子午线

学习笔记

作者：leekarry
组织：果壳
时间：November 18, 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

1　算法和算法分析　1

1.1　算法问题求解基础　1

1.2　算法分析基础　3

2　基本搜索和遍历方法　7

2.1　基本搜索和遍历方法　7

3　分治法　10

3.1　一般方法　10

3.2　求最大最小元　11

3.3　二分搜索　12

3.4　排序问题　12

3.5　选择问题　14

3.6　斯特拉森矩阵乘法　14

3.7　本章小结　14

4　贪心法　16

4.1　一般方法　16

4.2　背包问题　17

4.3　带时限的作业排序　17

4.4　最佳合并模式　18

4.5　最小代价生成树　19

4.6　单源最短路径　19

4.7　磁带最优存储　20

4.8　贪心法的要素　21

4.9　本章小结　21

5　动态规划法　22

5.1　一般方法和基本要素　22

5.2　每对结点间的最短路径　24

5.3　最长公共子序列　25

5.4　0/1 背包　27

5.5　流水作业调度　27

6　回溯法　28

7　分枝限界法 **29**

8　NP 完全问题 **30**

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

问题分解是求解复杂问题时很自然的做法。求解一个复杂问题可以将其分解成若干个子问题，子问题还可以进一步分解成更小的问题，直到分解所得的小问题是一些基本问题，并且其求解方法是已知的，可以直接求解为止。分治法作为一种算法设计策略，要求分解所得的子问题是同类问题，并要求原问题的解可以通过组合子问题的解来获取。本章首先介绍分治法的一般方法，它的算法框架，算法分析的递推关系。本章以后各小节将通过若干常见的分治算法问题，如二分搜索、选择问题和矩阵相乘问题等，加深对分治法所能求解的问题特征的理解，并学会运用公法策略来求解问题的方法。分析递归算法的时间复杂度是本章的另一项任务。

3.1 一般方法

分治法顾名思义就是分而治之。一个问题能够用分治法求解的要素是：第一，问题能够按照某种方式分解成若干个规模较小、相互独立且与原问题类型相同的子问题；第二，子问题足够小时可以直接求解；第三，能够将子问题的解组合成原问题的解。因此，分治法求解很自然的导致一个递归算法。

```
SolutionType DandC(ProblemType P)
{
    ProblemType P1,P2,..., Pk;
    //子问题足够小，直接求解
    if(Small(P)) return S(P);
    else
    {
        //将问题P分解成子问题P1,P2,..., Pk
        Divide(P, P1, P2, ... ,Pk);
        //求解子问题，并合并解
        Return Combine(DandC(P1), DandC(P2), ... ,DandC(Pk));
    }
}
```

算法分析

采用分治法求解问题通常得到一个递归算法。往往可得到如下的递推关系式：

$$T(n) = aT(n/b) + cn^k,\ T(1) = c \tag{3.1}$$

使用主方法得到下面的定理

定理 3.1

$$T(n)=\begin{cases}\Theta(n^{\log_b^a}) & \text{如果} a>b^k\\ \Theta(n^k\log n) & \text{如果} a=b^k\\ \Theta(n^k) & \text{如果} a<b^k\end{cases} \tag{3.2}$$

♡

3.2 求最大最小元

讨论用分治法在一个元素集合中寻找最大元素和最小元素的问题。

```
1  void MaxMin(int i, int j, T& max, T& min)
2  {
3      T min1, max1;
4      // 表中只有一个元素时
5      if(i == j) max=min=l[i];
6      // 表中有两个元素时
7      else if(i == j-1)
8      {
9          if(l[i] < l[j])
10         {
11             max=l[j]; min=l[i];
12         }else{
13             max=l[i]; min=l[j];
14         }
15     }else{
16         int m = (i + j) / 2;
17         MaxMin(i, m, max, min);
18         MaxMin(m+1, j, max1, min1);
19         if(max < max1) max = max1;
20         if(min > min1) min = min1;
21     }
22
23 }
```

时间分析

$$T(n)=3n/2-2 \tag{3.3}$$

3.3 二分搜索

搜索运算是数据处理中经常使用的一种重要运算。在一个表中搜索确定一个关键字值为给定值的元素是一种常见的运算。若表中存在这样的元素，则称搜索成功，搜索结果可以返回整个数据元素，也可指示该元素在表中的位置；若表中不存在关键字值等于给定值的元素，则称搜索不成功。本节讨论采用分治法求解在有序表中搜索给定元素的问题。

