

Discrete

Mathematics - P 26

Exercises

Exercises

注意：矩阵的加法是可交换的 ($A+B=B+A$) 和可结合的 ($(A+CB)+C=(A+B)+C$)

而矩阵的乘法只可结合 ($A(BC)=(ABC)C$), 但不满足交换律

矩阵的乘法满足对加法的结合律, 即 $(A+B)C=AC+BC$, $C(A+B)=CA+CB$

矩阵的逆 (inverse), 对于 $n \times n$ 矩阵 A , 如果存在 $n \times n$ 矩阵 B , 使得 $AB=BA=I_n$, 则称矩阵 B 是矩阵 A 的逆

同时这样的 B 是唯一的, 而称 A 为可逆的. 同时用 A^{-1} 表示矩阵 A 的逆.

对于任意 $n \in \mathbb{Z}^+$, $(A^n)^{-1} = (A^{-1})^n$

如果 $n \times n$ 矩阵 A 和 B 均可逆, 则有 $(AB)^{-1} = B^{-1}A^{-1}$

$$\text{线性方程组 方程组 } \begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2, \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m. \end{cases} \text{ 可表示为 } AX = B \quad \left[\begin{array}{cccc|c} a_{11} & a_{12} & \dots & a_{1n} & | & x_1 \\ a_{21} & a_{22} & \dots & a_{2n} & | & x_2 \\ \vdots & \vdots & \ddots & \vdots & | & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & | & x_n \end{array} \right] = \left[\begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_m \end{array} \right]$$

其中 A 是 $n \times n$ 矩阵, X 为 $n \times 1$ 矩阵, B 为 $n \times 1$ 矩阵.

注意方程组有确定解的条件为矩阵 A 可逆, 且解为矩阵 $X = BA^{-1}$

乌拉姆数 (Ulam number), 定义 $U_1=1$, $U_2=2$, 在判断 $1 \leq k \leq n$ 的数是否为乌拉姆数之后,

如果 n 可被唯一地写作两个不同的乌拉姆数之和, 则 n 为下一个乌拉姆数.

有 $U_3=3$, $U_4=4$, $U_5=6$, $U_6=8$

$\lambda U_i + U_j = n$

即 $U_1=1$, $U_2=2$. $\forall n \in \mathbb{N}$ (n 是乌拉姆数 $\leftrightarrow \exists! (U_i, U_j) \in \mathbb{Z}^+ \times \mathbb{Z}^+ (U_i < U_j < n \wedge U_i, U_j \text{ 是乌拉姆数})$)

注意证明有无穷多个乌拉姆数的方为非构造性的, 假设 U_{n-1} 和 U_n 是最大的两个乌拉姆数.

令 $U = U_{n-1} + U_n$, 则如果 U 符合乌拉姆数定义, 与假设矛盾.

如果 U 不符合定义, 则存在另一组 (U_i, U_j) 使得 $U = U_i + U_j$,

而 U_i, U_j 中必有一个大于 U_n 的乌拉姆数, 与假设矛盾.

即不存在最大的乌拉姆数, 即乌拉姆数有无限多个.

算法

(algorithm): 进行一项计算或解决一个问题的准确指令的有限序列

是一个过程, 能够遵循一系列步骤导致找到所求的答案.

algorithm 的来源是 9 世纪的数学家花剌子密 (al-khowarizmi) 的通用

伪代码 (pseudo code) 介于过程步骤的自然语言描述和用实际编程语言描述过程的规范之间的中间体

算法步骤用模仿程序设计语言指令的伪命令来描述, 也可以使用任何良定义的运算或语句的指令

尽管伪代码不遵循任何编程语言的语法, 但易于理解, 且具有编程可用的运算逻辑

Discrete

Mathematics - P 27

principles

principles

- 算法性质
输入：算法从一个指定的集合得到输入值（合法输入，前置条件）
输出：对每个输入值集合，算法都要从一个指定的集合中产生输出值。
输出值即问题的解（合法输出，后置条件）
确定性：算法的步骤必须是准确定义的（不存在歧义）
正确性：对每一组输入值，算法都应产生正确的输出值（基本要求）
有限性：对任何输入算法都应在有限（可能很多）步之后产生期望的输出
(即其一是不可陷入死循环，其二是尽可能少的步数)
有效性：算法的每一步都应能精确地在有限时间内完成
通用性：算法过程应该可以应用于期望形式的所有问题。
而不只适用于一组特定的输入值

