Chapter 1 The Role of Algorithms in Computing

Basic Concepts:

(1) 算法：任何良定义的计算过程，该过程取某个值或值的集合作为输入并产生某个值或值的集合作为输出

↓ 与元素的选取无关

无歧义 + 结果唯一 + 不自相矛盾.

Input → Algorithm → Output

(2) 计算问题：从总体上规定了所期望的输入与输出间的关系.

(3) 问题实例：由计算该问题解所必需的（满足问题陈述中强加的各种约束的）输入组成

的特殊设备的种类（主存、磁盘或者磁带）。

若对每个输入实例算法都以正确的输出停机，则称该算法是**正确的**，并称正确的算法**解决了**给定的计算问题。不正确的算法对某些输入实例可能根本不停机，也可能以不正确的回答停机，与人们期望的相反，不正确的算法只要其错误率可控有时可能是有用的。在第 31 章，当我们研究求大素数算法时，将看到一个具有可控错误率的算法例子。但是通常我们只关心正确的算法。

① ②

(4) 数据结构：一种存储和组织数据的方式，旨在便于访问和修改

(5) 难题 - NP-complete problems（NPC问题/NP完全问题）

Chapter 2 算法基础

(1) Framework

① Problem formalization (问题格式化)

② Algorithm (设计算法→伪代码)

③ Analysis (分析 → 是否正确 / → 效率)

An example

(1) Problem formalization

Sorting Problem:

Input: A sequence of n numbers $\langle a_1, a_2, \cdots, a_n \rangle$

Output: A permutation (排列) $\langle a_1', a_2', \cdots a_n' \rangle$ of the input sequence such that (满足)

$a_1' \le a_2' \le \cdots a_n'$

(2) Algorithm (pseudo-code (伪代码), design)

我们首先介绍**插入排序**，对于少量元素的排序，它是一个有效的算法。插入排序的工作方式像许多人排序一手扑克牌。开始时，我们的左手为空并且桌子上的牌面向下。然后，①我们每次从桌子上拿走一张牌并将它插入左手中正确的位置。②为了找到一张牌的正确位置，我们从右到左将它与已在手中的每张牌进行比较，如图 2-1 所示。③拿在左手上的牌总是排序好的，原来这些牌是桌子上牌堆中顶部的牌。

对于插入排序，我们将其伪代码过程命名为 INSERTION-SORT，其中的参数是一个数组 $A[1..n]$，包含长度为 n 的要排序的一个序列。（在代码中，A 中元素的数目 n 用 $A.length$ 来表示。）该算法原址排序输入的数：算法在数组 A 中重排这些数，在任何时候，最多只有其中的常数个数字存储在数组外面。在过程 INSERTION-SORT 结束时，输入数组 A 包含排序好的输出序列。



图 2-1 使用插入排序来排序手中扑克牌

③插入正确位置

```
INSERTION-SORT(A)
    for j=2 to A.length
        key = A[j]
        i = j-1                          ↗ 将A[j]插入A[1,…j-1]
        while i>0 and A[i]>key
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
```

(3) Analysis

① Correctness √

② Efficiency → Turing machines / RAM (random-access machine)

eg1:

Task: 判断整数的奇偶性

$\Gamma = \{0, 1, b\}$

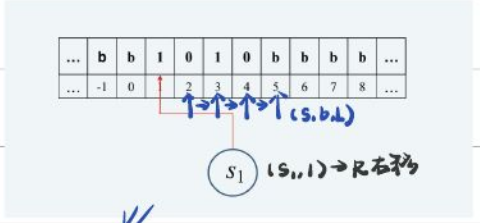
$Q = \{s_1, s_2, s_3, s_y, s_n\}$

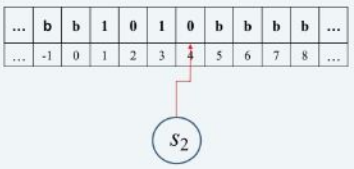
	0	1	b
s_1	$(s_1, 0, R)$	$(s_1, 1, R)$	(s_n, b, L)
s_2	$(s_y, 0, N)$	$(s_2, 1, N)$	(s_y, b, N)
s_3	$(s_y, 0, N)$	$(s_n, 1, N)$	(s_3, b, N)

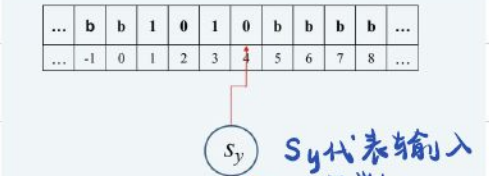
优化

	0	1	b
s_1	$(s_1, 0, R)$	$(s_1, 1, R)$	(s_n, b, L)
s_2	$(s_y, 0, N)$	$(s_n, 1, N)$	(s_n, b, N)

定位二进制最后一位







eg2: Task: 实现图灵机+1

	0	1	b
s_1	$(s_1, 0, R)$	$(s_1, 1, R)$	(s_n, b, L)

S_2 | $(S_{n,1},N)$ | $(S_{2,0},b)$ | $(S_{n,1},N)$ 特殊情况

处理进位

b 1 1 1 b
↑ ↑ ↑ ↑
1 2 0 0

12) Analysis

RAM (Random-Access Machine)

① 单处理器的计算模型，指令一条接一条地执行，无并发操作

② primitive operations (Instructions):

arithmetic (+、-、×、÷、%、⌈⌉、⌊⌋)
data movement (load, store, copy)
control

⇒ 每条这样的指令所需时间都为常量.

③ 量化:

Running time: Number of primitive operations

Memory: Number of memory cells utilized.

An example:

Sorting Problem: 插入排序

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

INSERT-SORT (A)		cost	times
1	for j ← 2 to ~~length[A]~~ A.length	c_1	n 多判断一次中止.
2	do key ← A[j]	c_2	n-1
3	▷ Insert A[j] into the sorted sequence A[1..j - 1]	0	n-1
4	i ← j − 1	c_4	n-1
5	while i > 0 and A[i] > key	c_5	$\sum_{j=2}^{n} t_j$
6	do A[i + 1] ← A[i] ▷ move item back	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	i ← i − 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	A[i + 1] ← key ▷ find the insertion position	c_8	n-1

t_j : **the number of times the while loop test in line 5 is executed for the j value.**

***Note*: When a *for* or *while* loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.**

当一个 for 或 while 循环按通常的方式退出时，由于循环头中的测试，执行该测试的次数比执行循环体的次数 + 1.

不定 ⇒ 依赖给定输入

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

最佳情况：输入数组已排好序 $(t_j = 1)$ ⇒ n 的线性函数

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
$$= an + b$$

最坏情况：输入数组已反向排序 $(t_j = j)$ (j: 1 → j-1) ⇒ n 的二次函数

$$\sum_{j=2}^{n} j = 2 + 3 + 4 + \cdots + n = 1 + 2 + 3 + 4 + \cdots + n - 1$$
$$= \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2}$$

$\therefore T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6 \cdot \frac{n(n-1)}{2} + c_7 \cdot \frac{n(n-1)}{2} + c_8(n-1)$

$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8)$

$\Rightarrow an^2 + bn + c$

(3) 增长量级

- Ignore the actual cost of each statement by abstract cost c_i

$$\left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) .$$

- Ignore the abstract cost c_i

$$an^2 + bn + c$$

- Ignore the lower-order terms

$$an^2$$

- Ignore the leading term's constant coefficient

$$n^2$$

$$O(n^2)$$

ch3 函数的增长.

渐近效率：当输入规模无限增加时，在极限中，算法的运行时间如何随着输入规模的变大而增加.

有效算法：多项式时间算法 $(n, n\log_2 n, n^2, n^3)$ → 设计算法目标 ① ② 对多项式时间所低.

渐近记号 → 抓定义

1) Big Θ notation

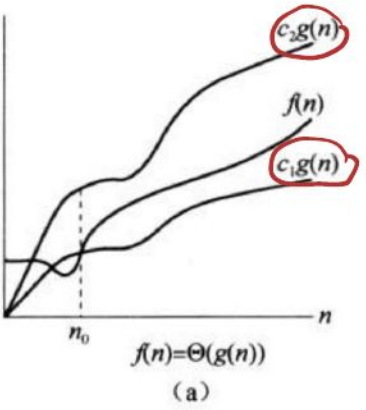
$f(n)=\Theta(g(n))$ / $f(n)\in\Theta(g(n))$

$f(n)$和$g(n)$增长时间相同

定义：$\Theta(g(n))=\{f(n)$：存在正常量 $c_1、c_2$和n_0，使得对所有 $n\geq n_0$，有 $0\leq c_1g(n)\leq f(n)\leq c_2g(n)\}$

$g(n)$是$f(n)$的一个渐近紧确界

$$\Theta(g(n))=\{f(n)\mid \exists c_1,c_2>0, n_0>0, \forall n\geq n_0, 0\leq c_1g(n)\leq f(n)\leq c_2g(n)\}$$



(a)

例：

证明$f(n)$和$g(n)$同阶

1. 证明 $\frac{1}{2}n^2-3n=\Theta(n^2)$

本质：proof $\exists c_1,c_2>0, n_0>0, \forall n\geq n_0, 0\leq c_1g(n)\leq \frac{1}{2}n^2-3n\leq c_2g(n)$

其中 $g(n)=n^2$

$\Rightarrow 0\leq c_1n^2\leq \frac{1}{2}n^2-3n\leq c_2n^2$

$c_1\leq \frac{1}{2}-\frac{3}{n}\leq c_2 \quad (n\geq 1)$

① 右边

$c_2\geq \frac{1}{2}$

② 左边

$c_1\leq \frac{1}{2}-\frac{3}{n}$ → 恒成立.

令 $n_0=7$ $c_1\leq \frac{1}{2}-\frac{3}{7}$ 满足

$n\geq n_0$时 $c_1\leq$ 任何

2. $f(n)=32n^2+17n+1$

证 $f(n)$ is $\Theta(n^2)$

本质：$\exists c_1,c_2>0, n_0>0, \forall n\geq n_0, 0\leq c_1n^2\leq 32n^2+17n+1\leq c_2n^2$

proof: $c_1n^2\leq 32n^2+17n+1\leq c_2n^2$

① 左

$c_1n^2\leq 32n^2+17n+1$

$c_1\leq 32$ 即可

$n^2\geq n\geq 1$

② 右

$32n^2+17n+1\leq c_2n^2$

$17n+1\leq (c_2-32)n^2$ → 放缩成我们想要的形式

$17n+1\leq 17n^2+n^2\leq (c_2-32)n^2$

$18n^2\leq (c_2-32)n^2$

$c_2-32\geq 18$

$c_2\geq 50$

$\therefore c_1=32, c_2=50, n_0=1$ 即可

证明 $f(n)$ 和 $g(n)$ 不同阶

1. $6n^3 \neq \Theta(n^2)$ → 不等问题考虑反证法

假设 $\exists c_1, c_2 > 0$, $n_0 > 0$, 对 $\forall n \geq n_0$ 均有 $0 \leq c_1 n^2 \leq 6n^3 \leq c_2 n^2$

关注 $6n^3 \leq c_2 n^2$

$6n \leq c_2$

由于 c_2 为常量, 对任意大的 n, 该不等式不可能成立.

2. $f(n) = 32n^2 + 17n + 1$

证 $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

① 反证法

假设 $\exists c_1, c_2, n_0 > 0$ 对 $\forall n \geq n_0$, 均有

$0 \leq c_1 n \leq 32n^2 + 17n + 1 \leq c_2 n$

关注 $32n^2 + 17n + 1 \leq c_2 n$

$32n^2 + (17 - c_2)n + 1 \leq 0$

由于 $n \geq 1$ 的, 当 $n \to \infty$ 上述不等式必不成立

② 假设 $\exists c_3, c_4, n_1 > 0$ $\forall n \geq n_1$, 均有

$0 \leq c_3 n^3 \leq 32n^2 + 17n + 1 \leq c_4 n^3$

关注 $c_3 n^3 \leq 32n^2 + 17n + 1$

$c_3 \leq \frac{32}{n} + \frac{17}{n^2} + \frac{1}{n^3}$

由于 $h(n) = \frac{32}{n} + \frac{17}{n^2} + 1$ ↓

故当 $n \to +\infty$ $h(n) \to 0$

而 $c_3 > 0$ 上式不能恒成立.

若 Θ 中的阶比原 $f(n)$ 小考虑右边存在问题

反之, 考虑左边不恒成立

思考题

考虑任意二次函数 $f(n) = an^2 + bn + c$ 其中 a, b 和 c 均为常量且 $a > 0$

证 $f(n) = \Theta(n^2)$

本质: proof $\exists c_1, c_2 > 0$, $n_0 > 0$ $\forall n \geq n_0$ 都有 $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

代入即证明 $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$

① 左侧 $an^2 + bn + c \geq c_1 n^2$

即需找到 $c_1 > 0$ 使 $(a - c_1)n^2 + bn + c \geq 0$

取 $c_1 = \frac{a}{4}$

$\frac{3a}{4} n^2 + bn + c \geq 0$

把最坏情况考虑进去

$\frac{3a}{4} n^2 + bn + c \geq \frac{3a}{4} n^2 - |b|n - |c|$ → $\frac{a}{2} n^2 - |b|n \geq 0$

$\frac{a}{2} n^2 - |b|n + \frac{a}{4} n^2 - |c|$ $\frac{a}{2} n \geq |b|$ $n \geq \frac{2|b|}{a}$

↳ $\frac{a}{4} n^2 - |c| \geq 0 \Rightarrow \frac{a}{4} n^2 \geq |c|$ $n^2 \geq \frac{|c|}{a}$

② 右侧

$an^2+bn+c \le c_2n^2$

$(c_2-a)n^2-bn-c \ge 0$

取 $c_2=\frac{7}{4}a$

$\frac{3}{4}an^2-bn-c \ge \frac{3a}{4}n^2-|b|n-|c| \ge 0$

同理可得，

因此 $\exists c_1=\frac{a}{4}$，$c_2=\frac{7a}{4}$，当 $n \ge 2\cdot max(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}})$ 时

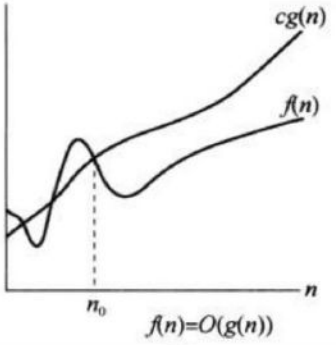
有 $\frac{3a}{4}n^2-bn-c \ge \frac{3a}{4}n^2-|b|n-|c| \ge \frac{3a}{4}n^2-\frac{a}{2}n^2-\frac{a}{4}n^2=0$

12) Big O notation → 渐近上界

定义：

$O(g(n))=\{f(n)$：存在正常量 c 和 n_0，使得对所有 $n \ge n_0$，有 $0 \le f(n) \le cg(n)\}$

$f(n)=O(g(n))$：函数 $f(n)$ 是集合 $O(g(n))$ 的成员



例：

1. $f(n)=32n^2+17n+1$

证 $f(n)$ is $O(n^2)$

本质证：$\exists$ 正常量 c 和 n_0 使得对 $n \ge n_0$ 有 $0 \le 32n^2+17n+1 \le cn^2$

① 左边恒成立

② 右边 $n \ge 1$ 时 $n^2 \ge n \ge 1$

$cn^2 \ge 32n^2+17n+1 \ge 32n^2+17n^2+n^2=50n^2$

$c \ge 50$

∴ 取 $n_0=1$，$c=50$

性质：

① 自反性 f is $O(f)$

② 乘积性 若 f_1 是 $O(g_1)$ 且 f_2 是 $O(g_2)$，则 f_1f_2 是 $O(g_1g_2)$

证明：

$\exists c_1>0, n_1>0$，$\forall n \ge n_1$ $0 \le f_1(n) \le c_1g_1(n)$

$\exists c_2>0, n_2>0$，$\forall n \ge n_2$ $0 \le f_2(n) \le c_2g_2(n)$

对 $\forall n \ge max\{n_1, n_2\}$，$0 \le f_1(n)\cdot f_2(n) \le \underbrace{c_1c_2}_{c} g_1(n)\, g_2(n)$

③ 求和性 若 f_1 是 $O(g_1)$ 且 f_2 是 $O(g_2)$ 则 f_1+f_2 是 $O(max\{g_1, g_2\})$

证：$\exists c_1>0, n_1>0$，$\forall n \ge n_1$ $0 \le f_1(n) \le c_1g_1(n)$

∃$c_2>0$, $n_2>0$ ∀ $n\geq n_2$ $0\leq f_2(n)\leq c_2 g_2(n)$

对∀ $n\geq \max\{n_1,n_2\}$ $0\leq f_1(n)+f_2(n)\leq c_1 g_1(n)+c_2 g_2(n)$

$\leq (c_1+c_2)\max\{g_1(n),g_2(n)\}$

c

④传递性 若f是$O(g)$且g是$O(h)$，则f是$O(h)$

Big O notation with multiple variables

定义：若存在常数$c>0$，$m_0\geq 0$和$n_0\geq 0$，使得对所有满足$n\geq n_0$且$m\geq m_0$的取值，均有$f(m,n)\leq c\cdot g(m,n)$称$f(m,n)$为$O(g(m,n))$

示例

设 $f(m,n)=32mn^2+17mn+32n^3$。

- $f(m,n)$ 既是 $O(mn^2+n^3)$，也是 $O(mn^3)$。
- $f(m,n)$ 既不是 $O(n^3)$，也不是 $O(mn^2)$。

典型应用

在包含 n 个节点、m 条边的有向图中，广度优先搜索（BFS）求解从节点 s 到节点 t 的最短路径所需时间复杂度为 $O(m+n)$。

(2) Big Ω notation → 渐近下界

定义：$\Omega(g(n))=\{f(n)$：存在正常量c和n_0，使得对所有$n\geq n_0$，有$0\leq cg(n)\leq f(n)\}$

例：

$f(n)=32n^2-17n+1$ 证 $f(n)=\Omega(n)$

本质：∃ $c>0$, $n_0>0$, ∀ $n\geq n_0$ 有 $0\leq cn\leq 32n^2-17n+1$

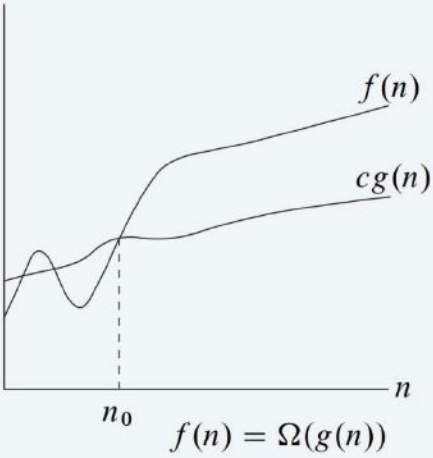
$n\geq 1$时 $n^2\geq n\geq 1$

令 $c=14$

$32n^2-17n+1\geq 14n$

$32n^2-31n+1\geq 32n-31n+1=n+1\geq 0$

∴取$c=14$，$n_0=1$



$f(n)=\Omega(g(n))$

- 对所有的$n\geq n_0$，函数f(n)在一个常数因子内等价于g(n)。
- 称g(n)是f(n)的一个渐进紧确界(*asymptotically tight bound*)
  - 从算法时间复杂度的角度看，就是算法在最好和最坏情况下的计算时间就一个常数因子范围内而言是相同的。可看作：

    既有 f(n) = Ω(g(n))，又有f(n) = O(g(n))

    即，g 既是f 的下界，又是f 的上界。

定理

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n)=\Theta(g(n))$ if and only if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$. ■

(4) Small o notation → 非渐近紧确的上界.