二分搜索框架

```
int BSearch(T& x, int left, int right)
{
    if (left <= right)
    {
        int m = Divide(left, right);
        if(x < l[m]) return BSearch(x, left, m-1);
        else if(x > l[m]) return BSearch(x, m+1, right);
        else return m;
    }
    return -1;
}
```

搜索算法的时间下界

在一个有 n 个元素的集合中，通过关键字值之间的比较，搜索指定关键字值的元素，任意这样的算法在最坏情况下至少需要进行 $\lfloor \log n \rfloor + 1$ 次比较。

3.4 排序问题

归并排序

归并排序的基本算法是把两个或多个有序序列合并成一个有序序列。使用分治法的两路归并排序算法可描述为：将待排序的元素序列一分为二，得到两个长度基本相等的子序列，类似对半搜索的做法；然后对两个子序列分别排序，如果子序列较长，还可继续细分，直到子序列的长度不超过 1 为止；当分解所得的子序列已排列有序时，将两个有序子序列合并成一个有序子序列，实现将子问题的解组合成原问题解。

```
void MergeSort(int left, int right)
{
    if(left < right)
    {
        int mid = (left + right) / 2;
        MergeSort(left, mid);
        MergeSort(mid+1, right);
```

```
        // 将两个有序子序列合并成一个有序序列
        Merge(left, mid, right);
    }
}
void Merge(int left, int mid, int right)
{
    T* temp = new T[right - left + 1];
    int i = left, j = mid+1, k = 0;
    while((i <= mid) && (j <= right))
    {
        if( l[i] <= l[j]) temp[k++] = l[i++];
        else temp[k++] = l[j++];
    }
    while(i <= mid) temp[k++] = l[i++];
    while(j <= right) temp[k++] = l[j++];
    for(i = 0, k = left;k <= right;)
    {
        l[k++] = temp[i++];
    }
}
```

快速排序

快速排序又称分划交换排序。当使用分治法设计排序算法时,可以采取与两路合并排序完全不同的方式对问题进行分解。分划操作是快速排序的核心操作。

```
#include <cstdio>
int Partition(int A[], int left, int right)
{
    int temp = A[left]; //将最左边的值存放
    while (left < right)
    {
        while (left < right && A[right] > temp) right--;
        A[left] = A[right];
        while (left < right && A[left] < temp) left++;
        A[right] = A[left];
    }
    //把temp放到left与right相遇的地方
    A[left] = temp;
    return left;
}
void quickSort(int A[], int left, int right)
{
```

```
18      if (left < right)
19      {
20          int pos = Partition(A, left, right);
21          quickSort(A, left, pos - 1);
22          quickSort(A, pos + 1, right);
23      }
24  }
```

合并排序和快速排序虽然都运用分治策略，但两者的角度不同，得到的排序算法也不同。可见，一种算法设计策略提供了一种设计算法的启示。对于同一问题，基于同一算法设计策略，算法设计者可以根据各自对问题的理解和分析，提出不同的具体解决方法设计出不同的算法。

排序算法的时间下界

任何一个通过关键字值比较对 n 个元素进行排序的算法，在最坏情况下，至少需要做 $(n/4)\log n$ 次比较。

3.5 选择问题

选择问题是指在 n 个元素的集合中，选出某个元素值大小在集合中处于第 k 位的元素，即所谓的求第 k 小元素的问题。

分治法求解

如果使用快速排序中所采用的分划方法，以主元为基准，将一个表划分成左右两个子表，左子表中的所有元素均小于或等于主元，而右子表中的元素均大于或等于主元。设其左子表长度为 p，那么 k=p，则主元就是第 k 小元素；否则若 k<p，第 k 小元素必定在左子表中，否则就在右子表中。

3.6 斯特拉森矩阵乘法

普通的矩阵相乘算法的时间复杂度为 $\Theta(n^3)$。斯特拉森分治法是一种尝试，他的巧妙设计使得矩阵乘法在计算时间的数量级上得到突破，成为 $O(n^{2.81})$。

3.7 本章小结

分治法是一种非常有用的算法设计技术，它可用于求解许多算法问题。分治法设计的算法一般是递归的。分析递归算法的时间得到一个递推方程。递推方程可使用替换方法、迭代方法和主方法求解。

本章通过对最大最小元、二分搜索、排序、选择及斯特拉森矩阵相乘等典型示例的讨论，详细介绍了如何运用法设计算法的方法，以及分析算法的时间和空间效率的方法。在按照分治法的要素分析一个问题时，如果分析问题的角度不同，则可以得到完全不同的算法。快速排序和合并排序算法说明了这一点。