搜索算法 (searching algorithm), 指在一个列表中定位一个元素的问题
在不相同的元素 a_1, a_2, \dots, a_n 的列表中定位 x (即找到 $i \in \mathbb{Z}^+, 1 \leq i \leq n$, 使 $x = a_i$) 或判定 x 不在表内

线性搜索 (linear search algorithm) 或称顺序搜索算法, end [following] 顺序搜索
即从 a_1 开始顺序与 x 比较，不相等时继续比较下一项。

当匹配到 $a_i = x$ 时，则返回该项的位置 i ，

如果已比较所有项而没有匹配，则返回 0。

```
procedure linear search (int x, [a1, a2, ..., an])  
    i := 1  
    while (i <= n and x ≠ ai) do  
        i := i + 1
```

if i <= n then location := i

else location := 0

i := 1] 设置搜索区间初

return location

j := n] 左右端点

while i < j

二分搜索算法 (binary search algorithm). 针对有序列表

m := (i+j)/2

即通过比较 x 与有序列表 a_1, a_2, \dots, a_n 的中间项 a_m 。 if $x > a_m$ then $i := m+1$

而后通过比较的结果，选择对应的子列表

else $j := m$

如果大于中间项，则选取后半部分。 if $x = a_i$ then location := i

else location := 0

直到匹配到 $a_i = x$ 时，返回该位置； return location

如果仅剩一项且 $a_i \neq x$ ，则返回 0

Discrete

Mathematics - P 28

89 -

排序 (Sorting): 按既定的顺序重新排列一个数组中元素的次序，通常为升序

冒泡排序 (bubble sort): 最简单，但并非最有效的排序算法之一

通过连续比较相邻元素，如果相邻元素不正确则交换相邻元素

即较小的元素“冒泡”到顶端

for $j=1$ to $n-i$

较大的元素“下沉”到底端

从而把一个列表排列为升序

注意冒泡排序可以通过加入一个计数器以统计每次遍历的交换次数

(当计数器达到一定值时当一轮结束而计数器为0，则可以提前终止)

另外冒泡排序在实现中适用于数据量较大而无法复制或移动的情形

插入排序 (insert sort): 简单但通常不是最有效的排序算法

在插入排序的第*j*步上

Procedure insertion sort ($\{a_1, a_2, \dots, a_n\} \subseteq R$)

表的第*j*个元素被选中

并插入到已经排序的*j-1*个元素

中的正确位置上

$i := 1$

while $a_j > a_i$ $i := i + 1$ 找到使得 $a_{i-1} \leq a_j \leq a_i$

通常在此过程中使用线性搜索

$m = a_j$

$a_{i-1}, a_i \leftarrow m$

与冒泡排序相比，插入排序无法提前结束

for $k := 0$ to $j-i-1$

$a_{i-1}, a_i \leftarrow m$

因为对于*n*个元素必须循环*n-1*次

$a_j := a_{j-k}$

$a_{j-k+1} \leftarrow a_j$

但是可以通过将搜索子-1个元素改为

$a_i := m$

可改写为 $a_{k+i+1} := a_{k+i}$

二分搜索来改善效率

{ a_1, a_2, \dots, a_n 按升序排列} 或 $\text{for } k := i+1$ to j

最优化问题 (optimization problem): 目标是寻找给定问题是满足某个参数值最小化或最大化的解

贪婪算法 (greedy algorithm): 在每一步都选择看起来“最好的”选项，常常能导致最优化解的最简单方法

在每一步都选择最好的选项，而不是通盘考虑可能导致最优解的全部步骤序列

如找零问题，用尽可能少的硬币

procedure change C int n, $\{c_1, c_2, \dots, c_r\} \in \mathbb{Z}^+$

使用任意一组有限面值 $\{c_1, c_2, \dots, c_r\}$

for $i := 1$ to r

注意: $\forall i < r$ 由 $\{c_1, \dots, c_r\}$ 的硬币

$d_i := 0$ {统计面值为 c_i 的硬币数量}

构成的零钱小于 c_i 的面值

while $n \geq c_i$

即 $\forall i < r$ ($c_i > \sum_{k=i+1}^r c_k \cdot d_k$)

