子午线

学习笔记

作者：leekarry
组织：果壳
时间：November 9, 2019
版本：0.1

ElegantLaTeX Program

我很快，快到时间都会变慢。而我一慢，时间就会过得飞快。

目　录

第 1 章 算法和算法分析

1.1 算法问题求解基础

算法是计算机学科的一个重要分支，它是计算机科学的基础，更是计算机程序的基石。算法是计算机求解问题的特殊方法。学习算法，一方面需要学习求解计算领域中典型问题的各种有效算法，还要学习设计新算法和分析算法性能的方法。本章给出算法的基本概念，介绍使用计算机求解问题的过程和方法，讨论递归算法及证明递归算法正确性的归纳法。

算法概述

什么是算法？一个算法是对特定问题求解步骤的一种描述，它是指令的有限序列。此外算法具有下列 5 个特征。

1. 输入(input)：算法有零个或多个输入值；
2. 输出(output)：算法至少产生一个输出量；
3. 确定性(definiteness)：算法的每一条指令都有确切的定义，没有二义性；
4. 可行性(effectiveness)：算法的每一条指令必须足够基本，它们可以通过已经实现的基本运算执行有限次来实现；
5. 有穷性(finiteness)：算法必须总能在执行有限步之后终止。

为什么学算法？克努特说过“一个受过良好的计算机科学知识训练的人知道如何处理算法，即构造算法、操纵算法、理解算法和分析算法。算法的知识远不只是为了编写好的计算程序，它是一种具有一般意义的智能工具，必定有助于对其他学科的理解，不论化学、语言学或者音乐等。”

问题求解方法

只要目前的情况与人们所希望的目标不一致，就会产生问题。当我们积累了问题求解的经验，这种对问题解法的猜测就不再是完全盲目的，而是形成某些问题求解的技术和策略。

问题求解过程大致分为 4 步：

1. 理解问题(understand the problem)
2. 设计方案(devise plan)
3. 实现方案(carry out the plan)
4. 回顾复查(look back)

算法设计与分析

算法问题求解过程在本质上与一般问题的求解过程是一致的。算法一般分为两类：精确算法和启发式算法。一般来讲，启发式算法往往缺少理论依据。对于最优化问题，一个算法如果致力于寻找近似解而不是最优解，被称为近似算法 (approximation algorithm)。如果在算法中需做出某些随机选择，则称为随机算法 (randomized algorithm)。

如何设计算法？一般来说，算法的设计是一项创造性活动，不可能完全自动化，但学习一些基本的算法设计策略是非常有用的。对于所求解的问题，只要符合某种算法设计策略的前提，便可以利用它设计出精致而有效的算法

如何表示算法？伪代码是自然语言和程序设计语言的混合结构。它所描述的算法通常比自然语言精确，又比实际程序设计语言简洁。

如何确认算法？确认一个算法是否正确的活动称为算法确认。算法确认的目的在于确认一个算法能否正确无误地工作。使用数学方法证明算法的正确性，称为算法证明。

如何分析算法？算法的分析活动是指对算法的执行时间和所需空间的估算。实际测量一个程序所消耗的时间和空间，这称为程序的性能测量。

递归和归纳

递归是一个数学概念，也是一种有用的程序设计方法。递归和归纳关系紧密。归纳法证明是一种数学证明方法，可用于证明一个递归算法的正确性。

递归定义。定义一个新事物、新概念或新方法，一般要求在定义中只包含已经明确定义或证明的事物、概念或方法。然而递归定义却不然，递归 (recursive) 定义是一种直接或间接引用自身的定义方法一个合法的递归定义假托两部分：基础情况和递归部分。基础情况以直接形式明确列举新事物的若干简单对象，递归部分给出由简单对象定义新对象的条件和方法。

递归算法：当一个算法采用递归方式定义时便成为递归算法。一个递归算法是指直接或间接调用自身的算法。递归本质上也是一种循环的算法结构，它把“较复杂”的计算逐次归结为“较简单”情形的计算，直到归结到“最简单”情形的计算，并最终得到计算结果为止。

递归数据结构，在数据结构中，树、二叉树和列表常采用递归方式来定义。使用递归方式定义的数据结构称为递归数据结构。

证明一个定理不成立的最好方法是举一反例。那么，如何证明一个程序是正确的？两种最常见的证明方法是归纳法和反证法。先来看归纳法。对于无限对象集上的命题，归纳法往往是唯一可行的证明方法。当归纳法应用于递归定义的数据结构时，称为结构归纳法。递归函数和归纳证明二者在结构上非常类似，这对于运用归纳法证明复杂的递归数据结构和算法命题是很有帮助的。