本章还讨论了基于元素间比较的搜索问题和排序问题的时间下界。问题求解的时间下界对算法设计有指导意义。

第 4 章 贪心法

贪心法求解最优化问题。本章讨论运用贪心法求解的一类问题的特征及求解方法。读者熟知的某些图算法，例如，最小代价生成树问题和单源最短路径问题可用贪心法求解。本章讨论的这些问题，如背包问题、最佳合并模式及带时限的作业排序问题等，都是贪心求解的典型问题。通过分析这些问题，掌握贪心法的基本要素，学会如何使用贪心策略设计算法。

4.1 一般方法

一般来讲，如果一个问题适合用贪心法求解，问题的解应可表示成一个 n-元组 $(x_0, x_1, \ldots, x_{n-1})$，其中每个分量 x_i 取自某个值集 S，所有允许的 n-元组组成一个候选集。问题中应给出用于判定一个候选解是否是可行解的约束条件，满足约束条件的候选解称为可行解。同时还给定一个数值函数称为目标函数，用于衡量每个可行解的优劣，使目标函数取最大 (或最小) 值的可行解为最优解。

贪心法是一种求解最优化问题的算法设计策略。贪心法是通过分步决策的方法来求解问题的。贪心法在求解问题的每一步上做出某种决策，产生 n-元组解的一个分量。贪心法要求根据题意，选定一种最优量度标准，作为选择当前分量值的依据。这种在贪心法每一步上用做决策依据的选择准则被称为最优量度标准或贪心准则，也称贪心选择性质。这种量度标准通常只考虑局部最优性。

在初始状态下，解向量 $solution = \emptyset$，其中未包含任何分量。使用最优量度标准，一次选择一个分量，逐步形成解向量 $(x_0, x_1, \ldots, x_{n-1})$。在根据最优量度标准选择分量的过程中，还需要使用一个可行解判定函数。

贪心法可用如下的算法框架描述

```
SolutionType Greedy(SType a[], int n)
{
    //初始时，解向量不包含任何分量
    SolutionType solution = ∅;
    //多步决策，每次选择解向量的一个分量
    for(int i = 0; i < n; i++)
    {
        //遵循最优量度标准选择一个分量
        SType x = Select(a);
        //判定加入新分量x后的部分解是否可行
        if(Feasible(solution, x))
        {
            //形成新的部分解
```

```
            solution = Union(solution, x);
        }
    }
     //返回生成的最优解
    return solution;
}
```

由于贪心策略并不是从整体上加以考虑的，它所做出的选择只是当前看似最佳的选择，这种选择仅依赖于以前的选择，但不依赖于以后的选择。对于一个具体的应用问题，无法确保贪心法一定产生最优解。因此，对于一个贪心算法，必须进一步证明该算法的每一步上所做出的选择，都必然最终导致问题的一个整体最优解。

4.2 背包问题

问题描述

已知一个载重为 M 的背包和 n 件物品，第 i 件物品的重量为 w_i，如果将第 i 件物品全部装入背包，将有收益 p_i，这里，$w_i > 0, p_i > 0, 0 \le i \le n$。所谓背包问题，是指求一种最佳装载方案，使得收益最大。

1. 物品不能分割 (0/1 背包)
2. 物品能分割 (此处要讲的)

贪心法求解

背包问题的解可以表示成一个 n-元组：$X = \{x_0, x_1, \ldots, x_{n-1}\}, 0 \le x_i \le 1, 0 \le i < n$。约束条件

$$\sum_{i=0}^{n-1} w_i x_i \le M \quad w_i > 0, 0 \le x_i \le 1, 0 \le i < n \tag{4.1}$$

目标函数

$$\max \sum_{i=0}^{n-1} p_i x_i \quad p_i > 0, 0 \le x_i \le 1, 0 \le i < n \tag{4.2}$$

贪心准则：选择使单位重量收益最大的物品装入背包。

4.3 带时限的作业排序

问题描述

设有一个单机系统、无其他资源限制且每个作业运行相等时间，不妨假定每个作业运行 1 个单位时间。现有 n 个作业，每个作业都有一个截止期限 $d_i > 0, d_i$ 为整数。如果作业能够在截止期限之内完成，可获得 $p_i >$ – 的收益。问题要求得到一种作业调试方案，该方

