

# 算法基础 笔记

原生生物

\*顾乃杰老师算法基础课堂笔记 [以 \* 开头的行代表动机与注释等, \*\* 开头的行代表补充内容]

## 目录

一 算法概念与数学基础	2
§1.1 算法简介与分析	2
§1.2 设计算法	3
§1.3 渐进记号与递归	5
§1.4 判断方法	7
§1.5 摊还分析	8
二 排序与顺序统计	8
§2.1 简单排序与希尔排序	9
§2.2 堆排序	9
§2.3 快速排序	11
§2.4 线性时间排序	12
§2.5 中位数与顺序统计	13
三 算法设计基本策略	14
§3.1 动态规划	14
§3.2 更多例子	16
§3.3 贪心算法	18
§3.4 分治策略案例	20
§3.5 快速傅里叶变换	21
四 数据结构	22
§4.1 二分检索树	22
§4.2 红黑树	24
§4.3 动态顺序统计	27
§4.4 斐波那契堆	28
§4.5 分离集合	31
五 图论算法与串匹配	32
§5.1 图的表示与遍历	32
§5.2 图论问题	34
§5.3 串匹配算法	37

## 一 算法概念与数学基础

定义：输入-> 算法-> 输出

可计算问题 [computational problem]：对输入输出要求的说明

问题的实例 [instance]：某个求解需要的具体输入

(排序算法中，输入数组、输出排好的数组是问题，而一个具体的乱序数组则是实例)

正确性：要求对任何输入能正确给出输出 (随机算法可能一定程度违反)

学习算法的意义：确定解决方式的正确性、选择容易实现的算法，比较时空复杂度

\*算法效率比电脑速度更重要

### §1.1 算法简介与分析

伪代码书写：注重表达清楚，不关注语言细节

约定：缩进表示块结构、循环结构与 C 类似解释、//代表注释、变量如无特殊说明表示局部变量

举例：插入排序 (就地 [in-place] 排序，需要的额外空间为  $O(1)$ ，即常量)

---

```
def INTERSECTION-SORT(A):
  for (j <- 2) to length(A)
    key <- A[j]
    i <- j-1
    while (i > 0 and A[i] > key)
      A[i+1] <- A[i]
      i <- i - 1
    A[i+1] <- key
```

---

\*如何证明正确性?

可选方法：循环不变式 [loop invariant]，保证初始化、迭代时均为真 (类似归纳)，循环终止时能提供有用性质 (局限性：只能用于判断循环，无法判断分支等)

插入排序中不变式：A[1..j-1] 在循环后有序

\*现实：理论证明几乎无法做到，通过大量测试假定正确

\*\*关于正确性 [correctness]：正确性指算法在程序规范下被认定为正确的判定，其中功能 [functional] 正确针对输入输出的行为，一般分为部分 [partial] 正确与完全 [total] 正确，前者指输出结果时结果正确，后者还额外要求必须能输出结果。

### 算法分析

含义：预测算法需要的资源，通常关心时间 [computational time] 与内存空间 [memory]，偶尔会涉及通信带宽、硬件使用等

为了统一评价，需要使用统一、简洁的计算模型

\*对串行算法一般使用 RAM 模型 [Random-access machine]，并行则为 PRAM 模型 [Parallel RAM]

RAM 模型特点：

1. 指令一条条执行 (不存在并发)

2. 包含常见算数指令、数据移动指令、控制指令，且指令所需时间为常量

\*按照真实计算机设定，不应滥用

\*特殊情况：如一般指数运算不是常量时间，但  $2^k$  通过左移可常量时间

3. 数据类型有整数与浮点

\*不关心精度

4. 假定数据字的规模存在范围，字长不能任意长

\*不关心内存层次（高速缓存、虚拟内存）

影响运行时间的主要因素：

1. 输入规模

2. 输入数据的分布

\*将算法运行时间描述成输入规模的函数

3. 算法实现所选用的底层数据结构

\*思考：RAM 模型中其他影响因素？

\*输入规模与运行时间如何严谨定义？

输入规模 [input size]：对许多问题为输入项的个数（如排序），但有时（如整数相乘）关心的是二进制表示的总位数，有时则用两个数表示更合适（图的顶点数与边数）

\*必须描述清楚输入规模的量度

运行时间 [running time]：执行的基本操作数或步数，与机器无关，一般假设每行伪代码恒定时间

\*实际计算时，由于循环嵌套所需的步数不同，很可能较为复杂，于是需要二次抽象 [最终将系数抽象为独立于数据规模的常数，只关心量级]

最好/最坏运行时间：最快/最慢情况的运行时间（插入排序的例子中，最好为  $O(n)$ ，最坏为  $O(n^2)$ ）

平均运行时间：运行时间在所有输入下的期望值（与数据的概率分布有关，一般默认均匀一致分布，插入排序的例子中为  $O(n^2)$ ）

\*平均 vs 最坏：最好运行时间的参考意义不大，而平均运行时间往往非常难以计算，因此一般采取最坏运行时间 [事实上平均运行时间往往和最坏运行时间量级相同]。最坏运行时间给出了运行时间的上界，课程中主要讨论最坏，偶尔讨论平均。

## §1.2 设计算法

分治、贪婪、动态规划、线性规划、回溯、分支定界……

分治 [divide-and-conquer] 法：分解 [Divide]、解决 [Conquer]、合并 [Combine]

举例：归并排序（分解成子序列，对子序列排序后合成）

---

```
def MergeSort(A,p,r):
    if(p < r)
        q <- (p + r) / 2
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
```

```

Merge(A,p,q,r)
def Merge(A,p,q,r):
    n1 = q - p + 1;
    n2 = r - q
    Let L[1..n1+1],R[1..n2+1] be new arrays
    for i <- 1 to n1
        L[i] <- A[p+i-1]
    for j <- 1 to n2
        R[j] <- A[q+j]
    L[n1+1] <- R[n2+1] <- INFTY [监视哨]
    i <- j <- 1
    for k <- p to r
        if L[i] <= R[j]
            A[k] <- L[i]
            i++
        else
            A[k] <- R[j]
            j++

```

---

\*采用无穷大作监视哨避免过多判断（注意：一定要保证充分大）

\*Merge 算法正确性：迭代时子数组  $A[p..k-1]$  按从小到大的顺序包含  $B[1..n1+1]$  与  $C[1..n2+1]$  中的  $k-p$  个最小元素

分析基于分治法的算法：递归式、递归方程

设  $T(n)$  是规模为  $n$  的运行时间（当  $n$  在某个常数之下时可当作常数）

假设分解为  $a$  个子问题，每个的规模是原本  $1/b$ ，分解所需时间为  $D(n)$ ，合并所需时间为  $C(n)$ ，则总时间为：

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

对归并排序： $T(n) = \begin{cases} \Theta(1) & n < c \\ 2T(n/2) + \Theta(n) & \text{otherwise.} \end{cases}$ ，事实上复杂度为  $\Theta(n \log n)$

思考：

(1) 自底向上 [bottom-up] 通过两两归并也可实现归并排序

(2) 如何使数组  $A[0..n-1]$  循环左移  $k$  位？

法一：颠倒  $0$  到  $k-1$ 、颠倒  $k$  到  $n-1$ 、颠倒  $0$  到  $n-1$ （约  $3n$  次内容读写）

法二：思路：把置换拆分成轮换进行（约  $n$  次内容读写）

---

```

d <- gcd(n,k)
for i <- 0 to d-1
    x <- A[i]
    t <- i
    for j <- 1 to n/d-1

```

```

A[t] <- A[(t+k)]
t <- (t+k) mod n
A[t] <- x

```

---

### §1.3 渐进记号与递归

$f(n) = \Theta(g(n))$  代表存在正常数  $C_1, C_2, n_0$  使得  $n \geq n_0$  时  $0 \leq C_1g(n) \leq f(n) \leq C_2g(n)$ 。

\*即趋于无穷时阶相同,  $g(n) = \Theta(f(n))$  时亦有  $f(n) = \Theta(g(n))$

\*一般证明中可以取较粗糙的  $C_1, C_2, n_0$ , 不用解出精确的点

$f(n) = O(g(n))$  代表存在正常数  $C, n_0$  使得  $n \geq n_0$  时  $0 \leq f(n) \leq Cg(n)$ 。

\*即趋于无穷时  $f$  的阶不超过  $g$ ,  $g(n) = \Theta(f(n))$  时必有  $g(n) = O(f(n))$

$f(n) = \Omega(g(n))$  代表存在正常数  $C, n_0$  使得  $n \geq n_0$  时  $0 \leq Cg(n) \leq f(n)$ 。

\*即趋于无穷时  $g$  的阶不超过  $f$ ,  $g(n) = \Theta(f(n))$  时必有  $g(n) = \Omega(f(n))$

$f(n) = o(g(n))$  代表任意正常数  $c$  存在正常数  $n_0$  使得  $n \geq n_0$  时  $0 \leq f(n) < cg(n)$ 。

$f(n) = \omega(g(n))$  代表任意正常数  $c$  存在正常数  $n_0$  使得  $n \geq n_0$  时  $0 \leq cg(n) < f(n)$ 。

\*大小写区别在存在与任意, 也可以理解为大写剔除  $\Theta$

\*实际使用时,  $O(f(n))$  可以代表某个满足  $g(n) = O(f(n))$  的  $g(n)$ , 这样的匿名 [anonymous] 函数可以正常参与运算, 但不应出现存在歧义的情况

渐进关系的性质:

\*五种关系都具有传递性

\* $\Theta, O, \Omega$  具有自反性

\* $\Theta$  与  $\Theta$ 、 $O$  与  $\Omega$ 、 $o$  与  $\omega$  互相置换对称

\*\*不是任何两个函数都渐进可比, 例如  $n$  与  $n^{1+\sin n}$ , 不存在  $O$  或  $\Omega$  关系

#### 上/下取整

取整性质: 对正整数  $a, b, n$ ,  $\lceil \frac{\lceil n/a \rceil}{b} \rceil = \lceil \frac{n}{ab} \rceil, \lfloor \frac{\lfloor n/a \rfloor}{b} \rfloor = \lfloor \frac{n}{ab} \rfloor$

引理:  $f(x)$  是连续单调上升函数, 且整点处才取整值, 则  $\lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil, \lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$

引理证明: 对第一个式子, 若  $\lceil f(x) \rceil \neq \lceil f(\lceil x \rceil) \rceil$ , 由单调增可知  $f(x) \leq f(\lceil x \rceil)$ , 从而  $\lceil f(x) \rceil < \lceil f(\lceil x \rceil) \rceil$ 。由向上取整定义可知  $\lceil f(x) \rceil < f(\lceil x \rceil)$ , 而  $f(x) \leq \lceil f(x) \rceil$ , 从而  $\lceil f(x) \rceil$  在  $f(x)$  与  $f(\lceil x \rceil)$  之间。由连续定义知存在  $x_0$  使得  $f(x_0) = \lceil f(x) \rceil$ , 由条件知  $x_0$  必然为整数, 但  $x \leq x_0 \leq \lceil x \rceil$ , 其由向上取整定义只能为  $\lceil x \rceil$ , 与  $\lceil f(x) \rceil < \lceil f(\lceil x \rceil) \rceil$  矛盾。另一个式子同理。