定义：$o(g(n)) = \{f(n)$：对任意正常量 $c>0$，存在正常量 $n_0>0$，使得对所有的 $n \geq n_0$，有 $0 \leq |f(n)| < c|g(n)|$

直观上，在 o 记号中，当 n 趋于无穷时，函数 $f(n)$ 相对于 $g(n)$ 变得微不足道

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

例：$f(n) = 2n$ $g(n) = n^2$

证 $f(n) = o(n^2)$

即证

$\forall c>0, \exists n_0>0$ $\forall n \geq n_0$ $f(n) < cg(n)$

$2n < cn^2$

$\frac{2}{c} < n$

令 $n_0 = \lceil \frac{2}{c} \rceil$

$\forall n \geq n_0 > \frac{2}{c} > \frac{2}{c}$

故得证.

(5) w notation → 非渐近紧确的下界

定义：$f(n) \in w(g(n))$ 当且仅当 $g(n) \in o(f(n))$

$w(g(n)) = \{f(n)$：对任意正常量 $c>0$，存在常量 $n_0>0$，对 $\forall n \geq n_0$ 有 $0 \leq cg(n) < f(n)$.

在 w 表示中，当 n 趋于无穷时，$f(n)$ 相对于 $g(n)$ 来说变得任意大了 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

(6) 比较各类函数.

实数的许多关系性质也适用于渐近比较。下面假定 $f(n)$ 和 $g(n)$ 渐近为正。

传递性：

$f(n) = \Theta(g(n))$ 且 $g(n) = \Theta(h(n))$ 蕴涵 $f(n) = \Theta(h(n))$
$f(n) = O(g(n))$ 且 $g(n) = O(h(n))$ 蕴涵 $f(n) = O(h(n))$
$f(n) = \Omega(g(n))$ 且 $g(n) = \Omega(h(n))$ 蕴涵 $f(n) = \Omega(h(n))$
$f(n) = o(g(n))$ 且 $g(n) = o(h(n))$ 蕴涵 $f(n) = o(h(n))$
$f(n) = \omega(g(n))$ 且 $g(n) = \omega(h(n))$ 蕴涵 $f(n) = \omega(h(n))$

自反性：

$f(n) = \Theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

对称性：

$f(n) = \Theta(g(n))$ 当且仅当 $g(n) = \Theta(f(n))$

转置对称性：

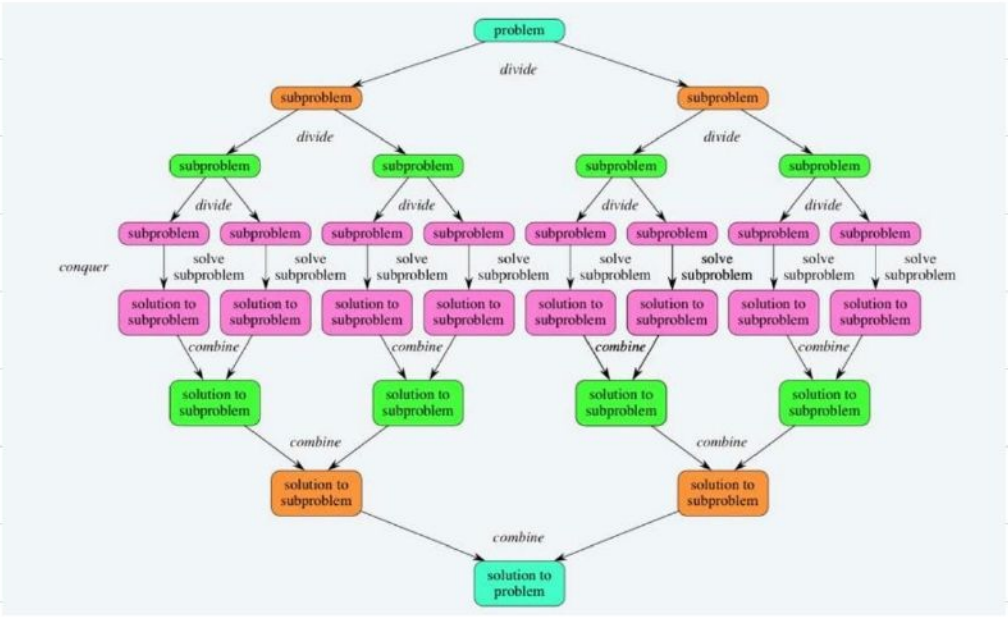
$f(n) = O(g(n))$ 当且仅当 $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ 当且仅当 $g(n) = \omega(f(n))$

ch4

分而治之思想 Divide-and-conquer

思想：① 分解 将原问题分解为n个规模较小但类似于原问题的子问题 ② 解决 递归地求解这些子问题，

③ 合并 然后再合并这些子问题的解来建立原问题的解.



2. 应用-归并排序

(1) 操作

分解：分解待排序的n个元素的序列成各具n/2个元素的两个子序列

解决：使用归并排序递归排序两个子序列

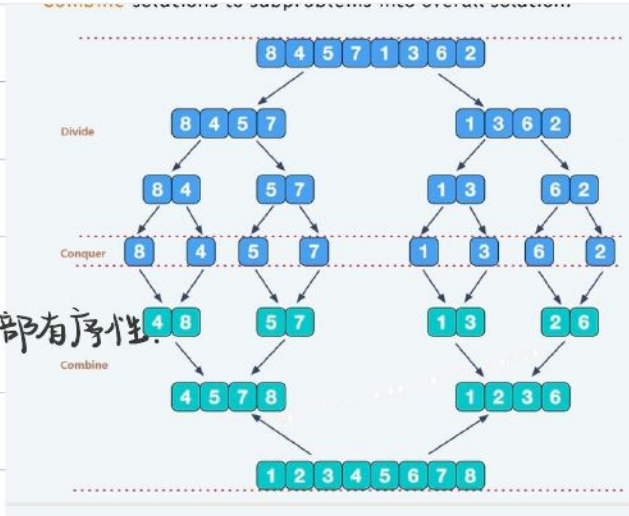
合并：合并两个已排序的子序列以产生已排序的答案

(2) 问题定义

Sorting Problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.



局部有序性

```
MERGE-SORT (L)
IF (list L has one element)
    Return L
Divide the list into two halves A and B
A ← MERGE-SORT (A)
B ← MERGE-SORT (B)
L ← MERGE (A, B)
Return L
```

分解

解决

合并 → 通过调用辅助过程MERGE来合并

有关 MERGE(A, p, q, r) → 任务：合并 A[p..q] 和 A[q+1..r] 形成单一已排好序的子数组代替当前 A[p..r]

（A：数组；p, q, r：数组下标 $p \le q \le r$）

假设 A[p..q] A[q+1..r] 都已排好序

MERGE(A, p, q, r)　A = [a₁, … [ap … aq], [aq+1 … ar] … an]

（L：ap … aq，长度 q−p+1；R：aq+1 … ar，长度 r−q；合并）

```
1  n1 = q − p + 1
2  n2 = r − q
3  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i − 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

保证一边排好，剩余元素仍可比较。哨兵牌

合并过程

(3) 分析算法

MERGE-SORT(L)　T(n)

IF (list L has one element)

　RETURN L.

Divide the list into two halves A and B.　← Θ(1)

A ← MERGE-SORT(A).　← $T(\frac{n}{2})$

B ← MERGE-SORT(B).　← $T(\frac{n}{2})$

L ← MERGE(A, B).　← Θ(n)

RETURN L.

一般情况：

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{其他} \end{cases}$$

分解问题成子问题

解决：假设把原问题分解成 a 个子问题，每个子问题的规模是原问题的 $\frac{1}{b}$

合并子问题的解成原问题的解

归并排序

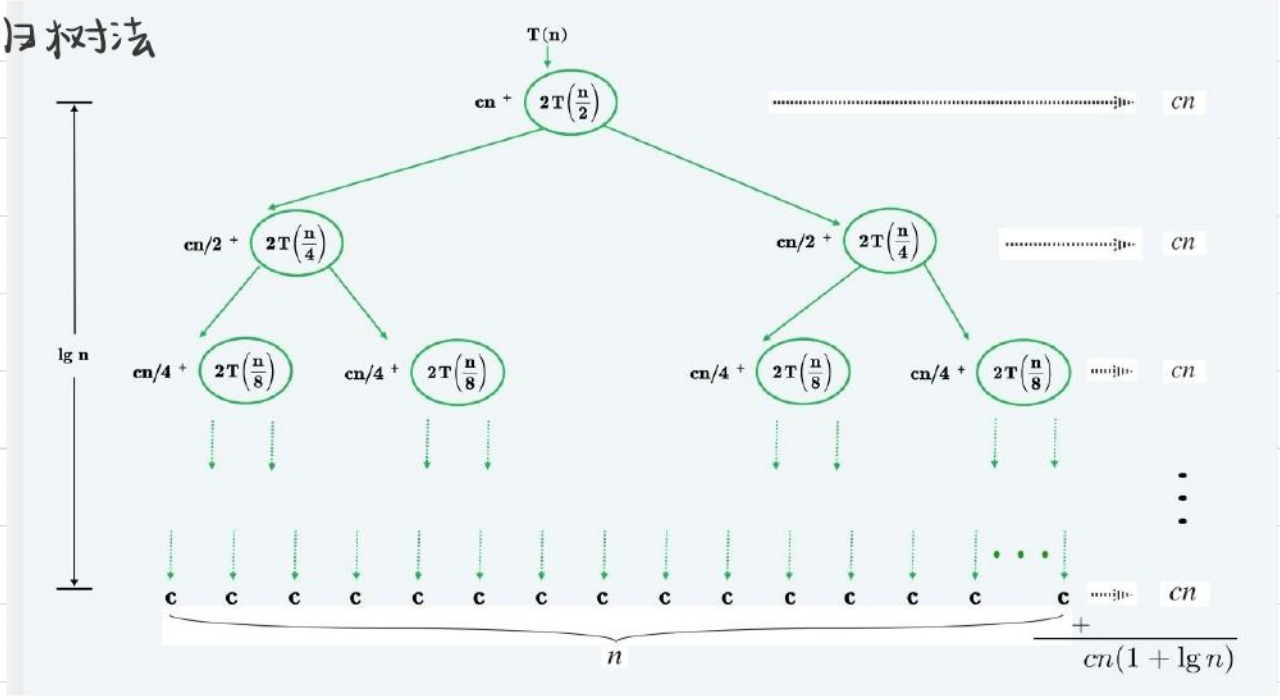
$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

（假定问题规模是 2 的幂）。

重写 $T(n)=\begin{cases} c & \text{if } n=1 \\ 2T(\frac{n}{2})+cn & \text{if } n>1 \end{cases}$ （c：简化运算）

或 $T(n)\le\begin{cases} c & \text{if } n=1 \\ T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+cn & \text{if } n>1 \end{cases}$

① 递归树法



② $T(n)=O(n\log n)$

数学归纳法证明：

$T(n)\le\begin{cases} c & \text{if } n=1 \\ T(\lfloor\frac{n}{2}\rfloor)+T(\lceil\frac{n}{2}\rceil)+cn & \text{if } n>1 \end{cases}$ $\xrightarrow{\text{For convenience 且假设 } n=2^k}$ $T(n)\le\begin{cases} c & \text{if } n=2 \\ 2T(\frac{n}{2})+cn & \text{if } n>2 \end{cases}$

猜测 → $T(n)=O(n\log n)$

1° $n=2^k$ 条件下

(1)

Base case:

当 $n=2$ 时 $T(2)\le c\le 2c\le cn\log_2 n$

(2) 假设 $T(n)\le cn\log_2 n$

目标 $T(2n)\le 2cn\log_2 2n$ 成立.

$$T(2n)\le 2T(n)+2cn$$
$$\le 2cn\log_2 n+2cn \quad \text{（假设）}$$
$$=2cn(\log_2(2n)-1)+2cn$$
$$=2cn\log_2(2n)$$

数学归纳法

第一 K→成立
证 K+1→成立

第二 假设 $n<K$ 成立
证 $n=K$ 成立

等价

2° n 任意正整数

(1)

Base case: $n=2$ (同上)

(2)

Define $n_1=\lfloor\frac{n}{2}\rfloor$ $n_2=\lceil\frac{n}{2}\rceil$ $n=n_1+n_2$

假设 true for $T(k)\le ck\lceil\log_2 k\rceil$ $k=1,2,\cdots n-1$

目标 $T(n)\le cn\lceil\log_2 n\rceil$

$$
\begin{aligned}
T(n) &\le T(n_1)+T(n_2)+cn \\
&\le cn_1\lceil \log_2 n_1\rceil + cn_2\lceil \log_2 n_2\rceil + cn \\
&\le cn_1\lceil \log_2 n_2\rceil + cn_2\lceil \log_2 n_2\rceil + cn \\
&= cn\lceil \log_2 n_2\rceil + cn \\
&\le cn(\lceil \log_2 n\rceil - 1) + cn \\
&= cn\lceil \log_2 n\rceil
\end{aligned}
$$

$n_2 = \lceil \frac{n}{2} \rceil \le \lceil 2^{\lceil \log_2 n\rceil}/2 \rceil = 2^{\lceil \log_2 n\rceil}/2$

$\log_2 n_2 \le \log_2 \frac{2^{\lceil \log_2 n\rceil}}{2} = \log_2 2^{\lceil \log_2 n\rceil} - \log_2 2 = \lceil \log_2 n\rceil - 1$

3. 应用一 最大子数组问题 maximum-subarray

Definition:

Input: An array $A[1 \ldots n]$ of (positive/negative) numbers.

Output:
(1) Indices i and j such that the subarray $A[i \ldots j]$ has the greatest sum of any nonempty contiguous subarray of A, and
(2) the sum of the values in $A[i \ldots j]$.

① 暴力求解办法

- Consider all pairs of i and j $(i \le j)$.



- BRUTE-FORCE-SUBARRAY(A)

```
1   max = −∞
2   max-i = 0      记录最优子数组左边界下标
3   max-j = 0      记录最优子数组右边界下标
4   for i = 1 to A.length
5       for j = i to A.length
6           sum = sum of A[i] through A[j]   计算区间[i,j]累加和
7           if sum > max
8               max = sum
9               max-i = i      更新最优左边界
10              max-j = j      更新最优右边界.
11  return max, max-i, max-j
```

- Time Complexity: $\Theta\left(\binom{n}{2}\right) = \Theta(n^2)$, where $n = A.length$

② 使用分治策略自己求解算法

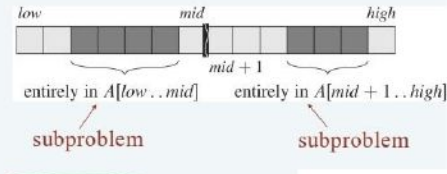
使用分治技术意味着我们要将子数组划分为两个规模尽量相等的子数组.

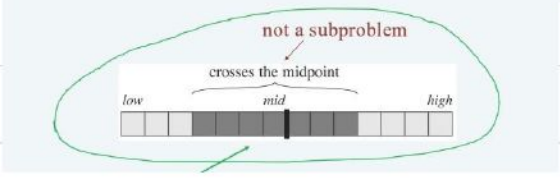
A[low..high]的任何连续子数组A[i..j]所处位置情况:

① 完全位于子数组 A[low..mid]中 $low \le i \le j \le high$

② 完全位于子数组 A[mid+1..high] $mid < i \le j \le high$

③ 跨越了中点 $low \le i \le mid < j \le high$



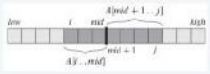


因此，$A[low..high]$的一个最大子数组所处的位置必然是这三种情况之一。实际上，$A[low..high]$的一个最大子数组必然是完全位于$A[low..mid]$中、完全位于$A[mid+1..high]$中或者跨越中点的所有子数组中和最大者。我们可以递归地求解$A[low..mid]$和$A[mid+1..high]$的最大子数组，因为这两个子问题仍是最大子数组问题，只是规模更小。因此，剩下的全部工作就是寻找跨越中点的最大子数组，然后在三种情况中选取和最大者。

特殊情况：跨越这个中点 $low \le i \le mid < j \le high$

（加入的特殊限制）

任何跨越中点的子数组都由两个子数组 $A[i..mid]$ 和 $A[mid+1..j]$ 组成，因此我们只需找出形如 $A[i..mid]$ 和 $A[mid+1..j]$ 的最大子数组，然后将其合并即可



```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞                          ┐ Find a max prefix
 2  sum = 0                                │ 往左搜索
 3  for i = mid downto low                 │
 4      sum = sum + A[i]                   │
 5      if sum > left-sum                  │   迭代 mid-low+1 次
 6          left-sum = sum                 │
 7          max-left = i                   ┘   Θ(n)
 8  right-sum = −∞                         ┐ Find a max suffix
 9  sum = 0                                │ 往右搜索
10  for j = mid + 1 to high                │
11      sum = sum + A[j]                   │   迭代 high-mid 次
12      if sum > right-sum                 │
13          right-sum = sum                │
14          max-right = j                  ┘
15  return (max-left, max-right, left-sum + right-sum)
```

总循环迭代：$(mid-low+1)+high-mid = high-low+1 = n$ 次

求解最大子数组问题的分治算法的伪代码：

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low                                              ┐
 2      return (low, high, A[low])                              │ Θ(1)
 3  else mid = ⌊(low + high)/2⌋                                 ┘
 4      (left-low, left-high, left-sum) =                       ┐
        ① FIND-MAXIMUM-SUBARRAY(A, low, mid)                    │
 5      (right-low, right-high, right-sum) =                    │ 2T(n/2)
        ② FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)               │
 6      (cross-low, cross-high, cross-sum) =                    ┘
        ③ FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)         Θ(n)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum        ┐
 8          return (left-low, left-high, left-sum)              │
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum   │ Θ(1)
10          return (right-low, right-high, right-sum)           │
11      else return (cross-low, cross-high, cross-sum)          ┘
```

比较取max

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

$T(n) = \Theta(n \log n)$

4.1-5 使用如下思想为最大子数组问题设计一个非递归的、线性时间的算法。从数组的左边界开始，由左至右处理，记录到目前为止已经处理过的最大子数组。若已知$A[1..j]$的最大子数组，基于如下性质将解扩展为$A[1..j+1]$的最大子数组：$A[1..j+1]$的最大子数组要么是$A[1..j]$的最大子数组，要么是某个子数组$A[i..j+1](1 \le i \le j+1)$。在已知$A[1..j]$的最大子数组的情况下，可以在线性时间内找出形如$A[i..j+1]$的最大子数组。

以第j个元素结尾的最大子数组和

$dp[j] = \max\{A[j], dp[j-1]+A[j]\}$

最终答案 $Ans = \max_{1 \le j \le n} dp[j]$

4. 求解递归式 Recurrence

① **代换法**：先猜测一个界，再使用数学归纳法证明猜测的正确性.

