前一项的关系可以用一个公式来表示，那么这个公式就叫做这个数列的递推方程。

算术级数的递推关系：

$$a_0 = a$$
$$a_n = a_{n-1} + d$$

几何级数的递推关系：

$$a_0 = a$$
$$a_n = a_{n-1} \times r$$

1.3.2 斐波那契数列（Fibonacci Sequence）

斐波那契数列 f_0，f_1，f_2，... 的递推公式为：

$$f(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ f(n-1) + f(n-2) & n >= 3 \end{cases}$$

斐波那契数列的通项公式为：

$$f_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{n+1}$$

斐波那契数列（递归）

```
1 int fibonacci(int n) {
```

```
    if(n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n-2) + fibonacci(n-1);
}
```

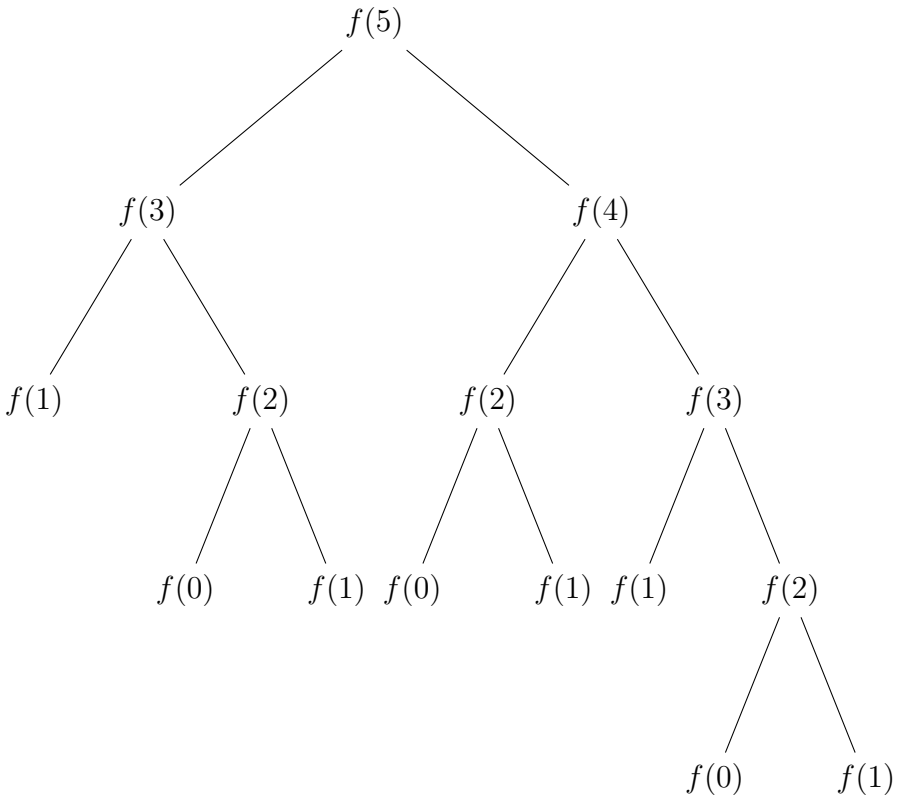

图 1.1: 递归树

斐波那契数列（迭代）

```
int fibonacci(int n) {
    int f[n];
    f[0] = f[1] = 1;
    for(int i = 2; i < n; i++) {
        f[i] = f[i-2] + f[i-1];
    }
    return f[n-1];
}
```

1.3.3 汉诺塔（Hanoi Tower）

给定三根柱子，其中 A 柱子从大到小套有 n 个圆盘，问题是如何借助 B 柱子，将圆盘从 A 搬到 C。

规则：

- 一次只能搬动一个圆盘。
- 不能将大圆盘放在小圆盘上面。

图 1.2: 汉诺塔

递归算法求解汉诺塔问题：

1. 将前 n - 1 个圆盘从 A 柱借助于 C 柱搬到 B 柱。
2. 将最后一个圆盘直接从 A 柱搬到 C 柱。
3. 将 n - 1 个圆盘从 B 柱借助于 A 柱搬到 C 柱。

汉诺塔

```python
def hanoi(n, A, B, C):
    if n == 1
        move(1, A, C)
    else
        hanoi(n-1, A, C, B)
        move(n, A, C)
        hanoi(n-1, B, A, C)
```

当圆盘数为 64 时，假设每次移动花费 1 秒，总共大约需要 5800 亿年。

汉诺塔递归算法的递推公式：

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

利用迭代法，不断用递推方程的右部替换左部，直到出现初值停止迭代。

$$\begin{aligned} T(n) &= 2 * T(n-1) + 1 \\ &= 2 * [2 * T(n-2) + 1] + 1 \\ &= 2 * [2 * [2 * T(n-3) + 1] + 1] + 1 \\ &= \dots \\ &= 2^k * T(n-k) + 2^k - 1 \end{aligned}$$

$$\begin{aligned} &\because n - k = 1 \\ &\therefore k = n - 1 \end{aligned}$$

$$\begin{aligned} T(n) &= 2^{n-1} * T(1) + 2^{n-1} - 1 \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2^n - 1 \\ &= O(2^n) \end{aligned}$$

1.3.4 插入排序

插入排序的递推公式: