算法导论

范潇

2024　春

目录

第一章　算法分析

当我们在研究算法时，需要分析算法是否正确以及是否高效，以便于挑选出最优的算法加以使用。

一个正确的算法需要保证对于所有规模的输入都能给出正确的输出。因此，在证明算法的正确性时，我们往往使用数学归纳法。

一个高效的算法需要在输入规模较大时仍能较为快速地给出输出，为此，我们使用渐进分析来描述算法的效率。

1.1. RAM 模型

为了简化分析算法的过程，我们要对运行算法的机器进行建模以及简化。我们所采用的模型被称为 RAM 模型，即“Random Access Model”，其中的核心假设如下：

1. 内存 (memory) 由一系列字 (word) 组成，可以像数组编号，并通过编号访问
2. CPU 一次只能取出一个字，并对其进行操作
3. 获取一个字只需要一个基本操作。因此我们假设一个字所含的比特数 $w = c\lg n$，其中 n 为数据规模，这保证了其能够在一个字内处理所需数据的编号。

1.2. 循环不变量

通常一个算法中有一个或多个循环语句，并且某些关键性质通过这些循环进行维护，从开始到结束保持着某种不变量。这种不变量我们称其为“循环不变量”。

给定一个核心为循环的算法，首先我们要找出其中的“循环不变量”，然后进行归纳证明。归纳证明主要分为 3 步：

1. **初始状态：**循环不变量是否在进入第一轮循环之前正确
2. **维护：**如果循环不变量在某一轮循环之前正确，它是否在进入下一轮循环之前保持正确
3. **结束状态：**当循环结束时，该循环不变量能否提供有用的性质以便于说明该算法的正确性

对于 `for` 语句而言，这三部的时机分别位于：1）计数器赋初值之后（严格而言并且是在判断循环是否结束之前）；2）执行循环体然后计数器更新这一过程；3）计数器更新导致循环结束后。也就是说，对于 `for` 语句，计数器初始化代表着整个循环语句的初始化，但是和任何一轮循环无关；一轮循环依次包括：检查循环是否结束、执行循环体、更新计数器。

1.3. 渐进分析

在本节中讨论的函数的定义域都是正整数，并且当自变量足够大时，函数值为正。

Definition 1.3.1 (渐进符号).

$$\begin{aligned}
O(g(n)) &= \{f(n) : \exists c, n_0 > 0 \quad s.t. \quad \forall n \geq n_0 \quad f(n)/g(n) \leq c\} \\
\Omega(g(n)) &= \{f(n) : \exists c, n_0 > 0 \quad s.t. \quad \forall n \geq n_0 \quad f(n)/g(n) \geq c\} \\
\Theta(g(n)) &= \{f(n) : \exists c_1, c_2, n_0 > 0 \quad s.t. \quad \forall n \geq n_0 \quad c_1 \leq f(n)/g(n) \leq c_2\} \\
o(g(n)) &= \{f(n) : \lim_{n\to\infty} f(n)/g(n) = 0\} \\
\omega(g(n)) &= \{f(n) : \lim_{n\to\infty} f(n)/g(n) = \infty\}
\end{aligned}$$

Remark.

记忆渐进符号中 f, g 的相对关系可以进行如下理解：渐进符号是一个集合，有许多函数，而该渐进符号却用一个函数来代表这个集合，那么这个函数是有特殊意义的，也就是各个渐进符号的含义。例如 $O(f)$ 就是一个函数的集合，其中的函数的一个紧上界为 f。

$O(g(n)), o(g(n))$ 中的函数的上界均可以用 $g(n)$ 来描述，区别在于前者的上界是“紧”的，而后者是“松”的——给出的上界远远超过实际值。类似地，$\Omega(g(n)), \omega(g(n))$ 中的函数的下界均可以用 $g(n)$ 来描述，区别在于前者的下界是“紧”的，而后者是“松”的——给出的下界远远小于实际值。而 $\Theta(g(n))$ 中的函数可以由 $g(n)$ 给出一个“紧”的估计。

渐近符号 $O, \Omega, \Theta, o, \omega$ 可以分别比作 $\leq, \geq, =, <, >$，享有着对应符号的性质，例如传递性，对称性等。但是，渐进符号不具有三歧性。

1.4. 分治法

分治法是设计算法的常用方法之一，它主要分为三步：分割、解决以及合并。它的主要思想便是将一个规模较大的问题拆分成多个规模较小的问题，然后分别解决这些较小的问题，最后将这些较小问题的解答合并，从而得到原先问题的解答。分治法通常涉及到递归，它的时间复杂度的一般形式为

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

这表示：1）我们假设当问题规模降低到一定程度时，便可以在常数时间内解决；2）当问题规模较大时，我们将原先规模为 n 的问题拆分为 a 个规模为 n/b 的子问题，将它们分别解决后然后进行合并，其中 $D(n), C(n)$ 分别表示拆分和合并所需要的时间，我们也可以用 $f(n)$ 来表示它们之和。

Remark.