② **递归树方法**：将递归式转化为树，树中的结点代表不同递归层次付出的代价
使用边界求和的技巧来求解递归

③ **主方法**：给出递归形式 $T(n)=aT(n/b)+f(n)$ 的界，其中 $a\geq 1$，$b>1$，$f(n)$ 给定的函数.

(1) 代换法

① 猜测解的形式

② 用数学归纳法求出解中的常数并证明解是正确的 ⟶ ☆需要证出与归纳假设严格一致的形式

eg：确定 $T(n)=2T(\lfloor \frac{n}{2} \rfloor)+n$ 的上界

猜测其解为 $T(n)=O(n\lg n)$ ⇒ 证明∃常数 $c>0$ $T(n)\leq cn\lg n$

③ Base case ☆扩展边界条件使归纳假设对较小的 n 成立

由于当 $n=1$ 时 $\lg 1=0$ 导致 $T(1)\leq 0$ 一般不成立
因此我们不从 $n=1$ 开始证明，而是 $n\geq 2$ 开始证

当 $n=2$ 时
$T(2)\leq 2c$
$c\geq \frac{T(2)}{2}$

④ 归纳假设

假设对所有正数 $m<n$，上述结论已经成立

特别地令 $m=\lfloor \frac{n}{2} \rfloor$

$T(\lfloor \frac{n}{2} \rfloor)\leq c\lfloor \frac{n}{2} \rfloor \lg(\lfloor \frac{n}{2} \rfloor)$

$T(n)\leq 2c\lfloor \frac{n}{2} \rfloor \lg(\lfloor \frac{n}{2} \rfloor)+n \leq 2c(\frac{n}{2})\lg(\frac{n}{2})+n \leq cn\lg(\frac{n}{2})+n$ （$\log_2^2$）

$=cn\lg n-cn\lg 2+n$

$=cn\lg n-cn+n\leq cn\lg n \quad (c\geq 1)$

当 $c\geq \max(1, \frac{T(2)}{2})$，使得对所有 $n\geq 2$ 有 $T(n)\leq cn\lg n$

∴ $T(n)=O(n\lg n)$

2. $T(n)=T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+1$

猜测解为 $T(n)=O(n)$ ⇒ 证明对某个恰当选出的常数 c，$T(n)\leq cn$

若将猜测直接代入递归式：$T(n)\leq c\lfloor n/2 \rfloor+c\lceil n/2 \rceil+1=cn+1$ ✗ 并不意味着对任意 c 都有 $T(n)\leq cn$

直觉上，我们的猜测是接近正确的：只差一个常数 1，一个低阶项。但是，除非我们证明与归纳假设严格一致的形式，否则数学归纳法还是会失败。克服这个困难的方法是从先前的猜测中减去一个低阶项。新的猜测为 $T(n)\leq cn-d$，d 是大于等于 0 的一个常数。我们现在有

⇓

新的猜测 $T(n) \le cn-d$

$T(n) \le (c\lfloor \frac{n}{2} \rfloor - d) + c(\lceil \frac{n}{2} \rceil - d) + 1$

$= cn - 2d + 1 \le cn - d$. 当 $d \ge 1$

3. $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$ → 改变变量简化

令 $m = \lg n \Rightarrow n = 2^m$

代入得 $T(2^m) = 2T(\lfloor 2^{\frac{m}{2}} \rfloor) + m$

重命名令 $S(m) = T(2^m)$

代入： $S(m) = 2S(\frac{m}{2}) + m$ → 同上述例1

$S(m) = O(m \lg m)$

$T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg(\lg n))$

(2) 递归树方法

eg: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

$T(n) = 3T(\lfloor n/4 \rfloor) + cn^2 \quad c > 0$

为了方便起见，我们假定 n 是 4 的幂

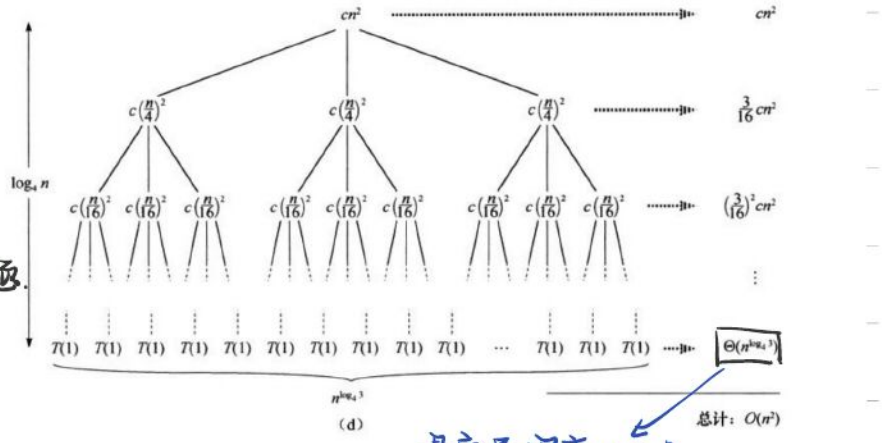
深度为 i 的结点，对应规模为 $n/4^i$ 的子问题.

令 $n/4^i = 1$

$n = 4^i$

$i = \log_4 n$ 子问题规模变为 1

∴ 递归树的层数: $\log_4 n + 1$



最底层深度 $\log_4 n$

节点数: $3^{\log_4 n} = n^{\log_4 3}$

总代价: $n^{\log_4 3} T(1) \Rightarrow \Theta(n^{\log_4 3})$

常数

$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + \Theta(n^{\log_4 3})$

$= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \Longrightarrow < \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$

$= \frac{cn^2 - \frac{3}{16}\cdot(\frac{3}{16})^{\log_4 n} cn^2}{1 - \frac{3}{16}} + \Theta(n^{\log_4 3}) \quad = \frac{1}{1-\frac{3}{16}} cn^2 + \Theta(n^{\log_4 3})$

$= \frac{1 - (\frac{3}{16})^{\log_4 n}}{1 - \frac{3}{16}} cn^2 + \Theta(n^{\log_4 3}) \quad = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$

$= O(n^2)$

代入法验证猜测的正确性.

$T(n) = O(n^2)$ → 存常数 $d > 0$ $T(n) \le dn^2$

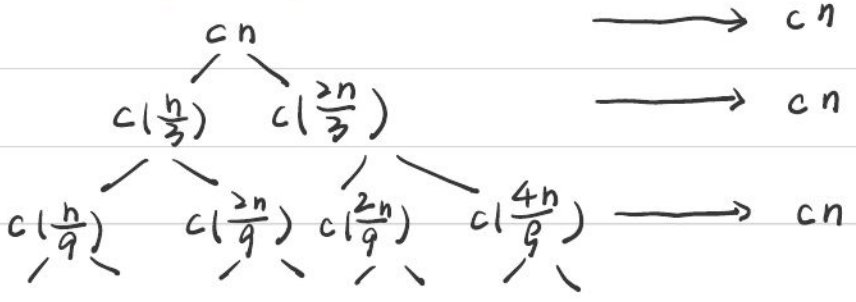
$T(n) \le 3d(\lfloor \frac{n}{4} \rfloor)^2 + cn^2 \le 3d(\frac{n}{4})^2 + cn^2 = (\frac{3}{16}d + c)n^2 \le dn^2$ 当 $d \ge \frac{16}{13}c$

$d \ge \frac{3}{16}d + c$

$\frac{13}{16}d \ge c$

$d \ge \frac{16}{13}c$

2. $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n)$



从根到叶的最长简单路径：$n \to \frac{2n}{3} \to (\frac{2}{3})^2 n \to \cdots \to 1$
①　②　③

当 $(\frac{2}{3})^k n = 1$

$n = (\frac{3}{2})^k$

$k = \log_{\frac{3}{2}} n$

树高：$\log_{\frac{3}{2}} n$

为 $\log_{3/2} n$。　　图 4-6 递归式 $T(n) = T(n/3) + T(2n/3) + cn$

直觉上，我们期望递归式的解最多是层数乘以每层的代价，即 $O(cn \log_{3/2} n) = O(n \lg n)$。但图 4-6 仅显示了递归树的顶部几层，并不是递归树中每个层次的代价都是 cn。考虑叶结点的代价。如果递归树是一棵高度为 $\log_{3/2} n$ 的完全二叉树，则叶结点的数量应为 $2^{\log_{3/2} n} = n^{\log_{3/2} 2}$。由于每个叶结点的代价为常数，因此所有叶结点的总代价为 $\Theta(n^{\log_{3/2} 2})$，由于 $\log_{3/2} n$ 是严格大于 1 的常数，因此叶结点代价总和为 $\Omega(n \lg n)$。但递归树并不是完全二叉树，因此叶结点数量小于 $n^{\log_{3/2} 2}$。而且，当从根结点逐步向下走时，越来越多的内结点是缺失的。因此，递归树中靠下的层次对总代价的贡献小于 cn。我们可以计算出所有代价的准确值，但记住我们只是希望得到一个猜测，用于代入法。我们还是忍受一些不精确，尝试证明猜测的上界 $O(n \lg n)$ 是正确的。

代入法验证 $O(n \lg n)$ 是递归式解的一个上界 $\Rightarrow T(n) \le dn \lg n$ 对常数 $d > 0$

$$T(n) \le d\frac{n}{3} \lg \frac{n}{3} + d\frac{2n}{3} \lg \frac{2n}{3} + cn$$

$$= d\frac{n}{3} \lg n - d\frac{n}{3} \lg 3 + d\frac{2n}{3} \lg n - d\frac{2n}{3} \lg \frac{3}{2} + cn$$

$$= dn \lg n - \left(d\frac{n}{3} \lg 3 + d\frac{2n}{3} \lg \frac{3}{2}\right) + cn$$

$$= dn \lg n - \left(d\frac{n}{3} \lg 3 + d\frac{2n}{3} \lg 3 - d\frac{2n}{3} \lg 2\right) + cn$$

$$= dn \lg n - dn\left(\lg 3 - \frac{2}{3}\right) + cn \le dn \lg n \quad \text{当 } d \ge \frac{c}{\lg 3 - \frac{2}{3}}$$

$$cn \le dn\left(\lg 3 - \frac{2}{3}\right)$$

$$d \ge \frac{c}{\lg 3 - \frac{2}{3}}$$

会用

(3) 主方法 Master Method.

主方法为如下形式的递归式提供了一种“菜谱”式的求解方法

$$T(n) = aT(n/b) + f(n) \qquad (4.20)$$

其中 $a \ge 1$ 和 $b > 1$ 是常数，$f(n)$ 是渐近正函数。为了使用主方法，需要牢记三种情况，但随后你就可以很容易地求解很多递归式，通常不需要纸和笔的帮助。

递归式(4.20)描述的是这样一种算法的运行时间：它将规模为 n 的问题分解为 a 个子问题，每个子问题规模为 n/b，其中 a 和 b 都是正常数。a 个子问题递归地进行求解，每个花费时间 $T(n/b)$。函数 $f(n)$ 包含了问题分解和子问题解合并的代价。例如，描述 Strassen 算法的递归式中，$a=7$，$b=2$，$f(n)=\Theta(n^2)$。

主定理

令 $a \ge 1$ 和 $b > 1$ 是常数，$f(n)$ 是一个函数，$T(n)$ 是定义在非负整数上的递归式

$T(n) = aT(\frac{n}{b}) + f(n)$ → 将其解释为“$\lfloor \frac{n}{b} \rfloor$”或“$\lceil \frac{n}{b} \rceil$”

① 若对某个常数 $\varepsilon > 0$ 有 $f(n) = O(n^{\log_b a - \varepsilon})$ 则 $T(n) = \Theta(n^{\log_b a})$

② 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$

③ 若对某个常数 $\varepsilon > 0$ 有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$ 且对某个常数 $c < 1$ 和所有足够大的 n 有 $af(\frac{n}{b}) \le cf(n)$ 则 $T(n) = \Theta(f(n))$

在使用主定理之前，我们花一点儿时间尝试理解一下它的含义。对于三种情况的每一种，我们将函数 $f(n)$ 与函数 $n^{\log_b a}$ 进行比较。直觉上，两个函数较大者决定了递归式的解。若函数 $n^{\log_b a}$ 更大，如情况 1，则解为 $T(n)=\Theta(n^{\log_b a})$。若函数 $f(n)$ 更大，如情况 3，则解为 $T(n)=\Theta(f(n))$。若两个函数大小相当，如情况 2，则乘上一个对数因子，解为 $T(n)=\Theta(n^{\log_b a}\lg n)=\Theta(f(n)\lg n)$。

注：三种情况并未覆盖 $f(n)$ 的所有可能性.

eg:

(1) $T(n)=9T(\frac{n}{3})+n$

Step: ① $a=9$ $b=3$ $f(n)=n$

② $n^{\log_b a}=n^{\log_3 9}=n^2$

③ $T(n)=\Theta(n^2)$

找出 $a, b, f(n)$ 对应的值或表达式

求 $n^{\log_b a}$ 并与 $f(n)$ 比较（直观大小即可）→ 多项式 小于/大于

看增长速度是有差 $\frac{1}{n^2}$ 的多项式因子.

(2) $T(n)=T(\frac{2n}{3})+1$

① $a=1$ $b=\frac{3}{2}$ $f(n)=1$

② $n^{\log_b a}=n^{\log_{\frac{3}{2}} 1}=n^0=1$

③ $T(n)=\Theta(\lg n)$

反例: $T(n)=2T(\frac{n}{2})+n\lg n$

$a=2$ $b=2$ $f(n)=n\lg n$

$n^{\log_b a}=n^{\log_2 2}=n$

× 非多项式 小于/大于 比值为分

$af(\frac{n}{b})=2f(\frac{n}{2})=2\cdot\frac{n}{2}\lg\frac{n}{2}$

$=n\lg\frac{n}{2}\le n\lg n$

无法满足 $af(\frac{n}{b})\le cf(n)$ $c<1$

(3) $T(n)=3T(\frac{n}{4})+n\lg n$

① $a=3$ $b=4$ $f(n)=n\lg n$

② $n^{\log_b a}=n^{\log_4 3}<f(n)$ $+3f(\frac{n}{4})=3\frac{n}{4}\lg\frac{n}{4}\le\frac{3}{4}n\lg n=cf(n)$ √

③ $T(n)=\Theta(n\lg n)$

ch15 动态规划 dynamic programming

↓

最优化问题 { continus optimization
discrete optimization

所有问题实例集合.

↑

discrete optimization —定义(四元组)→ (I, f, m, g)

- I: 问题的一个具体输入, x∈I
- f: 所有可行解集合
- m(x,y): 给定一个实例x及一个可行解y, m(x,y)表示这个解的评价值
- g: 目标函数, min/max

$$m(x,y) = g\{m(x,y') \mid y' \in f(x)\}$$

一个可行解y是最优解. 当且仅当其评价值m(x,y)等于所有可行解评价值中的最优值.

一、钢条切割 Rod-Cutting.

1. 定义

Definition:

Input: Given a rod of length n, and the price p_i of each length i, $1 \le i \le n$.

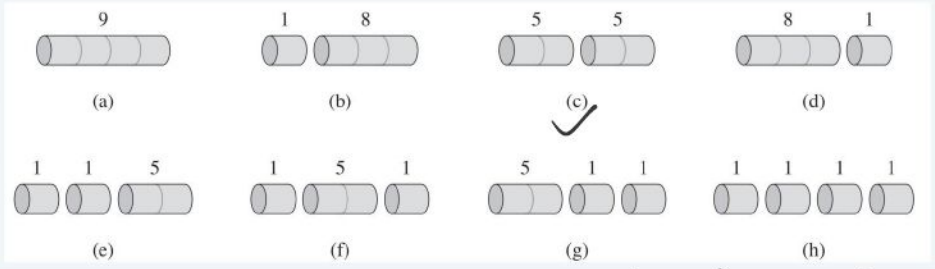
Output: The maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

给定一段长度为n英寸的钢条和一个价格表 p_i $(i=1,2,\cdots,n)$

求切割钢条方案, 使得销售收益 r_n 最大

eg: 若 n=4

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

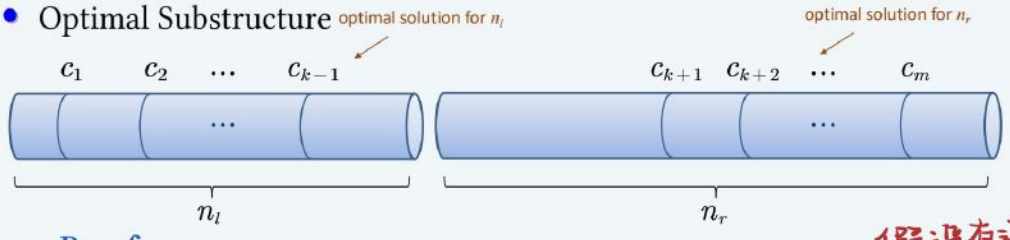


长度为n英寸的钢条共有 2^{n-1} 种不同的切割方案 (在距离钢条左端i处总是可以选择) 切割/不切割 $(i=1,2,\cdots,n-1)$

2. 分析

Optimal Substructure

- Optimal Substructure



- Proof

如果它的最优切法中间切了一刀，把钢条分成了左边长度 n_l 和右边长度 n_r，那么：

- 左边长度 n_l 的切法，必须也是左边这个子问题的最优切法；
- 右边长度 n_r 的切法，也必须是右边这个子问题的最优切法。

否则整体就不是最优。

切钢条问题可以拆成更小的切钢条问题；如果整体切法是最优的，那么被切出来的每一段也必须按照最优方式切，否则我们就可以替换掉那一段，让整体变得更优，这与"原来已经最优"矛盾。

反证法

假设有误

已知长度为n的钢条最优切法 $c_1, c_2, \cdots, c_m$

假设左边长度 n_l 的部分不是最优的

即存在另一种更好的切法 $c_1', c_2', \cdots, c_{k-1}'$

可以把原来整体最优解的左半部分替换掉

↓

整体收益比原来更高

↓

原来的切法不是最优的 矛盾.