一个递归算法比较容易用归纳法证明其正确性。

1.2 算法分析基础

一旦确信一个算法是正确的，下一个重要的步骤就是分析算法。算法分析是指对算法利用时间和空间这两种资源的效率进行研究。本章讨论衡量算法效率的时间复杂度和空间复杂度，算法的最好、平均和最坏情况时间复杂度，讨论用于算法分析的渐近表示法，介绍如何使用递推关系来分析递归算法的方法及分摊分析技术。

算法复杂度

什么是好的算法？一个好的算法就具有以下 4 个重要特性：

1. 正确性：毫无疑问，算法的执行结果应当满足预选规定的功能和性能要求。
2. 简明性：算法应思路清晰、层次分明、容易理解、利于编码和调试。
3. 效率：算法应有效使用存储空间，并具有高的时间效率。
4. 最优性：算法的执行时间已达到求解该类问题所需时间的下界。

折中和 (tradeoffs and consequences) 是计算机学科的重要概念之一。

影响程序运算时间的因素

1. 程序所依赖的算法；
2. 问题规模和输入数据；
3. 计算机系统性能。

算法的时间复杂度：是指算法运行所需的时间。

1. 最好情况：$B(n) = \min\{T(n, I) | I \in D_n\} = T(n, I')$
2. 最坏情况：$W(n) = \max\{T(n, I) | I \in D_n\} = T(n, I^*)$
3. 平均时间情况：$A(n) = \sum_{I \in D_n} p(I)T(n, I)$

这三种时间复杂度从不同角度反映算法的效率，各有用途，也各有局限性。其中，比较容易分析和计算，并且也最有实际价值的是最坏情况时间复杂度。还有一种类型的时间效率称为分摊效率。它并不针对算法的单次运行，而是计算算法在同一数据结构上执行一系列运算的平均时间。

程序运行时间不公与算法的优劣和输入数据直接相关，还与运行程序的计算机软、硬件环境有关。为了分析算法的效率，总希望略去计算机系统因素，对算法自身的特性进行事前分析。算法的事后测试是通过运行程序，测试一个程序在所选择的输入数据下实际运行所需要的时间。

一个程序步是指在语法上或语义上有意义的程序段，该程序段的执行时间必须与问题实例的规模无关。

算法的空间复杂度是指算法运行所需的存储空间。程序运行所需的存储空间包括以下两部分。

1. 固定空间需求：这部分空间与所处理数据的大小和个数无关；
2. 可变空间需求：这部分空间大小与算法在某次执行中处理的特定数据的规模有关。

渐近表示法

引入程序步的目的在于简化算法的事前分析。事实上一个程序在一次执行中的总程序步的精确计算往往是困难的。那么,引入程序步的意义何在? 本节中定义的渐近时间复杂度,使得有望使用程序步在数据级上估计一个算法的执行时间,从而实现算法的事前分析。

1. **大 O 记号**:设函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数,如果存在两个正常数 c 和 n_0,使得当 $n \geq n_0$ 时,有 $f(n) \leq cg(n)$,则记做 $f(n) = \mathrm{O}(g(n))$。
   大 O 记号可以看成 n 的函数的集合。$\mathrm{O}(g(n))$ 表示所有增长阶数不超过 $g(n)$ 的函数的集合,它用以表达一个算法运行时间的上界。称一个算法具有 $\mathrm{O}(g(n))$ 的运行时间,是指当 n 足够大时,该算法在计算机上的实际运行时间不会超过 $g(n)$ 的某个常数倍,$g(n)$ 是它的一个上界。
2. **Ω 记号**:设函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数,如果存在两个正常数 c 和 n_0,使得当 $n \geq n_0$ 时,有 $f(n) \geq cg(n)$,则记做 $f(n) = \Omega(g(n))$
   Ω 记号可以看成 n 的函数的集合。$\Omega(g(n))$ 表示所有增长阶数不低于 $g(n)$ 的函数的集合,它用于表达一个算法运行时间的下界。
3. **Θ 记号**:
4. **Ω 记号**: 设函数 $f(n)$ 和 $g(n)$ 是定义在非负整数集合上的正函数,如果存在正常数 c_1, c_2 和 n_0,使得当 $n \geq n_0$ 时,有 $c_1 g(n) \leq f(n) \leq c_2 g(n)$,则记做 $f(n) = \Theta(g(n))$
   Θ 记号可以看成 n 的函数的集合。$\Theta(g(n))$ 表示所有增长阶数与 $g(n)$ 相同的函数的集合,它用于表示一个算法运行时间具有与 $g(n)$ 相同的阶。称一个算法具有 $\Theta(g(n))$ 的运行时间,是指当 n 足够大时,该算法在计算机上的实际运行时间大约为 $g(n)$ 某个常数倍大小的时间量。
5. **小 o 记号**:$f(n) = \mathrm{o}(g(n))$ 当且仅当 $f(n) = O(g(n))$ 且 $f(n) \neq \Omega(g(n))$
   小 o 记号可以看成 n 的函数的集合。$\mathrm{o}(g(n))$ 表示所有增长阶数小于 $g(n)$ 的所有函数的集合,它用于表示一个算法运行时间 $f(n)$ 的阶比 $g(n)$ 低。

