算法概论

第一讲：基础知识

薛健

Last Modified: 2019.11.10 时间

主要内容

1 引言

1.1 算法简介

什么是算法？

- 词源
  - 古代中国的数学著作《周髀算经》、《九章算术》⇒ 计算的方法 (**算法**)
  - 古代阿拉伯的数学著作“Al-Kitāb al-mukhtaṣar fī hīsāb al-ğabr wa'l-muqābala”（大致意思“The Compendious Book on Calculation by Completion and Balancing”）
  - 作者：Abū 'Abdallāh Muḥammad ibn Mūsā al-Khwārizmī (花拉子米，约 780-850) 花刺子模
  - al-ğabr ⇒ algebra
  - al-Khwārizmī ⇒ Algorism 和 **Algorithm**
- 定义
  - 算法是解某一特定问题的一组有穷规则的序列 (一个计算的具体步骤)。

Euclidean Algorithm

- **欧几里德算法** (Euclidean Algorithm, Euclid's Algorithm)：最早由古希腊数学家欧几里德在其《几何原本》第 VII 和第 X 卷中描述的求最大公约数 (Greatest Common Divisor) 的过程，从 20 世纪 50 年代开始，欧几里德所描述的这一过程被称为“Euclidean Algorithm”，Algorithm 这个术语在学术上便具有了现在的含义。

Algorithm GCD(a, b)

```
1 while b ≠ 0 do
2 |   t ← b;
3 |   b ← a mod b;
4 |   a ← t;
5 end
6 return a;
```

```python
x,y = eval(input('Enter two numbers: '))
if x < y:
   x,y = y,x
r = 1
while r != 0:
   r = x%y
   x = y
   y = r
print(x)
```

欧几里德算法的迭代过程：

$$r_{k-2} = q_k r_{k-1} + r_k$$

因此 $\mathrm{GCD}(a,b)$ 实际上是如下的计算序列：

$$\begin{aligned} a &= q_0 b + r_0 \quad (r_0 < b) \\ b &= q_1 r_0 + r_1 \quad (r_1 < r_0) \\ r_0 &= q_2 r_1 + r_2 \quad (r_2 < r_1) \\ r_1 &= q_3 r_2 + r_3 \quad (r_3 < r_2) \\ &\vdots \end{aligned}$$

当 $r_N = 0$ 时停止，则可得 $\mathrm{GCD}(a,b) = r_{N-1}$。

欧几里德算法的其他版本：

- 基于减法的版本：

Algorithm GCD(a,b)

```
1 if a = 0 then return b ;
2 while b ≠ 0 do
3 |   if a > b then a ← a − b ;
4 |   else b ← b − a ;
5 end
6 return a;
```

这是欧几里德算法的最初表述，其原理基于这样一个定理：两个整数的最大公约数等于其中较小的数和两数差的最大公约数。

- 递归版本：

Algorithm GCD(a,b)

```
1 if b = 0 then return a ;
2 else return GCD(b, a mod b) ;
```

算法的基本特征

Donald E. Knuth:

- **有限性** (Finiteness)：算法在执行有限步之后必须终止
- **确定性** (Definiteness)：算法的每个步骤都有精确的定义，即要执行的每一个动作都是清晰的、无歧义的
- **输入** (Input)：一个算法有 0 个或多个输入，作为算法开始执行前的初始值或初始状态
- **输出** (Output)：一个算法有一个或多个输出，这些输出与输入存在特定关系
- **能行性** (Effectiveness)：算法中的待实现的运算都是基本运算，原则上可由人用纸和笔在有限时间内精确地完成

Donald E. Knuth (唐纳德 · E · 克努特，中文名：高德纳)

世界顶级计算机科学家之一，算法和程序设计技术的先驱，计算机排版系统 TEX 和 METAFONT 的发明者。1974 年图灵奖得主，作为斯坦福大学计算机科学系的荣誉退休教授，他目前正专注于完成其关于计算机科学的史诗性七卷集巨著：“The Art of Computer Programming (TAOCP)”(《计算机程序设计艺术》)。这一伟大工程在 1962 年他还是加利福尼亚理工学院的研究生时就开始了。在完成该项工作之余，他还用了十年时间发明了两个数字排版系统，并编写了六本著作对其做了详尽的解释说明，现在这两个系统已经被广泛地运用于全世界的数学刊物的排版中。

一个例子

问题 假定 E 是包含 n 个元素的序列，给定键值 K，若 K 在序列 E 中，则返回其索引位置，否则返回 -1。

输入 E, n, K：E 为包含 n 个元素的序列 (索引从 0 到 $n-1$)，K 为某给定键值

输出 K 在序列 E 中的位置；或 -1 若未找到

方案 将 K 与序列 E 中的元素逐个比较，直到找到某一匹配或比较完所有序列元素

算法描述

Algorithm SeqSearch$(E[], n, K)$

```
1 for i ← 0 to n − 1 do
2     if E[i] = K then
3         return i;
4     end
5 end
6 return −1;
```

```
import time
def displaytime(func):
    def wrapper(*args):
        t = time.clock()
        result = func(*args)
        e = time.clock()
        print('time cost: >>> '+str(e-t))
        return result
    return wrapper

@displaytime
def search(my_l,word):
    for index,value in enumerate(my_l):
        if value == word:
            print(word+' exist!')
            return index
    return -1 #如果不存在输出-1
my_list = ['a','x','w','b','c','y','k']
word = 'd'
ans = search(my_list,word)
print(ans)
```

算法分析

如何衡量一个算法的效率 (工作量、执行时间.....)？

- 基本操作 (Basic Operation)
  - 元素 (键值) 的比较
- 最坏情况 (Worst-case) 分析
  - 对于特定算法，其执行时间的长短与输入数据的多少 (或问题规模) 有关；
  - 最坏情况下的执行时间是输入数据量 n 的函数 $W(n)$；

因为出现在数列中的概率是q，没出现的概率是(1-q)1)出现的情况：如果出现在第一个数，则时间为1×q出现在第二个数，时间为2×q...总时间为q(1+2+3+...+n)=qn(n+1)/2平均则为1/n×qn(n+1)/2=q(n+1)/22)
不出现：不出现则需要比较n次，n(1-q)总时间上面两者相加即可证毕

  - 在上一例子中，显然 $W(n) = n$

- **平均情况** (Average-case) **分析**
  - 假设 K 出现在序列 E 中的概率为 q
  - 在平均情况下执行时间 $A(n) = n(1 - \frac{1}{2}q) + \frac{1}{2}q$ ($Why?$)
- 最好情况 (Best-case) 分析
  - 在某些特定输入条件下执行时间最短
  - 多数情况下无实际意义

Python源代码

```python
def bin_search(data_list, val):
    low = 0                         # 最小数下标
    high = len(data_list) - 1       # 最大数下标
    while low <= high:
        mid = (low + high) // 2     # 中间数下标
        if data_list[mid] == val:   # 如果中间数下标等于val，返回
            return mid
        elif data_list[mid] > val:  # 如果val在中间数左边，移动high下标
            high = mid - 1
        else:                       # 如果val在中间数右边，移动low下标
            low = mid + 1
    return # val不存在，返回None
ret = bin_search(list(range(1, 10)), 3)
print(ret)
```

算法分析

还有优化的可能吗？

- 是否存在更快的算法？
- 如果序列 E 是有序序列：也叫二分查找
  - 使用折半查找 (Binary Search) 会更快
  - $W(n) = \lceil \lg(n+1) \rceil$ 时间复杂度为O(lgn)以2为底，解释见百度百科
  - 可以证明对于有序序列，折半查找是最快的

时间复杂度无非就是while循环的次数！

总共有n个元素，

渐渐跟下去就是n,n/2,n/4,....n/2^k（接下来操作元素的剩余个数），其中k就是循环的次数

由于你n/2^k取整后>=1

即令n/2^k=1

可得k=log2n,（是以2为底，n的对数）

所以时间复杂度可以表示O(h)=O(log2n)

n/2^k>=1是因为将某段分成w份，w肯定是整数。
所以取=1就是最差情况，得到的k表示
最差情况的时候进行的操作步骤为k次

如何证明算法的正确性 (Correctness)？

几点结论

- 算法 $\neq$ 程序；算法设计 $\neq$ 程序设计
- 数据结构 + 算法 $\Rightarrow$ 程序
- 算法是程序中抽象的、一般化的、本质的部分
- 算法分析一般包括对算法执行时间和占用空间的分析，而时间复杂度是我们首要考虑的问题
  - **时间复杂度**：对算法执行时间的度量
  - **空间复杂度**：对算法执行过程中所占用空间的度量
- 算法复杂度一般表示成输入数据规模的函数，与具体计算环境无关，一般只关心其复杂度的阶
- 对于某一类特定问题来说，解决问题的算法其效率存在某一上限，而我们无论如何改进算法都无法逾越这一上限
  - 问题的复杂度存在下界 (Lower Bounds)
  - 如何找到这个下界？

1.2 关于本课程

关于本课程

- 主要内容：计算机算法设计与分析的一般性理论和方法
  - **基础知识**：背景知识；基本数据结构；算法分析基础
  - **递归与分治法**：递归与数学归纳法；分治法原理及实例分析
  - **排序算法**：各种排序算法设计及其复杂度分析
  - **选择和检索**：选择算法及对手论证法；动态集合搜索及分摊时间分析
  - **高级设计与分析技术**：贪心算法；动态规划；字符串匹配算法
  - **图算法**：图的表示及其基本算法设计和分析
  - **$\mathcal{NP}$-完全问题介绍**：问题分类及多项式时间复杂度；$\mathcal{NP}$-完全性及其证明；$\mathcal{NP}$-完全问题

七部分

- 目的：建立良好的算法理论概念模型，培养使用计算机解决实际问题的能力
- 要求：
  - 掌握算法复杂度分析的基本理论和方法，能够分析各类算法的时间和空间复杂度
  - 掌握算法设计的几类常用策略，能够根据具体问题及时间复杂度要求设计相应的算法

教材和主要参考书目:

[1] Sara Baase and Allen Van Gelder.《计算机算法——设计与分析导论》(第 3 版影印版) (*Computer Algorithms: Introduction to Design and Analysis (3rd Edition)*). Pearson Education, 2000. 高等教育出版社, 2001.7.

[2] Thomas H. Cormen, Charles E. Leiserson, et al. 著, 潘金贵等译.《算法导论》(*Introduction to Algorithms (2nd Edition)*). 机械工业出版社, 2006.9.

[3] Donald E. Knuth. *The Art of Computer Programming - Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley, 1997. 影印版：清华大学出版社, 2002. 中译本：《计算机程序设计艺术 第一卷 基本算法》. 苏运霖译, 国防工业出版社, 2002.

[4] 沈孝钧.《计算机科学与技术学科研究生教材：计算机算法基础 (第 1 版)》(*Essentials of Computer Algorithms*). 机械工业出版社, 2014.

[5] Aditya Y. Bhargava. *Grokking Algorithms*. Manning Publications, 2016. 中译本：《算法图解》. 袁国忠译, 人民邮电出版社, 2017.

实践工具

Python:

- 一种面向对象、解释型计算机编程语言 (高级动态编程语言)
- 具有近二十年的发展历史，成熟且稳定
- 语法简捷和清晰，尽量使用无异义的英语单词
- 设计哲学："优雅"、"明确"、"简单"
- 具备垃圾回收功能，能够自动管理内存使用
- 虚拟机本身几乎可以在所有的操作系统中运行
- 入门："简明 Python 教程" (`http://woodpecker.org.cn/`)

考核方式及其他

- **考核方式：**
  - 期末考试 (60%)：课堂开卷
  - 课后习题 (40%)
    - 每一讲都会留一些课后习题，需要在 deadline 之前提交到课程网站或 ceital@163.com，或提交纸质版
    - 邮件主题及附件命名规则：算法作业 _<序号>_<姓名>_<学号>
    - 尽量采用 pdf 格式文件
    - 允许合作，严禁抄袭！
  - 期末成绩 $=\max($考试成绩, 考试成绩 $\times 60\% +$ 作业成绩 $\times 40\%)$
- **任课教师：**薛健 (工程科学学院)
  - 电话：88256650 (office)，13810307841 (cell)
  - 邮箱：xuejian@ucas.ac.cn
  - Office Hour：每周日上午 9:30~10:30，人文楼 128
  - 课程资源：课程网站：http://sep.ucas.ac.cn

2 背景知识

2.1 集合论

集合 (Set)

- **集合：**具有某种属性的事物的全体，或是一些确定对象的汇合
- $a \in \mathbf{S}$；$a \notin \mathbf{S}$

- **集合的表示：**
  - $\mathbf{S} = \{a, b, c\}$
  - $\mathbf{S} = \{x | x > 3, x \in \mathbf{Z}\}$
- **空集：** $\mathbf{S} = \{\} = \varnothing$
- **子集：** $\forall x \in \mathbf{S}_1, x \in \mathbf{S}_2 : \ \mathbf{S}_1 \subseteq \mathbf{S}_2$
- **交集：** $\mathbf{S} \cap \mathbf{T} = \{x | x \in \mathbf{S} \text{ 且 } x \in \mathbf{T}\}$
- **并集：** $\mathbf{S} \cup \mathbf{T} = \{x | x \in \mathbf{S} \text{ 或 } x \in \mathbf{T}\}$
- **集合的特点：** 确定性、互异性、无序性、完备性

集合的基数 (势，Cardinality)

- **有限集：** 由有限个元素组成的集合
  - 存在一个自然数 n，使得集合 $\mathbf{S}$ 中的元素与集合 $\{1, 2, \cdots, n\}$ 中的元素一一对应，记作 $|\mathbf{S}| = n$
  - 一个包含 n 个元素的集合共有 2^n 个不同子集
  - 一个包含 n 个元素的集合共有 $C_n^k = \frac{n!}{(n-k)!k!}$ 个不同的势为 k 的子集
- **无限集：** 由无限个元素组成的集合
  - **可数集：** 可以与自然数 (正整数) 集合 $\{1, 2, 3, \cdots\}$ 建立一一映射
  - **不可数集：** 与自然数集合不存在一一映射 (基数大于自然数集的基数)
  - **无限集的基数：** 阿列夫数 $\aleph$。自然数集 (可数) 的基数记作 $\aleph_0$；实数集 $\mathbf{R}$ (不可数) 的基数记作 $2^{\aleph_0}$ 或 $\beth_1$

序列 (Sequence) /ˈsiːkwəns/

- **定义：**
  - 序列是按照某种顺序排成一列的对象
  - 其项属于集合 $\mathbf{S}$ 的**有限序列**是一个从 $\{1, 2, \cdots, n\}$ 到 $\mathbf{S}$ 的函数，这里 $n \geq 0$，也称做 n 元组
  - 其项属于 $\mathbf{S}$ 的**无限序列**是从 $\{1, 2, \cdots\}$ (自然数集合) 到 $\mathbf{S}$ 的函数
- **表示：**
  - $\mathbf{S}_1 = (a, b, c)$; $\mathbf{S}_2 = (b, c, a)$; $\mathbf{S}_3 = (a, a, b, c)$; $\mathbf{S}_4 = (a_1, a_2, \cdots, a_n) = (a_n)$
- **一些定义和结论：**

  - 一个给定序列的**子序列**是从给定序列中去除一些元素，而不改变其他元素之间相对位置而得到的
  - 一个元素互不相同的有限序列称做包含相同元素有限集的一个**排列**；一个包含 n 个元素的有限集有 $n!$ 个不同排列
  - 若序列的项属于一个偏序集，则**单调递增序列**就是其中每个项都大于等于之前的项；若每个项都严格大于之前的项，这个序列就是**严格单调递增**的

元组和笛卡儿积

- **元组** (Tuples) 是一个有限序列：
  - 有序对 (x, y)，三元组 (x, y, z)，四元组，五元组
  - k-元组：包含 k 个元素的有限序列
- 集合 $\mathbf{S}$ 和 $\mathbf{T}$ 的**笛卡儿积** (Cartesian/Cross Product) 定义为 $\mathbf{S} \times \mathbf{T} = \{(x, y) \mid x \in \mathbf{S}, y \in \mathbf{T}\}$
- **笛卡儿积的性质**：
  - 对于任意集合 $\mathbf{S}$，根据定义有 $\mathbf{S} \times \varnothing = \varnothing \times \mathbf{S} = \varnothing$
  - 一般来说笛卡儿积不满足交换律和结合律
  - 笛卡儿积对集合的并和交满足分配律：
    - $\mathbf{A} \times (\mathbf{B} \cup \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) \cup (\mathbf{A} \times \mathbf{C})$
    - $(\mathbf{B} \cup \mathbf{C}) \times \mathbf{A} = (\mathbf{B} \times \mathbf{A}) \cup (\mathbf{C} \times \mathbf{A})$
    - $\mathbf{A} \times (\mathbf{B} \cap \mathbf{C}) = (\mathbf{A} \times \mathbf{B}) \cap (\mathbf{A} \times \mathbf{C})$
    - $(\mathbf{B} \cap \mathbf{C}) \times \mathbf{A} = (\mathbf{B} \times \mathbf{A}) \cap (\mathbf{C} \times \mathbf{A})$
  - $|\mathbf{S} \times \mathbf{T}| = |\mathbf{S}||\mathbf{T}|$

关系 (Relations) 和函数 (Functions)

- **关系** (或**二元关系**) 是两个集合笛卡尔积的子集，即 $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{T}$
- 例如：实数集上的“小于”关系可定义为 $\{(x, y) \mid x \in \mathbf{R}, y \in \mathbf{R}, x < y\}$
- **关系的性质** ($\mathbf{R} \subseteq \mathbf{S} \times \mathbf{S}$)：
  - 自反性 (reflexive)：$\forall x \in \mathbf{S}, (x, x) \in \mathbf{R}$
  - 对称性 (symmetric)：$\forall x, y \in \mathbf{S}, (x, y) \in \mathbf{R} \Rightarrow (y, x) \in \mathbf{R}$
  - 传递性 (transitive)：$\forall x, y, z \in \mathbf{S}, (x, y) \in \mathbf{R}, (y, z) \in \mathbf{R} \Rightarrow (x, z) \in \mathbf{R}$
- **等价关系**：同时满足自反性、对称性和传递性的二元关系

- **等价类**：给定一个集合 $\mathbf{S}$ 和在 $\mathbf{S}$ 上的一个等价关系 $\mathbf{R}$，则 $\mathbf{S}$ 中的一个元素 x 的等价类是在 $\mathbf{S}$ 中等价于 x 的所有元素构成的子集，即 $[x] = \{y \in \mathbf{S} \mid x\mathbf{R}y\}$
- 从输入元素集合 $\mathbf{X}$ 到可能的输出元素集合 $\mathbf{Y}$ 的**函数** f (记作 $f : \mathbf{X} \to \mathbf{Y}$) 是 $\mathbf{X}$ 与 $\mathbf{Y}$ 的关系，满足如下条件：
  - f 是完全的：对集合 $\mathbf{X}$ 中任一元素 x 都有集合 $\mathbf{Y}$ 中的元素 y 满足 xfy
  - f 是多对一的：若 xfy 且 xfz，则 $y = z$

2.2 逻辑

逻辑

- **逻辑**是研究“有效推论和证明的原则与标准”的一门学科，做为一门形式科学，逻辑透过对推论的形式系统与自然语言中的论证等来研究并分类命题 (statement) 与论证的结构
- 最简单的命题称为**原子式** (atomic formula)
- 更复杂的命题可以由原子式和**逻辑运算符** (或连接符，logical connectives) 组合生成，基本运算符包括：
  - **非** ($\neg$)：$\neg A$ 为真当且仅当 A 为假
  - **与** ($\wedge$)：$A \wedge B$ 为真当且仅当 A 为真且 B 为真
  - **或** ($\vee$)：$A \vee B$ 为真当且仅当 A 为真或 B 为真或二者都为真
  - **蕴涵** ($\to$ 或 $\Rightarrow$，读作“implies”)：$A \to B$ 读作 “A implies B” 或 “if A then B”
- $A \to B$ 等价于 $\neg A \vee B$
- De Morgan's laws 摩根定律
  - $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
  - $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$

量词 (Quantifier，$\forall$ 和 $\exists$)

- **全称量词** $\forall$，“for all ...”：
  - $\forall x P(x)$ 为真 $\Leftrightarrow$ $P(x)$ 对于任意 (所有) x 为真
- **存在量词** $\exists$，“there exist ...”：
  - $\exists x P(x)$ 为真 $\Leftrightarrow$ 存在 x 使 $P(x)$ 为真
- $\forall x P(x) \Leftrightarrow \neg\exists x(\neg P(x))$
- $\exists x P(x) \Leftrightarrow \neg\forall x(\neg P(x))$
- $\forall x(A(x) \to B(x))$：对于任意 x，如果 $A(x)$ 成立则 $B(x)$ 成立

证明：反例，逆否命题，反证

- **反例证明：**
  - 要证明 $\forall x(A(x) \to B(x))$ 为假，只需举出一个反例，即存在某个 x 使得 $A(x)$ 为真而 $B(x)$ 为假
  - $\neg(\forall x(A(x) \to B(x))) \Leftrightarrow \exists x(A(x) \wedge \neg B(x))$

表述：对于任意的x，当A为真则B为真

存在x，当A与非B同时为真

三大证明方法

- **逆否命题证明：**
  - 要证明 $A \to B$，只需证明 $\neg B \to \neg A$
- **反证：**
  - 要证明 $A \to B$，只需假设 $\neg B$，然后证明 B 本身，即证明 $(A \wedge \neg B) \to B$
  - $A \to B \Leftrightarrow (A \wedge \neg B) \to B$
  - $A \to B \Leftrightarrow (A \wedge \neg B)$ 为假
  - 假设 $(A \wedge \neg B)$ 为真，找到矛盾（如 $A \wedge \neg A$），然后可得结论 $(A \wedge \neg B)$ 为假，即 $A \to B$ 成立

A与非B为真，则B为真

反证法例子

$(B \wedge (B \to C)) \to C$

Proof. 假设 $\neg C$ $\neg C \wedge (B \wedge (B \to C)) \Rightarrow \neg C \wedge (B \wedge (\neg B \vee C)) \Rightarrow \neg C \wedge ((B \wedge \neg B) \vee (B \wedge C)) \Rightarrow \neg C \wedge ((B \wedge C)) \Rightarrow \neg C \wedge C \wedge B \Rightarrow$ 矛盾 $\Rightarrow C$ □

推理规则

- **推理规则**（推论规则）是构造有效推论的方案。这些方案建立在一组叫做前提的公式和叫做结论的断言之间的语法关系。这些语法关系用于推理过程中，新的真的断言从其他已知的断言得出
- **证明系统**形成自一组规则，它们可以被链接在一起形成证明或推导。任何推导都只有一个最终结论，它是要证明或推导的陈述。如果在推导中留下了未满足的前提，则推导就是假言陈述：“如果前提成立，那么结论成立。”
- **常用规则：**
  - **肯定前件式** (Modus ponens)：if $\{B \to C \text{ and } B\}$ then $\{C\}$
  - **否定后件式** (Modus tollens)：if $\{B \to C \text{ and } \neg C\}$ then $\{\neg B\}$
  - **三段论法** (Syllogism)：if $\{A \to B \text{ and } B \to C\}$ then $\{A \to C\}$
  - **否定消除**：if $\{B \to C \text{ and } \neg B \to C\}$ then $\{C\}$

布尔 (代数) 逻辑

- 在包含两个元素 0 (逻辑假) 和 1 (逻辑真) 的集合 $\mathbf{B}$ 上定义了两个二元运算：$\wedge$ (或记作 $\cdot$，逻辑与) 和 $\vee$ (或记作 $+$，逻辑或) 以及一个一元运算 $\neg$ (或记作 $\sim$，逻辑非)
- **闭合：** 如果 $x, y \in \mathbf{B}$，则 $x \wedge y \in \mathbf{B},\ x \vee y \in \mathbf{B},\ \neg x \in \mathbf{B}$
- **单位元：** $x \wedge 1 = x,\ x \vee 0 = x$
- **交换律：** $x \wedge y = y \wedge x,\ x \vee y = y \vee x$
- **结合律：** $x \wedge (y \wedge z) = (x \wedge y) \wedge z,\ x \vee (y \vee z) = (x \vee y) \vee z$
- **分配律：** $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z),\ x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- **吸收律：** $x \vee (x \wedge y) = x,\ x \wedge (x \vee y) = x$
- **互补律：** $x \wedge \neg x = 0,\ x \vee \neg x = 1$

重言式 (Tautology) 和真值表

一个逻辑公式，不论其组成部分的命题变项为真的可能性怎么样，它总是为真。在布尔代数中发现重言式的最简单的方法是使用真值表。

Example 2.1. $(B \wedge (B \to C)) \to C$ 是重言式。

B	C	$B \to C$	$(B \wedge (B \to C))$	$(B \wedge (B \to C)) \to C$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	1	1	1	1

用推理规则证明

$(B \wedge (B \to C)) \to C$

Proof. $(B \wedge (B \to C)) \to C \ \Rightarrow \neg(B \wedge (B \to C)) \vee C \ \Rightarrow \neg(B \wedge (\neg B \vee C)) \vee C$ $\Rightarrow \neg((B \wedge \neg B) \vee (B \wedge C)) \vee C \ \Rightarrow \neg(B \wedge C) \vee C \ \Rightarrow \neg B \vee \neg C \vee C \ \Rightarrow$ True (tautology) □

2.3 概率

事件 (Event)

- **单位事件：** 在一次随机试验中可能发生的不能再细分的结果
- **事件空间:** 在随机试验中可能发生的所有单位事件的集合 $\mathrm{U} = \{s_1, s_2, \cdots, s_k\}$
- **单位事件 s_i 概率：** 取实数 $P(s_i)$ 使得

  1. $0 \le P(s_i) \le 1,\ 1 \le i \le k$

2. $P(s_1) + P(s_2) + \cdots + P(s_k) = 1$

- **随机事件**：事件空间的子集 $\mathrm{S} \subseteq \mathrm{U}$，其概率 $P(\mathrm{S}) = \sum_{s_i \in \mathrm{S}} P(s_i)$
- **必然事件**：$\mathrm{U} = \{s_1, s_2, \cdots, s_k\},\ P(\mathrm{U}) = 1$
- **不可能事件**：$\varnothing,\ P(\varnothing) = 0$
- **互补事件**："$\bar{\mathrm{S}}$"，$\mathrm{U} - \mathrm{S}$，$P(\bar{\mathrm{S}}) = P(\mathrm{U} - \mathrm{S}) = 1 - P(\mathrm{S})$

条件概率

- **条件概率**：事件 S 在另外一个事件 T 已经发生条件下的发生概率，表示为 $P(\mathrm{S} \mid \mathrm{T})$，读作"在 T 条件下 S 的概率"
- **联合概率**：两个事件共同发生的概率，表示为 $P(\mathrm{S} \cap \mathrm{T})$ 或 $P(\mathrm{S}, \mathrm{T})$
- **条件概率的计算**：

$$P(\mathrm{S} \mid \mathrm{T}) = \frac{P(\mathrm{S} \cap \mathrm{T})}{P(\mathrm{T})} = \frac{\sum_{s_i \in \mathrm{S} \cap \mathrm{T}} P(s_i)}{\sum_{s_j \in \mathrm{T}} P(s_j)}$$

- **统计独立性**：当且仅当两个随机事件 S 与 T 满足 $P(\mathrm{S} \cap \mathrm{T}) = P(\mathrm{S})P(\mathrm{T})$，称它们是统计独立 (statistically independent) 或随机独立 (stochastically independent)

随机变量及其期望值

- **随机变量**：事件空间上的实值函数 $f : \mathrm{U} \to \mathbf{R}$，即为事件空间中的每一个单位事件 e 赋予一个实数值 $f(e)$
  - 例如：随机掷两个骰子，整个事件空间可以由 36 个元素组成：$\mathrm{U} = \{(i, j) \mid i = 1, \cdots, 6; j = 1, \cdots, 6\}$，这里可以构成多个随机变量，比如随机变量 X：获得的两个骰子的点数和；或者随机变量 Y：获得的两个骰子的点数差绝对值。随机变量 X 可以有 11 个整数值，而随机变量 Y 只有 6 个整数值
- **随机变量的期望值**：是试验中每次可能结果的概率乘以其结果的总和；换句话说，期望值是随机试验在同样的机会下重复多次的结果计算出的等同"期望"的平均值

$$E(f) = \sum_{e \in \mathrm{U}} f(e)P(e)$$

条件期望和期望定律

- **条件期望**：在给定事件 S 已经发生的条件下，随机变量 f 的期望值

$$E(f \mid \mathrm{S}) = \sum_{e \in \mathrm{S}} f(e)P(e \mid \mathrm{S})$$

- **期望定律**：f 和 g 是定义在事件空间 U 上的随机变量，则对任意随机事件 S，有

$$E(\alpha f + \beta g) = \alpha E(f) + \beta E(g)$$

$$E(f) = P(\mathrm{S})E(f \mid \mathrm{S}) + P(\bar{\mathrm{S}})E(f \mid \bar{\mathrm{S}})$$

2.4 代数

不等式和取整函数

- **不等式的基本性质**
  - **传递性**：if $(a \le b)$ and $(b \le c)$ then $(a \le c)$
  - **可加性**：if $(a \le b)$ and $(c \le d)$ then $(a + c \le b + d)$
  - **乘法 (正向缩放)**：if $(a \le b)$ and $(\alpha > 0)$ then $(\alpha a \le \alpha b)$
- **取整函数** (Floor and Ceiling Functions)
  - **向下取整**：$\lfloor x \rfloor$ 或记作 $Floor(x)$：小于等于 x 的最大整数
  - **向上取整**：$\lceil x \rceil$ 或记作 $Ceiling(x)$：大于等于 x 的最小整数

关于取整函数

- 常用计算规则
  - $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$
  - $\lfloor -x \rfloor = -\lceil x \rceil$; $\lceil -x \rceil = -\lfloor x \rfloor$
  - $\lfloor x \rfloor = n \iff n \le x < n + 1$
  - $\lfloor x \rfloor = n \iff x - 1 < n \le x$
  - $\lceil x \rceil = n \iff n - 1 < x \le n$
  - $\lceil x \rceil = n \iff x \le n < x + 1$
  - $\lfloor x + n \rfloor = \lfloor x \rfloor + n$; $\lceil x + n \rceil = \lceil x \rceil + n$
  - $x < n \iff \lfloor x \rfloor < n$
  - $n < x \iff n < \lceil x \rceil$
  - $x \le n \iff \lceil x \rceil \le n$
  - $n \le x \iff n \le \lfloor x \rfloor$
  - $\left\lfloor \sqrt{\lfloor x \rfloor} \right\rfloor = \lfloor \sqrt{x} \rfloor$; $\left\lceil \sqrt{\lceil x \rceil} \right\rceil = \lceil \sqrt{x} \rceil$

关于取整函数 (cont.)

Theorem 2.1. $f(x)$ 连续、递增，且满足：$f(x) \in \mathbf{N} \longrightarrow x \in \mathbf{N}$，则：

1. $\lfloor f(\lfloor x \rfloor) \rfloor = \lfloor f(x) \rfloor$
2. $\lceil f(\lceil x \rceil) \rceil = \lceil f(x) \rceil$

由此，可得：

$$\left\lfloor \frac{\lfloor x \rfloor + m}{n} \right\rfloor = \left\lfloor \frac{x+m}{n} \right\rfloor; \quad \left\lceil \frac{\lceil x \rceil + m}{n} \right\rceil = \left\lceil \frac{x+m}{n} \right\rceil$$

参考：

[1] Ronald L. Graham, Donald E. Knuth, Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* 《具体数学：计算机科学基础》(英文版 · 第 2 版), 机械工业出版社, 2002.8

对数

- **对数：** $\log_b x$ 称为为以 b 为底的 x 的对数，其中 $b > 0, b \neq 1, x > 0$，它的值是使得 $b^L = x$ 成立的实数 L
- **对数的性质：**
  - 对数函数 $\log_b x$ 是严格单调函数，当 $0 < b < 1$ 时单调递减，当 $b > 1$ 时单调递增
  - 对数函数是单射，即 $\log_b x = \log_b y \Rightarrow x = y$，其反函数为 b^x
  - $\log_b 1 = 0;\ \log_b b = 1;\ \log_b x^a = a \log_b x$
  - $\log_b xy = \log_b x + \log_b y$
  - $x^{\log y} = y^{\log x}$
  - $\log_b x = \dfrac{\log_a x}{\log_a b}$

数列与级数

- **数列：** 按照某种规则排列的一组数，如 $a_1, a_2, \cdots, a_n, \cdots$
- **级数：** 数列求和，即 $a_1 + a_2 + \cdots + a_n + \cdots$，记作 $\displaystyle\sum_{k=1}^{\infty} a_k$
- **常用级数的部分和**
  - $\displaystyle\sum_{i=1}^{n} i = \frac{1}{2} n(n+1)$
  - $\displaystyle\sum_{i=1}^{n} i^2 = \frac{1}{6} n(n+1)(2n+1) \approx \frac{n^3}{3},\ \sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1}$
  - $\displaystyle\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ $\qquad$ $\displaystyle\sum_{i=k}^{n} 2^i = 2^{n+1} - 2^k$
  - $\displaystyle\sum_{i=1}^{n} i 2^i = (n-1)2^{n+1} + 2$

单调函数与凸函数

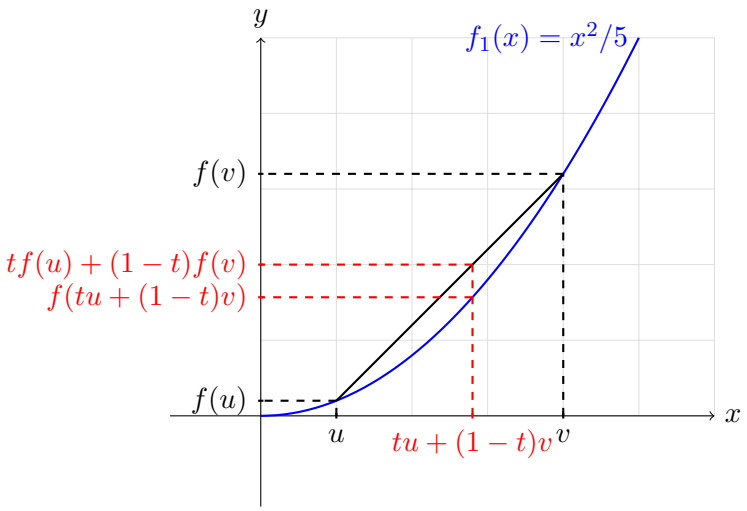
Definition 2.2 (单调函数 (monotonic function)). $f(x)$ 为单调增函数，当且仅当 $x \le y \Rightarrow f(x) \le f(y)$；$f(x)$ 为单调减函数，当且仅当 $-f(x)$ 为单调增函数。

Definition 2.3 (凸函数 (convex function)). 若函数 $f(x)$,$\forall u, v \in$ 定义域 $\mathbf{D}, t \in [0,1]$ 都有

$$f(tu + (1-t)v) \le tf(u) + (1-t)f(v)$$

则称 $f(x)$ 为凸 (convex) (下凸，上凹) 函数；若 $-f(x)$ 为凸函数，则称 $f(x)$ 为凹 (concave) (上凸，下凹) 函数

单调函数与凸函数 (图示)



凸函数的判定

Lemma 2.4. *1.* $f(x)$ 是实数集 $\mathbf{R}$ 上的连续函数，则 $f(x)$ 为凸函数当且仅当 $\forall u, v \in \mathbf{R}$

$$f\left(\frac{1}{2}(u+v)\right) \le \frac{1}{2}\left(f(u) + f(v)\right)$$

2. 定义在整数集 $\mathbf{Z}$ 上的函数 $f(n)$ 为凸函数，当且仅当 $\forall n \in \mathbf{Z}$ 有

$$f(n+1) \le \frac{1}{2}\left(f(n) + f(n+2)\right)$$ 相当于计算n,n+2上的函数点和n+1这个中间点的比较

单调函数和凸函数的判定

Lemma 2.5. *1.* $f(n)$ 是定义在整数集上的函数，$f^*(x)$ 是 $f(n)$ 采用相邻点的线性插值在实数集上的扩展，则有

(a) $f(n)$ 是单调增 (减) 函数当且仅当 $f^*(x)$ 是单调增 (减) 函数

(b) $f(n)$ 是凸函数当且仅当 $f^*(x)$ 是凸函数

2. 如果 $f(x)$ 的一阶导数 $f'(x)$ 存在且非负，则 $f(x)$ 为单调增函数 导数=0为极值点情况

如y=x^2,(x>=0)

3. 如果 $f(x)$ 的一阶导数 $f'(x)$ 存在且为单调增函数，则 $f(x)$ 为凸函数

4. 如果 $f(x)$ 的二阶导数 $f''(x)$ 存在且非负，则 $f(x)$ 为凸函数

用积分求和

- 一些常用积分公式

$$\int_0^n x^k dx = \frac{1}{k+1} n^{k+1}, \quad \int_0^n e^{ax} dx = \frac{1}{a}(e^{an} - 1),$$

$$\int_1^n x^k \ln(x) dx = \frac{1}{k+1} n^{k+1} \ln(n) - \frac{1}{(k+1)^2} n^{k+1}$$

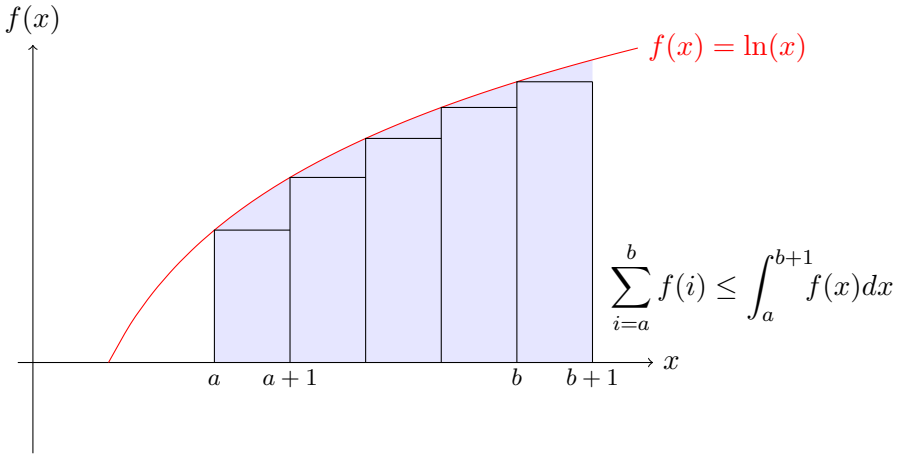
- 对于一些数列的求和，使用以下结论往往可以用积分来计算其近似值 (或给出其上下界)
  - 若 $f(x)$ 单调递增，即 $x \le y \Rightarrow f(x) \le f(y)$，则

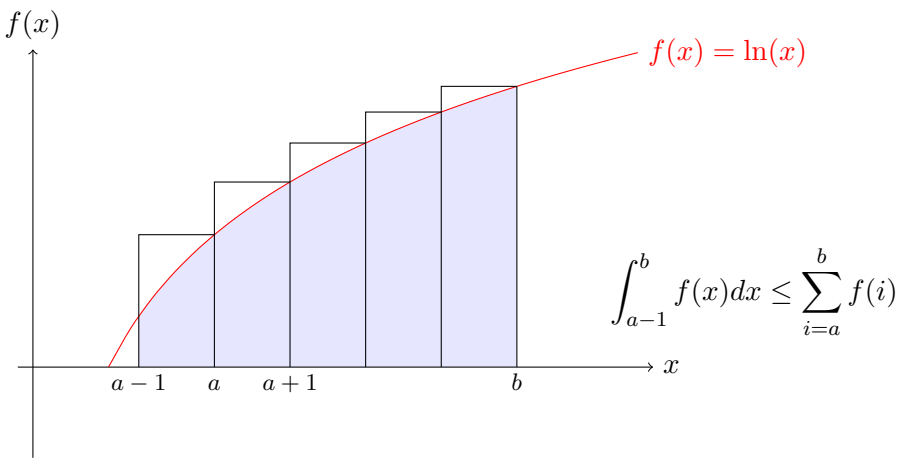
$$\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_a^{b+1} f(x)dx$$

  - 类似地，若 $f(x)$ 单调递减，则

$$\int_a^{b+1} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a-1}^{b} f(x)dx$$

用积分求近似和 (图示)





用积分求近似和

Example 2.2. 求 $\sum_{i=1}^{n}\frac{1}{i}$ 近似上下界：

注意：这里f(i)是单调递减，所以跟上面的相反

$$\sum_{i=1}^{n}\frac{1}{i}\le 1+\int_1^n\frac{dx}{x}=1+\ln x\Big|_1^n=1+\ln n-\ln 1=\ln n+1$$

$$\sum_{i=1}^{n}\frac{1}{i}\ge \int_1^{n+1}\frac{dx}{x}=\ln x\Big|_1^{n+1}=\ln(n+1)-\ln 1=\ln(n+1)$$

$$\ln(n+1)\le\sum_{i=1}^{n}\frac{1}{i}\le\ln n+1$$

关于调和级数 $\sum_{i=1}^{\infty}\frac{1}{i}$

- 其部分和称为**调和数** (Harmonic number)，记作 $H_n=\sum_{i=1}^{n}\frac{1}{i}$
- Euler 给出了其积分形式：$H_n=\int_0^1\frac{1-x^n}{1-x}dx$ (用数学归纳法可以证明)
- 一个更精确的近似：$\sum_{i=1}^{n}\frac{1}{i}\approx\ln(n)+\gamma$ 其中 $\gamma=\lim_{n\to\infty}\left[\left(\sum_{i=1}^{n}\frac{1}{i}\right)-\ln(n)\right]\approx 0.5772156649\ldots$ 称为欧拉-马修罗尼常数 (Euler-Mascheroni constant)

用积分求近似和

Example 2.3. 求 $\sum_{i=1}^{n}\lg i$ 下界：

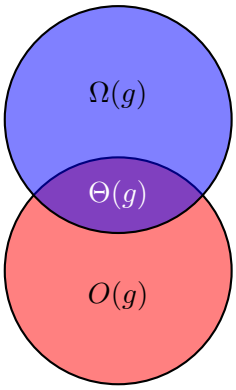
$$\begin{aligned}\sum_{i=1}^{n}\lg i &= 0+\sum_{i=2}^{n}\lg i\ge\int_1^n\lg x dx\\ \int_1^n\lg x dx &= \int_1^n(\lg e)(\ln x)dx=(\lg e)\int_1^n\ln x dx\\ &= (\lg e)(x\ln x-x)\Big|_1^n=(\lg e)(n\ln n-n+1)\\ &= n\lg n-(n-1)\lg e\end{aligned}$$

$$\sum_{i=1}^{n}\lg i\ge n\lg n-1.443(n-1)$$

函数分类

- 根据函数的**渐近增长速度** (asymptotic growth rate)、**渐近阶** (asymptotic order) 或简称**阶** (order) 来对函数进行分类，常数项或增长较慢对结果影响不大的项通常在分析中被忽略
- $\Omega(g)$、$\Theta(g)$ 和 $O(g)$

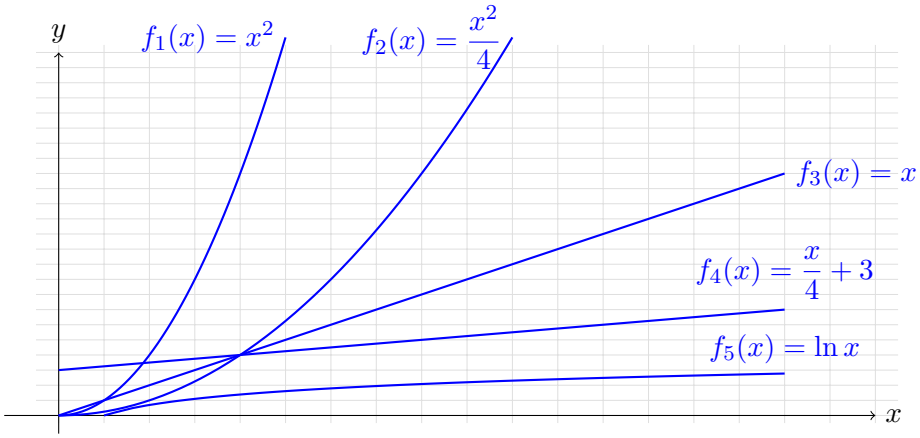
- $\Omega(g)$：增长速度比 g 快，至少一样快的函数
- $\Theta(g)$：增长速度与 g 一样快的函数
- $O(g)$：增长速度比 g 慢，至多一样快的函数



$\Omega(g)$、$\Theta(g)$ 和 $O(g)$

- 在描述算法复杂度时所使用的函数通常是从非负整数集合到非负实数集合的函数
- 函数 f 和 g 是从非负整数集合到非负实数集合的函数，存在实数 $c > 0$ 和非负整数 n_0：
  - $O(g)$ 是所有满足 $\forall n \geq n_0, f(n) \leq cg(n)$ 的函数 f 构成的集合
  - $\Omega(g)$ 是所有满足 $\forall n \geq n_0, f(n) \geq cg(n)$ 的函数 f 构成的集合
  - $\Theta(g) = O(g) \cap \Omega(g)$

渐近阶比较



渐近阶比较

Example 2.4. $f(n) = \dfrac{n^3}{2}$, $g(n) = 37n^2 + 120n + 17$

1. 当 $n \geq 78$ 时，$g(n) < 1f(n)$，所以 $g \in O(f)$
2. $\lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} = \lim\limits_{n\to\infty} (74/n + 240/n^2 + 34/n^3) = 0$

3. 反证法：假设 $f \in O(g)$，则 $\exists c > 0, n_0 \in \mathbf{N}$, $\forall n \geq n_0$, $\frac{n^3}{2} \leq 37cn^2 + 120cn + 17c$，即 $\frac{n}{2} \leq 37c + \frac{120c}{n} + \frac{17c}{n^2} \leq 174c$，因为 c 是常数，所以当 n 很大时该不等式显然不成立，因此得出矛盾，所以 $f \notin O(g)$

渐近阶比较

Lemma 2.6. *1.* $f \in O(g)$ 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$，包括 $c = 0$

2. $f \in \Omega(g)$ 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$，包括 $c = \infty$

3. $f \in \Theta(g)$ 如果 $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = c$，且 $0 < c < \infty$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in o(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f \in \omega(g)$$

渐近阶比较

Theorem 2.7 (洛必达法则 (L'Hôpital's Rule))**.** 函数 f 和 g 可导，且满足 $\lim\limits_{n\to\infty} f(n) = \lim\limits_{n\to\infty} g(n) = 0$ 或 $\pm\infty$，则有

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Example 2.5. $f(n) = n, g(n) = \lg n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{\lg n} = \lim_{n\to\infty} \frac{n}{\ln n/ln2} = \lim_{n\to\infty} \frac{\ln 2}{1/n} = \lim_{n\to\infty} n \ln 2 = \infty$$

$\therefore f \in \Omega(g)$

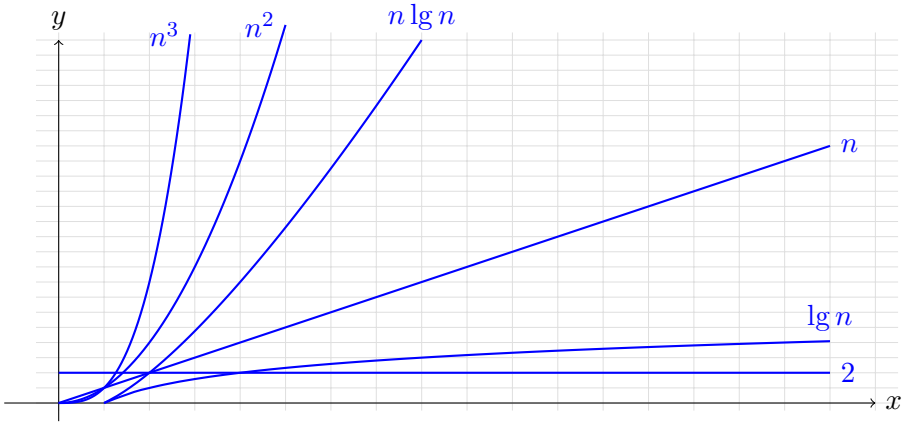
$\Omega(g)$、$\Theta(g)$ 和 $O(g)$ 的性质

- **传递性**：$f \in O(g)$ and $g \in O(h) \Longrightarrow f \in O(h)$
- **自反性**：$f \in \Theta(f)$
- **对称性**：$f \in \Theta(g) \Longrightarrow g \in \Theta(f)$
- Θ 定义了函数空间上的一个等价关系，每一个 $\Theta(f)$ 函数集合是一个等价类
- $f \in O(g) \Longleftrightarrow g \in \Omega(f)$
- $O(f + g) = O(\max(f, g))$，对于 Θ 和 Ω 有类似结论

函数分类

- $O(1)$：渐近阶至多为常数的函数
- $f \in \Theta(n)$：线性阶函数
- $f \in \Theta(n^2)$：平方阶函数
- $f \in \Theta(n^3)$：立方阶函数；
- 对任意常数 $\alpha > 0$ (包括分数)，$\log n \in o(n^\alpha)$
- 对任意常数 $k > 0$ 和 $c > 1$，$n^k \in o(c^n)$
- $\sum_{i=1}^{n} i^d \in \Theta(n^{d+1})$ $\quad$ $\sum_{i=1}^{n} \log i \in \Theta(n \log n)$
- $\sum_{i=a}^{b} r^i \in \Theta(\text{序列最大项})$，$r > 0$ 且 $r \neq 1$，a 和 b 可以是 n 的函数

函数分类



函数渐近阶分析的意义

	解题时间 (假定单位时间为微秒)			
输入规模 (n)	$33n$	$460n \lg n$	$13n^2$	2^n
10	0.00033sec.	0.015sec.	0.0013sec.	0.001sec.
100	0.0033sec.	0.3sec.	0.13sec.	4×10^{16}yr.
1,000	0.033sec.	4.5sec.	13sec.	
10,000	0.33sec.	61sec.	22min.	
100,000	3.3sec.	13min.	1.5days	

线性结构：栈，对列，数组，线性表
树形结构：二叉树，二叉堆
图结构：有向图，无向图
抽象数据类型（Abstract Data Type，ADT）由一种数据结构和在该数据结构上的一组操作所组成

3 基本数据结构

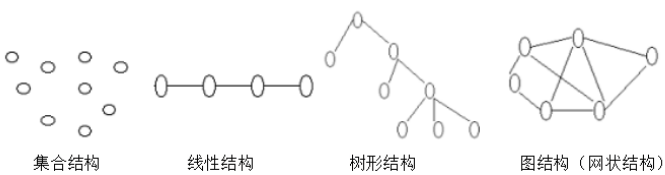
3.1 列表

线性表 (Linear List)

数据结构基础知识

数据结构的知识繁杂，但根据数据元素之间关系的不同特性，通常可以归类为下列四类基本结构：

（1）集合：数据元素间的关系仅仅是同属一个集合。
（2）线性结构：数据元素间存在一对一的关系。
（3）树形结构：结构中的元素间的关系是一对多的关系。
（4）图结构（网状结构）：结构中的元素间的关系是多对多的任意关系。



- **线性表**是由 $n(n \geq 0)$ 个数据元素 (结点) $a_0, a_1, a_2, \cdots, a_{n-1}$ 组成的**有限序列**
  - 数据元素的个数 n 定义为表的长度 $= list.\text{length}()$ ($list.\text{length}() = 0$ (表里没有一个元素) 时称为空表)
  - 非空的线性表记作：$(a_0, a_1, a_2, \cdots, a_{n-1})$
  - 数据元素 a_i $(0 \leq i \leq n-1)$ 只是个抽象符号，其具体含义在不同情况下可以不同
- 一个数据元素可以由若干个数据项组成。数据元素称为**记录**，含有大量记录的线性表又称为文件。
- 这种结构具有下列特点：
  - 存在一个唯一的没有前驱的 (头) 数据元素
  - 存在一个唯一的没有后继的 (尾) 数据元素
  - 每一个数据元素均有一个直接前驱和一个直接后继数据元素
- 线性表的常用存储结构：顺序表 (数组)；链表 (单链表、双链表、循环链表)

广义表

- **广义表**一般记作 $LS = (a_0, a_1, \cdots, a_{n-1})$，$n$ 是它的长度，a_i 可以是单个元素 (原子)，也可以是广义表 (子表)
- 当广义表非空时，称第一个元素 a_0 为**表头**，称其余元素组成的表为**表尾**。
- 表头是元素 (可以是原子，也可以是广义表)，表尾一定是广义表。
- 广义表是一种非线性的数据结构。但如果广义表的每个元素都是原子，它就变成了线性表

Example 3.1. $E = (a, E)$ 是一个递归的表； $D = ((), (e), (a, (b, c, d)))$ 是多层次的广义表，长度为 3，深度为 3； $((a), a)$ 的表头是 (a)，表尾是 (a)，$((a))$ 的表头是 (a)，表尾是 $()$。

3.2 栈与队列

栈 (Stack)

- **栈** (**堆栈**)是后进先出 (LIFO, Last In First Out) 的线性表
- 栈只允许在线性表的一端 (称为栈顶，top) 进行添加元素和删除元素的操作，在具体应用中常用数组来实现
- 堆栈的两种基本操作：推入 (push) 和弹出 (pop)：
  - 推入 (push)：将数据放入堆栈的顶端，栈顶指针加一。
  - 弹出 (pop)：将顶端数据资料输出 (回传)，栈顶指针减一。

队列 (Queue)

- **队列**是先进先出 (FIFO, First In First Out) 的线性表
- 队列只允许在后端 (rear) 进行插入操作，在前端 (front) 进行删除操作，在具体应用中通常用链表或者数组来实现
- 队列的两种基本操作：入队 (enqueue) 和出队 (dequeue)
- **优先队列** (Priority Queue)是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素

3.3 二叉树

二叉树

- **二叉树**是一组结点的集合，该集合是空集或满足下列条件的非空集合：
  - 有且仅有一个根结点
  - 其余结点分为两个不相交的子集，每个子集都是一棵二叉树，分别称为根结点的左子树 (left subtree) 和右子树 (right subtree)
- **结点的深度** (depth)：根结点的深度为 0，其他结点的深度等于其父结点深度 +1
- **二叉树的高度** (height)：叶结点的最大深度
- **结点的度** (degree)：结点的子树个数 (≤ 2)；0 度结点称为叶结点 (leaf)
- **二叉树的性质**：
  - 任一二叉树至多包含 2^d 个深度为 d 的结点
  - 高度为 h 的二叉树至多有 $2^{h+1}-1$ 个结点
  - 包含 n 个结点的二叉树其高度至少为 $\lceil \lg(n+1) \rceil - 1$
  - 若非空二叉树有 n_0 个叶结点和 n_2 个度为 2 的结点，则 $n_0 = n_2 + 1$
- 注意：二叉树和树是两个不同的概念，尽管二者有许多相同的性质

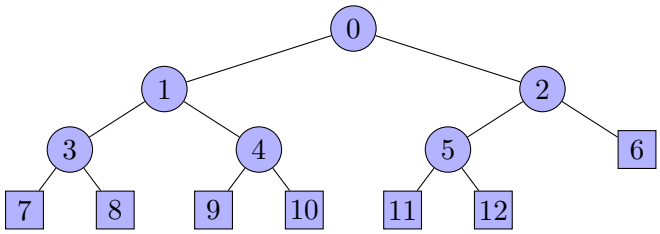
完全二叉树

- **满二叉树** (full/proper binary tree)：每个结点都恰有 0 个或 2 个子结点 (除叶结点外每个结点都有两个子结点)
- **完美二叉树** (perfect binary tree)：高度为 h 且有 $2^{h+1}-1$ 个结点的满二叉树
- **完全二叉树** (complete binary tree)：若除最后一层外其余层都是满的，并且最后一层或者是满的，或者是在右边缺少连续若干结点，则此二叉树为完全二叉树

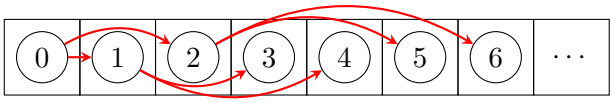
- 高度为 h 的完全二叉树至少有 2^h 个结点，至多有 $2^{h+1}-1$ 个结点
- 完全二叉树可以按深度 (层次) 顺序存储在数组中，结点 k 的父结点是 $\lfloor (k-1)/2 \rfloor$，左子结点是 $2k+1$，右子结点是 $2k+2$
- 有 n 个结点的完全二叉树其叶结点个数为 $\lceil n/2 \rceil$

例子

Example 3.2 (完全二叉树).