定理证明: 取  $f(x) = \frac{x}{b}$ ,  $x = \frac{n}{a}$  即可。

其他数学: 模、指数、对数、阶乘 (Stirling 公式)、函数迭代 ( $f^{(n)}(x)$ )

\*斐波那契 [Fibonacci] 数  $a_1 = a_2 = 1, a_n = a_{n-1} + a_{n-2}$ , 通项为  $\frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$

\*思考: 服务器每隔  $g$  秒送包, 拥有  $k$  个端口 (同时发  $k$  个), 收到的服务器需要花  $L$  秒时间解压, 自此每隔  $g$  秒给新服务器发送信息, 时间与接受的服务器总数关系? (起初 1 对大兔子, 大兔子每个隔  $g$  个月可生  $k$  对小兔子, 小兔子需要  $L$  个月成熟, 求第  $n$  个月的兔子对数?) [广义斐波那契数]

#### 递归介绍

\*递归可能分解为规模不等的子问题

求解递归方程的方法: 替代法 (猜测上界后证明)、递归树法 (转化成树, 结点表示不同层次产生的代价, 再采用边界求和)、主方法 (求解  $T(n) = aT(n/b) + f(n)$ ,  $a \geq 1, b > 1$ ,  $f(n)$  是某给定函数 (并非对任何都可解))

\*\*技术细节:

1. 假定自变量整数，忽略上下取整。
2. 对足够小的  $n$  假设  $T(n)$  为常数，忽略边界。  
(一些特殊情况可能导致技术细节非常重要，上面两种条件仍然值得重视)
3. 有时可能存在不等式情况，如  $T(n) \leq 2T(n/2) + O(n)$ ，此时一般用  $O$  描述上界，反之对大于等于可用  $\Omega$  描述下界。

例：最大子数组问题（给定数组，求和最大的连续子数组）

分治策略：找到数组中央位置，任何连续子数组必然在其左侧、右侧，或包含它。左侧与右侧可直接通过递归，由此只需要找到包含中间位置的最大子数组后取最大值即可。

---

```
def find_max_crossing_subarray(A, low, mid, high):
    left_sum = -INFTY
    sum = 0
    for i = mid downto low
        sum += A[i]
        if (sum > left_sum)
            left_sum = sum
            max_left = i
    right_sum = -INFTY
    sum = 0
    for i = mid -> low
        sum += A[i]
        if (sum > right_sum)
            right_sum = sum
            max_right = i
    return (max_left, max_right, left_sum+right_sum)
```

---

整体算法：利用上方的算法进行递归， $low=high$  即为终止条件。

复杂度分析：包含中间位置的部分的复杂度为  $\Theta(n)$ ，递归方程为  $T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & otherwise. \end{cases}$ ,

可发现复杂度与归并排序相同，为  $\Theta(n \log n)$ 。

\*算法改进 ( $\Theta(n)$  算法)：从左侧开始，找到第一个大于  $\theta$  的位置  $i_1$  开始，依次求和 ( $sum+=A[j]$ )， $max_1$  记录目前的最大值，并记录当前的  $j_1$ 。直到  $sum < \theta$  时中止，然后继续向右找到下一个大于  $\theta$  的位置  $i_2$ ，清空  $sum$ ，重复此过程，在比较中得到  $max_k$  中的最大值即可（证明思路：反证，若否则可以拼接为更大）。

---

```
def find_max_subarray(A, low, high):
    now <- low
    sum <- 0
    max <- -INFTY
```

```

for now <- low to high
  if (sum > 0)
    sum += A[now]
    if (sum > max_now)
      j_now <- now
      max_now <- sum
    if (sum <= 0 or now == high)
      if (max_now > max)
        i_max <- i_now
        j_max <- j_now
        max <- max_now
      sum <- 0
    else if (A[now] > 0)
      max_now <- sum <- A[now]
      i_now <- j_now <- i
return (max,i_max,j_max)

```

---

## §1.4 判断方法

### 替代法

包含两个步骤：猜测解的形式、归纳常数（直接替代）并证明解正确（要求易于猜得）

例：针对  $T(n) = 2T(\lfloor n/2 \rfloor) + n$ ，先猜测解为  $T(n) = O(n \log n)$ ，选取常数  $C > T(2) + 1$  即可。

更复杂的例子：求  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$  的解的上界。

解法（平移的思路）：令  $n = m + 34$ ，可发现  $T(m + 34) = 2T(\lfloor m/2 \rfloor + 34) + m + 34$ ，由此  $T(m + 34) + 34 = 2(T(\lfloor m/2 \rfloor + 34) + 34) + m$  类似上一种情况可直接估算出上界。

\*猜测出渐近界未必能归纳成功，有时是因为归纳假设偏弱，可以尝试加强假设、调整低阶项、初等变换等，如  $T(n) = 2T(\lfloor n/2 \rfloor) + 1$ ，归纳假设  $cn$  无法继续，假设为  $cn - 2$  即可。

\*不应将猜测无理由放大

\*变量代换：对  $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$ ，取  $m = \log n$  可发现最终结果为  $O(\log n \log \log n)$ 。

\*\*弊端：猜测、证明都可能困难

### 递归树

每个结点代表相应子问题代价，行和代表某层代价，总和得总代价

\*一般用来获得猜测解再替代，省略大部分细节。也可细化直接解出结果。

例： $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \implies T(n) = 3T(n/4) + cn^2$  简化，接着画出递归树求得复杂度大约为

$$\sum_{i=0}^{\infty} \frac{3^i}{16^i} cn^2 = c'n^2, \text{ 因此为 } \Theta(n^2) \text{ 量级。}$$

代价不相同： $T(n) = T(n/3) + T(2n/3) + O(n)$ ，作出递归树可发现每层的和为  $cn$ ，再由行数（通过解方程可计算出层数具体值，不过估算中没有精确必要）为  $O(\log n)$  可知复杂度  $O(n \log n)$ 。同理，对于

$T(n) = T(n/4) + T(n/2) + n^2$ ，可类似算得结果为等比级数的  $n^2$  倍，仍为  $n^2$  量级。

\*关键为求行和与总代价，需要上下行和的关联

### 主方法

主要适用范围:  $T(n) = aT(n/b) + f(n)$ , 其中  $a \geq 1, b > 1, f$  渐近非负。

主定理 [The master theorem]: 若  $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0$ , 则  $T(n) = \Theta(n^{\log_b a})$ ; 若  $f(n) = O(n^{\log_b a} \log^k n)$ , 则  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ ; 若  $f(n) = \Omega(n^{\log_b a - \epsilon}), \epsilon > 0$ , 且  $af(n/b) \leq cf(n)$ , 则  $T(n) = \Theta(f(n))$ 。

\*三种情况存在间隙, 可能无法判断 (如  $T(n) = 3T(n/3) + \log \log^2 n$ ), 并不代表递归方程无解

### §1.5 摊还分析

思路: 考虑一系列操作的可能的最坏情况的平均复杂度为

\*不涉及概率问题, 与平均复杂度分析不同

例: 栈操作

只有 PUSH 和 POP 时, 视二者代价为 1, 则操作的平均代价必然为 1。

在 PUSH 和 POP 上增添一个 MULTIPOP(k), 一次允许弹出多个元素 (最多到栈底), 其代价实际上为  $\min(s, k)$ , 其中  $s$  代表栈中元素。

\*加入此操作后,  $n$  次操作的平均代价?

虽然 MULTIPOP 的上限代价为  $O(n)$ , 但由于最多入栈  $n$  次, 出栈也最多  $n$  次, 实际上平均代价仍为  $O(1)$ 。

例: 二进制计数器

$k$  位二进制计数器, 将每位的修改作为代价 1。在允许加法时, 一次加法最高需要修改  $k$  位, 但是  $n$  次的总代价可以估算为  $\frac{n}{2} + 2\frac{n}{4} + 3\frac{n}{8} + \dots = O(n)$ , 于是平均代价仍为  $O(1)$ 。

#### 核算法 [accounting method]

给每个操作赋予摊还代价, 保证每次操作摊还代价的总和不小于实际代价 (而当某步摊还代价比实际代价大时, 将其称为信用, 用来支付之后的差额)。

\*栈操作的例子中, PUSH 的摊还代价为 2, 其余全为 0; 对二进制计数器, 假设 0 到 1 操作 [置位] 的摊还代价为 2, 1 到 0 操作 [复位] 的摊还代价为 0, 由于每次增加恰好产生一次置位, 而复位时每个为 1 的位都保留信用支付, 因此可得到结论。

#### 势能法 [potential method]

对每个状态定义势函数 [映射到实数], 一般初始状态为 0。将摊还代价定义为代价 + 操作后的势 - 操作前的势。

\*栈操作的例子中, 势为栈中元素个数; 对二进制计数器, 势为数中 1 的个数。可发现得到的摊还代价与核算法中一致。

\*通过势能法分析, 当二进制计数器计数器不是从零开始时, 这样的定义仍然合理, 于是  $n$  个操作的实际代价为  $2n + b_n - b_0$ ,  $b$  为势函数。于是, 充分大时平均复杂度仍然  $O(1)$ 。

## 二 排序与顺序统计

### 一些概念

稳定性: 不论输入数据如何分布, 关键字相同的数据对象 (如两个 3) 在整个过程中保持相对位置不变  
内排序/外排序: 是/否在内存中进行

时间开销: 通过比较次数与移动次数衡量

评价标准: 所需时间 [主要因素]、附加空间 [一般都不大, 矛盾不突出]、实现复杂程度

## §2.1 简单排序与希尔排序

直接插入、简单选择、冒泡

希尔排序：调用直接插入，不稳定，复杂度更低。

\*直接插入见第一章

简单选择： $n-1$  遍处理，第  $i$  遍将  $a[i..n]$  中的最小元与  $a[i]$  交换。比较次数  $O(n^2)$ ，移动次数最多  $3(n-1)$ ，平均时间复杂度  $O(n^2)$ 。需要常数额外空间，就地排序。存在跨格跳跃，分析可发现不稳定。

冒泡：至多  $n-1$  遍处理，第  $i$  遍在  $a[i..n]$  中依次比较，相邻交换，某次发现没有需要交换时结束。比较次数  $O(n^2)$ ，移动次数亦为最多  $O(n^2)$ ，平均时间复杂度  $O(n^2)$ 。稳定就地排序。

希尔 [shell] 排序：又称缩小增量排序，看作有增量的子列进行插入排序，对位置间隔较大的结点进行比较以跨过较大距离。性能与增量序列有关，尽量避免序列中的值互为倍数，优于直接插入。

\*最后一趟必须以 1 作为增量以保证正确

\*由于几趟后接近分块有序，希尔排序的实际效率大大优于直接插入排序。复杂度分析非常困难，统计结论一般时间复杂度在  $O(n^{1.25})$  左右，而空间效率也很高。虽然如此，其理论最有复杂度亦无法达到  $O(n \log n)$ 。

---

```
def ShellPass(A,d):
    for i <- d+1 to n
        if (A[i]<A[i-d])
            A[0] <- A[i]; j <- i-d;
            while (j>0 and A[0] < A[j])
                A[j+d] <- A[j]
                j <- j-d
            A[j+d] <- A[0]
def ShellSort(A,D):
    for i <- 1 to length(D)
        ShellPass(A,D[i])
```