注意到，为了求解规模为 n 的原问题，我们先求解形式完全一样，但规模更小的子问题。即当完成首次切割后，我们将两段钢条看成两个独立的钢条切割问题实例。我们通过组合两个相关子问题的最优解，并在所有可能的两段切割方案中选取组合收益最大者，构成原问题的最优解。我们称钢条切割问题满足**最优子结构**(optimal substructure)性质：问题的最优解由相关子问题的最优解组合而成，而这些子问题可以独立求解。

⇓优化

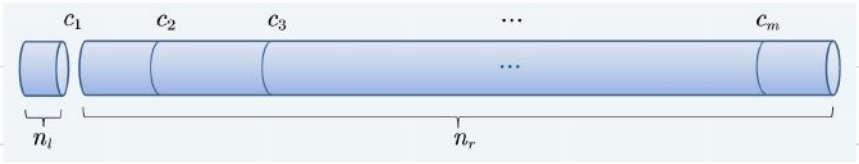
除了上述求解方法外，钢条切割问题还存在一种相似的但更为简单的递归求解方法：我们将钢条从左边切割下长度为 i 的一段，只对右边剩下的长度为 $n-i$ 的一段继续进行切割(递归求解)，对左边的一段则不再进行切割。即问题分解的方式为：将长度为 n 的钢条分解为左边开始一段，以及剩余部分继续分解的结果。这样，不做任何切割的方案就可以描述为：第一段的长度为 n，收益为 p_n，剩余部分长度为 0，对应的收益为 $r_0=0$。于是我们可以得到公式(15.1)的简化版本：

$$r_n = \max_{1\le i\le n}(p_i + r_{n-i}) \qquad (15.2)$$

在此公式中，原问题的最优解只包含一个相关子问题(右端剩余部分)的解，而不是两个。

Bellman equation

$$r(n)=\begin{cases}0 & \text{if } n=0\\ \max\limits_{1\le i\le n}\{p_i + r(n-i)\} & \text{否则}\end{cases}$$



3. 实现

① Recursive top-down 自顶向下递归实现

```
CUT-ROD(p,n)        (p → 数组)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n-i))
6  return q
```

不是动态规划算法 → 反复求解相同的子问题

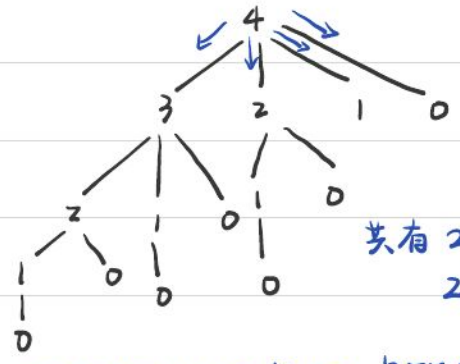
eg:

$r(4)=\max\{p_1+r(3),\ p_2+r(2),\ p_3+r(1),\ p_4+r(0)\}$

$r(3)=\max\{p_1+r(2),\ p_2+r(1),\ p_3+r(0)\}$

$r(2)=\max\{p_1+r(1),\ p_2+r(0)\}$

$r(1)=\max\{p_1+r(0)\}=p_1$



共有 2^n 个结点，2^{n-1} 个叶结点

从根结点到叶结点的一条路径对应长度为 n 的钢条的 2^{n-1} 种方案之一

本质：所有情况枚举一遍

Show that the lower bound of running-time of ROD-CUT(n) is $\Omega(2^n)$.

为了分析 CUR-ROD 的运行时间，令 $T(n)$ 表示第二个参数值为 n 时 CUT-ROD 的调用次数。此值等于递归调用树中根为 n 的子树中的结点总数，注意，此值包含了根结点对应的最初的一次调用。因此 $T(0)=1$，且

$$T(n)=1+\sum_{j=0}^{n-1}T(j)$$

$T(n)=2^n$ —proof→

① Base case

$n=0$ $T(0)=1$

$T(0)=2^0=1$

② 归纳假设

假设对所有小于 n 的规模，都有 $T(j)=2^j$ $0\le j\le n-1$

根据递推式 $T(n)=1+\sum_{j=0}^{n-1}2^j=1+2^n-1=2^n$

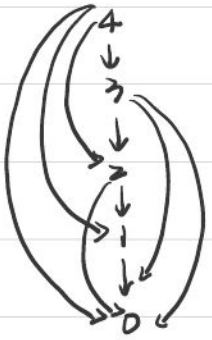
动态规划算法 ⇒ 付出额外的内存空间来节省计算时间

③ Top-down with memoization 带备忘录的自顶向下法

第一种方法称为**带备忘的自顶向下法**(top-down with memoization)[⊖]。此方法仍按自然的递归形式编写过程，但过程会保存每个子问题的解(通常保存在一个数组或散列表中)。当需要一个子问题的解时，过程首先检查是否已经保存过此解。如果是，则直接返回保存的值，从而节省了计算时间；否则，按通常方式计算这个子问题。我们称这个递归过程是**带备忘的**(memoized)，因为它"记住"了之前已经计算出的结果。

```
MEMOIZED-CUT-ROD(n)
1  let r[0..n] be a new array    // a global array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(n)
```

```
MEMOIZED-CUT-ROD-AUX(n)
1  if r[n] ≥ 0                  首先检查所需值是否已知
2      return r[n]
3  if n == 0                    常规解q
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(n - i))
8  r[n] = q
9  return q
```



难一些：当求解一个之前已计算出结果的子问题时，递归调用会立即返回，即 MEMOIZED-CUT-ROD 对每个子问题只求解一次，而它求解了规模为 0，1，…，n 的子问题；为求解规模为 n 的子问题，第 6～7 行的循环会迭代 n 次；因此，MEMOIZED-CUT-ROD 进行的所有递归调用执行此 **for** 循环的迭代次数也是一个等差数列，其和也是 $\Theta(n^2)$，与 BOTTOM-UP-CUT-ROD 内

时间复杂度：$\Theta(n^2)$

③ Bottom-up 自底向下

思想：对于任何子问题，直至它依赖的所有子问题均已求解完成，才会求解它。

```
BOTTOM-UP-CUT-ROD(n)
1  let r[0..n] be a new array    创建一个新数组r[0..n]保存子问题的解
2  r[0] = 0                      长度为0的钢条没有收益
3  for j = 1 to n                升序求解规模为j的子问题
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

时间复杂度：$\Theta(n^2)$

$1+2+3+\cdots+n = \frac{n(n+1)}{2}$

扩展版本：对每个子问题不仅保存最优收益值，还保存对应的切割方案

下面给出的是 BOTTOM-UP-CUT-ROD 的扩展版本，它对长度为 j 的钢条不仅计算最大收益值 r_j，还保存最优解对应的第一段钢条的切割长度 s_j：

```
EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
let r[0..n] and s[0..n] be new arrays
r[0]= 0
for j=1 to n
    q=-∞
    for i=1 to j
        if q<p[i]+r[j-i]
            q=p[i]+r[j-i]
            s[j]=i
    r[j]=q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)        → 调用算法求r和s
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]                   → 输出长度为n的钢条的
4      n = n - s[n]                   完整的最优切割方案
```

示例.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

求r和s

i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

$r[1] = p[1] + r[0] = 1$

$r[2] = \max\begin{cases} p[1]+r[1] = 1+1 = 2 \\ p[2]+r[0] = 5+0 = 5 \end{cases} = 5$

$r[3] = \max\begin{cases} p[1]+r[2] = 1+5 = 6 \\ p[2]+r[1] = 5+1 = 6 \\ p[3]+r[0] = 8 \end{cases} = 8$

$r[4] = \max\begin{cases} p[1]+r[3] = 1+8 = 9 \\ p[2]+r[2] = 5+5 = 10 \\ p[3]+r[1] = 8+1 = 9 \\ p[4]+r[0] = 9 \end{cases} = 10$

$r[5] = \max\begin{cases} p[1]+r[4] = 1+10 = 11 \\ p[2]+r[3] = 5+8 = 13 \\ p[3]+r[2] = 8+5 = 13 \\ p[4]+r[1] = 9+1 = 10 \\ p[5]+r[0] = 10 \end{cases} = 13$

$r[6] = \max\begin{cases} p[1]+r[5] = 1+13 = 14 \\ p[2]+r[4] = 5+10 = 15 \\ p[3]+r[3] = 8+8 = 16 \\ p[4]+r[2] = 10+5 = 15 \\ p[5]+r[1] = 10+1 = 11 \\ p[6]+r[0] = 17 \end{cases} = 17$

$r[7] = \max\begin{cases} p[1]+r[6] = 1+17 = 18 \\ p[2]+r[5] = 5+13 = 18 \\ p[3]+r[4] = 8+10 = 18 \\ p[4]+r[3] = 9+8 = 17 \\ p[5]+r[2] = 10+5 = 15 \\ p[6]+r[1] = 17+1 = 18 \\ p[7]+r[0] = 17 \end{cases} = 18$

$r[8] = \max\begin{cases} p[1]+r[7] = 1+18 = 19 \\ p[2]+r[6] = 5+17 = 22 \\ p[3]+r[5] = 8+13 = 21 \\ p[4]+r[4] = 9+10 = 19 \\ p[5]+r[3] = 10+8 = 18 \\ p[6]+r[2] = 17+5 = 22 \\ p[7]+r[1] = 17+1 = 18 \\ p[8]+r[0] = 20 \end{cases} = 22$

$r[9] = \max\begin{cases} p[1]+r[8] = 1+22 = 23 \\ p[2]+r[7] = 5+18 = 23 \\ p[3]+r[6] = 8+17 = 25 \\ p[4]+r[5] = 9+13 = 22 \\ p[5]+r[4] = 10+10 = 20 \\ p[6]+r[3] = 17+8 = 25 \\ p[7]+r[2] = 17+5 = 22 \\ p[8]+r[1] = 20+1 = 21 \\ p[9]+r[0] = 24 \end{cases} = 25$

$r[10] = \max\begin{cases} p[1]+r[9] = 1+25 = 26 \\ p[2]+r[8] = 5+22 = 27 \\ p[3]+r[7] = 8+18 = 26 \\ p[4]+r[6] = 9+17 = 26 \\ p[5]+r[5] = 10+13 = 23 \\ p[6]+r[4] = 17+10 = 27 \\ p[7]+r[3] = 17+8 = 25 \\ p[8]+r[2] = 20+5 = 25 \\ p[9]+r[1] = 25+1 = 26 \\ p[10] = 30 \end{cases} = 30$

Summary:

设计一个动态规划算法:

① 刻画一个最优解的结构特征 ⇒ 最优子结构 (Optimal Substructure.)

② 递归定义最优解值. → Bellman Equation.

③ 计算最优解的值, 通常采用自底向上的方法

④ 利用计算出的信息构造一个最优解.

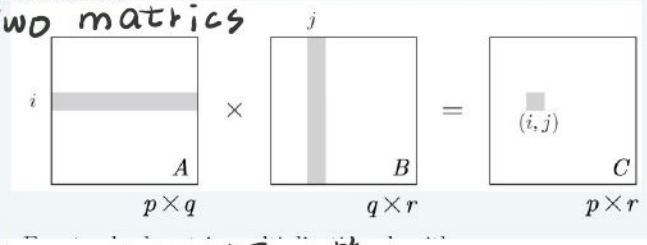
自底向上: 首先求得子问题的最优解, 然后求原问题的最优解.

二、

矩阵链乘问题 Matrix Chain Multiplication

1. 引入

Two matrics



对于标准矩阵乘法算法：

① 计算一个数字：q次标量乘法 → 时间复杂度 $\Theta(pqr)$.

共p×r个数：pqr次标量乘法

② 计算一个数字：q-1次标量加法

共p×r个数：p(q-1)r次标量加法

Three matrics

- A chain of matrices
  - E.g., $\langle A_1^{10\times100},\ A_2^{100\times5},\ A_3^{5\times50}\rangle$
    - $((A_1A_2)A_3)$: 7500 scalar multiplications. $10\times100\times5+10\times5\times50=5000+2500=7500$
    - $(A_1(A_2A_3))$: 75000 scalar multiplications. $100\times5\times50+10\times100\times50=25000+50000=75000$

10×5　10×100　100×50　A big difference!

2. 问题定义

Definition:

Input: Given a chain of matrices $\langle A_1, A_2, \cdots, A_n\rangle$, where matrix A_i has dimension $p_{i-1}\times p_i$. $(1\le i\le n)$

Output: A fully parenthesization of product $A_1A_2\ldots A_n$, which minimizes the number of scalar multiplications.

完全括号化的. 求完全括号化方案，使得计算乘积 $A_1A_2\cdots A_n$ 所需标量乘法次数最少

如下性质的矩阵乘积链为**完全括号化的**(fully parenthesized)：它是单一矩阵，或者是两个完全括号化的矩阵乘积链的积，且已外加括号。例如，如果矩阵链为$\langle A_1, A_2, A_3, A_4\rangle$，则共有 5 种完全括号化的矩阵乘积链：

$(A_1(A_2(A_3A_4)))$
$(A_1((A_2A_3)A_4))$
$((A_1A_2)(A_3A_4))$
$((A_1(A_2A_3))A_4)$
$(((A_1A_2)A_3)A_4)$

3. 分析：

暴力枚举：对一个n个矩阵的链，令P(n)表示可供选择的括号化方案的数量.

$$P(n)=\begin{cases}1 & \text{if } n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & \text{if } n\ge2.\end{cases}$$

数量。当$n=1$时，由于只有一个矩阵，因此只有一种完全括号化方案。当$n\geqslant2$时，完全括号化的矩阵乘积可描述为两个完全括号化的部分积相乘的形式，而两个部分积的划分点在第k个矩阵和第$k+1$个矩阵之间，k为1，2，…，$n-1$中的任意一个值。因此，我们可以得到如下递归

证 $P(n)=\Omega(2^n)$ ⇒ 对常数c $P(n)\ge c\cdot2^n$

$n\ge3$时 $P(n)\ge P(1)P(n-1)+P(n-1)P(1)$

$\because P(1)=1$

$\therefore P(n) \geq 2P(n-1) \geq 2^2 P(n-2) \cdots \geq 2^{n-2} P(2)$

$\therefore P(n) \geq 2^{n-2} = \frac{1}{4} 2^n$

$\therefore P(n) = \Omega(2^n)$

应用动态规划

Step 1：刻画一个最优解的结构特征 最优括号化方案的结构特征

动态规划方法的第一步是寻找最优子结构，然后就可以利用这种子结构从子问题的最优解构造出原问题的最优解。在矩阵链乘法问题中，此步骤的做法如下所述。为方便起见，我们用符号 $A_{i..j}$ $(i \leqslant j)$ 表示 $A_i A_{i+1} \cdots A_j$ 乘积的结果矩阵。可以看出，如果问题是非平凡的，即 $i<j$，那么为了对 $A_i A_{i+1} \cdots A_j$ 进行括号化，我们就必须在某个 A_k 和 A_{k+1} 之间将矩阵链划分开（k 为 $i \leqslant k<j$ 间的整数）。也就是说，对某个整数 k，我们首先计算矩阵 $A_{i..k}$ 和 $A_{k+1..j}$，然后再计算它们的乘积得到最终结果 $A_{i..j}$。此方案的计算代价等于矩阵 $A_{i..k}$ 的计算代价，加上矩阵 $A_{k+1..j}$ 的计算代价，再加上两者相乘的计算代价。

① ② ③

Optimal Substructure.

$(A_i A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$

将问题划分为两个子问题的最优解组合起来

下面我们给出本问题的最优子结构。假设 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案的分割点在 A_k 和 A_{k+1} 之间。那么，继续对“前缀”子链 $A_i A_{i+1} \cdots A_k$ 进行括号化时，我们应该直接采用独立求解它时所得的最优方案。这样做的原因是什么呢？如果不采用独立求解 $A_i A_{i+1} \cdots A_k$ 所得的最优方案来对它进行括号化，那么可以将此最优解代入 $A_i A_{i+1} \cdots A_j$ 的最优解中，代替原来对子链 $A_i A_{i+1} \cdots A_k$ 进行括号化的方案（比 $A_i A_{i+1} \cdots A_k$ 最优解的代价更高），显然，这样得到的解比 $A_i A_{i+1} \cdots A_j$ 原来的“最优解”代价更低：产生矛盾。对子链 $A_{k+1} A_{k+2} \cdots A_j$，我们有相似的结论：在原问题 $A_i A_{i+1} \cdots A_j$ 的最优括号化方案中，对子链 $A_{k+1} A_{k+2} \cdots A_j$ 进行括号化的方法，就是它自身的最优括号化方案。

Step 2：递归定义最优解的值

令 $m[i,j]$ 表示计算 $A_i A_{i+1} \cdots A_j$ 所需标量乘法次数的最小值. ⇒ Goal $m[1,n]$.

Bellman equation

$$m[i,j] = \begin{cases} 0 & i=j \\ \min\limits_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \} & i<j. \end{cases}$$

① ② ③

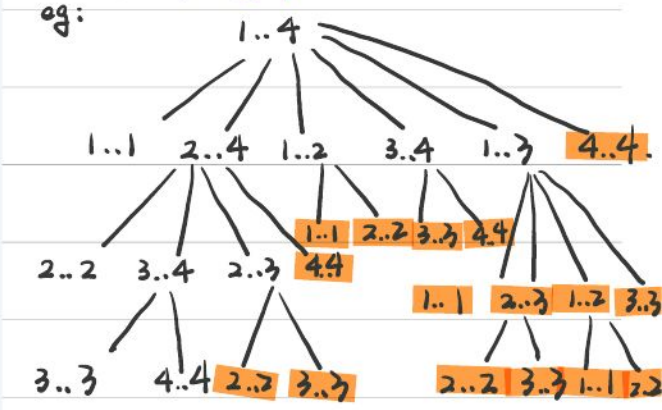
Step 3：计算最优代价.（通常采用自底向上的方法）.

实现① Recursive top-down

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

不是动态规划算法

eg:



在带备忘录的自顶向下直接查表.

计算时间复杂度.

令 $T(n)$：计算 n 个矩阵的矩阵链的最优括号化方案所花费的时间

$$T(n) \geq \begin{cases} 1 & \text{if } n=1 \\ 1+\sum_{k=1}^{n-1}(T(k)+T(n-k)+1) & \text{if } n>1 \end{cases}$$

$T(n) \geq 2\sum_{k=1}^{n-1} T(k)+n$ 证 $T(n)=\Omega(2^n)$ 即常数 c $T(n) \geq c\cdot 2^n$

$T(n) \geq 2\sum_{k=1}^{n-1} T(k)+n \geq 2\sum_{k=1}^{n-1} T(k) \geq 2T(n-1)$

$T(n) \geq 2T(n-1) \geq 2^2 T(n-2) \cdots \geq 2^{n-1} T(1) = 2^{n-1}$

∴ the lower bound of running-time of RECURSIVE-MATRIX-CHAIN(p,i,j) is $\Omega(2^n)$

实现② Top-down with memoization

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

保存计算出矩阵 $A_{i..j}$ 的最小计算代价 m[i,j]

设定未求解标记

创建一个新的二维表 m，行和列都从 1-n（创建 n×n 的表格）。

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

递归 top-down

首先检查所需值是否已知。

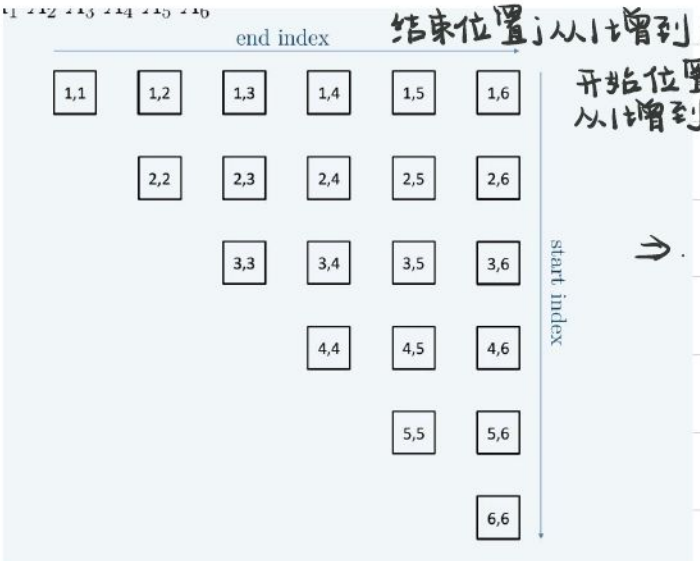
时间复杂度 $O(n^3)$

思想：

带备忘的递归算法为每个子问题维护一个表项来保存它的解。每个表项的初值设为一个特殊值，表示尚未填入子问题的解。当递归调用过程中第一次遇到子问题时，计算其解，并存入对应表项。随后每次遇到同一个子问题，只是简单地查表，返回其解。

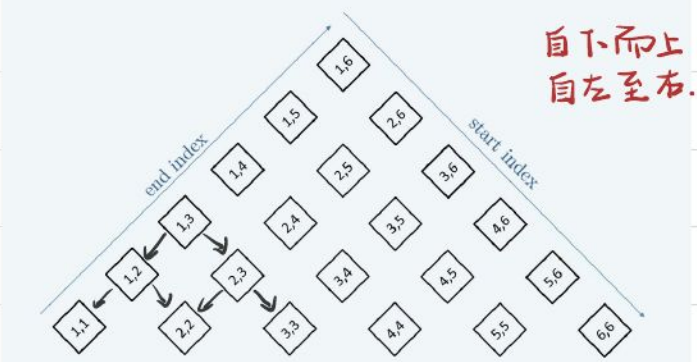
下面给出的是带备忘的 RECURSIVE-MATRIX-CHAIN 版本。注意它与带备忘的自顶向下

eg: A_1 A_2 A_3 A_4 A_5 A_6



结束位置 j 从 1 增到 6

开始位置从 1 增到 6



自下而上
自左至右。

按照长度递增的顺序求解矩阵链括号化问题，并按对应的顺序填写表 m

实现③ Bottom-up

- **Implementation-3: Bottom-up**

