1、渐进记号

1.1 O

1.2 Θ

1.3 Ω

1.4 ω

1.5 o

2、分治策略

2.1T(n)=aT(n/b)+f(n) 分成 a 个子问题，子问题的规模是原问题的 1/b，合并分解子问题花费时间为 f(n)

2.2 递归细节：在推导渐进界时可以忽略

若规模不是偶数，却要分为相同的两个子问题。

边界条件：当规模足够小时，该子问题花费的时间会影响递归式的精确解但不会超过一个常数因子，因此可以忽略。

3、最大子数组：数组中存在连续若干元素之和最大，那么这连续若干个元素构成的数组就称为**一个最大字数组**，因为和最大可能有多个解

4、堆，n 个元素的堆的高度为$\lfloor lgn \rfloor$，$\lfloor n/2 \rfloor$+1、$\lfloor n/2 \rfloor$+2…n 为叶节点

堆中节点的高度：该节点到叶节点最长简单路径上边的数目。一个节点构成的堆，高度就为 0

原址排序：算法在数组 A 中重排这些数，在任何时候，最多只有其中的常数个数字存储在数组外面。
稳定排序：具有相同值的元素在输出数组中的相对次序与它们在输入数组中的相对次序相同。

排序算法：

	是否原址	是否稳定
插入排序	是	是
归并排序	不是？	是
堆排序	否	不是
快速排序	是	不是
计数排序	否	是
基数排序	依赖于位的稳定排序算法	是（每一位必须稳定）
桶排序	否	依赖于桶中的排序方法

6、合并两个元素个位 n 个的有序表的下界 2n-1（元素都不相同时）

7、桶排序如何找到对应的桶的编号

若是[a,b)的均匀分布，那么值 i 对应的编号为 $\left\lfloor \frac{i-a}{b-a} * n \right\rfloor$

要找到**某个量是均匀分布**

8、合并 k 个有序链表的复杂度 O(nlgk)（用最大堆或者最小堆）
k 个元素的堆排复杂度为 klgk，有 n 个元素，因此 nklgk=O(nlgk)

9 0-1 排序引理：如果一个遗忘比较算法能够对所有只含 0 和 1 的输入序列排序，那么它也可以对包含任意值的输入序列排序
反证：遗忘比较算法未能对 A[1…n]排序，A[p]是算法 X 未能将其放到正确位置的最小元素，而 A[q]是被算法 X 放在 A[p]原本应该在的位置上的元素。

值	A[q]	A[p]
位置	q	p

9.1 A[p]只能错排在比他原本应在的位置之后的位置，如果错排在之前，那么最小的错排元素就不是 A[p]
9.2 并且 A[q]也是错排的元素，而且 A[q]>=A[p]

10、循环不变式：在每次循环开头，保持不变的式子

Chapter 6 堆排序

1、HEAPIFY(A,i): O(lgn)

2、BUILD-MAX-HEAP(A): O(nlgn).这个上届正确但是不紧确，实际上是 O(n)

2.1 在线性时间内可以把一个无序数组构造成一个最大堆

3、优先队列可以用二叉堆来实现

Chapter 11 散列

解决冲突的方法：链接法、开放寻址法

1、全域散列函数：对于任意一对不同的关键字，k,l∈U,满足 h(k)=h(l)的散列函数 h∈H 的个数至多为|H|/m。换句话说：从 H 中随机选择一个散列函数，当关键字 l≠k 时，两者发生冲突的概率不大于 1/m。（m 是散列表槽的数量）

2、开放寻址法：**核心：探查序列的均匀化**

2.1 均匀散列：每个关键字的探查顺序等可能地为<0,1,…,m-1>的 m！种排列的任意一种

在一次探查中，在 n 个元素被散列至 m 个槽位中的条件下，下一次探查的槽位被占用的概率是 n/m

线性探查：h(k,i)= (h'(k)+i)mod m

二次探查：$h(k,i)=(h'(k)+c_1i+c_2i^2)\bmod m$

双重探查：$h(k,i)=(h_1(k)+ih_2(k))\bmod m$ （**需要保证 $h_2(k)$与 m 互质才能保证 h(k,i)能遍历散列表，一般取 m 为素数，$h_2(k)$只要比 m 小即可；或者 m 为 2 的幂次，$h_2(k)$产生奇数**）

3、完全散列：

3.1 引出：要保证一次散列无冲突：令 $m=n^2$，适当选择哈希函数即可保证无冲突，即一个槽内至多有一个元素（即不存在解决冲突的链表），但是这种方式当 n 很大时，会造成空间的严重浪费