---

## §2.2 堆排序

堆 [heap] 排序：集中了归并排序与插入排序的优点，时间复杂度  $O(n \log n)$ ，空间复杂度  $\Theta(1)$ 。使用堆 [heap] 数据结构，不止可以用在堆排序中，还可以构造另一种有效的数据结构：优先队列。

\*\*和内存的“堆”并不是同一个东西

(二叉)堆定义：树上每个结点对应数组中一个元素的完全二叉树，分为大根堆 [父结点大于等于其任何子结点] 与小根堆 [父结点小于等于其任何子结点]，排序算法中使用大根堆。

\*表示堆的数组  $A$  有两个属性：数组元素个数 [length] 与数组中属于堆的元素的个数 [heapsize]， $heapsize \leq length$ ，数组的第一个元素代表根结点。

下标 1 开始的数组中，由于其为完全二叉树，可各层顺序排列， $A[i]$  的父结点是  $[i/2]$ ，左孩子  $2i$ ，右孩子  $2i+1$ 。

\*[标准定义] 满 [full] 二叉树：每层均为满；完全 [complete] 二叉树：最后一层的结点都在左侧，未必满；严格 [strict] 二叉树：每个结点的孩子都为零个或两个

\*0 开始： $A[i]$  的父结点是  $[(i-1)/2]$ ，左孩子  $2i+1$ ，右孩子  $2i+2$ 。

结点高度：该结点到叶结点最长简单路径上边的数目（堆的高度：根结点的高度，为  $\lceil \log_2(n+1) \rceil$ ）。

**基本操作**

维护堆：假定  $A[i]$  的左右子树都为大根堆，但  $A[i]$  有可能小于其孩子，则通过逐级下降使得下标为  $i$  的根结点子树重新遵循大根堆。

---

```
def MAX_HEAPIFY(A, i):
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heap_size and A[l] > A[i]
        Largest = l
    else
        Largest = i
    if r <= A.heap_size and A[r] > A[Largest]
        Largest = r
    if (Largest != i)
        swap A[i], A[Largest]
        MAX_HEAPIFY(A, Largest)
```

---

\*每次至少下移一层，时间复杂度  $O(\log n)$ 。

建堆：从后往前进行转化，一半处开始即可。

---

```
def BUILD_MAX_HEAP(A):
    A.Heap_size <- A.length
    for i <- [A.length/2] downto 1
        MAX_HEAPIFY(A, i)
```

---

\*求和估算可知时间复杂度  $O(n)$ 。

堆排序：建好堆后，每次将根结点 [当前最大] 与最后一个元素互换，堆长度减小 1，然后调整堆。

---

```
def HEAPSORT(A):
    BUILD_MAX_HEAP(A);
    for i <- length(A) downto 2
        swap A[1], A[i]
        A.Heap_size -= 1
        MAX_HEAPIFY(A, 1)
```

---

二叉堆的扩展：优先队列 [并不是堆的基本操作! ]

\*用来维护一组元素构成的集合  $S$  的数据结构，每个元素都有一个相关值，称关键字 [key]

最大优先队列 [可用大根堆构造] 基本操作：

1. `insert(S,x)` 插入元素进  $S$

2. `maximum(S)` 返回  $S$  中最大关键字元素
3. `extract_max(S)` 去掉并返回  $S$  中最大关键字元素
4. `increase_key(S, x, k)` 将  $x$  的关键字增加到  $k$  [只允许增加]

\*最小优先队列类似相反

\*大根堆实现时, 最大关键字元素即为第一个元素 [复杂度  $O(1)$ ]; 删除最大值直接将最大值与最后一个元素交换, 堆长度减小 1 并调整即可 [复杂度  $O(\log n)$ ]; 增大某元素值通过不断和父结点比较交换实现 [复杂度  $O(\log n)$ ]; 加入结点堆长度增加 1, 先将末尾元素赋值为  $-\infty$  再增大值到目标即可 [复杂度  $O(\log n)$ ].

### §2.3 快速排序

对于包含  $n$  的数的输入数组, 时间复杂度最坏情况  $O(n^2)$ , 不稳定、就地, 期望时间  $\Theta(n \log n)$  且常数因子很小。

\*分治思想

划分: 划分为左右两个子数组与中间, 使得左侧均小于等于中间, 右侧均大于等于中间, 中间下标  $q$  在划分中确定。

解决: 递归调用, 两边分别排序。

合并: 由于子数组有序, 合并直接已经排好。

```
def QUICKSORT(A, p, r):
    if (p < r)
        q = Partition(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)
def PARTITION(A, p, r):
    x <- A[r]
    i <- p - 1
    for j <- p to r-1
        if A[j] <= x
            i <- i + 1
            swap A[i], A[j]
    swap A[i+1], A[r]
    return i + 1
```

另一种 PARTITION:

```
def PARTITION(A, p, r):
    i <- p; j <- r; temp <- A[i];
    while (i != j)
        while (A[j] >= temp and i < j)
            j <- j - 1
        if (i < j)
            A[i] <- A[j]
```

```

    i <- i + 1
  while (A[i] <= temp and i < j)
    i <- i + 1
  if (i < j)
    A[j] <- A[i]
    j <- j - 1
  A[i] <- temp
  return i

```

---

\*看似双方向进行，但由于**必须检查越界**，需要额外比较

复杂度：分解均匀时接近归并，不均匀时最坏情况，为  $O(n^2)$

\*解递推式  $T(n) = \max_q(T(q) + T(n-1-q)) + Cn$  可知最坏情况上界，而每次最不均匀划分能取到  $cn^2$  时间，从而可知最坏情况复杂度

平均情况：可以发现，对任何不超过固定比例（如每次少比多不超过 **1:9**）的划分，复杂度都是  $O(n \log n)$ ，利用此估算，设平均复杂度  $T(n)$ ，有  $T(n) = \frac{1}{n} \sum_{k=0}^{n-1} (T(k) + T(n-1-k) + cn) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + cn$ 。考虑  $nT(n) - (n-1)T(n-1)$  可估算出  $\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n}$ ，利用  $1 + \dots + \frac{1}{n} \sim \log n$  可知平均复杂度。

随机快速排序：每次 PARTITION 并非将最右元素作为分割，而是**选取随机元素**分割以优化性能。

\*\*堆排序与快速排序比较：按照通常 RAM 模型会发现堆排序的复杂度更低，而引进一级缓存（缓存的量级在  $n^{1/3}$  左右）时即会有快速排序的复杂度更低。

## §2.4 线性时间排序

### 比较排序的时间下界

比较排序的算法可以用**决策树**的方式表示，边代表判定过程，而结点代表已经确定顺序的部分。

\*决策树中至少需要  $n!$  个叶结点表示  $n!$  种判定结果

定理：比较排序的最坏情况时间  $\Omega(n \log n)$ 。

证明：由于一次比较产生一层，比较  $h$  次的高度为  $h$ ，而此时最多容纳  $2^h$  个叶结点，因此  $2^h \geq n!$ ，从而  $h \geq \log(n!) = \Omega(n \log n)$ 。

\*线性时间排序一定不能直接基于比较

### 计数 [counting] 排序

思路：对  $\theta$  到  $k$  中的整数组成的数列，只要知道每个整数出现了几次就会知道结果所在的位置。

---

```

def COUNTING_SORT(A, B, k):
  for i <- 0 to k
    C[i] <- 0
  for i <- 1 to length(A)
    C[A[j]] <- C[A[j]] + 1 //C[t] = sum(A==t)
  for i <- 1 to k
    C[i] <- C[i] + C[i-1] //C[t] = sum(A<=t)
  for i <- length(A) downto 1
    B[C[A[j]]] <- A[j]

```

---

```
C[A[j]] <- C[A[j]] - 1
```

---

\*考虑四个循环可发现其复杂度为  $\Theta(n+k)$ ，且为稳定排序。

### 基数 [radix] 排序

思路：对整数，按位数从末位向前排序（对每位采用稳定排序算法，如计数排序）。

时间复杂度： $n$  个  $d$  位数，每位  $k$  种取值，计数排序时耗时  $O(d(n+k))$

\*由上方可知，给定  $n$  个  $b$  位二进制数与正整数  $r \leq b$ ，时间复杂度可以控制在  $\Theta(b(n+2^r)/r)$  内。

### 桶 [bucket] 排序

思路：对  $\theta$  到  $1$  之间，先划分所有数据到  $n$  个不同的“桶”内再进行排序。

---

```
def BUCKET_SORT(A):
    n <- length(A)
    for i <- 1 to n
        insert A[i] into list B[floor(nA[i])]
    for i <- 0 to n-1
        sortlist B[i] with insertion sort
    Print B[i] in order
```

---

时间复杂度：针对均匀一致分布才能达到较好效果，最坏情况  $O(n^2)$ 。平均性能较好，为  $\Theta(n)$ 。

## §2.5 中位数与顺序统计

定义：第  $i$  小的元素称为第  $i$  个顺序统计量， $n$  个元素的低中位数为第  $\lfloor \frac{n+1}{2} \rfloor$  个顺序统计量，高中位数为第  $\lceil \frac{n+1}{2} \rceil$  个，一般默认为低中位数。

\*寻找最大或最小值时间复杂度必然为  $\Theta(n)$ ，但同时找最大最小值问题至少可通过  $\lceil \frac{3n}{2} \rceil - 2$  次比较完成。

证明：记  $N$  状态为从未参与过比较， $L$  状态为参与过但只大不小， $S$  为参与过但只小不大， $M$  为参与过且大小都成为过，则状态间的转换关系为  $N \rightarrow L/S \rightarrow M$ 。所有元素只有一个能保证  $L/S$  不变，即为最大/最小，剩下的都会成为  $M$ ，总状态转换数为  $2(n-2) + 1 + 1 = 2n - 2$ 。

下面考虑一次比较能造成的状态转换：只有当  $N$  与  $N$  比较时一定会有两次状态转换，其余在最坏情况下至多一次状态转换 [严谨性问题：整体最坏未必每次都能遇到最坏]。由此最坏情况至少  $\frac{n}{2} + (2n-2 - (2\frac{n}{2})) = \frac{3n}{2} - 2$ ，而由于比较次数一定为整数，需要取上整，即为结果。

算法：先每两位进行比较，将小的放在奇数位置。在偶数位中找到最大值，奇数位中找到最小值即可。

思考：找第二大元素需要的最少比较次数？

寻找任意第  $i$  小元素：类似快速排序，通过  $PARTITION$  后左右元素数量确定应在哪一侧找。

---

```
def RANDOMIZED_SELECT(A, p, r, i):
    if p == r:
        return A[p]
    q = RANDOMIZED_PARTITION(A, p, r)
    k = q - p + 1
    if i == k:
        return A[q]
```

```

if i < k:
    return RANDOMIZED_SELECT(A, p, q-1, i)
else:
    return RANDOMIZED_SELECT(A, q+1, r, i-k)