案给出作业的一个子集和该作业子集的一种排列，使得若按照这种排列次序调度作业运行，该子集中的每个作业都能如期完成，并且能够获得最大收益。

贪心法求解

设 n 个作业以编号 0 ~ $n-1$ 标识，每个作业有唯一的作业编号，$I = \{0, 1, \ldots, n-1\}$ 是 n 个输入作业的集合。带时限作业排序问题的解是 I 的一个子集 X，可表示成一个 $n-$元组：$X = (x_0, x_1, \ldots, x_{r-1}), 0 < r \leq n$。每个 $x_i (0 \leq x_i \leq n-1)$ 是一个作业编号。

贪心准则：一种直观而局部的想法是选择一个作业加入部分解向量中，在不违反截止时限的前提下，使得至少就当前而言，已选入部分解向量中的那部分作业的收益之和最大。为满足这一最度标准，只需先将输入作业集合 I 中的作业按收益的非增次序排列，即 $p_0 \geq p_1 \geq \cdots \geq p_{n-1}$。

带时限作业排序的贪心算法

```
void GreedyJob(int d[], Set X, int n)
{
    //前置条件: 按收益非增排序
    X = {0};
    for(int i=1; i<n; i++)
        if(集合 X and {i}中作业都能在给定的时限内完成)
            X = X and {i};
}
```

一种改进算法

改进的可行解判定方法的基本思想是：令 $b = \{n, \max\{d_i | 0 \leq i \leq n-1\}\}$，b 是一种可行的作业调度方案所需的最大时间。

最优量度标准仍然采取使得部分解向量的收益最大这一准则，设作业已按收益的非增次序排列，作业 i 的时限是 d_i，为它所分配的时间片是 $[\gamma - 1, \gamma]$，其中 γ 是使 $0 \leq \gamma \leq d_i$ 的最大整数且时间片 $[\gamma - 1, \gamma]$ 是空闲的。具体做法是：为收益最大的作业 0 分配时间片 $[d_0 - 1, d_0]$，为收益次大的作业 1 分配作业时，首先考虑时间片 $[d_1 - 1, d_1]$，如果该时间片已分配，再考虑前一个时间片 $[d_1 - 2, d_1 - 1]$，依次向前寻找第一个空闲的时间片分配之。

总之，这种方法采取的作业调度原则是尽可能推迟一个作业的执行时间。

4.4 最佳合并模式

问题描述

在数据结构中介绍的构造哈夫曼树的哈夫曼算法和设计 K 路合并外排序最佳方案的方法都属于最佳合并模式问题。两路合并外排序算法通过反复执行将两个有序子文件合并成一个有序文件的操作，最终将 n 个长度不等的有序子文件合并成一个有序子文件。

贪心法求解

两路合并树表达的合并方案确定了合并排序过程中所需读/写的记录总数，这个量正是该两路合并树的带权外路径长度。带权外路径长度是针对扩充二叉树而言的。扩充二叉树中除叶子结点外，其余结点都必须有两个孩子。扩充二叉树的带权外路径长度定义为：

$$WPL = \sum_{k=1}^{m} w_k l_k \tag{4.3}$$

式中，m 是叶子结点的个数，w_k 是第 k 个叶子结点的权，l_k 是从根到叶子结点的路径长度。

贪心法是一种多步决策的算法策略，一个问题能够使用贪心法求解，除了具有贪心法问题的一般特性外，关键问题是确定最优量度标准。两路合并最佳模式问题的最优量度标准为带权外路径长度最小。具体做法是在有序子文件集合中，选择两个长度最小的子文件合并之。

4.5 最小代价生成树

问题描述

一个无向连通图的生成树是一个极小连通子图，它包括图中全部结点，并且有尽可能少的边。一棵生成树的代价是树中各条边止的代价之和。一个网络的各生成树中，具有最小代价生成树称为该网络的最小代价生成树 (minimum-cost spanning tree)。

贪心法求解

将贪心策略用于求解无向连通图的最小代价生成树时，核心问题是需要确定贪心准则。根据最优量度标准，算法的每一步从图中选择一条符合准则的边，共选择 n-1 条边，构成无向连通图的一棵生成树。贪心法求解的关键是该量度标准必须足够好。它应当保证依据此准则选出 n-1 条边构成原图的一棵生成树，必定是最小代价生成树。

最简单的最优量度标准是：选择使得迄今为止已入选 S 中边的代价之处增量最小的边。对于最优量度标准的不同解释将产生不同的构造最小代价生成树算法。对于上述量度标准有两种可能的理解，它们是普里姆 (Prim) 算法和克鲁斯卡尔 (Kruskal) 算法。