$d_i := d_i + 1$

由此可知贪婪算法产生使用尽可能少硬币的

$n := n - c_i$ 从 n 中减去 c_i 的面值

找零方案

return $\{d_1, d_2, \dots, d_r\}$

Discrete

Mathematics - P 29

59 - Hasch

贪婪算法根据某一个条件在每一步都做出最佳选择，但在多个条件中选择哪一个则可能难以确定。

如用贪婪算法来生成一个最佳调度 procedure schedule ($\{(s_i, e_i), (s_2, e_2), \dots, (s_n, e_n)\}$) \rightarrow Time

$(s_i < s_j) \wedge (e_i < e_j)$ 在有限的时间内安排讲座

使得安排数量尽可能多且

$$S = \emptyset$$

已安排的讲座

满足所有已安排的讲座之间没有时间冲突

for (s_i, e_i) in S do

此类问题也称为 Activity Selection (如果 (s_i, e_i) 与 S 相容 then $S = S \cup \{s_i, e_i\}$)

贪婪算法解决 Activity Selection Problem

return S 为解

注意：贪婪算法实际上可以视为 DFS (depth-first search)，深度优先搜索的一个简化版本。

即假设树叶为解且搜索中出现的第一个叶节点即为所求的解。

但是可能出现算法终止时得到的是无效解或者程序卡死而无法得到解。

如用 {5, 2} 找零 16，按照基本贪婪算法

会无法得到解，或得到不合法的解。

注意 $D = \{3, 0\}$ 是不合法的解

可以对贪婪算法进行改进，加入回退机制，使得算法可以得到有效解。

或者在条件允许的范围内搜索尽可能多的有效解并加以比较，以获得更优解。

可解问题 (Solvable problem)，指可以由算法，或者存在一个过程 (procedure) 可以求解的问题。

不可解问题 (Unsolvable problem)，相对地，指不能由算法求解，或者说不存在一个过程可以求解的问题。

停机问题 (Halting problem)，通过证明停机问题是不可解进而证明了存在不可解问题。

询问是否存在过程，对一个计算机程序和该程序的一个输入，可以判断给定程序在给定输入运行

首先知道无法通过简单观察来判断是否停机。

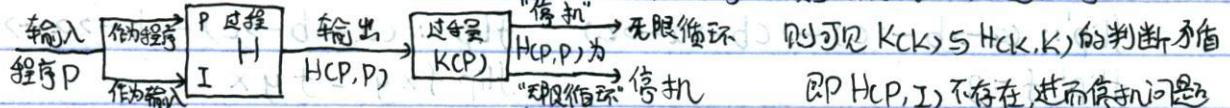
因为在任意固定时间 t 的运行后仍未停止，并无法保证程序是否永远不停止。

此处为一个并不严格 (未定义过程) 的反证法证明。

假设存在一个过程 $H(P, I)$ ，如果程序 P 在输入 I 时能够终止，则输出“停机”，否则“无限循环”。

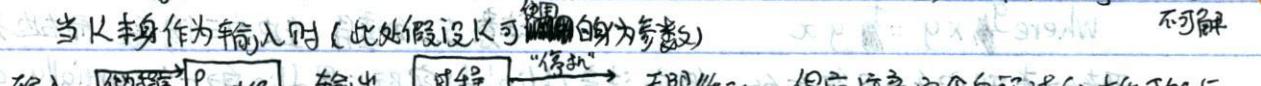
由于 P 本身可作为输入使用，则可以如 $H(P, P)$ 一样 (此处假定了 P 本身的数据 E)

而定义一个过程 $K(P)$ ，如果 $H(P, P)$ 结果为“停机”则无限循环，反之则停机。



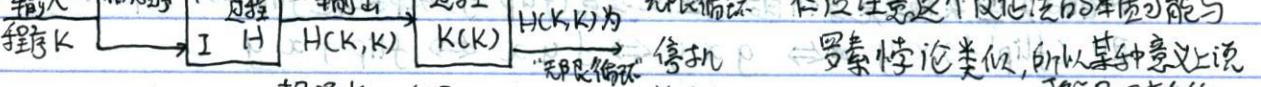
“停机” \rightarrow 无限循环，则可见 $K(K)$ 与 $H(K, K)$ 的判断矛盾。