```

\*采用随机 PARTITION 增加效率

复杂度：最坏情况是  $\Theta(n^2)$ ，平均性能为  $\Theta(n)$ （假设均匀一致分布，可通过随机变量计算得结果）。

\*最坏情况  $O(n)$  的算法：针对划分不均匀情况改进，使划分均匀。

想法：将数组每五个分组，插入排序找到中值，再从排好序的组列表中提出中值。递归调用，利用中值的中值作 PARTITION。

证明：3.3 节提到，PARTITION 对任何不超过固定比例的划分都是较好的，而估算可知这样的取中值方式可以保证两侧的比例不超过 3:7。计算时间复杂度：

$$T(n) \leq T(n/5) + an + \max\{T(left), T(right)\} \leq an + T(n/5) + T(7n/10)$$

猜测可解出  $T(n) \leq 10an$ ，从而为  $O(n)$ 。

## 三 算法设计基本策略

### §3.1 动态规划

Richard Bellman, 1950s: 最优性原理、动态规划 [dynamic programming]

与分治法相似，但保存已求解的子问题，不需要重复求解。

最优化问题：寻找符合约束条件时优化函数的最值

\*可行解（满足约束条件的解）、最优解（获得最佳值的可行解）

例子 (Thirsty baby): 有  $n$  种不同饮料，每种最多有  $a_i$ ，满意度为  $x_i$ ，总共需要  $t$ ，求满意度最高的饮用方法。

\*评价函数为  $\sum_{i=1}^n x_i s_i$ ， $x_i$  总和为  $t$  且范围为  $[0, a_i]$ 。

**最优性原理** [过程的最优决策序列的性质]：无论过程的初始状态和初始决策是什么，其余的决策必须相对初始决策产生的状态构成最优决策序列（也即从任何一步开始看都是最优的）。

\*全局最优具有一定的局部最优性

刻画最优解的结构特征-> 递归定义最优解值-> 自底向上计算最优解值-> 通过计算信息构造最优解

能用动态规划求解的条件：**最优子结构** [optimal substructure]、子问题覆盖

\*这要求了全局最优存在某种局部最优，且局部最优可以导出全局最优

运行时间估计：子问题个数乘子问题最多需要考察的选择数得到上界

常用方法：自底向上，先计算子问题再计算原问题 [或带记忆的递归方法]

\*\*不满足最优性原理的例子：对有向图，寻找两点间的最短路径满足最优性原理，但最长简单 [无环] 路径不满足最优性原理。

思考：如何求解最长简单路径问题？

子问题覆盖：会涉及重复求解子问题，于是可以存储、利用

\*利用表格存储辅助信息，从而递归构造最优解

例：钢条切割

已知长度为  $i$  的钢条价格  $p_i$ ，长度均为整数，总长固定，求最大收益的分割方案。

思路：每次只需要考虑第一次切割（或不切割），剩下的部分可以由更低长度时的最优值 [最优子结构] 确定

定义  $r_n$  为长为  $n$  的钢条的最优切割方案，递推为  $r_n = \max_{0 \leq i \leq n/2} (r_i + r_{n-i})$ 。

\*事实上可以将  $i$  的范围提到  $n$ ,  $r_i$  写为  $p_i$ , 因为长度已经确定

\*自顶向下的递归写法会导致大量重复计算，因此需要**自底向上**

[递归若检测是否已经计算过子问题，仍可做到多项式复杂度]

```
def BOTTOM_UP_CUT_ROD(p, n):
    let r[1..n] be array
    r[0] = 0
    for j = 1 to n
        q = - INFTY
        for i = 1 to j
            q = max(q, p[i] + r[j-i])
        r[j] = q
    return r[n]
```

获得最优解的方法：将每个  $r[i]$  第一次切割的位置保存为  $c[i]$ ，反复回看以确定最优解  
打印方式：

```
while n > 0:
    print(c[n])
    n -= c[n]
```

\*时间复杂度： $\Theta(n^2)$

### 矩阵链乘

\*直接计算的方法： $A(1:p,1:q)$  与  $B(1:q,1:r)$  相乘的复杂度是  $pqr$

给定  $n$  个矩阵组成的序列  $A_1, \dots, A_n$ ，且前一个的行数等于后一个的列数，要计算它们的乘积，求计算速度最快的运算顺序（由结合律可任意加括号）。

\*总运算顺序可能：通过递推可知为**卡特兰数 [Catalan Number]**

自顶向下分析：在最优顺序中，考虑最后一次运算，划分出的左右两块应各自为最优运算顺序。

于是，记  $m_{ij}$  为  $A_i \dots A_j$  所需的最小乘法次数， $p_i$  为  $A_i$  的列数， $p_0$  为  $A_1$  的行数，则有  $m_{ij} =$

$$\begin{cases} 0 & i = j \\ \min_k \{m_{ik} + m_{k+1,j} + p_{i-1}p_kp_j\} & i < j \end{cases}$$

自底向上的方式：按照  $j-i$  的大小逐步向上生成  $m_{ij}$ ，需要三重循环（ $j-i$ 、 $j$ 、比较），并用额外数组  $s$  记录分割点。

```
def Matrix_Chain_Order(p):
    for i = 1 to n
        m[i][i] = 0
    for l = 1 to n-1
        for i = 1 to n-1
            j = i + 1
```

```

    m[i][j] = INFTY
    for k = i to j-1
        q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
        if q < m[i][j]
            m[i][j] = q
            s[i][j] = k
    return (m, s)

```

---

\*时间复杂度:  $O(n^3)$

计算最优解:

---

```

PRINT_RESULT(s, i, j):
    if i == j:
        print('A'i)
    else:
        print('(')
        PRINT_RESULT(s, i, s[i][j])
        PRINT_RESULT(s, s[i][j]+1, j)
        print(')')

```

---

思考: 如果要求顺序打出, 如何打印?

### §3.2 更多例子

#### 最长公共子序列

子序列定义: 从左到右取出的一列未必连续的元素。

问题: 已知两序列 X、Y, 求最长公共子序列 Z。

#### 1、最优子结构特性:

\*记  $X(i)$  为 X 前  $i$  个元素构成的子串,  $\langle A, B \rangle$  为 A 与 B 的最长公共子序列, X 为  $X[1..i]$ , Y 为  $Y[1..j]$ , Z 为  $Z[1..k]$

当  $X[i]=Y[j]$  时, Z 必为  $\langle X(i-1), Y(j-1) \rangle$  加上  $X[i]$ 。

否则, 当  $Z[k] \neq X[i]$  时, Z 为  $\langle X(i-1), Y(j) \rangle$ ; 当  $Z[k] \neq Y[j]$  时, Z 为  $\langle X(i), Y(j-1) \rangle$ 。

于是, 为进行求解, 需要对每个  $a, b$  考虑  $\langle X(a), Y(b) \rangle$ , 记其长度为  $e(a, b)$ 。

#### 2、构造递推解法:

$$e(a, b) = \begin{cases} e(i-1, j-1) + 1 & x_a = y_b \\ \max\{e(a-1, b), e(a, b-1)\} & x_a \neq y_b \end{cases}$$

边界条件:  $ab = 0$  时  $e(a, b) = 0$ 。

#### 3、自底向上构造:

---

```

def LCS_LENGTH(X, Y):
    m = length(X)
    n = length(Y)

```

```

c = zeros(m+1,n+1)
for i = 1 to m, j = 1 to n:
    if X[i] == Y[j]:
        c[i][j] = c[i-1][j-1] + 1
        b[i][j] = "upleft"
    elif c[i-1][j] > c[i][j-1]:
        c[i][j] = c[i-1][j]
        b[i][j] = "up"
    else:
        c[i][j] = c[i][j-1]
        b[i][j] = "left"
return b, c

```

---

#### 4、递归寻找最优解：

最优解的值为  $c$  右下角元素。从右下角开始按照  $b$  中指示行动，直到到达点的  $c$  为  $\theta$ ，每次向左上走的对应的  $X$  或  $Y$  下标构成最优解 [注意得到的序列需要进行 `reverse`]。

#### 最优二分检索树

假设有  $n$  个互不相同的关键字  $k[1..n]$  从小到大排列，此外要查找的关键字在  $k[i]$  与  $k[i+1]$  之间 (查找不成功) 的情况记作  $d[i]$ ，可能出现  $d[0..n]$ 。

假定  $p[1..n]$  为查找  $k[1..n]$  的概率， $q[0..n]$  为结果为  $d[0..n]$  的概率，使得期望查找次数 [即每个结果所在的深度加一与概率相乘求和] 最小的二分检索树称为**最优二分检索树**。

**\*\*事实上查找失败时所需要的比较次数即为深度，只有查找成功时才需要加一，教材为方便而统一为深度加一**

将子问题定义为  $K(i, j)$ ，代表对  $b[i..j+1]$  与  $k[i..j]$  构造的二分检索树。假定某树是最优二分检索树，其左子树与右子树一定是对应子问题的最优二分检索树 [类似矩阵链乘，运算顺序本就相似于构造二叉树]。

于是，计算可发现代价  $m[i, j] = \min_r \{m[i, r-1] + m[r+1, j] + \sum_{s=i}^j p_s + \sum_{s=i-1}^j q_s\}$ ，且边界条件  $m[i, i-1] = q_{i-1}$ 。

\*可先用  $w(i, j)$  保存  $\sum_{s=i}^j p_s + \sum_{s=i-1}^j q_s$ ，从而不用每次都计算

---

```

def Optimal_BST(p, q, n):
    for i = 1 to n+1
        e[i][j-1]=q[i-1]
        w[i][j-1]=q[i-1]
    for l = 0 to n-1, i = 1 to n-1
        j = i + 1
        e[i][j] = INFTY
        w[i][j] = w[i][j-1] + p[j] + q[j]
        for r = i to j
            t = e[i][r-1] + e[r+1][j] + w[i][j]
            if t < e[i][j]
                e[i][j] = t

```

```

    root[i][j] = r
return e, root

```

---

\*复杂度  $\Theta(n^3)$

\*事实上可以做到  $n^2$  复杂度：将第三重循环变为  $root[i][j-1]$  到  $root[i+1][j]$ ，利用抵消可以发现总和复杂度为  $\Theta(n^2)$ ，合理性需要严格的数学证明

### §3.3 贪心算法

基本思路：每一步按照一定准则做出看上去最优的决策，不会再行修改

贪心准则：做出贪心决策的依据

实现过程：从初始解出发，每一步求出可行解的一个解元素。由所有元素组合成可行解

\*\*贪心算法能求解时动态规划必然能求解，但贪心算法效率更高

例：活动安排问题

给定  $S$  由  $n$  个活动  $a[1..n]$  构成，每个有开始时间  $s[i]$  与结束时间  $f[i]$ ，开始时间小于结束时间，求尽量多的不会冲突的活动选择。

\*不妨设按结束时间  $f[i]$  从小到大排序，增添开始结束都在  $0$  时间的与无穷时间的活动，以使下标范围  $[0..n+1]$ 。记  $S_{ij}$  为所有在第  $i$  个结束后开始，第  $j$  个开始前结束的活动，容易发现只在  $i < j$  时可能不为空。

利用动态规划的求解思路，假设  $S_{ij}$  中的最佳安排包含  $a[k]$ ，则其必然为  $A_{ik}, A_{kj}$  与  $a[k]$  之并（由此，考虑  $A_{ij}$  的元素个数  $c[i][j]$ ，可通过尝试  $S_{ij}$  中的所有  $a[k]$  取最大值自底向上构造）。