完全二叉树的顺序存储方式：



二叉堆 (Binary Heap)

- **堆** (Heap)：基于树的数据结构，其满足堆特性：若 B 是 A 的子结点，则 $\text{key}(A) \geq \text{key}(B)$ (大根堆)
- **二叉堆**：用完全二叉树来表达的堆结构，可以说是实现优先队列的最有效的数据结构之一
- **二叉堆的基本操作**：
  - 往堆中添加元素 (heapify-up, up-heap, bubble-up)：
    1. 将新元素插入堆的尾部
    2. 将新元素与其父结点比较，若满足堆特性，则结束
    3. 否则，将其与父结点交换位置，并重复上一步骤
  - 提取并删除堆首元素 (heapify-down, down-heap, bubble-down)：
    1. 将堆尾元素放到堆首代替已删除的堆首元素
    2. 将其与子结点比较，若满足堆特性，则结束
    3. 否则，将其与子结点中的一个交换位置，使三者满足堆特性，并重复上一步骤
- ✠ **效率**：入队和出队的复杂度 $W(n) \in O(\log n)$

3.4 抽象数据类型

抽象数据类型 (Abstract Data Type)

- 基本数据结构经常采用**抽象数据类型**来表达
- **抽象数据类型**的定义包括：
  - **结构** (Structures)：数据结构的声明
  - **函数** (Functions)：数据操作方法的定义
- 抽象数据类型通常用类 (Class) 来实现 (C++，Java，...)
- 算法的设计通常建立在对抽象数据类型的操作上

4 算法分析基础

4.1 分析的目的

算法分析的目的

- 分析算法的目的：
  - 提高算法性能 (更快的速度和更少的额外存储空间占用)；
  - 从几个可能的算法中挑出最好的算法；
  - 设计一个新的算法
- 在分析一个算法时，通常要考察这个算法的
  - 正确性
  - 效率 (时间复杂度) 和存储空间占用情况 (空间复杂度)
  - 最优性和简洁性

4.2 算法的正确性

算法正确性证明

- 一个完整的算法包含一系列步骤 (操作、指令、语句)，将输入 (前件，precondition) 转换成输出 (后件/效果，postcondition)
- 证明的过程：如果前件被满足，则当算法所有步骤按序执行完毕，后件必为真
- 证明工具：反证法、数学归纳法

  Example 4.1. 阶乘的结果总是大于等于 1 的整数 因此一个正确的计算阶乘的算法，对于满足要求的输入，其输出结果必须是大于等于 1 的整数

4.3 算法的复杂度

算法工作量的度量 (效率问题)

- **算法的效率：时间复杂度** (time complexity)
  - 需要一种方法来计算算法解题所需的工作量，以便衡量算法的效率高低 (执行时间的长短)
  - 这一度量方法必须是与所使用的计算机、程序设计语言、程序员等一切实现细节无关
  - 工作量的大小通常依赖于输入数据的多少 (工作量是输入规模的函数)
- **度量单位：基本操作** (basic operation)
  - 针对某一特定问题，确定解决该问题所需的最基本的操作是什么
  - 算法为解决问题所执行的所有操作的总和大致正比于执行基本操作的数量
  - 例如：排序算法中最重要且最基本的操作是比较，因此我们一般用比较次数的多少来衡量一个排序算法效率的高低
- 注意输入数据的某些特性是否会影响算法的行为 (边界条件、输入规模...)

最坏情况时间复杂度 (Worst-case Complexity)

- **最坏情况复杂度定义：**

$$W(n) = \max\{t(I) | I \in \mathbf{D}_n\}$$

  - $\mathbf{D}_n$ 是算法输入规模为 n 的输入的集合 (通常是无限集)
  - I 是 $\mathbf{D}_n$ 中的某一个输入
  - $t(I)$ 表示在输入 I 下算法执行基本操作的数量
  - 即：最坏情况复杂度标志了一个算法在各种输入条件下最长 (阶最大) 的执行时间
- 对于特定问题，最坏情况下的输入 I 依赖于特定算法，即算法不同，最坏情况的输入也可能不同

平均时间复杂度 (Average Complexity)

- **平均复杂度定义：**

$$A(n) = \sum_{I \in \mathbf{D}_n} P(I)t(I)$$

  - $P(I)$ 是输入 I 出现的概率
  - $P(I)$ 一般难以用解析的方法得到

- 如果考虑算法执行失败的情况：

$$A(n) = P(succ)A_{succ}(n) + P(fail)A_{fail}(n)$$

- $\mathbf{D}_n$ 中的元素 I 可以是以同样方式影响算法行为的一个集合或等价类

Dn通常是无限集，因此通常将之分为很多类

例如：将 $\mathbf{D}_n$ 分为 k 类：$\mathbf{S}_1, \mathbf{S}_2, \cdots, \mathbf{S}_k$，每一类输入算法执行时间相同为 $t(I_{S_i})$，则平均复杂度：

$$\begin{aligned} A(n) &= \sum_{I \in \mathbf{D}_n} P(I)t(I) \\ &= \sum_{I \in \mathbf{S}_1} P(I)t(I) + \sum_{I \in \mathbf{S}_2} P(I)t(I) + \cdots + \sum_{I \in \mathbf{S}_k} P(I)t(I) \\ &= t(I_{S_1}) \sum_{I \in \mathbf{S}_1} P(I) + t(I_{S_2}) \sum_{I \in \mathbf{S}_2} P(I) + \cdots + t(I_{S_k}) \sum_{I \in \mathbf{S}_k} P(I) \\ &= \sum_{i=1}^{k} t(I_{S_i})P(I_{S_i}) \end{aligned}$$

输入落在每一类中的概率与之对应的时间乘积

一个例子

Example 4.2 (在一个无序数组中搜索给定键值).

Algorithm SeqSearch$(E[], n, K)$

```
1 for i ← 0 to n − 1 do
2     if E[i] = K then
3         return i;
4     end
5 end
6 return −1;
```

最坏情况复杂度：$W(n) = n \quad W(n) \in \Theta(n)$ 阶，跟n一个阶

平均复杂度分析 (例)

- $A(n) = P(succ)A_{succ}(n) + P(fail)A_{fail}(n)$
- 在 $\mathbf{D}_n$ 中共有 $n+1$ 类输入情况 依据执行时间一样来分类
  - 搜索成功情况 (K 在数组中)：有 n 类 (I_i : $K = E[i]$)
  - 搜索失败情况 (K 不在数组中)：1 类

实际上应该写成A(n)=P(I|succ)Asuss(n)
+P(I|fail)Afail(n).
这是一个条件概率，即总的复杂度是在：
假设是成功的情况与成功的复杂度乘积+假设是失败的情况下与失败的复杂度的乘积

- 假定 K 等于数组 n 个元素中任一元素的概率相同，即有 $P(I_i|succ) = 1/n$
- $t(I_i) = i + 1,\ i = 0, 1, \cdots, n-1$
- $A_{succ} = \sum_{i=0}^{n-1} P(I_i)t(I_i) = \sum_{i=0}^{n-1}(1/n)(i+1) = (n+1)/2$
- $P(I|fail) = 1,\ t(I) = n,\ \therefore A_{fail}(n) = P(I|fail)t(I) = n$
- 假定 K 在数组中出现的概率为 q，则有

$$A(n) = q\frac{n+1}{2} + (1-q)n = n(1 - \frac{1}{2}q) + \frac{1}{2}q$$

空间复杂度

- 如果可以确定输入数据的存储单元，则可分析最坏情况下或平均情况下算法运行所需占用的存储空间
- 时间与空间的折衷：
  - 存储空间有限 $\Rightarrow$ 算法复杂 $\Rightarrow$ 时间复杂度高
  - 提高算法效率 (降低时间复杂度) $\Rightarrow$ 额外的处理或预处理步骤 $\Rightarrow$ 更多的存储空间

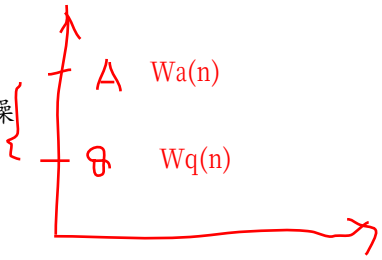
Exercise (1). 已知一个长度为 n 的数组和一个正整数 k，并且最多只能使用一个用于交换数组元素的附加空间单元，试设计算法得到原数组循环右移 k 次的结果并分析算法的时间复杂度。 *deadline: 2019.11.24*

每次移动一格，最后一个放第一格。
如果是最后一个先放附加格中，则需要的复杂度阶为\Theta(kn)；
如果有无限的空间，则需要O(n)

4.4 算法的最优性

算法的最优化

- **问题的复杂度：**
  - 问题也存在复杂度的概念，每一类问题都有一个固有的时间复杂度，即解决该类问题所需的最少工作量，无论使用何种算法都不可能少于这个工作量 下界/限
- **问题复杂度分析：**
  - 选择用于解决该问题的一类算法
  - 基于对该类算法复杂度的分析建立并证明解决该问题所需的基本操作数量的下界 (lower bound)
  - 该下界即为该问题的复杂度
- 问题复杂度的分析一般只针对最坏情况



证明算法的最优性

证明某个算法 A 是否是解决这类问题的最优算法

- 分析算法 A 的最坏情况复杂度 $W_A(n)$
- 证明对于任意解决同类问题的算法，在所有规模为 n 的输入中，存在某些输入情况使这些算法的时间复杂度至少为 $W_{[A]}(n)$
- 如果 $W_A(n) = W_{[A]}(n)$，则证明算法 A 是最优的，同时问题复杂度为 $W_{[A]}(n)$，否则：
  - 还存在比 A 更优的算法；
  - 或者存在比 $W_{[A]}(n)$ 更准确的下界

例子

Example 4.3 (查找一个无序数组中的最大元素).

Algorithm `FindMax`$(E[], n)$

```
1 max ← E[0];
2 for i ← 1 to n − 1 do
3 |   if max < E[i] then
4 |   |   max ← E[i];
5 |   end
6 end
7 return max;
```

把比较(而不是赋值)作为基本操作，因为比较对每个元素都进行，赋值未必；复杂度找最坏情况

复杂度分析

- 基本操作
  - 数组元素之间的比较
- 最坏情况分析
  - 对于元素数为 n 的数组，需要 $n-1$ 次比较操作
  - $W_A(n) = n - 1$
- 问题的复杂度
  - 假定数组元素各异 (符合最坏情况的前提)
  - 在包含 n 个各不相同元素的数组中，$n-1$ 个元素必然不是最大的
  - 对每一个元素来说，要证明其不是最大元素，必须存在另一个元素比它大，即至少需要一次比较操作
  - 要确定 $n-1$ 个元素不是最大元素，至少需要 $n-1$ 次比较
  - $\therefore W_{[A]}(n) = n - 1$ 至少需要n-1次操作,
- $W_A(n) = W_{[A]}(n) \Longrightarrow$ 上述算法是最优的 反证法

查找 (搜索) 算法

- 回顾：顺序查找算法
- 对于无序数组，$W_{[\texttt{SeqSearch}]}(n) = n$，即算法 `SeqSearch` 是最优的
- 对于有序数组，`SeqSearch` 是否还是最优的呢？
  - `SeqSearch` 未利用数组的有序性
  - 是否有比 `SeqSearch` 更优的算法？
- 对有序数组查找算法的一种改进策略：
  - 将 K 与第 ik 处的元素比较，$i = 1, 2, \cdots, n/k$

  - 若结果为小于，则往回比较第 $ik-1, ik-2, \cdots, ik-k+1$ 处的元素
  - 若结果为大于，则往前跳过 $k-1$ 个元素，比较第 $(i+1)k$ 处的元素
  - 则在最坏情况下要比较 $n/k+k$ 次，即 $W(n)=n/k+k$
  - 当 $k=\sqrt{n}$ 时，$W(n)$ 取得最小值 $2\sqrt{n}$
  - 我们将复杂度从 $O(n)$ 降到了 $O(\sqrt{n})$
- 还有没有更优的算法？

二分查找 (Binary Search) 算法

假定数组元素按不减 (nondecreasing) 序排列

Algorithm `BinarySearch`$(E[], n, K)$

```
1 first ← 0;
2 last ← n − 1;
3 while first ≤ last do
4 |   mid ← (first + last)/2;
5 |   if K = E[mid] then return mid;
6 |   else if K < E[mid] then last ← mid − 1;
7 |   else first ← mid + 1;
8 end
9 return −1;
```

`BinarySearch` 最坏情况复杂度分析

- **基本操作**：键值 K 与数组元素的比较
- 算法在每一次比较处产生三个分支
- 在最坏情况下每次比较都不相等，而要查找的部分数组则被一分为二，根据比较结果选择前一半或后一半作为下面要继续查找的部分
- 在待查找部分仅剩一个元素之前需要经过多少次对半分？
  - $n/(2^d) \geq 1 \Rightarrow d \leq \lg n$
- $W(n) = \lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil \in \Theta(\lg n)$ 最坏情况

`BinarySearch` 平均复杂度分析

- 按 K 在数组中出现的位置将所有输入分为 $n+1$ 种情况，其中有 1 种为查找失败情况
- 不妨设 $n = 2^d - 1$，则 $A_{fail}(n) = \lg(n+1)$

  为了刚好可以对半分

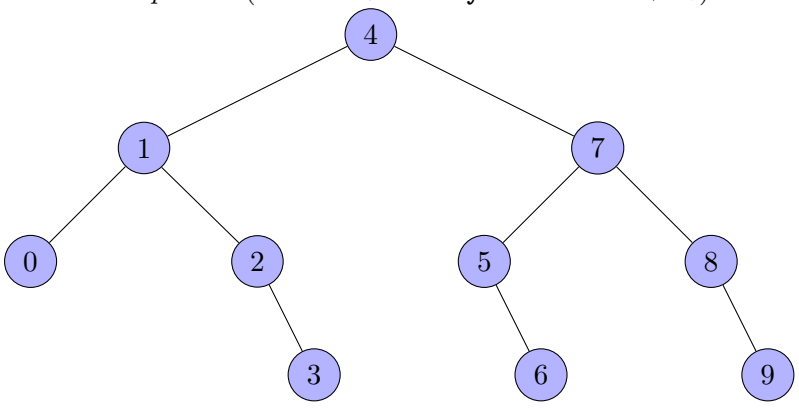
- 设在查找成功情况下 K 在数组各元素位置出现的概率为 $P(I_i|succ) = 1/n$
  - 将数组的 n 个元素位置分组：$\mathbf{S}_t$ 表示恰比较 t 次可知与 K 相等的元素位置集合
  - 则显然有：$|\mathbf{S}_1| = 1 = 2^0$, $|\mathbf{S}_2| = 2^1$, $|\mathbf{S}_3| = 2^2$, $\cdots$, $|\mathbf{S}_t| = 2^{t-1}$
  - 所以，$A_{succ}(n) = \sum_{t=1}^{d}(|\mathbf{S}_t|/n)t = ((d-1)2^d + 1)/n = \lg(n+1) - 1 + \lg(n+1)/n$ 后一项可以省略，所以约为lg(n+1)
- $A(n) = qA_{succ}(n) + (1-q)A_{fail}(n) = \lg(n+1) - q + \dfrac{q\lg(n+1)}{n}$

最优性

- **查找算法类**：在大小为 n 的数组 E 中查找与给定键值 K 相等的元素，返回其位置；对数组元素除比较外无其他操作
- **决策树** (decision tree)：对查找算法类中的某个输入规模为 n 的算法，其决策树是包含标有 $0 \sim n-1$ 结点的二叉树，其构造规则如下
  1. 树根标记第一个与键值 K 比较的元素位置
  2. 对标记为 i 的结点，其左子结点标记为当 $K < E[i]$ 时下一个与 K 比较的元素位置，其右子结点标记为当 $K > E[i]$ 时下一个与 K 比较的元素位置
  3. 一个结点若无左子结点或右子结点，则表示在得到 $K < E[i]$ 或 $K > E[i]$ 的结果后算法终止
- 决策树中从根开始到叶子结点的每条路径代表了其对应算法在给定输入下的比较操作序列
- 对于给定算法，最坏情况下的比较操作序列对应于其决策树中从根到叶子结点的最长路径
- 显然，任何合理的查找算法都可以根据以上规则构造决策树；而其比较次数不会多于其决策树的层数

决策树实例

Example 4.4 ($n = 10$ 时 `BinarySearch` 的决策树).



查找算法类的复杂度

- 查找算法最坏情况下的比较操作序列对应于其决策树中从根到叶子结点的最长路径，设该路径经过的结点数 (即最坏情况下比较次数) 为 p，而决策树共包含 N 个结点，则有：即层数为P
  - $N \leq 1+2+4+\cdots+2^{p-1}=2^p-1 \Longrightarrow 2^p \geq N+1$
- 如何得到 p 与输入规模 n 之间的关系？$N \overset{?}{\geq} n$
- 即需要证明：对从 0 到 $n-1$ 的每一个 i，决策树中至少有一个结点标记为 i
- $2^p \geq N+1 \geq n+1 \Longrightarrow p \geq \lg(n+1)$

Theorem 4.1. 任何使用比较的方法在包含 n 个元素的数组中查找键值为 K 的元素的算法至少要做 $\lceil \lg(n+1) \rceil$ 次比较

证明 $N \geq n$

反证法证明.

- 假设从 0 到 $n-1$ 中存在某一位置 i 在决策树中没有对应的结点标注
- 构造两个输入数组 E_1 和 E_2 如下：
  - $E_1[i]=K,\ E_2[i]=K'>K$
  - $\forall j<i,\ E_1[j]=E_2[j]<K$
  - $\forall j>i,\ E_1[j]=E_2[j]>K'$
- 因为决策树中不存在 i 结点，算法将永远不会比较 K 与 $E_1[i]$ 或 $E_2[i]$，这样，对于 E_1 和 E_2 这两种不同的输入，同一算法处理的元素完全相同，因而其运行过程和结果都是一样的
- 所以，E_1 和 E_2 中至少有一个输入，算法给出的结果是错误的，与算法的正确性构成矛盾

□

进一步思考

- 若只需要判定某一给定键值是否在数组中，而不需要给出其在数组中出现的具体位置，则是否存在更快的算法？
  - 数组无序；
  - 数组有序；
  - 数组规模非常大；
  - 使用更复杂的数据结构；
  - 空间复杂度。

即查看元素是否在集合中；
c++: std::set红黑树

- 如果允许出现少量错判，则情况又如何？ bloom filter

算法概论

第二讲：递归和分治法

薛健

Last Modified: 2018.12.9

主要内容

1 递归和数学归纳法

1.1 递归

递归 (Recursion)

- **John McCarthy** (人工智能之父)：最早认识到递归对于计算机程序设计语言的重要性，建议在 *Algol60* (Pascal、PL/I 和 C 的前身) 中增加递归特性，并自己设计了一种引入递归数据结构的语言：*Lisp*。现在，几乎所有主流的计算机程序设计语言均支持递归
- 在数学和计算机科学中，递归指在函数的定义中使用函数自身的方法，也常用于描述由一种 (或多种) 简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况 (用自相似的方法描述事物的过程)
- 例如，关于某人祖先的定义：
  - 某人的双亲是他的祖先 (基本情况)
  - 某人祖先的双亲同样是某人的祖先 (递归步骤)
- 像这样的定义在数学中十分常见。例如，集合论对自然数的正式定义是：1 是一个自然数，每个自然数都有一个后继，这一个后继也是自然数

递归三个要点：1，函数调用函数本身
2，有结束条件（终止条件）
3，问题规模逐渐减小

例子

Example 1.1. 斐波那契数列 (Fibonacci Sequence)

- $F_0 = 0$
- $F_1 = 1$

- $F_n = F_{n-1} + F_{n-2}$

Algorithm `Fib`(n)

```
1 if n < 2 then f ← n;
2 else
3 |   f1 ← Fib(n − 1);
4 |   f2 ← Fib(n − 2);
5 |   f ← f1 + f2;
6 end
7 return f;
```

用递归来解决问题

- **递归过程的建立：**
  1. 我们已经完成了吗？如果完成了，返回结果 (如果没有这样的终止条件，递归将会永远地继续下去)
  2. 如果没有，则简化问题，解决较容易的问题，并将结果组装成原始问题的解决办法；然后返回该解决办法
- Method 99：一种更便于理解的构造方式
  - 假设要解决的问题规模为 100 (fantasy precondition)
  - 假设已经有一个名为 p99 的子程序能够解决规模为 0∼99 的问题
  - 划分不需递归的部分 (base case)
  - 构造解决问题的程序 p，写出任何不需要 p99 就可以解决的情况的代码，其他情况下在需要的时候调用 p99 (将规模太大不能直接解决的情况分解为规模在 0∼99 之间的子问题，分别用 p99 解决)

例子

Example 1.2 (二分查找的递归版本).

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1 if last < first then return −1;
2 else
3 |   mid ← (first + last)/2;
4 |   if K = E[mid] then return mid;
5 |   else if K < E[mid] then
6 |   |   return BinarySearch99[→ BinarySearchRec](E, first, mid − 1, K);
7 |   else return BinarySearch99[→ BinarySearchRec](E, mid + 1, last, K);
8 end
9 return −1;
```

课后习题

Exercise (2). 定义文件 `xx.tar.gz` 的产生方式如下：

- 以 `xx` 为文件名的文件通过 tar 和 gzip 打包压缩产生，该文件中以字符串的方式记录了一个非负整数；
- 或者以 `xx` 为名的目录通过 tar 和 gzip 打包压缩产生，该目录中包含若干 `xx.tar.gz`。

其中，`x` ∈ `[0, 9]`。现给定一个根据上述定义生成的文件 `00.tar.gz` (该文件从课程网站下载)，请确定其中包含的以 `xx` 为文件名的文件个数以及这些文件中所记录的非负整数之和。

文本文件 而非二进制

deadline: 2018.12.01

课后习题 (提示)

- 编写递归程序求解
- 可使用任意你所熟悉的编程语言和工具
- 大多数高级语言或脚本语言提供了解 gzip 压缩包的工具

用 Python 解压缩包

```
import os, tarfile
def unpack_path_file(pathname):
    archive = tarfile.open(pathname, 'r:gz')
    for tarinfo in archive:
        archive.extract(tarinfo, os.getcwd())
    archive.close()
```

1.2 数学归纳法 用于证明递归

数学归纳法

- **什么是证明?** (回顾): 在一个特定的公理系统中，由公理 (axioms) (和定理 (theorems)) 推导出某些命题 (proposition) 的过程
  - 证明方法: 演绎推理，构造证明，反证法，数学归纳法，...
- **数学归纳法:** 用于证明某个给定命题对于一个无限集内的所有对象成立的证明方法
- 最常见的归纳在自然数集上进行，但归纳法同样适用于更一般的无限集:
  - 给定集合是偏序集 (partially ordered set)，即集合中的部分元素对之间定义了偏序关系 (满足自反性、反对称性和传递性的二元关系)
  - 给定集合中不存在无限递减的链 偏序关系：如小于等于 如自然数集，存在最小的自然数

在数学中，特别是序理论中，偏序集合 (简写为 poset) 是配备了偏序关系的集合。这个关系形式化了排序、顺序或排列这个集合的元素的直觉概念。这种排序不必然需要是全部的，就是说不需要但也可以保证在这个集合内的所有对象的相互可比较性。

非严格偏序 (自反偏序) 给定集合 $\mathbf{S}$，“$\leq$”是 $\mathbf{S}$ 上的二元关系，若“$\leq$”满足:

1. 自反性: $\forall x \in \mathbf{S}$ 有 $x \leq x$; 小于：反自反性
2. 反对称性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 且 $y \leq x$，则 $x = y$;
3. 传递性: $\forall x, y, z \in \mathbf{S}$，$x \leq y$ 且 $y \leq z$，则 $x \leq z$

则称“$\leq$”是 $\mathbf{S}$ 上的非严格偏序或自反偏序。

严格偏序 (反自反偏序) 给定集合 $\mathbf{S}$，“$<$”是 $\mathbf{S}$ 上的二元关系，若“$<$”满足:

1. 反自反性: $\forall x \in \mathbf{S}$ 有 $x \not< x$;
2. 非对称性: $\forall x, y \in \mathbf{S}$，$x < y \Rightarrow y \not< x$;
3. 传递性: $\forall x, y, z \in \mathbf{S}$，$x < y$ 且 $y < z$，则 $x < z$

则称“$<$”是 $\mathbf{S}$ 上的严格偏序或反自反偏序。

全序关系 给定集合 $\mathbf{S}$，“$\leq$”是 $\mathbf{S}$ 上的二元关系，若“$\leq$”满足:

1. 反对称性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 且 $y \leq x$，则 $x = y$;
2. 传递性: $\forall x, y, z \in \mathbf{S}$，$x \leq y$ 且 $y \leq z$，则 $x \leq z$
3. 完全性: $\forall x, y \in \mathbf{S}$，$x \leq y$ 或 $y \leq x$

数学归纳法

- **证明模式**：要证明关于 n 的语句 $F(n)$，数学归纳法可分两步：*

  1. **奠基**：当 $n=0$ 时待证语句为真，即须证 $F(0)$ 真
  2. **归纳**：在**一定的假设**下，证明情形 $n+1$ 时待证语句为真，即证明 $F(n+1)$ 真

- 完成上述两步，即可得出结论“依数学归纳法，待证语句得证”

- “**一定的假设**” —— **归纳假设**：
  - **简单归纳假设**：在假设“$F(n)$ 真”之下，去证明 $F(n+1)$ 真 $\Rightarrow$ **简单归纳证明**
  - **强归纳假设**：在假设“$F(0)$ 真，$F(1)$ 真，$\cdots$，$F(n)$ 真”之下去证明 $F(n+1)$ 真 $\Rightarrow$ **强归纳证明** ★
  - **参变归纳假设**：待证语句含有参数，如 $F(n,u)$，则奠基是：$F(0,u)$ 对一切 u 真；在归纳步骤中，假设“$F(n,u)$ 对一切 u 真”，从而证明 $F(n+1,u)$ 亦真 $\Rightarrow$ **参变归纳证明**

数学归纳法与递归

> “几乎可以说：每有一种递归式，便有一种数学归纳法；反之，每有一种数学归纳法，便有一种递归式。$\cdots$ 例如：简单归纳法对应于原始递归式，强归纳法对应于串值原始递归式，参变归纳法对应于参数变异递归式。”
>
> ——莫绍揆：《递归函数论》

- 数学归纳法与递归有着非常紧密的联系
  - 数学归纳法证明可以看作是一种递归证明
  - 数学归纳法使得递归程序正确性证明得到极大的简化

递归程序的数学归纳法证明 — 2-tree 外路径长度

Definition 1.1 (2-tree (full binary tree))**.** 2-tree 是这样一种二叉树，它是一棵空树或仅包含下列两种结点： 满二叉树

1. **外结点** (external node)：没有子树 (度为 0) 的结点，即叶结点
2. **内结点** (internal node)：恰有两棵子树的结点

Definition 1.2 (外路径长度 (external path length))**.** 一棵 2-tree T 的**外路径长度**是从根结点到所有外结点路径长度之和 (路径长度等于路径经过的边的数目)。其等价的递归定义：

1. 以外结点为根的 2-tree (单结点 2-tree) 其外路径长度为 0
2. 以内结点为根的 2-tree 外路径长度等于其左子树 L 的外路径长度加 L 的外结点个数加其右子树 R 的外路径长度加 R 的外结点个数

*莫绍揆：《递归函数论》

计算外路径长度

初始条件

Algorithm `CalculateEPL`(T)

Input: A 2-tree T
Output: A pair $(epl, extNum)$

```
1 if T is a leaf then return (0,1) ;   外路径长度和外节点个数
2 else
3     (eplL, extNumL) ← CalculateEPL(T.left);
4     (eplR, extNumR) ← CalculateEPL(T.right);   递归调用
5     epl ← eplL + eplR + extNumL + extNumR;
6     extNum ← extNumL + extNumR;
7     return (epl, extNum);
8 end
```

外路径长度引理

Lemma 1.3. *T 是一棵 2-tree,e 和 m 分别对应于 `CalculateEPL` 返回值中的 epl 和 $extNum$, 则有:*

1. *e 等于 T 的外路径长度*
2. *m 等于 T 的外结点个数*
3. *$e \geq m \lg m$*

Proof. 对 T 用数学归纳法。
奠基: T 是叶结点，根据 `CalculateEPL` 第 1 行，$e = 0, m = 1$ 满足 1 和 2，$0 \geq 1 \lg 1 = 0$ 满足 3。
归纳: 对于非叶结点 T，假设 T 的所有子树 S 引理成立，令 L 和 R 分别为 T 的左、右子树，则根据归纳假设，对于 L 和 R 引理也成立。由 `CalculateEPL` 第 3 到 7 行有:

$$\begin{aligned} e &= e_L + e_R + m_L + m_R \\ m &= m_L + m_R \end{aligned}$$

由外路径长度的递归定义可知 e 为 T 的外路径长度；又因为 T 的外结点要么在 L 中，要么在 R 中，所以 m 等于 T 的外结点个数；另外，根据归纳假设，有:

$$e \geq m_L \lg m_L + m_R \lg m_R + m$$

注意到 $x \lg x$ 是凸函数，则根据凸函数性质有:

$$m_L \lg m_L + m_R \lg m_R \geq 2\left(\frac{m_L + m_R}{2}\right) \lg \left(\frac{m_L + m_R}{2}\right)$$

所以得: $e \geq m(\lg m - 1) + m = m \lg m$ □

Corollary 1.4. *内结点数为 n 的 2-tree 外路径长度 $e \geq (n+1)\lg(n+1)$*

1.3 程序正确性证明

内节点个数和外节点个数-1

定义和术语

Definition 1.5.

程序块 (block) 一段有且仅有一个入口和一个出口的程序代码

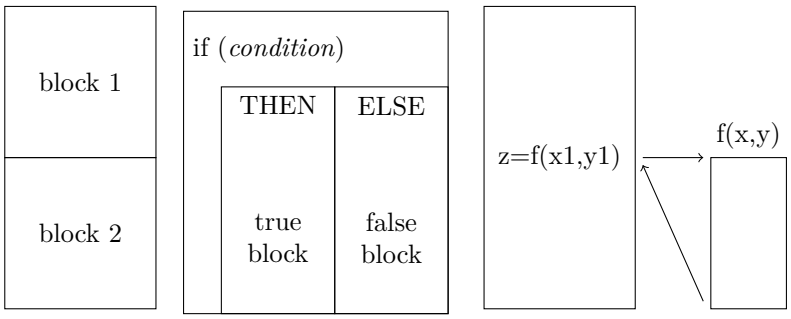
过程 (procedure) 赋予名称且可被调用的程序块，一般包含输入和输出参数

函数 (function) 有输出参数的过程

前件 (precondition) 用于描述一个程序块的输入参数和非局部数据且在进入程序块时为真的逻辑语句 相当于前提条件 输入条件

后件 (postcondition) 用于描述一个程序块的输入参数、输出参数及非局部数据且在退出程序块时为真的逻辑语句 结果

基本控制结构



sequence 顺序条件　　*alternation* 条件语句　　*procedure call* 过程调用(包括递归)

循环结构 (**for**, **while**, ...) $\Longrightarrow$ 递归！ 循环总可以转换为递归

基本证明形式

Proposition 1.6 (General correctness lemma form)**.** 如果在进入程序块时前件成立，则在退出程序块时后件也成立。

Proposition 1.7 (Sequence correctness lemma form)**.**

1. 整个程序块的前件 $\Rightarrow$ 程序块 *1* 的前件
2. 程序块 *1* 的后件 $\Rightarrow$ 程序块 *2* 的前件
3. 程序块 *2* 的后件 $\Rightarrow$ 整个程序块的后件

Proposition 1.8 (Procedure-call correctness lemma form)**.**

1. 程序块的前件 $\Rightarrow$ 被调用过程的前件及其实际调用参数
2. 被调用过程的后件及其实际调用参数 $\Rightarrow$ 程序块的后件

Proposition 1.9 (Alternation correctness lemma form)**.**

1. 程序块的前件以及选择分支条件为真 $\Rightarrow$ 条件为真时的分支程序块前件
2. 条件为真时的分支程序块后件以及选择分支条件为真 $\Rightarrow$ 程序块的后件
3. 程序块的前件以及选择分支条件为假 $\Rightarrow$ 条件为假时的分支程序块前件
4. 条件为假时的分支程序块后件以及选择分支条件为假 $\Rightarrow$ 程序块的后件

单赋值形式

- 证明的困难：到逻辑表达式的转换 当转换到逻辑表达式之后可以有很多方法证明
  - 无条件跳转 (goto) 语句：可以消除 goto会影响完整结构，因为可以goto任何地方
  - 赋值 (assignment) 语句："y=y+1"，"x=y+1; y=z;"，... 赋值语句不满足逻辑表达式
- **单赋值形式**：Single-Assignment Paradigm 或 Static Single Assignment Form (SSA)，如果在某个过程内赋值的每一个变量作为赋值目标只出现一次，称这个过程是 (静态) 单赋值形式。
- 在编译器设计中，单赋值形式是一种中间表示 (IR)，它能有效地将程序中的运算值和它们的存储位置分开，从而使得若干优化能具有更有效的形式
- 使用单赋值形式的语言 (编译器)：Prolog, ML, Haskell, SISAL (Streams and Iteration in a Single Assignment Language), SAC (Single Assignment C)

循环变为递归：因为单赋值形式在循环中不满足（i++, i+=1等），所以办法是将循环变为递归，递归内只有单赋值

例子

a code fragment

```
1 if y < 0 then
2 |   y = 0;      逻辑表达式矛盾
3 end
4 x = 2 * y;
```

$\Rightarrow$

single assignment version

```
1 if y < 0 then
2 |   y1 = 0;
3 else
4 |   y1 = y;
5 end
6 x = 2 * y1;
```

$$(y < 0 \Rightarrow y1 = 0) \wedge (y \geq 0 \Rightarrow y1 = y) \wedge (x = 2 * y1)$$ 逻辑表达式

无循环过程

Example 1.3 (顺序查找的递归版本).

Algorithm `SeqSearchRec`$(E[], m, num, K)$

```
1 if m ≥ num then
2 |   ans ← −1;
3 else if E[m] = K then
4 |   ans ← m;
5 else
6 |   ans ← SeqSearchRec(E, m + 1, num, K);
7 end
8 return ans;
```

循环转换成递归

准备工作：

1. 将循环内部的局部变量转换为单赋值形式
2. 对所有在循环内需要更新的变量 (通常是定义在循环外的)，将所有更新工作放到循环体末尾进行

转换步骤：

1. 循环体内更新的变量变为递归过程的输入参数，其初始值对应于顶层调用递归过程时实际传入的参数值 (active parameters)
2. 在循环外定义但在循环内只访问不更新的变量也变为递归过程的输入参数，在递归调用过程中其值从不改变 (passive parameters)
3. 递归过程起始处检测循环条件，若为假则返回结果 (退出)，若为真则执行循环体内容；循环体中的 break 语句也对应于递归过程的返回
4. 当转换至循环体结束时，调用递归过程本身，其实际调用参数即为循环内更新后的变量值

例子

Example 1.4 (阶乘函数).

Algorithm FactLoop(n)

```
1 k ← 1;
2 f ← 1;
3 while k ≤ n do
4 |   fnew ← f * k;
5 |   knew ← k + 1;
6 |   k ← knew;
7 |   f ← fnew;
8 end
9 return f;
```

转换结果

Example 1.5 (阶乘函数的递归版本).

Algorithm Fact(n)

```
1 return FactRec(n, 1, 1);
```

Algorithm FactRec(n, k, f)

```
1 if k > n then ans ← f;
2 else
3 |   fnew ← f * k;
4 |   knew ← k + 1;
5 |   ans ← FactRec(n, knew, fnew);
6 end
7 return ans;
```

证明实例

Example 1.6 (二分查找的规范化递归版本).

Algorithm `BinarySearchRec`$(E[], first, last, K)$