3.2 二次散列：解决 3.1 的冲突问题，且所占空间为 O(n)

第一次散列将 n 个元素散列到含有 m 个槽的散列表中，其中 m=n

第二次散列，利用 $m_j=n_j^2$ 的散列解决冲突

Chapter 14

Chapter 15 动态规划（dynamic programming）

programming 指的是表格

步骤：

1. 刻画一个最优解的结构特征
2. 递归地定义最优解的值
3. 计算最优解的值，通常采用自底向上的方法
4. 利用计算出的信息构造一个最优解

特征：

1、最优子结构：动态编程只能应用于有最优子结构的问题。最优子结构的意思是局部最优解能决定全局最优解(对有些问题这个要求并不能完全满足，故有时需要引入一定的近似)。简单地说，问题能够分解成子问题来解决。

1.1 最优子结构性质：如果一个问题的最优解包含其子问题的最优解，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。

2、问题重叠性质

方法：自底向上求出该问题的子问题的解，该问题的解可以由两个子问题的解求得，分解成两个子问题的方法有若干种（一般为 O(n)）

动态规划用于解决最优解问题：这些问题可以有很多解，每个解都有一个值，希望寻找具有最优值得解。我们称这样的解为问题的一个最优解，而不是最优解，因为可能有多个解都达到最优

1、钢条切割问题

1.1 最优子结构性质：问题的最优解由相关子问题的最优解组合而成（或者说问题可以分解为多个子问题，而构成问题最优解所对应的子问题的解也是各自子问题的最优解），且这些子问题可以独立求解

1.2 只包含一个相关子问题的解：将钢条从左边切割下长度为 i 的一段，只对右边剩下的长度为 n-i 的一段继续进行切割（递归求解）

1.3 动态规划方法：仔细安排求解顺序，对每个子问题之求解一次，并将结果保存下来，付出额外的内存空间来节省计算时间，是典型的时空权衡

1.4 动态规划有两种方法：带备忘的自顶向下法；自底向上法

2、矩阵链乘法（确定代价最低的计算顺序）

1.1 如何得出完全括号化的递推公式：完全括号化的矩阵（n 个相乘）可以描述为两个完全括号化的矩阵的乘积，这两个完全括号化矩阵必须是挨着的，因此，这个分界点共有 n-1 个

3、 子集和问题能否用动态规划来求解？与钢条的切割不同，钢条从左边切割，右边部分必然存在一个值，而子集和则不是，如果包含左边部分，那么右边（剩余部分，可能不可分）

不能：不满足最优子结构性质？不能随意分割和，因为子和可能无法用子集表示

4、在何种情况下使用动态规划

无论如何问题都有一个解，只是需要求出其最优解

4.1 最优子结构

4.1.1 挖掘最优子结构性质的通用模式

①证明问题最优解的第一个组成部分是做出一个选择，做出这次选择会产生一个或多个待解的子问题（如钢条切割）

②对于一个给定的问题，在其可能的第一步选择中，你假定已经知道那种选择才会得到最优解

③给定可获得最优解选择后，你确定这次选择会产生那些子问题，以及如何最好地刻画子问题空间（保持子空间尽可能简单，只在必要时才扩展它）

④利用“剪切—粘贴”技术证明：作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。利用反证法证明：假定子问题的解不是其自身的最优解，那么我们就可以从原问题的解中“剪切”掉这些非最优解，将最优解“粘贴”进去，从而得到一个更优的解，这与最初的解释原问题的最优解的前提假设矛盾

4.1.2 不同问题最优子结构的不同之处：

①原问题的最优解涉及多少个子问题（钢条切割：1； 矩阵链乘法：2）(思考这两种区别)

对于钢条切割，如果分成两个子问题，那么当前问题的最优分割方案有可能是当前问题本身，想要求解当前问题，就要知道当前问题的最优解，这显然有问题

②在确定最优解使用那些子问题时，我们需要考察多少种选择

4.1.3 可以用**子问题总数**和**每个子问题需要考察多少种选择**这两个因素的乘积来粗略分析运行时间

4.1.4 原问题的最优解的代价通常就是**子问题最优解的代价**加上**由此次选择直接产生的代价**

4.2 子问题重叠

4.2.1 子问题空间必须足够“小”

4.2.2 与之相对，分治法求解的问题通常在递归每一步都产生全新的子问题

4.3 重构最优解

4.3.1 一般来说，将每个子问题所做的选择都存在一个表中，这样就不必根据代价值来重构这些信息（对于矩阵链乘法，若没有存储 s[i,j]则要根据 m[i,j]遍历 j-1 种可能分法，从而找出最优分法）