贪心算法假设：只需要寻找使得  $A_{ik}$  为空的  $a[k]$  即可 [取  $S_{ij}$  中下标最小的活动]。

证明思路：通过替换可说明其一定可以包含此元素。

递归写法：

---

```

def RECURSIVE_ACTIVITY_SELECTOR(s, f, i, j)
    m = i + 1
    while m < j and s[m] < f[i]
        m++
    if m < j
        return {a[m]} ∪ RECURSIVE_ACTIVITY_SELECTOR(s, f, m, j)
    return ∅

```

---

\*也即每次找到最早完成的活动，并去掉与之冲突的

非递归写法：

---

```

def ACTIVITY_SELECTOR(s, f):
    n = length(s)
    A = {a[1]}
    i = 1
    for m = 2 to n
        if s[m] >= f[i]
            A = A ∪ {a[m]}
            i = m

```

```
return A
```

---

\*复杂度为  $\Theta(n)$ ，而动态规划方法的复杂度为  $\Theta(n^2)$

### 设计贪心算法

\*可先考虑递归问题的情况，找到其中正确的贪心选择后构造

1. 将最优子问题简化为做出选择后只剩一个子问题需要求解的形式。
2. 证明贪心选择后原问题存在最优解（即选择安全）。
3. 证明贪心选择后剩余子问题满足性质。

于是，贪心算法需要最优化问题具有**最优子结构**与**贪心选择性质**，后者即可以通过**局部**最优解构造出**全局**最优解。

### 贪心算法与动态规划对比

**0/1 背包问题**：已知每件物品的重量与价值，背包有承重上限，求最大价值装法。

**分数背包问题**：条件同上，但每个物品可以拿取一部分。

\*第二个问题可以直接每次选择单位重量价值最高的物品，于是具有贪心结构，但第一个问题如此可能导致最终剩余空间不是最优（类似钢条切割时），只能采用动态规划 [其可以被拿取和不拿的子问题覆盖，于是动态规划是可行的]。

### 例：哈夫曼编码

要求：进行**变长**编码以缩减储存空间，但需要**前缀编码**（两个不同的编码具有不同前缀，于是可以区分）以避免歧义。已知每个字符出现的频率，求最优编码方式。

**前缀码实现方式**：构造**满**二叉树，需要编码的字符在叶结点，从根结点出发，向左表示 **0**，向右表示 **1**，靠到达每个叶结点的方式进行编码。

\*于是问题转化为最优二叉树的构建

**算法 [自底向上]**：将每个结点的权值记作其左右孩子的权值之和，叶结点的权值即为频率。从每个字符作为根结点出发，每次新建根结点，将权值最小的两棵树作为其左右子树，重复 **n-1** 次。

\*利用**优先队列**进行存储，方便每次取出最小值

---

```
def HUFFMAN(C):
    n = length(C)
    Q = C
    for i = 1 to n-1
        z = new node
        left[z] = x = EXTRACT_MIN(Q)
        right[z] = y = EXTRACT_MIN(Q)
        f[z] = f[x] + f[y]
        INSERT(Q, z)
    return EXTRACT_MIN(Q) # return the root
```

---

\*根据之前的优先队列分析可发现复杂度为  $O(n \log n)$

**证明最优性**

引理：若  $x, y$  是出现最少的两个字符，则存在一种最优编码使得  $x, y$  对应的编码长度相同且只有最后一位不同。

证明：由于满二叉树性质，深度最深处必然会有一对叶结点，将  $x, y$  与这对结点替换后情况不会变差。利用引理，每次将建立的根下的叶结点合并为一个字符即可得到结果。

**§3.4 分治策略案例****硬币问题**

16 个硬币，找到其中比其他轻的假币。

直接思路：分为八组比较得结果。

分治思路：每次分两组比较，四次得到结果。

**大整数乘法**

\* 假设位数为  $n$

直接相乘： $\Theta(n^2)$  次，按照每位相乘后相加。

分治：

将其分为前  $n/2$  与后  $n/2$  位 [记  $2^{n/2} = m$ ]，但直接计算导致  $T(n) = 4T(n/2) + O(n)$ ，无法改进。

改进方式：设  $X = am + b, Y = cm + d$ ，可发现  $XY = acm^2 - ((a-b)(c-d) - ac - bd)m + bd$ ，于是如此计算可以达到  $T(n) = 3T(n/2) + O(n)$ ，复杂度  $O(n^{\log 3})$ 。

\*中间采用减法而非加法是为了避免溢出

\*\*更快方法？利用快速傅里叶变换 [FFT]， $O(n \log n)$  解决

**Strassen 矩阵乘法**

\*方阵相乘问题

分块矩阵可得  $\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$  与直接看为每位相乘的形式一致，但是直接计算会导致  $T(n) = 8T(n/2) + O(n^2)$ ，无法改进。

改进方式：利用更好的组合策略

考虑

$$D = A_1(B_2 - B_4), E = A_4(B_3 - B_1), F = (A_3 + A_4)B_1, G = (A_1 + A_2)B_4$$

$$H = (A_3 - A_1)(B_1 + B_2), I = (A_2 - A_4)(B_3 + B_4), J = (A_1 + A_4)(B_1 + B_4)$$

则

$$C_1 = E + I + J - G, C_2 = D + G, C_3 = E + F, C_4 = D + H + J - F$$

于是递推式变为  $T(n) = T(n/2) + cn^2$ ，复杂度降至  $n^{\log 7}$ 。

\*\*更快方法？利用其他划分策略，目前最好上界约  $O(n^{2.376})$

**残缺棋盘问题**

$2^n \times 2^n$  差一块的棋盘，要求用 3 小块的 L 形木板覆盖，求覆盖策略。

分治思路：分为四个，考虑残缺方格所在的区域，其他三个趋于用一个小 L 形盖住中间使得剩下每区域恰好残缺一个。

复杂度位为  $t(k) = 4t(k-1) + c$ ，于是复杂度与格子数一致。由于这和 L 形板数目相同量级，这必然是最优的算法。

### 距离最近点对

平面中  $n$  个点，求最近点对。

直接法：挨个比较，复杂度  $\Theta(n^2)$ 。

分治思路：分为两组，每组内比较并组间比较。

优化方式：优化组间比较。先按一个平行  $x$  轴的直线分割，假设两边最近点对较小距离为  $d$ ，则组间比较只需要考虑到直线距离小于  $d$  的点。

\*复杂度分析：由于最复杂的部分是对单坐标轴进行若干次排序，复杂度只需  $O(n \log n)$ 。

### §3.5 快速傅里叶变换

考虑范围：代数域 [可以考虑实数/复数] 上次数小于  $n$  的多项式

相加：直接系数相加， $O(n)$ 。

相乘：直接计算的简单方法  $\Theta(n^2)$  (实际上是计算向量的卷积)。

\*如何达到  $\Theta(n \log n)$ ?

多项式表示方式：系数表示法 (直接表示)、点值表示法 ( $n$  个不同点的取值确定)

点值表示好处：假设知道足够多点，多项式相乘只需要  $\Theta(n)$  次

表示方法转化

\*点值到系数?

做法：计算多项式插值

(考虑范德蒙德行列式可以知解的唯一性，从而点值唯一确定系数)

利用拉格朗日插值公式可以  $\Theta(n^2)$  可以得到结果

系数到点值直接方法： $a_0 + a_1(x + a_2(x + \dots))$  计算不同点，复杂度  $\Theta(n^2)$ 。

\*FFT 思路：选择合适的点让结果为  $O(n \log n)$ ?

#### 对数复杂度方法

数学基础： $n$  次单位根  $\omega_n^k = e^{2\pi ki/n}$ ， $k$  取 1 时称为单位原根，其他为原根的方幂。所有  $n$  次单位根构成循环群，单位原根为其生成元之一。

\*一些等式： $\omega_{dn}^{dk} = \omega_n^k, 2 | n \Rightarrow \omega_n^{n/2} = -1, \sum_{k=0}^{n-1} \omega_n^{dk} = \begin{cases} n & n | d \\ 0 & n \nmid d \end{cases}$

\*利用单位根作为系数到点值的特殊点

对多项式  $a(x) = a_0 + \dots + a_r x^r$ ，可不妨设其次数在  $n = 2^t$  以下，则令  $y_i = f(\omega_n^i)$ ， $i = 0, \dots, n-1$ ，称  $\mathbf{y}$  为  $\mathbf{a}$  的傅里叶变换，记作  $\mathbf{y} = \text{DFT}_n(\mathbf{a})$ 。

\*计算 DFT 的复杂度是  $\Theta(n \log n)$  的：

记  $a_0, a_2, \dots$  构成  $\mathbf{b}$ ， $a_1, a_3, \dots$  构成  $\mathbf{c}$ ，则  $\mathbf{a}(x) = \mathbf{b}(x^2) + x\mathbf{c}(x^2)$ 。

$n$  为偶数时  $(\omega_n^k)^2 = \omega_{n/2}^k$ ，于是通过分治可知 DFT 复杂度  $T(n) = 2T(n/2) + \Theta(n)$ ，由此为  $\Theta(n \log n)$ 。

---

```
def Recursive_FFT(a):
    n = length(a)
    wn = exp(2PI*i/n)
    w = 1
    aeven = {a[0], a[2], ..., a[n-2]}
    aodd = {a[1], a[3], ..., a[n-1]}
    yeven = Recursive_FFT(aeven)
    yodd = Recursive_FFT(aodd)
```

```

for k = 0 to n/2-1
  y[k] = yeven[k] + w * yodd[k]
  y[k+n/2] = yeven[k] + w * yodd[k]
  w = w * wn
return y

```

\*如何计算逆变换?