```
1 if last < first then index ← −1;
2 else
3 |   mid ← (first + last)/2;
4 |   if K = E[mid] then index ← mid;
5 |   else if K < E[mid] then
6 |   |   index ← BinarySearchRec(E, first, mid − 1, K);
7 |   else index ← BinarySearchRec(E, mid + 1, last, K);
8 end
9 return index;
```

证明

Lemma 1.10. 当 `BinarySearchRec`$(E, first, last, K)$ 被调用，其问题规模为 $last - first + 1 = n$,且 $E[first], \cdots, E[last]$ 以不减序排列,则对任意 $n \geq 0$,如果 K 不在 $E[first], \cdots, E[last]$ 中，过程调用返回 -1，否则返回 $index$ 满足 $K = E[index]$.

Proof. 对问题规模 n 归纳：
奠基：当 $n = 0$ 时，$last = first - 1$，第 1 行条件满足，返回值为 -1，结论成立；
归纳：当 $n > 0$ 时，假设对于 $0 \leq k < n$，`BinarySearchRec`(E, f, l, K) 调用结论成立，其中 $l - f + 1 = k$。则当 $k = n$ 时，因为 $n > 0$，所以第 1 行条件为假，程序执行到第 3 行，这时 $mid = \lfloor (first + last)/2 \rfloor$，所以 $first \leq mid \leq last$。若第 4 行条件为真，则返回 $index = mid$ 为 K 在序列 $E[first], \cdots, E[last]$ 中的位置，结论成立；
若第 4 行条件为假，由上述条件可知

$$(mid - 1) - first + 1 < last - first + 1 = n$$
$$last - (mid + 1) + 1 < last - first + 1 = n$$

因此第 6、7 行的递归调用满足归纳假设，可以返回正确结果。
当第 5 行条件为真时，第 6 行递归调用被执行，若返回非负整数值，则问题解决，结论成立；若返回 -1，则表示 K 不在 $E[first], \cdots, E[mid - 1]$ 中，而第 5 行条件为真表示 K 也不在 $E[mid], \cdots, E[last]$ 中，因此返回 -1 对当前调用也是正确的。
与之类似，可证明第 5 行条件为假时结论也成立。 □

1.4 递推方程

递推方程

- **递推方程** (Recurrence Equation)：一种递推地定义一个序列的方程式：序列的每一项定义为前一项的函数。
- 递推方程可以很自然地用来描述递归程序执行过程的资源 (时间、空间...) 使用情况 (统一称其为“开销 (cost) ”)
- 最坏情况下程序开销递推方程的建立 $T(n) = ?$：
  1. 对于顺序执行的程序块，直接加上它的开销；
  2. 对于有选择分支的程序块 (非递归终止分支，nonbase cases)，加上分支中最大开销；
  3. 对于子程序调用，加上所调用子程序的开销 $T_s(n_s(n))$，其中 $n_s(n)$ 为所调用子程序的输入规模，是主程序输入规模 n 的函数；
  4. 对于递归调用，加上递归调用开销 $T(n_r(n))$，其中 $n_r(n)$ 为递归调用的规模，是主程序输入规模 n 的函数
- 如何求 $T(n)$?

例子

- 递归版的顺序查找：

$$\big(0+(1+\max(0,T(n-1)))\big)+0$$
$$T(n)=T(n-1)+1$$

- 递归版的二分查找：

$$T(n)=T(n/2)+1$$

常用递推方程形式

- Divide and Conquer：规模为 n 的问题被分解为 b 个规模为 n/c $(c>1)$ 的子问题，$f(n)$ 为非递归部分（分解和合并）的开销，$b\geq 1$ 称为分支因子 (branching factor)

$$T(n)=bT(n/c)+f(n)$$

- Chip and Conquer：规模为 n 的问题被“裁剪”成规模为 $n-c$ 的子问题

$$T(n)=T(n-c)+f(n)$$

- Chip and Be Conquered：规模为 n 的问题被分解并“裁剪”成 b 个规模为 $n-c$ 的子问题

$$T(n)=bT(n-c)+f(n)$$

递推方程求解

Example 1.7 $(T(n)=2T(n/2)+n\lg n)$.

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2})+n\lg n = 2(2T(\frac{n}{4})+\frac{n}{2}\lg\frac{n}{2})+n\lg n\\
&= 2^2T(\frac{n}{2^2})+n\lg\frac{n}{2}+n\lg n\\
&= 2^3T(\frac{n}{2^3})+n\lg\frac{n}{2^2}+n\lg\frac{n}{2}+n\lg n\\
&= \cdots\\
&= 2^dT(\frac{n}{2^d})+n(\lg\frac{n}{2^{d-1}}+\cdots+\lg\frac{n}{2^0})\\
&= nT(1)+n(\lg n-(d-1)+\lg n-(d-2)+\cdots+\lg n-0)\\
&= nT(1)+nd\lg n-n\frac{d(d-1)}{2}\\
&= nT(1)+\frac{n}{2}\lg^2 n+\frac{n}{2}\lg n\in\Theta(n\lg^2 n)
\end{aligned}$$

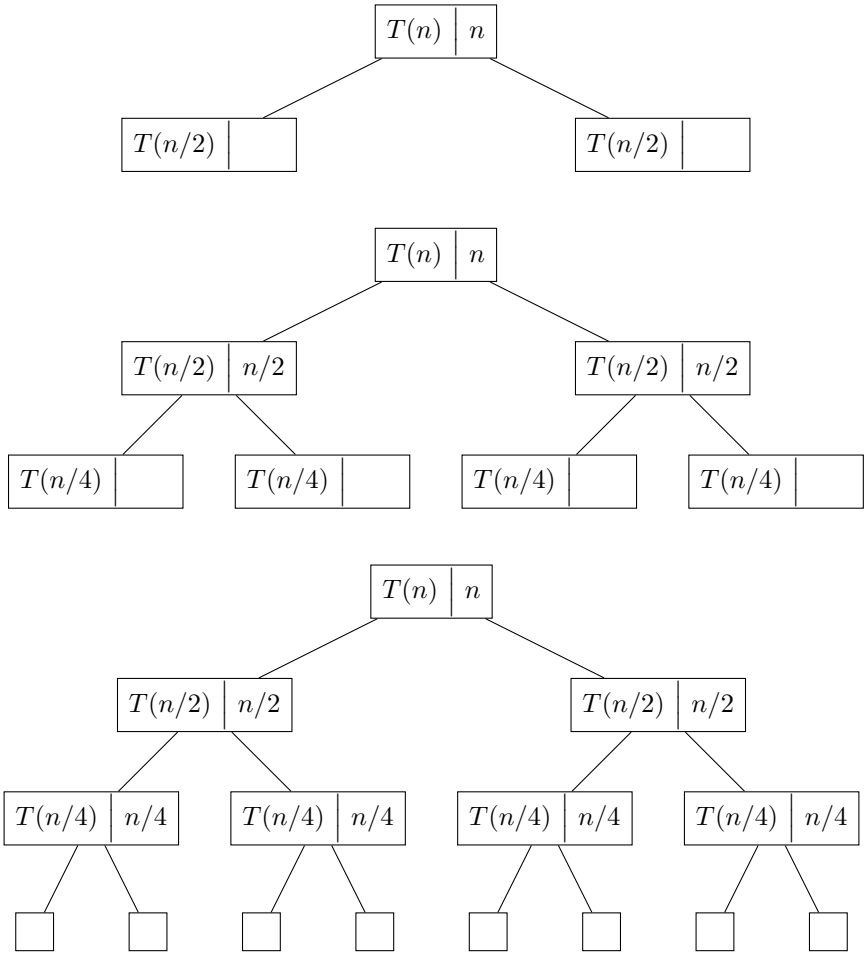
递归树 (Recursion Tree)

Divide and Conquer

$$T(n)=2T(n/2)+n$$

$T(size)$	$nonrec.\ cost$

$T(n)$	



$$n/2^d = 1 \Rightarrow T(n) \approx n \lg n$$

Divid and Conquer 一般情况

$$T(n) = bT(n/c) + f(n)$$

定义 $E = \log_c b = \dfrac{\lg b}{\lg c}$ 称为关键指数 (critical exponent)

Lemma 1.11. *Divid and Conquer* 递归树的叶结点个数 $L \approx n^E$

Lemma 1.12.

1. 递归树的深度 (高度) $D \approx \lg n / \lg c = \log_c n$
2. 第 0 层的开销和为 $f(n)$
3. 假设递归基础的开销为 1，第 D 层的开销和为 n^E
4. $T(n)$ 等于递归树中所有结点的非递归开销总和，也就是递归树每一层“开销和”的总和

主定理 预习

Theorem 1.13 (Little Master Theorem)**.** 对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若递归树每一层的开销和呈递增几何级数，则 $T(n) \in \Theta(n^E)$

2. 若递归树每一层的开销和保持不变，则 $T(n) \in \Theta(f(n)\log n)$

3. 若递归树每一层的开销和呈递减几何级数，则 $T(n) \in \Theta(f(n))$

Theorem 1.14 (Master Theorem)**.** 对于递推方程 $T(n) = bT(n/c) + f(n)$，若 $T(1) \in \Theta(1)$ 及 $E = \log_c b$

1. 若对常数 $\epsilon > 0$，$f(n) \in O(n^{E-\epsilon})$，则 $T(n) \in \Theta(n^E)$

2. 若 $f(n) \in \Theta(n^E)$，则 $T(n) \in \Theta(n^E \log n)$
   更一般地：若 $f(n) \in \Theta(n^E \log^k n)$，则 $T(n) \in \Theta(n^E \log^{k+1} n)$

3. 若对常数 $\epsilon > 0$，$f(n) \in \Omega(n^{E+\epsilon})$，并且存在常数 $\delta < 1$ 以及整数 n_0，使得当 $n > n_0$ 时有 $bf(n/c) \le \delta f(n)$，则 $T(n) \in \Theta(f(n))$

主定理的证明

证明要点. 递归树中在第 d 层 (深度为 d) 有 b^d 个结点，每个非叶结点的非递归开销为 $f(n/c^d)$，因此有

$$T(n) = n^{\log_c b}\Theta(1) + \sum_{d=0}^{\log_c n - 1} b^d f\left(\frac{n}{c^d}\right)$$

令 $g(n) = \sum_{d=0}^{\log_c n - 1} b^d f\left(\frac{n}{c^d}\right)$，则：

1. 在条件 1 下，有 $g(n) \in O(n^E)$

2. 在条件 2 下，有 $g(n) \in \Theta(n^E \log n)$

3. 在条件 3 下，有 $g(n) \in \Theta(f(n))$

□

主定理的应用

Example 1.8 ($T(n) = 9T(n/3) + n$). $\log_c b = \log_3 9 = 2$，取 $\epsilon = 0.5$，则有 $f(n) = n \in O(n^{2-0.5}) = O(2^{1.5})$，
所以，$T(n) \in \Theta(n^2)$

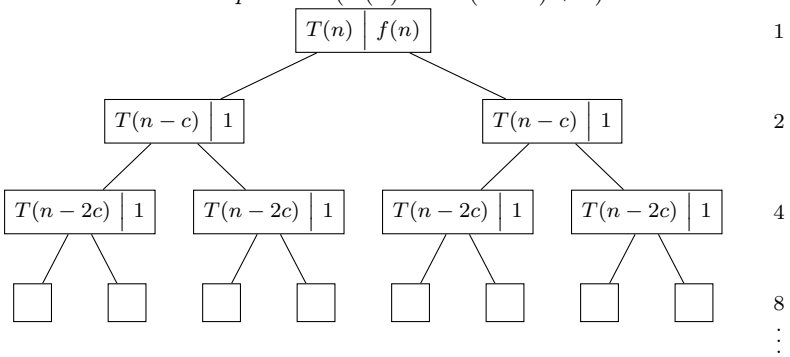
Example 1.9 ($T(n) = T(2n/3) + 1$). $\log_c b = \log_{3/2} 1 = 0$，故 $f(n) = 1 \in \Theta(n^0)$，
所以 $T(n) \in \Theta(n^0 \log n) = \Theta(\log n)$

Example 1.10 ($T(n) = 3T(n/4) + n\lg n$). $\log_c b = \log_4 3 < 1$，故存在 $\epsilon > 0$ 使 $\log_4 3 + \epsilon < 1$，
则 $f(n) = n\lg n \in \Omega(n) \subset \Omega(n^{\log_4 3+\epsilon})$；
另外，$bf(n/c) = 3(n/4)\lg(n/4) = \frac{3n}{4}(\lg n - 2) \le \frac{3}{4}f(n)$，
所以 $T(n) \in \Theta(n\lg n)$

递归树

Chip and Conquer (Be Conquered)

Example 1.11 ($T(n) = 2T(n-c) + 1$).



Chip and Conquer (Be Conquered)

- $T(n) = bT(n-c) + f(n)$
- 递归树深度大约为 $d = n/c$
- $T(n) \doteq \sum_{d=0}^{n/c} b^d f(n-cd) = b^{n/c} \sum_{h=0}^{n/c} \frac{f(ch)}{b^h} = b^{n/c} g(n), \quad h = (n/c) - d$
- 在大多数实际情况下，$g(n) \in \Theta(1)$，这时 $T(n) \in \Theta(b^{n/c})$，是一个指数阶的函数！
- 若 $b = 1$：$T(n) \doteq \sum_{h=0}^{n/c} f(ch) \approx \frac{1}{c} \int_0^n f(x)dx$
  - $f(n) = n^\alpha$：$T(n) \in \Theta(n^{\alpha+1})$
  - $f(n) = \log n$：$T(n) \in \Theta(n \log n)$

2 分治法

2.1 基本原理

分治法 (Divide and Conquer)

- **工作原理**：“解决几个小问题通常比解决一个大问题来得简单”
- **设计思想**：将一个直接难以解决的大问题分成较小规模的数个子问题；分别解决 (治) 这些子问题；将子问题结果合成原问题的结果
  - 由分治法产生的子问题往往是原问题的较小模式，在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解，这自然导致递归过程的产生
- **适用条件**：
  1. 问题的规模缩小到一定的程度就可以容易地解决；
  2. 问题可以分解为若干个规模较小的相同问题；
  3. 利用该问题分解出的子问题的解可以合并为该问题的解；
  4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题 (效率)
- 使用分治法的例子：`BinarySearchRec`

算法基本结构

Algorithm `Solve`(I)

```
1 n ← Size(I);
2 if n ≤ smallSize then
3 |   solution ← DirectlySolve(I);
4 else
5 |   {I_1, ··· , I_k} ← Divide(I);
6 |   foreach i ∈ {1, ··· , k} do  S_i ← Solve(I_i) ;
7 |   solution ← Combine(S_1, ··· , S_k);
8 end
9 return solution
```

时间复杂度：$\begin{cases} T(n) = D(n) + \sum_{i=1}^{k} T(\texttt{Size}(I_i)) + C(n) & n > smallSize \\ T(n) \in \Theta(1) & n \le smallSize \end{cases}$

2.2 矩阵相乘

斯特拉森矩阵乘法

- 由德国数学家 Volker Strassen 于 1969 年提出，被称为 Strassen Algorithm
- A 和 B 是两个 $n \times n$ 矩阵，A 和 B 相加时间复杂度为 $\Theta(n^2)$，A 和 B 相乘时间复杂度为 $\Theta(n^3)$
- 分治法：将矩阵分成大小为 $\frac{n}{2} \times \frac{n}{2}$ 的子矩阵 (假设 $n = 2^k$)

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

- $T(n) = 8T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \le 2) \Rightarrow T(n) \in \Theta(n^3)$
- 在分治法基础上通过增加加法次数来减少乘法次数

$$AB = \begin{bmatrix} P + S - T + V & R + T \\ Q + S & P + R - Q + U \end{bmatrix}$$

其中：

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

- $T(n) = 7T(n/2) + c_1 n^2, \quad T(n) = c_2 \ (n \le 2) \Rightarrow T(n) \in \Theta(n^{2.807})$

2.3 大整数乘法

问题描述

- 在分析算法复杂度时经常将乘法作为基本运算处理，即认为其时间复杂度为 $\Theta(1)$
- 但这样的分析有一个前提，就是计算机字长允许直接表示和处理参与运算的整数

- 在某些情况下需要处理很大的整数，超过了计算机字长表示范围
- 用浮点数运算限制了计算精度和有效数字位数
- 因此，对于大整数相乘，只能以软件的方式设计算法来实现
- **目标**：设计一个有效的算法进行两个 n 位大整数的乘法运算

设计思路

- 采用列竖式笔算的方法，若将每 2 个 1 位数的相乘作为基本运算单位，其时间复杂度为 $O(n^2)$
- 分治法思想：
  - 将 2 个 n 位 b 进制整数各分为两段表示如下：(为便于分析，不妨设 $n = 2^k$)

$$X = Ab^{n/2} + B, \quad Y = Cb^{n/2} + D$$

  - 则 $XY = ACb^n + (AD + BC)b^{n/2} + BD$
  - 递归开销为 4 次 $n/2$ 位整数的乘法，非递归开销为 3 次不超过 n 位整数的加法 $(O(n))$、2 次数组移位 $(O(n))$
  - 时间复杂度：$W(n) = 4W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^2)$
  - 继续改进：$XY = ACb^n + ((A - B)(D - C) + AC + BD)b^{n/2} + BD$
  - 递归开销为 3 次 $n/2$ 位整数的乘法，非递归开销为 6 次加减法和 2 次移位
  - 时间复杂度：$W(n) = 3W(n/2) + O(n),\ W(1) = 1 \Rightarrow W(n) \in \Theta(n^{\lg 3}) \in O(n^{1.59})$

算法描述

- 假定整数保存在大小为 $n+1 (n > 0)$ 的数组 A 中，$A[0]$ 为符号位，取 $+1$ 或 -1，$A[i]$ 对应第 i 位数字

Algorithm `LargeMul`$(X[], Y[], n)$

```
1  S ← Array(2 * n + 1, 0);
2  S[0] ← X[0] * Y[0];
3  if n = 1 then
4  |  S ← Mul(X[1], Y[1]);
5  else
6  |  mid ← n div 2;
7  |  A ← X[mid + 1..n];  B ← X[1..mid];
8  |  C ← Y[mid + 1..n];  D ← Y[1..mid];
9  |  m1 ← LargeMul(A, C, n − mid);
10 |  m2 ← LargeMul(Sub(A, B), Sub(D, C), Max(mid, n − mid));
11 |  S ← LargeMul(B, D, mid);
12 |  S ← Add(S, ShiftLeft(m2, n − mid));
13 |  S ← Add(S, ShiftLeft(m1, 2n − 2mid));
14 end
```

2.4 最接近点对问题

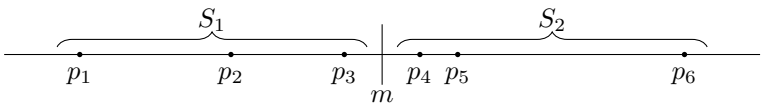
问题描述

- 最小套圈问题：给定一个套圈游戏场中的布局，固定每个玩具的位置，试设计圆环套圈的半径尺寸，使得它每次最多只能套中一个玩具。但同时为了让游戏看起来更具有吸引力，这个套圈的半径又需要尽可能大

- 空中交通控制问题：监控空中飞行的飞机，找出其中碰撞危险最大的飞机 (给予警告或修正航线)
- 问题的抽象 ——**最接近点对问题**[†]：给定平面上 (空间) n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小
- 解决方案：
  - 两两计算点对距离，找出其中的最小值，复杂度为 $O(n^2)$
  - 可以证明该问题的复杂度下界为 $\Omega(n\log n)$
  - 是否存在复杂度为 $\Theta(n\log n)$ 的算法？ ——分治法？

分治法思路

- 将所给的平面上 n 个点的集合 S 分成 2 个子集 S_1 和 S_2，每个子集中约有 $n/2$ 个点
- 在每个子集中递归地求其最接近的点对，但 S_1 和 S_2 的最接近点对未必就是 S 的最接近点对
- **关键问题**：如何合并子问题求解的结果，即由 S_1 和 S_2 的最接近点对，如何求得原集合 S 中的最接近点对
- 一维的情况：



  - 假定分割点为 m，则如果出现最小点对分别落在 S_1 和 S_2 中的情况，这两点只可能是 S_1 中的最大点和 S_2 中的最小点 $(O(n))$
  - 分割点 m 的选择：中位数 $(\Theta(n))$
  - 时间复杂度：$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n\log n)$

一维情况算法描述

Algorithm `MinPairI`(S)

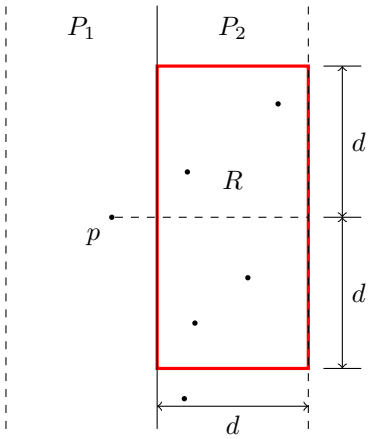
```
1  if |S| < 2 then d ← ∞ ;
2  else
3      m ← S 中点坐标中位数;
4      S1 ← {x ∈ S|x ≤ m};
5      S2 ← {x ∈ S|x > m};
6      d1 ← MinPairI(S1);
7      d2 ← MinPairI(S2);
8      x1 ← max(S1);
9      x2 ← min(S2);
10     d ← min(d1, d2, x2 − x1);
11 end
12 return d;
```

[†]王晓东. 计算机算法设计与分析 (第 2 版). 电子工业出版, 2004.

推广到二维

- 点集分割：为了将平面上点集线性分割为大小大致相等的 2 个子集 S_1 和 S_2，选取一垂直线 $l : x = m$ 来作为分割直线。其中 m 为 S 中各点 x 坐标的中位数
- 递归求解：分别得到 S_1 和 S_2 中的最小距离 d_1 和 d_2
- 若设 $d = \min(d_1, d_2)$，则如果 S 中的最接近点对 (p, q) 距离小于 d，则 p 和 q 必分属于 S_1 和 S_2，如何找到距离可能小于 d 的点对？
  - p, q 必然位于直线 l 两侧左右各宽 d 的垂直长条内，设为 P_1、P_2
  - 对于 P_1 中任一候选点，P_2 中能与其构成候选点对的点必然落在一个 $d \times 2d$ 的矩形 R 中
  - 矩形 R 中最多只有 6 个点 (鸽笼原理)
  - 如何在线性时间复杂度内找到这 6 个点？—— 对 y 坐标预排序

示意图



算法描述

Algorithm `MinPairIIRec`(P)

```
1  if |S| < 2 then d ← ∞ ;
2  else
3      m ← P 中点的 x 坐标中位数;
4      P1[] ← {P[i]_x ≤ m};
5      P2[] ← {P[i]_x > m};
6      d1 ← MinPairIIRec(P1);
7      d2 ← MinPairIIRec(P2);
8      d ← dm ← min(d1, d2);
9      P1*[] ← {m − P1[i]_x < dm};
10     P2*[] ← {P2[i]_x − m < dm};
11     foreach P1*[i] do
12         搜索 P2* 中与 P1*[i] 距离小于 dm 的点 (最多 6 个) 更新 d;
13     end
14 end
15 return d;
```

算法分析

- 3, 4, 5, 9, 10 显然复杂度为 $O(n)$
- 8 的开销是常数
- 6, 7 是递归部分，复杂度为 $2W(n/2)$
- 关键是 11～13，在 P 是有序的前提下才能够达到线性复杂度，而且在递归调用的任何时候都不会破坏子集中的点 y 坐标的有序性
- 在上述条件下，`MinPairIIRec` 的时间复杂度：$W(n) = 2W(n/2) + O(n) \Rightarrow W(n) \in \Theta(n \log n)$
- 因此在调用 `MinPairIIRec` 之前需要对 P 中的点按 y 坐标排序，排序的时间复杂度可以达到 $\Theta(n \log n)$，所以不影响整个算法的时间复杂度

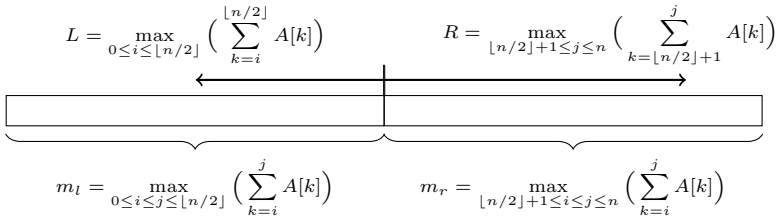
Procedure `MinPairII`(S)

```
1 P ← S 中的点按 y 坐标排序;
2 MinPairIIRec(P);
```

2.5 股价增值问题

问题描述及分析

- 某公司过去 n 天的股票价格波动保存在数组 A 中，求在哪段时间 (连续哪几天) 其股价的累计增长最大
- 例如：过去 7 天内其股价波动序列为 $+3, -6, +5, +2, -3, +4, -4$，累计增长最大值出现在第 3 天到第 6 天，累计增长值为 $5 + 2 - 3 + 4 = 8$
- 实际上就是求 i 和 j $(0 \le i \le j \le n-1)$，使 $\sum_{k=i}^{j} A[k]$ 最大
- 采用分治法设计算法：



$$m = \max(m_l, m_r, L + R)$$

算法描述

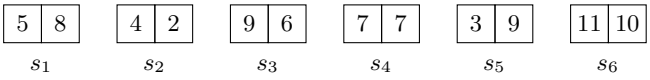
Algorithm `LargestIncrease`$(A[], first, last)$

```
1  if first ≥ last then i ← first, j ← last, m ← A[last] ;
2  else
3      mid ← ⌊(first + last)/2⌋;
4      (il, jl, ml) ← LargestIncrease(A, first, mid);
5      (ir, jr, mr) ← LargestIncrease(A, mid + 1, last);
6      找到 im 使得 L = Σ_{k=im}^{mid} A[k] 最大;
7      找到 jm 使得 R = Σ_{k=mid+1}^{jm} A[k] 最大;
8      if ml ≥ mr and ml ≥ L + R then i ← il, j ← jl, m ← ml ;
9      else if mr ≥ ml and mr ≥ L + R then i ← ir, j ← jr, m ← mr ;
10     else i ← im, j ← jm, m ← L + R ;
11 end
12 return (i, j, m);
```

- 算法复杂度：$W(n) = 2W(n/2) + \Theta(n) \Rightarrow \Theta(n \log n)$

课后习题：多米诺骨牌

Exercise (3). 现有 n 块“多米诺骨牌” $s_1, s_2, \cdots, s_n$ 水平放成一排，每块骨牌 s_i 包含左右两个部分，每个部分赋予一个非负整数值，如下图所示为包含 6 块骨牌的序列。骨牌可做 180 度旋转，使得原来在左边的值变到右边，而原来在右边的值移到左边，假设不论 s_i 如何旋转，$L[i]$ 总是存储 s_i 左边的值，$R[i]$ 总是存储 s_i 右边的值，$W[i]$ 用于存储 s_i 的状态：当 $L[i] \leq R[i]$ 时记为 0，否则记为 1，试采用分治法设计算法求 $\sum_{i=1}^{n-1} R[i] \cdot L[i+1]$ 的最大值，以及当取得最大值时每个骨牌的状态。下面是 $n = 6$ 时的一个例子：



deadline: 2018.12.08

算法概论

第三讲：排序算法

薛健

Last Modified: 2018.12.9

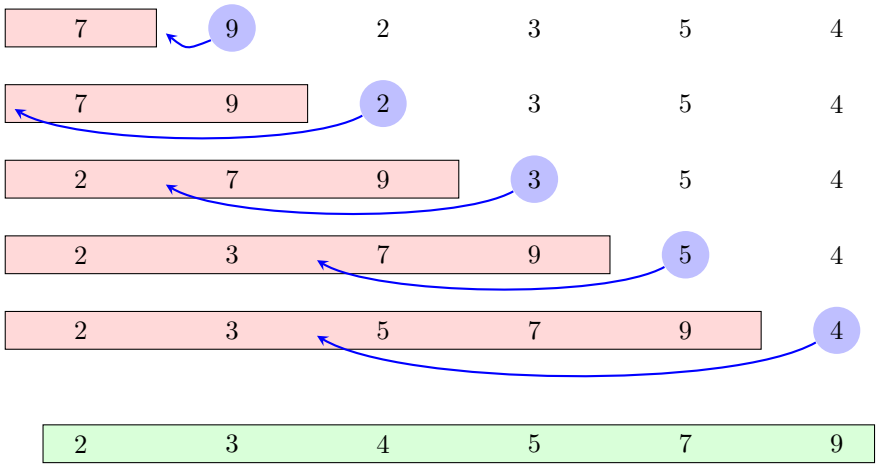
主要内容

1 简单排序算法

1.1 插入排序

基本策略

- 依次将序列元素插入到已排好序的部分：



算法

Algorithm `InsertionSort`$(E[], n)$

```
1 for xindex ← 1 to n − 1 do
2 |   current ← E[xindex];
3 |   xloc ← ShiftVac(E, xindex, current);
4 |   E[xLoc] ← current;
5 end
```

ShiftVac

Algorithm `ShiftVac`$(E[], xindex, x)$

```
1  vacant ← xindex;
2  xloc ← 0;
3  while vacant > 0 do
4  |   if E[vacant − 1] ≤ x then
5  |   |   xloc ← vacant;
6  |   |   break;
7  |   end
8  |   E[vacant] ← E[vacant − 1];
9  |   vacant ← vacant − 1;
10 end
11 return xloc;
```

复杂度分析

- **基本操作**：元素比较
- Worst-Case：
  - $W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$
- Average-Case：
  - 不妨假设序列元素各不相同
  - 对于 `ShiftVac`，将当前第 i 个待排序元素插入前 i 个排好序的元素序列中，共有 $i+1$ 个可能的插入位置，在每个位置插入的可能性为 $1/(i+1)$，则平均情况下比较次数为
    $$\frac{1}{i+1}\sum_{j=1}^{i} j + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$
  - `InsertionSort` 总共进行了 $n-1$ 次插入操作，比较次数总和为
    $$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{j} \approx \frac{n^2}{4} \in \Theta(n^2)$$

1.2 类似算法分析

类似算法的复杂度下界

- **类似算法**：每次比较之后仅交换一对相邻元素位置的排序算法
- 设原序列 $E = (x_1, x_2, \cdots, x_n)$，定义 π 为原序列的一种排列，即对于 $1 \le i \le n$，$\pi(i)$ 是 x_i 在排好序的序列中的实际位置

- 排列中的反序对 $(\pi(i), \pi(j))$：$i < j$ 且 $\pi(i) > \pi(j)$
- 上述算法的最少比较次数等于输入序列中反序对的个数
- 有一种排列其反序对个数为 $n(n-1)/2$，因此上述算法最坏情况下复杂度下界为 $n(n-1)/2 \in \Omega(n^2)$
- 平均情况？

Theorem 1.1. 任何使用比较为基本操作，并且每次比较后最多去除一个反序对的排序算法，输入规模为 n 时，在最坏情况下的最少比较次数为 $n(n-1)/2$，在平均情况下的最少比较次数为 $n(n-1)/4$

另一个类似算法：选择排序

Algorithm `SelectionSort`$(E[], n)$

```
1 for i ← 0 to n − 1 do
2 |   k ← i;
3 |   for j ← i to n − 1 do
4 |   |   if E[j] < E[k] then k ← j;
5 |   end
6 |   E[i] ↔ E[k];
7 end
```

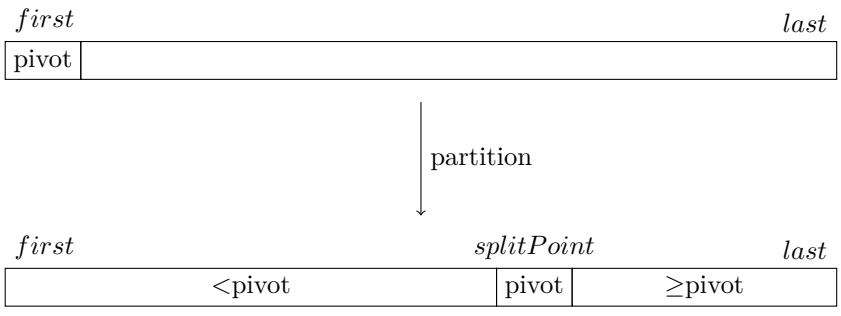
- Worst-Case：$W(n) = n(n-1)/2$
- Average-Case：$A(n) = n(n-1)/2$
- **优点**：不需频繁移动数组元素

2 用分治法设计排序算法

2.1 快速排序

基本策略

- 将原序列分解为两个子序列，使一个子序列里面的元素小于另一个子序列中的元素
- 对两个子序列分别排序 (递归求解)
- C. A. R. Hoare 在 1962 年提出

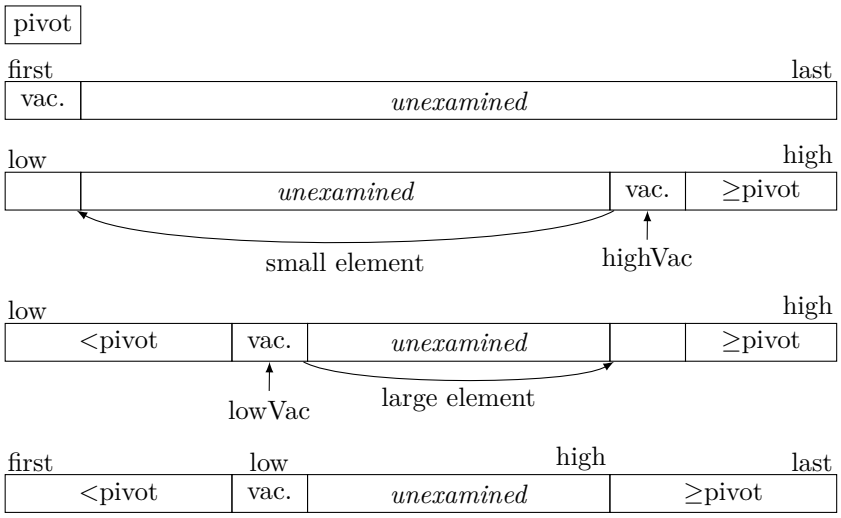


`QuickSort` 算法描述

Algorithm `QuickSort`$(E[], first, last)$

```
1 if first < last then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 end
```

In Place `Partition`



`Partition`

Procedure `Partition`$(E[], pivot, first, last)$

```
1 low ← first;
2 high ← last;
3 while low < high do
4 |   highVac ← ExtendLargeRegion(E, pivot, low, high);
5 |   lowVac ← ExtendSmallRegion(E, pivot, low + 1, highVac);
6 |   low ← lowVac;
7 |   high ← highVac − 1;
8 end
9 return low;
```

`ExtendLargeRegion`

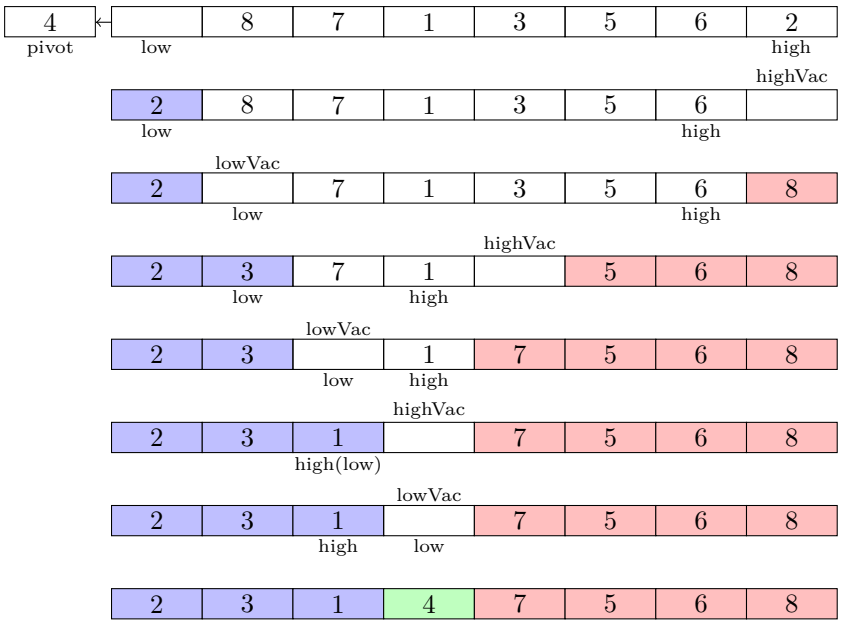
Procedure `ExtendLargeRegion`$(E[], pivot, lowVac, high)$

```
1  highVac ← lowVac;                          // In case no element < pivot.
2  curr ← high;
3  while curr > lowVac do
4  |  if E[curr] < pivot then
5  |  |  E[lowVac] ← E[curr];
6  |  |  highVac ← curr;
7  |  |  break;
8  |  end
9  |  curr ← curr − 1;
10 end
11 return highVac;
```

`ExtendSmallRegion`

Procedure `ExtendSmallRegion`$(E[], pivot, low, highVac)$

```
1  lowVac ← highVac;                          // In case no element ≥ pivot.
2  curr ← low;
3  while curr < highVac do
4  |  if E[curr] ≥ pivot then
5  |  |  E[highVac] ← E[curr];
6  |  |  lowVac ← curr;
7  |  |  break;
8  |  end
9  |  curr ← curr + 1;
10 end
11 return lowVac;
```



最坏情况复杂度分析

- `Partition`: 对于长度为 k 的序列，共需比较 $k-1$ 次；
- 最坏情况下的 `Partition` 结果是 $splitPoint = first$ or $last$，原序列被分成一个空序列和一个比原序列少一个元素 (pivot) 的子序列；

- 因此，最坏情况下时间复杂度为：

$$W(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

- *QuickSort* 徒有虚名？

平均情况复杂度分析

- 我们已经知道：当一次比较最多仅能消除一个反序对的时候，平均情况下比较次数至少为 $n(n-1)/4$
- 对于 QuickSort 来说，一次比较后，元素在序列中可能移动相当大的距离，这意味着每次比较后可能消除多个反序对 (最多 $n-1$)
- 每次 Partition 之后得到的两个子序列各自的元素之间两两未经比较，因此可以认为子序列各种排序出现的概率保持相等
- 假设序列元素每种排列出现的概率相等，Partition 返回的 *splitPoint* 位置出现在每个序列元素位置的概率也相等，则：

$$A(n) = (n-1) + \sum_{i=0}^{n-1}\frac{1}{n}(A(i) + A(n-1-i)) \qquad \text{for } n \geq 2$$
$$A(1) = A(0) = 0$$

- 整理一下得：$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \quad \text{for } n \geq 1$

求 $A(n)$

- 做一个猜想：如果 Partition 每次都将序列分为长度相同的两个子序列

$$A(n) \approx n + 2A(n/2) \xrightarrow{\text{主定理}} A(n) \in \Theta(n\log n)$$

Theorem 2.1. 对于如下定义的 $A(n)$：

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \quad \textit{for } n \geq 1$$

存在常数 c，使得 $A(n) \leq cn\ln n$.

- 一个更精确的近似：$A(n) \approx 1.386n\lg n - 2.846n$

Proof. 用数学归纳法。
奠基：$A(1) = 0 \leq c1\ln 1 = 0$.
归纳：

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}A(i) \leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1}ci\ln i$$

用积分定界：

$$\sum_{i=1}^{n-1}ci\ln i \leq c\int_1^n x\ln x dx = c\left(\frac{1}{2}n^2\ln n - \frac{1}{4}n^2\right)$$

因此：

$$A(n) \leq n - 1 + \frac{2c}{n}\left(\frac{1}{2}n^2\ln n - \frac{1}{4}n^2\right) = cn\ln n + n(1 - \frac{c}{2}) - 1$$

当 $c \geq 2$ 时，$A(n) \leq cn\ln n$ □

更精确的分析：

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$

$$A(n-1) = n - 2 + \frac{2}{n-1}\sum_{i=1}^{n-2} A(i)$$

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

令：

$$B(n) = \frac{A(n)}{n+1}$$

即得：

$$B(n) = B(n-1) + \frac{2(n-1)}{n(n+1)} \qquad B(1) = 0$$

$$B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)} \approx 2(\ln n + 0.577) - \frac{4n}{n+1}$$

因此：

$$A(n) \approx 1.386n \lg n - 2.846n \qquad (\text{注：} \ln n = \frac{\lg n}{\lg e} \approx 0.693 \lg n)$$

改进

- 选择合适的 pivot
  - 随机选择；选择 $E[first]$、$E[(first + last)/2]$、$E[last]$ 的中间值
- 减少递归调用
  - 将递归变为循环；减小递归调用深度 (`SmallSort`)

Algorithm `QuickSort`$(E[], first, last)$

```
1 if last − first > smallSize then
2 |   pivot ← E[first];
3 |   splitPoint ← Partition(E, pivot, first, last);
4 |   E[splitPoint] ← pivot;
5 |   QuickSort(E, first, splitPoint − 1);
6 |   QuickSort(E, splitPoint + 1, last);
7 else
8 |   SmallSort(E, first, last);
9 end
```

改进 (cont.)

- 快速排序的递归深度：$O(n) \Rightarrow O(\lg n)$

Algorithm `QuickSort2`$(E[], first, last)$

```
1  while first < last do
2  |  pivot ← E[first];
3  |  splitPoint ← Partition(E, pivot, first, last);
4  |  E[splitPoint] ← pivot;
5  |  if splitPoint − first < last − splitPoint then
6  |  |  QuickSort2(E, first, splitPoint − 1);
7  |  |  first ← splitPoint + 1;
8  |  else
9  |  |  QuickSort2(E, splitPoint + 1, last);
10 |  |  last ← splitPoint − 1;
11 |  end
12 end
```

2.2 归并排序

第三次作业：平均情况下快速排序和归并排序移动元素的次数

归并操作 (Merge)

- **归并操作**，或**归并算法**，指的是将两个有序序列合并成一个有序序列的操作
- **基本原理**：
  1. 申请空间，使其大小为两个已经排序序列之和，用来存放合并后的序列；
  2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
  3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
  4. 重复上一步骤直到某一指针达到序列尾；
  5. 将另一序列剩下的所有元素直接复制到合并序列尾

递归描述

Algorithm `Merge`$(A[], B[], C[])$

```
1  if A is empty then rest of C ← rest of B;
2  else if B is empty then rest of C ← rest of A;
3  else
4  |  if first of A ≤ first of B then
5  |  |  first of C ← first of A;
6  |  |  Merge(rest of A, B, rest of C);
7  |  else
8  |  |  first of C ← first of B;
9  |  |  Merge(A, rest of B, rest of C);
10 |  end
11 end
```

非递归版本

Algorithm `Merge`$(A[], k, B[], m, C[])$

```
1  n ← k + m;
2  indexA ← indexB ← indexC ← 0;
3  while indexA < k and indexB < m do
4      if A[indexA] ≤ B[indexB] then
5          C[indexC] ← A[indexA];
6          indexA ← indexA + 1;
7          indexC ← indexC + 1;
8      else
9          C[indexC] ← B[indexB];
10         indexB ← indexB + 1;
11         indexC ← indexC + 1;
12     end
13 end
14 if indexA ≥ m then  Copy B[indexB, ..., m − 1] to C[indexC, ..., n − 1] ;
15 else  Copy A[indexA, ..., k − 1] to C[indexC, ..., n − 1] ;
```

归并操作时间复杂度分析

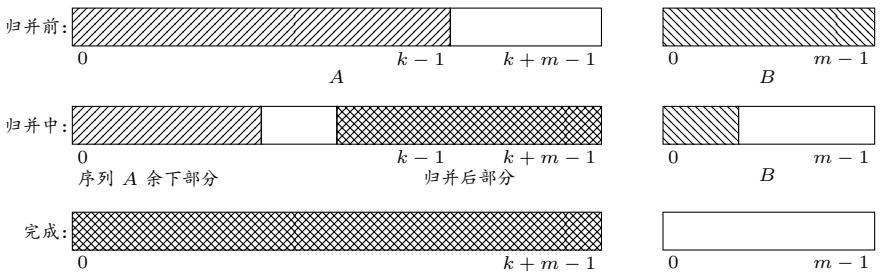
- Worst-Case：
  - 每做完一次比较，至少有一个元素被移到合并后的序列中
  - 最后一次比较时，至少还剩 2 个元素未移到合并序列中，这时合并序列中最多有 $n-2$ 个元素，加最后一次至多经过了 $n-1$ 次比较
  - $W(n) = n - 1 \in \Theta(n)$
- Average-Case?

Theorem 2.2. 任何对两个均包含 $k = m = n/2$ 个元素的有序序列，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

Corollary 2.3. 任何对两个分别包含 k 和 m 个元素的有序序列，并且 $|k-m| = 1$，仅使用元素间的比较操作进行归并的算法，在最坏情况下至少需要 $n-1$ 次比较

归并操作空间复杂度分析

- 前面的算法在存储空间占用上，除了输入数据 $k+m=n$ 个元素占用的空间外，显然还需要 n 个元素的额外空间存储结果
- 如果输入序列 A 和 B 用链表存储的话，则可以做到不需要额外存储空间，但前提是不需要保持输入数据
- 如果输入序列 A 和 B 用数组存储，且输入数据不需要保持，则额外空间占用可以有一定程度的减少



最坏情况下额外占用的存储空间：$n/2 \in \Theta(n)$ 当 $k = m = n/2$

归并排序 (Mergesort)

- 快速排序最大的问题实际上是每次序列分割不一定能将原序列分成长度相等的两个子序列
- **归并排序**则每次将序列分割为恰好相等的两个子序列，分别对子序列进行递归排序，再将排好序的两个子序列通过归并操作合并成一个序列

Algorithm `Mergesort`$(E[], first, last)$

```
1 if first < last then
2 |   mid ← (first + last)/2;
3 |   Mergesort(E, first, mid);
4 |   Mergesort(E, mid + 1, last);
5 |   Merge(E, first, mid, last);
6 end
```

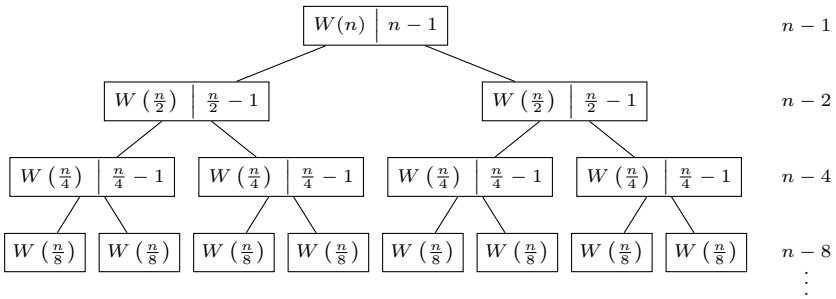
归并排序时间复杂度

- Worst-Case:

$$
\begin{aligned}
&W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \\
&W(1) = 0 \\
&\qquad \Downarrow \text{主定理} \\
&W(n) \in \Theta(n \log n)
\end{aligned}
$$

- 一个更精确的结果：$\lceil n \lg n - n + 1 \rceil \le W(n) \le \lceil n \lg n - 0.914n \rceil$

`Mergesort` 递归树



- 不包含叶结点的每层非递归开销为：$n - 2^d$
- 叶结点 $(W(1) = 0)$ 深度为 $\lceil \lg(n+1) \rceil - 1$ 或 $\lceil \lg(n+1) \rceil$
- 恰有 n 个叶结点
- 令递归树最大深度 (高度) 为 $D = \lceil \lg(n+1) \rceil$，在深度为 $D-1$ 层有 B 个叶结点，则在 D 层有 $n - B$ 个叶结点，在 $D-1$ 层有 $(n-B)/2$ 个非叶结点，且 $B = 2^D - n$ ($Why?$)

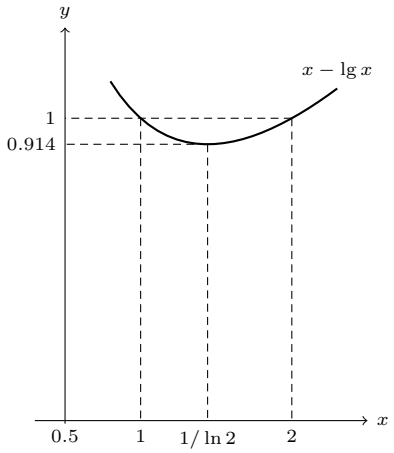
时间复杂度计算

$$
\begin{aligned}
W(n) &= \sum_{d=0}^{D-2}(n-2^d) + \frac{n-B}{2}W(2) \\
&= n(D-1) - 2^{D-1} + 1 + \frac{n-B}{2} \\
&= nD - 2^D + 1
\end{aligned}
$$

$$
\alpha = \frac{2^D}{n} \Rightarrow 1 \le \alpha < 2
$$

$$
W(n) = n\lg n - (\alpha - \lg\alpha)n + 1
$$

$$
0.914 \le \alpha - \lg\alpha \le 1
$$



$$
\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil
$$

一点讨论

- **快速排序时间复杂度：** $W(n) \in \Theta(n^2),\ A(n) \approx 1.386n\lg n - 2.846n$
- **归并排序：** $\lceil n\lg n - n + 1\rceil \le W(n) \le \lceil n\lg n - 0.914n\rceil$ Java中标准排序std::sort()使用的是快速排序
- 是否意味着归并排序比快速排序更好？ 结论：平均意义而言，快速排序比归并排序好，也更多被采用
  - 时间复杂度
  - 空间复杂度 归并排序需要额外的空间，所以归并排序的空间复杂度比快速排序大

Exercise (4). 试分析比较快速排序和归并排序在平均情况下的元素移动次数。

deadline: 2018.12.15

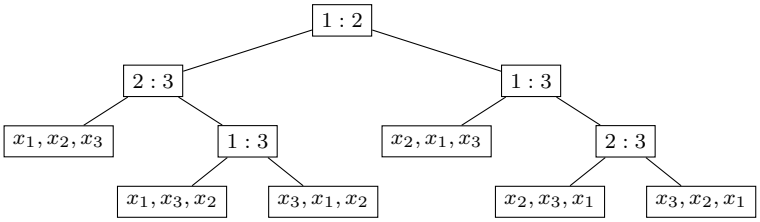
2.3 比较排序的复杂度下界

排序算法的决策树

- 任意一种以元素间的比较为基本操作的排序算法其执行过程都对应于一棵二叉决策树：
  - 内结点标注 $(i:j)$，表示数组元素 x_i 与 x_j 的一次比较操作，其左子树对应 $x_i < x_j$ 的情况下下一步要进行的比较操作（或输出结果操作），右子树对应 $x_i > x_j$ 的情况下下一步要进行的比较操作（或输出结果操作）

  - 每个外结点表示一个排序结果

- 例：输入规模 $n = 3$ 时的决策树：



最坏情况复杂度下界

Lemma 2.4. 若高度为 h 的二叉树有 L 个叶结点，则有：$L \le 2^h$ 及 $h \ge \lceil \lg L \rceil$

Lemma 2.5. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法其对应决策树的高度至少为 $\lceil \lg n! \rceil$

Theorem 2.6. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法在最坏情况下至少要执行 $\lceil \lg n! \rceil \approx \lceil n \lg n - 1.443n \rceil$ 次比较操作

$$\text{注：}\ \lg n! = \sum_{j=1}^{n} \lg j \ge n \lg n - (\lg e)n$$

归并排序的平均时间复杂度nlgn

平均情况复杂度下界

- 根据决策树定义，很显然是 2-tree，而每一条从根结点到叶结点的路径代表了一次成功的排序过程
- 设决策树外路径长度为 epl，叶结点个数为 L，则平均比较次数为 epl/L
- $\Longrightarrow$ 找到 epl 的下界：$epl \ge L \lg L$

Theorem 2.7. 对于给定输入规模 n，任何以元素比较为基本操作的排序算法在平均情况下至少要执行 $\lg n! \approx n \lg n - 1.443n$ 次比较操作

2.4 堆排序

回顾：二叉堆 (Binary Heap)

- **堆** (Heap)：基于树的数据结构，其满足堆特性：若 B 是 A 的子结点，则 $\text{key}(A) \ge \text{key}(B)$ (大根堆)
- **二叉堆**：用完全二叉树来表达的堆结构，是实现优先队列的最有效的数据结构之一
- **二叉堆的基本操作**：
  - 往堆中添加元素 (heapify-up, up-heap, bubble-up)：
    1. 将新元素插入堆的尾部
    2. 将新元素与其父结点比较，若满足堆特性，则结束
    3. 否则，将其与父结点交换位置，并重复上一步骤
  - 提取并删除堆首元素 (heapify-down, down-heap, bubble-down)：
    1. 将堆尾元素放到堆首代替已删除的堆首元素
    2. 将其与子结点比较，若满足堆特性，则结束
    3. 否则，将其与子结点中的一个交换位置，使三者满足堆特性，并重复上一步骤
- ✠ **效率**：入队和出队的复杂度 $W(n) \in O(\log n)$

堆排序 (Heapsort) 算法要点

Algorithm `Heapsort`$(E[], n)$

```
1 从 E 构造堆 H;
2 for i = n down to 2 do
3 |   curMax ← GetMax(H);
4 |   DeleteMax(H);
5 |   E[i] ← curMax;
6 end
```

Algorithm `DeleteMax`(H)

```
1 拷贝堆 H 的底层最右边结点中的元素到 K 中;
2 删除堆 H 的底层最右边结点中的元素;
3 FixHeap(H, K);
```

堆的调整：`FixHeap`

Algorithm `FixHeap`(H, K)

```
1  if H 是叶结点 then
2  |   将 K 插入 Root(H);
3  else
4  |   取左右子树中根结点元素较大的那棵作为 largerSubHeap;
5  |   if K ≥Root(largerSubHeap) then
6  |   |   将 K 插入 Root(H);
7  |   else
8  |   |   将 Root(largerSubHeap) 插入 Root(H);
9  |   |   FixHeap(largerSubHeap, K);
10 |   end
11 end
```

堆的构造：`ConstructHeap`

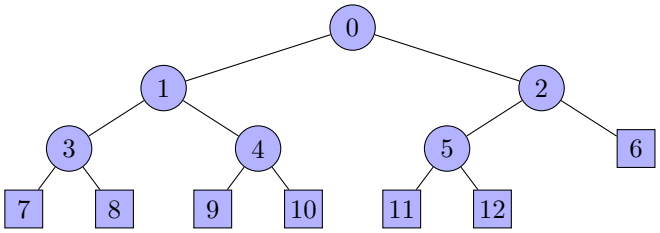
Algorithm `ConstructHeap`(H)

```
1 if H 不是叶结点 then
2 |   ConstructHeap(H 的左子树);
3 |   ConstructHeap(H 的右子树);
4 |   K ← Root(H);
5 |   FixHeap(H, K);
6 end
```

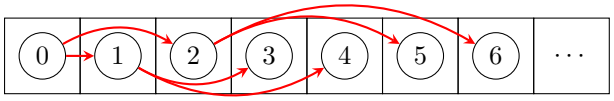
- 最坏情况复杂度分析：
  - `FixHeap`: 大约需要 $2\lg n$ 次比较
  - `ConstructHeap`: $W(n) = W(n-r-1)+W(r)+2\lg n$, r 为右子树结点数, 且 $n > 1$
  - $\Rightarrow W(n) \in \Theta(n)$ (*Why?*)

回顾：二叉堆 (完全二叉树) 的数组存储方式

Example 2.1 (完全二叉树).



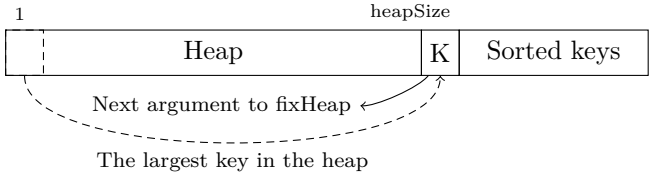
完全二叉树的顺序存储方式：



数组实现版本

Algorithm `Heapsort`$(E[], n)$

```
1 ConstructHeap(E, n);
2 for heapSize ← n down to 2 do
3 |   curMax ← E[0];
4 |   K ← E[heapSize − 1];
5 |   FixHeap(E, heapSize − 1, 0, K);
6 |   E[heapSize − 1] ← curMax;
7 end
```



`FixHeap` 的数组实现

Algorithm `FixHeap`$(E[], heapSize, root, K)$