即 $H(P, P)$ 不存在进而停机问题是



“无限循环” \rightarrow 停机 P ，即 $H(P, P)$ 不存在进而停机问题是

不可解



但应注意这个反证法的本质可能与

罗素悖论类似，所以某种意义上说

可能是无效的。

Discrete

Mathematics - P 30

$f = f_{\text{student}}$

三分搜索算法

在递增序表中通过连续地把表分成大小相等或尽可能相等的三个子表，并将搜索限制在一个合适的子表中的方法来定位元素。

```
procedure trinary search ({a1, a2, ..., ak} | a1 < a2 < ... < ak)
```

i := 1, j := k [设置搜索的表的起始与结束]

```
while i < j:
```

m = L(i+j)/3], n = L(i+j) × 2/3] [设置子表的分界点]

x < a_m if x < a_m, then j := m-1] 确定下一步搜索的

else if x < a_n then i := m, j := n-1] 合适的子表

else i := n]

if a_i = x then location := i else location := 0 [输出x的位置或输出0]

return location {x在{a₁, a₂, ..., a_n}中的位置}

选择排序算法(Selection sort)

通过不断地找出列表中的最小元素，并将其移动到列表的合适位置

即第i步：将列表中第i小的元素放在列表第i位上

```
procedure selection sort ({a1, a2, ..., an} ⊆ R)
```

i = for i := 1 to n-1

min := i

for j := i+1 to n

] 找到最小元素的索引值

[a_j ← a_i] ← i <= (if a_j < a_{min} then min := j)

temp := a_{min}

for k := i+1 to min-1

a_{k+i} := a_k

a_i := temp

return {a₁, a₂, ..., a_n}

二分插入排序

即用二分搜索算法代替插入排序中线性搜索部分的优化算法

```
procedure binary insertion sort ({a1, a2, ..., an} ⊆ R), x ∈ R
```

for j := 2 to n x = a_j

if x < a₁ then i := 1

for k := 1 to j-1

a_k = a_{k-1}

a₁ = x

a_{in} = m

m = a_{in}

a_{in} = a<sub

Discrete

Mathematics - P31

假设有 S 位男士 m_1, m_2, \dots, m_s 和 S 位女士 w_1, w_2, \dots, w_s , 期望为每人匹配一位异性

假设每人按自己的偏爱程度对异性进行排序, 且不存在并列

稳定的 (stable), 即将一组异性结合为夫妇的一个匹配是稳定的

当且仅当 $\forall m \in M \forall w \in W (m \text{ 和 } w \text{ 不是伴侣}) \wedge (m \text{ 喜欢 } w \text{ 胜过他的伴侣} \wedge w \text{ 喜欢 } m \text{ 胜过她的伴侣})$

$\rightarrow (\text{m喜欢他的伴侣胜过 } w \vee w \text{ 喜欢她的伴侣胜过 } m)$

稳定的匹配 (stable matching)

延迟接受算法 (deferred acceptance algorithm), 也称为Gale-Shapley算法, 用于构造稳定的男女匹配

定义一个性别的人为求婚者, 另一个性别的人为被求婚者

初始时, 每个求婚者都有一个待定的求婚者:

for each 上一回合被拒绝的求婚者

从该求婚者的排序中选择没有拒绝过且排名最靠前的对象

"向他求婚"

如果求婚者在该对象的排序中比该对象已选择的待定求婚者更靠前

then 该对象拒绝待定的求婚者, 并指定新求婚者为待定

否则将原待定的求婚者加入本回合被拒绝的求婚者

else 将该求婚者加入本回合被拒绝的求婚者

return 所有对象

注意: 延迟接受算法必定可以终止, 因为每个求婚者只求有限次婚

而且终止时必定能产生一个稳定匹配.

最直接是通过假设结束时存在求婚者 a_i 和被求婚者 b_j 不符合稳定匹配的条件

则有对于 (a_i, b_i) 和 (a_j, b_j) , 对 a_i 而言 $b_j > b_i$, 对 b_j 而言 $a_i > a_j$.

考虑求婚者 a_i 向 b_i 求婚必定发生于被 b_j 拒绝以后

而 b_j 不可能在无待定或待定 $< a_i$ 的情况下拒绝 a_i , 所以应有 $a_j > a_i$.

与假设矛盾, 与假设矛盾, 所以延迟接受算法终止时必定能产生一个稳定匹配.