算法按时间复杂度分类: 凡渐近时间复杂度为多项式时间阶界的算法称做多项式时间算法,而渐近时间复杂度为指数函数限界的算法称做指数时间算法。

1. 多项式时间算法

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) \tag{1.1}$$

2. 指数时间算法

$$O(2^n) < O(n!) < O(n^n) \tag{1.2}$$

递推关系

递推关系经常用来分析递归算法的时间和空间代价。分析一个递归算法的时间一般需要列出关于时间复杂度函数的递推关系式。计算递推式通常有三种方法: 迭代方法(iterating)、替换方法(substitution)和主方法(master methed)。

1. **替换方法**要求首先猜测递推式的解，然后用归纳法证明。下面使用替换方法分析汉诺塔问题：分析汉诺塔问题，得到递推式：$T(1)=1, T(n)=2T(n-1)+1$。可以先对以下这些小的示例进行计算：

$$T(3)=7=2^3-1; T(4)=15=2^4-1; \ldots$$

似乎 $T(n)=2^n-1, n\geq 1$，下面再用归纳法证明这一结论。

2. **迭代方法**的思想是扩展递推式，将递推式先转换成一个和式，然后计算该和式，得到渐近复杂度。它需要较多的数学运算。

使用迭代方法分析汉诺塔问题：函数 Hanoi 中Ⓕ两次调用自身，函数调用使用的实在参数均为 n-1，函数 Move 所需时间具有常数界 $\Theta(1)$，可以将其视为一个程序步，于是有

$$T(n)=\begin{cases}1 & n=1\\ 2T(n-1)+1 & n>1\end{cases} \tag{1.3}$$

扩展并计算此递推式：

$$\begin{aligned}T(n)&=2T(n-1)+1=2(2T(n-2)+1)+1=2^2T(n-2)+2+1\\&=2^3T(n-3)+2^2+2+1\\&\ldots\\&=2^n-1\end{aligned} \tag{1.4}$$

3. 在递归算法分析中，常需要求解如下形式的递推式：

$$T(n)=aT(n/b)+f(n) \tag{1.5}$$

求解这类递推式的方法称为**主方法**。主方法依赖于下面的主定理，**主定理**：设 $a\geq 1$ 和 $b>1$ 为常数，$f(n)$ 是一个函数，$T(n)$ 由下面的递推式定义

$$T(n)=aT(n/b)+f(n) \tag{1.6}$$

式中，n/b 指 $\lfloor n/b\rfloor$ 或 $\lceil n/b\rceil$ 则 $T(n)$ 有如下的渐近界：

(a). 若对某常数 $\epsilon>0$，有 $f(n)=O(n^{log_b^{a-\epsilon}})$，则 $T(n)=\Theta(n^{log_b^a})$

(b). 若 $f(n)=\Theta(n^{log_b^a})$，则 $T(n)=\Theta(n^{log_b^a}logn)$

(c). 若对某常数 $\epsilon>0$，有 $f(n)=\Omega(n^{log_b^{a+\epsilon}})$，且对某个常数 $c<1$ 和所有足够大的 n，有 $af(n/b)\leq cf(n)$，则 $T(n)=\Theta(f(n))$

需要注意的是，主定理的三种情况并没有覆盖所有的 $f(n)$，存在某些 $f(n)$ 不满足以上任何一种情况的条件，则此时就不能用主方法求解递推式。

举个例子：$T(n)=16T(n/4)+n$

因为 $a=16, b=4, n^{\log_b^a}=n^2, f(n)=n=\mathrm{O}(n^{\log_b^{a-\epsilon}})=\mathrm{O}(n^{2-\epsilon})$，其中，$\epsilon=1$ 与主定理