```
1  left ← 2 ∗ root + 1, right ← 2 ∗ root + 2;
2  if left ≥ heapSize then E[root] ← K;
3  else
4  |   if left = heapSize − 1 then largerSubHeap ← left;
5  |   else if E[left] > E[right] then largerSubHeap ← left;
6  |   else largerSubHeap ← right;
7  |   if K ≥ E[largerSubHeap] then E[root] ← K;
8  |   else
9  |   |   E[root] ← E[largerSubHeap];
10 |   |   FixHeap(E, heapSize, largerSubHeap, K);
11 |   end
12 end
```

堆排序算法复杂度分析

最坏情况：

- `ConstructHeap`: $\in \Theta(n)$
- `FixHeap`: $2\lfloor \lg k \rfloor$

- 循环：$2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor$

$$\begin{aligned} 2\sum_{k=1}^{n-1}\lfloor \lg k \rfloor &\le 2\int_1^n (\lg e)\ln x dx \\ &= 2(\lg e)(n\ln n - n) \\ &= 2(n\lg n - 1.443n) \end{aligned}$$

Theorem 2.8. `Heapsort` 排序算法在最坏情况下所需的比较操作次数为 $2n\lg n + O(n)$，其时间复杂度为 $\Theta(n\log n)$

平均情况?

改进

Algorithm `BubbleUpHeap`$(E[], root, K, vacant)$

```
1 if vacant = root then E[vacant] = K;
2 else
3 |   parent ← (vacant − 1)/2;
4 |   if K ≤ E[parent] then  E[vacant] ← K ;
5 |   else
6 |   |   E[vacant] ← E[parent];
7 |   |   BubbleUpHeap(E, root, K, parent);
8 |   end
9 end
```

- 取出堆的最大元素后，首先每次将左右子结点中的较大元素放到根结点空位上而暂时不与 K 比较 (risky `FixHeap`)
- 当空位降到底层后使用 `BubbleUpHeap` 将 K “提升”(bubble up) 到适当位置
- 由于我们取的 K 是堆尾元素，应该是一个“相当小”的元素，因此在平均情况下应该用比空位下降更少的比较就能上升到合适位置
- 注意：最坏情况仍然需要 $2\lfloor \lg k \rfloor$ 次比较
- 能不能做得更好一些？

进一步的改进 —— Divide and Conquer

- Risky `FixHeap` 每次只下降当前堆 $\frac{h}{2}$ 高度 (可以称为 `Promote`)
- 当下降到当前空位父结点元素小于 K 时开始 `BubbleUpHeap`

Algorithm `Promote`$(E[], hStop, vacant, h)$

```
1  left ← 2 ∗ vacant + 1;
2  right ← 2 ∗ vacant + 2;
3  if h ≤ hStop then
4  |   vacStop ← vacant;
5  else if E[left] ≤ E[right] then
6  |   E[vacant] ← E[right];
7  |   vacStop ← Promote(E, hStop, right, h − 1);
8  else
9  |   E[vacant] ← E[left];
10 |   vacStop ← Promote(E, hStop, left, h − 1);
11 end
12 return vacStop;
```

更快的 FixHeap

Algorithm FixHeapFast($E[], n, K, vacant, h$)

```
1  if h ≤ 1 then
2  |  处理高度为 0 或 1 的情况;
3  else
4  |  hStop ← h/2;
5  |  vacStop ← Promote(E, hStop, vacant, h);
6  |  vacParent ← (vacStop − 1)/2;
7  |  if E[vacParent] ≤ K then
8  |  |  E[vacStop] = E[vacParent];
9  |  |  BubbleUpHeap(E, vacant, K, vacParent);
10 |  else
11 |  |  FixHeapFast(E, n, K, vacStop, hStop);
12 |  end
13 end
```

时间复杂度分析

- 直观地看:
  - 如果在 $\frac{h}{2}$ 处 BubbleUpHeap 就被调用，需要最多 $\frac{h}{2}$ 次比较找到 K 的合适位置，而之前 Promote 也用去了 $\frac{h}{2}$ 次元素比较
  - 若在 $\frac{h}{4}$ 处 BubbleUpHeap 被调用,则 Promote 之前共进行了 $\frac{3h}{4}$ 次比较,而 BubbleUpHeap 此时只需返回最多 $\frac{h}{4}$ 的高度
  - 以此类推，递归过程中 Promote 的比较次数加上大约一次 BubbleUpHeap 调用需要约 $h+1$ 次比较
  - FixHeapFast 本身的非递归开销大约为 $\lg h$ 次比较
  - 因此，总的元素比较次数大约为 $h+\lg h$
- 递推方程: $T(h) = \lceil h/2 \rceil + 1 + \max\left(\lceil h/2 \rceil, T(\lfloor h/2 \rfloor)\right) \quad T(1) = 2$
- 假设 $T(h) \geq h$: $T(h) = \lceil h/2 \rceil + 1 + T(\lfloor h/2 \rfloor) \quad T(1) = 2 \Longrightarrow T(h) \approx h + \lceil \lg(h+1) \rceil \approx \lg(n+1) + \lg\lg(n+1)$

Theorem 2.9. 使用 FixHeapFast 来调整堆结构的堆排序算法在最坏情况下其时间复杂度为 $n\lg n + \Theta(n \log\log n)$

3 其他排序算法

3.1 希尔排序

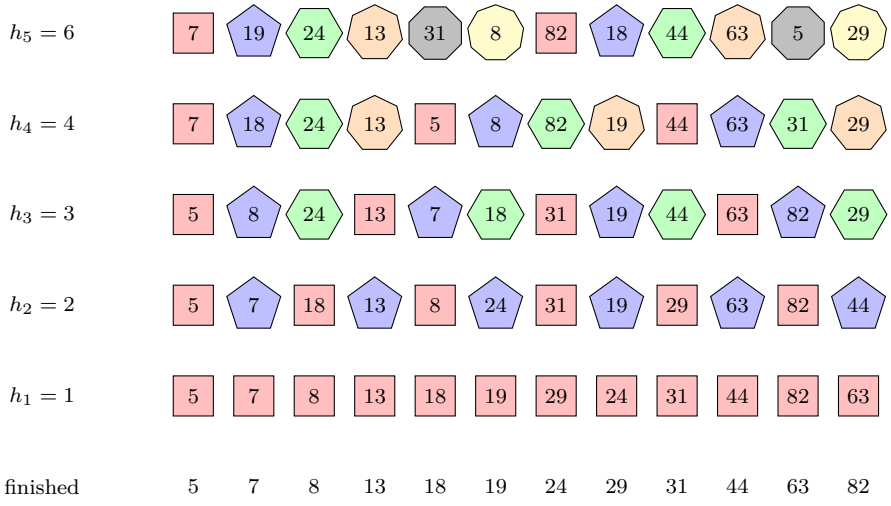
基本思想

- 回顾: **插入排序** (InsertionSort)
  - 如果原始数据的大部分元素已经有序，那么插入排序的速度很快 (因为需要移动的元素很少)
  - 从这个事实我们可以想到，如果原始数据只有很少元素，那么排序的速度也很快 (SmallSort)
  - 插入排序的低效之处在于每次比较交换相邻元素最多只能消除一个反序对
- **希尔排序** (Shell Sort)又称**缩小增量排序** (diminishing increment sort)，就是利用插入排序的特点设计的一种优秀的排序法，算法本身不难理解，也易于实现，而且速度很快。其基本原理为:

  - 指定一组递减增量序列 (最后一个增量为 1)
  - 依次取序列中的增量为矩阵行宽，将待排序序列按矩阵排列
  - 对矩阵每一列进行插入排序

- 算法名称取自其发明人 D. L. Shell 的名字，于 1959 年发表

示例



算法描述

Algorithm `Shellsort`$(E[], n, h[], t)$

```
1 for s ← t − 1 to 0 do
2 |  for xindex ← h[s] to n − 1 do
3 |  |  current ← E[xindex];
4 |  |  xloc ←ShiftVacH(E, h[s], xindex, current);
5 |  |  E[xloc] ← current;
6 |  end
7 end
```

`ShiftVacH`

Procedure `ShiftVacH`$(E[], h, xindex, cur)$

```
1 vacant ← xindex;
2 while vacant ≥ h do
3 |  if E[vacant − h] ≤ cur then
4 |  |  break;
5 |  end
6 |  E[vacant] ← E[vacant − h];
7 |  vacant ← vacant − h;
8 end
9 return vacant;
```

复杂度分析

- `Shellsort` 的时间复杂度取决于增量序列的选择，其分析难度较大，仍然是一个开放的问题

- 已知复杂度上界
  - [Pratt, 1971]：$\Theta(n(\log n)^2)$，
    增量序列：$1, 2, 3, 4, 6, 9, 8, 12, 18, 27, \cdots, 2^i 3^j$ (2+3)^n的系数，比如(2+3)^2,系数为4+2*6+9
  - [Papernov-Stasevich, 1965; Pratt, 1971]：$\Theta(n^{3/2})$，
    增量序列：$1, 3, 7, 15, \cdots, 2^k - 1$
  - [Sedgewick, 1982]：$O(n^{4/3})$，
    增量序列：$1, 8, 23, 77, 281, 1073, 4193, 16577, \cdots, 4^{j+1} + 3 \cdot 2^j + 1$
  - [Incerpi-Sedgewick, 1985]：对于任意 $\epsilon > 0$，必存在一种增量序列使得 `Shellsort` 的时间复杂度为 $O(n^{1+\epsilon/\log n})$，且只用 $\frac{8}{\epsilon^2}\log n$ 遍排序
- 已知复杂度下界
  - [Poonen, 1993]：对大小为 n 的序列进行 m 遍排序的 `Shellsort` 至少需要 $n^{1+c/\sqrt{m}}$ 次比较 $(c > 0)$
- 平均情况
  - [Knuth, 1973]：两遍 $(h, 1)$ `Shellsort` 需要 $2n^2/h + \sqrt{\pi n^3 h}$ 次比较 $(h \in O(n^{1/3}) \Rightarrow A(n) \in O(n^{5/3}))$
  - [Yao, 1980]：三遍 $(h, k, 1)$ `Shellsort` 需要 $\frac{2n^2}{h} + \frac{1}{k}\left(\sqrt{\frac{\pi n^3 h}{8}} - \sqrt{\frac{\pi n^3}{8h}}\right) + \psi(h, k)n$ 次比较
- 悬而未决的问题
  - 是否还存在更好的增量序列？
  - 最坏情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 平均情况下 `Shellsort` 时间复杂度是否能达到 $O(n \log n)$？
  - 如果使用其他排序算法代替每遍排序使用的插入排序算法，时间复杂度是否还能降低？
- 参考文献

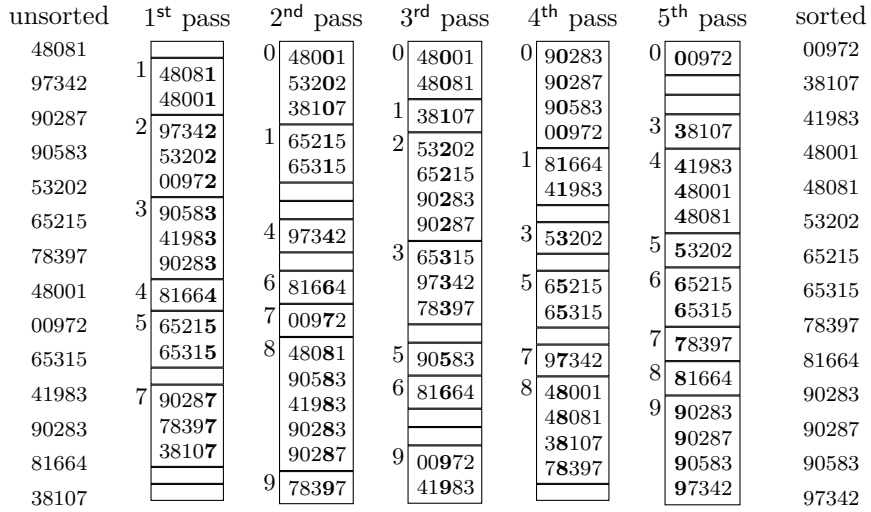
[1] Robert Sedgewick. *Analysis of Shellsort and Related Algorithms*. ESA '96: Fourth Annual European Symposium on Algorithms, 1996

3.2 基数排序

桶排序

- 前面介绍的排序算法基于这样的假设：基本操作只是元素之间的比较
- 如果改变基本操作或增加其他前提假设，是否会对降低时间复杂度有所帮助？
- **桶排序** (Bucket Sorts)：
  - 假设已知待排序元素的某些结构特性使其可以等可能的被安排在等间隔的区间内
  - 等间隔的区间称为桶，每个桶内存放该区间的元素
  - 算法的基本思想是：(1) 分配待排序元素到各个桶；(2) 对每个桶内分别进行排序；(3) 将排序结果组合到一起
  - 假定元素分配尽可能平均，每个桶内排序基于比较操作，则桶排序的时间复杂度可以达到 $O(n \log(n/k))$，其中 k 为桶的个数，若取 $k = n/c$，复杂度为 $O(n \log c) = O(n)$
- **基数排序** (Radix Sort)：一种复杂度能达到 $O(n)$ 的桶排序算法

基数排序 前提：只针对整数进行



算法描述

- 用链表存储待排序元素和桶元素

Algorithm `RadixSort`(List L, int $radix$, int $numFields$)

```
1 List[] buckets ← new List[radix];
2 List newL ← L;
3 for field ← 0 to numFields do
4 |   初始化 buckets 中元素为空链表;
5 |   Distribute(newL, buckets, radix, field);
6 |   newL ← Combine(buckets, radix);
7 end
8 return newL;
```

`Distribute`

Procedure `Distribute`(List L, List[] $buckets$, $radix$, $field$)

```
1 List remL ← L;
2 while remL ≠ null do
3 |   K ← First(remL);                          /* 取表头元素 */
4 |   b ← MaskShift(field, radix, K);           /* 取当前位数字 */
5 |   buckets[b] ← Cons(K, buckets[b]);         /* 插入到相应桶 */
6 |   remL ← Rest(remL);                        /* 取表尾 */
7 end
```

`Combine`

```
Procedure Combine(List[] buckets, int radix)
for b ← radix − 1 down to 0 do
    remBucket ← buckets[b];
    while remBucket ≠null do
        K ← First(remBucket);
        L ← Cons(K, L);
        remBucket ← Rest(remBucket);
    end
end
return L;
```

复杂度分析

- 时间复杂度：
  - Distribute: $\Theta(n)$
  - Combine: $\Theta(n)$
  - RadixSort:总共调用 $numFields$ 次 Distribute 和 Combine,一般情况下 $numFields$ 为常数，因此 RadixSort 复杂度为 $\Theta(n)$
- 空间复杂度：
  - 额外空间主要用在存储元素的链表结构上，所需空间为 $\Theta(n)$

4 排序算法比较

排序算法的稳定性

- **排序算法的稳定性**：若待排序的序列中，存在多个具有相同键值的记录，经过排序，这些记录的相对次序保持不变，则称该算法是稳定的；若经排序后，记录的相对次序发生了改变，则称该算法是不稳定的。
- **稳定性的好处**：排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。
- **何时需要稳定的排序算法**：不希望改变具有相同键值的记录原有的顺序。例如：对学生按成绩排序，对于成绩相同的学生，希望保持原有的学号顺序

排序算法比较

跟冒泡排序一样

算法	最坏情况	平均情况	空间占用	稳定性
插入排序	$n^2/2$	$n^2/4$	$\Theta(1)$	是
选择排序	$n^2/2$	$n^2/2$	$\Theta(1)$	否
快速排序	$n^2/2$	$1.386n\lg n$	$\Theta(\log n)$	否
归并排序	$n\lg n$	$n\lg n$	$\Theta(n)$	是
堆排序	$2n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
快速堆排序	$n\lg n$	$\Theta(n\log n)$	$\Theta(1)$	否
希尔排序	$\Theta(n\log^2 n)$	?	$\Theta(1)$	否
基数排序	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	是

选择排序不稳定：如4 3 4 2 8 7，第一次找最小的即2，然后和第一个4换，这样第二个4就变成了在前了

有点像归并排序

可以通过改进变为nlgn

对于需要稳定的算法，则归并排序最好；非稳定则快速排序最好

算法概论

第四讲：选择和检索

薛健

Last Modified: 2018.12.15

主要内容

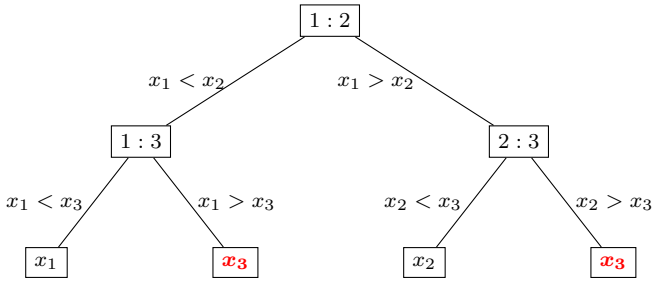
1 选择算法

1.1 选择问题

选择问题

- **选择问题** (Selection Problem)：在一个有 n 个元素的序列中找到第 k 小的元素 ($1 \leq k \leq n$)，即：如果该序列按从小到大排序，则第 k 个元素就是所需要的结果
- 基本操作：比较。
  - 以比较为基本操作的排序问题复杂度下界是 $\Theta(n \log n)$ 先将数组排好序，然后找到第k个数
  - 选择问题的复杂度下界是 $\Theta(n \log n)$ 或更低？
  - 决策树分析？ 2-tree 且n0=n2+1,所以至少有2n-1个结点，高度为h>=[lg(2n-1+1)]-1=[lgn+1]-1=[lgn] 这里[]表示向上取整
- 常见选择问题：
  - 选择最大元素 (the **maximum**) 或最小元素 (the **minimum**) 最坏情况n-1次
  - 同时选择最大和最小元素
  - 选择中间元素 (中位数，the **median**) ($k = \lceil n/2 \rceil$)

用决策树分析：选择问题决策树至少有 n 个叶结点，决策树高度至少为 $\lceil \lg n \rceil$，但这个下界不够准确。我们已经知道找最大元素至少需要 $n-1$ 次比较 (反证法证明)，哪里出问题了呢？在选择问题的决策树中，某些输出将出现在超过一个叶子结点上，如 $n = 3$ 选择最小元素的决策树：



对手论证法

- **猜数字的游戏**：心中选定一个数字让对方猜，对方可按如下方式提问：“这个数字比 * 大 (小) 吗?”，可提问多次，每次提问后，必须给出回应：“是”或“否”
  - 如何让对方在尽可能多次的提问后仍然猜不到所选定的数字？
  - 换一种思维方式：在最开始并不真正选定具体数字，而是随对手的问题不断调整 (*cheating*?)
  - 必须保证最后结果与对手提出的所有问题都不矛盾

- **对手论证法** (Adversary Arguments)：
  - 对于解决某类问题的算法，有一个假想的对手在千方百计阻挠其得到最终结果
  - 在算法作出每一次决策后，对手总给出不利于算法的结果
  - 对于对手的唯一约束是：他所给出的所有决策结果必须是自洽的
  - 该方法可以用来分析选择问题的复杂度下界

1.2 查找最大和最小元素

查找最大和最小元素

- 如果我们找最大元素后再找最小元素，比较次数为 $(n-1)+(n-2)=2n-3$，这显然不是最佳算法，在找最大元素时有一些结果可以被后面找最小元素的过程所共享
- 一个更快的算法：
  1. 比较每两个元素的大小 ($n/2$ 次比较)
  2. 在得到的 $n/2$ 个较大元素中找出最大元素 ($n/2-1$ 次比较)
  3. 在得到的 $n/2$ 个较小元素中找出最小元素 ($n/2-1$ 次比较)
- 加上对 n 是奇数的考虑，总共的比较次数为 $\lceil 3n/2 \rceil - 2$ 次比较

Theorem 1.1. *任何在 n 个元素中查找最大和最小元素的算法在最坏情况下至少需要 $3n/2-2$ 次比较操作.*

复杂度下界分析

- 一些约定：
  - 将比较看作竞赛：$x > y$ 的结果为：x 赢 y 输
  - 元素状态：

W	至少赢了一次且从未输过
L	至少输了一次且从未赢过
WL	输赢各至少一次
N	还未参与比较

- 查找最大和最小元素的对手策略：

比较前状态	对手回应	新状态	有效信息数
N, N	$x > y$	W, L	2
W, N 或 WL, N	$x > y$	W, L 或 WL, L	1
L, N	$x < y$	L, W	1
W, W	$x > y$	W, WL	1
L, L	$x > y$	WL, L	1
W, L 或 WL, L 或 W, WL	$x > y$	无改变	0
WL, WL	实际情况	无改变	0

对手回应的策略是使得有效信息最小；所以会尽量保持原来状态

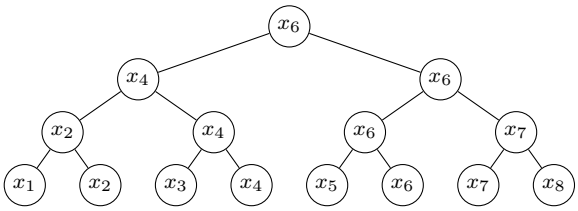
复杂度下界分析 (cont.)

- 要找到最大和最小元素，需要知道除最大元素以外的每个元素至少输掉了某次比较，而除最小元素以外的每个元素至少赢得了某次比较，即：至少需要 $2n-2$ 条有效信息才能作出最终决定
- 从算法的角度讲，每次比较最多能从对手那里得到 2 条有效信息，而这样的比较最多能进行 $n/2$ 次，共得到 n 条有效信息
- 对于余下的 $n-2$ 条信息，由于每次比较最多只能得到 1 条有效信息，所以至少需要 $n-2$ 次比较
- 因此，要得到最终结果，至少需要 $n/2 + n - 2 = 3n/2 - 2$ 次比较

1.3 查找第二大元素

查找第二大元素

- 如果用找最大元素的方法找第二大元素，则需要 $(n-1) + (n-2) = 2n - 3$ 次比较
- 更快的方法：**联赛方法** (tournament method)
  - 每两个元素进行比较，其赢家进入下一轮，直到决出最终的胜利者
  - 若元素数为奇数，则剩余一个元素直接进入下一轮
  - 按照上述规则，可建立一棵二叉树，其根结点为最大元素，比较次数为 $n-1$，而只有跟最大元素直接比较过的元素才有可能是第二大



  - 与最大元素直接比较的路径上的其他元素中最大的那个即为第二大元素，比较次数 $\lceil \lg n \rceil - 1$，总的比较次数为 $\boldsymbol{n + \lceil \lg n \rceil - 2}$

复杂度下界分析

- 我们已经知道：第二大元素肯定在比较至少输过一次的元素中，而要区分出所有的输家，至少需要 $n-1$ 次比较
- 如果最大元素参与了其中 $\lceil \lg n \rceil$ 次比较，则为了得到第二大元素，其中 $\lceil \lg n \rceil - 1$ 个元素将再次在与第二大元素的比较中落败，所以至少需要 $n + \lceil \lg n \rceil - 2$ 次比较
- 因此，我们只要找到一种对手策略能够迫使算法将最大元素至少与 $\lceil \lg n \rceil$ 个不同元素进行比较：

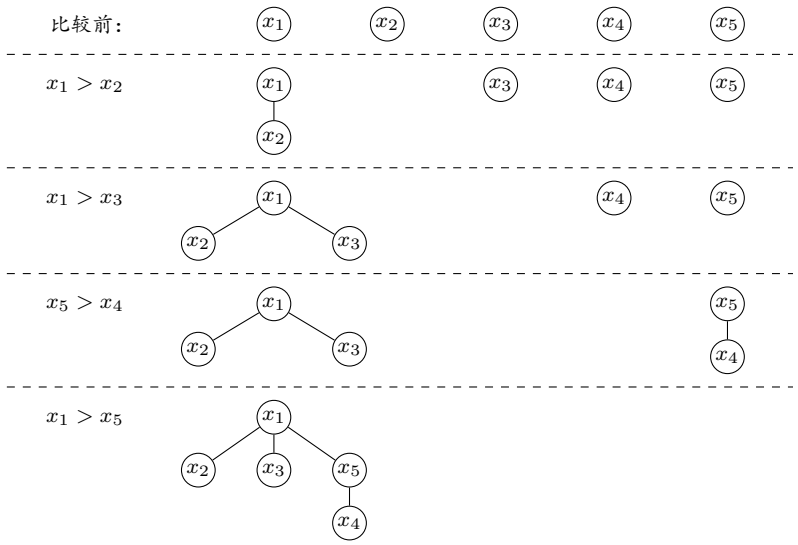
– 为每个元素指定一个权值 $w(x)$，初始化为 1，当算法比较 x 和 y 时根据对手回应调整权值

比较前权值	对手回应	更新权值
$w(x) > w(y)$	$x > y$	$w(x) \leftarrow \text{prior}(w(x)+w(y)); w(y) \leftarrow 0$
$w(x) = w(y) > 0$	$x > y$	$w(x) \leftarrow \text{prior}(w(x)+w(y)); w(y) \leftarrow 0$
$w(x) < w(y)$	$x < y$	$w(y) \leftarrow \text{prior}(w(x)+w(y)); w(x) \leftarrow 0$
$w(x) = w(y) = 0$	实际情况	无改变

对手回应也是大的越大，尽量不改状态

这里由于wx=wy，所以对手回应也可以为y<x

例子



比较元素对	权值	胜者	更新后权值	元素键值
x_1, x_2	$w(x_1) = w(x_2)$	x_1	$2,0,1,1,1$	$20,10,*,*,*$
x_1, x_3	$w(x_1) > w(x_3)$	x_1	$3,0,0,1,1$	$20,10,15,*,*$
x_5, x_4	$w(x_5) = w(x_4)$	x_5	$3,0,0,0,2$	$20,10,15,30,40$
x_1, x_5	$w(x_1) > w(x_5)$	x_1	$5,0,0,0,0$	$41,10,15,30,40$

复杂度下界分析

- 上述对手策略是否满足要求？
  1. 一个元素输掉一次比较当且仅当其权值为 0
  2. 在前三种情况中，被选为胜者的元素有非 0 权值，这表示该元素在前面的比较中还未输过，因此可以给他赋予任意大的键值从而保证与前面作出的回应不产生矛盾
  3. 由权值的更新方式可知所有元素的权值和始终等于 n
  4. 当算法停止时，仅有一个元素可有非 0 权值，否则将至少有两个元素从未输过一次比较，这时对手可以合理选择这两个元素的键值使算法所找出的第二大元素不正确
- 假设当算法终止时，x 是唯一一个键值非 0 的元素，则根据上述结论有 $x = \max, w(x) = n$
- 设 x 第 k 次赢得比较后权值 $w(x) = w_k$，则根据权值更新规则有 $w_k \leq 2w_{k-1}$
- 设 K 为 x 赢得的比较次数，则有 $n = w_K \leq 2^K w_0 = 2^K \Rightarrow K \geq \lg n \Rightarrow \boldsymbol{K \geq \lceil \lg n \rceil}$

1.4 中位数问题

查找中位数

- **中位数问题**：查找包含 n 个元素序列的中位数 (median，即排序后处于 $\lceil n/2 \rceil$ 位置处的元素)
- 分治法 (`QuickSort` 的分治法策略)：
  - 将元素分为两组，使其中一组中的元素小于另一组中的元素
  - 在包含元素较多的一组中继续查找　　则变成了在元素多的那一组找第[n/2]-基准数位置。
- 问题：递归调用不再是查找中位数 (原始序列的中位数并不也是子序列的中位数)
- 推广 $\Rightarrow$ 一般**选择问题** (Selection Problem)：查找序列中第 k 小的元素 (即从小到大排序后处于第 k 个位置处的元素)
- 先排序后选择 ($O(n \log n)$)？
- 有更好的算法 ($O(n)$)！

算法描述

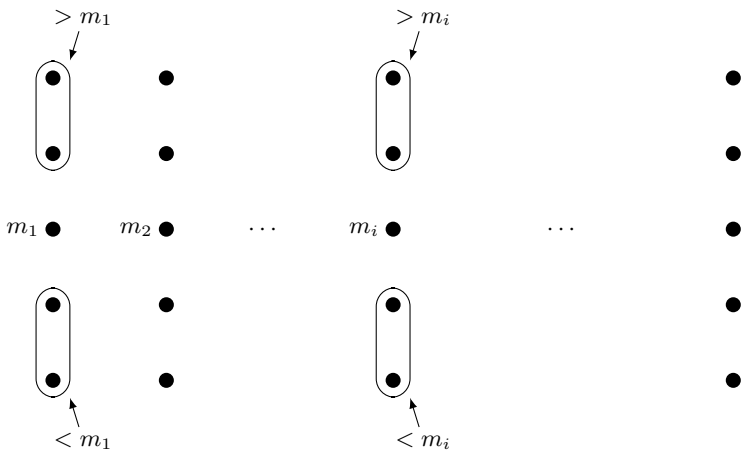
Algorithm `Select`(Set S, int k)

1 if $|S| \leq 5$ then return S 中第 k 小元素的直接查找结果;
2 将原集合元素 5 个一组分成 $|S|/5$ 组，取每组的中位数放入集合 M 中;
3 $m^* \leftarrow$ `Select`$(M, \lceil |M|/2 \rceil)$;
4 根据 m^* 将 S 分成 4 个子集：

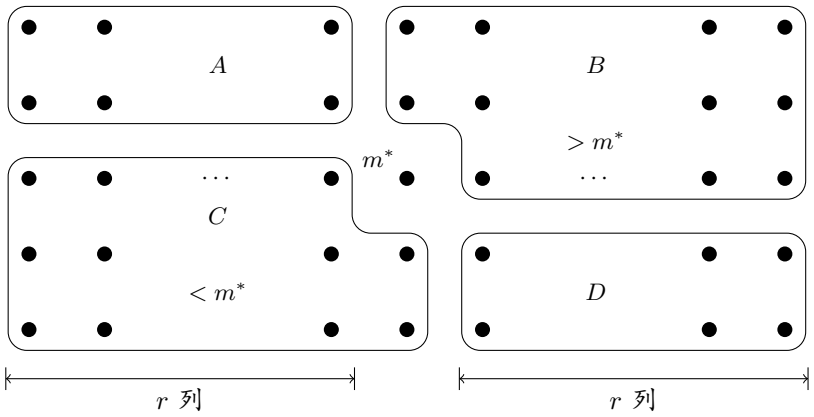
A　中位数小于 m^* 的 5 元组中大于各自中位数的元素
B　中位数大于 m^* 的 5 元组中大于等于各自中位数的元素 (都大于 m^*)
C　中位数小于 m^* 的 5 元组中小于等于各自中位数的元素 (都小于 m^*)
D　中位数大于 m^* 的 5 元组中小于各自中位数的元素

5 将 A 和 D 中的元素与 m^* 比较，其中小于 m^* 的元素与 C 合并成 S_1，大于 m^* 的元素与 B 合并成 S_2;
6 if $k = |S_1| + 1$ then return m^*;
7 else if $k \leq |S_1|$ then return `Select`(S_1, k);　　表示S1中找第k小，即中位数再S1中的第k位置
8 else return `Select`$(S_2, k - |S_1| - 1)$;　中位数在S2中的第k-|S1|-1位置

分组求中位数



求中位数的中位数



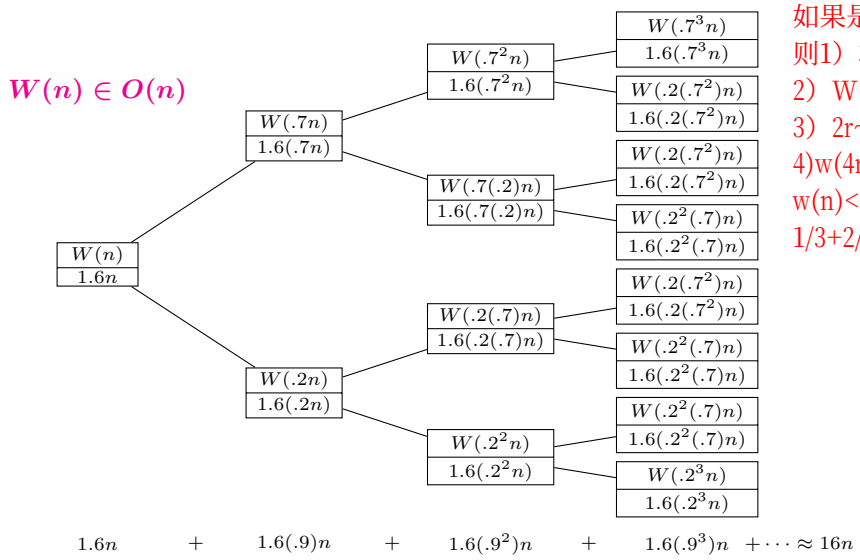
选择算法复杂度分析

- 为便于分析，假设 $n = 5(2r+1)$，忽略递归调用不满足该式所造成的影响，则可逐步分析上述算法如下：
  1. 找所有 5 元组的中位数：$6(n/5)$ 次比较 ($Why?$) 5个一组，在5个元素中最多6次比较可以找到中位数
  2. 递归查找中位数集合的中位数：$W(n/5)$ 次比较
  3. 子集 A、D 中的元素与 m^* 比较：$4r$ 次比较
  4. 递归调用 `Select`，在最坏情况下，A、D 中的元素都大于或都小于 m^*：$W(7r+2)$ 次比较 最坏情况即A,D中的元素都归在一个集合B或C中
- 由 $n = 5(2r+1)$ 可知 $r \approx 0.1n$，所以有：

$$\begin{aligned} W(n) &\le 1.2n + W(0.2n) + 0.4n + W(0.7n) \\ &= 1.6n + W(0.2n) + W(0.7n) \end{aligned}$$

- 主定理不适用，用递归树求解

选择算法递归树



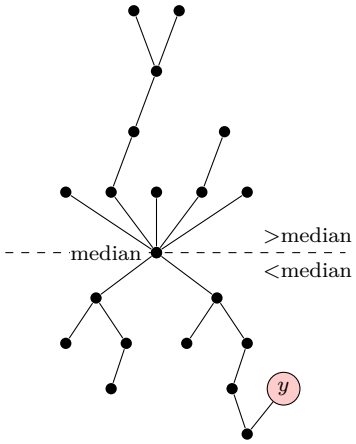
$W(n) \in O(n)$

$1.6n + 1.6(.9)n + 1.6(.9^2)n + 1.6(.9^3)n + \cdots \approx 16n$

如果是三个一组：
则1）3*n/3表示三个数比较要3次，共n/3组
2）W(n/3)次中位数比较
3）2r~n/3
4)w(4r+1)
w(n)<=n+n/3+w(n/3)+w(2n/3)
1/3+2/3=1 ===>W(n)~O(nlgn)

中位数问题的复杂度下界

- 既然是复杂度下界分析，不妨设集合中元素各异，元素个数为奇数
- 一个算法要想找到中位数，必须知道其他每个元素和中位数的大小关系；否则，假设某元素 y 与中位数大小关系未知，则在对手论证法中，算法对手可以改变元素 y 的键值，使其落在中位数分界线的另一边，使得算法所选的中位数不再是真正的中位数
- 根据上述分析，一个算法要想找到正确的中位数，至少需要 $\boldsymbol{n-1}$ 次比较
- $n-1$ 这个下界是否足够准确？



中位数问题的复杂度下界 (cont.)

即通过x与y的比较可以知道x与中位数的大小关系

- **关键比较**：元素 x 与 $y \geq$ median 的第一次比较，且结果为 $x > y$；或与 $y \leq$ median 的第一次比较，且结果为 $x < y$。从关键比较可以知道 x 与中位数的大小关系
- 可以设计一种对手策略，迫使算法进行非关键比较：
  - 开始比较前，选定一个键值作为中位数键值，但不指定具体元素作为中位数
  - 每次比较给首次参与比较的元素赋值，尽量使元素位于中位数两侧
  - 约束条件：比中位数大的元素总数不得超过 $(n-1)/2$；同样，比中位数小的元素总数也不得超过 $(n-1)/2$
  - 元素状态：L —被赋予比中位数大的值；S —被赋予比中位数小的值；N —未参与比较
  - 元素赋值规则：

比较前状态	元素赋值规则	比较后状态
N, N	一个大于中位数，一个小于中位数	L, S
L, N 或 N, L	未参与比较的元素赋予小于中位数的值	L, S
S, N 或 N, S	未参与比较的元素赋予大于中位数的值	L, S

都是非关键比较

中位数问题的复杂度下界 (cont.)

- 对手策略 (续)
  - 赋值规则表中的比较都是非关键比较
  - 当大于中位数的元素总数或小于中位数的元素总数达到 $(n-1)/2$ 后，上述元素赋值规则不再适用，这时对手策略将按照实际情况赋值，直到剩下一个元素，该元素即为中位数

- 显然，上述对手策略可迫使任何查找中位数的算法至少执行 $(n-1)/2$ 次非关键比较，而关键比较次数至少为 $n-1$，因此，我们可以得出结论：

  Theorem 1.2. 任何以比较为关键操作选取 n *(*n 为奇数*)* 个元素中的中位数的算法至少需要 $3n/2-3/2$ 次比较

- $3n/2-3/2$ 是最好的下界吗？

关于中位数问题更深入的探讨

- 中位数选择算法：
  - [Manuel Blum, et al., 1973] 提出了比较次数为 $5.43n$ 的算法
  - [A. Schönhage, et al., 1976] 提出了比较次数为 $3n+o(n)$ 的算法
  - [Dorit Dor and Uri Zwick, 1999] 提出了比较次数为 $2.95n+o(n)$ 的算法
- 中位数问题的复杂度下界：
  - [Vaughan R. Pratt and Foong Frances Yao, 1973]：$1.75n-\log n$
  - [C. K. Yap, 1976]：$\frac{38}{21}n+O(1)\approx 1.81n$ 和 $\frac{79}{43}n+O(1)\approx 1.83n$
  - [Samuel W. Bent and John W. John, 1985]：$2n+o(n)$
  - [Dorit Dor, et al., 1996]：$(2+\epsilon)n+o(n)$
  - [Dorit Dor, et al., 2001]：$2.01227n+o(n)$
- 参考文献：

[1] Dorit Dor, Johan Håstad, Staffan Ulfberg and Uri Zwick. *On Lower Bounds for Selecting the Median.* SIAM journal on discrete mathematics, 14(3), pp.299–311, 2001.

利用对手论证法来设计算法

- 前面利用对手论证法来分析问题的复杂度下界
  - 在对算法每一步操作的回应中尽可能少地透露有用信息
- 同样可以利用对手论证法来设计算法，使复杂度尽可能低：
  - 每次基本操作尽量使对手回答“是”或“否”所透露的信息一样多 (某种平衡)
  - 决策树 $\Longrightarrow$ 平衡二叉树 (高度尽可能低)
  - 归并排序、查找最小和最大元素、查找第二大元素等算法都很好地运用了这样的策略

Exercise (5). 试设计比较策略：用 6 次比较在 5 个元素中找到中位数；用 7 次比较完成 5 个元素的排序. *deadline: 2018.12.15*

2 动态集合和搜索

2.1 基本概念

动态集合

- **动态集合** (Dynamic Sets)：元素 (值、结构、个数等) 随算法运行而不断改变的集合，例如：
  - 初始状态为空集，在后续运算过程中不断有新的元素添加到集合中，并且最终集合中包含多少元素并不确定

  - 初始状态为一个大的集合，在运算过程中不断从中删除元素 (通常以空集作为算法停止的状态)
  - 或者在运算过程中同时添加和删除元素

- 最常用也是最基本的集合动态增长技术是**数组加倍** (Array Doubling)：在往集合中添加元素时，若用于存储的数组空间不足，则成倍增加数组大小 (例：STL 中的 vector 类)

Procedure `ArrayDouble`(Set s)

1 $newLength \leftarrow 2 * s.elements.length$;
2 $newElements \leftarrow$ new Object[$newLength$];
3 将 $s.elements$ 中的元素拷贝到 $newElements$ 中;
4 $s.elements \leftarrow newElements$;

分摊时间分析

- 上面提到的 `ArrayDouble` 乍一看非常耗时，复杂度达到 $\Theta(n)$，而其最终要完成的操作仅仅是往集合中插入一个新的元素；但从集合插入 n 个元素的总执行时间来看，其复杂度并不高，也是 $\Theta(n)$
- 这样的效果实际上可以看作数组加倍的额外开销被分摊到了整个操作序列的每个插入操作上，从而使整体的时间复杂度并未提高
- 动态集合上基本操作时间开销的不确定性给相关算法的分析和设计带来了一定的难度
- **分摊时间分析** (Amortized Time Analysis)：分析动态数据结构基本操作执行时间的方法

$$\underset{amortized\ cost}{\text{分摊开销}} = \underset{actual\ cost}{\text{实际开销}} + \underset{accounting\ cost}{\text{账户开销}}$$

- 其关键是构造账户开销，使其达到以下目标：
  1. 从动态集合创建起，任意合法的操作序列，其账户开销总和非负
  2. 不管实际开销在单个操作之间波动多大，每个操作的分摊开销应基本保持一致

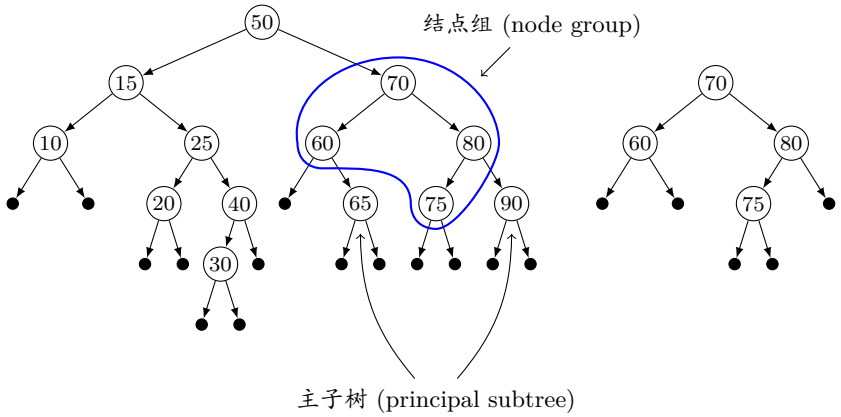
例子

- 在 Stack 数据结构中用数组加倍的方式扩展存储空间
- pop 操作的实际开销为 1；push 操作的实际开销：
  - 不需要扩展数组空间时为 1
  - 需要扩展数组空间时为 $1 + nt \in \Theta(n)$ (t 是拷贝一个元素的开销)
- 构造账户开销：
  1. 不需要数组加倍时 push 操作的账户开销为 $2t$
  2. 数组空间从 n 加倍到 $2n$ 时 push 操作的账户开销为 $-nt + 2t$
  3. pop 操作的账户开销为 0
- Stack 空间加倍的操作发生在元素数达到 $N, 2N, 4N, 8N, \cdots$ 时
  - 当达到 N 时，账户中储存的开销达到 $2Nt$，由于数组加倍而降到 $Nt + 2t$
  - 当达到 $2N$ 时，储存开销达到 $3Nt$，数组加倍又使其降到 $Nt + 2t$
  - 以此类推，当储存的开销达到 $Nt + 2t$ 后便不再低于它，符合账户开销总和非负的要求
- push 操作的分摊开销为 $1 + 2t \in \Theta(1)$

2.2 红黑树

扩展二叉树

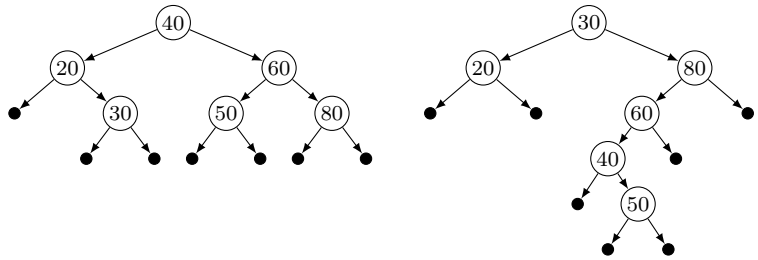
将空子树作为外结点补充到二叉树中，使其成为 2-tree



二叉搜索树

Definition 2.1 (二叉搜索树 (Binary Search Tree))**.** 二叉搜索树是满足如下特性的二叉树：每个结点元素的键值大于其左子树的所有结点元素而小于等于其右子树的所有结点元素

- 例：



- 用中序 (inorder) 遍历二叉搜索树可得到从小到大排序的元素序列
- 包含相同元素的二叉搜索树平衡度 (degrees of balance) 可以差别很大

搜索算法

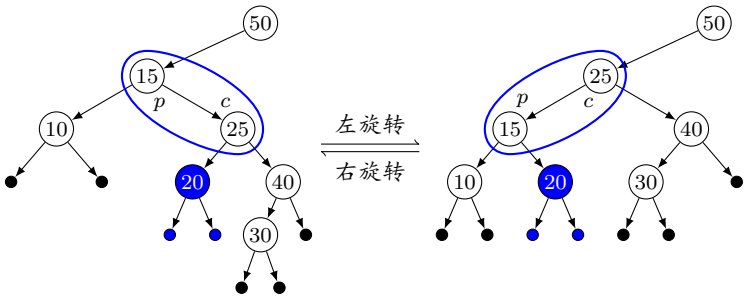
Algorithm `BSTSearch`(BinaryTree bst, Element K)