大O记号 (big-O notation), 也称兰道符号, 用于估计当输入增长时, 一个算法所用的操作的数量

用于判定当输入规模增大时, 用一个特定算法求解某问题是否实际可行

$f(x) \text{ is } O(g(x))$, 表示对于从 N 到 R 的函数 f 和 g , $\exists (c, k) \forall x (x > k \rightarrow |f(x)| \leq c|g(x)|)$

即存在常数 c, k , 使得只要当 $x > k$ 时, 就有 $|f(x)| \leq c|g(x)|$

直观上说, 即当 x 无限增长时, $f(x)$ 的增长速度慢于 $g(x)$ 的某个固定倍数

凭证 (witness) 称常数 c 和 k 为 $f(x)$ 是 $O(g(x))$ 的凭证,

即只要找到一对凭证 (c, k) 使得 $\forall x (x > k \rightarrow |f(x)| \leq c|g(x)|)$ 即可证明 $f(x)$ 是 $O(g(x))$ 的凭证

注意: 当存在一对凭证 (c, k) 时, 即存在无数对凭证, $\forall c' > c \forall k' > k (c', k')$ 是 $f(x)$ 是 $O(g(x))$ 的凭证

Discrete

Mathematics - P. 32

Efficient

(same order) 如果 $f(x)$ 与 $g(x)$ 为同阶的，当且仅当 $f(x)$ is $O(g(x))$ 且 $g(x)$ is $O(f(x))$

注意： $f(x)$ is $O(g(x))$ 有时写作 $f(x) = O(g(x))$ ，需要注意此处等号不表示相等

或写作 $f(x) \in O(g(x))$ ，即用 $O(g(x))$ 表示 is $O(g(x))$ 的函数的集合

注意：如果有对 $x > k$, $|f(x)| > |g(x)|$ 且 $f(x)$ is $O(g(x))$ ，则有 $f(x)$ is $O(h(x))$

所以在使用大O记号时， $O(g(x))$ 中的 $g(x)$ 的选择应该尽可能小

$!(f(x) \text{ is } O(g(x)))$ 通常有两种方式证明 $f(x)$ is $O(g(x))$ 不成立

$\forall c \in \mathbb{R}^+$ ($f(x) \leq c g(x)$ 可以推导出 x 的一个上界，即总有更大的 x 使 $f(x) > c g(x)$)

或 $\exists k \in \mathbb{R}^+ \forall c \in \mathbb{R}^+$ ($c g(x) - f(x)$ 的值并不总是非负的，即 $\{x > k \mid c g(x) - f(x) < 0\}$ 非空)

注意：对于 $f(x) = \sum_{i=0}^n a_i x^i = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ ，有 $f(x)$ is $O(x^n)$

注意：对于 $f(n) = n!$ ，有 $f(n)$ is $O(n^n)$ ，并由此推出 $\log n!$ is $O(n \log n)$

$n! = \prod_{i=1}^n i$ ，当 $n \geq 1$ 时， $\prod_{i=1}^n i \leq \prod_{i=1}^n n$ ，即 $n! \leq n^n$ ，同时 $\log n! \leq \log n^n = n \log n$

常用函数序列 有 $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

注意：对于 $1 < c < d$ ，有 n^c is $O(n^d)$ ，但 $\lceil n^d \rceil$ is $O(n^c)$

对于 $b > 1$ ，有 $\log_b n$ is $O(n)$ 。因为 $b=2$ 时 $\log n$ is $O(n)$ ， $b \neq 2$ 时 $\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$

进而有 $\forall c, d \in \mathbb{R}^+ (\lceil \log_b n \rceil^c \text{ is } O(n^d) \wedge \lceil n^d \rceil \text{ is } O(\log_b n^c))$

另外对于 $b > 1$ ，有 n is $O(b^n)$ ，进而有 $\forall d \in \mathbb{R}^+ (n^d \text{ is } O(b^n) \wedge \lceil b^n \rceil \text{ is } O(n^d))$

对于 $f_1(x)$ is $O(g_1(x))$ 和 $f_2(x)$ is $O(g_2(x))$ ，有 $f_1(x) + f_2(x)$ is $O(\max(g_1(x), g_2(x)))$

如果 $f_1(x)$ is $O(n^2)$, $f_2(x)$ is $O(n \log n)$ ，则 $(f_1 + f_2)(x)$ is $O(n^2)$

另外如果 $f_1(x), f_2(x)$ are * $O(g(x))$ ，则 $(f_1 + f_2)(x)$ is $O(g(x))$

由 $|f_1(x)| + |f_2(x)| \leq C_1 |g_1(x)| + C_2 |g_2(x)| \leq C |\max(g_1(x), g_2(x))|$ 取 $C = C_1 + C_2$