4.3.2 对于自顶向下的备忘算法，**首先要将表项设为特殊值**

4.3.3 **自底向上算法与自带备忘的自顶向下算法复杂度相同，但是由于自底向上算法没有递归，因此常数系数要小于自带备忘的自顶向下算法**

5、最长公共子序列（LCS:longest-commom-subsequence problem）

5.1 子序列的定义：一个给定序列的子序列，就是将给定序列中零个或多个元素去掉后得到的结果

5.2 定义 X 的第 i 前缀为 $X_i=<x_1, x_2, ...,x_i>$

5.3 若 $X=<x_1, x_2, ...,x_m>;Y=<y_1, y_2, ...,y_n>;Z=<z_1, z_2, ...,z_k>$为 X 和 Y 的任意 LCS

1、如果 $x_m= y_n$，那么 $z_k= x_m = y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS。

2、如果 $x_m\neq y_n$，那么 $z_k\neq x_m$，意味着 Z 是 X_{m-1} 和 Y 的一个 LCS。

3、如果 $x_m\neq y_n$，那么 $z_k\neq y_n$，意味着 Z 是和 X 和 Y_{n-1} 的一个 LCS。

5.4 LCS 递归式的建立：

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

5.4.1 递归式思考

钢条切割：常数加子问题；**不可以分成两个问题，为什么？**

归并排序：两个等分的子问题（非动态规划问题，因为不需要一个最优的分法，直接取中点）；

矩阵链乘法：两个子问题；

LCS：长度少一的子问题；

5.5 LCS 不同于钢条切割和矩阵链乘法的一个点：如何对当前问题作出选择，尝试减少 X Y 的长度，分解成子问题（所得到的子问题的类型需要分类讨论）

5.6 b[i,j]= □ 意味着 c[i,j]>c[i-1,j]或 c[i,j]>c[i,j-1]（好像有问题 P225 i=5,j=5 时，没有□而是↑）（考虑 111 与 1111 的公共序列）？？？好像有问题。**如何只用 c[i,j] c[i-1,j] c[i,j-1] c[i-1,j-1]构造最优解？**

5.7 **最长单调子序列：**

5.7.1 **排序后转化为 LCS 问题，太赞了**

5.7.2
```
for (int i = 0; i < A.size(); i++) //c[i]存储的是 A[1...i]序列中的最长子序列，循环前初始化为 1
    for (int j = 0; j < i; j++)
        if (A[i] > A[j] && c[j] + 1>c[i])
            ++c[i];
```

5.7.3 **设计子问题：求所有 c[i]**

c[i]:以 A[i]为末尾元素的子序列的最长单调子序列（该单调子序列一定包含 A[i]）

b[i]: 以 A[i]为末尾元素的子序列的最长单调子序列中第二大元素的索引

进一步，由于 c[i]的存储是无序的，通过 d[c[i]]将其进行排序，就能用二分法查找

5.7.4 **总结：对于如何做出选择以分解为子问题，最长单调序列相比于前几个显得较为特殊，子问题与原问题的形式不同，原问题是最长子单调序列，而子问题是，以该序列末尾元素结尾的最长单调子序列**

5.8 **最优二叉搜索树**

5.8.1 该问题与矩阵链乘法中分割成子问题所作出的选择相似，即在序列中挑一个点作为根节点，左右两边即变为了子问题

Chapter 16 贪心算法

1、概念：

1.1 在每一步都做出当时看起来最佳的选择，即总是做出局部最优的选择

2、活动选择问题

2.1 **贪心选择：每次选择结束最早的活动**

2.2 **该最早结束的活动必然可以替换掉当前子问题中最大兼容集合 A_{ij} 的第一项活动，因此最早结束的活动必然存在于某一个或多个最优解中**

2.3 **贪心算法通常是这种自顶向下的设计**：做出一个选择，然后求解剩下那个子问题，而不是自底向上求解出很多子问题，然后在作出选择

3、贪心算法步骤

A、**将最优化问题转化为这样的形式：对其作出一次选择后，只剩下一个子问题需要求解（钢条切割其实是分为两个部分，其中一个是子问题，一个是被切割长度的价值，并不能直接看出最优解）**

B、**证明作出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的**

C、**证明作出贪心选择后，剩余子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构**

4、贪心选择性质：

4.1 **贪心选择性质：可以通过作出局部最优选择来构造全局最优解。换句话说，在选择时，我们直接作出在当前问题中看来最优的选择，而不必考虑子问题的解**

4.2 贪心算法与动态规划的不同之处：**在动态规划中每个步骤都要进行一次选择，但选择通常依赖于子问题的解；在贪心算法进行选择时可能依赖之前的选择，但是不依赖于任何将来的选择或子问题的解。因此贪心算法在进行第一次选择之前不求解任何子问题**