```
1 Element r ← null;
2 if bst ≠ null then
3 |   Element root ← Root(bst);
4 |   if K = root then  r ← root ;
5 |   else if K < root then  r ← BSTSearch(LeftSubtree(bst), K) ;
6 |   else  r ← BSTSearch(RightSubtree(bst), K) ;
7 end
8 return r;
```

- 比较次数与二叉树高度相当
- 如果二叉搜索树可以有任意结构，则最坏情况搜索时间复杂度为 $\Theta(n)$
- 如果二叉搜索树尽可能达到平衡，则最坏情况复杂度可降到 $\lg n \in \Theta(\log n)$

AVL树是最平衡的树，左右子树高度差小于或等于1，AVL是1962年提出，三个人名的简写

二叉树的旋转操作



- 旋转操作在两个相邻结点之间进行 (父结点 p 和其子结点 c)，涉及 3 棵主子树，如图所示左旋转过程可描述为：
  1. 改变 p 和 c 之间边的方向，p 成为 c 的左子结点
  2. 原来 p 的父结点成为 c 的父结点
  3. 原来 c 的左子树 (即中间那棵主子树) 变为 p 的右子树
- 左、右旋转互为反变换

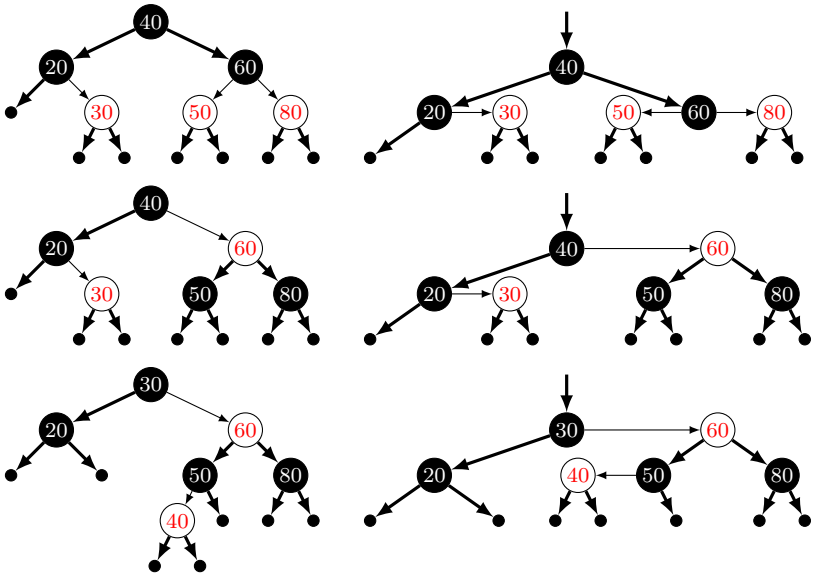
红黑树定义

Definition 2.2 (红黑树 (Red-Black Tree，RB Tree))**.** 将一棵二叉树的每个结点涂成红色或黑色，终点是黑色结点的边称为**黑边** (black edge)，路径的**黑色长度** (black length)定义为该路径上的黑边数目，结点的**黑色深度** (black depth)定义为从根结点到该结点路径的黑色长度，从给定结点到某一外结点路径称为该结点的**外路径** (external path)，则一棵二叉树是红黑树，当且仅当：

1. 任何红色结点无红色孩子结点
2. 给定结点 u，其所有外路径的黑色长度均相等，其值称为结点 u 的**黑色高度** (black height)
3. 根结点和所有外结点是黑色结点

根结点为红色且满足上述条件的二叉树称为**近似红黑树** (almost-red-black tree, ARB tree).

例子



红黑树递归定义

Definition 2.3 (RB_h 树和 ARB_h 树)**.** 顶点着色为红或黑且外结点为黑的二叉树按如下定义构成 RB_h 树或 ARB_h 树：

1. 外结点是 RB_0 树
2. 对 $h \geq 1$，根结点为红色且左、右子树是 RB_{h-1} 树的二叉树是 ARB_h 树
3. 对 $h \geq 1$，根结点为黑色且左、右子树是 RB_{h-1} 树或 ARB_h 树的二叉树是 RB_h 树

Lemma 2.4. RB_h 树和 ARB_h 树的黑色高度为 h

一些结论

Lemma 2.5. 若 T 是一棵 RB_h 树 (黑色高度为 h 的红黑树)，则有：

1. T 至少有 $2^h - 1$ 个黑色内结点
2. T 至多有 $4^h - 1$ 个内结点
3. 任一黑色结点的深度至多是其黑色深度的 2 倍

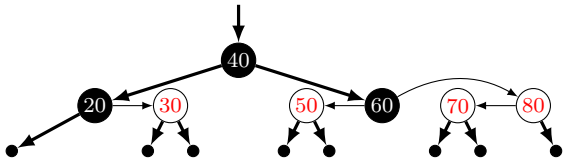
若 A 是一棵 ARB_h 树 (黑色高度为 h 的近似红黑树)，则有：

1. A 至少有 $2^h - 2$ 个黑色内结点
2. A 至多有 $\frac{1}{2}(4^h) - 1$ 个内结点
3. 任一黑色结点的深度至多是其黑色深度的 2 倍

Theorem 2.6. 若 T 是一棵包含 n 个内结点的红黑树，则其任一结点深度均不大于 $2\lg(n+1)$，即 T 的高度至多为 $2\lg(n+1)$

红黑树结点插入操作

- 红黑树的定义给出了关于结点颜色和黑色高度的约束条件，若要插入一个结点，则必须保证结点插入后的二叉树仍然能满足红黑树的约束条件
- 插入过程分为两个阶段：
  1. 首先看待插入元素 K 是否已经包含在红黑树中，即在红黑树所代表的二叉搜索树中查找元素 K，直到到达一个外结点 (空树)，然后将该外结点替换成一棵仅包含一个内结点 K 的子树
  2. 修正任何违反颜色约束的地方 (在插入操作中，任何时候都不会出现违反黑色高度约束的地方)
- 例：插入的第一阶段产生的违反颜色约束的情况



修正操作

Definition 2.7.

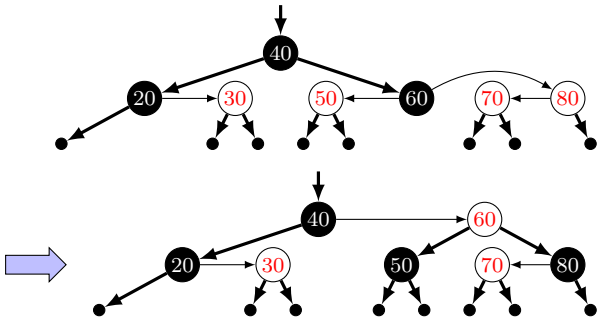
- **结点组** (cluster)：包含一个黑色内结点以及从该结点仅通过非黑色边可以到达的红色结点所组成的内结点集合，唯一的黑色结点称为该结点组的根
- **关键组** (critical cluster)：若结点组中存在这样的结点：从结点组的根到该结点的路径长度大于 1，则该结点组称为关键组（存在违反颜色约束的情况）

一些结论：

- 二叉树存在违反红黑树颜色约束的情况当且仅当其结点组中存在关键组
- 在插入操作过程中出现的关键组可能包含 3 个或 4 个结点
- 插入操作的第一阶段最多产生一个关键组
- 插入操作的第二阶段 (rebalancing) 采取某种策略消除产生的关键组，若不再产生新的关键组，则插入结束；否则最多在更接近根的位置产生一个新的关键组，修正过程将继续进行，直到无关键组产生

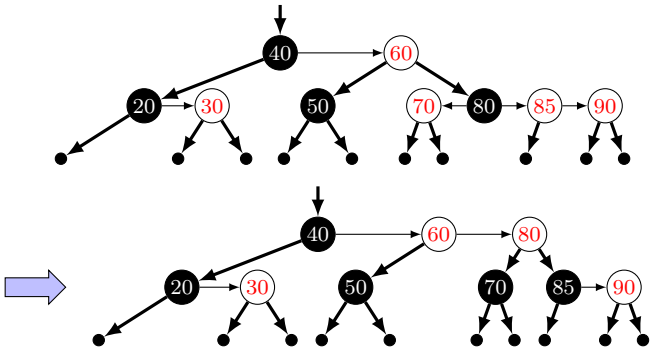
4 结点关键组消除：Color Flip

- 关键组根结点改成红色结点
- 根结点的左右子结点改成黑色结点
- 如果关键组根结点恰好也是整棵树的根结点，则将其改回黑色，只有在这种情况下整棵树的黑色高度被改变
- 例 1：不产生新的关键组

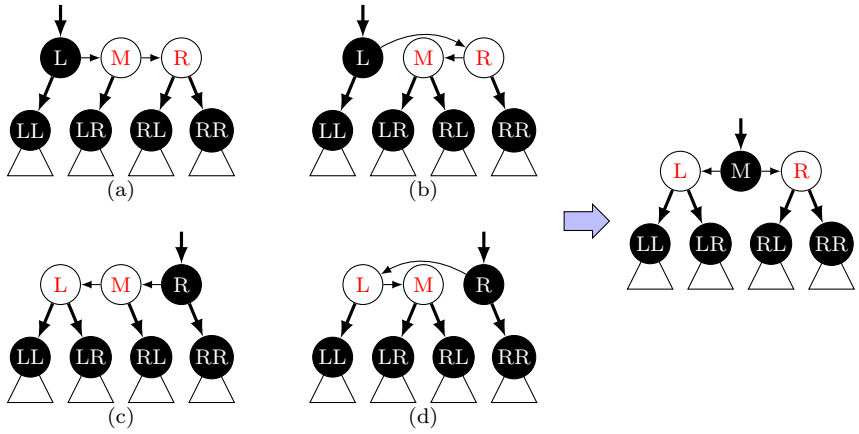


4 结点关键组消除：Color Flip (cont.)

- 例 2：有新的关键组产生



3 结点关键组消除



红黑树插入算法设计

- 返回值结构：

RBTree	**Element** root **RBTree** left **RBTree** right **int** color
InsReturn	**RBTree** newTree **int** status

- 常量定义：

color	red	根结点为红色
	black	根结点为黑色
status	ok	插入结束
	rbr	根结点为黑色，左右子结点为红色
	brb	根结点为红色，左右子结点为黑色
	rrb	根结点和左子结点为红色
	brr	根结点和右子结点为红色

插入算法

Algorithm `RBTIns`(RBTree $oldTree$, Element $newNode$)

```
1  InsReturn ans, ansLeft, andRight;
2  if oldTree = null then
3  |  ans.newTree ← 根结点是 newNode 的单结点 RBTree;
4  |  ans.status ← brb;
5  else
6  |  if newNode < oldTree.root then
7  |  |  ansLeft ← RBTIns(oldTree.left, newNode);
8  |  |  ans ← RepairLeft(oldTree, ansLeft);
9  |  else
10 |  |  ansRight ← RBTIns(oldTree.right, newNode);
11 |  |  ans ← RepairRight(oldTree, ansRight);
12 |  end
13 end
14 return ans;
```

左子树修正：

Algorithm `RepairLeft`(RBTree $oldTree$, InsReturn $ansLeft$)

```
1  InsReturn ans;
2  if ansLeft.status = ok then
3  |   ans.newTree ← oldTree;
4  |   ans.status ← ok;
5  else
6  |   oldTree.left ← ansLeft.newTree;
7  |   if ansLeft.status = rbr then                          /* 无需再修正 */
8  |   |   ans.newTree ← oldTree;
9  |   |   ans.status ← ok;
10 |   else if ansLeft.status = brb then     /* 左子树没问题，检查根结点颜色 */
11 |   |   if oldTree.color = black then
12 |   |   |   ans.status ← ok;
13 |   |   else  ans.status ← rrb ;
14 |   |   ans.newTree ← oldTree;
15 |   else if oldTree.right.color = red then              /* 4 结点关键组 */
16 |   |   ColorFlip(oldTree);
17 |   |   ans.newTree ← oldTree;
18 |   |   ans.status ← brb;
19 |   else                                                /* 3 结点关键组 */
20 |   |   ans.newTree ← RebalLeft(oldTree, ansLeft.status);
21 |   |   ans.status ← ok;
22 |   end
23 end
24 return ans;
```

3 结点关键组消除：

Algorithm `RebalLeft`(RBTree $oldTree$, int $leftStatus$)

```
1  RBTree L, M, R, LR, RL;
2  if leftStatus = rrb then                                 /* 情况 (c) */
3  |   R ← oldTree;
4  |   M ← oldTree.left;
5  |   L ← M.left;
6  |   RL ← M.right;
7  |   R.left ← RL;
8  |   M.right ← R;
9  else                                          /* leftStatus = brr, 情况 (d) */
10 |   R ← oldTree;
11 |   L ← oldTree.left;
12 |   M ← L.right;
13 |   LR ← M.left;
14 |   RL ← M.right;
15 |   R.left ← RL;
16 |   L.right ← LR;
17 |   M.right ← R;
18 |   M.left ← L;
19 end
20 L.color ← red;
21 R.color ← red;
22 M.color ← black;
23 return M;
```

4 结点关键组消除：

Algorithm `ColorFlip`(RBTree $oldTree$)

```
1 oldTree.color ← red;
2 oldTree.left.color ← black;
3 oldTree.right.color ← black;
```

插入算法 (cont.)

- 由于最终的修正结果可能使根结点成为红色，仅当此时需要再次修正，为了不破坏递归调用的一致性，需要提供一个封装过程来处理这种情况：

Procedure `RBTInsert`(RBTree $oldTree$, Element $node$)

```
1 InsReturn ans ← RBTIns(oldTree, node);
2 if ans.newTree.color ≠ black then
3 |   ans.newTree.color ← black;
4 end
5 return ans.newTree;
```

算法分析

Lemma 2.8. 如果 `RBTIns` 的参数 $oldRBTree$ 是一棵 RB_h 树或 ARB_{h+1} 树，则返回值中的 newTree 和 status 取如下组合之一：

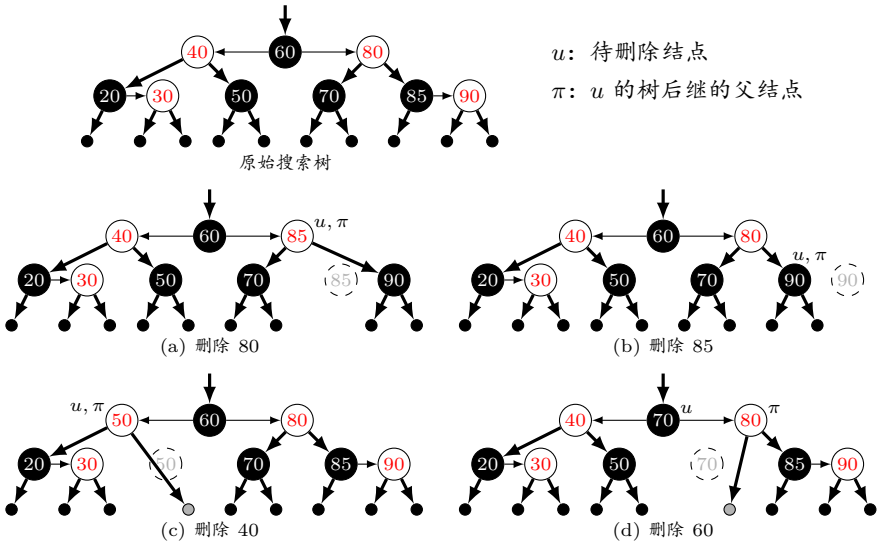
1. status=ok，newTree 是一棵 RB_h 树或 ARB_{h+1} 树
2. status=rbr，newTree 是一棵 RB_h 树
3. status=brb，newTree 是一棵 ARB_{h+1} 树
4. status=rrb，newTree.color=red，newTree.left 是一棵 ARB_{h+1} 树，newTree.right 是一棵 RB_h 树
5. status=brr，newTree.color=red，newTree.right 是一棵 ARB_{h+1} 树，newTree.left 是一棵 RB_h 树

Theorem 2.9. 上述红黑树插入算法能够将新结点正确地插入到红黑树中而不破坏其性质，插入一个结点到包含 n 个结点的红黑树最坏情况时间复杂度为 $\Theta(\log n)$

红黑树结点删除操作

- 从红黑树删除一个结点要比插入一个结点复杂的多
  - 插入总可以在叶结点处进行，而删除可以出现在任意位置
  - 删除操作有可能造成对高度规则的破坏，修正高度的错误比修正颜色的错误来得复杂
- 从二叉搜索树删除一个结点 (不考虑高度规则)
  - 逻辑删除：结点中存储的元素被删除
  - 结构删除：结点被删除
  - 树后继 (tree successor)：2-tree 中任一内结点 u 的树后继是其右子树中最左边的内结点或者其右子树 (如果其右子树的左子树为外结点)
  - 在二叉搜索树中删除一个结点，实际上是用其树后继的元素替换该结点元素 (逻辑删除) 并删除其树后继 (结构删除)
  - 如果其树后继是外结点，则表明该结点元素最大，可以直接对该结点进行结构删除

结点删除实例

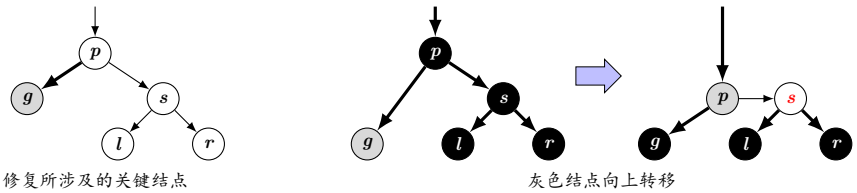


红黑树结点删除

- 红黑树结点删除操作可归纳为以下几个步骤
  1. 搜索定位待删除结点 u
  2. 若 u 的右子结点为外结点，则可直接对 u 进行结构删除
  3. 若 u 的右子结点为内结点，找到 u 的树后继，将其所存储的元素拷贝到 u 中 (u 的颜色暂时不作调整)，并对其树后继结点进行结构删除
  4. 修复所有由于结构删除所造成的对黑色高度约束的破坏
- 其中，步骤 4 是关键，需要详细考察；回顾前面的例子：
  - 在 (a) 中，尽管有黑色结点被结构删除，但它的右子结点是一个内结点，该结点一定是红色 (Why?)，因此可以直接将其改为黑色而不破坏红黑树特性
  - 在 (b) 中，对红色结点进行结构删除不会对红黑树产生任何影响
  - (c) 和 (d) 待删除结点的树后继是黑色结点，并且其右子结点也是黑色结点 (一定是外结点)，因此结构删除此树后继结点后，其右子结点 (灰色) 的黑色高度不足，红黑树将不再保持平衡，需要修正

恢复黑色高度

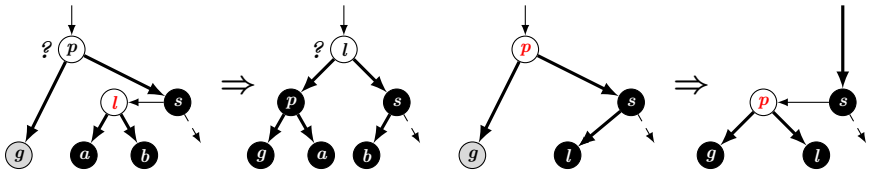
- **灰色结点** (gray node)：一棵 RB_{h-1} 树的根结点 (若标记为黑色)，但其父结点需要一棵 RB_h 子树 (黑色高度少了 1)
- 在修正开始时，灰色结点是一个外结点，随着修正过程进行，灰色状态将向上传递
- 修复黑色高度的基本策略：
  - 寻找附近能够被变成黑色的红色结点，通过局部结构调整恢复高度平衡
  - 如果找不到这样的红色结点，则将结点灰色状态向上传递 (递归处理)，当传递到根结点，则直接将根改回黑色即可



恢复黑色高度 (cont.)

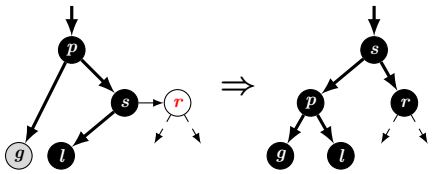
- p, s, l, r 中任一个是红色结点，则灰色结点不需要向上转移，经局部结构调整即可消除黑色高度不平衡状态
- 构造以 p 为根的结点组，使得其所有主子树为 RB_{h-1} 树 (和灰色结点 g 为根的子树一样)，结构调整便局限在这个结点组及其主子树中，每棵主子树均可用一个外结点代表，则修复目标如下：
  1. 若 p 是红色结点，则该结点组应该调整为 RB_1 或 ARB_2 树
  2. 若 p 是黑色结点，则该结点组应该调整为 RB_2 树
- 若灰色结点 g 是左子结点，则按 l, p, r, s 的顺序分为 4 种情况进行处理 (附后)
- 灰色结点 g 是右子结点的情况与左子结点处理完全对称
- 时间复杂度：
  - 子结构调整 $O(1)$
  - 灰色状态转移 $O(\log n)$

子结构调整

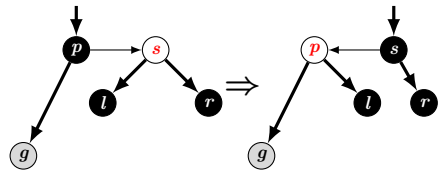


(a) ***l*** 是红色结点；***p*** 可以是红色或黑色，调整后 ***l*** 的颜色与之相同

(b) ***p*** 是红色结点



(c) ***r*** 是红色结点



(d) ***s*** 是红色结点；调整后根据 ***l*** 左子结点的颜色转变为 (a) 或 (b)

2.3 散列技术

散列技术

- **散列** (Hashing) (也称哈希) 技术通常用于实现字典数据结构：对每一个可能出现的键值赋予唯一序列索引使得元素的查找、插入、删除等操作非常方便和快捷
- 一些基本概念：
  - **散列码** (hash code)：键值对应的存储位置索引
  - **散列函数** (hash function)：用于计算键值所对应的散列码的函数
  - **冲突** (collision)：不同键值映射到同一个散列码
  - **散列表** (hash table)：数组 $H[0..h-1]$，用于存放 h 个散列元 (hash cell)，元素键值被散列函数映射到 $[0, h-1]$；通常散列技术的具体实现就是设计和维护一个这样的散列表
- 散列表的设计所需解决的问题：
  1. 使用什么样的散列函数？
  2. 如何处理冲突？

封闭地址散列

- **封闭地址散列** (Closed Address Hashing)也叫链式散列 (Chained Hashing) 采用最简单的冲突处理策略：
  - H 中的每个散列元 $H[i]$ 是一个链表，链表中元素的散列码均为 i
  - 插入元素 K：计算 K 的散列码 i；将该元素插入链表 $H[i]$
  - 搜索元素 K：计算 K 的散列码 i；在链表 $H[i]$ 中查找元素 K
  - 负载因子 (load factor) $\alpha = n/h$，其中，n 为散列表中存储的元素数，即负载因子是散列表中平均每个散列元链表所存储的元素数
- 成功搜索的平均复杂度：$a + \frac{1}{n}\sum_{i=0}^{h-1}\frac{(L_i+1)L_i}{2}$
- 最坏情况下 (散列函数映射不均匀或者添加的元素分布不均匀) 搜索时间复杂度可到 $\Theta(n)$，与线性搜索复杂度相当
- 如果散列函数设计得足够好，使元素键值平均地映射到 $[0, h-1]$，则成功搜索的平均复杂度可以达到 $O(1+\alpha)$
- 不成功的搜索平均复杂度约为成功搜索的 2 倍

如果元素键值平均地映射到 $[0, h-1]$，$L_i = \alpha$，成功搜索的平均复杂度为

$$a + \frac{1}{n}\sum_{i=0}^{h-1}\frac{(\alpha+1)\alpha}{2} = a + \frac{1}{n}\cdot\frac{(\alpha+1)h\alpha}{2} = a + \frac{1+\alpha}{2}$$

开放地址散列

- **开放地址散列** (Open Address Hashing)：
  - H 中每个散列元 $H[i]$ 直接存储元素而不是指向一个链表
  - 当冲突发生时，重新计算散列码 (rehashing) 找下一个可能的存储位置，因此，对于每个元素都有一系列可能的散列码与之对应
  - 灵活性不如封闭地址散列 (负载因子不大于 1 的情况很少出现)
  - 但比封闭地址散列更节省空间，搜索速度也更快
- 最简单的散列码重算策略——线性探查 (linear probing)：
  - $\text{Rehash}(i) = (i+1) \mod h$
  - 可以证明，当负载因子达到 1 时，成功搜索的平均复杂度将达到 $\sqrt{n}$ (长探查链的存在导致)
  - 为什么还要引入开放地址散列？$\Rightarrow$ 在低负载因子的情况下有更好的表现 (负载因子 $\alpha < 1$ 时成功搜索可以达到 $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$)
  - 利用“数组加倍”技术可使负载因子始终保持在 0.5 以下
- 更好的散列码重算策略——双散列 (double hashing)：
  - $d = \text{HashIncr}(K)$
  - $\text{Rehash}(i, d) = (i+d) \mod h$

散列函数设计

- 一个简单的设计原则是生成尽可能随机的散列码
- 例如采用“乘同余法 (multiplicative congruential) ”生成均匀分布的伪随机序列：乘以一个常数，然后对另一个常数求余
- 一个例子：
  1. 选择 h 为 2 的幂：$h = 2^x,\ h \geq 8$
  2. $a = 8\lfloor h/23 \rfloor + 5$
  3. 若元素键值是整数：$\text{HashCode}(K) = (aK) \mod h$
  4. 若元素键值是一对整数 (K_1, K_2)：$\text{HashCode}(K_1K_2) = (a^2K_1 + aK_2) \mod h$
  5. 若元素键值是一个字符串，将其当作一列整数 $k_1, k_2 \cdots$：$\text{HashCode}(K) = (a^l k_1 + a^{l-1}k_2 + \cdots + ak_l) \mod h$
  6. 当负载因子大于 0.5 时，将数组空间加倍，更新 h 和 a 的值，将原表中的元素在新参数值下插入到新表中
  7. 若采用双散列策略，HashIncr 可设计得尽量简单，但要保证在这个增量下不断重新计算散列码可以覆盖整个散列表，例如可取 $\text{HashIncr}(K) = (2k_1 + 1) \mod h$

动态集合 + 散列技术：布隆过滤器

- **布隆过滤器** (Bloom Filter) 1970 年由 Burton Howard Bloom 在论文“Space/Time Trade-offs in Hash Coding with Allowable Errors”中首先提出，是一种空间效率很高的动态集合，用于快速地查询一个元素是否在一个集合中
- 数据结构：Bit Vector (m 位)
- 添加元素：通过 k 个散列函数将这个元素映射成 Bit Vector 中的 k 个 bit，把它们置为 1
- 查询操作：用同样的散列函数进行计算，看散列值对应的 bit 位是否为 1，如果有一个不为 1，则这个元素一定不在集合中，否则只能表明这个元素很有可能在集合中
- 时间复杂度：插入和查询都是 $O(k)$
- 错误率 (probability of false positives)：假定散列值均匀分布，且各散列函数相互独立

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k$$

更多关于散列技术的分析

- 参考文献：

[1] Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching (2nd Edition)* (Section 6.4), Addison-Wesley, 1997

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms (2nd Edition)* (Chapter 11), The MIT Press, 2001

[3] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures*, Addison-Wesley, 1991

2.4 动态等价关系及合并-查找程序

动态等价关系

- 集合 $\mathbf{S}$ 上的 **等价关系** (equivalence ralation) $\mathbf{R}$ 是 $\mathbf{S}$ 上满足下列性质的二元关系：
  1. **自反性** (reflexive)：$\forall x \in \mathbf{S},\ x\mathbf{R}x$
  2. **对称性** (symmetric)：$\forall x, y \in \mathbf{S},\ x\mathbf{R}y \Rightarrow y\mathbf{R}x$
  3. **传递性** (transitive)：$\forall x, y, z \in \mathbf{S},\ x\mathbf{R}y, y\mathbf{R}z \Rightarrow x\mathbf{R}z$
- **等价类** (equivalence class)：给定一个集合 $\mathbf{S}$ 和在 $\mathbf{S}$ 上的一个等价关系 $\equiv$，则 $\mathbf{S}$ 中的一个元素 x 的等价类是在 $\mathbf{S}$ 中等价于 x 的所有元素构成的子集，即 $[x] = \{y \in \mathbf{S} | x \equiv y\}$
- **动态等价关系** (dynamic equivalence relation)：随计算过程不断改变的等价关系
- 问题：动态等价关系的表示、更改和查询，即处理包含下面两种指令的指令序列
  1. **IS** $s_i \equiv s_j$?
  2. **MAKE** $s_i \equiv s_j$ (之前 $s_i \equiv s_j$ 为假)
- 复杂度分析包含两个因素 (输入数据)：
  1. 包含 n 个元素的集合
  2. 包含 m 条指令 (**IS** 或 **MAKE**) 的指令序列
- 两种显而易见的实现方式：
  - 矩阵实现：利用矩阵表示等价关系，最少需要 $n^2/2$ 个存储单元，**IS** 指令只需检查一个单元，复杂度为 $\Theta(1)$，**MAKE** 需要拷贝矩阵某些行的值，因此在最坏情况下，指令序列包含 m 个 **MAKE** 指令，需要至少 mn 次操作
  - 数组实现：用数组存储等价关系，数组元素 $A[i]$ 记录包含集合元素 s_i 的等价类标识，则 **IS** $s_i \equiv s_j$ 指令需要检查 $A[i]$ 和 $A[j]$ 是否相等，复杂度为 $\Theta(1)$，而 **MAKE** $s_i \equiv s_j$ 指令则需要检查所有数组元素，将所有等于 $A[i]$ 的元素重新赋值为 $A[j]$，因此在最坏情况下仍然需要至少 mn 次操作，但与矩阵方式相比，空间复杂度有所降低
- 有没有更快的方法？

“合并-查找”程序

- **合并-查找** (Union-Find) 程序是包含如下两条基本操作的程序：
  1. `Union`(u, v)：合并等价类 $[u]$ 和 $[v]$ 成为一个等价类
  2. `Find`(s)：查找集合元素 s 所在的等价类 (标识)
- 前面所定义的 **IS** 和 **MAKE** 可以用这两条基本操作完成：

IS $s_i \equiv s_j$	**MAKE** $s_i \equiv s_j$
$u =$ `Find`(s_i);	$u =$ `Find`(s_i);
$v =$ `Find`(s_j);	$v =$ `Find`(s_j);
$(u = v)$?	`Union`(u, v)

- 初始化等价类：`MakeSet(1)`, `MakeSet(2)`, $\cdots$, `MakeSet`(n)
- 等价类的存储：in-tree (结点仅可访问其祖先，不能访问其子孙)

`MakeNode`	构造一棵包含一个结点的 in-tree
`SetParent`	改变一个结点的父结点
`SetNodeData`	设置结点所存储的元素值
`IsRoot`	测试结点是否是根结点 (没有父结点的结点)
`Parent`	返回结点的父结点
`NodeData`	返回结点存储的元素值

“合并-查找”程序的应用

- 合并-查找程序主要应用在数据的内容和结构在处理过程中不断改变的情形 (on-line operation)
  - 可用于求带权无向图的最小生成树的 Kruskal 算法 (在有关图算法的内容中会提到)
  - 可用于在程序设计语言中对等价声明 (equivalence declarations) 的处理。等价声明意味着多个变量或数组元素将共享同样的存储位置，例如，如下 Fortran 中的 EQUIVALENCE 语句调用

    EQUIVALENCE $(A, B(3)), (B(4), C(2)), (X, Y, Z),$
    $(J(1), K), (B(1), X), (J(4), L, M)$

    将产生如下效果：

		A					$J(1)$	$J(2)$	$J(3)$	$J(4)$	$J(5)$
$B(1)$	$B(2)$	$B(3)$	$B(4)$	$B(5)$			K			L	
X		$C(1)$	$C(2)$	$C(3)$	$C(4)$	$C(5)$				M	
Y											
Z											

  - 可以用于实现动态集合的其他基本操作

“合并-查找”程序的时间复杂度

- 每棵 in-tree 的根结点可作为等价类标识，则：
  - `Find`(s) 即返回包含结点 s 的 in-tree 的根结点：从 s 开始不断调用 `Parent`直到遇到根结点
  - `Union`(u, v) 的参数 u 和 v 必须为根结点且 $u \neq v$，其操作就是将 u 和 v 所代表的 in-tree 合并成一棵 in-tree：直接调用 `SetParent`(u, v) 即可完成
- 时间复杂度分析的输入规模：包含 n 个元素的集合以及规模为 m 的合并-查找程序：由 `MakeSet` 构成的等价类初始化序列及 m 个 `Union` 或 `Find` 操作组成的操作序列
- 显而易见，合并-查找程序的执行时间与对根结点的访问 (lookup 和 assignment) 次数成正比，因此可以将根结点的访问作为基本操作，并称其为链接操作 (link operation)，认为其时间复杂度为 $\Theta(1)$，则：
  - `MakeSet` 和 `Union` 包含 1 次根结点赋值操作
  - `Find`(s) 包含 $d+1$ 次根结点查询操作，d 为结点 s 的深度
  - 合并-查找程序最坏情况时间复杂度为 $\Theta(mn+n) = \boldsymbol{\Theta(mn)}$

加权合并

- 合并-查找程序复杂度为 $\Theta(mn)$ 的主要原因是 in-tree 的高度得不到限制 (最高可达 $n-1$)，使得最坏情况下 `Find` 操作很低效
- 改进办法是设计更好的 `Union` 操作，使 in-tree 的高度得到控制
- **加权合并** (Weighted Union)：`WUnion`(u, v) 总是取 u 和 v 中权值较大的那个作为合并后 in-tree 的根，其时间复杂度仍然为 $\Theta(1)$
- 最简单的权值就是取 in-tree 所包含的结点数

Lemma 2.10. 如果用 `WUnion` 实现等价类 *in-tree* 的合并操作，即选择结点数较多的 *in-tree* 的根做为合并后 *in-tree* 的根，而另一棵 *in-tree* 则作为其一棵子树，则一系列 `WUnion` 构造的任何包含 k 个结点的 *in-tree* 高度至多为 $\lfloor \lg k \rfloor$.

Proof. 使用数学归纳法.
奠基：$k=1$，高度为 $0=\lfloor \lg 1 \rfloor$.
归纳：假设对 $m<k$，结论成立，即任何由 WUnion 序列构造的包含 m 个结点的树高度至多为 $\lfloor \lg m \rfloor$，则对于由 T_1 和 T_2 两棵树通过 WUnion 构造的含 k 个结点，高度为 h 的树，不妨设 T_1 结点更多，则 T_2 作为子树链接到 T_1 的根结点，设 k_1, h_1 和 k_2, h_2 分别为 T_1 和 T_2 的结点数和高度，则有：$h_1 \le \lfloor \lg k_1 \rfloor$，$h_2 \le \lfloor \lg k_2 \rfloor$，合并后树的高度 $h=\max(h_1, h_2+1)$，又因为 $k_2 \le k/2$，故 $h_2 \le \lfloor \lg k \rfloor - 1$，而显然 $h_1 \le \lfloor \lg k \rfloor$，所以 $h \le \lfloor \lg k \rfloor$. □

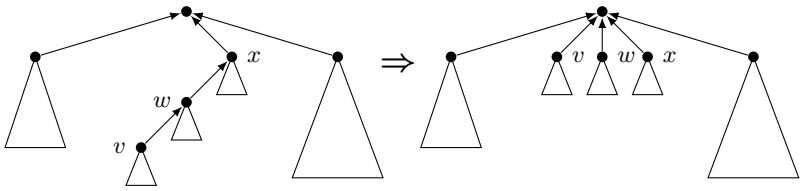
使用加权合并的合并-查找程序时间复杂度

Theorem 2.11. 包含 n 个元素的集合上含有 m 条基本操作的合并-查找程序，若使用 WUnion 和 Find，则在最坏情况下需要执行 $\Theta(n+m\log n)$ 次链接操作.

Proof. 对于 n 个元素的集合，最多可执行 $n-1$ 次 WUnion，构造出一棵含 n 个结点的树，根据引理，其高度至多为 $\lfloor \lg n \rfloor$，则每个 Find 操作最多执行 $\lfloor \lg n \rfloor + 1$ 次链接操作，而每个 WUnion 操作执行一次链接操作，因此最坏情况下的合并-查找程序对应 m 条操作全部为 Find，总共需要最多 $m(\lfloor \lg n \rfloor + 1)$ 次的链接操作，因此可以得到合并-查找程序链接操作次数的上界为 $O(n+m\log n)$，此外可以很容易构造出一个合并-查找程序使其链接操作次数下界为 $\Omega(n+m\log n)$. □

路径压缩

- Find 操作也可以改进以提高其速度
- **路径压缩** (path compression)：CFind(v) 在向上搜索 v 所在树的根结点过程中，将搜索路径上遇到的所有结点的父结点都改为最终找到的根结点



带路径压缩的查找操作

Procedure CFind(v)

```
1 oldParent ← Parent(v);
2 if oldParent =null then root ← v ;
3 else
4 |   root ← CFind(oldParent);
5 |   if oldParent ≠ root then SetParent(v, root);
6 end
7 return root;
```

- CFind 链接操作次数是 Find 的两倍
- 但 CFind 的使用使得 in-tree 的高度保持在很矮的状态
- 可以证明，使用 CFind 和无加权的 Union 作为基本操作，合并-查找程序的最坏情况时间复杂度仍然为 $\Theta(n+m\log n)$ *

*Michael J. Fischer. Efficiency of Equivalence Algorithms. In: *Complexity of Computer Computations*, pages 153–167. Plenum Press, New York, 1972.

同时使用 WUnion 和 CFind

- WUnion 和 CFind 操作是否相容？
  - 显然 WUnion 不会影响 CFind
  - 由于 CFind 只改变 in-tree 的高度而不改变其结点数，因此不会影响 in-tree 的权值，因而也不影响 WUnion 的操作
- 复杂度分析
  - 从直观上讲应该比 $\Theta(n + m \log n)$ 更好一些
  - CFind 的时间开销比较难把握，比数组加倍的分析更复杂一些
  - 可以用分摊时间分析法来进行分析

由 WUnion 所构造的 in-tree 的一些性质

Definition 2.12 (森林 F，结点高度和阶). 对于一个特定的合并-查找程序 P，F 是 WUnion 操作序列 (忽略所有的 CFind 操作) 构造出的森林 (forest)，结点 v 的高度 (node height) 是森林 F 中以 v 为根的子树的高度，v 的阶 (rank) 定义为其结点高度.

Lemma 2.13. 在森林 F 中：

1. 阶为 r 的结点至多有 $n/2^r$ 个
2. 任意结点的阶不大于 $\lfloor \lg n \rfloor$
3. 在执行合并-查找程序 P 的任意时刻，任一 *in-tree* 从叶结点到根结点的路径上结点的阶呈严格递增序列，当 CFind 改变一个结点的父结点时，新的父结点的阶大于原父结点阶.

一个新的函数：$\lg^*$

Definition 2.14 (函数 H 和 $\lg^*$). 定义函数 H 如下：

$$H(0) = 1,$$
$$H(n) = 2^{H(n-1)} \quad \text{for } n > 0$$

则：$\lg^*(m) = \min\{n | H(n) \geq m\} \quad (m > 0)$

n	0	1	2	3	4	5	6	$\cdots$
$H(n)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$	$\cdots$
$\lg^*(n)$		0	1	2	2	3	3	$\cdots$

n	16	17	$\cdots$	65536	65537
$H(n)$	??	??	$\cdots$	??	??
$\lg^*(n)$	3	4	$\cdots$	4	5

- 由定义可知：$\forall$常数$p \geq 0,\ \lg^*(n) \in o(\log^{(p)}(n))$

$\lg^*$ 的递归定义：

$$\lg^*(1) = 0,$$
$$\lg^*(n) = 1 + \lg^*(\lceil \lg n \rceil) \quad \text{for } n > 1$$

结点组

Definition 2.15 (结点组 (node groups))**.** 结点组 $s_i = \{v | v \in F, \lg^*(1 + \text{rank}(v)) = i\}$

- 不同的阶和其对应的结点组：

r (rank)	0	1	$2 \sim 3$	$4 \sim 15$	$16 \sim 65535$	$65536 \sim (2^{65536} - 1)$
i (group)	0	1	2	3	4	5

Lemma 2.16. 包含 n 个元素的集合 **S** 中不同结点组个数至多为 $\lg^*(n+1)$

Proof. 任一结点的阶最大为 $\lfloor \lg n \rfloor$，因此最大结点组下标为：

$$\lg^*(1 + \lfloor \lg n \rfloor) = \lg^*(\lceil \lg(n+1) \rceil) = \lg^*(n+1) - 1$$

而下标从 0 开始，所以最多结点组个数为 $\lg^*(n+1)$ □

时间分摊策略设计

1. `MakeSet`
   - *accounting cost*：$4\lg^*(n+1)$；*actual cost*：1； *amortized cost*：$1 + 4\lg^*(n+1)$
2. `WUnion`
   - *accounting cost*：0；*actual cost*：1；*amortized cost*：1
3. `CFind`
   - 设调用 `CFind`(v) 时从 v 到根结点的路径为 $w_0, w_1, \cdots, w_k$，其中 w_k 为根结点：当 $k \geq 2$ 时，对路径上的每一个结点对 (w_{i-1}, w_i)，若 w_{i-1} 和 w_i 所在的结点组相同且 $1 \leq i \leq k-1$，则 *accounting cost* 为 -2，称结点 w_{i-1} 提取开销 2；其他情况 *accounting cost* 为 0
   - *actual cost*：$2k$
   - *amortized cost*：$2k - 2((k-1) - (\text{路径上不同结点组个数} - 1)) \leq \mathbf{2k - 2((k-1) - (\lg^*(n+1) - 1)) = 2\lg^*(n+1)}$
   - 尽管最坏情况下 `CFind` 的实际开销可达 $2\lg n$，经过时间分摊后，每个 `CFind` 的开销最多仅为 $\mathbf{2\lg^*(n+1)}$

分摊策略的合理性

Lemma 2.17. 上述时间分摊策略是合理的，即其 *accounting cost* 始终非负

Proof. 等价类初始化总共调用 n 次 `MakeSet`，存入 *accounting cost* 开销总和为 $4n\lg^*(n+1)$；任一结点 w 提取开销发生在 w 处于某一次 `CFind` 操作路径上，且与其父结点在同一结点组内，且其父结点不是根结点。当 `CFind` 给其指定新的父结点时，根据前面的引理，新父结点的阶大于其原父结点的阶，则在不断调用 `CFind` 的同时，其不断更换的父结点的阶也在不断增长，一旦某次新父结点的阶高到与其不在同一个结点组内，则 w 就不再有提取开销的可能，在此之前，其提取开销的次数最多不超过其所在结点组中不同阶的结点个数 $(< H(i))$，因此，我们有了一个 F 中所有结点可能提取开销次数的上界：

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i) \cdot (\text{结点组 } i \text{ 中的结点个数})$$

根据引理，阶为 r 的结点最多为 $n/2^r$，因此结点组 i 中的结点个数可计算如下：

$$\sum_{r=H(i-1)}^{H(i)-1} \frac{n}{2^r} \leq \frac{n}{2^{H(i-1)}} \sum_{j=0}^{\infty} \frac{1}{2^j} = \frac{2n}{2^{H(i-1)}} = \frac{2n}{H(i)}$$

则提取开销总和上界为：

$$2\sum_{i=0}^{\lg^*(n+1)-1} H(i)\left(\frac{2n}{H(i)}\right) = 4n\lg^*(n+1)$$

不超过由 MakeSet 存入的开销总和，因此得证。 □

WUnion + CFind 实现的合并-查找程序复杂度上界

Theorem 2.18. 在有 n 个元素的集合上执行包含 m 条由 WUnion 和 CFind 操作组成的合并-查找程序最坏情况下总共需要 $O((n+m)\lg^*(n))$ 次链接操作

- 一点讨论：
  - 从 $\lg^*$ 的定义可以看出它是一个增长速度相当缓慢的函数，在实际有意义的输入规模下，基本上可以认为 $\lg^* n \leq 5$，可以把它当作常数
  - 从证明过程可以看出我们在建立不等式时条件是相当宽松的，这是否意味着由 WUnion 和 CFind 实现的合并-查找程序复杂度有可能达到 $\Theta(n+m)$？
  - 是否存在 Union 和 Find 的某种实现技术使合并-查找程序达到线性复杂度？这仍然是一个未解决的问题
- 参考文献：

[1] J. E. Hopcroft and J. D. Ullman. Set Merging Algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973

[2] R. E. Tarjan. On the Efficiency of a Good but Not Linear Set Union Algorithm. *Journal of the ACM*, 22(2):215–225, 1975

[3] M. L. Fredman and M. E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989

2.5 优先队列和配对森林

优先队列和减少键值操作

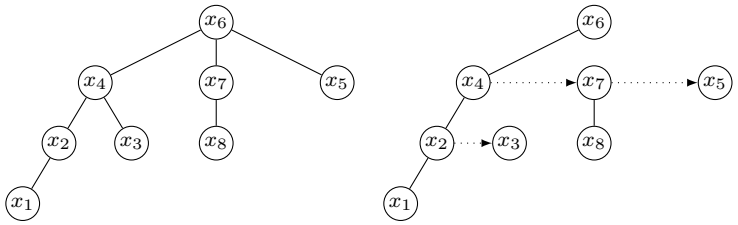
- 最小优先队列 (minimizing priority queue) 所需提供的操作：

```
构造方法：  Create
状态访问：  IsEmpty, GetMin, GetPriority
基本操作：  Insert, DeleteMin, DecreaseKey
```

- 利用二叉堆实现最小优先队列：
  - 基本操作可在 $O(\log n)$ 时间复杂度内实现 (DecreaseKey 需要辅助数据结构)
  - IsEmpty 和 GetMin 时间复杂度为 $O(1)$
  - GetPriority 在辅助数据结构的帮助下可以达到 $O(1)$
  - 辅助数据结构：设置一个字典数据结构 xref，关键字为元素唯一 id，对应的值为该元素在二叉堆中的位置 (最简单的实现：id 为整数，字典可用数组存储)
- 如果对 DecreaseKey 操作调用相当频繁，是否有进一步降低复杂度的方法？

配对森林

- **配对森林** (Pairing Forest)：一组满足偏序树特性的 out-tree 构成的森林 (这里只研究满足最小偏序树特性的配对森林)
  - **偏序树特性** (partial order tree property)：树中任一结点的键值小于 (或大于) 其所有子结点的键值
  - out-tree：与 in-tree 相对，即一般的树结构，从任一结点可以访问其所有子结点
  - 在配对森林中，每一条从根结点到叶结点的路径都构成一个键值 (或优先值) 的递增序列
  - 配对森林可用链表结构存储：



取最小元素 (联赛策略)

Algorithm `GetMin`(pq)

```
1 while pq.forest 超过 1 棵树 do pq.forest ← PairForest(pq.forest) ;
2 return 仅剩一棵树的根结点 id;
```

Procedure `PairForest`($oldForest$)

```
1  remainTrees ← oldForest;
2  while remainTrees ≠ null do
3  |  t1 ← First(remainTrees);
4  |  remainTrees ← Rest(remainTrees);
5  |  if remainTrees = null then newForest ← Cons(t1, newForest) ;
6  |  else
7  |  |  t2 ← First(remainTrees);
8  |  |  t ← PairTree(t1, t2);
9  |  |  newForest ← Cons(t, newForest);
10 |  |  remainTrees ← Rest(remainTrees);
11 |  end
12 end
13 return newForest;
```

取最小元素 (cont.)

Procedure `PairTree`(Tree $t1$, Tree $t2$)

```
1 if Root(t1).priority < Root(t2).priority then
2 |  newTree ← BuildTree(Root(t1), Cons(t2, Children(t1)));
3 else
4 |  newTree ← BuildTree(Root(t2), Cons(t1, Children(t2)));
5 end
```

- 复杂度分析
  - 如果在调用 `GetMin` 之前配对森林中有 k 棵树，则比较次数为 $k-1$ (找最小元素所需的最少比较次数)

– 在最坏情况下 k 可能等于结点数 n，但配对操作会使树的数目减少，因此最坏情况并不多见

– 关于该操作的准确复杂度分析仍然是未知的

插入和删除

Procedure `Insert`(pq, v, w)

```
1 构造 newNode 使其 id 为 v，优先值 priority 为 w;
2 newTree ← BuildTree(newNode, null);
3 xref[v] ← newTree;
4 pq.forest ← Cons(newTree, pq.forest);
```

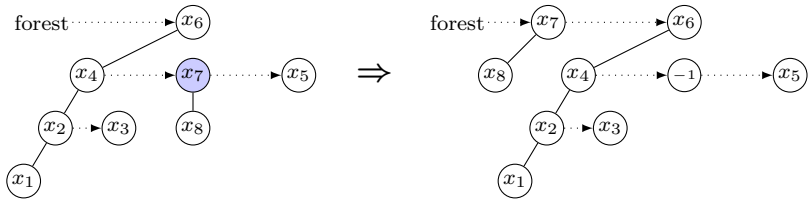
时间复杂度为 $O(1)$.

Procedure `DeleteMin`(pq)

```
1 GetMin(pq);
2 t ← First(pq.forest);
3 pq.forest ← Children(t);
```

时间复杂度与 `GetMin` 相同.

降低键值



Procedure `DecreaseKey`(pq, v, w)

```
1 构造 newNode 使其 id 为 v，优先值 priority 为 w;
2 oldTree ← xref[v];
3 oldNode ← Root(oldTree);
4 newTree ← BuildTree(newNode, Children(oldTree));
5 xref[v] ← newTree;
6 oldNode.id ← −1;
7 pq.forest ← Cons(newTree, pq.forest);
```

时间复杂度为 $O(1)$.

算法概论

第五讲：高级设计与分析技术

薛健

Last Modified: 2019.1.7

主要内容

1 动态规划

1.1 基本方法

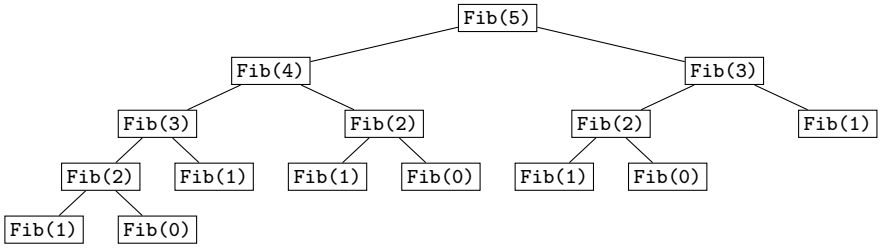
例子

Algorithm Fib(n)

```
1 if n < 2 then f ← n;
2 else
3 |   f1 ← Fib(n − 1);
4 |   f2 ← Fib(n − 2);
5 |   f ← f1 + f2;
6 end
7 return f;
```

- 算法时间复杂度达到 $O(c^n)$!
- 在递归调用中，有大量的重复调用，例如在 Fib(5) 的计算中，Fib(2) 被重复调用了 3 次，而 Fib(1) 被调用了 4 次

- 如何改进？



时间复杂度：

$$
\begin{aligned}
W(n) &= W(n-1) + W(n-2) + 1 \\
&\geq 2W(n-2) + 1 \\
&\geq 1 + 2 + 2^2 + \cdots + 2^{n/2} \\
\Rightarrow & W(n) \in \Omega(\sqrt{2}^n) \\
W(n) &\leq 2W(n-1) + 1 \\
&\leq 1 + 2 + 2^2 + \cdots + 2^n \\
\Rightarrow & W(n) \in O(2^n)
\end{aligned}
$$

动态规划

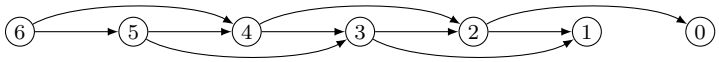
- **动态规划** (Dynamic Programming)是运筹学的一个分支，是求解决策过程 (Decision Process) 最优化的数学方法
- **多阶段决策过程** (Multistep Decision Process)：有一类问题的解决过程可以分为若干个阶段，而在任一阶段后的行为都仅依赖于阶段 i 的过程状态，而与阶段 i 如何通过之前的过程达到这种状态的具体方式无关
- 20 世纪 50 年代初美国数学家 Richard Bellman 等人在研究多阶段决策过程的优化问题时，根据这类问题多阶段决策的特性，提出了著名的**最优性原理** (Principle of Optimality)，把多阶段过程转化为一系列单阶段问题，逐个求解，从而创立了解决最优化问题的新的算法设计方法——动态规划
- 动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其它方法求解更为方便

子问题图

- 分治法的基本思路是将原始问题分解成子问题递归求解

Definition 1.1. <2->[子问题图 (Subproblem Graph)] 假设算法 A 是解决某个问题的递归算法，A 的子问题图是一个有向图，其结点代表输入数据，从结点 v_i 到 v_j 的有向边 $v_i \to v_j$ 代表在解决输入为 v_i 的子问题时需要递归调用算法 A 解决输入为 v_j 的子子问题。对于算法 A 的某个输入 P，A(P) 的子问题图就是 A 的子问题图中从结点 P 可以到达的部分子图。

- 如果算法 A 总是能够正常终止，则其子问题图必定是**有向无环图** (Directed Acyclic Graph, DAG)，动态规划的本质即是对子问题图中的结点进行逆**拓扑排序** (topological sort)，每一个结点子问题的解决总是依赖于它前一个结点子问题的解决。一个拓扑序列即对应一个解决方案。
- `Fib(6)` 的子问题图



拓扑排序：将偏序集 $(\mathbf{S}, \prec)$ 中的元素排成一行，使得在该序列中元素 a 排在元素 b 的前面当且仅当 $a \prec b$。偏序集一般可用 DAG 表示，则拓扑排序就对应于 DAG 的深度优先搜索。

递归算法的动态规划版本

- 给定递归算法 A，其动态规划版本 (dynamic programming version) $\mathcal{DP}(\mathrm{A})$ 是子问题图 $\mathrm{A}(P)$ 深度优先遍历过程，且每当处理完一个子问题，将其结果存入字典 soln，由此，递归算法 A 的动态规划版本 $\mathcal{DP}(\mathrm{A})$ 可对 A 按如下方式修改得到：
  1. 在递归调用 $\mathrm{A}(Q)$ 之前检查 soln 中是否记录了 Q 的结果
     a 若 soln 中还没有 Q 的结果，则递归调用 $\mathrm{A}(Q)$，即在子问题图中沿 $P \to Q$ 遍历结点 Q
     b 若 soln 中已有 Q 的结果，则直接取出结果，不再做递归调用
  2. 在返回当前问题 P 的处理结果之前将其存入 soln
- 上述方法仍然采用自顶向下 (top down) 的设计思路
- 暂存子问题结果的方法被称为 memoization
- 一般情况下 $\mathcal{DP}(\mathrm{A})$ 需要一个封装 (wrapper) 调用来初始化存储子问题结果的字典结构

`Fib2`

- 仍然采用自顶向下 (top-down) 的分治法思想
- 将已经计算过的子问题结果缓存下来 (memoization)

Algorithm `Fib2`(n)