一般而言，每个阶段我们将拆分得到多个规模相同的，且结构和原问题一致的子问题。如果我们需要拆分出来结构和原问题发生较大变化的子问题，我们则将它对应的时间复杂度算入合并所需的时间中。

这也就意味着，分析使用分治法的算法的时间复杂度时，需要求解一个递推函数。为此，本节将介绍三种求解方法。

1.4.1. 替换法

替换法的核心步骤为

1. 猜测解的形式
2. 使用数学归纳法进行证明

第一步通常需要较多的经验。当然也可以通过不断缩小上下界来找到紧的界。

在进行数学归纳法的证明时，我们可能会在证明初始情况遇到困难，此时我们可以将初始情况进行适当调整，因为我们进行的是渐进分析。在证明递推情况时，我们可能也会遇到问题，此时的解决方法通常是在原先估计的基础上，减去一个低阶量。

Remark.
通过减去低阶量，往往能够使得我们的归纳假设变得更强，从而更轻松地完成证明。

1.4.2. 递归树法

递归树法便是通过绘制树状的复杂度分布图来进行分析：从上至下，每一层代表着一层递归，结点上标明对应的时间复杂度，然后将每层的时间复杂度进行求和，最终再对所有层的时间复杂度进行求和，得到整个算法的时间复杂度。

递归树法主要用于辅助替代法寻找到解的形式，因此在使用递归树法时，我们可以适当化简问题，例如忽略取整符号、限制问题规模为某个整数的倍数等等——我们只需找到解的形式，然后利用数学归纳法进行严格证明。

1.4.3. 主定理法

Theorem 1.4.1.
设常数 $a \geq 1, b > 1$，$f(n)$ 为一个函数，并令 $T(n)$ 为定义域为非负整数的函数，且

$$T(n) = aT(n/b) + f(n),$$

其中 n/b 可以视为 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。那么 $T(n)$ 有如下渐进界：

1. 如果 $f(n) = O(n^{\log_b a - \epsilon})$，其中 ϵ 为某正数，则 $T(n) = \Theta(n^{\log_b a})$。
2. 如果 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$。
3. 如果 $f(n) = \Omega(n^{\log_b a + \epsilon})$，其中 ϵ 为某正数，同时 $af(n/b) \leq cf(n)$ 对于某个常数 $c < 1$ 当 n 足够大时成立，则 $T(n) = \Theta(f(n))$。

Remark.
$\log_b a$ 中的 a, b 的位置可以用 $aT(n/b)$ 中的位置来辅助记忆。
渐进符号也和 $\log_a b$ 相对于 $f(n)$ 的关系相吻合。

该定理将情况按照 $f(n)$ 与 $n^{\log_b a}$ 相对渐进关系进行分类讨论。总的而言，该定理指出，两个哪个更“大”，则哪个是是所求的解，如果一样“大”，则解为 $\Theta(n^{\log_b a}\lg n)=\Theta(f(n)\lg n)$。

进一步，在第一个情况中，要求 $f(n)$ 以多项式级别渐进小于 $\log_b a$；在第三个情况中，要求 $f(n)$ 以多项式级别渐进大于 $\log_b a$，同时，还需要满足“正则条件”$af(n/b)\leq cf(n)$。

1.5. 概率分析与随机算法

算法分为确定性算法和随机算法。对于确定性算法，给定一个输入，输出也就确定了；而对于随机算法，给定一个输入，每次的输出可能不同。这也就意味着，对于随机算法，并没有特定的输入会导致最坏时间复杂度。

对于确定性算法，它的输入可能是随机的，或者说是服从一定分布的。对于随机算法也是，但是随机算法可能会进一步对其进行随机化处理，改变输入原有的分布，使得对于同一个输入，用于实际运算的数据有所不同。例如某个算法的输入是 n 元排序，原有的输入可能服从一定的分布，但是我们可以设计一个随机算法，在开头将输入进行随机转置，从而将其视为服从均匀分布，此时我们实际上无需对输入原来服从的分布作任何假设。

Remark.
“转置服从均匀分布”指的是成为任何一种可能的转置都是等概率的。

当算法的输入服从一定的分布时，我们将考虑了该分布后得出的时间复杂度称为平均时间复杂度。而当算法自身引入随机性时，我们称该算法的时间复杂度为期望时间复杂度

1.5.1. 示性随机变量

示性随机变量常常配合期望的线性性来辅助对于期望的分析。

Definition 1.5.1 (示性随机变量 I).