的情况 (1) 相符合，$T(n) = \Theta(n^{\log_b^a}) = \Theta(n^2)$

分摊分析

在很多情况下，对一个数据结构的操作往往不是单独执行一次运算，而是重复多次执行多个运算，形成一个运算执行序列。如果执行 n 个运算的总时间为 T(n)，则每个运算的平均代价（average cost）为 T(n)/n，分摊分析的目的是求平均代价。

分摊分析和平均情况分析的不同之处在于它不需要假定每个运算的概率，因而不涉及概率。分析分析保证在最坏情况下一个运算序列中每个运算的平均性能。分摊分析一般有三个方法：

1. 聚集方法：需要对所有 n，计算由 n 个运算构成的运算序列在最坏情况下总的执行时间 $T(n)$，则每个运算的平均代价为 $T(n)/n$。注意，序列中允许包含不同种类的运算，但每个运算的分摊代价是相同的。
2. 会计方法：对每个运算预先赋予不同的费值（charge）。会计方法的分摊代价可以不同。与聚集分析不同的是，会计方法的分摊代价可以不同，而聚集分析中每个运算有相同的分摊代价。
3. 势能方法：为数据结构的每个状态定义一个被称为势能的量。

本章小结

本章首先概述有关算法、问题、问题求解过程及算法问题求解方法等重要概念和方法。算法可以看做求解问题的一类特殊的方法，它是精确定义的，能在有限时间内获得答案的一个求解过程。对算法的研究主要包括如何设计算法，如何表示算法，如何确认算法的正确性，如何分析一个算法的效率，以及如何测量程序的性能等方面。算法设计技术是问题求解的有效策略。算法的效率通过算法分析来确定。递归是强有力的算法结构。递归和归纳关联紧密。归纳法是证明递归算法正确性和进行算法分析的强有力工具。

另外，重点介绍算法分析的基本方法。算法的时间和空间效率是衡量一个算法性能的重要标准，对于算法的性能分析可以采用事前分析和事后测量形式进行。算法分析通常是指使用渐近表示法对一个算法的时间和空间需求做事前分析。算法复杂度的渐近表示法用于在数量级上估算一个算法的时空资源耗费。算法的运行时间可使用程序步来衡量。一个算法可以讨论其最好的、平均和最坏情况时间复杂度，其中，最坏情况分析最有实际价值。算法的空间复杂度一般只做最坏情况分析。

递归算法是一类重要的算法结构，也是较难掌握的一种算法技术。在第 1 节的基础上。第 2 节讨论使用递推关系分析递归算法的方法及求解递推式的 3 种途径。最后讨论算法时间分析的分摊方法。

第 2 章 基本搜索和遍历方法

2.1 基本搜索和遍历方法

搜索和遍历是计算机问题求解最常用的技术之一。本章讨论基本搜索和遍历方法，分析它们的性能。

基本概念

1. 搜索：是一种通过系统地检查给定数据对象的每个结点，寻找一条从开始结点到答案结点的路径，最终输出问题的求解方法。
2. 遍历：要求系统地检查数据对象的每个结点。根据被遍历的数据对象的结构不同，可分为树遍历和图遍历。
3. 状态空间：用于描述所求问题的各种可能的情况，每一种情况对应于状态空间中的一个状态。
4. 无知搜索：按事先约定的某种次序，系统地在状态空间中搜索目标状态，而无须对状态空间有较多了解。
5. 有知搜索：具有某些关于问题和问题解的知识，那么，便可运用这些知识，克服无知搜索的盲目性。采用经验法则的搜索方法称为启发式搜索 (heuristic search)。

深度优先搜索 (depth first search) 和广度优先搜索 (breadth first search) 是两种基本的盲目搜索方法，介于两者之间的有 D-搜索 (depth search)。

图的搜索和遍历

遵循某种次序，系统地访问一个数据结构的全部元素，并且每个元素仅访问一次，这种运算称为遍历。很显然，实现遍历运算的关键是规定结点被访问的次序。

在树形结构中，一个结点的直接后继结点是它的孩子结点；在图形结构中，一个结点的后继结点是邻接于该结点的所有邻接点。为深入认识搜索算法的特点，不妨将被搜索的数据结构中的结点按其状态分成 4 类：