另有 $(f_1 f_2)(x)$ is $O(g_1(x) g_2(x))$ ，即 $f_1(x) f_2(x)$ is $O(g_1(x) g_2(x))$

如 $f_1(x)$ is $O(n^2)$, $f_2(x)$ is $O(n \log n)$ ，则 $(f_1 f_2)(x)$ is $O(n^3 \log n)$

由 $|f_1(x)| |f_2(x)| \leq C_1 |g_1(x)| C_2 |g_2(x)| \leq C_1 C_2 |g_1(x) g_2(x)|$ 取 $C = C_1 C_2$

Discrete

Mathematics - P 33

大Ω记号 (big Omega notation), 相比于大O记号, 提供了 $f(x)$ 在 x 增大时的一个下限。
 $f(x) \text{ is } \Omega(g(x))$ 表示对于从 \mathbb{R} 到 \mathbb{R} 的函数 f 和 g , $\exists c, k \in \mathbb{R}^+$ $\forall x (x > k \rightarrow |f(x)| \geq c|g(x)|)$
 即存在正常数 c, k , 使得只要 $x > k$, 就有 $|f(x)| \geq c|g(x)|$
 直觉上说, 即当 x 无限增长时, $f(x)$ 的增长快于 $g(x)$ 的某个固定倍数

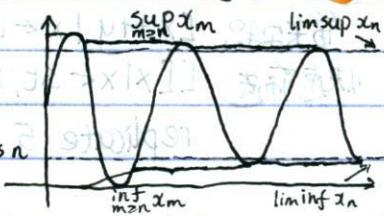
特别注意: 对于从 \mathbb{R} 到 \mathbb{R} 的函数 f 和 g , $f(x) \text{ is } \Omega(g(x)) \leftrightarrow g(x) \text{ is } O(f(x))$
 由于存在 (c, k) 使得 $\forall x (x > k \rightarrow |f(x)| \geq c|g(x)|)$
 则取 $c' = 1/c$, 有 (c', k) 使得 $\forall x (x > k \rightarrow |g(x)| \leq c'|f(x)|)$

大Θ记号 (big Theta notation), 相当于大O记号与大Ω记号的结合, 提供函数大小的一个上界和一个下界。
 $f(x) \text{ is } \Theta(g(x))$ 表示对于从 \mathbb{R} 到 \mathbb{R} 的函数 f 和 g , $\exists c_1, c_2, k \in \mathbb{R}^+$ $\forall x (x > k \rightarrow c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|)$
 即存在正常数 c_1, c_2, k , 使得只要 $x > k$ 时, 就有 $c_1|g(x)| \leq |f(x)| \leq c_2|g(x)|$
 直觉上说, 即当 x 无限增长时, $f(x)$ 的增长界于 $g(x)$ 的某两个固定倍数之间
 于是有 $f(x) \text{ is } \Theta(g(x)) \leftrightarrow f(x) \text{ is } O(g(x)) \wedge f(x) \text{ is } \Omega(g(x)) \leftrightarrow f(x) \text{ is } O(g(x)) \wedge g(x) \text{ is } O(f(x))$
 由于引出两类证明 $f(x) \text{ is } \Theta(g(x))$ 的方法, 即 $f(x) \text{ is } O(g(x))$ 或 $f(x) \text{ is } \Omega(g(x))$
 当 $f(x) \text{ is } \Theta(g(x))$, 则称 $f(x)$ 是 $g(x)$ 同阶的, 或称 $f(x)$ 与 $g(x)$ 是同阶的
 如有 $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, $a_n, \dots, a_0 \in \mathbb{R}$, $a_n \neq 0$, 则 $f(x)$ 是 x^n 阶的

Big O Notation						
Notation	$f(x) = o(g(x))$	$O(g(x))$	$\Theta(g(x))$	$\sim g(x)$	$\Omega(g(x))$	$\omega(g(x))$
Name	Small O	Big O	Big Theta	on the order	Big Omega	Big Omega
Description	dominated by g	bounded above by g	bounded both above and below by g	is equal to g	not dominated by g	dominated by g
Formal Definition	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$	$\forall k > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall \epsilon > 0 \exists n_0 \forall n > n_0 \exists k_1 > 0 \exists k_2 > 0 \forall n > n_0$
Limit Definition	$\lim_{n \rightarrow \infty} f(n) < k \cdot g(n)$	$\limsup_{n \rightarrow \infty} f(n) \leq k \cdot g(n)$	$f(n) \sim k \cdot g(n)$	$\left \frac{f(n)}{g(n)} - 1 \right < \epsilon$	$ f(n) \geq k \cdot g(n)$	$ f(n) \geq k \cdot g(n)$

lim sup "S" (limit superior) If $S = \limsup_{n \rightarrow \infty} x_n$, then interval $[I, S]$ need not contain any of numbers x_n , but every enlargement $[I - \epsilon, S + \epsilon]$ will contain x_n for all but finitely many indices n .