克鲁斯卡尔算法的贪心准则是：按边代价的非减次序考察 E 中的边，从中选择一条代价最小的边 $e = (u, v)$。这种做法使得算法在构造生成树的过程中，边集 S 代表的子图不一定是连通的。普里姆算法的贪心准则是：在保证 S 所代表的子图是一棵树的前提下选择一条最小代价的边 $e = (u, v)$。

4.6 单源最短路径

最短路径是另一种重要的图算法。有两类不同的最短路径问题：单源最短路径问题和所有结点间的最短路径问题。对于这两类问题，存在不同的求解算法。

问题描述

单源最短路径问题是：给定带权的有向图 $G=(V,E)$ 和图中结点 $s\in V$，求从 s 到其余各结点的最短路径，其中，s 称为源点。

贪心法求解

用贪心法的观点看，从源点到另一个结点的任何一条路径均可视为一个可行解，其中长度最短的路径是从源点到该结点的最短路径。从源点到其余每个结点的最短路径构成了单源最短路径问题的最优解。

迪杰斯特拉 (Dijkstra) 提出了按路径长度的非递减次序逐一产生最短路径的算法：首先求得长度最短的一条最短路径，再求得长度次短的一条最短路径，其余类推，直到从源点到其他所有结点之间的最短路径都已求得为止。也就是说，对于最终求得的最优解 $L=(L_1,L_2,\ldots,L_{n-1})$，算法先求得其中最短路径，然后再求次短的.....

设 $S=\{v_0,v_1,\ldots,v_k\}$ 是已经求得的最短路径的结点集合，一个结点 v_i 属于 S 当且仅当从源点 s 到 v_i 的最短路径已经计算。单源最短路径的最优量度标准是：使得从 s 到 S 的所有结点的路径长度之和增量最小。所以迪杰斯特拉算法总是在集合 $V-S$ 中选择“当前最短路径”长度最小的结点加入集合 S 中。

4.7 磁带最优存储

单带最优存储问题描述

设有 n 个程序编号分别为 $0,1,\ldots,n-1$，要存放在长度为 L 的磁带上，程序 i 在磁带上存储长度为 $a_i, 0\le i<n, \sum_{i=0}^{i-1}a_i\le L$。假定存放在磁带上的程序随时可能被检索，且磁带在每次检索前均已倒带到最前端。那么，如果 n 个程序在磁带上的存放次序为 $\gamma=(\gamma_0,\gamma_1,\ldots,\gamma_{n-1})$，则检索程序 γ_k 所需时间 t_k 与 $\sum_{i=0}^{k}a_{\gamma_i}$ 成正比。假定每个程序被检索的概率相等，则平均检索时间 (mean retrieval time,MRT) 定义为 $\frac{1}{n}\sum_{k=0}^{n-1}t_k$。单带最优存储问题是求这 n 个程序的一种排列，使得 MRT 有最小值。这也等价于求使 $D(\gamma)=\sum_{k=0}^{n-1}\sum_{i=0}^{k}a_{\gamma_i}$ 有最小值的排列。

贪心法求解

贪心法通过逐个选择解向量的分量求解问题。一个问题可用贪心法求解的关键是设计最优量度标准。一种可以考虑的量度标准是：计算迄今为止已选的那部分程序 D 值，选择下一个程序的标准应使得该值的增加最小。容易看到，这一量度标准等价于将程序按长度非减次序排列后依次存放。

多带最优存储问题描述

可以将单带存储问题扩展到多带存储问题。设有 $m(m>1)$ 条磁带 $T_0,\ldots,T_{m-1}$ 和 n 个程序，要求将这 n 个程序分配到这 m 条磁带上，令 $I_j(0\le j<m)$ 是存放在第 j 条磁带上

的程序子集的某种排列，$D(I_j)$ 的定义与前面相同，那么，求 m 条磁带上检索一个程序的平均检索时间的最小值等价于求 $\sum_{0\le j<m} D(I_j)$ 的最小值。令 $TD = \sum_{0\le j<m} D(I_j)$。多带最优存储问题是求 n 个程序在 m 条磁带上的一种存储方式，使得 TD 有最小值。

贪心法求解

在多带情况下计算最优平均检索时间，可以先将程序按长度的非减次序排列，即 $a_0 \le a_1 \le \cdots \le a_{n-1}$，其中 $a_i(0 \le i < n)$ 是程序 i 的长度。从程序 0 和磁带 T_0 开始分配，一般将程序 i 存放在磁带 i mod m 上。