```
1 map m ← map(0 ← 0, 1 ← 1);
2 return FibRec(m, n);
```

Procedure `FibRec`$(\text{map } m, n)$

```
1 if m 不包含关键字 n then
2 |   f1 ← FibRec(m, n − 1);
3 |   f2 ← FibRec(m, n − 2);
4 |   m[n] ← f1 + f2;
5 end
6 return m[n];
```

时间复杂度：$O(n)$　　**空间复杂度**：$O(n)$

实际上已经不需要递归，可以很容易地转化为循环：

Algorithm `Fib2Loop`(n)

```
1 f[0] ← 0;
2 f[1] ← 1;
3 for i ← 2 to n do
4 |   f[i] ← f[i − 1] + f[i − 2];
5 end
6 return f[n];
```

`Fib3`

- 采用自底向上 (bottom-up) 的设计思想，先解小问题，再用小问题的结果解决更大的问题

Algorithm `Fib3`(n)

```
previous ← 0;
current ← 1;
for i ← 1 to n − 1 do
    new ← previous + current;
    previous ← current;
    current ← new;
end
return current;
```

时间复杂度：$O(n)$　　空间复杂度：$O(1)$

1.2 矩阵链乘问题

矩阵链乘法 (Matrix Chain Multiplication)

- 问题描述：
  - 矩阵相乘：需要 rst 次乘法和 $rt(s-1)$ 次加法，我们用乘法次数来衡量矩阵相乘的时间复杂度

$$A_{r\times s}=[a_{ij}],\quad B_{s\times t}=[b_{ij}]$$
$$C_{r\times t}=AB=[c_{ij}]=[\sum_{k=1}^{s}a_{ik}b_{kj}]$$

  - 计算 n 个矩阵的乘积，需要 $n-1$ 次矩阵乘法运算
  - 矩阵乘法满足结合律，因此每次可以任选两个相邻矩阵进行乘法运算
  - **不同的选择次序可以导致不同的总执行时间**
- 目标：**找到一种结合次序使得总的时间消耗最少**

例子

Example 1.1 (4 个矩阵相乘). 有 $A_{30\times1},B_{1\times40},C_{40\times10},D_{10\times25}$ 4 个矩阵，找最快的方式求乘积 $ABCD$

- 总共有 5 种相乘次序：

乘积次序	乘法次数
$(((AB)C)D)$	$30\times1\times40+30\times40\times10+30\times10\times25=20700$
$(A(B(CD)))$	$40\times10\times25+1\times40\times25+30\times1\times25=11750$
$((AB)(CD))$	$30\times1\times40+40\times10\times25+30\times40\times25=41200$
$((A(BC))D)$	$1\times40\times10+30\times1\times10+30\times10\times25=8200$
$(A((BC)D))$	$1\times40\times10+1\times10\times25+30\times1\times25=1400$

- 如果有 n 个矩阵相乘，则可能的结合次序数目将变得十分庞大 (等于 Catalan number $C_{n-1}=C_{2n-2}^{n-1}/n$)，用穷举法就不合适了

关于 Catalan number：

$$C_n = \frac{C_{2n}^n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

在组合数学里跟多个计数问题相关，如 C_n 等于给 $n+1$ 个因子加括号的可能性数目 (或 $n+1$ 个因子由二元操作符结合次序的可能性数目)；也等于有 $n+1$ 个叶结点的不同满二叉树 (2-tree) 个数。

自顶向下的递归算法设计

- 计算 $A_1A_2\cdots A_n$，其中 A_i 大小为 $d_{i-1}\times d_i$，$1\le i\le n$
- **回溯算法** (backtracking algorithm)：
  - 选择第一对矩阵乘积的位置 i，求 $B=A_iA_{i+1}$
  - 用 B 代替 A_i 和 A_{i+1}，原问题变成规模为 $n-1$ 的子问题
  - 穷尽第一次相乘位置 i 所有可能的选择，则可求出最优解

Algorithm `MatrixChainOrderTry`$(d[], len, seq[])$

```
1  if len < 3 then bestCost ← 0 ;
2  else
3      bestCost ← ∞;
4      for i ← 1 to len − 1 do
5          c ← 位置 seq[i] 处的乘积开销;
6          newSeq ← seq[0..i − 1, i + 1..len];
7          b ← MatrixChainOrderTry(d, len − 1, newSeq);
8          bestCost ← min(bestCost, b + c);
9      end
10 end
11 return bestCost;
```

$$T(n) = (n-1)T(n-1)+n \Rightarrow T(n)\in\Theta((n-1)!)$$

`MatrixChainOrderTry` 的动态规划版本？

- 子问题图：从原始序列 $0,\cdots,n$ 开始，任意不少于 3 个元素的子序列都可以成为原始问题结点可到达的子问题结点
- 这样的子问题结点数目可达 2^n！
- 动态规划算法设计的一条重要原则：子问题必须可以被简洁地标识 (即子问题图的结点数必须得到有效控制)
- 子问题图结点数的阶若能控制在多项式级别，则可以保证所设计出的动态规划算法时间复杂度也能控制在多项式级别
- 对于矩阵链乘问题，我们需要另外的子问题划分方法：找最后一次矩阵相乘的位置
  - $B_{1(d_0\times d_i)}=A_1\cdots A_i$，$B_{2(d_i\times d_n)}=A_{i+1}\cdots A_n$
  - 最后一步是 B_1 和 B_2 相乘，开销为 $d_0d_id_n$
  - 原问题 $(0,n)\Rightarrow$ 子问题 $(0,i)$ 和 (i,n)

新分割方法尝试

Algorithm `MatrixChainOrderTryII`$(d[], lo, hi)$

```
1 if hi − lo = 1 then bestCost ← 0;
2 else bestCost ← ∞;
3 for k ← lo + 1 to hi − 1 do
4 |   a ← MatrixChainOrderTryII(d, lo, k);
5 |   b ← MatrixChainOrderTryII(d, k, hi);
6 |   c ← 位置 k 处的乘积开销;
7 |   bestCost ← min(bestCost, a + b + c);
8 end
9 return bestCost;
```

- 这仍然是一个回溯算法
- 时间复杂度：准确的递归方程比较复杂，但我们可以通过简化后的形式得到它的一个下界：2^n
- 从时间复杂度来看仍然不够好，但我们的目的已经达到

在循环中忽略规模较小的子问题，而只保留规模最大的两个子问题，则可以得到时间复杂度的递归不等式：

$$T(n) \geq 2T(n-1) + n$$

`MatrixChainOrderTryII` 的动态规划版本

- 子问题图：
  - 每个子问题可以用 $0, \cdots, n$ 内的一对整数 (i, j) 表示，$i < j$
  - $0, \cdots, n$ 内可以挑出大约 $n^2/2$ 个这样的整数对
  - 对每一个 (i, j) 以及从 $i + 1$ 到 $j - 1$ 的 k，有两个子问题需要解决，因此从结点 (i, j) 出发边小于 $2n$
  - 整个子问题图的结点数小于 n^2 而边数小于 n^3
  - 子问题图的深度优先搜索时间复杂度为 $O(n^3)$
- 因此，我们可以写出 `MatrixChainOrderTryII` 的动态规划版本
- 如何恢复最优链乘开销所对应的链乘次序？
  - 使用另一个字典记录子问题 (i, j) 所对应的最优相乘位置
  - 算出最优解后就可以根据该字典回溯出每个子问题的最后相乘位置，也就可以得到整个矩阵链的相乘次序
- 时间复杂度：与子问题图中的结点数和边数相关，在动态规划算法中，每个结点和每条边只需被处理和访问一次，因此时间复杂度为 $O(n^3)$，比指数阶的复杂度要好得多

`MatrixChainOrderTryIIDP`

Algorithm `MatrixChainOrderTryIIDP`$(d[], lo, hi, cost[][])$

```
1  if hi − lo = 1 then bestCost ← 0;
2  else bestCost ← ∞;
3  for k ← lo + 1 to hi − 1 do
4  |  if Member(cost, lo, k) then a ← Retrieve(cost, lo, k) ;
5  |  else a ← MatrixChainOrderTryIIDP(d, lo, k, cost) ;
6  |  if Member(cost, k, hi) then b ← Retrieve(cost, k, hi) ;
7  |  else b ← MatrixChainOrderTryIIDP(d, k, hi, cost) ;
8  |  c ← 位置 k 处的乘积开销;
9  |  if bestCost > a + b + c then
10 |  |  bestCost ← a + b + c; bestPos ← k;
11 |  end
12 end
13 Store(cost, lo, hi, bestCost); Store(last, lo, hi, bestPos);
14 return bestCost;
```

自底向上 (bottom up) 的设计思路

- 如果能够找到一种方便的方式生成子问题图的逆拓扑排序，则可以采用自底向上的方式来设计算法，消除递归：
  1. 对每个矩阵计算最优乘积次序，显然对单个矩阵不需要乘法
  2. 对每两个相邻矩阵计算最优乘积次序，显然只有一种次序
  3. 对每三个相邻矩阵计算最优乘积次序，需要用到上面两步的结果
  4. 假设任意 k 个相邻矩阵的最优乘积次序已经得到，则根据已知结果计算 $k+1$ 个相邻矩阵的最优乘积次序
  5. 得到任意 n 个相邻矩阵的最优乘积次序，算法结束
- 对于子问题 (i, j)，若固定 i，随 j 增加，对应逆拓扑序列中结点 (i, j) 之后的结点，而固定 j，随 i 减少，也对应逆拓扑序列中结点 (i, j) 之后的结点，因此可以用两重循环来构造逆拓扑序列，外层循环对应 i 递减，内层循环对应 j 递增

算法描述

Algorithm `MatrixChainOrder`$(d[], n)$

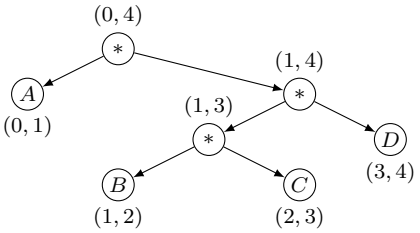
```
1  for lo ← n − 1 down to 0 do
2  |  for hi ← lo + 1 to n do
3  |  |  if hi − lo = 1 then bestCost ← 0; bestPos ← −1;
4  |  |  else bestCost ← ∞;
5  |  |  for k ← lo + 1 to hi − 1 do
6  |  |  |  c ← cost[lo][k] + cost[k][hi] + d[lo]d[k]d[hi];
7  |  |  |  if c < bestCost then
8  |  |  |  |  bestCost ← c;
9  |  |  |  |  bestPos ← k;
10 |  |  |  end
11 |  |  end
12 |  |  cost[lo][hi] ← bestCost; last[lo][hi] ← bestPos;
13 |  end
14 end
15 return cost and last;
```

时间复杂度：$\Theta(n^3)$

例子

$$A_{30\times 1} \times B_{1\times 40} \times C_{40\times 10} \times D_{10\times 25}$$

$$cost = \begin{bmatrix} \cdot & 0 & 1200 & 700 & 1400 \\ \cdot & \cdot & 0 & 400 & 650 \\ \cdot & \cdot & \cdot & 0 & 10000 \\ \cdot & \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \quad last = \begin{bmatrix} \cdot & -1 & 1 & 1 & 1 \\ \cdot & \cdot & -1 & 2 & 3 \\ \cdot & \cdot & \cdot & -1 & 3 \\ \cdot & \cdot & \cdot & \cdot & -1 \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$



$\Rightarrow \quad \boldsymbol{A((BC)D)}$

提取链乘次序

Algorithm `ExtractOrder`$(n, last[][])$

```
1 global next ← 0;
2 mult ← Array(n − 1);
3 ExtractOrderRec(0, n, last, mult);
```

Procedure `ExtractOrderRec`$(lo, hi, last[][], mult[])$

```
1 if hi − lo > 1 then
2 |   k ← last[lo][hi];
3 |   ExtractOrderRec(lo, k, last, mult);
4 |   ExtractOrderRec(k, hi, last, mult);
5 |   mult[next] ← k;
6 |   next ← next + 1;
7 end
```

时间复杂度：$T(n) = 2n - 1 \in \Theta(n)$　递归深度：$\Theta(n)$

1.3 最优二叉搜索树问题

问题描述

- 回顾：二叉搜索树 (Binary Search Tree)
  - 用中序 (inorder) 遍历二叉搜索树可得到从小到大排序的元素序列
  - 包含相同元素的二叉搜索树平衡度 (degrees of balance) 可以差别很大
  - `BSTSearch`：搜索所需比较次数与所找结点在树中的深度有关，最坏情况搜索时间复杂度为 $\Theta(n)$，如果二叉搜索树尽可能达到平衡，则最坏情况复杂度可降到 $\Theta(\log n)$
- 设可供搜索的元素键值为 $K_1, K_2, \cdots, K_n$，每个元素被搜索的可能性为 $p_1, p_2, \cdots, p_n$
- 将所有元素放入一棵二叉搜索树中，设找到 K_i 所需的键值比较次数为 c_i，则二叉搜索树 T 的平均比较次数为：

$$A(T) = \sum_{i=1}^{n} p_i c_i$$

- 若每个元素被搜索的可能性不相等，则平衡二叉树未必能得到较低的 $A(T)$

问题描述 (cont.)

- **构造最优二叉搜索树：**
  - 给定元素键值 $K_1, K_2, \cdots, K_n$ 及其被搜索的概率 $p_1, p_2, \cdots, p_n$，构造一棵二叉搜索树使搜索的平均比较次数 $A(T)$ 最小
- 基本思路：
  - 假定 $K_1, K_2, \cdots, K_n$ 按从小到大排序
  - 若选择 K_i 为二叉搜索树的根结点，则其左子树包含 $K_1, \cdots, K_{i-1}$，右子树包含 $K_{i+1}, \cdots, K_n$
  - 则问题转化为两个子问题：构造最优左子树和右子树
  - 根结点取遍所有 K_i 即可找到最优解
- 与矩阵链乘问题类似，子问题可用整数对 $(low, high)$ 表示，即构造 $K_{low}, \cdots, K_{high}$ 的最优二叉搜索树
- 每个子问题的时间开销 (平均比较次数) 计算需要进一步考察

子问题时间开销的计算

- **几个定义：**
  - $\boldsymbol{A(low, high, r)}$：选择 K_r 为根结点时子问题 $(low, high)$ 的最小带权搜索开销
  - $\boldsymbol{A(low, high)}$：子问题 $(low, high)$ 的最小带权搜索开销 (取遍所有可能 K_r 作为根结点)
  - $\boldsymbol{p(low, high)} = \sum_{i=low}^{high} p_i$：实际上是所搜索的元素键值落在区间 $[K_{low}, K_{high}]$ 中的概率
- 包含 $K_{low}, \cdots, K_{high}$ 的二叉搜索树的带权搜索开销为 W，其根结点深度为 0，若将其作为某一个新的根结点的子树，其根结点深度变为 1，其带权搜索开销则变为 $W + p(low, high)$
- **建立递推方程：**

$$\begin{aligned} A(low, high, r) &= p_r + p(low, r-1) + A(low, r-1) \\ &\quad + p(r+1, high) + A(r+1, high) \\ &= p(low, high) + A(low, r-1) + A(r+1, high) \\ A(low, high) &= \min\{A(low, high, r) \,|\, low \le r \le high\} \end{aligned}$$

算法描述

Algorithm `OptimalBST`$(p[], n)$

```
1 for low ← n + 1 down to 1 do
2 |   for high ← low − 1 to n do BestChoice(p, cost, root, low, high) ;
3 end
4 return cost and root;
```

Procedure `BestChoice`$(p[], cost[][], root[][], low, high)$

```
1 if high < low then bestCost ← 0; bestRoot ← −1 ;
2 else bestCost ← ∞;
3 for r ← low to high do
4 |   c ← p(low, high) + cost[low][r − 1] + cost[r + 1][high];
5 |   if c < bestCost then bestCost ← c; bestRoot ← r ;
6 end
7 cost[low][high] ← bestCost;
8 root[low][high] ← bestRoot;
```

时间复杂度：$\Theta(n^3)$　　如何计算 $\mathrm{p}(i,j)$？

- $\mathrm{p}(i,j)$ 可以通过 $\Theta(n)$ 的预处理得到在算法运行过程中 $\Theta(1)$ 的复杂度
- $\mathrm{p}(i,j)$ 的计算方法：
  - 方法 1：用一个 $\Theta(n^2)$ 的预处理过程计算每一个 $\mathrm{p}(i,j)$（注意：$i \le j$），这样在 BestChoice 中可以用 $\Theta(1)$ 的时间取得 $\mathrm{p}(low, high)$
  - 方法 2：保持 BestChoice 中 $\mathrm{p}(low, high)$ 的复杂度为 $\Theta(1)$，但预处理的复杂度可以降到 $\Theta(n)$，考虑下面的关系：

    $$\mathrm{p}(i,j) = \mathrm{p}(1,j) - \mathrm{p}(1,i)$$

    因此在预处理步骤只要计算出 $\mathrm{p}(1,1)$ 到 $\mathrm{p}(1,n)$ 的值即可，显然该步骤时间复杂度为 $\Theta(n)$

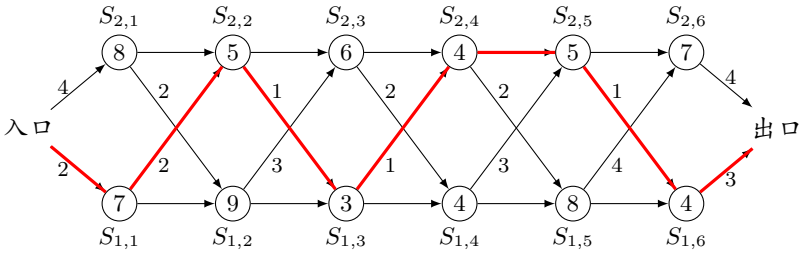
1.4 装配线调度问题

装配线调度问题

- **问题描述**
  - 假定某汽车生产线有两条装配线，待装配的汽车底盘可选择进入任一条装配线，经过 n 道工序进行装配，每道工序负责添加某些零件，当最后一道工序完成即装配成整车
  - 记两条装配线上的工序分别为 $S_{1,1}, S_{1,2}, \cdots, S_{1,n}$ 和 $S_{2,1}, S_{2,2}, \cdots, S_{2,n}$
  - 假设装配线 1 上的工序 $S_{1,k}$ 和装配线 2 上的工序 $S_{2,k}$ $(1 \le k \le n)$ 完成同样的装配工作，因此从 $S_{1,k-1}$ 可以选择进入 $S_{1,k}$ 或 $S_{2,k}$ 继续下一道工序，但若要从装配线 1 切换到装配线 2 需要额外的转换时间 $t_{1,k-1}$，从装配线 2 切换到装配线 1 与之类似
  - 完成每道工序所需时间为 $a_{i,k}$ $(i = 1, 2; 1 \le k \le n)$，进入装配线和移出装配线的时间分别为 e_i 和 x_i $(i = 1, 2)$
- **目标**：找到一种调度方法使得总的装配时间最短

例子

Example 1.2 (有 6 道工序的装配线).



实际上就是求从“入口”到“出口”的带权最短路径。

算法设计

- 问题分割：
  - 子问题 (i,k)：装配线 i 上的第 k 道工序完成时已消耗时间的最小值 $(i=1,2;1\leq k\leq n)$
- 是否具有最优子结构性质：
  - 假设 $s,s_1,s_2,\cdots,s_n,t$ 是从入口 s 到出口 t 的最优装配路径
  - 其子路径 $s,s_1,s_2,\cdots,s_i$ 一定是从 s 到 s_i 的最优装配路径，否则，若存在另一条从 s 到 s_i 的更快的装配路径 $s,s'_1,\cdots,s'_{i-1},s_i$，则 $s,s'_1,\cdots,s'_{i-1},s_i,s_{i+1},\cdots,t$ 是比 $s,s_1,s_2\cdots,s_n,t$ 更优的装配路径，与原假设矛盾
- 建立子问题求解的递推方程：
  - $f_{i,k}$ 记录装配线 i 上的第 k 道工序完成时已消耗时间的最小值，则：

$$
\begin{aligned}
f_{1,1} &= e_1 + a_{1,1} \\
f_{2,1} &= e_2 + a_{2,1} \\
f_{1,k+1} &= \min\{(f_{1,k} + a_{1,k+1}),\ (f_{2,k} + a_{1,k+1} + t_{2,k})\} \\
f_{2,k+1} &= \min\{(f_{2,k} + a_{2,k+1}),\ (f_{1,k} + a_{2,k+1} + t_{1,k})\}
\end{aligned}
$$

算法描述

Algorithm FastestWay$(f[][],a[][],t[][],n,e1,e2,x1,x2)$

```
1  f[1][1] ← e1 + a[1][1];
2  f[2][1] ← e2 + a[2][1];
3  l ← array[2][n];
4  for i ← 2 to n do
5  |  StepForward(f, a, t, l, i);
6  end
7  if f[1][n] + x1 ≤ f[2][n] + x2 then
8  |  totalf ← f[1][n] + x1;
9  |  lastl ← 1;
10 else
11 |  totalf ← f[2][n] + x2;
12 |  lastl ← 2;
13 end
14 PrintSchedure(l, n, lastl);
```

StepForward

Procedure `StepForward`$(f[][], a[][], t[][], l[][], i)$

```
1  if f1[i − 1] + a1[i] ≤ f2[i − 1] + a1[i] + t2[i − 1] then
2  |   f1[i] ← f1[i − 1] + a1[i];
3  |   l[1][i] ← 1;
4  else
5  |   f1[i] ← f2[i − 1] + a1[i] + t2[i − 1];
6  |   l[1][i] ← 2;
7  end
8  if f2[i − 1] + a2[i] ≤ f1[i − 1] + a2[i] + t1[i − 1] then
9  |   f2[i] ← f2[i − 1] + a2[i];
10 |   l[2][i] ← 2;
11 else
12 |   f2[i] ← f1[i − 1] + a2[i] + t1[i − 1];
13 |   l[2][i] ← 1;
14 end
```

`PrintSchedure`

Procedure `PrintSchedure`$(l[][], n, lastl)$

```
1  i ← lastl;
2  k ← n;
3  repeat
4  |   print “装配线” i, “工序” k − 1;
5  |   i ← l[i][k];
6  |   k ← k − 1;
7  until k < 2;
```

1.5 最佳断行问题

问题描述

- 在文字排版技术中存在这样的问题：
  - 如何给一个英文单词序列断行，使构成的段落排版结果最美观？
- **最优断行** (Optimization Line-breaking) 问题：
  - 在 Knuth 为 “The Art of Computer Programming” 设计的排版系统 TEX 中引入
  - 输入：n 个单词的长度 $w_1, \cdots, w_n$ 及排版行宽 W
  - 输出：在行宽为 W 的限制下，n 个单词构成的段落
- 问题的简化：
  - 假定字母的宽度相等，且每个单词后面跟一个空格，则 w_i 可由构成单词的字母个数加 1 来表示
  - W 以每一行可放下的字母个数表示，则从第 i 个单词到第 j 个单词若要排在同一行中必须满足：$\sum_{k=i}^{j} w_k \leq W$，而这一行的剩余空白长度为 $X = W - \sum_{k=i}^{j} w_k$
  - 剩余的空白会影响美观，因此可以构造一个惩罚函数来反映这种情况，例如取 X^3，但段落的最后一行例外
  - 目标：使每一行惩罚函数值之和最小

算法设计思路

- 子问题分割：
  - 将原问题 $(1,n)$ 分割为 $(1,k)$ 和 $(k+1,n)$
  - k 有可能是一个很糟糕的断行位置
  - 段落最后一行不计惩罚，但不包含段落最后一行的所有子问题各自的最后一行照常需要计惩罚
  - 有两类不同结构的子问题
  - 需要证明：取遍所有 k，$(1,k)$ 和 $(k+1,n)$ 相结合的最优结果就是原问题 $(1,n)$ 的最优结果
- 有没有更好的分割方案？
  - 在上面的分析中有一类子问题只需要一个整数来表示：(i,n)
  - 另一类子问题是否还需要？$\Rightarrow$ Method99
  - 取遍所有的 k 使得单词 1 到 k 可以放进第一行，剩下的子问题就是如何对单词 $k+1$ 到 n 在段落剩下的部分中断行 (从第二行开始)
  - 综合第一行的惩罚函数值和子问题的惩罚函数值，其中和最小的一个即对应原始问题的最优解

算法描述

Algorithm `LineBreak`$(w[],W,i,n,L)$

```
1  if ∑_{k=i}^{n} w[k] ≤ W then
2  |  单词 i 到 n 放入第 L 行 ; penalty ← 0;
3  else
4  |  k ← 1; penalty ← ∞; kmin ← 0;
5  |  while ∑_{t=i}^{i+k-1} w[t] ≤ W do
6  |  |  X ← W − ∑_{t=i}^{i+k-1} w[t];
7  |  |  kp ← LineCost(X)+ LineBreak(w, W, i + k, n, L + 1);
8  |  |  if kp < penalty then penalty ← kp; kmin ← k ;
9  |  |  k ← k + 1;
10 |  end
11 |  将单词 i 到 i + kmin − 1 放入第 L 行;
12 end
13 return penalty;
```

`LineBreak` $\Rightarrow$ `LineBreakDP`　时间复杂度：$\Theta(Wn)=\Theta(n)$

对每一行的 k，至多有 $W/2$ 种选择，子问题图中约有 n 个结点，每个结点至多发出 $W/2$ 条边，W 通常可看作常数，因此 `LineBreakDP` 的时间复杂度为 $\Theta(Wn)=\Theta(n)$。

1.6 动态规划一般步骤

动态规划算法设计的一般步骤

1. 使用自顶向下 (top down) 的方式分析问题，给出一般递归算法 (复杂度可能是指数级的)
2. 分析所得递归算法的子问题图，看是否可以用暂存子问题结果 (memoization) 的方式减少子问题的重复计算，若是，则可用固定的转换步骤将一般递归算法转换为其动态规划版本，此处的关键是：子问题的标识应尽可能简洁，以便限制子问题图的规模和所需暂存空间的大小

3. 根据子问题图 (深度优先搜索) 分析动态规划版本的递归算法时间复杂度，看是否达到要求

4. 设计用于暂存子问题结果的字典数据结构

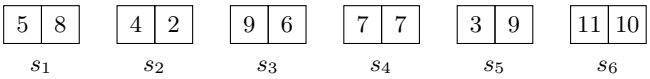
5. 分析子问题图的结构，看是否可以简单地得到结点的逆拓扑序列，若是则可以按自底向上 (bottom up) 的方式设计出动态规划算法的非递归版本

6. 确定从字典数据中提取最优解 (决策过程) 的方法

1.7 课后习题

“多米诺骨牌”问题的动态规划算法

Exercise (6). 现有 n 块“多米诺骨牌” $s_1, s_2, \cdots, s_n$ 水平放成一排，每块骨牌 s_i 包含左右两个部分，每个部分赋予一个非负整数值，如下图所示为包含 6 块骨牌的序列。骨牌可做 180 度旋转，使得原来在左边的值变到右边，而原来在右边的值移到左边，假设不论 s_i 如何旋转，$L[i]$ 总是存储 s_i 左边的值，$R[i]$ 总是存储 s_i 右边的值，$W[i]$ 用于存储 s_i 的状态：当 $L[i] \leq R[i]$ 时记为 0，否则记为 1，试设计时间复杂度为 $O(n)$ 的动态规划算法求 $\sum_{i=1}^{n-1} R[i] \cdot L[i+1]$ 的最大值，以及当取得最大值时每个骨牌的状态。下面是 $n = 6$ 时的一个例子。



deadline: 2018.12.29

2 贪心方法

2.1 基本方法

贪心方法

- 与动态规划方法一样，贪心方法一般也用来解决某种优化问题，与动态规划不同的是贪心方法并不穷尽所有子问题的最优解
- 基本思路：
  - 在一个决策序列中，每一步单独的决策其优劣有一个度量标准来衡量
  - 每一步决策总是选择在度量标准下最优的那个分支 (决定)
  - 每一步决策一旦决定便不再更改 (不同于回溯算法)
- 几点注意：
  - 贪心方法给出的解不一定是 (全局) 最优解
  - 给定具体问题设计贪心算法，一般需要证明所设计的算法求得的确实是给定问题的最优解
  - 对同一个问题，贪心算法的效率一般要高于动态规划算法
- 经典贪心算法实例：
  - Prim 算法、Dijkstra 算法、Kruskal 算法
  - 将在图算法一讲具体介绍

2.2 活动日程安排问题

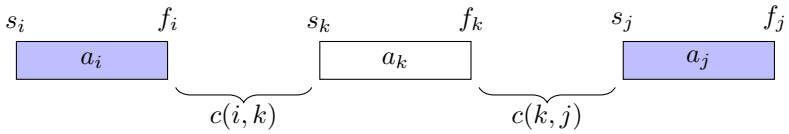
问题描述

- 某一公共场地计划安排 n 个不同活动：$a_1, a_2, \cdots, a_n$
- 活动 a_i 的开始时间和结束时间分别为 s_i 和 f_i $(0 \le s_i < f_i < \infty)$，因此，如果 a_i 得以进行，则公共场地将在 $[s_i, f_i)$ 时间段内被占用
- 两个不同活动 a_i 和 a_j 在时间上不存在冲突，则称这两个活动是相容的，即 $f_i \le s_j$ 或 $f_j \le s_i$
- **活动日程安排**：从活动 $a_1, a_2, \cdots, a_n$ 中选择时间不冲突的活动在公共场地举行
- **优化目标**：安排的活动尽可能多，即从 $\mathbf{S} = \{a_1, a_2, \cdots, a_n\}$ 中选出一个相容子集并使其中包含的元素最多 (场地的利用率最高)

动态规划方法

- 子问题：$\mathbf{S}(i,j) = \{a_k \in \mathbf{S} | f_i \le s_k < f_k \le s_j\}$
- 令 $f_0 = 0$，$s_{n+1} = \infty$，则原问题可用 $\mathbf{S}(0, n+1)$ 表示
- 将所有活动按结束时间从小到大排序，则当 $i \ge j$ 时 $\mathbf{S}(i,j) = \varnothing$
- 若用 $c(i,j)$ 表示子问题 $\mathbf{S}(i,j)$ 最优解所安排的活动个数，则：

$$c(i,j) = \begin{cases} 0 & \mathbf{S}(i,j) = \varnothing, \\ \max\limits_{i<k<j} \{c(i,k) + c(k,j) + 1\} & \mathbf{S}(i,j) \ne \varnothing. \end{cases}$$



- 写出动态规划算法 (课后练习)

贪心算法设计

- 贪心方法思路：每次取剩下活动中结束时间最早且与已安排活动不冲突的的那个加入日程安排

Algorithm `GreedyActivitySelector`$(a[], n)$

```
1  按 a[i].f 从小到大对 a 排序;
2  A ← {a[1]};
3  i ← 1;
4  for m ← 2 to n do
5  |  if a[m].s ≥ a[i].f then
6  |  |  A ← A ∪ {a[m]};
7  |  |  i ← m;
8  |  end
9  end
10 return A;
```

时间复杂度：$\Theta(n \log n)$

算法的正确性

Theorem 2.1. 令 $a_m \in \mathbf{S}(i,j)$ 且 $f_m = \min\{f_k | a_k \in \mathbf{S}(i,j)\}$，则有：

1. a_m 在 $\mathbf{S}(i,j)$ 的最大相容子集中；
2. $\mathbf{S}(i,m) = \varnothing$

Proof. 先看第 2 个结论，若假设 $\mathbf{S}(i,m) \neq \varnothing$，则存在 a_k 使得 $f_i \leq s_k < f_k \leq s_m < f_m$，这与 f_m 在 $\mathbf{S}(i,j)$ 中最小矛盾，因此 $\mathbf{S}(i,m) = \varnothing$；再看第 1 个结论，设 $\mathbf{A}(i,j)$ 是 $\mathbf{S}(i,j)$ 的最大相容子集，若假设 $a_m \notin \mathbf{A}(i,j)$，对 $\mathbf{A}(i,j)$ 中的元素按结束时间从小到大排序，设 a_k 是排序后第一个元素，则对 $\mathbf{A}(i,j)$ 中任意其他元素 a_x 均有 $f_k \leq s_x$，现构造 $\mathbf{A}'(i,j) = (\mathbf{A}(i,j) - \{a_k\}) \cup \{a_m\}$，则有 $f_m \leq f_k \leq s_x$，因此 $\mathbf{A}'(i,j)$ 也是一个相容子集，且元素数目与 $\mathbf{A}(i,j)$ 相同，所以 $\mathbf{A}'(i,j)$ 也是最大相容子集 (另一个最优解)。 □

2.3 旅行加油问题

问题描述

- 开车从城市 A 到城市 B，途经 n 个加油站
- 出发点和终点的加油站标记为第 0 个和第 $n+1$ 个加油站
- 每个加油站与前一个加油站之间的距离为 d_i ($d_0 = 0$)
- 每个加油站油价为 p_i (已折算成每公里耗油的油价，$p_{n+1} = 0$)
- 汽车加满油可行驶 L 公里
- 在城市 A、B 之间的任意 L 公里路程上至少有 1 个加油站，至多有 k 个加油站
- 出发前油箱是空的
- **问题**：在每个加油站应该加多少油？ (折算成能行驶的公里数)
- **优化目标**：油费最少

贪心算法设计

- 假设当行至加油站 i 时，油箱剩余的油还能走 r_i 公里
- 在加油站 i 需加 g_i 公里油
- 则要确定 g_i 和下一个停车加油的加油站，需分两种情况讨论：

1. 若在 L 公里内，u 是第一个油价低于 i ($p_u \leq p_i$) 的加油站
   - 加油量 $g_i = \sum_{j=i+1}^{u} d_j - r_i$，即刚够行驶到 u
   - 任何最优方案在从加油站 i 到加油站 $u-1$ 的过程中至少要加 g_i 的油
   - 同时也不会加多于 g_i 的油，因为多出的部分在加油站 u 加更划算
2. 若在 L 公里内，u 是油价最低的一个加油站，但 $p_u > p_i$
   - 加油量 $g_i = L - r_i$，即将油箱加满：假定某一最优方案在 i 加油量少于 g_i，则在下 L 公里中必然要在其他加油站加油，而这些油如果在 i 加显然更划算
   - 下一个要加油的加油站为 u：若在 u 之前或之后停车加油，显然所加的一部分油在 u 加更划算

算法描述

Algorithm `MinTrip`$(p[], d[], n)$

```
1  for k ← 0 to n do g[k] ← 0;
2  r[0] ← 0; cost ← 0; i ← 0;
3  while i < n do
4  |  D ← 0; v ← u ← i + 1;
5  |  while D + d[v] ≤ L and v ≤ n do
6  |  |  D ← D + d[v];
7  |  |  if p[v] ≤ p[u] then u ← v;
8  |  |  v ← v + 1;
9  |  |  if p[u] ≤ p[i] then break;
10 |  end
11 |  if p[u] ≤ p[i] or v > n then  g[i] ← D − r[i]; r[u] ← 0 ;
12 |  else  g[i] ← L − r[i]; r[u] ← L − ∑_{j=i+1}^{u} d[j] ;
13 |  i ← u; cost ← cost + p[i] * g[i];
14 end
15 return g and cost;
```

复杂度分析

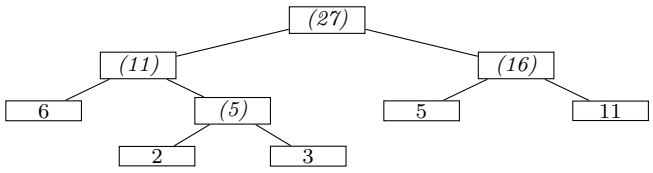
- 在每一个加油站最多需要向后搜索 k 个加油站
- 时间复杂度 $W(n) \in O(kn)$
- 如果 k 为常数，时间复杂度为 $O(n)$
- 如果 k 不是常数，是否有复杂度为 $O(n)$ 的算法？(课后练习)

2.4 钢管焊接问题

问题描述

- 现需要将 n 段钢管 $a_1, a_2, \cdots, a_n$ 焊接成一整条钢管，每次可任选两根钢管焊接在一起，假设钢管 a_i 的重量为 $W[i]$ $(1 \le i \le n)$，将每次焊接代价计为所焊的两根钢管的重量之和。例如，焊接 5 段钢管，其重量分别为 $5, 2, 6, 11, 3$，若按下图所示二叉树的顺序来焊接这些钢管，则焊接总代价为：

$$(2+3)+(5+6)+(5+11)+(11+16)=59$$



- 显然，每一种焊接流程都可以用如上图所示的一棵满二叉树来表示
- 试设计一个贪心算法找出一种最优焊接流程并分析算法的时间复杂度

贪心算法设计

- 每次选取剩余钢管中最轻的两段焊接在一起：

Algorithm `LeastCostWeld`$(W[], n)$

```
1  用 W[1..n] 构造小根堆 H;
2  for i ← 1 to n − 1 do
3  |  x ← ExtractMin(H);
4  |  y ← ExtractMin(H);
5  |  构造结点 z;
6  |  z.left ← x; z.right ← y;
7  |  z.weight ← x.weight + y.weight;
8  |  Insert(H, z);
9  end
10 return Root(H);
```

- 显然，该算法时间复杂度为 $O(n \log n)$

算法正确性分析

Lemma 2.2. $\mathrm{Cost}(T) = \sum_{k=1}^{n} W[k] \cdot \mathrm{depth}(k)$

Proof. 用数学归纳法证明，对二叉树高度 h 归纳：

奠基：$h = 0$ 和 $h = 1$ 时结论显然成立；

归纳：假设对于任意高度小于或等于 h 的二叉树结论成立，则对于高度为 $h+1$ 的二叉树 T，假设其左子树和右子树为 L 和 R，分别包含叶结点 $W[1] \sim W[j]$ 和 $W[j+1] \sim W[n]$，而 T 的根结点代表了最后一步焊接。则根据假设，左子树和右子树的焊接总代价分别为：

$$\mathrm{Cost}(L) = \sum_{k=1}^{j} W[k] \cdot \mathrm{depth}_L(k),\ \mathrm{Cost}(R) = \sum_{k=j+1}^{n} W[k] \cdot \mathrm{depth}_R(k)$$

而其完成的钢管总重量分别为 $\sum_{k=1}^{j} W[k]$ 和 $\sum_{k=j+1}^{n} W[k]$，因此最后一步焊接代价为 $\sum_{k=1}^{j} W[k] + \sum_{k=j+1}^{n} W[k]$，因此，最终的焊接总代价为：

$$\begin{aligned}
\mathrm{Cost}(T) &= \mathrm{Cost}(L) + \mathrm{Cost}(R) + \sum_{k=1}^{j} W[k] + \sum_{k=j+1}^{n} W[k] \\
&= \sum_{k=1}^{j} W[k] \cdot (\mathrm{depth}_L(k) + 1) + \sum_{k=j+1}^{n} W[k] \cdot (\mathrm{depth}_R(k) + 1) \\
&= \sum_{k=1}^{j} W[k] \cdot \mathrm{depth}_T(k) + \sum_{k=j+1}^{n} W[k] \cdot \mathrm{depth}_T(k) \\
&= \sum_{k=1}^{n} W[k] \cdot \mathrm{depth}_T(k)
\end{aligned}$$

因此对于高度为 $h+1$ 的二叉树，结论也成立。 □

Lemma 2.3. 若称焊接总代价最小的焊接流程为最优焊接流程，则在最优焊接流程所对应的满二叉树中，最底层的叶结点中必有一结点代表重量最轻的那段钢管。

Proof. 用反证法证明。

不失一般性，设 $W[1]$ 为最小重量，假设 $W[1]$ 不在二叉树最底层，$W[k]$ 为最底层的一个叶结点，则有 $W[k] > W[1]$，现在交换 $W[1]$ 和 $W[k]$ 的位置，得到新的焊接总代价：

$$\begin{aligned}
\mathrm{Cost}(new) &= \mathrm{Cost}(T) - W[1]\,\mathrm{depth}(1) - W[k]\,\mathrm{depth}(k) \\
&+ W[1]\,\mathrm{depth}(k) + W[k]\,\mathrm{depth}(1) \\
&= \mathrm{Cost}(T) - (W[1] - W[k])\,\mathrm{depth}(1) + (W[1] - W[k])\,\mathrm{depth}(k) \\
&= \mathrm{Cost}(T) - (W[1] - W[k])(\mathrm{depth}(1) - \mathrm{depth}(k))
\end{aligned}$$

因为 $W[1] < W[k]$ 且 $\text{depth}(1) < \text{depth}(k)$，所以 $(W[1] - W[k])(\text{depth}(1) - \text{depth}(k)) > 0$，所以 $\text{Cost}(new) < \text{Cost}(T)$，这与 T 是最优焊接流程矛盾，故假设不成立，$W[1]$ 在最底层。 □

Lemma 2.4. 在最优焊接流程所对应的满二叉树中，重量第二轻的钢管所对应的叶结点也在二叉树最底层的叶结点中。

Proof. 由于最优焊接流程所对应的二叉树为满二叉树，故其最底层至少有两个叶结点，如果第二轻的钢管不在最底层，则最底层除最轻叶结点外，必有一个结点重量大于第二轻的结点，交换这两个结点的位置，参照前面的推导可得一棵总代价更小的树，因此导致矛盾，原结论成立。 □

Theorem 2.5. 必有一种最优焊接流程在第一步焊接时将最轻的两段钢管焊接在一起。

Proof. 由前面的结论可知最优焊接二叉树最底层的叶结点必包含重量最轻的两段钢管，假设为 $W[1]$ 和 $W[2]$。若二者有共同父结点，则结论成立；若二者父结点不同，则必存在一个叶结点 $W[x]$ 与最轻的结点 $W[1]$ 共有父结点，现交换 $W[x]$ 和 $W[2]$ 的位置，由于都在最底层，结点深度未变，根据第一条引理的结论，焊接总代价也未变，这时 $W[1]$ 和 $W[2]$ 共有父结点，且焊接流程是一种最优焊接流程，因此结论成立。 □

2.5 课后习题

邮局位置问题

Exercise (7). 在一条街上有 n 所房子，$H[i]$ $(1 \le i \le n)$ 是第 i 所房子离街道起点处的距离 (以米为单位)，假定 $H[1] < H[2] < \cdots < H[n]$。目前该街道上还没有一所邮局，现计划新建若干所邮局，使得每所房子到最近的邮局距离在 100 米以内。试设计一个时间复杂度为 $O(n)$ 的算法，计算出新建邮局的位置，即每所新建邮局离街道起点处的距离 $P[j]$ $(1 \le j \le m)$，同时确保新建邮局个数 m 最小。 *deadline: 2019.01.12*

3 字符串匹配算法

3.1 基本问题

问题描述

- **字符串匹配问题**：在一个字符串 (通常称作文本或 text) 中搜索给定子串 (通常称为模式串或 pattern)
- 字符串匹配问题最早来自文本处理领域，如果将匹配的基本元素从字符扩展到其他元素，则字符串匹配算法可以应用在很多领域的数据检索中，在这一部分只讨论字符串匹配算法
- **记号约定**：

P	待搜索的模式串
T	待搜索文本
m	模式串长度
n	文本长度
p_i, t_i	模式串和文本中的第 i 个字符 (下标从 1 开始)
j	文本的当前匹配位置
k	模式串的当前匹配位置

一个简单算法

Algorithm `SimpleScan`$(P[], T[], m)$

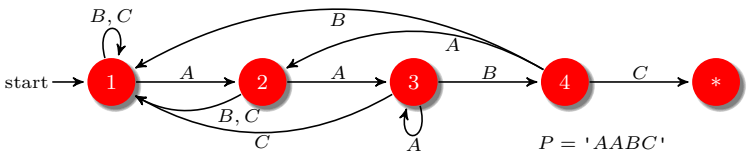
```
1  match ← −1; i ← j ← k ← 1;
2  while not EndText(T, j) do
3  |  if k > m then match ← i; break;
4  |  if t_j = p_k then j ← j + 1; k ← k + 1;
5  |  else
6  |  |  backup ← k − 1;
7  |  |  j ← j − backup; k ← k − backup;
8  |  |  i ← j ← j + 1;
9  |  end
10 end
11 return match;
```

- 最坏情况时间复杂度：$\Theta(mn)$
- 一些统计实验表明，文本 T 中的每个字符的平均比较次数约为 1.1
- 文本匹配位置 j 在扫描过程中需要经常回溯，因此该算法在某些环境下不适用

3.2 KMP 算法

模式匹配的有限自动机

- 用于模式匹配的**有限自动机** (finite automaton)：设 Σ 是文本和模式串中可能出现的所有字符的字母表，$\alpha = |\Sigma|$，则有限自动机可用包含下面两种结点的流程图来表示
  1. “读 (read)”结点：表示“读下一个文本字符，如果到达文本末尾，则停机；无匹配结果”
  2. “停 (stop)”结点：表示“停机；有匹配结果”，通常用 $*$ 标记
- 该流程图从每个“读”结点发出 α 个箭头，每个箭头标以字母表 Σ 的一个字母，表示匹配该字符所到达的下一个状态

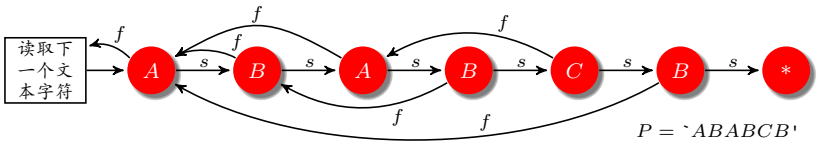


- 一旦构造出与模式串 P 对应的有限自动机，按该自动机构造的算法，文本中的每个字符只需比较 1 次，匹配算法的复杂度可以降到 $O(n)$，最大的困难在于有限自动机的构造

KMP 流程图

- Knuth-Morris-Pratt (KMP) 流程图简化了有限自动机的构造，它与通常的有限自动机有一些区别：
  - 待匹配字符直接标记在结点上而不是箭头上
  - 每个结点只发出两个箭头，分别代表匹配成功和失败
  - 读取下一个字符发生在成功箭头之后
  - 当发生一个失败匹配之后，同一个文本字符将按流程图再次匹配
  - 有一个额外结点表示读取文本中下一个字符

- 例如 $P =$ `$ABABCB$', 其 KMP 流程图为:



KMP 流程图的构造

- KMP 流程图可用两个数组存储: 一个存储模式串 (匹配结点), 一个存储失败匹配链接 (箭头)
- 假设失败匹配链接存储在数组 $fail$ 中, $fail[k]$ 为第 k 个结点匹配失败所指向的下一个结点下标; $fail[1] = 0$
- 设置失败链接 ($fail[7] = 5$):

$$\begin{array}{cccccc|cc} A & B & A & B & A & B & C & B \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \\ A & B & A & B & A & B & x & \cdots \end{array} \Rightarrow \begin{array}{cccc|cccc} A & B & A & B & A & B & C & B \\ & & & & & & & \\ A & B & A & B & x & & & \cdots \end{array}$$

Definition 3.1. <3->[失败链接] 失败链接 $fail[k]$ 是 $p_1 \cdots p_{r-1}$ 与 $p_{k-r+1} \cdots p_{k-1}$ 匹配的 r ($r < k$) 中的最大值, 即: 模式串 P 的 $r-1$ 个字符前缀与结束于 p_{k-1} 的 $r-1$ 个字符构成的子串相等

KMP 流程图的构造 (cont.)

- 设置失败链接的递归过程 ($s = fail[k-1]$):

$$\begin{array}{cc|ccc|cccc} p_1 & \cdots & p_{k-r+1} & \cdots & p_{k-2} & p_{k-1} & p_k & \cdots & p_m \\ & & \uparrow & \cdots & \uparrow & & & & \\ & & p_1 & \cdots & p_{s-1} & p_s & & \cdots & \\ & & & \cdots & \uparrow & & & & \\ & & & p_1 \cdots & p_{fail[s]-1} & p_{fail[s]} & & \cdots & \end{array}$$

Algorithm `KMPSetup`$(P[], m)$