```
MATRIX-CHAIN-ORDER(p)
1  n = p.length − 1
2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3  for i = 1 to n
4      m[i, i] = 0
5  for l = 2 to n          // l is the chain length
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          m[i, j] = ∞
```

$p_0, p_1, \cdots p_n$

共有 n 个矩阵

⇒ 最优值 m[i,j] 对应的分割点 k。

⇒ 长度为 1 的链的最小计算代价

按子问题从小到大填表

确定当前这段矩阵链起点 i，终点 j。

其中 l：链长。

n-l+1：起点 i 最多只能到 n-l+1 否则后面的矩阵数量就不够了。

因此 s 表只记录那些长度至少为 2 的矩阵链，也就是：

$$i < j$$

所以范围写成：

$$s[1..n-1, 2..n]$$

意思是：

```
i 可以从 1 到 n-1
j 可以从 2 到 n
```

并且实际使用时满足：

$$i < j$$

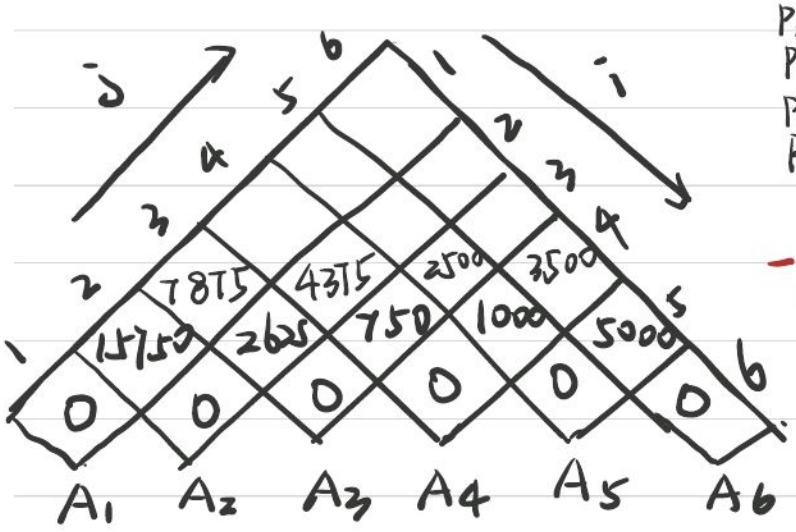
```
for k = i to j − 1          试分位置.
    q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
        s[i, j] = k          保存当前划分
return m and s
```

时间复杂度 $O(n^3)$

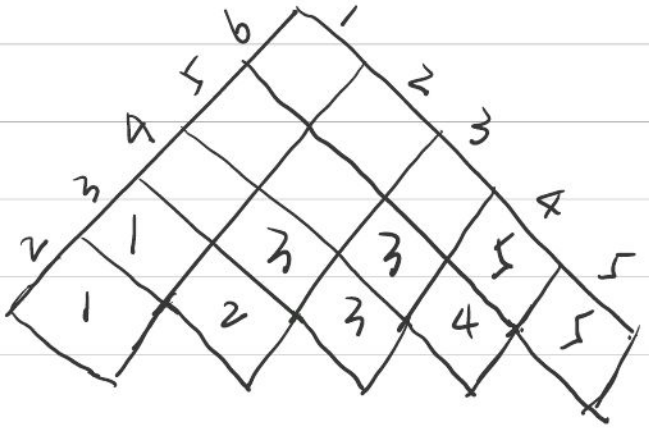
示例：A_1 A_2 A_3 A_4 A_5 A_6

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

$P_0=30$
$P_1=35$
$P_2=15$
$P_3=5$
$P_4=10$
$P_5=20$
$P_6=25$



一层层算.（从左到右，从下到上）



$L=1$

$m[1,2] = \min\{m[1,1]+m[2,2]+P_0P_1P_2\}$
$= 30\times35\times15 = 15750$

$m[2,3] = \min\{m[2,2]+m[3,3]+P_1P_2P_3\}$
$= 35\times15\times5 = 2625$

$m[3,4] = \min\{m[3,3]+m[4,4]+P_2P_3P_4\}$
$= 15\times5\times10 = 750$

$m[4,5] = \min\{m[4,4]+m[5,5]+P_3P_4P_5\}$
$= 5\times10\times20 = 1000$

$m[5,6] = \min\{m[5,5]+m[6,6]+P_4P_5P_6\}$
$= 10\times20\times25 = 5000$

$L=2$

$m[1,3] = \min\{m[1,1]+m[2,3]+P_0P_1P_3 = 2625+30\times35\times5 = 7875,\ m[1,2]+m[3,3]+P_0P_2P_3 = 15750+30\times15\times5 = 18000\} = 7875$

$m[2,4] = \min\{m[2,2]+m[3,4]+P_1P_2P_4 = 750+35\times15\times10 = 6000,\ m[2,3]+m[4,4]+P_1P_3P_4 = 2625+1750 = 4375\} = 4375$

$m[3,5] = \min\{m[3,3]+m[4,5]+P_2P_3P_5 = 1000+15\times5\times20 = 2500,\ m[3,4]+m[5,5]+P_2P_4P_5 = 750+15\times10\times20 = 3750\} = 2500$

$m[4,6] = \min\{m[4,4]+m[5,6]+P_3P_4P_6 = 5000+5\times10\times25 = 6250,\ m[4,5]+m[6,6]+P_3P_5P_6 = 1000+5\times20\times25 = 3500\} = 3500$

step 4：构造最优解

计算 $A_{[1..n]+1..n}$时应进行的最后一次矩阵乘法运算。下面给出的递归过程可以输出$\langle A_i, A_{i+1}, \cdots, A_j\rangle$的最优括号化方案，其输入为 MATRIX-CHAIN-ORDER 得到的表 s 及下标 i 和 j。调用 PRINT-OPTIMAL-PARENS(s, 1, n)即可输出$\langle A_1, A_2, \cdots, A_n\rangle$的最优括号化方案。

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i          Ai
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i,j])
5      PRINT-OPTIMAL-PARENS(s, s[i,j]+1, j)
6      print ")"
```

对图 15-5 中的例子，调用 PRINT-OPTIMAL-PARENS(s, 1, 6)输出括号化方案

$((A_1(A_2A_3))((A_4A_5)A_6))$

三. 最长公共子序列 Longest Common Subsequence (LCS)

Definition:

Input: Given two sequences $X = \langle x_1, x_2, \ldots, x_m\rangle$ and $Y = \langle y_1, y_2, \ldots, y_n\rangle$.

Output: A maximum-length common subsequence of X and Y.

解释：何为公共子序列.

给定一个序列 $X=\langle x_1, x_2, \cdots x_m\rangle$，另一个序列 $Z=\langle z_1, z_2, \cdots z_k\rangle$ 满足存在一个严格递增的下标序列 $\langle i_1, i_2, \cdots i_k\rangle$ 对所有 $j=1,2,\cdots n$，满足 $x_j = z_i$，则 Z 为 X 的子序列 ⇒ 若 Z 既是 X 的子序列，又是 Y 的子序列，则称其为 X 和 Y 的公共子序列.

[不要求连续出现，但先后顺序一致]

eg: $X=\langle A, B, C, B, D, A, B\rangle$

$Y=\langle B, D, C, A, B, A\rangle$

X 和 Y 的公共子序列：$\langle B\ C\ B\ A\rangle$ (部分).

$\langle B\ D\ A\ B\rangle$

step 1：最优子结构 Optimal Substructure.

暴力枚举：

如果用暴力搜索方法求解 LCS 问题，就要穷举 X 的所有子序列，对每个子序列检查它是否也是 Y 的子序列，记录找到的最长子序列。X 的每个子序列对应 X 的下标集合 $\{1, 2, \cdots, m\}$的一个子集，所以 X 有 2^m 个子序列，因此暴力方法的运行时间为指数阶，对较长的序列是不实用的。

LCS 最优子结构：

令 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots y_n\rangle$ 为两个序列，$Z=\langle z_1, z_2, \cdots z_k\rangle$ 为 X 和 Y 的任意 LCS

① 若 $x_m = y_n$，则 $z_k = x_m = y_n$ 且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个 LCS

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

③ 若 $x_m \neq y_n$ 那么 $z_k \neq y_n$，意味着 Z 是 X 和 Y_{n-1} 的一个 LCS

重要思想："前缀" ⇒ 给定一个序列 $X=\langle x_1, x_2, \cdots x_m\rangle$，对 $i=0,1,2,\cdots m$，定义 X 的第 i 前缀为 $X_i=\langle x_1, x_2, \cdots x_i\rangle$，其中 X_0 为空串

证明定理：

①假设z_k为X和Y最长公共子序列

(1)如果$z_k \neq x_m$，那么可将$z_k = x_m = y_n$追加到z的末尾，得到X和Y的一个长度为$k+1$的公共子序列，与已是X和Y最长公共子序列矛盾

∴一定有$z_k = x_m = y_n$

②这样z_{k-1}是x_{m-1}和y_{n-1}的一个长度为$k-1$的公共子序列，我们希望其为LCS

假设存在x_{m-1}和y_{n-1}的一个长度大于$k-1$的公共子序列W，则将$x_m = y_n$追加到W的末尾会得到X和Y的一个长度大于k的公共子序列，矛盾

(2)如果$z_k \neq x_m$，那么z是x_{m-1}和Y的一个公共子序列，下证其为LCS

如果存在x_{m-1}和Y的一个长度大于k的公共子序列W，那么W也是X和Y的公共子序列，

与已是X和Y的最长公共子序列假设矛盾

(3)与(2)对称

Step 2: Bellman Equation

定理 15.1 意味着，在求 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 的一个 LCS 时，我们需要求解一个或两个子问题。①如果 $x_m = y_n$，我们应该求解 X_{m-1} 和 Y_{n-1} 的一个 LCS。将 $x_m = y_n$ 追加到这个 LCS 的末尾，就得到 X 和 Y 的一个 LCS。②如果 $x_m \neq y_n$，我们必须求解两个子问题：求 X_{m-1} 和 Y 的一个 LCS 与 X 和 Y_{n-1} 的一个 LCS。两个 LCS 较长者即为 X 和 Y 的一个 LCS。由

定义c[i,j]表示x_i和y_j的LCS的长度.

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

Step3：计算LCS的长度 自底向上计算

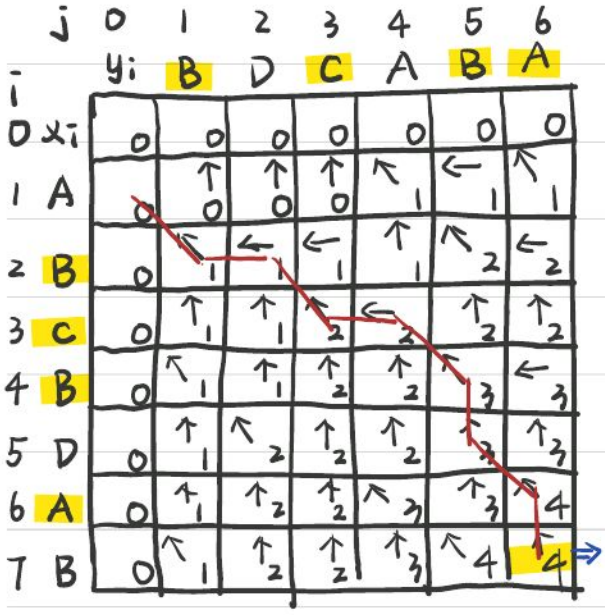
```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m                          初始化
 5      c[i, 0] = 0                         初始化
 6  for j = 0 to n                          初始化
 7      c[0, j] = 0                         初始化
 8  for i = 1 to m                          从左到右计算c的第一行，然后
 9      for j = 1 to n                      计算第二行.
10          if x_i == y_j
11              c[i, j] = c[i - 1, j - 1] + 1
12              b[i, j] = "↖"
13          elseif c[i - 1, j] ≥ c[i, j - 1]
14              c[i, j] = c[i - 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j - 1]
17              b[i, j] = "←"
18  return c and b
```

过程 LCS-LENGTH 接受两个序列 $X=\langle x_1, x_2, \cdots, x_m\rangle$ 和 $Y=\langle y_1, y_2, \cdots, y_n\rangle$ 为输入。它将 $c[i,j]$ 的值保存在表 $c[0..m, 0..n]$ 中，并按**行主次序**(row-major order)计算表项(即首先由左至右计算 c 的第一行，然后计算第二行，依此类推)。过程还维护一个表 $b[1..m, 1..n]$，帮助构造最优解。$b[i,j]$ 指向的表项对应计算 $c[i,j]$ 时所选择的子问题最优解。过程返回表 b 和表 c，$c[m,n]$ 保存了 X 和 Y 的 LCS 的长度。

过程运行时间$\Theta(mn)$，每个表项计算时间$\Theta(1)$.

示例

X=<A,B,C,B,D,A,B> Y=<B,D,C,A,B,A> 一个LCS: <B C B A>



⇒ X和Y的一个LCS

书本← 和个值相同优先↑

一行行地填写

做题:

判断最末尾两个数字是否相同,若相同↖

若不同比较↑←

不同 谁大写谁

相同 优先↖

红线序列中每个↖对应的表项表示 $x_i=y_j$ 是LCS的一个元素

Step 4: 构造 LCS 利用计算出的信息构造最优解.

```
PRINT-LCS(b, X, i, j)
1  if i==0 or j==0
2      return
3  if b[i,j]=="↖"
4      PRINT-LCS(b, X, i−1, j−1)
5      print x_i
6  elseif b[i,j]=="↑"
7      PRINT-LCS(b, X, i−1, j)
8  else PRINT-LCS(b, X, i, j−1)
```

运行时间 O(m+n)

优化:

不过，LCS-LENGTH 的空间需求是可以渐近减少的，因为在任何时刻它只需要表 c 中的两行：当前正在计算的一行和前一行(实际上，练习 15.4-4 要求设计一个算法，只使用一行多一点的空间来计算 LCS 的长度)。如果我们只需计算 LCS 的长度，这一改进是有效的。但如果需要重

```
LCS_ONE_DIM(X, Y, m, n):
  // 初始化dp数组，等价于原c[0][*]=0
  dp = array[0...n] initialized to 0
  for i from 1 to m:          // 逐个遍历X字符，逐行更新
    pre = dp[0]               // pre保存：本轮开始前 dp[0]=c[i-1][0]=0，即左上初始值
    for j from 1 to n:        // 遍历Y字符
      temp = dp[j]            // temp暂存更新前dp[j] = c[i-1][j]（原上侧值）
      if X[i] == Y[j]:
        dp[j] = pre + 1// 匹配，用上一轮左上值pre=c[i-1][j-1]
      else:
        dp[j] = max(dp[j], dp[j-1])
        // dp[j]：旧值=上c[i-1][j]；dp[j-1]：新值=左c[i][j-1]
      pre = temp              // 更新pre为下一轮的左上值（本轮旧dp[j]）
  return dp[n]                // dp[n]就是LCS长度
```

四、
动态规划原理

何时用动态规划

要素 ① 最优子结构：如果一个问题的最优解包含其子问题的最优解

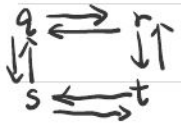
1. 证明问题最优解的第一个组成部分是做出一个选择，例如，选择钢条第一次切割位置，选择矩阵链的划分位置等。做出这次选择会产生一个或多个待解的子问题。

2. 对于一个给定问题，在其可能的第一步选择中，你假定已经知道哪种选择才会得到最优解。你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择。

3. 给定可获得最优解的选择后，你确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间。

4. 利用“剪切－粘贴”(cut-and-paste)技术证明：作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。证明这一点是利用反证法：假定子问题的解不是其自身的最优解，那么我们就可以从原问题的解中“剪切”掉这些非最优解，将最优解“粘贴”进去，从而得到原问题一个更优的解，这与最初的解是原问题最优解的前提假设矛盾。如果原问题的最优解包含多个子问题，通常它们都很相似，我们可以将针对一个子问题的“剪切－粘贴”论证方法稍加修改，用于其他子问题。

反例
无权最长简单路径.



q→t最长路径：
q→r→t
但q→r和r→t不是q与r和r与t之间的最长路径.