由于变换可以看作  $y = V_n a$ , 其中  $(v_{ij})_n = \omega_n^{ij}, i, j = 0, \dots, n-1$ , 计算可得  $V_n^{-1}$  的  $(i, j)$  位置为  $\frac{1}{n}\omega_n^{-ij}$  于是,  $a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj}$ , 可以完全类似利用分治得到  $\Theta(n \log n)$  的解法。

\*优化: 利用蝴蝶操作 [butterfly operation] 计算公用子表达式, 归并方式计算, 从而减少计算次数

\*下方算法中 BIT\_REVERSE\_COPY 为按二进制位逆序

```

def ITERATIVE_FFT(a):
  BIT_REVERSE_COPY(a, A)
  n = length(a)
  for s = 1 to log n
    m = 2**s
    omegam = exp(2PI*i/m)
    for k = 0 to n-1 by m
      omega = 1
      for j = 0 to m/2-1
        t = omega * A[k+j+m/2]
        u = A[k+j]
        A[k+j] = u + t
        A[k+j+m/2] = u - t
      omega *= omegam
  return A

```

## 四 数据结构

### §4.1 二分检索树

要求: 左子树  $\leq$  根  $\leq$  右子树

检索算法: 按照与根结点的比较向左/右走

最小/最大: 根结点不断寻找左/右孩子, 不存在时即为最小/最大, 复杂度  $O(h)$ 。

后继: 分为有右孩子与无右孩子讨论, 复杂度  $O(h)$

```

def TREE_SUCCESOR(x):
  if x.right
    return TREE_MINIMUM(x.right)
  y = x.p
  while y and x == y.right
    x = y
    y = y.p

```

```
return y
```

---

\*前驱算法与后继对称

二分检索问题: 对有序数组, 元素按关键字从小到大排列, 任给关键字, 查找出位置 (或不存) 并返回。  
许多具体实现方法 (折半查找、**Fibonacci 查找**等)

---

```
def Binary_Search(A, K, low, high):
    if high < low
        return 0
    else mid = (low + high) / 2
        if K = A[mid]
            return Binary_Search(A, K, low, mid-1)
        else
            return Binary_Search(A, K, mid+1, high)
```

---

\*正态分布数据, 不对半分: 将 mid 每次更新为  $low * t + high * (1-t)$

\*\*平衡二叉树与平衡旋转 (思考: 要使 LL、LR、RR、RL 旋转各发生一次, 至少需要在空二叉树上插入多少个结点? )

插入结点

---

```
def TREE_INSERT(T, z):
    y = NIL; x = T.root
    while x
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    z.p = y
    if !y
        T.root = z
    else if z.key < y.key
        y.left = z
    else
        y.right = z
```

---

删除结点

(数据结构书) 基本思路: 无孩子直接删除, 单孩子连接后删除, 两个孩子复制为后继后删除后继结点  
(算法书) 具体实现分为四种情形:

#### 1. 无左孩子

直接用右孩子替换 z (无论是否为 NIL)

## 2. 有左孩子无右孩子

直接用左孩子替换  $z$

3. 有两个孩子, 后继  $y$  是  $z$  的右孩子

用  $y$  替换  $z$ , 并仅留下  $y$  的右孩子 (此时  $y$  不可能有左孩子)

4. 有两个孩子, 右孩子  $y$  不是  $z$  的右孩子

用  $y$  的右孩子替换  $y$ , 并用  $y$  替换  $z$

子算法: 替换

---

```
def TRANSPLANT(T, u, v):
    if !u.p
        T.root = v
    else if u == u.p.left
        u.p.left = v
    else u.p.right = v
    if v
        v.p = u.p
```

---

(主算法直接利用替换与上方逻辑构造即可)

\*插入与删除的时间复杂度亦为  $O(n)$

\*\*数据随机分布时高度期望为  $O(\log n)$

## §4.2 红黑树

二分检索树, 此外每个结点另外保存一位结点的颜色, 红或黑

\*没有一条路径比另一条长出两倍, 因此近似平衡

结点包含五个属性: color、key、left、right、p [三叉链表]

[left 或 right 的 NIL 视为叶结点, 其余结点为内部结点]

## 1. 每个结点红色或黑色

## 2. 根结点黑色

## 3. 叶结点 (NIL) 为黑色

## 4. 红结点的子结点为黑色

## 5. 每个结点到所有后代叶结点的简单路径上黑结点数目相同

\*此数目称为结点的黑高度 [bh], 包含叶结点, 不包含自身

\*树的黑高度为根结点黑高度

性质:  $n$  个内部结点的红黑树高度不会超过  $2\log(n+1)$

证明思路: 归纳说明  $x$  为根的子树至少  $2^{bh(x)} - 1$  个结点, 又由红结点子结点为黑可知树高度不超过黑高度两倍, 从而得证。

\*由此之前对二分检索树的静态操作在红黑树上时间复杂度  $O(\log n)$ , 但插入删除可能需要调整颜色 [不过事实上仍可限制在  $O(\log n)$ ]

### 旋转

\*保持红黑性质的操作，用于插入、删除

$y$  左孩子  $x$ ，右子树  $c$ ， $x$  左右子树为  $ab$ ，则  $y$  为中心右旋后  $x$  顶替  $y$ ，且  $x$  左子树  $a$ ，右孩子  $y$ ， $y$  左右子树  $bc$ ，左旋则将此操作反向。

---

```
def LEFT_ROTATE(T, x):
    y = x.right
    x.right = y.left
    if y.left != T.nil:
        y.left.p = x
    y.p = x.p
    if x.p == T.nil:
        T.root = y
    elif x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.left = x
    x.p = y
```

---

右旋与左旋类似，复杂度  $O(1)$

### 插入

\*需要在正常插入后进行调整

正常插入过程与普通二叉搜索树 `INSERT` 完全相同，且置新结点颜色为红色。但是插入后需要调用 `RB_INSERT_FIXUP` (因为此时可能出现新结点与父结点均为红):

---

```
def RB_INSERT_FIXUP(T, z):
    while z.p.color == RED:
        if z.p == z.p.p.left:
            y = z.p.p.right
            if y.color == RED:
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
            z = z.p.p
        else:
            if z == z.p.right:
                z = z.p
                LEFT_ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT_ROTATE(T, z.p.p)
```

```

else:
    y = z.p.p.left
    if y.color == RED:
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
    else:
        if z == z.p.left:
            z = z.p
            RIGHT_ROTATE(T, z)
        z.p.color = BLACK
        z.p.p.color = RED
        LEFT_ROTATE(T, z.p.p)
T.root.color = BLACK

```

---

\*通过复杂的分情况讨论验证正确性，由于每层最多  $O(1)$ ，最终复杂度  $O(\log n)$

### 删除

\*先调用二分检索树删除，类似并进行替换 [此处需要注意颜色的传递与二分检索树 `NIL` 与红黑树 `T.nil` 的区别]

\*只有在真正删除的结点 `y` 是黑色结点时才需要进行 `fixup`[由于 `y` 的颜色可能改变，需要记录原始颜色]

删除黑色结点时，若其为原来的根结点，而红色孩子成为根结点，则违反性质 2；若 `x` 与 `x.p` 均红，违反性质 4；由于 `y` 的任何祖先经过 `y` 的路径上黑结点个数减小 1，违反性质 5 [做法：先认为占有 `y` 原来位置的结点 `x` 有两重黑色，再去掉一层]。由此构造算法：

---

```

def RB_DELETE_FIXUP(T, x):
    while x != T.root and x.color == BLACK
        if x == x.p.left
            w = x.p.right
            if w.color == RED
                w.color = BLACK
                LEFT_ROTATE(T, x.p)
            w = x.p.right
            if w.left.color == BLACK and w.right.color == BLACK
                w.color = RED
                x = x.p
        else
            if w.right.color == BLACK
                w.left.color = BLACK
                w.color = RED
                RIGHT_ROTATE(T, w)
            w = x.p.right

```

```

    w.color = x.p.color
    x.p.color = BLACK
    w.right.color = BLACK
    LEFT_ROTATE(T, x.p)
    x = T.root
else
    w = x.p.left
    if w.color == RED
        w.color = BLACK
        RIGHT_ROTATE(T, x.p)
        w = x.p.left
    if w.right.color == BLACK and w.left.color == BLACK
        w.color = RED
        x = x.p
    else
        if w.left.color == BLACK
            w.right.color = BLACK
            w.color = RED
            LEFT_ROTATE(T, w)
            w = x.p.left
        w.color = x.p.color
        x.p.color = BLACK
        w.left.color = BLACK
        RIGHT_ROTATE(T, x.p)
        x = T.root
x.color = BLACK

```

---

\*类似插入算法，分析可知复杂度  $O(\log n)$

### §4.3 动态顺序统计

\*数据结构扩张一般会增加维护成本，而希望维护成本可控

\*例：扩张红黑树为顺序统计树以快速查找

在红黑树中添加域 `size`，代表子树中内部结点个数（含自身），而空结点大小视为  $\emptyset$   
 查找第  $i$  小元素：

---

```

def OS_SELECT(T, i, x):
    r = x.left.size + 1
    if i == r
        return x
    if i < r
        return OS_SELECT(x.left, i)
    return OS_SELECT(x.right, i - r)

```

---

判断结点序号：

---

```
def OS_RANK(T, x)
    r = x.left.size + 1
    y = x
    while y != T.root
        if y = y.p.right
            r = r + y.p.left.size + 1
        y = y.p
    return r
```

---

\*插入、删除方式：实际改变的结点递归向上更新 size

一般扩张方式：选择基础数据结构-> 添加所需域-> 检验基本修改操作维护附加信息-> 添加新操作

\*扩张红黑树，若新增域的值只依赖左右孩子，则插入删除可以保证在  $O(\log n)$  完成

思路：每次插入/删除最多向上更新至根

### 区间树

扩张红黑树使之支持区间构成的动态集合上的操作

\*区间用有序实数对表示，均视为闭区间，相交当且仅当  $i.high \geq j.low$  and  $j.high \geq i.low$ ，记为  $Overlap(i, j)$

\*每个结点包含一个区间，查找操作为查找任意一个与给定区间相交的区间（或不存在）

排序依据：结点对应的区间  $[int]$  的左端点

添加域：max，代表以某结点为根的子树中各区间右端点最大值

区间维护： $x.max = \max(x.int.high, x.left.max, x.right.max)$

---

```
def INTERVAL_SEARCH(T, i):
    x = T.root
    while x != T.nil and !Overlap(i, x.int)
        if x.left != T.nil and x.left.max >= i.low
            x = x.left
        else
            x = x.right
    return x
```

---

\*证明思路：考虑结点处向左向右都不会错过区间

## §4.4 斐波那契堆

可合并堆 [最小堆]：包含优先队列操作，并且允许合并两个建立新堆 [基本操作建堆、插入、返回最小、删除最小、合并]

斐波那契堆：额外允许减小关键字、删除

\*\*[真正实用] 二项堆实现可合并堆操作（斐波那契堆程序设计复杂、常数项大，事实上一般不如二项堆）

斐波那契堆定义：一系列具有最小堆序的有根树集合，每个结点属性：

1.  $x.p$  指向父结点的指针

2. `x.child` 指向某孩子的指针 [所有孩子连成双向循环列表, 称为 `child list`]

3. `x.degree` 孩子数目

4. `x.mark` 上一次成为另一个结点孩子后是否失去过孩子 [初始为 `false`]

\*所有的根也组织成环形双向列表 [`root list`], `H.min` 指向所有根中最小的, `H.n` 表示结点数目

\*为进行摊还分析, 定义堆 `H` 的势函数  $\Phi(H) = t(H) + 2m(H)$ , `t` 为树的数目, `m` 为被标记结点的数目

建堆直接建空, `H.min` 为 `NIL`, 插入结点:

---

```
def FIB_HEAP_INSERT(H, x):
    x.degree = 0; x.p = x.child = NIL; x.mark = FALSE
    if !H.min:
        create a root list for H containing just x
        H.min = x
    else:
        insert x in H's root list
        if x.key < H.min.key:
            H.min = x
    H.n++
```

---

\*复杂度  $O(1)$ , 势函数增加 1

合并

---

```
def FIB_HEAP_UNION(H1, H2):
    H = MAKE_FIB_HEAP()
    H.min = H1.min
    concatenate the root list of H2 with the root list of H
    if !H1.min or (H2.min and H2.min.key < H1.min.key):
        H.min = H2.min
    H.n = H1.n + H2.n
    return H
```

---

\*势函数为 `H1` 与 `H2` 直接求和, 不变

提取最小

---

```
def FIB_HEAP_EXTRACT_MIN(H):
    z = H.min
    if z != NIL
        for each child x of z:
            add x to the root list of H
            x.p = NIL
        remove z from the root list of H
```

```

    if z == z.right:
        H.min = NIL
    else:
        H.min = z.right
        CONSOLIDATE(H)
    H.n -= 1
    return z