1. 未访问。x 尚未访问
2. 未检测。x 自身已访问，但 x 的后继结点尚未全部访问。
3. 正扩展。检测一个结点 x，从 x 出发，访问 x 的某个后继结点 y，x 被称为扩展结点也叫 E-结点。在算法执行的任何时刻，最多只有一个结点为 E-结点。
4. 已检测。x 自身已访问，且 x 的后继结点也已全部访问。

根据如何选择 E-结点的规则不同，得到两种不同的搜索算法：深度优先搜索和广度优先搜索。

广度优先搜索

对于广度优先搜索，一个结点 x 一旦成为 E-结点，算法将依次访问完它的全部未访问的后继结点。每访问一个结点，就将它加入活结点表。直到 x 检测完毕，算法才从活结点表另选一个活结点作为 E-结点。广度优先搜索以队列作为活结点表。

时间分析：$O(n+e)$，如果用邻接矩阵表示图，则所需时间为 $O(n^2)$

深度优先搜索

如果一个遍历算法在访问了 E-结点 x 的某个后继结点 y 后，立即使 y 成为新的 E-结点，去访问 y 的后继结点，直到完全检测结点 y 后，x 才能再次成为 E-结点，继续访问 x 的其他未访问的后继结点，这种遍历称为深度优先搜索。深度优先搜索使用堆栈为活结点表。

时间分析：$O(n+e)$，如果用邻接矩阵表示图，则所需时间为 $O(n^2)$

双连通分量

无向图的双连通性在网络应用中非常有价值。如果一个无向图的任意两个结点之间至少有两条不同的路径相通，则称该无向图是双连通的。

在一个无向连通图 $G=(V,E)$ 中，可能存在某个 (或多个) 结点 a，使得一旦删除 a 及其相关联的边，图 G 不再是连通图，则结点 a 称为图 G 的关节点。如果删除图 G 的某条边 b，该图分离成两个非空子图，则称边 b 是图 G 的桥。如果无向连通图 G 中不包含任何关节点，则称图 G 为双连通图(biconnected graph)。一个远射连通图 G 的双连通分量是图 G 的极大双连通子图。

两个双连通分量至多有一个公共结点，且此结点必为关节点。两个双连通分量不可能共有同一条边。同时还可以看到，每个双连通分量至少包含两个结点 (除非无向图只有一个结点)。

发现关节点在网络应用中，通常不希望网络中存在关节点，因为这意味着一旦在这些位置出现故障，势必导致大面积的通信中断。一个无向连通图不是双连通图的充要条件是图中存在关节点。在无向图中识别关节点的最简单的做法是：从图 G 中删除一个结点 a 和该结点的关联边，再检查图 G 的连通性。如果图 G 因此而不再是连通图，则结点 a 是关节点。这一方法显然太费时，

采用深度优先搜索识别无向图的关节点的方法有很好的时间性能。无向图的深度优先树中只包含树边和反向边两类边；一个双连通图中不包含关节点，即要求图中任意一对结点之间存在简单回路。因此具有下列性质

给定无向连通图 $G=(V,E)$，$S=(V,T)$ 是图 G 的一棵深度优先树，图中结点 a 是一个关节点，当且仅当

1. a 是根，且 a 至少有两个孩子；
2. 或者 a 不是根，且 a 的某棵子树上没有指向 a 的祖先的反向边。

构造双连通图

与或图

很多复杂问题很难或无法直接求解，但可以将它们分解成一系列（类型可以不同）子问题。这些子问题又可进一步分解成一些更小的子问题，这种问题分解过程可以一直进行下去，直到所生成的子问题已经足够简单，可用一些已知的普通求解方法求解为止。然后由这些子问题的解再逐步导出原始问题的解。这种将一个问题分解成若干个子问题，继而分别求解子问题，最后又从子问题的解导出原始问题的解的方法称为问题归约。

本章小结

搜索一个数据结构就是以一种系统的方式访问该数据结构中的每一个结点。对树和图的搜索和遍历是许多算法的核心。许多人工智能问题的求解过程就是搜索状态空间树。通过系统地检查问题的状态空间树中的状态，寻找一条从起始状态到答案状态的路径作为搜索算法的解。搜索和遍历也是许多重要图算法基础。通过对图的搜索获取图的结构信息来求解问题。另一些图算法实际上是由基本的图搜索算法经过简单的扩充而成的。本章讨论图的搜索与遍历。回溯法和分枝限界法将介绍问题求解的状态空间树搜索方法。

第 3 章　分治法

第 4 章　贪心法

第 5 章　动态规划法

第 6 章 回溯法

第 7 章　分枝限界法

第 8 章 NP 完全问题