② 重叠子问题：问题的递归算法会反复求解相同的子问题

对于每个子问题求解一次，将解存入一个表中. 当再次需要这个子问题时直接查表，每次查表的代价为常量时间

我们可以用子问题的总数和每个子问题需要考察多少种选择这两个因素的乘积来粗略分析动态规划算法的运行时间。对于钢条切割问题，共有 $\Theta(n)$ 个子问题，每个子问题最多需要考察 n 种选择，因此运行时间为 $O(n^2)$。矩阵链乘法问题共有 $\Theta(n^2)$ 个子问题，每个子问题最多需要考察 $n-1$ 种选择，因此运行时间为 $O(n^3)$。(练习 15.2-5 要求证明运行时间实际为 $\Theta(n^3)$。)

⇒ 时间复杂度分析思路.

Ch16 贪心算法 → 在每一步做出当时看起来最佳的选择. 总是做出局部最优的选择.

一、活动选择问题 Activity-Selection problem

1. 定义

Definition:

Input: A set $S = \{a_1, a_2, \ldots, a_n\}$ of proposed activities. Each activity a_i has a start time s_i and a finish time f_i.

Output: A maximum-size subset $S' \subset S$ of compatible activities. 最大兼容活动集

假定活动已按结束时间单调递增顺序排序

相兼容的活动集合。假定有一个 n 个**活动**(activity)的集合 $S=\{a_1, a_2, \cdots, a_n\}$，这些活动使用同一个资源(例如一个阶梯教室)，而这个资源在某个时刻只能供一个活动使用。每个活动 a_i 都有一个**开始时间** s_i 和一个**结束时间** f_i，其中 $0 \leqslant s_i < f_i < \infty$。如果被选中，任务 a_i 发生在半开时间区间 $[s_i, f_i)$ 期间。如果两个活动 a_i 和 a_j 满足 $[s_i, f_i)$ 和 $[s_j, f_j)$ 不重叠，则称它们是**兼容的**。也就是说，若 $s_i \geqslant f_j$ 或 $s_j \geqslant f_i$，则 a_i 和 a_j 是兼容的。在**活动选择问题**中，我们希望选出一个最大兼容活动集。假定活动已按结束时间的单调递增顺序排序：

2. 动态规划方法求解

Step1: optimal substructure.

令 S_{ij} 表示在 a_i 结束之后开始，且在 a_j 开始之前结束的活动集合(不包含 a_i、a_j)

假定我们希望求 S_{ij} 的一个最大的相互兼容的活动子集，将其记作 A_{ij}

假设 A_{ij} 包含活动 a_k，我们可以得到两个子问题：寻找 S_{ik} 和 S_{kj} 中的兼容活动

令 $A_{ik} = A_{ij} \cap S_{ik}$　$A_{kj} = A_{ij} \cap S_{kj}$

$\therefore A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

$|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$

反证法证明最优解 A_{ij} 必然包含两个子问题 S_{ik} 和 S_{kj} 的最优解.

否则我们可以找到 S_{kj} 的一个兼容活动子集 A'_{kj}，满足 $|A'_{kj}| > |A_{kj}|$，则将 A'_{kj} 代替 A_{kj} 作为 S_{ij} 的最优解的一部分，则可构造一个兼容活动子集.

其大小 $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1$，与 A_{ij} 是最优解的假设矛盾.

同理 S_{ik} 类似可证

Step2: Bellman Equation

$c[i,j]$ 表示集合 S_{ij} 的最优解的大小

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Step3: 自底向下算法

$s[k]$：活动 k 开始时刻

$e[k]$：活动 k 结束时刻

```
1  ACTIVITY-DP-BOTTOMUP(s,e,n)
2  1  m = n+2          // 下标范围: 0~n+1
3  2  let c[0..n+1, 0..n+1] be new table
4  3  for i = 0 to n+1
5  4       c[i,i] = 0
6  5  // len为区间[i,j]跨度, len=j-i, 从小到大枚举区间
```

```
6  for len = 1 to n+1
7      for i = 0 to (n+1-len)
8          j = i + len
9          maxval = 0
10         for k = i+1 to j-1
11             // 判断ak属于S_ij: e[i]<s[k] 且 e[k]<s[j]
12             if e[i] < s[k] and e[k] < s[j]
13                 tmp = c[i,k] + c[k,j] + 1
14                 if tmp > maxval
15                     maxval = tmp
16         c[i,j] = maxval
17 return c
```

保证所有子问题在求解前可求

(2) 贪心算法求解.

1. Earliest start time: min $s(i)$



2. Shortest interval: min $\{f(i) - s(i)\}$



3. Fewest conflicts: $\min_{i=1..n} |\{j: j \text{ is not compatible with } i\}|$

每次优先挑选冲突活动数量最少的活动i



4. Earliest finish time: min $f(i)$ ✓

① 递归贪心算法 活动开始时间↗活动结束时间

时间复杂度 $\Theta(n)$

每个活动被且只被while循环检查一次.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

要求解子问题 $S_k = \{a_i \in S: a_i \text{ 为在 } a_k \text{ 结束之后开始的任务集合.}\}$

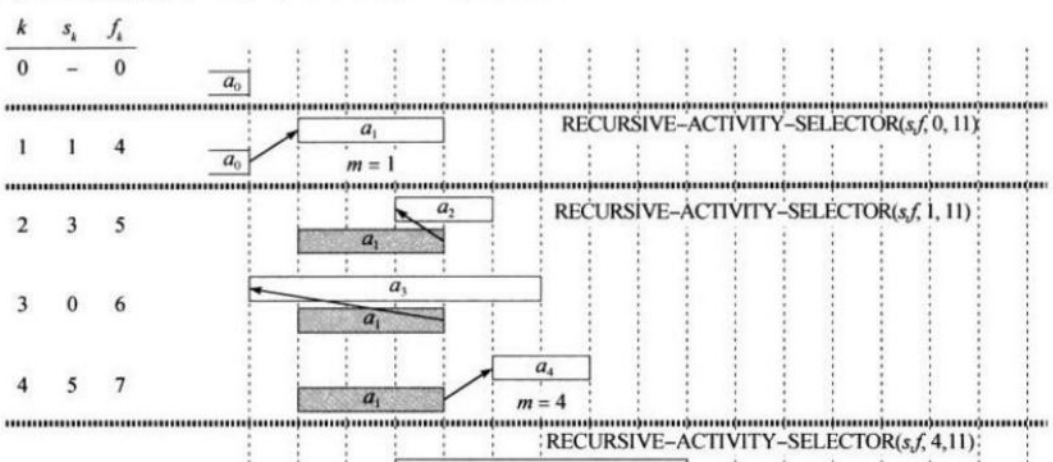
隐含 s[m] ≥ f[k]

为了方便算法初始化,我们添加一个虚拟活动 a_0,其结束时间 $f_0 = 0$.

⇒ S_0 即为完整的活动集 S

f, k, n)的过程中，第 2～3 行 **while** 循环查找 S_k 中最早结束的活动。循环检查 a_{k+1}，a_{k+2}，…，a_n，直至找到第一个与 a_k 兼容的活动 a_m，此活动满足 $s_m \geq f_k$。① 如果循环因为查找成功而结束，第 5 行返回 $\{a_m\}$ 与 RECURSIVE-ACTIVITY-SELECTOR(s，f，m，n)返回的 S_m 的最大子集的并集。② 循环也可能因为 $m > n$ 而终止，这意味着我们已经检查了 S_k 中所有活动，未找到与 a_k 兼容者。在此情况下，$S_k = \emptyset$，因此第 6 行返回 $\emptyset$。

示例



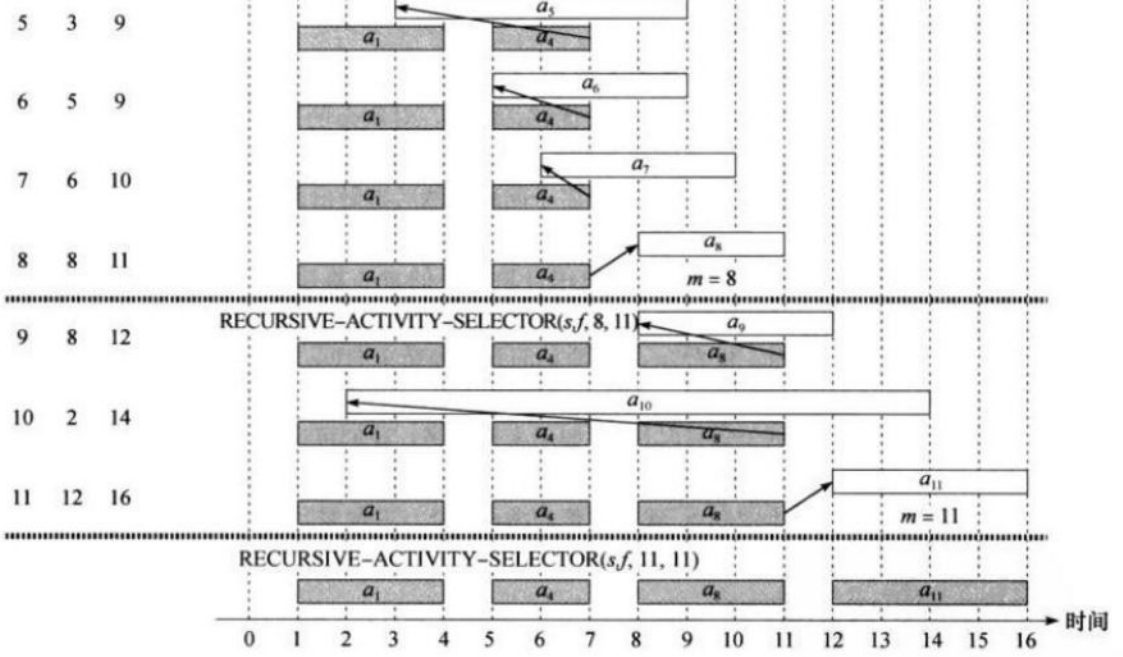


图 16-1 对前文给出的 11 个活动执行 RECURSIVE-ACTIVITY-SELECTOR 的过程。每次递归调用中处理的活动位于水平线之间。虚拟活动 a_0 于时刻 0 结束，因此第一次调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11)会选择活动 a_1。在每次递归调用中，被选择的活动用阴影表示，而白底方框表示正在处理的活动。如果一个活动的开始时间早于最近选中的活动的结束时间(两者间的箭头是指向左侧的)，它将被丢弃。否则(箭头指向右侧)，将选择该活动。最后一次递归调用 RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11)返回∅。选择的活动的最终结果集为$\{a_1, a_4, a_8, a_{11}\}$