```

---

\*最复杂处: CONSOLIDATE, 不断将度相同的根结点合并, 且维护 H.min

CONSOLIDATE 需要先计算出最大度数上界  $D(n)$ , 此上界为  $\log_{\phi} n, \phi = \frac{\sqrt{5}+1}{2}$ 。

证明思路: 记  $F_i$  为斐波那契数, 有  $F_{k+2} = 1 + \sum_{i=0}^k F_i$ , 而利用此性质可说明度为  $k$  时结点个数至少  $F_{k+2}$ , 再有斐波那契数列递推式可知结果。

---

```

def CONSOLIDATE(H):
    D = D(H.n)
    let A[0..D] be a new arrays initialed by NIL
    for w in root list of H:
        x = w; d = x.degree
        while A[d]:
            y = A[d]
            if x.key > y.key: swap(x, y)
            FIB_HEAP_LINK(H, y, x)
            A[d] = NIL
            d += 1
        A[d] = x
    H.min = NIL
    for i = 0 to D if A[i]:
        if !H.min:
            create a root list for H containing just A[i]
            H.min = A[i]
        else:
            insert A[i] into rootlist and set H.min if need

```

---

\*FIB\_HEAP\_LINK 将  $y$  从根移到  $x$  的孩子并将标记设为 false

\*复杂度分析: 抽取最小结点的实际工作量为  $O(D(n) + t(H))$ , 而摊还后势函数  $m(H)$  部分不变,  $t(H)$  最多  $D(n) + 1$ , 摊还代价  $O(\log n)$

关键字减值

---

```

def FIB_HEAP_DECREASE_KEY(H, x, k):
    if k > x.key: ERROR
    x.key = k
    y = x.p
    if y and x.key < y.key:

```

```

    CUT(H, x, y)
    CASCADING_CUT(H, y)
    if x.key < H.min.key
        H.min = x

```

\*CUT 为将结点放置到根结点并设置对应属性 ( $p = \text{NIL}$ ,  $\text{mark} = \text{FALSE}$ ); CASCADING\_CUT 为不断向上寻找, 只要结点  $\text{mark}$  为  $\text{TRUE}$  且不是根结点就进行切割, 若终止不是在根结点就将  $\text{mark}$  设置为  $\text{TRUE}$ 。

\*摊还代价为常数 (因为改变标记会减小势), 而删除结点只需要将关键字减为负无穷再提取最小

\*\*红黑树与斐波那契堆无论最坏还是摊还, 至少由一项重要操作时间复杂度  $O(\log n)$ , 又由于其基于关键字比较决定, 根据排序下界可知必然有  $\Omega(\log n)$  复杂度操作。而若关键字具有性质, 如有界整数, 则可以利用如 **van Emde Boas** 树来加快操作, 达到  $O(\log \log n)$  复杂度。

## §4.5 分离集合

分离集合定义: 将  $n$  个元素分成若干个不相交的集合, 需要能允许查找属于哪个、合并集合。数据结构由一些不交的动态集合组成, 每个集合需要一个代表元 [保证未修改时代表元不变]。

操作: 建立包含单元素的集合 [MAKE\_SET]、合并 [UNION]、确定在何处 [FIND\_SET]。

摊还: 假设建立单元素集合  $n$  次 [元素个数], 总操作  $m$  次, 以下的时间复杂度以此分析。

\*作用举例: 图的连通分量 [思考: 输出连通分支]

```

def CONNECTED_COMPONENTS(G):
    for each vertex v in G.V:
        MAKE_SET(v)
    for each edge (u, v) in G.E:
        if FIND_SET(u) != FIND_SET(v):
            UNION(u, v)

```

(判断是否在同一连通分支只需要确定 FIND\_SET 是否相等)

链表实现: 头结点作为代表, 每个元素有指针指向头结点

MAKE\_SET 与 FIND\_SET 均为  $O(1)$ , 只需要创建单结点链表/返回结点对应的头结点

UNION 较为复杂: 由于需要更新指向头结点指针, 对  $n$  个元素的集合复杂度为  $\Theta(n)$ , 考虑若干次 MAKE\_SET 之后若干次合并, 代价为  $O(n)$

若每次用短合并长, 复杂度可控制为  $O(m + n \log n)$ 。

森林实现: 根结点作为代表, 每个元素仅指向父结点, 根节点指向自己

关键: 通过按秩合并 [对每个结点维护高度上界, 使较小秩的根指向较大秩的根] 与路径压缩 [在查找过程中让路径上的结点父结点直接为根] 改进结果

```

def MAKE_SET(x):
    x.p = x; x.rank = 0
def UNION(x, y):
    LINK(FIND_SET(x), FIND_SET(y))

```

```

def LINK(x, y):
    if x.rank > y.rank:
        y.p = x
    else:
        x.p = y
        if x.rank == y.rank:
            x.rank += 1
def FIND_SET(x):
    if x != x.p:
        x.p = FIND_SET(x.p)
    return x.p

```

---

\*注意 FIND\_SET 中递归两趟法思想的应用

\*\*复杂度分析：最坏运行时间为  $O(m\alpha(n))$ ，其中  $\alpha$  增长极慢，因此可以视为对元素个数线性。

## 五 图论算法与串匹配

### §5.1 图的表示与遍历

图的表示：邻接矩阵 [顶点作为维度的  $01$  方阵，相连的有向边为 1]、邻接表 [每个顶点通过链表存储与其相连的顶点]

\*\*邻接表存放顶点序号而不是内容

广度优先搜索 [BFS]

\*利用队列，颜色标记是否访问过

---

```

def BFS(G, s):
    for each vertex u in G.V - {s}
        u.color = WHITE; u.d = INFTY; u.pi = NIL
    s.color = GRAY; s.d = 0; s.pi = NIL
    Q = EMPTYSET
    ENQUEUE(Q, s)
    while Q is not empty
        u = DEQUEUE(Q)
        for each v in G.Adj[u]:
            if v.color == WHITE:
                v.color = GRAY; v.d = u.d + 1; v.pi = u
                ENQUEUE(Q, v)
        u.color = BLACK

```

---

\*复杂度  $O(|V| + |E|)$

作用：最短路径 (其中 d 存储的即是 s 到某个结点的最短路径长度)

广度优先树：每个结点的 pi 属性成为广度优先树中的父结点

深度优先搜索 [DFS]

\*以下的算法是**遍历**，而广度优先若需要遍历需要在未搜索到的结点中继续寻找源结点

\*利用**时间戳**记录发现与离开结点的时间  $u.d$  与  $u.f$

---

```

def DFS(S):
    for each vertex u in G.V
        u.color = WHITE; u.pi = NIL
    time = 0
    for each vertex u in G.V
        if u.color == WHITE:
            DFS_VISIT(G, u)
def DFS_VISIT(G, u):
    time += 1; u.d = time; u.color = GRAY;
    for each v in G.Adj[u]:
        if v.color == WHITE:
            v.pi = u
            DFS_VISIT(G, v)
    u.color = BLACK; time += 1; u.f = time

```

---

\*复杂度  $O(|V| + |E|)$

\*每个结点的  $pi$  属性构成**深度优先树森林**

**括号化定理**: 每个结点的  $d$  域与  $f$  域形成的区间不会互相交叉。

证明: 不妨设其中某个结点先被发现, 则后发现的结点要么在先发现结点探索完后还未开始, 要么包含在先发现结点遍历完的过程中, 从而讨论得证。

\*深度优先森林中  $v$  为  $u$  的后代当且仅当发现  $u$  时有从  $u$  到  $v$  的白结点路径 (利用过程容易说明)

**边分类**:

1. 树边 深度优先森林中的边
2. 后向边 将结点连到祖先结点的边 (包括自环)
3. 前向边 将结点连到后代 (非孩子) 结点的边
4. 横向边 其他所有边

\*考虑结点处理先后可发现, **无向图**深度优先遍历的过程中只会出现树边与后向边

思考: 设计算法寻找无向连通图中所有**关键结点** [去掉就不连通的结点]

**拓扑排序**

定义: **有向无环图**中的一个满足存在边  $(u,v)$  则有  $u$  在  $v$  前的全序关系

算法: DFS 中设置  $u.f$  之前将  $u$  压入栈, 最后自顶向底输出栈即可。

\*利用结点访问结束顺序与拓扑排序顺序相反可以实现, 时间复杂度仍为  $O(|V| + |E|)$

\*有向图无环当且仅当深度搜索不产生后向边

**强连通分量**

定义: 有向图中的一个极大子集  $C$ , 满足其中任何两点互相可到达。

\*定义图的**转置**为邻接矩阵转置对应的图, 即顶点不变, 边的方向变为反向

算法：对  $G$  进行 DFS，在离开结点时将结点压入栈。将图变为  $G$  转置 [并清空上一轮计算结果]，按照从栈顶到栈底的顺序，对白色结点进行 DFS，得到的深度优先森林中每一棵树对应一个强连通分量。

\*事实上是以拓扑排序的次序进行访问

\*时间复杂度仍为  $O(|V| + |E|)$

\*\*利用合并强连通分量得到的分支图可以证明正确性

## §5.2 图论问题

### 最小生成树

定义：所有边权值之和最小的生成树。

基本思路：从空集开始不断添加边，直到成为生成树。

性质：最小生成树部分边集合  $A$  中涉及的顶点记为  $S$ ，其余为  $V-S$ ，取一条最短的连接两部分的边加入，添加后  $A$  必然还是最小生成树的部分边。

[此性质为所有算法的基本思路]

算法：

1. 破圈法 起初包含所有边，只要有圈，就去掉其最大边，直到不存在圈。
2. Kruskal算法 将所有边排序，每次从剩下的中选最小代价的，只要不会产生环路就加入。
3. Prim算法 每次选择使入选的边保持为一棵树的最小的边。
4. Sollin算法：起初每个树为独立顶点，边集合为空，每步每个顶点集各自选择从其连接到外部的最短的边，连接成新的顶点集。[时间复杂度很大，适合并行]

利用分离集合实现 Kruskal 算法：

---

```
def Kruskal(G):
    T = EMPTYSET
    for each v in G.V:
        MAKE_SET(v)
    sort edges of G.E by increasing edge weight w
    for each (u,v) in G.E:
        if FIND_SET(u) != FIND_SET(v):
            add (u,v) into T
            Union(u, v)
    return T
```

---

\*边集合可以用二叉堆实现，时间复杂度为  $O(|V| + |E| \log |E|)$ ，思考：可不可能为复杂度  $O(|V|^2)$ ?