4.8 贪心法的要素

贪心法被用于求解一类最优化问题，它使用多步决策求解方法，根据选定的最优量度标准（也称贪心准则），每次确定问题解的一个分量。由于贪心法每步所做的选择只是当时的最佳选择，因此并不一定总能产生最优解。一般说来，适于用贪心法求解的问题大都具有下面两个特性：最优量度标准和最优子结构。

最优量度标准

是使用贪心法求解问题的核心问题。贪心法的当前选择可能会依赖于已经做出的选择，但不依赖于尚未做出的选择和子问题，因此它的特征是自顶向下，一步一步地做出贪心决策。虽然贪心算法的每步选择也将问题简化为一个规模更小的子问题，但由于贪心算法每步选择并不信赖子问题的解，每步选择只按最优量度标准进行，因此，对于一个贪心算法，必须证明所采用的最度标准能够导致一个整体最优解。

最优子结构

当一个问题的最优解中包含了子问题的最优解时，则称该问题具有最优子结构特性。一般而言，如果一个最优化问题的解结构具有元组形式，并具有最优子结构特性，我们可以尝试选择量度标准。如果经证明 (一般是归纳法)，确认该量度标准能导致最优解，便可容易地按算法框架设计出求解该问题的具体的贪心算法。

并非对所有具有最优子结构特性的最优化问题，都能够幸运地找到最优量度标准，此时考虑用动态规划法求解。

4.9 本章小结

贪心法是求解最优化问题的非常有用的算法设计技术。一个问题能够使用贪心策略的条件是该问题的解是向量结构的，具有最优子结构特性，还要求能够通过分析问题获取最优量度标准。但是，按照该量度标准依次生成解的分量所形成的解是否确实是最优解仍需证明。

第 5 章 动态规划法

动态规划是另一种求解最优化问题的重要算法设计策略。对于一个问题，如果能从较小规模子问题的最优解求得较大规模同类子问题的最优解，最终得到给定问题的最优解，这就是问题最优解的最优子结构特征。最优子结构特性使动态规划算法可以采用自底向上的方式进行计算。如果能在求解中保存已计算的子问题的最优解，当这些子最优解被重复引用时，则无须再次计算，从而节省了大量的计算时间。

5.1 一般方法和基本要素

动态规划的实质也是将较大问题分解为较小的同类子问题，在这一点上它与分治法和贪心法类似。但动态规划法有自己的特点。分治法的子问题相互独立，相同的子问题被重复计算，而动态规划法解决了这种子问题重叠现象。贪心法要求针对问题设计最优量度标准，但这在很多情况下并不容易做到，而动态规划法利用最优子结构，自底向上从子问题的最优解逐步构造出整个问题的最优解，动态规划可以处理不具备贪心准则的问题。

一般方法

与贪心法类似，动态规划法是一种求解最优化问题的算法设计策略，它也采用分步决策的方式求解问题。贪心算法在求解问题的每一步上根据最优量度标准做出某种决策。产生 n-元组解的一个分量。用于决策的贪心准则仅依赖于局部的和以前的选择，但不依赖于尚未做出的选择和子问题的解。动态规划法每一步的决策依赖于子问题的解。直观上，为了在某一步上做出的选择，需要先求解若干子问题，再根据子问题的解做出决策，这就使得动态规划法求解问题的方法是自底向上的。

最优性原理指出，一个最优策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，其余决策必定构成最优策略。这便是最优决策序列的最优子结构特性。