②迭代贪心算法

假定输入活动已按结束时间单调递增顺序排序

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1: n = s.length
2: SORT activities by finish times and renumber so that f_1 ≤ f_2 ≤ ··· ≤ f_n
3: A = {a_1}
4: k = 1
5: for m = 2 to n do
6:     if s[m] ≥ f[k] then
7:         A = A ∪ {a_m}
8:         k = m
9:     end if
10: end for
11: return A
```

k记录最近加入集合A的活动下标

f_k 总是A中活动的最大结束时间

运行时间$\Theta(n)$

③证明贪心算法的正确性

法一：

反证法

假设贪心算法得到的不是最优解.

令$i_1, i_2, \cdots i_k$表示由贪心算法选择的活动集

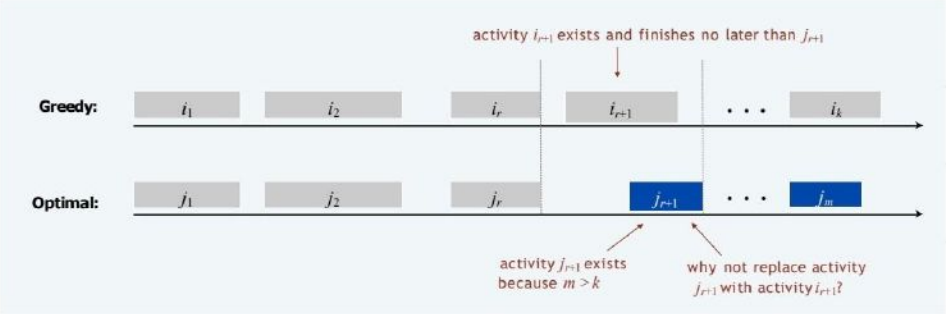
$j_1, j_2, \cdots j_m$表示某个最优解选出的活动集合且$m>k$

找到贪心解和这个最优解前r个活动是一样的且r已经尽可能大

即$i_1=j_1, i_2=j_2 \cdots i_r=j_r$

由于贪心算法选完i_r之后，会在还能选择的活动中挑选结束时间最早活动，所以$f(i_{r+1})$一定不小于$f(j_{r+1})$，因此我们可用i_{r+1}替换j_{r+1}且仍维持最优

此时我们有前r+1个活动是一样的.
与原本r最大可能值矛盾.



假设贪心算法不是最优的，那么存在一个比贪心解更好的最优解。我们选取一个与贪心解前缀相同最长的最优解，设它们前 r 个活动相同。接下来比较第 $r+1$ 个活动。由于贪心算法每一步都选择当前可选活动中结束时间最早的活动，所以贪心选出的 i_{r+1} 的结束时间不会晚于最优解中的 j_{r+1}。因此，用 i_{r+1} 替换 j_{r+1} 后，后续活动仍然可以安排，解仍然可行，而且活动数量不变，所以仍然是最优解。但这样得到的新最优解与贪心解前 $r+1$ 个活动相同，和 r 是最大相同前缀长度矛盾。因此，贪心算法一定是最优的。

法二：

考虑任意非空子问题 S_k，令 a_m 是 S_k 中结束时间最早的活动，则 a_m 在 S_k 的某个最大兼容活动子集中.

令 A_k 是 S_k 的一个最大兼容活动子集且 a_j 是 A_k 中结束时间最早的活动.

① 若 $a_j = a_m$，则已经证明 a_m 在 S_k 的某个最大兼容活动子集中.

② 若 $a_j \neq a_m$ 令集合 $A_k' = A_k - \{a_j\} \cup \{a_m\}$ 即将 A_k 中的 a_j 替换为 a_m

易知 A_k' 中的活动不相交的（A_k 中活动不相交且 $f_m \leq f_j$）

又 $|A_k'| = |A_k| \Rightarrow A_k'$ 也是 S_k 的一个最大兼容活动子集且其包含 a_m

二、贪心算法原理

在一起，就会得到 S_k 的最优解。更一般地，我们可以按如下步骤设计贪心算法：

1. 将最优化问题转化为这样的形式：对其做出一次选择后，只剩下一个子问题需要求解。
2. 证明做出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的。
3. 证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构。

① 贪心选择性质. ⇒ 不依赖将来的选择或是子问题的解.

一个贪心算法通常是自顶向下的，进行一次又一次选择，将给定问题实例变得更小.

实例变得更小。

当然，我们必须证明每个步骤做出贪心选择能生成全局最优解。如定理 16.1 所示，这种证明通常首先考查某个子问题的最优解，然后用贪心选择替换某个其他选择来修改此解，从而得到一个相似但更小的子问题。

如果进行贪心选择时我们不得不考虑众多选择，通常意味着可以改进贪心选择，使其更为高效。例如，在活动选择问题中，假定我们已经将活动按结束时间单调递增顺序排好序，则对每个活动能够只需处理一次。通过对输入进行预处理或者使用适合的数据结构（通常是优先队列），我们通常可以使贪心选择更快速，从而得到更高效的算法。



② 最优子结构 将子问题的最优解与贪心选择组合到一起生成原问题的最优解.

Ch22 基本的图算法

1. 图的表示

图 $G=(V,E)$：V → 顶点集合 vertex set；E → 边集合 edge set

两种表示方法：
- 邻接链表 ⇒ 常用于稀疏图：边的条数$|E|$远远小于$|V|^2$
- 邻接矩阵 ⇒ 常用于稠密图/图规模较小；快速判断任意两个结点之间是否有边相连。

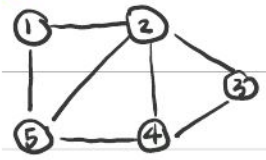
① 邻接链表

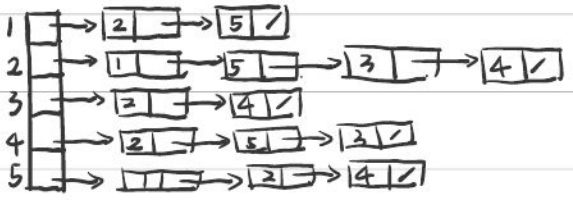
对于图 $G=(V,E)$ 来说，其**邻接链表表示**由一个包含 $|V|$ 条链表的数组 Adj 所构成，每个结点有一条链表。对于每个结点 $u\in V$，邻接链表 $Adj[u]$ 包含所有与结点 u 之间有边相连的结点 v，即 $Adj[u]$ 包含图 G 中所有与 u 邻接的结点（也可以说，该链表里包含指向这些结点的指针）。由于邻接链表代表的是图的边，在伪代码里，我们将数组 Adj 看做是图的一个属性，就如我们将边集合 E 看做是图的属性一样。因此，在伪代码里，我们将看到 $G.Adj[u]$ 这样的表示。

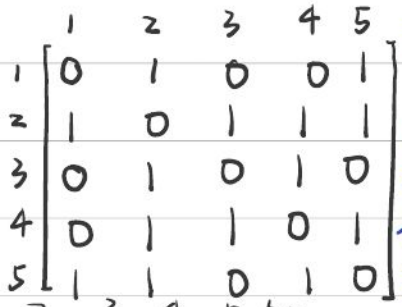
如果 G 是一个有向图，则对于边 (u,v) 来说，结点 v 将出现在链表 $Adj[u]$ 里，因此，所有邻接链表的长度之和等于 $|E|$。如果 G 是一个无向图，则对于边 (u,v) 来说，结点 v 将出现在链表 $Adj[u]$ 里，结点 u 将出现在链表 $Adj[v]$ 里，因此，所有邻接链表的长度之和等于 $2|E|$。但不管是有向图还是无向图，邻接链表表示法的存储空间需求均为 $\Theta(V+E)$，这正是我们所希望

优点

eg：无向图



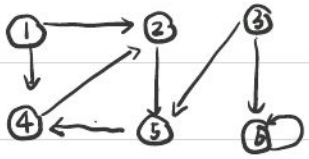


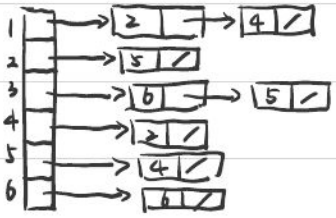


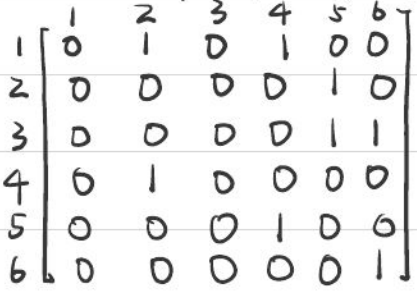
对称矩阵 $A=A^T$

（某些应用可能只需存储对角线及其以上自己部分）

有向图







缺陷：

图变种。

邻接链表的一个潜在缺陷是无法快速判断一条边 (u,v) 是否是图中的一条边。唯一的办法是在邻接链表 $Adj[u]$ 里面搜索结点 v。邻接矩阵表示则克服了这个缺陷，但付出的代价是更大的存储空间消耗（存储空间的渐近数量级更大）（关于如何在邻接链表上进行快速边搜索的信息，请参阅练习 22.1-8）。

② 邻接矩阵

$|V|\times|V|$ 矩阵 $A=(a_{ij})$ → $a_{ij}=\begin{cases}1 & (i,j)\in E\\ 0 & \text{其他}\end{cases}$

不管一个图有多少条边，邻接矩阵的空间需求皆为 $\Theta(V^2)$

优：简单快速判断一条边 (u,v) 是否是图中的一条边

缺：存储空间需求大

2. 图的性质

① Handshaking lemma 握手定理

无向图：

在一个无向图 $G=(V,E)$ 中，所有顶点的度数加起来等于边数的2倍

$$\sum_{v\in V} d(v)=2|E|$$

proof：

因为在**无向图**中，一条边一定连接两个顶点。

比如有一条边：

$$(u,v)$$

那么这条边会：

- 给顶点 u 的度数贡献 1；
- 给顶点 v 的度数贡献 1。

所以一条边在统计所有顶点度数时，会被算两次。

因此如果图中有 $|E|$ 条边，那么所有顶点的度数总和就是：

$$2|E|$$

有向图：

所有顶点的入度之和=所有顶点的出度之和=有向边的数量

对于有向图，每条边都有方向。

一条有向边：

$$u \to v$$

会：

- 从 u 出去，所以给 u 的出度贡献 1；
- 进入 v，所以给 v 的入度贡献 1。

因此在有向图中：

$$\sum_{v\in V} indeg(v) = |E|$$

$$\sum_{v\in V} outdeg(v) = |E|$$

所以：

$$\sum indeg(v) = \sum outdeg(v) = |E|$$

② 奇度数顶点个数是偶数

在任意无向图中，度数为奇数的顶点个数一定为偶数

proof：

根据握手定理：

$$\sum_{v\in V} d(v) = 2|E|$$

右边 $2|E|$ 一定是偶数。

所以所有顶点度数的总和一定是偶数。

现在把顶点分成两类：

- 度数为偶数的顶点；
- 度数为奇数的顶点。

偶数度顶点的度数加起来，结果还是偶数。

所以剩下的奇数度顶点的度数加起来，也必须是偶数。

但是：

- 奇数个奇数相加，结果是奇数；
- 偶数个奇数相加，结果是偶数。

既然奇度数顶点的度数和必须是偶数，那么奇度数顶点的个数就只能是偶数。

3. 路径.

(1) 路径

从J顶点u到J顶点u'的一条长度为k的路径记为 $\langle v_0, v_1, v_2, \cdots v_k \rangle$ 且满足 $u=v_0$, $u'=v_k$, $(v_{i-1}, v_i) \in E$ 其中 $i=1, 2, \cdots k$.

(2) 路径长度：路径中边的数量

(3) reachable 可达：存在一条路径p，能从u走到u'，那么u'从u可达的 $u \overset{p}{\rightsquigarrow} u'$

(4) 简单路径：所有顶点都不能重复出现（不包含环）

4. 环/回路：$p=\langle v_0, v_1, \cdots v_k \rangle$ 如果满足 $v_0=v_k$ 那么这条路径就是一个环.

- 无向图中，一个简单环至少要有3条边.
- 有向图中，一个简单环至少可以只有1条边 ⇒ self-loop 自环

5. 子图.

① 子图：$G'=(V', E')$ 是 $G=(V, E)$ 的子图当且仅当 $V' \subseteq V$ 且 $E' \subseteq E$

② spanning sub-graph 生成子图：$G'=(V', E')$ 满足 $V'=V$, $E' \subseteq E$

生成子图必须保留原图中的所有顶点，但可以只保留一部分边

③ vertex induced subgraph 顶点诱导子图 $G'=(V', E')$ 满足 $V' \subseteq V$ $E'=\{(u,v) \in E : u, v \in V'\}$

先从原图中选出一些顶点V'，然后把这些顶点之间在原图中本来存在的所有边都保留下来

6. 连通性

① 无向图：一个无向图是连通的，当且仅当任意两个顶点之间都有一条路径.

1° 连通分量：最大的连通子图

2° 任意一个连通无向图 $G=(V, E)$ 满足 $|E| \geq |V|-1$ n个顶点，让所有点连成一个整体，至少需要n-1条边

② 有向图：一个有向图是强连通的，当且仅当任意两个顶点都可以互相到达.
强连通分量是有向图中一个最大的强连通子图

7. 特殊类图.

① complete graph 完全图　无向图+任意两个不同顶点之间都有边

② bipartite graph 二分图　顶点集合可以分成两个部分 V_1 和 V_2，并且每条边都只能连接两个不同部分顶点 ⇒ 边只能跨集合连接 $(u,v) \in E$ $u \in V_1$ $v \in V_2$ 或者 $u \in V_2$ $v \in V_1$

③ tree 树 是一个 connected acyclic undirected. 连通、无环、无向图　$|E|=|V|-1$

④ forest 森林　无环、无向图（可由多棵树组成）.

8. 简单图 vs 多重图.

简单图：无重边和自环

多重图：允许出现重边和自环
(multi-graph)

ch24 单源最短路径 (邻接链表)

一、Dijkstra's algorithm

问题定义：

给定一个带权重的有向图 $G=(V,E)$，其中每条边的长度 $w_e \geq 0$ 并给定源点 $s \in V$

要求：从源点 s 到图中每一个节点 $u \in V$ 的最短路径问题

Suppose that you change the length of every edge of G as follows. For which is every shortest path in G a shortest path in G'?

A. Add 17.

B. ✓ Multiply by 17.

乘以正数不会改变路径长度的相对顺序，加上固定数会受到路径边数的影响，可能改变最短路径

（一条路径经过的边数越多，被加次数就越多）。

解决：①带权重的②有向图上③单源最短路径问题 + 要求所有边的权重都为非负值。

1. 思想：Greedy approach 贪心思想

算法维护一个结点集合 S（已经确定最短路径长度的点的集合）

如果一个点 u 已经在 S 中，则 $d[u]$ 即为从源点 s 到 u 的最终最短距离

step：

① 初始化 $S \leftarrow \{s\}$ $d[s] \leftarrow 0$ 其他 j 顶点 v $d[v] \leftarrow \infty$ ⇒ 从 s 到 u 的最短距离 + 从 u 到 v 的边长

② 重复选择还没有被探索的点 $v \notin S$ 使得 $d(v) = \min_{e=(u,v):u\in S} \{d[u]+w_e\}$

将 v 加入 S 中，令 $d[v] \leftarrow d'[v]$

③ 为了恢复最短路径，把 $d'[v]$ 设置为使最小值成立的那个节点 u

[从 s 到 v 的最短路径中，v 的前一个节点是谁] ⇒ 前驱节点

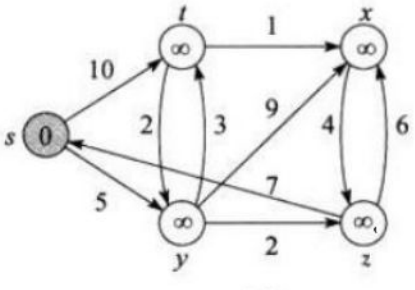
Basic step. Choose unexplored node $u \notin S$ with minimum $d[u]$.

- Add u to S.
- For each edge $e=(u,v)$ leaving u, if $d[v] > d[u]+w_e$ then: （松弛操作）
  - $d[v] \leftarrow d[u]+w_e$ $s \rightsquigarrow v$ 最短路径长度
  - $\pi[v] \leftarrow u$ 在 $s \rightsquigarrow v$ 最短路径上 v 的前驱结点

运行时间。

Dijkstra 算法在运行过程中维持的关键信息是一组结点集合 S。从源结点 s 到该集合中每个结点之间的最短路径已经被找到。算法重复从结点集 $V-S$ 中选择最短路径估计最小的结点 u，将 u 加入到集合 S，然后对所有从 u 发出的边进行松弛。在下面给出的实现方式中，我们使用一个最小优先队列 Q 来保存结点集合，每个结点的关键值为其 d 值。

2. 示例



v	d	π
s	0	NIL
t	∞	NIL
x	∞	NIL
y	∞	NIL
z	∞	NIL

s →

v	d	π
s	0	NIL
t	10	s
x	∞	NIL
y	5	s
z	∞	NIL

y →

v	d	π
s	0	NIL
t	8	y
x	14	y
y	5	s
z	7	y

z →

v	d	π
s	0	NIL
t	8	y
x	13	z
y	5	s
z	7	y

S = {s, y, z}

t →

v	d	π
s	0	NIL
t	8	y
x	9	z
y	5	s
z	7	y

S = {s, y, z, t}

x →

v	d	π
s	0	NIL
t	8	y
x	9	z
y	5	s
z	7	y

S = {s, y, z, t, x}

S = {s}

S = {s, y}

3. 证明 Dijkstra 正确

核心：每次把一个点加入集合S时，这个点的d[u]已经是真正的最短路径长度，以后不会结论再变小

采用归纳法（对|S|进行归纳，其中|S|表示当前集合S中的点数）

① Base case |S|=1

此时 S={s} d[s]=0 显然是最短距离

② Inductive hypothesis 归纳假设

假设当前集合里的所有点的d[u]已经是正确的最短距离

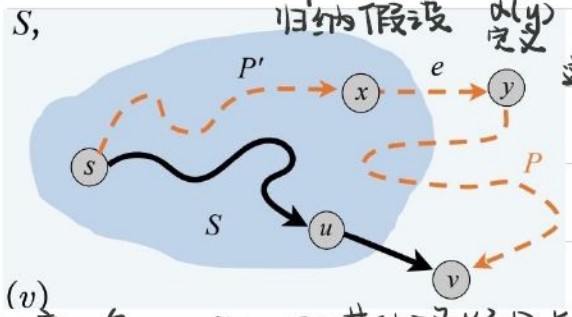
令v为下一个加入到S中的结点，(u,v)为到达v的最后一条边

根据算法 d[v]=α[v]=d[u]+w(u,v) 现在我们要证明没有其他路径比α(v)更短.

考虑任意另一条从s到v的路径P，e(x,y)为路径P中第一条离开集合S的边且P'是从s到x的子路径

$w(P) \geq w(P') + w_e \geq d[x] + w_e \geq \alpha(y) \geq \alpha(v)$

↑ 归纳假设　↑ d(y)定义　↑ Dijkstra贪心策略这一步算法选择v而不是y.



(v)

∴任意一条从s到v的其他路径P，长度都不会小于α(v)

1. 要证明 Dijkstra 正确，就要证明“当前选中的 v 不会被以后更新得更短”。
2. 假设存在另一条到 v 的路径 P。
3. 因为 $s \in S$，而 $v \notin S$，所以 P 一定要从 S 走出去。
4. 找到它第一次走出 S 的边 (x, y)。
5. 利用 $x \in S$：可以用归纳假设，知道 $d[x]$ 是真的最短距离。
6. 利用 $y \notin S$：可以和 Dijkstra 当前选择的 v 比较。
7. 由于 Dijkstra 选的是最小的 $\alpha(v)$，所以这条绕路在刚离开 S 时就已经不可能更短。
8. 边权非负，后面也不可能变短。

4. 时间复杂度

① 初始化 $O(|V|)$ { d[v], s[v], π[v]

② 探索结点 for node 1 to node |V| { 找到当前最小的 d[u] $O(|V|)$；更新和 u 相邻的点。$(u,v)\in E$，更新 d[v]，$O(\deg(u))$

∴ total:

$$O(|V|)+\sum_{i=1}^{|V|}(O(|V|)+O(\deg(u)))=O(|V|)+O(|V|^2)+O(|E|)=O(|V|^2)$$

5. 优化：用优先队列实现 Dijkstra 算法

使用一个最小优先队列来选择一个尚未探索的节点 u，使得 d[u] 最小

一旦结点 u 从优先队列中删除，d[u] 等于 s~u 的最短路径。