```
1 fail[1] ← 0;
2 for k ← 2 to m do
3 |   s ← fail[k − 1];
4 |   while s ≥ 1 and p_s ≠ p_{k−1} do s ← fail[s] ;
5 |   fail[k] ← s + 1;
6 end
7 return fail;
```

`KMPSetup` 复杂度分析

- 粗看 `KMPSetup` 最坏情况时间复杂度为 $O(m^2)$, 实际上没有这么差
- p_s 和 p_{k-1} 的成功比较最多为 $m-1$ 次
- p_s 和 p_{k-1} 的每一次失败比较导致 s 递减 ($fail[s] < s$), 因此我们可以用 s 值减小的次数来计算失败比较的次数:
  - 当 $k = 2$ 时 s 初始值为 0
  - s 的值当在本次循环执行 $fail[k] \leftarrow s+1$ 及下次循环执行 $s \leftarrow fail[k-1]$ 后增 1, 随 for 循环总共 $m-2$ 次

– s 不可能为负值

- 根据以上分析，s 的值减少次数不可能多于 $m-2$
- 因此 KMPSetup 字符比较次数最多为 $2m-3$
- KMPSetup 最坏情况时间复杂度为 $O(m)$!

KMP 算法

Algorithm KMPScan($P[], T[], m, fail[]$)

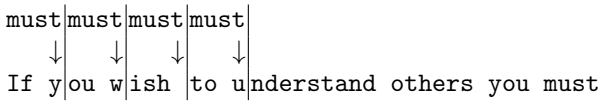
```
1 match ← −1; j ← k ← 1;
2 while not EndText(T, j) do
3 |   if k > m then match ← j − m; break;
4 |   if k = 0 then j ← j + 1; k ← 1;
5 |   else if t_j = p_k then j ← j + 1; k ← k + 1;
6 |   else k ← fail[k] ;
7 end
8 return match;
```

- 时间复杂度：
  - KMPScan 字符比较次数至多为 $2n$ ($Why?$)
  - KMPSetup 和 KMPScan 最坏情况时间复杂度为 $\Theta(n+m)$，比 SimpleScan 的 $\Theta(mn)$ 要好得多
  - 对于自然语言文本，有实验表明 KMP 算法和 SimpleScan 的平均字符比较次数是一样的，但 KMP 算法中不存在文本回溯问题

3.3 BM 算法

基本思路

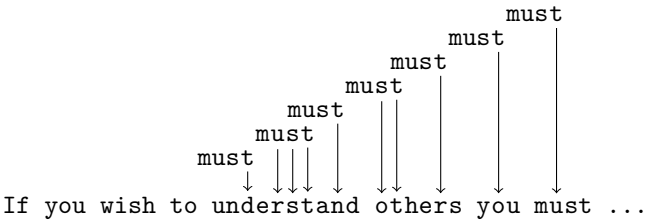
- 无论是 SimpleScan 还是 KMPScan 都需要逐一比较文本串中的字符，但不是所有的比较都是必须的，存在比 KMPScan 更快的算法
- 例如：



- 模式串越长，所能提供的匹配信息越多
- 一个好的模式匹配算法应该尽可能快地在文本串中跳过不可能出现模式串的部分，避免不必要的比较
- ⇒Boyer-Moore **算法** (BM **算法**)

改进 1

- 从右向左扫描模式串，直接跳过文本串中不可能产生匹配的部分
- 例如：



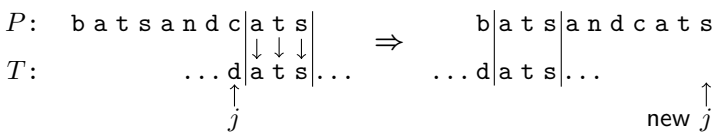
- 计算字符匹配失败情况下，j 在文本串中向后跳跃的字符数：

Algorithm `ComputeJumps`$(P[], m, \Sigma[], \alpha)$

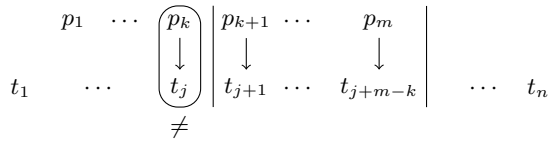
```
1 for ch ← 0 to α − 1 do  charJump[Σ[ch]] ← m ;
2 for k ← 1 to m do  charJump[p_k] ← m − k ;
3 return charJump;
```

改进 2

- 参考 KMP 算法的失败链接概念，每次失败匹配后可以跳过更多的不可能匹配字符
- 例如：

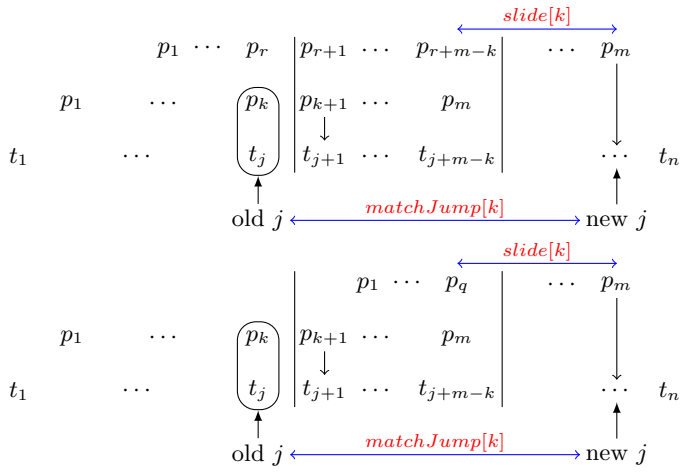


- 一般情况：



改进 2 (cont.)

- 一般情况 (续)：



matchJump 和 *slide* 的定义

Definition 3.2 ($matchJump$ 和 $slide$)**.** 首先，$matchJump[k]$ 可根据 $slide[k]$ 计算如下：

$$matchJump[k] = slide[k] + m - k, \quad 1 \le k \le m$$

令 r 为使 $p_{k+1} \cdots p_m$ 和 $p_{r+1} \cdots p_{r+m-k}$ 匹配且 $p_k \neq p_r$ 的最大下标，即：模式串 P 的 $m-k$ 个字符后缀与从 p_{r+1} 开始的 $m-k$ 个字符构成的子串相等。则：

$$slide[k] = k - r, \quad r < k$$

若 $k = m$，则 $p_{k+1} \cdots p_m$ 为空，则 r 为使得 $p_r \neq p_k$ 的最大下标；若找不到子串匹配整个 $p_{k+1} \cdots p_m$，则找模式串 P 前缀和后缀的最长匹配，若该匹配包含 q 个字符，则：

$$slide[k] = m - q$$

计算 $matchJump$

Algorithm `ComputeMatchJumps`$(P[], m)$

```
for k ← 1 to m do matchJump[k] ← m + 1 ;
sufx[m] ← m + 1;
for k ← m − 1 down to 0 do
    s ← sufx[k + 1];
    while s ≤ m and p_{k+1} ≠ p_s do
        matchJump[s] ← min(matchJump[s], s − (k + 1));
        s ← sufx[s];
    end
    sufx[k] ← s − 1;
end
low ← 1; shift ← sufx[0];
while shift ≤ m do
    for k ← low to shift do matchJump[k] ← min(matchJump[k], shift) ;
    low ← shift + 1; shift ← sufx[shift];
end
for k ← 1 to m do matchJump[k] ← matchJump[k] + m − k ;
return matchJump;
```

`ComputeMatchJumps` 的正确性？

证明要点：

1. $sufx$ 和 KMP 算法中的 $fail$ 是对称的
2. $sufx[k] = x$ 当且仅当子串 $p_{k+1} \cdots p_{k+m-x}$ 和后缀 $p_{x+1} \cdots p_m$ 匹配
3. 对于 $r_0 = sufx[0]$, $r_{i+1} = sufx[r_i]$，子串 $p_{r_i+1} \cdots p_m$ 是 P 的一个前缀

BM 算法

Algorithm `BMScan`$(P[], T[], m, charJump[], matchJump[])$

```
match ← −1; j ← k ← m;
while not EndText(T, j) do
    if k < 1 then match ← j + 1; break;
    if t_j = p_k then j ← j − 1; k ← k − 1;
    else
        j ← j + max(charJump[t_j], matchJump[k]);
        k ← m;
    end
end
return match;
```

一点讨论

- 当 $m \geq 5$ 时，`BMScan` 平均字符比较次数为 cn，其中 $c < 1$；当 $m \leq 3$ 时性能的微弱提升则会被预处理步骤所抵消 (主要跟字母表的大小有关)
- 对于自然语言文本，$m \geq 5$ 时有实验表明 BM 算法的平均字符比较次数在 0.24 到 0.3 之间，即文本串中只有 1/4 到 1/3 的字符参与了比较
- 对于二进制流，$charJump$ 的信息已经没有太大意义，此时字符平均比较次数约为 0.7
- 由于 BM 算法采用自右向左的比较方式，因此需要一个字符缓存保存一批待比较的字符
- 参考文献：

– G. de V. Smit. A Comparison of Three String Matching Algorithms. *Software: Practice and Experience*, Vol.12, No.1, pages 57–66, 1982

3.4 近似匹配算法

基本问题

- 在某些应用环境下不需要或者无法得到精确匹配结果，如单词拼写检查、语音识别等等
- 寻找文本串中模式串的一个近似匹配
- 问题描述：

  Definition 3.3. <2->[k-近似匹配] 给定模式串 $P = p_1 \cdots p_m$ 和文本串 $T = t_1 \cdots t_n$，假定 $n \gg m$，k-近似匹配是指 P 和 T 最多只有 k 个不同之处的匹配。所谓“不同之处”包括以下几种情况 (对应于 T 向 P 靠近所需的操作)：

  - **修改** (revise) P 和 T 对应字符不同
  - **删除** (delete) T 中包含一个 P 中没有的字符
  - **插入** (insert) P 中包含一个 T 中没有的字符

  需要从 T 中找出给定 P 的 k-近似匹配。

动态规划算法设计

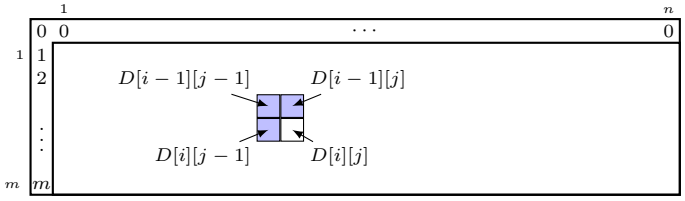
- 在扫描过程中每比较一对字符有四种选择：移动模式串或者执行前面所定义的三种不同情况的对应操作
- 子问题标识：(i, j)
  - 若采用自左向右的扫描模式，i 对应模式串某个后缀的起始位置，j 对应文本串当前匹配的开始位置，子问题可表述为寻找 $p_i \cdots p_m$ 和从 t_j 开始的一段文本的区别最小匹配，则对删除操作来说，子问题 (i, j) 依赖于 $(i, j+1)$，而 $(i, j+1)$ 依赖于 $(i, j+2)$
  - 若采用自右向左的扫描模式，i 对应模式串某个前缀的结束位置，j 对应文本串当前匹配的结束位置，子问题可表述为寻找 $p_1 \cdots p_i$ 和结束于 t_j 的一段文本的区别最小匹配，则对删除操作来说，子问题 (i, j) 依赖于 $(i, j-1)$，而 $(i, j-1)$ 依赖于 $(i, j-2)$
  - 问题的求解顺序一般按 j 的递增顺序，因此自右向左的扫描模式更合适

动态规划算法设计 (cont.)

- 若定义 $D[i][j] = p_1 \cdots p_i$ 和结束于 t_j 的一段文本的最小区别数，则 $D[i][j]$ 是下述四个值中的最小值：

$$
\begin{aligned}
matchCost &= D[i-1][j-1] && \text{若} p_i = t_j \\
reviseCost &= D[i-1][j-1] + 1 && \text{若} p_i \neq t_j \\
insertCost &= D[i-1][j] + 1 && \text{若} p_i \neq t_j \\
deleteCost &= D[i][j-1] + 1 && \text{若} p_i \neq t_j
\end{aligned}
$$

- 任何结束于 t_j 且满足 $D[m][j] \leq k$ 的文本子串都是 k-近似匹配
- 如果只需要找到文本串中出现的第一个 k-近似匹配，则算法只需计算到矩阵 D 最后一行第一个满足上述条件的位置即可停止



算法分析

- 显然，矩阵 D 每个元素的计算开销为常数 (包括一个字符比较)
- 因此整个算法复杂度应该在 $O(mn)$，和 `SimpleScan` 一样快
- 空间复杂度：
  - 矩阵 D 大小为 $m \times n$，通常 n 非常大
  - 但算法在计算过程中并不需要存储矩阵 D 中的所有值，而只需要存储当前计算的一列和它的前一列数值
  - 因此空间复杂度为 $O(m)$
- 算法描述 $\Rightarrow$ 课后练习
- 更深入的参考：复杂度为 $O(kn)$ 的 k-近似匹配算法：
  - Gad M. Landau and Uzi Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 220–230

算法概论

第六讲：图算法

薛健

Last Modified: 2019.1.12

主要内容

1 基本概念

图问题

- 不仅在计算机相关领域，在现实生活中的其他各个领域内都有大量的问题可以抽象为某种图的问题，因此基本的图算法应用范围十分广泛
- 所有与图相关的问题可以粗略地分为以下几类：
  1. **简单图问题**：已知线性复杂度 $(W(n) \in O(n))$ 的算法用于解决该类问题，最典型的就是图的遍历，这类算法可以解决规模非常大的问题
  2. **中等图问题**：已知多项式复杂度 $(W(n) \in O(n^a), a > 1)$ 的算法用于解决该类问题，这类算法可以解决规模相对较大的问题
  3. **困难图问题**：用于解决该类问题且复杂度低于多项式级别的算法还未找到，现有的算法仅能解决规模很小的问题 $\Rightarrow$ 算法研究的前沿

基本概念

Definition 1.1 (有向图 (directed graph / digraph))**. 有向图** $G = (V, E)$，其中 V 是一组顶点 (vertex) 也称结点 (node) 的集合，E 是一组顶点有序对的集合。E 中的顶点有序对称为**边** (edge) 或**有向边** (directed edge) 或**弧** (arc)。对于 E 中的有向边 (v, w)，v 称为尾 (tail)，w 称为头 (head)，在图中一般用箭头 $v \to w$ 表示

Definition 1.2 (无向图 (undirected graph))**. 无向图** $G = (V, E)$，其中 V 是一组顶点 (vertex) 也称结点 (node) 的集合，E 是一组顶点无序对的集合。E 中的顶点有序对称为**边** (edge) 或**无向边** (undirected edge)。E 中的每一条边都是 V 的包含 2 个元素的子集。E 中的边 $\{v, w\}$，在图中一般用连接两个顶点的直线 $v - w$ 表示。

Definition 1.3 (子图、对称图和完全图)**.**

- **子图** (subgraph)：$G = (V, E)$ 的子图是满足 $V' \subseteq V, E' \subseteq E$ 且 $E' \subseteq V' \times V'$ 的图 $G' = (V', E')$；
- **对称图** (symmetric graph)：是有向图，且其中的每一条边都存在一条与其方向相反的边；
- **完全图** (complete graph)：每两个顶点之间有边相连的图，一般是指无向图

Definition 1.4 (邻接关系 (adjacency relation))**.** 由有向图或无向图 $G = (V, E)$ 的边可以在顶点集合 V 上定义二元关系 A，对于 $v, w \in V$，vAw 当且仅当 $vw \in E$，该二元关系称为**邻接关系**。若 G 是无向图，则 A 是对称的。

Definition 1.5 (路径 (path))**.** 图 $G = (V, E)$ 由顶点 v 到顶点 w 的**路径**是 E 中的一组边构成的序列 $v_0v_1, v_1v_2, \cdots, v_{k-1}v_k$，其中 $v = v_0, w = v_k$。该路径的长度为 k。一个独立的顶点 v 可看作从 v 到其自身的长度为 0 的路径。$v_0, v_1, \cdots, v_k$ 各不相等的路径称为**简单路径** (simple path)。若顶点 v 和 w 之间存在一条路径，则称顶点 w 对顶点 v **可达** (reachable)。

Definition 1.6 (连通和强连通)**.**

- 称无向图 $G = (V, E)$ 是**连通的** (connected)，当且仅当 V 中的任意一对顶点 v 和 w 之间存在一条路径；
- 称有向图 $G = (V, E)$ 是**强连通的** (strongly connected)，当且仅当 V 中的任意一对顶点 v 和 w 之间存在一条从 v 到 w 的路径；

Definition 1.7 (环路 (cycle))**.**

- 有向图中的**环路**是指一条首顶点和末顶点相同的非空路径，若环路中除首末顶点外其他顶点各不相同，则该环路称为**简单环路** (simple cycle)；
- 无向图中的**环路**定义与有向图类似，额外条件是环路中任意出现多次的边必须为同一朝向，即：若环路 $vv_1, v_1v_2, \cdots, v_kv$ 中存在 $v_i = x, v_{i+1} = y$，则不允许再出现 $v_j = y, v_{j+1} = x$；
- 不存在环路的图称为**无环图** (acyclic graph)；
- 无向无环图称为**无向森林** (undirected forest)，若该图同时是连通的，则称为**自由树** (free tree) 或**无向树** (undirected tree)；
- 有向无环图通常缩写为 DAG

Definition 1.8 (连通分量和强连通分量)**.**

- 无向图 $G = (V, E)$ 的**连通分量** (connected component) 是指 G 的最大连通子图；
- 有向图 $G = (V, E)$ 的**强连通分量** (strongly connected component) 是指 G 的最大强连通子图；
- “最大”的含义：不是任何其他连通 (强连通) 子图的子图

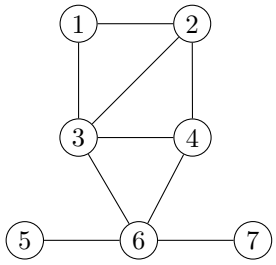
Definition 1.9 (带权图 (weighted graph))**. 带权图**是一个三元组 (V, E, W)，其中 (V, E) 是图，W 是从 E 到 $\mathbf{R}$ 的函数，对于 E 中的边 e，$W(e)$ 称为 e 的权。

2 图的表示和数据结构

图的矩阵表示和链表表示

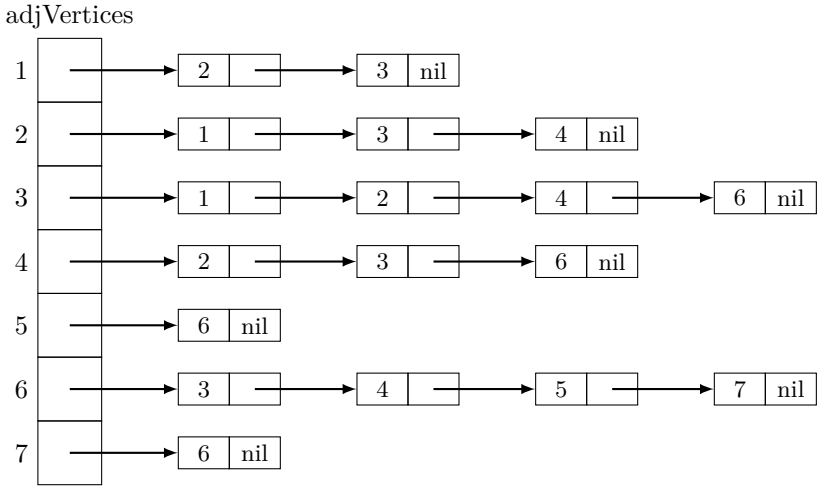
- **邻接矩阵表示**：用一个 $n \times n$ 的矩阵 $A = (a_{ij})$ 表示图 G，n 为顶点数
  - 无向图：$a_{ij} = \begin{cases} 1 & \text{if } v_i v_j \in E \\ 0 & \text{otherwise} \end{cases}$
  - 有向带权图：$a_{ij} = \begin{cases} W(v_i v_j) & \text{if } v_i v_j \in E \\ c & otherwise \end{cases}$
  - 其中 c 是一个常数，根据具体的应用环境而有所不同，例如权值代表某种开销的时候，一般取 $c = \infty$
  - 对无向图来说，其邻接矩阵是一个对称阵
- **邻接链表表示**：为图 G 的每个顶点生成一个链表，其中存储从该顶点出发的所有边的信息 (结束顶点、权值等)
  - 对无向图来说，每条边被记录了两次
  - 对有向图来说，每条边只被记录一次

无向图的邻接矩阵表示

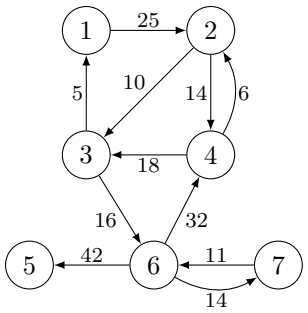


$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

无向图的邻接链表表示

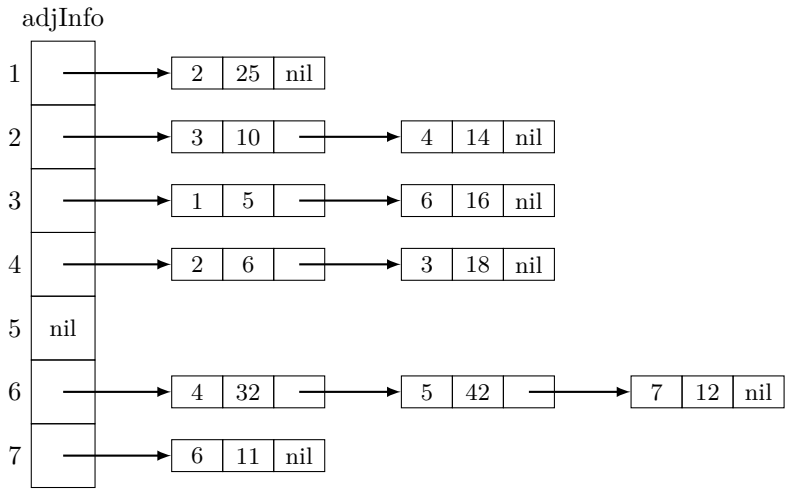


有向图的邻接矩阵表示



$$\begin{bmatrix} 0 & 25 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 10 & 14 & \infty & \infty & \infty \\ 5 & \infty & 0 & \infty & \infty & 16 & \infty \\ \infty & 6 & 18 & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 32 & 42 & 0 & 14 \\ \infty & \infty & \infty & \infty & \infty & 11 & 0 \end{bmatrix}$$

有向图的邻接链表表示



3 图的搜索与遍历

3.1 深度和广度优先搜索

深度优先搜索 (Depth-First Search)

- 图的深度优先搜索 (或遍历) 是一般的树遍历算法的推广
- 其要点是：不断向前探索 (explore)，无路可走的时候原路返回 (backtrack)
- 深度优先搜索算法概要：

Algorithm DFS(G, v)

```
1 标记 v 为“发现”;
2 foreach G 中与 v 有边相连的顶点 w do
3 |   if w 状态为“未发现” then
4 |   |   DFS(G, w);
5 |   else
6 |   |   “检查”边 vw 而不访问 w;
7 |   end
8 end
9 标记 v 为“结束”;
```

所有顶点的遍历

- 对于给定的图，从某一顶点出发做深度优先搜索并不是所有其他顶点都能到达
- 如果要遍历图 G 的所有顶点，则在深度优先搜索的基础上做一些额外的工作

Algorithm DFSSweep(G)

```
1 将 G 中所有顶点标记为“未发现”;
2 foreach v ∈ G do
3 |   if v 的状态为“未发现” then
4 |   |   DFS(v);
5 |   end
6 end
```

用深度优先搜索求连通分量

Algorithm ConnectedComponents$(adjVertices[], n)$

```
1 初始化 color[1], ··· , color[n] 为 white;
2 for v ← 1 to n do
3 |   if color[v] = white then CCDFS(adjVertices, color, v, v, cc) ;
4 end
5 return cc;
```

Procedure CCDFS$(adjVertices[], color[], v, ccNum, cc[])$

```
1 color[v] ← gray; cc[v] ← ccNum; remAdj ← adjVertices[v];
2 while remAdj ≠ null do
3 |   w ← First(remAdj);
4 |   if color[w] = white then CCDFS(adjVertices, color, w, ccNum, cc) ;
5 |   remAdj ← Rest(remAdj);
6 end
7 color[v] ← black;
```

时间复杂度：$\Theta(n+m)$；　空间复杂度：$\Theta(n)$

广度优先搜索 (Breadth-First Search)

- 广度优先搜索和深度优先搜索的“勇往直前”不同，它更像是有很多人同时从某一顶点出发，分别沿从该顶点出发的每条边向前搜索
- 一个非递归的广度优先搜索算法：

Algorithm `BreadthFirstSearch`$(adjVertices[], n, s)$

```
1  初始化 color[1], ··· , color[n] 为 white; Queue pending ← Create(n);
2  parent[s] ← −1; color[s] ← gray; Enqueue(pending, s);
3  while pending 非空 do
4  |  v ← Front(pending); Dequeue(pending);
5  |  foreach 链表 adjVertices[v] 中的顶点 w do
6  |  |  if color[w] =white then
7  |  |  |  color[w] ← gray; Enqueue(pending, w); parent[w] ← v;
8  |  |  end
9  |  end
10 |  color[v] ← black;
11 end
12 return parent;
```

3.2 有向图的深度优先搜索

深度优先搜索树 (Depth-First Search Tree)

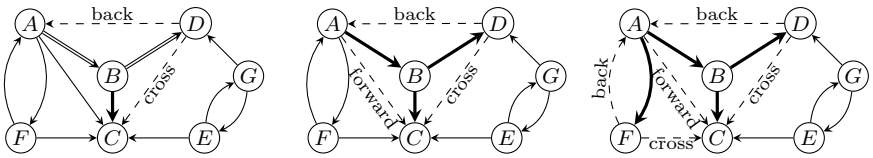
Definition 3.1 (深度优先搜索树和森林). 在深度优先搜索过程中连接处于“未发现”状态的顶点的边构成一棵以出发顶点为根的树，称为**深度优先搜索树**，或称**深度优先生成树** (depth-first spanning tree)。若从出发顶点不能到达图 G 中所有顶点，则对图 G 的完全深度优先遍历将把顶点分割成几棵互不相交的深度优先搜索树，称为**深度优先搜索森林** (depth-first search forest)，或称**深度优先生成森林** (depth-first spanning forest)。

Definition 3.2 (祖先 (ancestor)). 在图 G 的深度优先搜索树中，若顶点 v 在从根到顶点 w 的路径上，则称顶点 v 是顶点 w 的**祖先**，若同时满足 $v \neq w$，则称 v 是 w 的**严格祖先** (proper ancestor)，相应的，称 w 为 v 的**子孙** (descendant)。

有向图边的分类

Definition 3.3. 有向图 G 的边可以根据它们被访问的方式分成如下几类：

1. 若当 vw 被访问时 w 状态为“未发现”，则称 vw 为**树边** (tree edge)，v 成为 w 的父结点；
2. 若 w 是 v 的祖先，则称 vw 为**返回边** (back edge) (包括 vv)；
3. 若 w 是 v 的子孙，但在访问 vw 时，w 已经被访问过 (即状态为“结束”)，则称 vw 为**子孙边** (descendant edge) (或称**前向边** (forward edge))；
4. 若 w 和 v 之间无祖先或子孙关系，则称 vw 为**交叉边** (cross edge)



深度优先搜索算法框架

Algorithm DFSSweep($adjVertices[], n, \cdots$)

```
1  将状态颜色数组 color 中的元素初始化为 white;
2  for v ← 1 to n do
3  |  if color[v] = white then
4  |  |  vAns ← DFS(adjVertices, color, v, ···);
5  |  |  处理 vAns;
6  |  end
7  end
8  return ans;
```

DFS 算法框架

Algorithm DFS($adjVertices[], color[], v, \cdots$)

```
1  color[v] ← gray;
2  处理顶点 v (preorder processing);
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5  |  w ← First(remAdj);
6  |  if color[w] = white then
7  |  |  处理边 vw (tree edge);
8  |  |  wAns ← DFS(adjVertices, color, w, ···);
9  |  |  回溯处理边 vw (使用 wAns);
10 |  else 检查边 vw (nontree edge) ;
11 |  remAdj ← Rest(remAdj);
12 end
13 处理顶点 v (postorder processing)，计算结果 ans;
14 color[v] ← black;
15 return ans;
```

深度优先搜索结构

- 在一些深度优先搜索的应用中，经常需要知道这样一些信息：
  - 从深度优先搜索树的根到当前访问顶点的路径上有哪些顶点 (已经被访问过的灰色顶点)；
  - 顶点第一次被访问和最后一次被访问的时间顺序关系
- 这些信息可以通过在搜索过程中将当前时间记录在 $discoverTime$ 和 $finishTime$ 两个数组中得到：
  - 顶点 v 颜色为 white：顶点尚未被发现
  - 当顶点 v 颜色变为 gray：当前时间记录在 $discoverTime[v]$
  - 当顶点 v 颜色变为 black：当前时间记录在 $finishTime[v]$
  - 若时间用递增的整数表示，则顶点 v 的活动区间 (active interval) 为

$$active(v) = discoverTime[v], \cdots, finishTime[v]$$

  - 在顶点 v 的活动区间中，其颜色为 gray

深度优先搜索跟踪

- 对深度优先搜索算法框架稍作修改即可得到深度优先搜索跟踪算法，用于计算 $discoverTime$ 和 $finishTime$ 数组

Algorithm DFSTrace($adjVertices[], n$)

```
1 将 color 中的元素初始化为 white;
2 global time ← 0;
3 global discoverTime, finishTime, parent;
4 for v ← 1 to n do
5 |   if color[v] = white then
6 |   |   parent[v] ← −1;
7 |   |   TDFS(adjVertices, color, v);
8 |   end
9 end
```

TDFS

Procedure TDFS($adjVertices[], color[], v$)

```
1  color[v] ← gray;
2  time ← time + 1;  discoverTime[v] ← time;
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5  |   w ← First(remAdj);
6  |   if color[w] = white then
7  |   |   parent[w] ← v;
8  |   |   TDFS(adjVertices, color, w);
9  |   end
10 |   remAdj ← Rest(remAdj);
11 end
12 time ← time + 1;  finishTime[v] ← time;
13 color[v] ← black;
```

一些结论

Theorem 3.4. 根据前面对 $active(v)$ 和边分类的定义，算法 *DFSTrace* 作用于有向图 $G = (V, E)$ 后，对于任意顶点 $v \in V$ 和 $w \in V$，有如下结论成立：

1. 在 *DFS* 森林中，w 是 v 的子孙当且仅当 $active(w) \subseteq active(v)$，若 $v \neq w$，则 $active(w) \subset active(v)$
2. 若在 *DFS* 森林中，w 和 v 无祖先/子孙关系，则它们的活动区间 $active(v)$ 和 $active(w)$ 不相交
3. 若边 $vw \in E$，则：
   (a) vw 是交叉边当且仅当 $active(w)$ 整个区间在 $active(v)$ 之前
   (b) vw 是子孙边当且仅当存在第三个顶点 x，使得 $active(w) \subset active(x) \subset active(v)$
   (c) vw 是树边当且仅当 $active(w) \subset active(v)$，并且不存在第三个顶点 x，使得 $active(w) \subset active(x) \subset active(v)$
   (d) vw 是返回边当且仅当 $active(v) \subset active(w)$

3.3 有向无环图

基本概念

- 有向无环图 (DAG) 是一种非常重要的有向图：

  - 很多实际问题可以用 DAG 来描述，例如任务调度问题 (scheduling problem)，某项任务的开始依赖于其他任务的结束，如果任务的依赖关系中存在环路，则意味着发生了死锁 (deadlock)
  - 很多实际问题用 DAG 解决比用一般有向图解决来得更简单和有效，很多时候可以使时间复杂度从指数级别降到线性级别
  - 在数学上，一个 DAG 对应于顶点间的某种偏序关系 (partial order relation)

- 对于给定 DAG，可以给出其顶点的拓扑顺序 (topological order)：

  Definition 3.5. <2->[拓扑顺序] $G = (V, E)$ 是一个包含 n 个顶点的有向图，图 G 的**拓扑顺序**就是给 V 中的顶点赋予 **拓扑数** (topological number) $1, \cdots, n$ 来表示顶点顺序，使得对每一条边 $vw \in E$，顶点 v 的拓扑数小于顶点 w 的拓扑数；若顶点 v 的拓扑数大于顶点 w 的拓扑数则称为 **逆拓扑顺序** (reverse topological order)。

逆拓扑排序

Lemma 3.6. 若有向图 G 中存在环路，则 G 无拓扑顺序。

- 拓扑排序是 DAG 的基本问题，任意给定 DAG 至少有一种拓扑顺序
- 若解决了拓扑排序问题，很多实际问题便迎刃而解 (recall：递归算法的子问题图)
- 通过简单修改前面给出的深度优先搜索算法框架便可以得到一个逆拓扑排序算法：

Algorithm `ReverseTopoOrdering`$(adjVertices[], n)$

```
1 将 color 中的元素初始化为 white; topo ← new int[n + 1];
2 global topoNum ← 0;
3 for v ← 1 to n do
4 |   if color[v] = white then RTODFS(adjVertices, color, v, topo) ;
5 end
6 return topo;
```

`RTODFS`

Procedure `RTODFS`$(adjVertices[], color[], v, topo[])$

```
1  color[v] ← gray;
2  remAdj ← adjVertices[v];
3  while remAdj ≠ null do
4  |   w ← First(remAdj);
5  |   if color[w] = white then RTODFS(adjVertices, color, w, topo) ;
6  |   remAdj ← Rest(remAdj);
7  end
8  topoNum ← topoNum + 1; topo[v] ← topoNum;
9  color[v] ← black;
10 return ans;
```

Theorem 3.7. 图 G 是给定包含 n 个顶点的 DAG，算法 `ReverseTopoOrdering` 在返回的 $topo$ 数组中给出了 G 的一个逆拓扑顺序，因而任意一个 DAG 至少有一个逆拓扑顺序和一个拓扑顺序。

证明要点

1. `RTODFS` 访问每个顶点一次 (状态从 gray 到 black)，第 8 行执行了 n 次，则数组 $topo$ 中保存了从 1 到 n 不同整数；
2. 验证 $\forall vw \in E$, $topo[v] > topo[w]$：

- vw 不可能是返回边，否则表明图中存在环路，与 DAG 矛盾；
- vw 是除返回边以外的其他类型边，则当 v 颜色变为 $black$ 时，w 颜色已经变成 $black$，即 $topo[w]$ 赋值早于 $topo[v]$，而 $topoNum$ 只增不减，因此 $topo[v] > topo[w]$。

关键路径 (Critical Path) 分析

- 关键路径分析是与拓扑排序有关的优化问题，其优化目标是找到给定 DAG 中的最长路径

Definition 3.8 (任务调度问题). 设某项目由 n 个任务组成，分别标记为 $1, \cdots, n$，每一个任务都有一个依赖关系列表，表明该任务的开始直接依赖于哪些任务的结束，则：

- 某个任务 v 的**最早开始时间** (earliest start time, est) 是指：
  1. v 与其他任务无依赖关系：$est = 0$
  2. v 与其他任务有依赖关系：est 是其所有依赖任务最早结束时间的最大值
- 某个任务的**最早结束时间** (earliest finish time, eft) 是其最早开始时间加上其任务本身的执行时间
- 该项目的**关键路径**是一个任务序列 $v_0, v_1, \cdots, v_k$，且满足：
  1. v_0 与其他任务无依赖关系
  2. 每个 v_i 直接依赖于 v_{i-1}，即 v_i 的 est 等于 v_{i-1} 的 eft
  3. v_k 的最早结束时间是项目的所有任务最早结束时间中的最大值

求关键路径

- 关键路径的意义：若要提前完成项目，必须缩短关键路径上任务的执行时间，仅缩短非关键路径任务的执行时间对整个项目来说是没有价值的
- 同样通过修改 DFS 算法框架可以得到关键路径算法：

Algorithm `CriticalPath`($adjVertices[], n, duration[]$)

```
1 将 color 中的元素初始化为 white;
2 global critDep, eft;
3 for v ← 1 to n do
4 |   if color[v] = white then
5 |   |   CPDFS(adjVertices, color, v, duration);
6 |   end
7 end
```

`CPDFS`

Procedure CPDFS($adjVertices[], color[], v, duration[]$)

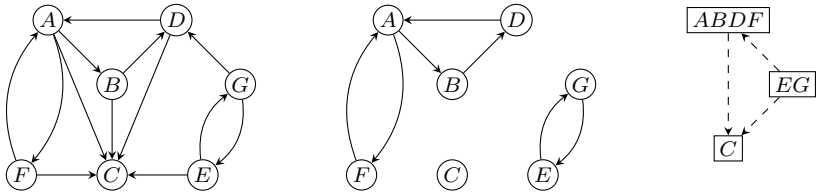
```
1  color[v] ← gray;
2  est ← 0; critDep[v] ← −1;
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5  |  w ← First(remAdj);
6  |  if color[w] = white then
7  |  |  CPDFS(adjVertices, color, w, duration);
8  |  end
9  |  if eft[w] ≥ est then  est ← eft[w]; critDep[v] ← w ;
10 |  remAdj ← Rest(remAdj);
11 end
12 eft[v] ← est + duration[v];
13 color[v] ← black;
```

3.4 有向图的强连通分量

有向图的凝聚图 (Condensation Graph)

Definition 3.9 (凝聚图). 设 $S_1, S_2, \cdots, S_p$ 是有向图 $G = (V, E)$ 的强连通分量，则 G 的**凝聚图**是 有向无环图 $G\downarrow = (V', E')$，其中 V' 有 p 个顶点 $s_1, \cdots, s_p$，$s_i s_j \in E'$ 当且仅当 $i \neq j$ 且在 E 中存在从 S_i 中的某一顶点到 S_j 中某一顶点的边，即将强连通分量 S_i 凝聚为一个顶点 s_i。

Example 3.1.



强连通分量的性质

Definition 3.10 (转置图 (transpose graph)). 将有向图 G 的所有边方向反转得到的图称为 G 的**转置图**，用 G^T 表示。显然，$(G\downarrow)^T = (G^T)\downarrow$。

Definition 3.11 (强连通分量的领头顶点 (leader)). 给定有向图 G 的强连通分量 S_i，$i = 1, \cdots, p$，对 G 进行深度优先搜索的过程中所发现的属于 S_i 的第一个顶点称为强连通分量 S_i 的**领头顶点**，记作 v_i。

Lemma 3.12. 有向图 G 的深度优先搜索森林中的每一棵树均包含一个或多个完整的强连通分量，且不包含“部分”强连通分量。

Corollary 3.13. 在深度优先搜索过程中，强连通分量 S_i 的所有顶点中领头顶点 v_i 最后一个结束 *(颜色变为 black)*。

Lemma 3.14. 在深度优先搜索过程中，当某个领头顶点 v_i 被发现时，不存在从 v_i 到其他灰色 *(gray)* 顶点的路径。

Lemma 3.15. 若 v 是强连通分量 S 的领头顶点，x 是另外一个强连通分量中的顶点，并且存在一条从 v 到 x 的路径，则在深度优先搜索过程中，当 v 被发现时，x 要么是黑色 *(black)* 顶点，要么存在一条从 v 到 x 的由白色 *(white)* 顶点 *(包括 x)* 构成的路径，且在任何情况下 v 比 x 更晚结束

强连通分量算法

- 深度优先搜索可以用于求有向图 G 的所有强连通分量，具体算法可分为如下两个阶段：
  1. 对 G 进行标准深度优先搜索，在每个顶点的结束时间将其压入堆栈
  2. 对图 G 的转置图 G^T 进行深度优先搜索，但在主循环中根据退栈顺序开始某个顶点 (white) 的搜索，而不再根据 $adjVertices$ 数组中的自然顺序；在搜索过程中，将每个顶点所属的强连通分量的领头顶点存储在数组 scc 中
- 算法描述 $\Rightarrow$ 课后练习
- 算法的正确性：

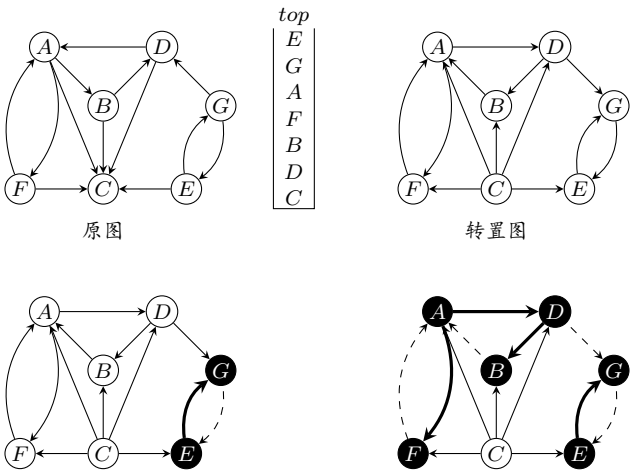
  Lemma 3.16. *在上述算法的阶段 2，每次从栈中弹出的白色 (white) 顶点一定是阶段 1 中某个强连通分量的领头顶点。*

强连通分量算法 (cont.)

Theorem 3.17. *在上述算法的阶段 2，每一棵深度优先搜索树包含且仅包含一个强连通分量的所有顶点。*

Proof. 由前面的引理可知每棵深度优先搜索树包含一个或多个强连通分量，且不包含部分强连通分量，因此我们只要证明在阶段 2 中每棵深度优先搜索树仅包含一个强连通分量。假设 v_i 是阶段 1 中强连通分量 S_i 的领头顶点，则根据引理，它是 S_i 中最后被压入堆栈的顶点，则在阶段 2 中当 v_i 从栈中弹出为白色顶点时，v_i 是阶段 2 深度优先搜索树的根，假设有另外一个强连通分量 S_j 其领头顶点为 v_j，在 G^T 中存在一条路径从 v_i 到达 S_j 中某一顶点，即可到达 v_j，则在图 G 中存在 v_j 到 v_i 的路径，根据前面的引理，v_j 在阶段 1 中比 v_i 更晚结束，因此在阶段 2 中更早从栈中弹出，当 v_i 被弹出时，S_j 中所有顶点均已变为 black，所以从 v_i 开始的搜索不可能再脱离强连通分量 S_i。 □

例子



课后习题

Exercise (8). 给定有向图 G：

(1) 证明图 G 的凝聚图 $G\downarrow$ 是有向无环图。

(2) 若图 G 以邻接表的形式存储，试写出一个算法求图 G 的转置图 G^T。

deadline: 2019.01.19

3.5 无向图的深度优先搜索

无向图的深度优先搜索算法框架

- 一般情况下，无向图的深度优先搜索可以采用有向图的搜索算法，实际上无向图在存储结构上等价于一个对称有向图 (每条边裂为两条方向相反的边)
- 在某些应用环境下需要保证每条边在深度优先搜索过程中仅访问一次，在这种情况下需要忽略等价对称有向图搜索过程中出现的某些边，这些边都是对应无向图搜索中第二次遇到的边，包括：
  - 从 v 到 p 的返回边 (back edge)，其中 pv 为树边
  - 从 v 到 w 的前向边 (forward edge)，此时 w 必为 black
  - 在对称有向图的深度优先搜索中不可能出现交叉边 (cross edge)
- 算法框架 `UDFSSweep` 与 `DFSSweep` 基本相同。

`UDFS` 算法框架

Algorithm `UDFS`$(adjVertices[], color[], v, p, \cdots)$

```
1  color[v] ← gray;
2  处理顶点 v (preorder processing);
3  remAdj ← adjVertices[v];
4  while remAdj ≠ null do
5  |  w ← First(remAdj);
6  |  if color[w] = white then
7  |  |  处理边 vw (tree edge);
8  |  |  wAns ← UDFS(adjVertices, color, w, v, ···);
9  |  |  回溯处理边 vw (使用 wAns);
10 |  else if color[w] = gray and w ≠ p then  检查边 vw (back edge) ;
11 |  remAdj ← Rest(remAdj);
12 end
13 处理顶点 v (postorder processing)，计算结果 ans;
14 color[v] ← black;
15 return ans;
```

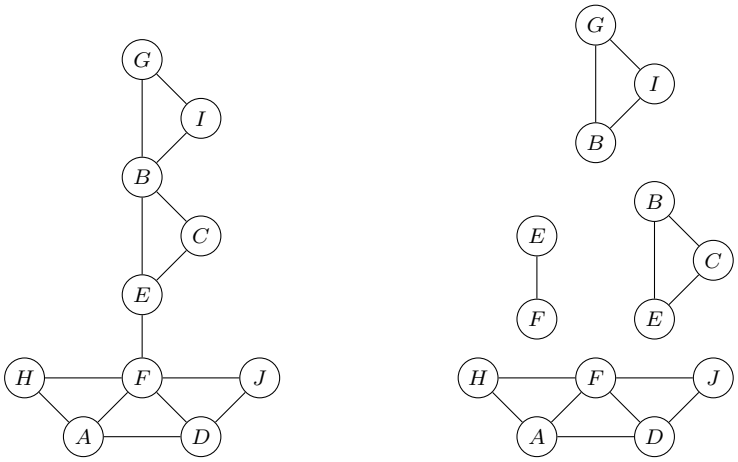
重连通分量

- 一些和无向图相关的实际问题：
  - 如果某个城市的机场由于天气原因而关闭，则其他任意两个城市之间是否仍有航线可以连接？
  - 若网络上某台路由器出现故障，该网络中任意两台计算机之间是否仍然可以通信？
- 问题的抽象：从一个给定连通图中去掉任一顶点 (包括和其连接的边)，剩余的子图是否仍然是连通图？

Definition 3.18 (重连通分量 (biconnected component))**.** 若从连通无向图 G 中移除任一顶点以及和该顶点连接的边后剩下的子图仍然是连通图，则称 G 是**重连通的** (biconnected)。无向图 G 的**重连通分量**是指 G 的最大重连通子图。

Definition 3.19 (关节点 (articulation point))**.** 若无向图 G 中存在两个不同顶点 w 和 x 之间的所有路径均通过不同于这两个顶点的第三个顶点 v，则称 v 为**关节点**。

例子



重连通分量算法设计

- 显然，一个无向连通图是重连通图当且仅当该图中不存在关节点
- 在深度优先搜索过程中，当从顶点 w 回溯到顶点 v 时，若以 w 为根的深度优先搜索子树中任一顶点都不存在到 v 的祖先的返回边 (back edge)，则意味着 v 一定在从深度优先搜索树的根到 w 的每一条路径上，因此 v 是一个关节点
- 以 w 为根的子树加上所有从其中出发的返回边以及边 vw 可以在顶点 v 与原图分离，但可能包含不止一个重连通分量，如果在搜索过程中一旦发现关节点即分离重连通分量，则可保证分离顺序由叶结点向根结点接近，从而使每次分离的都是单个的重连通分量
- 可以在深度优先搜索的递归过程中计算并返回一个返回时间 $back$,若从当前顶点出发通过递归调用搜索的子树返回的返回时间不小于当前顶点的发现时间 $(discoverTime[v])$，则表明当前顶点是一个关节点

算法正确性保证

Theorem 3.20. 在深度优先搜索树中，非根顶点 v 是关节点当且仅当 v 不是叶结点并且存在某棵 v 的子树没有返回边连接到 v 的祖先结点 *(*不包括 v 本身*)*。

Proof. (1) 若 v 为关节点，则存在其他两个不同顶点 x 和 y，连接 x 和 y 的每条路径都经过 v，则 x 和 y 中至少有一个是 v 的子孙，否则 x 和 y 之间一定存在某条路径不经过 v，因此 v 不是叶结点。下面假设 v 的每一棵子树都有一条连接到 v 的祖先的返回边，有两种情况：x 和 y 中只有一个是 v 的子孙，则该子孙结点可通过返回边跳过 v 到达另一个顶点，与前提矛盾；x 和 y 都是 v 的子孙，则二者不可能在同一棵子树中，而在不同子树中则可以通过各自的返回边跳过 v 相互连接，同样与前提矛盾。因此肯定存在某棵子树没有返回边连接到 v 的祖先。 (2) 若 v 不是叶结点并且存在某棵 v 的子树没有返回边连接到 v 的祖先结点，则取 x 为 v 的某个祖先，y 为没有通向 v 祖先结点的返回边的子树中的某一顶点，则显然 x 和 y 之间的每条路径都经过 v。 □

重连通分量算法

- 根据上述分析，通过修改无向图的深度优先搜索算法框架可以得到重连通分量算法
- 在分离重连通分量时，所有边的输出可以通过一个栈结构来实现

Algorithm `Bicomponents`$(adjVertices[], n)$

```
1 global time ← 0;
2 global edgeStack ← CreateStack();
3 初始化 color[1], ··· , color[n] 为 white;
4 for v ← 1 to n do
5 |   if color[v] = white then
6 |   |   BicompDFS(adjVertices, color, v, −1);
7 |   end
8 end
```

`BicompDFS`

Algorithm `BicompDFS`$(adjVertices[], color[], v, p)$