设计一个动态规划算法，通常可以按以下 4 个步骤进行：

1. 刻画最优解的结构特性；
2. 递归定义最优解值；
3. 以自底向上方式计算最优解值；
4. 根据计算得到的信息构造一个最优解。

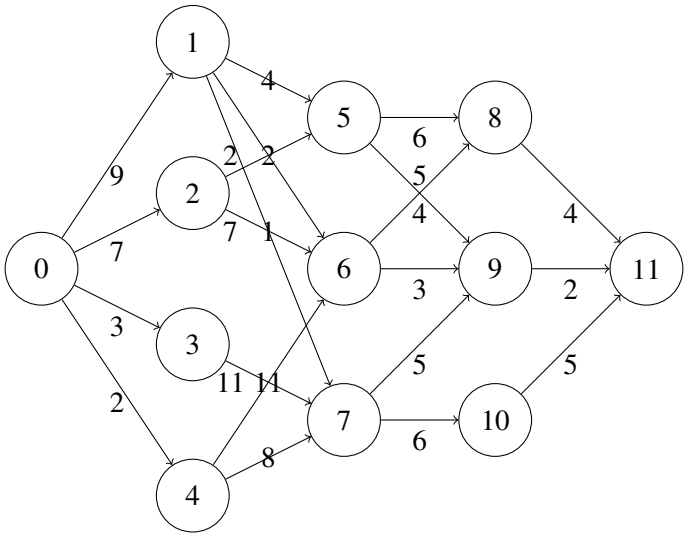
基本要素

一个最优化多步决策问题是否适合用动态规划法求解有两个要素：最优子结构特性和重叠子问题。

动态规划法要求较小子问题的最优解与较大子问题的最优解之间存在数值关系，这就是最优子结构特性。

多段图问题

问题描述：



多段图问题是一种特殊的有向无环图的最短路径问题。

多段图的最优子结构

对于上述多段图，很容易证明它的最优解满足最优子结构特性。

多段图的向前递推关系式

动态规划法每一步的决策信赖于子问题的解。对于多段图问题，一个阶段的决策与后面所要求解的子问题相关，所以不能在某个阶段直接做出决策。但由于多段图问题具有最优子结构特性，启发我们从最后阶段开始，采用逐步向前递推的方式，由子问题的最优解来计算原问题的最优解。由于动态规划法的递推关系是建立在最优子结构的基础上的，相应的算法容易用归纳法证明其正确性。。

$$\begin{aligned} &\text{cost}(k,t) = 0 \\ &\text{cost}(i,j) = \min_{j\in V_i, p\in V_{i+1}, <j,p>\in E}\{c(j,p) + \text{cost}(i+1,p)\}, 0 \le i \le k-2 \end{aligned} \tag{5.1}$$

多段图的重叠子问题

对于多段图显然也存在重叠子问题现象。采用备忘录的方法，可避免重复计算它的值。

多段图的动态规划算法

```
Dijkstra(G, d[], s)
{
    //G为图，一般设成全局变量;
    //数组d为源点到达各点的最短路径长度，s为起点
```

```
    初始化
    for (循环n次)
    {
        u = 使d[u]最小的还未被访问的顶点的标号
        记u已被访问
        for (从u出发能到达的所有顶点v)
        {
            if (v未被访问&&以u为中介点合使s到顶点v的最短距离d[v]更优)
                优化d[v];
        }
    }
}
```

资源分配问题

将 n 个资源分配给 r 个项目，已知如果把 j 个资源分配给第 i 个项目，可以收益 $N(i,j), 0 \le j \le n, 1 \le i \le r$，求总收益最大的资源分配方案。

关键路径问题

关键路径问题是求一个带权有向无环图中两结点间的最长路径问题。关键路径问题是一个 AOE 网络问题。

为了设计求关键路径的动态规划算法，现定义以下 3 个术语：

1. 事件 i 可能的最早发生时间 earliest(i)：是指从开始结点 s 到结点 i 的最长路径的长度。

$$\begin{cases} \text{earliest}(0) & = 0 \\ \text{eariiest}(j) & = \max\limits_{i \in P(j)} \{\text{earliest}(i) + w(i,j)\} \quad 0 < j < n \end{cases} \tag{5.2}$$

2. 事件 i 允许的最迟发生时间 latest(i)：是指在不影响工期的条件下，事件 i 允许的最晚发生时间。

$$\begin{cases} \text{latest}(n-1) & = \text{earliest}(n-1) \\ \text{latest}(i) & = \min\limits_{j \in S(i)} \{\text{latest}(j) - w(i,j)\} \quad 0 < i < n-1 \end{cases} \tag{5.3}$$

3. 关键活动：若 latest(j)-earliest(i)=w(i,j)，则边 $< i, j >$ 代表的活动是关键活动。对关键活动组成的关键路径上的每个结点 i，都有 latest(i)=earlist(i)。

5.2 每对结点间的最短路径

问题描述

每对结点间的最短路径问题是指求图中任意一对结点 i 和 j 之间的最短路径。

动态规划求解

最优子结构

设图 $G=(V,E)$ 是带权有向图，$\delta(i,j)$ 将从结点 i 到结点 j 的最短路径长度，k 是这条路径上的一个结点，$\delta(i,k)$ 和 $\delta(k,j)$ 分别是从 i 到 k 和从 k 到 j 的最短路径长度，则必有 $\delta(i,j)=\delta(i,k)+\delta(k,j)$。这表明每对结点之间的最短路径问题的最优解具有最优子结构特性。