$$I\{A\}=\begin{cases}1 & \text{if A occurs,}\\ 0 & \text{if A does not occur.}\end{cases}$$

显然，令 $X_A=I\{A\}$，则 $E[X_A]=Pr\{A\}$

第二章　排序算法

现实中的许多应用需要对数据进行排序。同时，许多算法中也包含了“排序”这一子步骤。因此，研究排序算法十分重要。有许多排序算法，评价它们的质量好坏主要依据一下几个指标：时间复杂度，所需附加存储，复杂性。

排序算法的时间复杂度主要体现在所需的平均比较次数和平均移动次序上。如果一个算法所需的平均辅助存储不随输入规模变化而变化，则称该算法为原地算法。如果一个排序算法在排序过程中，会直接将待排序的最大（小）元素调整至它的最终位置，则称该排序算法具有部分排序功能。如果一个排序算法不会改变排序依据——关键字——相同的数据的相对位置，则称该排序算法具有稳定性。

2.1. 插入排序及其变形

2.1.1. 直接插入排序

在整理手牌时，我们常用直接插入排序。它的核心思想为：不断将未排序的数据以不破坏排序的方式插入到已排序的数据中。其伪代码如下：

Algorithm 2.1.1: Insertion-Sort

```
// A 为待排序序列
for j = 2 to A.length do
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > 0 do
        A[i + 1] = A[i]
        i = i - 1
    end while
    A[i + 1] = key
end for
```

平均情况下，$A[i]$ 需要向前移动 $i/2$ 个位置。因此，平均时间复杂度为 $O(n^2)$。易知，该算法为原地算法，且具有稳定性，但是不具有部分排序功能。

2.1.2. 折半插入排序

直接插入排序算法中的内层循环本质上是在寻找待插入的位置，因为待插入的是一个有序的子序列，所以我们可以在这一过程中使用二分法寻找待插入的位置，而这便是折半插入排序算法。

折半插入排序有较优的平均比较次数——$O(n \lg n)$，其中折半查找的时间复杂度为 $O(\lg n)$。但是，无论数据初始状态如何，该算法始终会执行折半查找，即使我们输入的是有序的序列（该情况会使得折半插

入排序算法达到最坏时间复杂度，但是对于排序算法却能达到最优时间复杂度)。

由于折半插入排序只改变了直接插入排序中寻找插入位置的方法，所以它仍为原地算法，且具有稳定性，但仍不具有部分排序功能。同时，它的*平均移动次数仍为* $O(n^2)$。

2.1.3. 希尔排序

直接插入排序中“将未排序数据插入已排序数据”这一步可以理解为“通过不断交换待插入元素和与它相邻的数据，使其移动到待插入的位置”。易知，这样的每一次交换都消去了一对逆序对，而当逆序对全部消去时，我们便完成了排序。因为直接插入排序中的每次操作只能消去一对逆序对，如果能够改进算法，使得每次操作能消去多对逆序对，我们便能获得更高效的算法，而这便是希尔排序的改进之处。

希尔算法的核心思想是：设定一个严格单调递减至 1 的间隔数序列，遍历这个间隔数序列：对于每个间隔数，将待排序序列分割成多个子序列，每个序列的下标是以间隔数为公差的等差数列，然后对每个子序列进行直接插入排序。

当间隔数较大时，子序列中的一次操作能够消去多对逆序对；当间隔数较小时，只需在之前的基础上进行微调即可，从而整体的时间复杂度较优。

> *Remark.*
> 间隔数序列设定的方法尚无定论。但是如果间隔数之间成倍数关系的话，可能会导致对于某些间隔数无需排序子序列，即产生冗余，所以一种策略是选取互素的间隔数。

由于关键字相同的数据可能位于两个子序列中，因此希尔排序无法保证稳定性。

> *Remark.*
> 虽然算法并没有对具体实现的细节作要求，但是实现的细节要符合算法的思想。插入算法的思想就是“插入”——将目前排序的元素提取出来，其他元素依次平移，*而不是用 swap 进行交换*。

2.2. 交换排序

交换排序的基本思想是：两两比较待排序对象的关键字，如果发生逆序 (即排列顺序与排序后的次序正好相反)，则交换之，直到所有对象都排好序为止。

2.2.1. 冒泡排序

基本思想是：进行 $n-1$ 次循环，每次循环从序列首元素开始，通过前后交换，将当前遇到的最大元素移动到序列末端。

> *Remark.*
> 这里设置 *change* 变量来辅助判断排序是否已经完成。

> *Remark.*
> 许多排序算法, 尤其是具有部分排序功能的, 只需要进行 $n-1$ 轮循环即可, 因为找出排出前 $n-1$ 大的数也就表明整个序列已经排序完成。