$$\forall \epsilon > 0 \exists n_0 \in \mathbb{N} \forall n > n_0 \liminf_{n \rightarrow \infty} x_n - \epsilon < x_n < \limsup_{n \rightarrow \infty} x_n + \epsilon$$

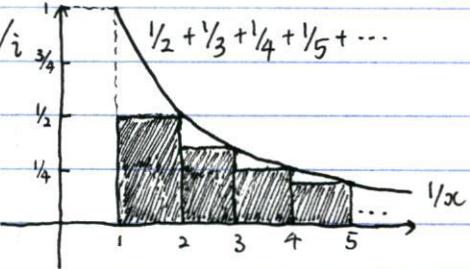


Discrete

Mathematics - P 34

调和数

令 第 n 项调和数 ($n \in \mathbb{Z}^+$), $H_n = \frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i}$
由 $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = 1 + \sum_{j=2}^n \frac{1}{j}$
 $(n > 1) < 1 + \int_1^n \frac{1}{x} dx = 1 + \ln x|_1^n = 1 + \ln n$
取 $k=2$ $< 2 \ln n = 2 \ln^2 \cdot \log n$
 $(c=2\ln^2)$, 于是有 H_n is $O(\log n)$



$\log n!$ is $\Theta(n \log n)$ 对于 $\log n!$ 和 $n \log n$, 有 $n! = n \times (n-1) \times \dots \times 2 \times 1$ ▶ 当 $n > 4$ 时, $n/2 > 2$, 有 $n! > n \times (n-1) \times \dots \times \Gamma(n/2)$

又 $n \times (n-1) \times \dots \times \Gamma(n/2) > \Gamma(n/2) \times \Gamma(n/2) \times \dots \times \Gamma(n/2) = \Gamma(n/2)^{(n-\Gamma(n/2)+1)}$

于是有 $\log n! > (n-\Gamma(n/2)+1) \cdot \log \Gamma(n/2)$

又当 $n > 4$ 时, $\sqrt{n} < n/2$, 即 $\frac{1}{2} \log n < \log \frac{n}{2} \leq \log \Gamma(n/2)$

且 $2(n-\Gamma(n/2)+1) \geq n+1 > n$

即 $n \log n < 4(n-\Gamma(n/2)+1) \cdot \log \Gamma(n/2) < 4 \log n!$

取 $C=4$, $k=4$, 则有 $n \log n$ is $O(\log n!)$, 又 $\log n!$ is $O(n \log n)$

于是有 $\log n!$ is $\Theta(n \log n)$

时间复杂度 (time complexity), 指当输入值具有一定规模时, 计算机按此算法解决问题所花的时间

空间复杂度 (space complexity), 指当输入值具有一定规模时, 实现这一算法计算机需要多大内存

计算复杂度 (computational complexity), 实际上是两种复杂度的综合

时间复杂度可以用算法所需运算次数表示

运算通常可以是 整数比较, 整数加法, 整数乘法, 整数除法 或任何其他基本运算

注意: 一般不用计算机实际使用的时间表示.

最坏情形复杂度 (worst-case time complexity), 指算法求解给定规模的问题需要的最大时间量

通常给出一个算法 需要多少次运算就能保证给出问题的解答

平均情形复杂度 (average-case time complexity), 指算法求解给定规模的问题需要的平均时间量

通常 平均情形复杂度分析 比最坏情形分析复杂.

平均情形可视为对时间量在不同规模下时间量求期望, $E[T(n)] / |S|$

线性搜索的最坏与平均复杂度均为 $\Theta(n)$, 而二分搜索均为 $O(\log n)$

冒泡排序的最坏情形复杂度为 $\Theta(n^2)$, 插入排序的最坏情形也是 $\Theta(n^2)$

最有效排序算法的 复杂度为 $O(n \log n)$