Prim 算法：

---

```
def Prim(G, w, r):
    for each u in G.V
        u.key = INFTY; u.pi = NIL
    r.key = 0
    Q = G.V
    while Q != EMPTYSET:
        u = EXTRACT_MIN(Q)
```

```

for each v in G.Adj[u]:
    if v in Q and w(u,v) < v.key:
        v.pi = u
        v.key = w(u,v)

```

---

\*不断更新当前的最小距离，最终所有的 pi 看作父结点形成的树即为最小生成树

\*时间复杂度为  $O(|E| \log |V|)$ ，事实上由于  $|E| \in [|V| - 1, |V|^2]$ ，时间复杂度也可看作  $O(|E| \log |E|)$ ，与 Kruskal 算法量级相同

\*\*目前最小生成树算法已经能做到  $O(|E|)$  复杂度

### 单源最短路径问题

单源最短路径问题：给定带权有向图，求从图中一个特定源点出发到其余各个顶点的最短路径。

单源简单最短路径问题：给定带权有向图，求从图中一个特定源点出发到其余各个顶点的简单最短路径。

\*对前者，有权值小于 0 回路时无意义，对后者仍然有意义。只有前者满足最优性原理。

松弛操作：

---

```

def Relax(u, v, w):
    if (d[v] > d[u] + w):
        d[v] = d[u] + w

```

---

\*Dijkstra 算法不能对有负边的图使用

Bellman Ford 算法：

---

```

def Bellman_Ford(G, s)
    for each v in G.V:
        d[v] = INFTY
    d[s] = 0
    for i = 1 to length(G.V)-1:
        for each edge (u,v) in G.E:
            Relax(u, v, w(u,v))
    for each edge (u,v) in G.E
        if(d[v] > d[u] + w(u,v)) return NO_SOLUTION
    return TRUE

```

---

\*复杂度  $O(|V||E|)$ ， $|V| - 1$  次松弛保证任何边的影响扩展到了所有路径上

\*优化思路：通过合适的次序减少松弛的趟数

\*对带权有向无环图，按拓扑排序结果，从源点开始依次对其后各个顶点引出的边做松弛，则一趟松弛即可完成。

优先队列 Dijkstra：

---

```

def Dijkstra(G):
    for each v in G.V:
        d[v] = INFTY
    d[s] = 0; S = EMPTYSET; Q = V

```

```

while Q != EMPTYSET:
    u = EXTRACT_MIN(Q)
    add u to S
    for each v in G.Adj[u]:
        Relax(u, v, w(u,v))

```

\*由于优先队列存储，复杂度  $O(|E| \log |V|)$

思考：是否对任何权值非负无向连通图，可以通过 **Dijkstra** 算法选取合适源点得到最小生成树？

[答案：否定，考虑  $w_{ab} = w_{bc} = w_{cd} = 2, w_{ac} = w_{bd} = 3$ 。]

\*\*给定两点间的最短路径时间复杂度与单源最短路径相同，因为必须找到所有单源最短路径才能断定最短

### 所有点对间最短路径

无负边：直接使用 **Dijkstra** 算法，理论复杂度可达  $O(|V||E| \log |V|)$  或  $O(|V|^2 \log |V| + |E||V|)$  (取决于优先队列实现)。

化为矩阵乘法 [运算重载技巧]

\*\* 技巧意义大于实际应用

当前最短路径矩阵 **D**，记录每个结点的前驱矩阵 **Pi** 为  $(\pi_{ij})$ ， $\pi_{ij}$  表示 **i** 到 **j** 的最短路径中 **j** 的前驱结点。

\*通过不断  $j = \text{Pi}[i][j]$  可递归找到最短路径

用邻接矩阵 **w** 表示所有边权，对角为  $\theta$ ，不连接为无穷，递归寻找：

定义  $l_{ij}^{(m)}$  为 **i** 到 **j** 的途径至多 **m** 条边的最短路径长度，当 **m** 为 **1** 时即为 **w**，由定义发现递推关系  $l_{ij}^{(m)} = \min_k (l_{ik}^{(m-1)} + w_{kj})$ ，而最终所求的结果即为  $l_{ij}^{(n-1)}$ ，利用动态规划即可求解。

\*直接动态规划复杂度  $O(|V|^2|E|)$  或  $O(|V|^4)$

优化：将每次的  $L^{(m)}$  视为矩阵 **L**，重定义矩阵乘法：

原本乘法  $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$ ，将其中乘法重定义为加法，加法重定义为取最小值，则可类似矩阵乘法计算 (核心：由于加法对取最小值可分配，仍然具有结合律)，由于计算幂可以两两累加，效率能得到提升。

\*复杂度可以为  $|V|^3 \log |V|$

### Floyd-Warshall 算法

考虑  $l_{ij}^{(m)}$  为 **i** 到 **j** 的顶点序号最大为 **m** 的最短路径，这时有递推  $l_{ij}^{(m)} = \min\{l_{ij}^{(m-1)}, l_{im}^{(m-1)} + l_{mj}^{(m-1)}\}$ ，从而仍然利用动态规划只需  $O(|V|^3)$ 。

\***Pi** 的确定：若递推中左侧较小则不变，右侧较小则更新成为  $\pi_{mj}^{(m-1)}$

```

def FLOYD_WARSHALL(W):
    n = rows(W)
    D = W
    for k = 1 to n:
        for i = 1 to n, j = 1 to n:
            d[i][j] = min(d[i][j], d[i][k] + d[k][j])
    return D

```

\*空间复杂度  $\Theta(|V|^2)$ ，注意到更新顺序使得不需要新建 **D**，原地迭代即可

\*传递闭包：用或代替 **min**，与代替 **+**，**01** 邻接矩阵，计算后为传递闭包

**稀疏图 Johnson 算法 [权值重定义技巧]**

\*更新所有边的权值使其均非负，然后对每点利用 Dijkstra

定理：给每点  $x$  赋予权值  $h(x)$ ，边权  $w(u,v)$  更新为  $w'(u,v) = w(u,v) + h(u) - h(v)$ ，则更新后最短路径与更新前相同。

证明：由于路径中裂项相消，起点到终点的任何路径改变恒为  $h$  的差，因此最短不变。

\*如何构造  $h$ ?

加入新顶点  $s$ ，到任何点都连有权为  $\theta$  的边，对点  $x$ ，将  $s$  到  $x$  的最短路径值设为  $h(x)$ 。由定义可发现  $h(x)$  小于等于  $\theta$ ，当不存在负边时一定为  $\theta$ 。

\*这样赋予后利用三角不等式  $h(v) \leq h(u) + w(u,v)$  可以证明新权值非负

思考：为何需要加入新顶点  $s$ ? [未必存在某点能到达所有其他点]

于是，算法的过程为：加入  $s$ 、计算单源最短路径、更新边、计算所有点间最短路径，二叉堆实现复杂度为  $O(|V||E| \log |V|)$ ，稀疏图时优于 Floyd-Warshall。

**§5.3 串匹配算法**

串匹配问题：在文本  $[T, \text{text}]$  串中找到一个或所有模式  $[P, \text{pattern}]$  串

\*找所有只需要实现找单个后不断右移

$n$  代表  $T$  的长度， $m$  代表  $P$  的长度， $n$  一般远大于  $m$ 。 $\sigma, \Sigma, \Sigma^*$  代表字符集大小、字符集与字符串 [允许为空]， $C_n$  表示比较次数，一般作为衡量标准。

一些定义：若  $x$  存在分解  $wy$ ，则称  $w$  为  $x$  的前缀， $y$  为  $x$  的后缀，记作  $w \sqsubset x, y \sqsupset x$ 。对  $P[1..m]$ ，记  $P_k$  代表其前  $k$  个字符， $T_k$  同理。

后缀重叠引理：若  $x \sqsupset z, y \sqsupset z$ ，则  $|x| \leq |y| \Rightarrow x \sqsupset y, |x| \geq |y| \Rightarrow y \sqsupset x, |x| = |y| \Rightarrow x = y$ 。

\*算法基本原理为滑动窗口机理，利用大小等于模式长度的窗口对文本串进行扫描，每次匹配后按需要右移窗口，直到超出文本右端。

1. 从左到右 Rabin-Karp, KMP
2. 从右到左 Boyer-Moore, Horspool
3. 任意顺序 Brute Force(原始算法)

**Brute Force**

\*每个位置检查，最坏情况  $O(mn)$

---

```
def BRUTE_FORCE(T, P):
    i = 0
    while i <= n - m:
        j = 0
        while j < m and P[j+1] == T[i+j+1]: j++
        if j == m: report match at i-j+1
        i++
```

---

\*如何避免重复计算的发生? [保存部分匹配信息]

**KMP 算法**

\*思想: 当前指针只右移不后退, 遇到不成功匹配后充分利用前一部分信息。

先计算  $\text{Next}[j+1] = \max(k+1, \text{使得 } P[1..k] \text{ 是 } P[1..j] \text{ 的后缀})$

---

```
def Next(P):
    j = 0
    for i = 1 to m:
        Next[i] = j
        while j > 0 and P[i] != P[j]:
            j = Next[j]
        j++
```

---

\*复杂度  $O(m)$

\*\*书中定义的  $\text{pi}[i] = \text{Next}[i+1] - 1$

---

```
def KMP(T, P):
    j = 1
    for i = 1 to n:
        while j < 0 and T[i] != P[j]:
            j = Next[j]
        if j == m: return i - m + 1
        j++
    return None
```

---

\*总复杂度  $O(m+n)$ , 当重复多时效果更好 (如二进制文件)

**Boyer-Moore 算法**

思路: 每次自右向左匹配后缀, 确定不匹配时最多能移动多少位, 移动情况分为  $G_s$ [好后缀] 与  $B_c$ [坏字符] 两种。

**Horspool 的简化**

\*当字符的可能性相对模式长度很大时 [如单词搜索], 坏字符移动效果很好

坏字符移动:  $P$  的某个后缀和  $T$  某段匹配, 更前一位并不匹配时, 设此时  $T$  中对应字符为  $b$ , 找到  $P$  中从右向左第一次出现  $b$  的位置, 并移动使两个  $b$  对齐 (若不出现, 直接移动过  $T$  中  $b$  的位置)。

---

```
def PRE_BC(P, m, Bc):
    for i = 1 to ASIZE: Bc[i] = m
    for i = 1 to m-1: Bc[P[i]] = m - i
def HORSOPPL(P, T):
    PRE_BC(P, m, Bc)
    i = m
    while i <= n:
        k = 1
        while k <= m and P[m-k+1] == T[i-k+1]:
            k++
```

```

    if k == m: report match at i-m+1
    else: i += BC[T[i]]

```

---

\*最坏情况时间复杂度  $O(mn)$ ，平均比较次数在  $(\frac{n}{\sigma}, \frac{2n}{\sigma+1})$

### Rabin-Karp 算法

思路：利用哈希函数检测文本与模式中“相似”的部分，并进一步检查匹配。

先写出 ord 函数将字符集对应为  $\theta$  到  $\text{sigma}-1$  的数，再构造哈希表，一个长度为  $m$  的词  $w$  的哈希函数定义为：

$$\begin{cases} x = \text{ord}(w) \\ \text{hash}(w) = \sum_i x_i d^{m-i} \pmod q \\ \text{rehash}(a, b, h) = (h - ad^{m-1})d + b \pmod q \end{cases}$$

其中  $q$  是某个大素数， $d$  为底数（一般可以取 2）

---

```

def KR(P, T):
    y = d ** m-1; Phash = 0; Thash = 0
    for i = 1 to m:
        Phash = (Phash * d + ord(P[i])) % q
        Thash = (Thash * d + ord(T[i])) % q
    j = 1
    while j <= n - m + 1:
        if Phash == Thash and memcmp(P, T+j-1, m) == 0:
            report match at j
        Thash = ((Thash - y*ord(T[j])) * d + ord(T[j+m])) % q
        j++

```

---

\*当期望的有效偏移很少时复杂度  $O(m+n)$