Algorithm 2.2.1: Bubble-Sort(nums)

```
for i = 1 to nums.length-1 do
    change = 0 for j = 1 to nums.length-1 do
        if nums[j+1]<nums[j] then Swap (nums[j],nums[j+1])
        change = 1
    end for
    if change == 0 then break
end for
```

2.2.2. 快速排序

快速排序算法利用了分治法，其主要思想是：将一个枢纽元素调整至最终的位置，并且确保其他元素和它的相对位置也是正确的，然后对位于其左右的子序列再次进行快速排序。

Algorithm 2.2.2: Quick-Sort(A,l,r)

```
if l<r then
    p = Partition(A,l,r)
    Quick-Sort (A,l,p-1)
    Quick-Sort (A,p+1,r)
end if
```

Function Partition(A,l,r)

```
// 默认以最后一个元素作为 pivot;l,r 分别为子序列的左右端点下标
// 返回 pivot 的最终下标
i = l - 1
for j = l to r-1 do
    if Compare(A[j],A[r]) then // 使用自定义的比较规则
        i = i + 1
        Swap (A[j],A[i])
    end if
end for
Swap (A[i+1],A[r])
return i+1
```

快速排序是一种不稳定的排序方式。最坏情况下，它会退化为选择排序。为了避免出现子问题规模不均而退化为选择排序，通常会将该算法随机化，即随机挑选枢纽元素。

2.3. 选择排序

选择排序的基本思想是：每一轮选出一个待排序元素，然后直接将其放入最终的位置。与插入排序和交换排序不同的是，在选择排序算法中，比较过程中不会移动元素位置。

2.3.1. 直接选择排序

Algorithm 2.3.1: Select-Sort(nums)

```
1 for i = 1 to nums.length-1 do
2 |   min = i
3 |   for j = i+1 to nums.length do
4 |   |   if nums[j]<nums[min] then min = j
5 |   end for
6 |   if min ≠ i then Swap (nums[min],nums[i])
7 end for
```

直接选择排序是一种不稳定的排序方式。

2.3.2. 锦标赛排序

锦标赛排序所用的数据结构是满二叉树，树上的元素包含排序关键字，关键字对应的叶子节点的下标以及是否已经排序完成的标志。

初始状态是将待排序元素放置在叶子结点上，若未满则用不参加排序的元素占位。然后通过兄弟节点两两比较，较小者成为父节点。建好树后便通过取根节点元素获取当前的最小元素。每取一次后，读取对应的叶子结点下标，从该叶子节点开始向上至根节点，设置完成排序的标志，并和兄弟结点比较，判断是否需要更新整个结点。

建树的时间复杂度和结点个数成正比，为 $\Theta(n)$，而每次循环需要 $O(\lg n)$，所以总的时间复杂度是 $O(n \lg n)$。需要的额外存储空间为 $\Theta(n)$.

2.3.3. 堆排序

堆排序所用的数据结构是完全二叉树。树上的元素只要有关键字即可。

初始状态时，将待排序元素依次从根节点开始放置到树上，然后从最后一个非叶子结点开始，依次执行“堆化”，确保满足性质：父节点大于孩子节点，如果发生调整，则需要对孩子节点进行迭代调整。建堆完成后，每次从根节点能取出目前的最大元素，将其和目前的末端元素互换，并缩小树的大小，然后再对新的根节点进行“堆化”。

可以证明，建堆的时间复杂度为 $\Theta(n)$，而排序的时间复杂度则是 $\Theta(n \lg n)$。

堆排序不具有稳定性，但是无需额外存储空间。

2.4. 归并排序

使用递归实现，是一种稳定的排序方式，但是需要 $\Theta(n)$ 的额外存储空间。

基本思想是将排序一个序列的问题转化为排序两个左右子列的问题，然后将这两个有序序列合并为一个有序序列的问题。

2.5. 基数排序

基数排序适用于多关键字排序。

多关键字排序的实现主要分为“最高位优先”和“最低位优先”。

进行最高位优先排序时，需要保存高位排序得到的结果，在其内部进行更低位的排序，这对存储效率提出较高要求。

在进行最低位优先排序时，需要确保各位所用的排序算法具有稳定性，不会破坏低位排序得到的相对位置关系。实现时常常采用链表作为数据结构：用一个链表存储每次的排序结果；用与关键词个数数量相同的多个链表来存储排序的中间过程。