5、**最优子结构：**如果一个问题的最优解包含其子问题的最优解，则称此问题具有最优子结构性质

5.1 此性质是能否应用于动态规划和贪心方法的关键要素。

6、**0-1 背包问题与分数背包问题 p243：可以用贪心策略求解分数背包问题，但不能求解 0-1 背包问题**

0-1 背包问题的 DP 解法：**O(nW) n 为商品数量 W 为背包总载重量**

第 i 个物品的价值为 p[i],重量为 w[i](暂且重量为整数)

子问题为:前 i 个商品放入容量为 v 的背包（1...i 的若干件）可以获得的最大价值 dp[i][v]

状态转移方程 dp[i][v]=max{dp[i-1][v],dp[i-1][v-w[i]]+p[i]}

若 dp[i][v]=dp[i-1][v]代表不取第 i 件商品，前 i-1 件商品放入容量为 v 的背包的最大价值

若 dp[i][v]=dp[i-1][v-w[i]]+p[i]代表取第 i 件商品，前 i-1 件商品放入容量为 v-w[i]的背包的最大价值

Chapter 17

1、聚合分析

1.1 一个 n 个操作的序列最坏情况下话费的总时间为 T(n)，在最坏情况下，每个操作的平均代价，或摊还代价为 T(n)/n。

2、核算法

2.1 核算法进行摊还分析时，对不同操作赋予不同费用。

2.2 将赋予一个操作的费用称为它摊还代价

3、势能法

3.1 每个操作的摊还代价等于其实际代价加上次操作引起的势能变化

4、表的扩张和收缩

4.1 当满表时进行扩张，当装载因子小于 1/4 时，进行收缩

Chapter18 高级数据结构

B 树：为磁盘存储二专门设计的一类平衡搜索树

1. 介绍

1.1 关键字是唯一整数而且来自集合{0,1,2,…,u-1},这里 u 恰是 2 的幂次。

1.2 B 树类似于红黑树，但是在降低磁盘 I/O 操作数方面要更好一些

1.3 B 树于红黑树的不同之处在于 B 树的节点可以有很多孩子，数个到数千个

1.4 含有 n 个节点的 B 树的高度为 O(lgn)，然而，B 树的严格高度可能比一颗红黑树小得多

2. B 树的定义

一棵 B 树 T 是具有以下性质的有根树（根为 T.root）

A、每个节点含有以下属性

a. x.n，当前存储在节点 x 中的关键字的个数

b. x.n 个关键字本身，$x.key_1,x.key_2,\ldots,x.key_{x.n}$，以非降序排列 $x.key1\leq x.key_2\leq \ldots\leq x.key_{x.n}$

c. x.leaf，一个布尔值，如果 x 是叶节点，则为 TRUE，若 x 是内部节点，则为 FALSE

B、每个内部节点 x 还包含 x.n+1 个指向其他孩子的指针 $x.c_1,x.c_2,\ldots,x.c_{x.n+1}$。叶节点没有孩子，所以它们的 c_i 属性没有定义

C、关键字 $x.key_i$ 对存储在各子树中的关键字范围加以分割，如果 k_i 为任意一个存储在以 $x.c_i$ 为根的子树中的关键字，那么

$$k_1\leq x.key_1\leq k_2\leq x.key_2\leq\ldots\leq x.key_{x.n}\leq k_{x.n+1}$$

D、每个叶节点具有相同的深度，即树的高度为 h

E、每个节点所包含的关键字的个数有上届和下届。用一个被称为 B 树的最小度数的固定整数 t≥2 来表示这些界

a. 除了根节点以外的每个节点必须至少有 t-1 个关键字。因此，除了根节点意外的每个内部节点至少有 t 个孩子。如果树非空，根节点至少有一个关键字

b. 每个节点之多可包含 2t-1 个关键字。因此，一个内部节点至多可有 2t 个孩子。当一个节点恰好有 2t-1 个关键字时，称该节点是满的。

t=2 时的 B 树是最简单的，每个内部节点有 2 个、3 个或 4 个孩子，即一颗 2-3-4 树。

Chapter 32

1、KMP 算法

1.1 模式 P[1...m]，即字符串，长为 m

1.2 辅助函数 π[1...m]：π[q]存储的是对于子模式 P[1...q]而言，其后缀等于前缀的最大长度

例如 abcab 对于’c’，也就是 π[3]而言，后缀’ab’与前缀’ab’相同，且最长，因此 π[3]=2