最优解的递推关系

$$\mathrm{d}_{n-1}[i][j]=\min\{\mathrm{d}_{n-2}[i][j],\mathrm{d}_{n-2}[i][n-1]+\mathrm{d}_{n-2}[n-1][j]\} \tag{5.4}$$

弗洛伊德算法

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
const int INF = 100000000;
const int MAXN = 200;//最大顶点数
int n, m ;//n为顶点数，m为边数
int dis[MAXN][MAXN];//dis[i][j]表示顶点i和顶点j的最短距离
void Floyd()
{
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dis[i][k] != INF && dis[k][j] != INF
                && dis[i][k] + dis[k][j] < dis[i][j])
                    dis[i][j] = dis[i][k] + dis[k][j];
            }
        }
    }
}
```

5.3 最长公共子序列

问题描述

给定两个序列 X 和 Y，当另一序列 Z 既是 X 的子序列又是 Y 的子序列时，称 Z 是序列 X 和 Y 的公共子序列 (common subsequence)。

动态规划求解

设 $X=(x_1,x_2,\ldots,x_m)$ 和 $Y=(y_1,y_2,\ldots,y_n)$ 为两个序列，$Z=(z_1,z_2,\ldots,z_k)$ 是它们的最长公共子序列，则

1. 若 $x_m=y_n$，则 $z_k=x_m=y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列；
2. 若 $x_m\neq y_n$ 且 $z_k\neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列；
3. 若 $x_m\neq y_n$ 且 $z_k\neq y_n$，则 Z 是 Y_{n-1} 和 X 的最长公共子序列。

最优解的递推关系：需要使用一个二维数组来保存最长公共子序列的长度，设 $c[i][j]$ 保存 $X_i=(x_1,x_2,\ldots,x_i)$ 和 $Y_j=(y_1,y_2,\ldots,y_j)$ 的最长公共子序列的长度。

$$c[i][j]=\begin{cases}0 & i=0,j=0\\ c[i-1][j-1]+1 & i,j>0,x_i=y_j\\ \max\{c[i][j-1],c[i-1][j]\} & i,j>0,x_i\neq y_j\end{cases}\tag{5.5}$$

最长公共子序列算法

```
#include <cstdio>
#include <algorithm>
#include <cstring>
using namespace std;
const int N = 100;
char A[N],B[N];
int dp[N][N];//表示A的i号和B的j号位之前的LCS长度
int main()
{
    int n;
    fgets(A+1,N,stdin);
    fgets(B+1,N,stdin);
    int lenA = strlen(A+1);
    int lenB = strlen(B+1);
    for (int i = 0; i < lenA; ++i) {
        dp[i][0] = 0;
    }
    for (int j = 0; j < lenB; ++j) {
        dp[0][j] = 0;
    }
    for (int i = 1; i < lenA; ++i) {
        for (int j = 1; j < lenB; ++j) {
            if (A[i] == B[j])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
```

```
27          }
28      }
29      printf("%d",dp[lenA][lenB]);
30      return 0;
31  }
```

5.4 0/1 背包

问题描述

如果物品不能分割，只能作为一个整体或者装入背包，或者不装入背包，称为 0/1 背包问题。

动态规划求解

判断一个问题是否适用于动态规划法求解，首先必须分析问题解的结构，考察它的最优解是否具有最优子结构特性。其次，应当检查分解所得的子问题是否相互独立，是否存在重叠子问题现象。

最优解的递归算法

给定一个 0/1 背包问题实例 KNAP(0,N-1,M)，可以通过对 n 个物品是否加入背包做出一系列决策进行求解，假定变量 $x_i \in \{0,1\}, 0 \le i < n$ 表示对物品 i 是否加入背包的一个决策。假定对这些 x_i 做出决策的次序是 $X = (x_{n-1}, x_{n-2}, \ldots, x_0)$。在对 x_{n-1} 做出决策后，存在两种情况：

1. $x_{n-1} = 1$，将编号为 n-1 的物品加入背包，接着求解子问题 KNAP$(0, n-2, M-w_{n-1})$
2. $x_{n-1} = 0$，将编号为 n-1 的物品不加入背包，接着求解子问题 KNAP$(0, n-2, M)$

上面分析得到

$$f(-1, X) = \begin{cases} -\infty & X < 0 \\ 0 & X \ge 0 \end{cases} \tag{5.6}$$

$$f(j, X) = \max\{f(j-1, X), f(j-1, x-w_j) + p_j\} \quad 0 \le j < n$$

5.5 流水作业调度

最优量度标准

最优子结构

第 6 章 回溯法

第 7 章　分枝限界法

第 8 章 NP 完全问题