```
1  color[v] ← gray; time ← time + 1; back ← discoverTime[v] ← time;
2  remAdj ← adjVertices[v];
3  while remAdj ≠ null do
4  |   w ← First(remAdj);
5  |   if color[w] = white then
6  |   |   Push(edgeStack, vw);
7  |   |   wBack ← BicompDFS(adjVertices, color, w, v);
8  |   |   if wBack ≥ discoverTime[v] then
9  |   |   |   从 edgeStack 中弹出边直至弹出 vw，输出一个重连通分量;
10 |   |   end
11 |   |   back ← min(back, wBack);
12 |   else if color[w] = gray and w ≠ p then
13 |   |   Push(edgeStack, vw);
14 |   |   back ← min(back, discoverTime[w]);
15 |   end
16 |   remAdj ← Rest(remAdj);
17 end
18 time ← time + 1; finishTime[v] ← time; color[v] ← black;
19 return back;
```

重连通分量算法分析

- 时间复杂度：
  - `Bicomponents` 的初始化工作开销为 $\Theta(n)$
  - 无向图的深度优先搜索算法框架复杂度为 $\Theta(n+m)$ (每个顶点处理 2 次，每条边处理 1 次)
  - 重连通分量的输出部分虽然每次开销可能不同，但总的来说每条边仅被压栈和退栈一次，因此总的开销为 $\Theta(m)$
  - 因此重连通分量算法的时间复杂度为 $\Theta(n+m)$
- 问题的推广：
  - k-连通分量：任意顶点之间存在 k 条不相交路径
  - 一个深度优先搜索求 3-连通分量的算法，参考：J. E. Hopcroft and R. E. Tarjan. Dividing A Graph into Triconnected Components. *SIAM Journal on Computing*, 2(3):135–157, 1973

4 最小生成树问题

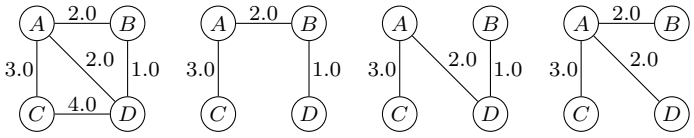
4.1 基本问题

问题描述

- 现实问题：现有一系列的车站、工厂、网络中的计算机等，如何用最小的代价将它们各自连接起来，使其两两之间都可到达？

Definition 4.1 (最小生成树 (minimum spanning tree))**.** 一个连通无向图 $G = (V, E)$ 的**生成树** (spanning tree) 是一棵树，并且是由 G 的所有顶点构成的子图；在带权图 $G = (V, E, W)$ 中，子图的权值是其所包含的所有边的权值之和，则带权连通无向图 G 的 **最小生成树** (MST) 就是图 G 权值最小的一棵生成树。

Example 4.1.



4.2 Prim 算法

基本思路

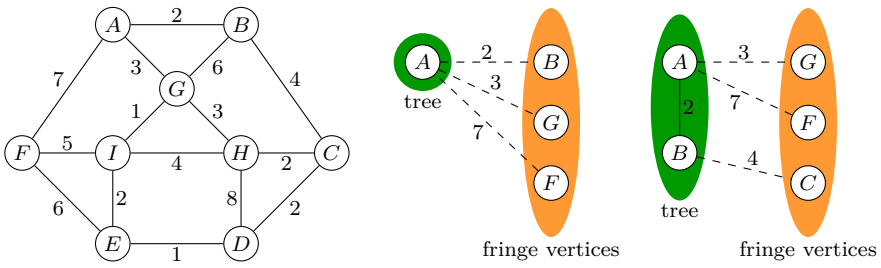
- 能否利用深度优先搜索？
- 贪心方法尝试：
  - 每次选择与当前子图相连 (仅有一个顶点在子图中) 的权值最小的边加入子图
  - 顶点类型：
    - tree　已经包含在构造的生成树中
    - fringe　不在树中，但与树中某一顶点相连
    - unseen　除上面两种情况外的其他顶点
- Prim 算法 (Robert C. Prim, 1957)：

Algorithm `PrimMST`(G, n)

```
初始化所有顶点为 unseen;
选择任一顶点 s 开始构造生成树，将其类型改为 tree;
将与 s 相连的所有顶点类型改为 fringe;
while 还有类型为 fringe 的顶点 do
    从中选出一个所在边权值最小的顶点 v;
    将其所在边加入生成树，并将 v 的类型修改为 tree;
    将所有与 v 相连的类型为 unseen 的顶点类型改为 fringe;
end
```

算法演示

Example 4.2.



- 问题：
  - 算法的正确性：PrimMST 产生的是不是最小生成树？
  - 算法的有效性：PrimMST 的时间复杂度如何？（与实现相关）

算法的正确性

Definition 4.2 (最小生成树性质). 设 T 是带权无向连通图 $G=(V,E,W)$ 的一棵生成树，若将 G 中一条不在 T 中的边 uv 加入 T 则 T 中将出现一条环路，如果对任意一条这样的边 uv，其权值是产生的环路所包含的边中最大的，则称生成树 T 具有**最小生成树性质** (minimum spanning tree property)。

Lemma 4.3. 给定带权无向连通图 $G=(V,E,W)$，若 T_1 和 T_2 是 G 的两棵具有最小生成树性质的生成树，则 T_1 和 T_2 权值相等。

算法的正确性 (cont.)

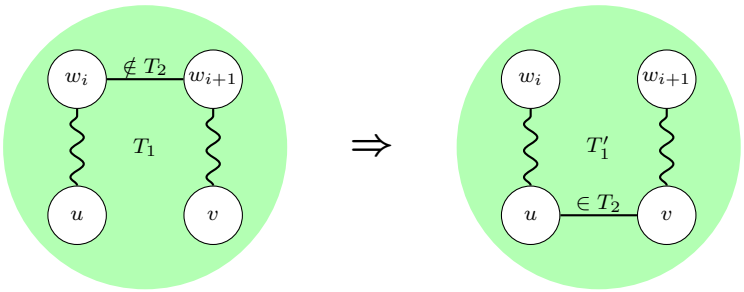
Proof. 用数学归纳法证明。对在 T_1 中而不在 T_2 中的边的数量 k 归纳。

奠基：$k=0$，这时 T_1 和 T_2 完全相同，结论成立；

归纳：假设在 T_1 中而不在 T_2 中的边的数量 $<k$ 时结论成立，则边数 $=k$ 时，设 uv 是只在 T_1 或只在 T_2 的边中权值最小的边，不失一般性，假设 $uv\in T_2$，考虑 T_1 中从 u 到 v 的路径 $w_0,w_1,\cdots,w_p$，其中 $w_0=u,w_p=v$ 且 $p\geq 2$，在这条路径中必然存在某条边不在 T_2 中，否则将在 T_2 中构成环路，设 $w_iw_{i+1}\notin T_2$，由于 T_1 具有最小生成树性质，因此 $W(w_iw_{i+1})\leq W(uv)$，又 uv 是所有不同边中权值最小的边，因此 $W(w_iw_{i+1})\geq W(uv)$，因此 $W(w_iw_{i+1})=W(uv)$，将 uv 加入 T_1 同时将 w_iw_{i+1} 移除，则可保证产生的新树 T_1' 仍是一棵生成树，且权值与 T_1 相等，而 T_1' 与 T_2 只相差 $k-1$ 条边，根据归纳假设它们的权值相等，因此 T_1 和 T_2 的权值也相等。 □

算法的正确性 (cont.)

- 示意图



算法的正确性 (cont.)

Theorem 4.4. 带权无向连通图 $G=(V,E,W)$ 的生成树 T 是最小生成树，当且仅当 T 具有最小生成树性质。

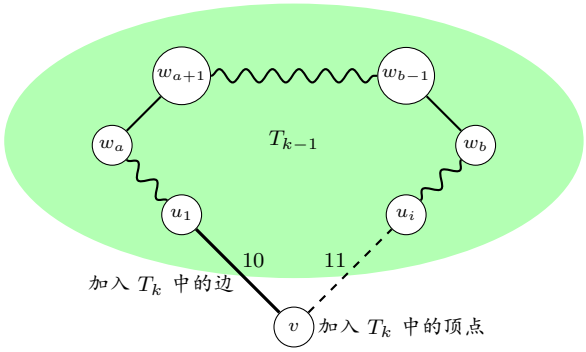
Lemma 4.5. 给定带权无向连通图 $G=(V,E,W)$，$|V|=n$，若 T_k 是 PrimMST 生成的包含 k 个顶点的树，$k=1,\cdots,n$，G_k 是由这 k 个顶点及其相连的边构成的 G 的子图，则 T_k 在 G_k 中具有最小生成树性质。

Proof. **提示**：用数学归纳法证明。在归纳步骤只需证明按 PrimMST 的步骤加入的边仍然能够保持新的树 T_k 具有最小生成树性质 (反证法)。 □

Theorem 4.6. 算法 PrimMST 的输出是其输入图的最小生成树。

证明示意

- 在归纳步骤加入 T_{k-1} 的顶点为 v，加入的边为 vu_1，而 T_{k-1} 中其他在原图中与 v 有边相连的顶点为 $u_2, \cdots, u_d$
- 证明 T_k 具有最小生成树性质，即证明对于任意 $xy \in G_k$ 而 $xy \notin T_k$，xy 是加入 T_k 构成的环路中权值最大的边：
  1. $x \neq v, y \neq v$：$xy \in G_{k-1}$ 且 $xy \notin T_{k-1}$，根据归纳假设结论成立
  2. xy 是 $u_2v, \cdots, u_dv$ 中的一条边：反证法



算法的有效性保证

- 用最小优先队列来存储和维护 fringe 顶点集合：在优先队列中每个 fringe 顶点只需记录与 tree 顶点相连的边中权值最小的那一条，称为**候选边** (candidate edge)，该边权值即为优先队列元素的权值

Algorithm `PrimMST`(G, n)

```
初始化所有顶点为 unseen，优先队列 pq 为空;
选择任一顶点 s 设置其候选边为 (−1, s, 0);
Insert(pq, s, 0);
while pq 非空 do
    v ← GetMin(pq);
    DeleteMin(pq);
    将 v 的候选边加入生成树，v 的类型修改为 tree;
    UpdateFringe(pq, G, v);
end
```

`UpdateFringe`

Procedure `UpdateFringe`(pq, G, v)

```
foreach 与 v 相连的顶点 w do
    newW ← W(v, w);
    if w 类型为 unseen then
        将 w 的候选边设置为 (v, w, newW);
        w 的类型修改为 fringe;
        Insert(pq, w, newW);
    else if w 类型为 fringe and newW < GetPriority(pq, w) then
        将 w 的候选边改为 (v, w, newW);
        DecreaseKey(pq, w, newW);
    end
end
```

$$T(n, m) \in O(nT(\texttt{GetMin}) + nT(\texttt{DeleteMin}) + mT(\texttt{DecreaseKey}))$$

Prim 算法时间复杂度分析

- 时间复杂度与优先队列的实现有关，但对一般的连通图总是有 $m > n$，因此总的来说 `DecreaseKey` 的复杂度更值得关注
- 当 Prim 设计这个算法时还没有出现优先队列的概念，因此只是简单地用数组实现
  - `GetMin` 和 `DeleteMin`: $\Theta(n)$
  - `DecreaseKey`: $\Theta(1)$
  - 因此 `PrimMST`: $\Theta(n^2)$
- 用二叉堆实现？($\Theta(m \log n) \Rightarrow O(n^2 \log n)$)
- 用配对森林实现：
  - `DecreaseKey`: $\Theta(1)$
  - `GetMin` 和 `DeleteMin` 在最坏情况下时间复杂度为 $\Theta(n)$，复杂度与简单数组实现类似，但是显然在平均情况下其时间复杂度要好
  - 已知结果：two-pass pairing heaps，当 $m \in \Theta(n^{1+c}), c > 0$，`GetMin` 的分摊时间开销为 $\Theta(\log n)$，因此 `PrimMST` 复杂度为 $\Theta(m + n \log n) = \Theta(m)$
  - 参考：Michael L. Fredman. On the Efficiency of Pairing Heaps and Related Data Structures. *Journal of the ACM* 46(4): 473–501, 1999

4.3 Kruskal 算法

基本思路

- 一个更“贪心”的方法：每次从剩余边集中取出权值最小且与生成树中已有边不构成环路的边加入到生成树中，直到所有的边都处理完毕
- Kruskal **算法** (Joseph B. Kruskal, 1956)：

Algorithm `KruskalMST`(G, n)

```
1 R ← E;
2 F ← ∅;
3 while R 非空 do
4 |   从 R 中移出权值最小的边 vw;
5 |   if vw 在 F 中不构成环路 then F ← F ∪ {vw} ;
6 end
7 return F;
```

算法的正确性

Definition 4.7 (生成树集). 对于带权无向图 $G = (V, E, W)$，其**生成树集** (spanning tree collection) 是其每个连通分量的生成树所构成的集合；**最小生成树集**是权值之和最小的生成树集，即每个连通分量的最小生成树的集合。

- `KruskalMST` 执行完毕后，G 中的每一个顶点都在某一棵树中吗？(孤立顶点)
- 若 G 无孤立顶点，则 `KruskalMST` 的输出是 G 的生成树集吗？即：是否 F 中的每一棵树恰是 G 的一个连通分量的生成树？

Lemma 4.8. 给定森林 F (即任意一个非连通无向无环图)，若边 $e = vw$ 不在 F 中，则 e 和 F 中的某些边构成环路当且仅当 v, w 在 F 的同一个连通分量中。

Proof. ‘⇒’：假设 e 和 F 中的某些边构成环路 $vw_1, \cdots, w_p w (p \geq 1)$，但 v, w 在 F 的不同连通分量 C_1 和 C_2 中，则去掉边 e 后，C_1 和 C_2 仍保持连通，这与 C_1 和 C_2 是不同连通分量矛盾；

‘⇐’：显然成立。 □

算法的正确性 (cont.)

- 若假设 G 中的某个连通分量对应 F 中两棵以上的树，则必存在 G 中的某条边 vw 连接其中的两棵树，即 v 和 w 分属 F 中两个不同连通分量，因此，当算法在处理边 vw 时并未将其加入 F 中，这意味着若将 vw 加入当时产生的树 F' 将构成环路，由前面的引理可知 v 和 w 属于 F' 的同一个连通分量，这与 v 和 w 分属 F 中两个不同连通分量矛盾。由此可知 G 中一个连通分量仅对应 F 中一棵树。

Theorem 4.9. 设 $G = (V, E, W)$ 是带权无向图，$F \subseteq E$，若 F 属于 G 的某个最小生成树集，e 是 $E - F$ 中权值最小的边，并且 $F \cup \{e\}$ 不包含环路，则 $F \cup \{e\}$ 也属于 G 的某个最小生成树集。

Proof. 用反证法，假设 $F \cup \{e\}$ 不属于 G 的最小生成树集，则在 G 的最小生成树集中必然不包含边 e，这表示若将 e 加入 G 的某一最小生成树集 F'，必构成环路，并且该环路上必存在一条边权值大于等于 e 的权值，否则环路上的其他边在考察边 e 之前就已经加入到 F 中，这样 $F \cup \{e\}$ 中就会产生环路，现将这条边换成 e，产生的仍是 G 的生成树集，但其权值小于等于 F'，这与假设矛盾。 □

算法实现细节

- 最小权值边的选取：最小优先队列
- 环路判断：当且仅当 v 和 w 属于 F 的同一个连通分量，vw 加入 F 将构成环路 ⇒ 动态等价关系、合并-查找程序

Algorithm `KruskalMST`(G, n)

```
1  根据 G 的所有边及其权值构造最小优先队列 pq;
2  构造动态等价类集合 sets，G 的每个顶点初始化为一个等价类;
3  F ← ∅;
4  while not IsEmpty(pq) do
5  |  vw ← GetMin(pq); DeleteMin(pq);
6  |  vSet ← Find(sets, v); wSet ← Find(sets, w);
7  |  if vSet ≠ wSet then
8  |  |  F ← F ∪ {vw}; Union(sets, vSet, wSet);
9  |  end
10 end
11 return F;
```

算法分析

- 边优先队列构造：$\Theta(m)$
- 从优先队列中取出和删除所有边：$\Theta(m \log m)$
  - 实际上这个复杂度可以降到 $\Theta(n \log m)$，因为 F 中最多只能有 $n-1$ 条边
- 动态等价类 (连通分量) 的维护：`Find` 操作最多执行 $2m$ 次，`Union` 最多执行 $n-1$ 次，这部分时间开销至多为 $O((m+n)\lg^*(n))$ (使用 `WUnion` 和 `CFind`)
- 通常情况下 $m \geq n$，因此在最坏情况下，`KruskalMST` 时间复杂度为 $\Theta(m \log m)$，若所有边事先已按权值排好序，则复杂度可降到 $O(m \lg^*(n))$，接近线性
- 与 Prim 算法的比较：
  - Prim 算法：$\Theta(n^2)$，适合处理稠密图 (边较多)
  - Kruskal 算法：$\Theta(m \log m)$，适合处理稀疏图 (边较少)

5 单源最短路径问题

5.1 基本问题

问题描述

- 问题的由来：
  - 地图上两地点间的最佳行车路径
  - 网络中两台计算机间的最佳路由
- 问题的抽象：
  - 在带权图中查找两个给定顶点间权值和最小的路径 (最短路径)
  - 在最坏情况下，寻找两个给定顶点间的最短路径不会比寻找从一个给定顶点到其可达的所有其他顶点间的最短路径更容易
  - 后者被称为**单源最短路径问题**
- 解决思路：问题分割 (分治法？)

  Lemma 5.1 (最短路径性质 (shortest path property))**.** 在带权图 G 中，顶点 x 到顶点 z 的最短路径由 x 到 y 的路径 P 和 y 到 z 的路径 Q 组成，则 P 是 x 到 y 的一条最短路径，Q 是 y 到 z 的一条最短路径。注意：反之不成立！

问题描述 (cont.)

Definition 5.2. 设 P 是带权图 $G=(V,E,W)$ 中由 k 条边 $xv_1, v_1v_2, \cdots, v_{k-1}y$ 构成的一条非空路径，则路径 P 的**权值** (weight) 等于构成路径的 k 条边的权值之和，记为 $W(P)$；若 $x=y$ 则称 x 到 y 的路径为空路径，其权值为 0。若从 x 到 y 的路径中没有权值小于 $W(P)$ 的路径，则称 P 为**最短路径** (shortest path) 或**最小权值路径** (minimum-weight path)。

Definition 5.3 (单源最短路径问题)**. 单源最短路径问题** (single-source shortest path problem) 即对给定带权图 $G=(V,E,W)$ 和图中的某个顶点 s，找到从 s 到图中每个顶点 v 的最短路径。

- 用深度优先搜索可否解决？

5.2 Dijkstra 算法

基本思路

- 采用与 Prim 算法类似的贪心策略
- 但每次在 fringe 顶点中选择的是离出发顶点 s 最近的顶点，而不是像 Prim 算法那样离生成树最近的顶点
- Dijkstra **算法** (Edsger W. Dijkstra, 1959)

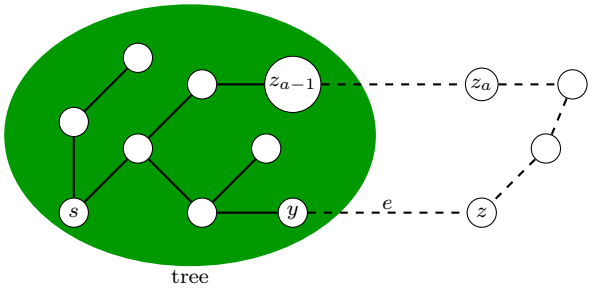
Algorithm `DijkstraSSSP`(G,n)

```
1  初始化所有顶点为 unseen;
2  选择任一顶点 s 开始构造生成树，将其类型改为 tree;
3  d(s,s) ← 0;
4  将与 s 相连的所有顶点类型改为 fringe;
5  while 还有类型为 fringe 的顶点 do
6  |  找到 fringe 顶点 v 及其相邻 tree 顶点 t 使 d(s,t)+W(tv) 最小;
7  |  将 tv 加入生成树，并将 v 的类型修改为 tree;
8  |  d(s,v) ← d(s,t)+W(tv);
9  |  将所有与 v 相连的类型为 unseen 的顶点类型改为 fringe;
10 end
```

Edsger W. Dijkstra：1930-2002，荷兰著名计算机科学家，其一身的贡献主要集中在程序设计和分布式计算理论领域，1972 年由于其在程序设计语言方面的杰出的基础性的贡献获得图灵奖，ACM Symposium on Principles of Distributed Computing 在他去世后将每年的 Influential Paper Award 更名为 Dijkstra Prize。另外，他还是结构化程序设计理论的创立者和倡导者，最早反对程序设计语言中 GOTO 语句的使用。

算法的正确性保证

Theorem 5.4. 设 $G = (V, E, W)$ 是一个权值非负的带权图，$V' \subseteq V, s \in V'$，若对每一个 $y \in V'$，$d(s, y)$ 是从 s 到 y 的最短距离，若 yz 是 $y \in V'$, $z \in V - V'$ 的所有边中使得 $d(s, y) + W(yz)$ 最小的边，则 s 到 y 的路径加上边 yz 构成 s 到 z 的最短路径。



算法的正确性保证 (cont.)

Proof. 设 $P = s, x_1, \cdots, x_r, y, z$，其中，$s, x_1, \cdots, x_r, y$ 是对应于 $d(s, y)$ 的从 s 到 y 的最短路径；假设 $P' = s, z_1, \cdots, z_a, \cdots, z$ 是从 s 到 z 的最短路径，z_a 是路径中第一个不在 V' 中的顶点，则：

$$W(P) = d(s, y) + W(yz) \leq d(s, z_{a-1}) + W(z_{a-1}z_a)$$

而根据前面的引理，$s, z_1, \cdots, z_{a-1}$ 是从 s 到 z_{a-1} 的最短路径，因此其权值为 $d(s, z_{a-1})$，又因为 $s, z_1, \cdots, z_{a-1}, z_a$ 是路径 P' 的一部分，且边的权值非负，所以：

$$d(s, z_{a-1}) + W(z_{a-1}z_a) \leq W(P')$$

所以，$W(P) \leq W(P')$。 □

算法实现

- 具体实现与 Prim 算法类似
- 复杂度分析完全一样

Algorithm `DijkstraSSSP`(G, n)

```
1 初始化所有顶点为 unseen，优先队列 pq 为空;
2 选择任一顶点 s 设置其候选边为 (−1, s, 0);
3 Insert(pq, s, 0);
4 while pq 非空 do
5 |   v ← GetMin(pq);
6 |   DeleteMin(pq);
7 |   将 v 的候选边加入生成树，v 的类型修改为 tree;
8 |   UpdateFringe(pq, G, v);
9 end
```

`UpdateFringe`

Procedure `UpdateFringe`(pq, G, v)

```
1  vDist ← GetPriority(pq, v);
2  foreach 与 v 相连的顶点 w do
3      wDist ← vDist + W(v, w);
4      if w 类型为 unseen then
5          将 w 的候选边设置为 (v, w, wDist);
6          w 的类型修改为 fringe;
7          Insert(pq, w, wDist);
8      else if w 的类型为 fringe and wDist < GetPriority(pq, w) then
9          将 w 的候选边改为 (v, w, wDist);
10         DecreaseKey(pq, w, wDist);
11     end
12 end
```

算法概论

第七讲：$\mathcal{NP}$-完全问题简介

薛健

Last Modified: 2019.1.12

主要内容

1 基本概念

基本问题

- 回顾：算法复杂度的渐近阶

	解题时间 (假定单位时间为微秒)			
输入规模 (n)	$33n$	$460n\lg n$	$13n^2$	2^n
10	0.00033sec.	0.015sec.	0.0013sec.	0.001sec.
100	0.0033sec.	0.3sec.	0.13sec.	4×10^{16}yr.
1,000	0.033sec.	4.5sec.	13sec.	
10,000	0.33sec.	61sec.	22min.	
100,000	3.3sec.	13min.	1.5days	

- 到目前为止我们所接触的问题都可以找到时间复杂度在 $O(n^c)$ 的算法来解决
- 是否存在这样的问题：解决该类问题的算法复杂度 (**问题的复杂度**) 在 $O(c^n)$ 或更高？

判定问题

- 涉及到上述疑问的一般都是优化问题 (optimization problem)，或更确切地讲是**组合优化问题** (combinatorial optimization problem)
- 这类问题可以重新描述成某个**判定问题** (decision problem)：有“是 (yes)”或“否 (no)”两种可能回答的问题，其表述一般包含两个部分
  1. **实例描述** (instance description) 部分：定义输入中所应包含的信息
  2. **问题** (question) 部分：提出需要判定“是”或“否”的问题，问题中包含实例描述中定义的变量
- 判定问题根据给定输入和问题的正确答案给出“是”或“否”的回答，因此判定问题可抽象为从所有可能输入的集合到 {yes, no} 的映射

问题举例

Definition 1.1 (图着色和着色数). 图 $G = (V, E)$ 的**着色** (coloring) 是指映射 $C : V \to S$，其中 S 是一个颜色的有限集合，并且如果 $vw \in E$，则 $C(v) \neq C(w)$，即：相邻顶点颜色不同；图 G 的**着色数** (chromatic number) 是对 G 进行着色所需的最少颜色数，记作 $\chi(G)$。

Problem 1.2 (图着色问题). 对给定无向图 $G = (V, E)$ 进行着色

优化问题：给定图 G，确定其着色数 $\chi(G)$，并给出对应的着色方案。

判定问题：给定图 G 和正整数 k，是否存在 G 的一种着色最多使用 k 种颜色？*(若存在，称 G 是 k-*__可着色的__ *$(k$-colorable$)$)*。

Problem 1.3 (哈密尔顿环路和哈密尔顿路径问题). **哈密尔顿环路/路径** *(Hamiltonian cycle / path)* 是无向图中通过所有顶点恰一次的简单环路/路径。

判定问题：给定无向图 G 中是否存在哈密尔顿环路 *(路径)*？

Problem 1.4 (货郎担问题). **货郎担问题** *(Traveling Salesperson Problem, TSP)* 也可以称作**最短周游问题** *(minimum tour problem)*，是著名的易于描述而难以解决的问题之一：某推销员要到其商品销售区域的各个城市推销商品，为节约开销提高效率，他需要从其所在地出发经过每个城市一次然后回到原所在地，且所花开销最少。

优化问题：给定带权完全图，寻找其中具有最小权值的哈密尔顿环路。

判定问题：给定带权完全图和数 k，图中是否存在一条权值小于等于 k 的哈密尔顿环路？

Problem 1.5 (子集和问题 (subset sum)). 给定一个正整数 C 和 n 个大小分别为正整数 $s_1, \cdots, s_n$ 的对象

优化问题：从 n 个对象的全集中选出包含对象数最多的一个子集，使得其中对象的和至多为 C。

判定问题：是否存在一个包含 k 个对象的子集，其中对象大小的和不大于 C？

Problem 1.6 (背包问题 (knapsack)). 有一个容量为 C 的背包和 n 个大小分别为 $s_1, \cdots, s_n$、价值分别为 $p_1, \cdots, p_n$ 的物品

优化问题：寻找总价值最大且能放进背包中的物品子集。

判定问题：给定正数 k，是否存在某个物品子集能够全部放入背包，且总价值不低于 k？

可满足性问题 (Satisfiability Problem)

- 一个命题 (布尔) 变量 (propositional (boolean) variable) 可被赋予 true 或 false 两种值，命题变量 v 的否定 $\bar{v}$ 值为 true 当且仅当 v 的值为 false
- 一个命题公式 (formula) 是由命题变量、命题常量或者由布尔运算符及其运算对象 (命题公式) 所构成的表达式
- **合取范式** (Conjunctive Normal Form, CNF) 是命题公式的一种常见表达方式，一个命题公式的合取范式由以布尔“与”操作符 ($\wedge$) 连接的一个子句序列构成，其中，每个**子句** (clause) 由以布尔“或”操作符 ($\vee$) 连接的一个命题变量 (或其否定变量) 序列构成。如：

$$(p \vee q \vee s) \wedge (\bar{q} \vee r) \wedge (\bar{p} \vee r) \wedge (\bar{r} \vee s) \wedge (\bar{p} \vee \bar{s} \vee \bar{q})$$

可满足性问题 (cont.)

- 对一个给定命题变量集合的每个元素赋予布尔值称为一个**真值指派** (truth assignment)，若对一个合取范式进行的真值指派使整个范式的值为 true，则称该真值指派**满足** (satisfy) 这个合取范式
- 显然，一个合取范式被满足当且仅当其每个子句为 true，而一个子句为 true 当且仅当至少一个命题变量 (或否定变量) 为 true

Problem 1.7 (可满足性问题). 给定一个合取范式，是否存在一个可满足该范式的真值指派？

▷ 该问题被称为**合取范式可满足性问题** *(CNF-satisfiability, CNF-SAT)* 或直接叫做**可满足性问题**，该问题在 $\mathcal{NP}$-完全问题的研究中扮演了非常重要的角色

2 $\mathcal{P}$ 问题和 $\mathcal{NP}$ 问题

$\mathcal{P}$ 问题

Definition 2.1 (以多项式为界 (polynomially bounded))**.** 以多项式为界的算法是指其最坏情况复杂度上界是其输入规模的多项式函数。

Definition 2.2 ($\mathcal{P}$ 问题)**.** $\mathcal{P}$ 问题是指以多项式为界的判定问题，即该问题可以在多项式时间内给出答案。

- 以多项式为判别界限的理由：
  - 并非所有的 $\mathcal{P}$ 问题有高效的算法来解决，但可以说不属于 $\mathcal{P}$ 类的问题必定时间复杂度较高甚至实际上无法解决
  - 多项式具有某种“闭包”性质，即多个多项式的加、乘、复合运算结果仍然是多项式，这意味着由多个多项式复杂度的简单算法合成的复杂算法，其复杂度仍然是以多项式为界的
  - 以多项式为界与具体的计算模型和实现环境无关

非确定算法

- 判定问题的输入和其一个可能的解决方案 (certificate) 可以用一个有限集中的符号所构成的串来描述，对于特定问题，每个符号的含义需要一组约定 (conventions) 来定义，这组约定称为问题的编码 (encoding)，则检查一个解决方案是否符合问题描述只需检查其对应符号串是否符合编码规则 (是否有意义) 和问题的判定准则。

Definition 2.3 (非确定算法 (nondeterministic algorithm))**.** 一个**非确定算法**包含两个阶段和一个输出步骤：

1. **“猜测** (guessing)**”阶段：**在给定存储区域写出一个完整符号串 s；
2. **确定“验证”** (deterministic “verifying”) **阶段：**执行一个确定的子过程，根据输入 *input* 和解决方案 s (或忽略 s) 返回结果 true 或 false (或进入无限循环)，即：检查 s 在当前输入下是否是判定问题的一个肯定 (yes) 答案
3. **输出步骤：**如果验证阶段返回结果为 true，则算法输出 yes，否则算法无输出

非确定算法 (cont.)

Algorithm `NondetA`(String *input*)

```
1 String s ← GetCertif();
2 checkOK ← VerifyA(input, s);
3 if checkOK then 输出 “yes” ;
```

- 算法的总执行步数为 `GetCertif` 和 `VerifyA` 的执行步数之和
- 对同一输入 x，上述算法执行结果可能不一样 (依赖于 s)
- 判定问题对输入 x 的回答为 yes 当且仅当上述算法某次执行 (针对某个 s) 给出了 yes 输出，否则回答为 no，即对任意 s，算法执行无输出结果

$\mathcal{NP}$ 问题

Definition 2.4. $\mathcal{NP}$ **问题**是指存在以多项式为界的非确定算法的判定问题。$\mathcal{NP}$ 取自 “**N**ondeterministic **P**olynomially bounded”。

Theorem 2.5. 图着色问题、哈密尔顿环路/路径问题、货郎担问题、子集和问题、背包问题、可满足性问题都是 $\mathcal{NP}$ 问题。

Theorem 2.6. $\mathcal{P} \subseteq \mathcal{NP}$

Proof. 所有 $\mathcal{P}$ 问题的确定算法 (deterministic algorithm，即普通意义上的算法) 都可以看作一种特殊的非确定算法，即将其作为非确定算法第二阶段的 `VerifyA`，若其输出为 yes，则 `VerifyA` 返回 true，否则返回 false，而第一阶段的 `GetCertif` 什么都不做，这样只要确定算法以多项式为界，则其对应的非确定算法也以多项式为界。 □

$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$

- $\mathcal{P} = \mathcal{NP}$ 还是 $\mathcal{P} \subset \mathcal{NP}$，即：是否有这样的问题，它存在以多项式为界的非确定算法，但无法在多项式时间内由确定算法解决
- 对于一个 $\mathcal{NP}$ 问题，如果我们取遍所有可能的解决方案 s 则可以对判定问题给出确定的 yes 或 no 回答，但可能的解决方案数目很大，若 `GetCertif` 给出的解决方案所对应的符号串长度至多为多项式 $p(n)$，而符号集包含的符号数为 c，则可能的解决方案数目将达到 $c^{p(n)}$！虽然有时候我们可以采用某些技巧使得不需要检查所有可能的解决方案就可以得到结果，但找到这样的技巧并不是一件容易的事情
- 因此，普遍认为 $\mathcal{NP}$ 应该是一个比 $\mathcal{P}$ 大得多的集合，但是迄今为止仍然没有找到一个 $\mathcal{NP}$ 问题可以证明其不是 $\mathcal{P}$ 问题，即存在大量的问题其多项式复杂度的算法未知，而已知的该问题的复杂度下界又不高于多项式
- $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ 仍然是一个开放的问题。

注意输入规模

- 要判定一个问题是否是 $\mathcal{P}$ 问题，需特别留心其输入规模
- 例如：给定一个正整数 n，判定其是否是质数，我们有一个非常简洁的算法：

Algorithm `IsPrime`(n)

```
1 r ← yes;
2 for i ← 2 to n − 1 do
3 |   if (n mod i) = 0 then r ← no; break;
4 end
5 return r;
```

- $(n \mod i)$ 可以在 $O(\log^2(n))$ 时间内完成，则该算法复杂度似乎不超过 $O(n^2)$
- 质数判定问题是否是 $\mathcal{P}$ 问题？
- 对大整数 n 其输入规模是 $h \in \Theta(\log_b n)$，该算法的复杂度为 b^h！

实际上质数判定问题是 $\mathcal{P}$ 问题，有学者在 2002 年提出了质数判定的多项式复杂度算法，即著名的 AKS 质数判定算法。

AKS 质数判定算法 (Agrawal-Kayal-Saxena primality test) 由三位印度坎普尔理工学院 (Indian Institute of Technology Kanpur) 的计算机科学家 Manindra Agrawal，Neeraj Kayal 和 Nitin Saxena 在 2002 年 8 月提出，发表在名为 “PRIMES is in $\mathcal{P}$ ” 的论文中，他们由于这项工作获得了很多奖，包括 2006 年的哥德尔奖 (Gödel Prize) 和 2006 年的 Fulkerson 奖 (Fulkerson Prize)。

再探欧几里德算法

Algorithm GCD(a, b)

```
while b ≠ 0 do
    t ← b;
    b ← a mod b;
    a ← t;
end
return a;
```

递归版本：

Algorithm GCD(a, b)

```
if b = 0 then return a ;
else return GCD(b, a mod b) ;
```

计算序列：

$$\begin{aligned} a &= q_0 b + r_0 \\ b &= q_1 r_0 + r_1 \\ r_0 &= q_2 r_1 + r_2 \\ r_1 &= q_3 r_2 + r_3 \\ &\vdots \end{aligned}$$

第 k 步计算：

$$r_{k-2} = q_k r_{k-1} + r_k$$

当 $r_N = 0$ 时停止：

$$\mathrm{GCD}(a, b) = r_{N-1}$$

时间复杂度分析*

- 循环步数：
  - 需要 N 次循环的最小自然数 a, b $(a > b > 0)$ 分别为斐波那契数 F_{N+2} 和 F_{N+1} (Gabriel Lamé 1844 年用数学归纳法证明，计算复杂性理论的发端，斐波那契数的第一个实际应用)
  - 当 b 足够大时，$b \geq F_{N+1} > F_N \approx \phi^N / \sqrt{5}$，$\phi$ 为黄金分割数 $\frac{1+\sqrt{5}}{2}$
  - $N < \log_\phi \sqrt{5} b \approx 4.785 \log_{10} b + 1.6723 < 5h$
- 单步开销：
  - 对大整数来说，用试商的方法求商和余数，其复杂度为 $O(hl)$，h 是除数的位数，l 是商的位数
  - 设 $r_0, r_1, \cdots, r_{N-1}$ 的位数分别为 $h_0, h_1, \cdots, h_{N-1}$
  - 第 k 步的开销为 $O(h_{k-1}(h_{k-2} - h_{k-1} + 1))$
  - N 步的开销总和为

$$\begin{aligned} O(\sum_{i<N} h_{i-1}(h_{i-2} - h_{i-1} + 1)) &\subseteq O(h \sum_{i<N} (h_{i-2} - h_{i-1} + 1)) \\ &\subseteq O(h(h + N)) \subseteq O(h^2) = O(\log^2 b) \end{aligned}$$

*Refers to "*The Art of Computer Programming - Volume 2: Seminumerical Algorithms (3rd Edition)*" by Donald E. Knuth.

大整数的质因数分解

- 求最大公约数也可以通过分别对两个整数进行质因数分解来求
- **整数的质因数分解** (integer factorization 或 prime factorization)：将合数分解为一系列质数的积
- 当待分解的整数 n 很大时，尚无公开的有效算法
- 目前公认的最好算法是 GNFS (General Number Field Sieve) 算法，但其复杂度仍为指数级：

$$O\left(\exp\left(\left(\tfrac{64}{9}\log n\right)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}\right)\right)$$

即：

$$O\left(\exp\left(\left(\tfrac{64}{9}h\right)^{\frac{1}{3}}(\log h)^{\frac{2}{3}}\right)\right)$$

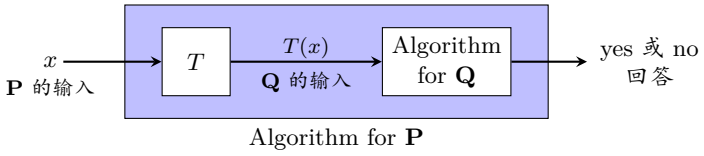
$h=\log n$ 为 n 的位数

- 多数加密协议的安全性均建立在“大整数分解的困难性”基础上

3 $\mathcal{NP}$-完全问题

问题的归约 (Reduction)

- $\mathcal{NP}$-完全问题 ($\mathcal{NP}$-complete problem) 用来描述 $\mathcal{NP}$ 问题中最难的那一类问题，一旦发现一个 $\mathcal{NP}$-完全问题存在以多项式为界的算法，则可以断定每一个 $\mathcal{NP}$ 问题都存在以多项式为界的算法 ($\Rightarrow \mathcal{P}=\mathcal{NP}$)
- 前面所举的问题实例实际上都属于 $\mathcal{NP}$-完全问题
- $\mathcal{NP}$-完全问题正式的定义采用多项式归约的方式给出：已有问题 **Q** 的算法，要解决问题 **P**，若存在某种转换函数 T，能够将问题 **P** 的任一输入 x 转换为问题 **Q** 的输入 $T(x)$，则我们就得到了解决问题 **P** 的算法，对输入 x 的输出为 yes 当且仅当 **Q** 的算法对输入 $T(x)$ 的输出为 yes



多项式归约

Definition 3.1. 设 T 是判定问题 **P** 的输入集合到判定问题 **Q** 的输入集合的函数，如果 T 满足下列条件，则称 T 为**多项式归约** (polynomial reduction)：

1. T 的计算时间以多项式为界；
2. 对每个输入 x，若 x 对 **P** 的输出为 yes，则 $T(x)$ 对 **Q** 的输出也为 yes；
3. 对每个输入 x，若 x 对 **P** 的输出为 no，则 $T(x)$ 对 **Q** 的输出也为 no；或者等价地：对每个输入 x，若 $T(x)$ 对 **Q** 的输出为 yes，则 x 对 **P** 的输出也为 yes；

如果存在这样的 T，则称问题 **P** **可多项式归约到** (polynomially reducible) 到 **Q**，记作：$\mathbf{P}\leq_P\mathbf{Q}$。

Theorem 3.2. 如果 $\mathbf{P}\leq_P\mathbf{Q}$，并且 **Q** 属于 $\mathcal{P}$ 问题，则 **P** 也属于 $\mathcal{P}$ 问题。

$\mathcal{NP}$-完全问题

Definition 3.3 ($\mathcal{NP}$-难问题和 $\mathcal{NP}$-完全问题)**.** 若任一 $\mathcal{NP}$ 问题 $\mathbf{P}$ 可多项式归约到问题 $\mathbf{Q}$，则称问题 $\mathbf{Q}$ 为 $\mathcal{NP}$-**难问题** ($\mathcal{NP}$-hard problem)；若同时问题 $\mathbf{Q}$ 也属于 $\mathcal{NP}$ 问题，则称问题 $\mathbf{Q}$ 为 $\mathcal{NP}$-**完全问题** ($\mathcal{NP}$-complete problem)。

Theorem 3.4. 若存在某个 $\mathcal{NP}$-完全问题属于 $\mathcal{P}$ 问题，则 $\mathcal{P} = \mathcal{NP}$。

- 要证明某一个判定问题 $\mathbf{Q}$ 属于 $\mathcal{NP}$-完全问题，首先要证明它是 $\mathcal{NP}$-难问题，即所有的 $\mathcal{NP}$ 问题 (包括未知的) 都可多项式归约到 $\mathbf{Q}$，*Mission Impossible!*
- 多项式归约具有传递性，这意味着只要有一个问题被证明属于 $\mathcal{NP}$-完全问题，对其他问题的判断就简单了

寻找 $\mathcal{NP}$-完全问题

Theorem 3.5 (Cook 定理)**.** 合取范式可满足性问题是 $\mathcal{NP}$-完全问题。

[1] Stephen A. Cook The Complexity of Theorem-Proving Procedures Proceedings of the third annual ACM symposium on Theory of computing, pages 151–158, 1971

Theorem 3.6. 图着色问题、哈密尔顿环路/路径问题、货郎担问题、子集和问题、背包问题都是 $\mathcal{NP}$-完全问题。

- 证明判定问题 $\mathbf{Q} \in \mathcal{NP}$ 是 $\mathcal{NP}$-完全问题：
  1. 找到一个合适的 $\mathcal{NP}$-完全问题 $\mathbf{P}$，则 $\forall \mathbf{R} \in \mathcal{NP}$，$\mathbf{R} \leq_P \mathbf{P}$
  2. 证明 $\mathbf{P} \leq_P \mathbf{Q}$
  3. 则 $\forall \mathbf{R} \in \mathcal{NP}$，$\mathbf{R} \leq_P \mathbf{Q}$，即 $\mathbf{Q}$ 也是 $\mathcal{NP}$-完全问题

证明实例

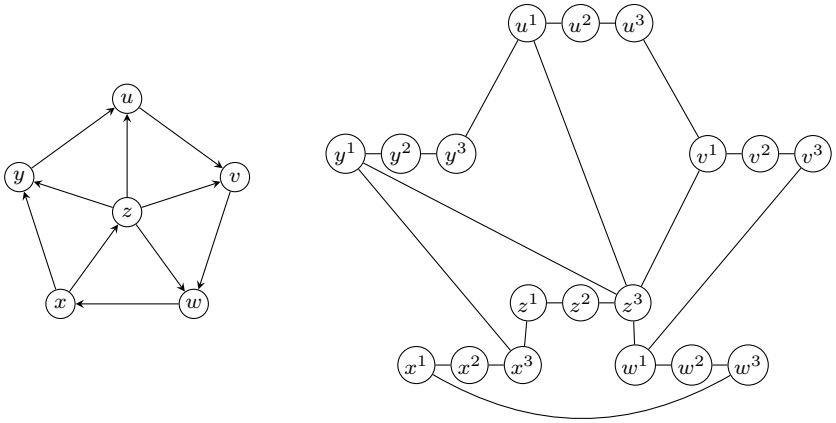
Theorem 3.7. 已知有向图的哈密尔顿环路问题是 $\mathcal{NP}$-完全问题，证明无向图的哈密尔顿环路问题也是 $\mathcal{NP}$-完全问题。

Proof. 显然，无向哈密尔顿环路问题属于 $\mathcal{NP}$ 问题，因此只要证明有向图的哈密尔顿环路问题可多项式规约到无向图的哈密尔顿环路问题 (已知有向图的哈密尔顿问题是 $\mathcal{NP}$-完全问题)。

令 $G = (V, E)$ 是包含 n 个顶点的有向图，现将其转换到无向图 $G' = (V', E')$：对每个 $v \in V$，V' 中包含 3 个顶点 v^1, v^2, v^3 与之对应，E' 中包含两条无向边 v^1v^2, v^2v^3 与之对应，对 E 中的每条边 vw，E' 中包含无向边 v^3w^1 与之对应，显然，该转换所需时间以多项式为界，若 $|V| = n, |E| = m$ 则 $|V'| = 3n, |E'| = 2n + m$。

若 G 中有一条有向哈密尔顿环路 $v_1, \cdots, v_n$，则 $v_1^1, v_1^2, v_1^3, v_2^1, v_2^2, v_2^3, \cdots, v_n^1, v_n^2, v_n^3$ 是 G' 中的一条 (无向) 哈密尔顿环路；另一方面，若 G' 中存在一条哈密尔顿环路，由于对每组顶点 v^1, v^2, v^3，v^2 只与 v^1 和 v^3 相连，因此在环路上必按 v^1, v^2, v^3 或 v^3, v^2, v^1 的顺序访问这三个顶点，而 G' 中的其他边仅连接上标为 1 和 3 的顶点，因此该环路上所有的三顶点组要么都按 $1, 2, 3$ 的顺序排列，要么都按 $3, 2, 1$ 的顺序排列，不妨设该环路为 $v_{i_1}^1, v_{i_1}^2, v_{i_1}^3, \cdots, v_{i_n}^1, v_{i_n}^2, v_{i_n}^3$，则 $v_{i_1}, \cdots, v_{i_n}$ 是 G 中的一条有向哈密尔顿环路。所以 G 包含有向哈密尔顿环路当且仅当 G' 包含无向哈密尔顿环路。 □

转换图示

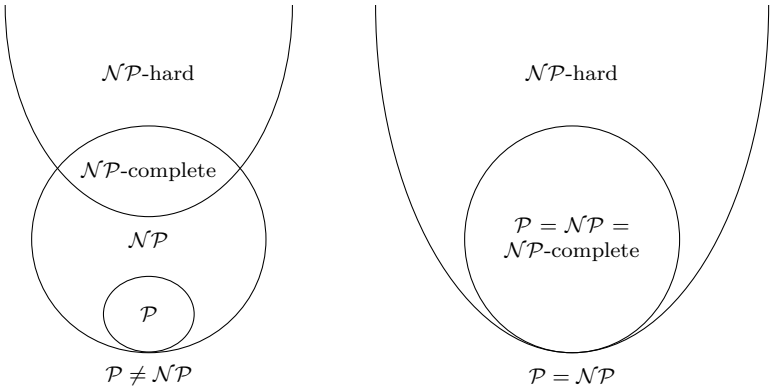


小结

- 在本讲之前，我们所讲的所有算法都以多项式为界，其所解决的问题都属于 $\mathcal{P}$ 问题；
- 本讲所给出的大部分算法问题都属于 $\mathcal{NP}$-完全问题；
- $\mathcal{P}$ 问题和非 $\mathcal{P}$ 问题的界限在于是否存在以多项式为界的 (确定) 算法；
- $\mathcal{NP}$ 问题和非 $\mathcal{NP}$ 问题的界限在于是否存在以多项式为界的非确定算法；
- $\mathcal{P} \subseteq \mathcal{NP}$，但 $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ 仍然是一个开放的问题；
- $\mathcal{NP}$-完全问题是 $\mathcal{NP}$ 问题中最难的一类问题，其难度体现在所有的 $\mathcal{NP}$ 问题都可以 (多项式) 规约到任一个 $\mathcal{NP}$-完全问题，即只要任意一个 $\mathcal{NP}$-完全问题找到了以多项式为界的确定算法，则所有的 $\mathcal{NP}$ 问题都可以找到以多项式为界的确定算法，从而都属于 $\mathcal{P}$ 问题；

小结 **(cont.)**

- 遗憾的是到目前为止，仍然没有发现一个 $\mathcal{NP}$-完全问题存在以多项式为界的确定算法，同时也无法证明这些问题的复杂度下界高于多项式，而 $\mathcal{NP}$-完全问题却在被大量的发现，鉴于 $\mathcal{NP}$-完全问题的解决难度，多数研究者更倾向于认为 $\mathcal{P} \subset \mathcal{NP}$



(参考：http://en.wikipedia.org/wiki/NP-complete)

更深入的学习

- 一本有关 $\mathcal{NP}$-完全性理论的经典书籍：

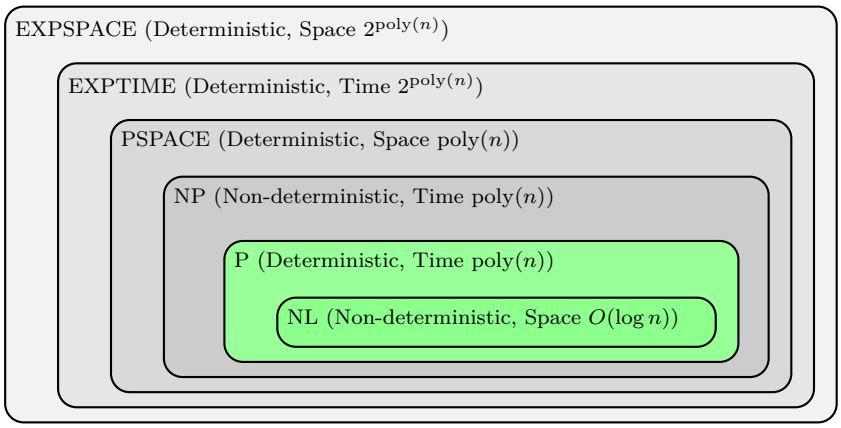
  Michael R. Garey, D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W.H. Freeman, 1979

- 一份已知的 $\mathcal{NP}$-完全问题清单：

  http://en.wikipedia.org/wiki/List_of_NP-complete_problems

- 一个研究领域：计算复杂性理论 (Computational Complexity Theory)

并非结束的结束



(参考：http://en.wikipedia.org/wiki/PSPACE)

- 到现在为止，我们能用计算机来解决的问题还是非常有限的，在算法领域还有非常广阔的未知空间等待我们去研究和探索！