```
DIJKSTRA (V, E, w, s)
FOREACH v ≠ s :  α[v] ← ∞,  π[v] ← null ,  α[s] ← 0.        初始化.
Create an empty priority queue PQ.
FOREACH v ∈ V : INSERT(PQ, v, α[v]).   ⇒ 对于图中的每个节点 v，把它插入优先队列中，并用 d[v] 作为它的优先级
WHILE (IS-NOT-EMPTY(PQ))
    u ← EXTRACT-MIN(PQ).          // put u into S
    FOREACH edge e = (u, v) leaving u:
        IF (α[v] > α[u] + we)
            DECREASE-KEY(PQ, v, α[u] + we).   降低节点 v 在优先队列中的 key 值.
                                              key 越小越优先.
            α[v] ← α[u] + we;
            π[v] ← u.
```

用优先队列：使用 heap 高效完成两个操作 { 插入一个元素；找到最大/最小元素。

操作	时间复杂度
$Insert(x)$: 插入元素 x	$O(\log n)$
$Decrease/Increase\text{-}key(i, key)$: 减小或增大第 i 个元素的 key	$O(\log n)$
$Extract\text{-}Min/Max()$: 取出最小值或最大值，并把这个值从堆中删除，恢复堆结构	$O(\log n)$

因为堆本质上是一棵 **完全二叉树**。

一棵有 n 个节点的完全二叉树，高度大约是：

$$\log n$$

无论是 **sift-up** 还是 **sift-down**，最多只会从叶子走到根，或者从根走到叶子。

所以最多走树高那么多层，也就是：

$$O(\log n)$$

时间复杂度分析：

① 初始化 1° $O(|V|)$ { d[v], s[v], π[v]

2° 把所有 v 顶点插入优先队列 每次 $O(\log|V|)$，共 $|V|$ 个，因此 $O(|V|\log|V|)$

② 探索结点 for node 1 to node |V| { 找到当前最小的 d[u] $O(\log|V|)$；更新和 u 相邻的点。$(u,v)\in E$，更新 d[v]，同时在优先队列中修改 v 的 key. $O(\log|V|\cdot\deg(u))$

③ total

$$O(|V|\log|V|)+\sum_{i=1}^{|V|}(O(\log|V|)+O(\log|V|\cdot\deg(u)))$$

$$=O(|V|\log|V|)+O(|V|\log|V|)+O(|E|\log|V|)=O((|V|+|E|)\log|V|)$$

连通图 $|E|\geq|V|-1$

$$=O(|E|\log|V|)$$

6. 说明：

稠密图中：数组最优 $O(n^2)$

稀疏图中：二叉堆/优先队列最优

二、Bellman-Ford algorithm

1. 问题定义：给定一个有向图 $G=(V,E)$ 其中每条边都有任意长度 $l(u,v)$，再给定一个源点 s，要求找到从 s 到图中每个结点 $v\in V$ 的最短路径

解决一般情况下的单源最短路径，边的权重可为负值。

值。给定带权重的有向图 $G=(V, E)$ 和权重函数 $w: E\to\mathbf{R}$，Bellman-Ford 算法返回一个布尔值（负环）以表明是否存在一个从源结点可以到达的权重为负值的环路。如果存在这样一个环路，算法将告诉我们不存在解决方案。如果没有这种环路存在，算法将给出最短路径和它们的权重。

lemma 1：如果某一条从 s 到 v 的路径中包含一个负环，那么从 s 到 v 不存在最短路径.

proof:

证明：如果存在这样一个负环 W，那么我们可以在经过 W 时绕这个环任意多次，从而构造出一条长度任意小的 $s\sim v$ 路径。

简单路径最多经过 n 个顶点，所以最多有 n-1 条边

lemma 2：如果图中没有负环，那么最短 $s\sim v$ 的路径可以是简单路径且最多只有 n-1 条边

proof:（只要图中没有负环，那么在所有最短路径里面，至少可以找到一条不包含环的最短路径）



① 在所有 $s\sim v$ 的最短路径中，选择一条使用边数最少的路径.

② 如果路径 P 包含一个有向环，那么可以把路径 P 对应于 W 的那一段删除，而且不会增加路径长度

因此我们得到一条更短路径与假设矛盾

∴路径 P 不包含环的.

假设路径 P 长这样：

$$s\to\cdots\to a\to W\to a\to\cdots\to v$$

也就是说，从 s 走到某个点 a，然后绕了一个环 W，又回到 a，再继续走到 v。

既然绕完环以后又回到同一个点 a，那么这个环对“从 s 到 v”没有必要。

删掉以后仍然可以得到一条从 s 到 v 的路径：

$$s\to\cdots\to a\to\cdots\to v$$

综上我们将问题研究分为两个部分：

如果图中可以有负权边，那么单源最短路径问题必须假设没有负环，否则最短路径可能不存在；因此除了求最短路径之外还需研究如何检测负环

第一类：单源最短路径问题

条件是：

允许负权边，但不允许负环

目标是：

求从 s 到所有点的最短路径

第二类：负环检测问题

条件是：

图中可能存在负环

目标是：

找到一个负环，或者判断不存在负环

2. 分析

(1) 有向无环图的单源最短路径 → 动态规划

Def: OPT(i, v) = 从源点s到结点v，在最多使用i条边的限制下，能够得到的最短路径长度。

Goal: OPT(n-1, v) for each v

Bellman equation

$$OPT(i,v)=\begin{cases}0 & \text{if } i=0 \quad v=s \ ①\\ \infty & \text{if } i=0 \quad v\neq s \ ②\\ OPT(i-1,v) & \text{if } i>0 \text{ and there exist no } (u,v)\in E \text{ or } v=s \ ③\\ \min\{OPT(i-1,v),\ \min_{(u,v)\in E}\{OPT(i-1,u)+\ell_{uv}\}\} & \text{if } i>0 \text{ 且 there exist } (u,v)\in E \text{ 且 } v\neq s \ ④\end{cases}$$

含u=v

SHORTEST-PATHS(V, E, ℓ, s)

FOREACH node $v \in V$:
  $M[0, v] \leftarrow \infty$. ②
$M[0, s] \leftarrow 0$. ①
FOR $i = 1$ TO $n - 1$
  FOREACH node $v \in V$:
    $M[i, v] \leftarrow M[i-1, v]$. ③
    FOREACH edge $(u, v) \in E$:
      $M[i, v] \leftarrow \min\{M[i, v], M[i-1, u] + \ell_{uv}\}$. ④

M[i,v]: 从源点s → 顶点v最多使用i条边时最短路径长度

① 时间复杂度

如果G=(V,E)中无负环，那么DP算法可以求出从源点s到每个顶点v的最短路径长度，时间复杂度 $\Theta(mn)$（→O(VE)），空间复杂度 $\Theta(n^2)$（→维护M[i,v]）

外层循环：

$$i = 1, 2, \ldots, n-1$$

一共执行：

$$n-1$$

轮，也就是 $\Theta(n)$ 轮。

每一轮需要检查所有边一次，时间是：

$$\Theta(m)$$

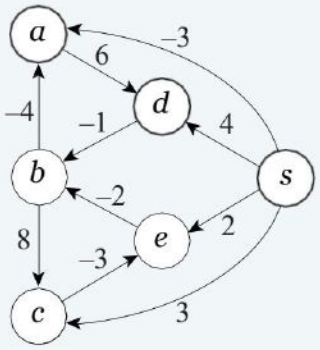
所以总时间是：

$$\Theta(n) \times \Theta(m) = \Theta(mn)$$

② 构建最短路径. 入[i,v]

前面的 DP 算法求出了最短路径长度 $M[i, v]$。如果还想恢复具体路径，可以在计算时记录每个状态的前驱节点；也可以在计算完成后，根据 $M[i, v] = M[i-1, u] + \ell_{uv}$ 找出哪些边属于最短路径，再沿这些边构造最短路径。

③示意 初始化后 最多5轮



v	d	π
s	0	NIL
a	∞	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL
e	∞	NIL

i=0

v	d	π
s	0	NIL
a	-3	s
b	∞	NIL
c	3	s
d	4	s
e	2	s

i=1

v	d	π
s	0	NIL
a	-3	s
b	0	e
c	3	s
d	3	a
e	0	c

i=2

续

v	d	π
s	0	NIL
a	-4	b
b	-2	e
c	3	s
d	3	a
e	0	c

i=3

v	d	π
s	0	NIL
a	-6	b
b	-2	e
c	3	s
d	2	a
e	0	c

i=4

v	d	π
s	0	NIL
a	-6	b
b	-2	e
c	3	s
d	0	a
e	0	c

i=5

④优化→空间上的优化：维持1维数组

d[v]：目前为止，我们已经找到的从源点s到顶点v的最短路径长度。

BELLMAN–FORD(V, E, ℓ, s)

FOREACH node $v \in V$:

  $d[v] \leftarrow \infty$.

$d[s] \leftarrow 0$.

FOR $i = 1$ TO $n - 1$

  FOREACH node $v \in V$:

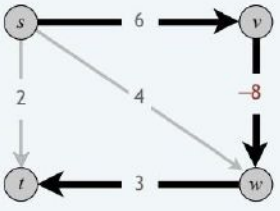
    FOREACH edge $(u, v) \in E$:

      IF ($d[v] > d[u] + \ell_{uv}$)

        $d[v] \leftarrow d[u] + \ell_{uv}$.

pass i
$O(m)$ time

原地更新



正确性证明

lemma3：对每个点v，d[v]不会变大，只会变小或不变（单调不增）

lemma4：第i轮后，d[v]不大于任何最多i条边路径的长度。 s~v

proof：归纳法证明

Base case i=0 此时只有一种路径 s~s

初始化d[s]=0

其他点d[v]=∞ 对于其他点没有0条路径能走到

显然成立。

Inductive hypothesis 归纳假设

① 假设第 i-1 轮结束后，结论已经成立

② 令 P 为任意为 s⇝v 最多使用 i 条边的路径，(u,v) 是路径 P 中最后一条边

P' 为 s⇝u 的路径

③ 由于 P' 是一条从 s⇝u 最多使用 i-1 条边的路径，

∴ d[u] ≤ l(P')

在第 i 轮中，我们考虑边 (u,v)

$$\begin{aligned} d[v] &\le d[u] + \ell_{uv} \\ &\le l(P') + \ell_{uv} \\ &= l(P) \end{aligned}$$

如果某条最短路径实际包含 k 条边，那么算法最多在第 k 轮之后找到它。

(2) 检测负环

lemma 5 如果对于每个结点 v，都有 OPT(n,v) = OPT(n-1,v) 那么图中不存在负环 (当且仅当关系)

如果图中没有负环，那么对于每个结点 v，都有 OPT(n,v) = OPT(n-1,v)

proof:

$$OPT(n+1,v) = \begin{cases} OPT(n,v) & \text{if } i>0, \text{ and there exist no } (u,v)\in E \text{ or } v=s \\ \min\limits_{(u,v)\in E}\{OPT(n,u)+\ell_{uv}\} & \text{if } i>0, \text{ there exist } (u,v)\in E \text{ and } v\neq s \end{cases}$$

$$= \begin{cases} OPT(n-1,v) & \text{if } i>0, \text{ and there exist no } (u,v)\in E \text{ or } v=s \\ \min\limits_{(u,v)\in E}\{OPT(n-1,u)+\ell_{uv}\} & \text{if } i>0, \text{ there exist } (u,v)\in E \text{ and } v\neq s \end{cases}$$

$$= OPT(n,v) \quad = OPT(n-1,v)$$

Similarly,

$$OPT(n+2,v) = \begin{cases} OPT(n+1,v) & \text{if } i>0, \text{ and there exist no } (u,v)\in E \text{ or } v \\ \min\limits_{(u,v)\in E}\{OPT(n+1,u)+\ell_{uv}\} & \text{if } i>0, \text{ there exist } (u,v)\in E \text{ and } v\neq s \end{cases}$$

$$= \begin{cases} OPT(n-1,v) & \text{if } i>0, \text{ and there exist no } (u,v)\in E \text{ or } v=s \\ \min\limits_{(u,v)\in E}\{OPT(n-1,u)+\ell_{uv}\} & \text{if } i>0, \text{ there exist } (u,v)\in E \text{ and } v\neq s \end{cases}$$

$$= OPT(n,v) = OPT(n-1,v)$$

事实上，对于任意 n' > n，都有 OPT(n',v) = OPT(n-1,v)

也就是所有 OPT(n,v) 的值已经收敛了 ⇒ s到v的最短路径存在的 ⇒ 不存在负环

lemma 6 如果存在某个结点 v，使得 OPT(n,v) < OPT(n-1,v)，那么任意一条最多使用 n 条边的最短 s⇝v 的路径都包含一个环 W，并且这个环是负环

一些直观认识：

正常情况下，如果没有负环，最短路径一定可以是简单路径。

如果图中有 n 个节点，那么简单路径最多只有：

$$n-1$$

条边。

所以理论上：

$$OPT(n-1,v)$$

理论证明：反证法

∵ OPT(n,v) < OPT(n-1,v)

∴ 任意一条最多使用 n 条边的最短 s⇝v 路径 P 实际上必须恰好有 n 条边

否则 OPT(n,v) = OPT(n-1,v)

就已经应该是最终最短距离了。

如果到了：

$$OPT(n, v)$$

还能更小，也就是：

$$OPT(n, v) < OPT(n-1, v)$$

这说明：

> 最优路径必须用了第 n 条边。

但一条有 n 条边的路径会经过 $n+1$ 个节点。图中只有 n 个节点，所以必然有节点重复，因此必然有环。

根据鸽巢原理，路径P中一定包含一个重复节点x.

图中有 n 个节点。

如果一条路径有 n 条边，那么它会经过：

$$n+1$$

个节点。

但是总共只有 n 个节点。

所以 $n+1$ 个位置放进 n 个节点里，至少有一个节点重复出现。

这就是鸽巢原理。

一旦节点重复，就说明路径中出现了一个环。

对于反证法深入理解.

我想证明一个结论 A 是对的。
我先假设它是错的，也就是假设"不是 A"成立。
然后从这个"不是 A"出发，结合题目原本给定的条件，推导出一个矛盾。
既然"不是 A"会导致矛盾，那说明"不是 A"不可能成立，所以 A 必须成立。

令W是路径P的任意一个环

如果W不是负环，那么删除W后可以得到一条s~v的路径，这条路径最多n-1条边，且总权重不会变大

∴ OPT(n,v) ≤ OPT(n-1,v)

与题设矛盾

∴ W为负环

(3) 完整算法的伪代码

```
BELLMAN–FORD(V, E, ℓ, s)
FOREACH node v ∈ V:
    d[v] ← ∞.
    predecessor[v] ← null.
d[s] ← 0.
FOR i = 1 TO n − 1                                   ┐
    FOREACH node v ∈ V:                              │
        FOREACH edge (u, v) ∈ E:                     │
            IF (d[v] > d[u] + ℓ_uv)                  │ pass i
                d[v] ← d[u] + ℓ_uv.                  │ O(m) time
                predecessor[v] ← u.                  │
FOREACH edge (u, v) ∈ E:                             ┘
    IF (d[v] > d[u] + ℓ_uv)
        RETURN TRUE.
RETURN FALSE.
```

θ(mn)时间

在Bellman-Ford算法正常做完n-1轮松弛后，再额外检查一遍所有边，如果还有某条边可让距离更小，存在负环

3. 为什么负环上一定至少有一条边可以松弛?

假设负环是：

$$v_1 \to v_2 \to v_3 \to \cdots \to v_k \to v_1$$

边权分别是：

$$\ell_{12}, \ell_{23}, \ldots, \ell_{k1}$$

因为它是负环，所以：

$$\ell_{12} + \ell_{23} + \cdots + \ell_{k1} < 0$$

现在反过来想。

如果负环上**没有任何一条边可以松弛**，那就说明每条边都满足：

$$d[v_2] \le d[v_1] + \ell_{12}$$
$$d[v_3] \le d[v_2] + \ell_{23}$$
$$\cdots$$
$$d[v_1] \le d[v_k] + \ell_{k1}$$

把这些不等式全部加起来，左边是：

$$d[v_2] + d[v_3] + \cdots + d[v_1]$$

右边是：

$$d[v_1] + d[v_2] + \cdots + d[v_k] + (\ell_{12} + \ell_{23} + \cdots + \ell_{k1})$$

左右两边的 $d[\cdot]$ 会抵消，最后得到：

$$0 \le \ell_{12} + \ell_{23} + \cdots + \ell_{k1}$$

也就是说：

$$0 \le \text{环的总权重}$$

这说明环的总权重非负。

可是我们一开始说它是负环：

$$\text{环的总权重} < 0$$

矛盾。

所以假设不成立。

ch 25 All-Pairs Shortest Paths 所有结点对的最短路径问题

邻接矩阵

问题定义：

给定一个带权重的有向图 $G=(V,E)$，其权重函数 $w: E \to R$，该函数将边映射到实数值的权重上。对所有的结点对 $u, v \in V$，找到一条从结点 u 到结点 v 的最短路径。（不考虑包含负环情况）

case that there are no neg

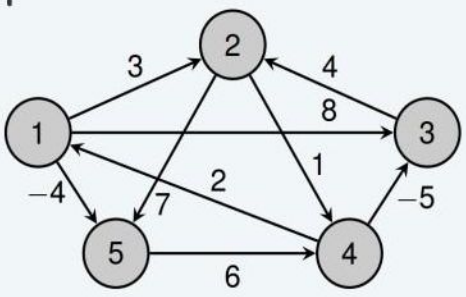
All-pairs Shortest-paths Problem:

Input: An $n \times n$ matrix W representing the edge weights of an n-vertex directed graph $G = (W, E)$. That is, $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{the weight of directed edge } (i,j) & \text{if } i \neq j \text{ and } (i,j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E \end{cases}$$

Output: An $n \times n$ matrix $D = (d_{ij})$, where d_{ij} denotes the shortest-path weight from vertex i to vertex j.

Input:



$$W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

如何理解？

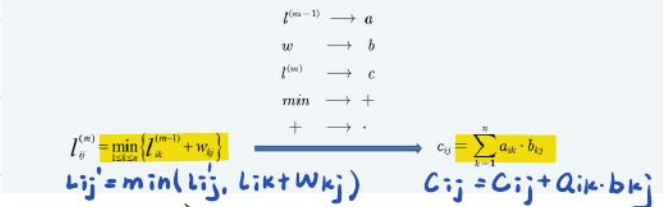
一、基于矩阵乘法的动态规划算法

1. 基本出版

Def. $L_{ij}^{(m)}$ 从 $i \rightsquigarrow j$ 最多使用 m 条边的最短路径长度

Goal. 求每一对节点 i, j 的 $L_{ij}^{(n-1)}$

如果图中没有负环，那么最短路径一定可以取为简单路径.



(1) 分析

① 最优子结构 一条最短路径的所有子路径都是最短路径.

短路径的最优子结构性质。

引理 24.1（最短路径的子路径也是最短路径） 给定带权重的有向图 $G=(V, E)$ 和权重函数 $w: E \to \mathbf{R}$。设 $p=\langle v_0, v_1, \cdots, v_k \rangle$ 为从结点 v_0 到结点 v_k 的一条最短路径，并且对于任意的 i 和 j，$0 \leqslant i \leqslant j \leqslant k$，设 $p_{ij}=\langle v_i, v_{i+1}, \cdots, v_j \rangle$ 为路径 p 中从结点 v_i 到结点 v_j 的子路径。那么 p_{ij} 是从结点 v_i 到结点 v_j 的一条最短路径。

证明 如果将路径 p 分解为 $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$，则有 $w(p)=w(p_{0i})+w(p_{ij})+w(p_{jk})$。现

proof: 反证法

证明 如果将路径 p 分解为 $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$，则有 $w(p)=w(p_{0i})+w(p_{ij})+w(p_{jk})$。现在，假设存在一条从 v_i 到 v_j 的路径 p'_{ij}，且 $w(p'_{ij})<w(p_{ij})$。则 $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ 是一条从结点 v_0 到结点 v_k 的权重为 $w(p_{0i})+w(p'_{ij})+w(p_{jk})$ 的路径，而该权重小于 $w(p)$。这与 p 是从 v_0 到 v_k 的一条最短路径这一假设相矛盾。 ■

② Bellman equation

$l_{ij}^{(m)}$ 为从结点 i 到结点 j 的至多包含 m 条边的任意路径中的最小权重

$$l_{ij}^{(m)} = \begin{cases} 0 & \text{if } m=0 \text{ 且 } i=j \\ \infty & \text{if } m=0 \text{ 且 } i\neq j \\ \min\limits_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\} & \text{if } m>0 \end{cases}$$ 通过对 j 的所有可能前驱 k 进行检查.

$$l_{ij}^{(m)} = \min\{l_{ij}^{(m-1)}, \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}\} = \min_{1\le k\le n}\{l_{ik}^{(m-1)} + w_{kj}\}$$

($l_{ij}^{(m-1)}$ 中 $k=j$.)

③ 自底向上算法实现

$L^{(m)} = (l_{ij}^{(m)})$ $m = 1, 2, \cdots, n-1$

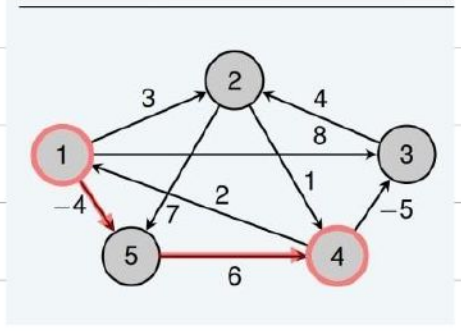
SLOW-APSP

```
ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows                    图中顶点个数
2  L^(1) = W                     最多只允许走1条边时，最短路径矩阵就是W
3  for m = 2 to n − 1
4      let L^(m) be a new n × n matrix      创建新矩阵 L^(m) → 用来保存最多走m条边时，任意两点之间的最短距离
5      L^(m) = EXTEND-SHORTEST-PATHS(L^(m−1), W)
6  return L^(n−1)
```

extends the shortest paths computed so far by one more edge

```
EXTEND-SHORTEST-PATHS(L, W)        n×n
1  n = L.rows                      顶点个数
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞               初始化
6          for k = 1 to n          枚举最后一条边的起点 k.
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

④ 示例



$L^{(1)} = W =$ 第1点作为横坐标为相加点. 其中横 i 纵 j. 看前一个 求还看 W

$$\begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$m=2$ ↓

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$l_{11}^{(2)} = \min\{0, 3+\infty, 8+\infty, \infty+2, -4+\infty\} = 0$

$l_{12}^{(2)} = \min\{0+3, 3+0, 8+4, \infty+\infty, -4+\infty\} = 3$

$l_{13}^{(2)} = \min\{0+8, 3+\infty, 8+0, \infty+(-4), -4+\infty\} = 8$

$m=3$ →

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$m=4$ ↓

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$t_{14}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\} = 2$

$t_{15}^{(2)} = \min\{0+(-4), 3+7, 8+\infty, \infty+\infty, -4+0\} = -4$

$t_{21}^{(2)} = \min\{\infty+0, 0+\infty, \infty+\infty, 1+2, 7+\infty\} = 3$

$t_{22}^{(2)} = \min\{\infty+3, 0+0, \infty+4, 1+\infty, 7+\infty\} = 0$

$t_{23}^{(2)} = \min\{\infty+8, 0+\infty, \infty+0, 1+(-5), 7+\infty\} = -4$

$t_{24}^{(2)} = \min\{\infty+\infty, 0+1, \infty+\infty, 1+0, 7+6\} = 1$

$t_{25}^{(2)} = \min\{\infty+(-4), 0+7, \infty+\infty, 1+\infty, 7+0\} = 7$

$t_{31}^{(2)} = \min\{\infty+0, 4+\infty, 0+\infty, \infty+2, \infty+\infty\} = \infty$

$t_{32}^{(2)} = \min\{\infty+3, 4+0, 0+4, \infty+\infty, \infty+\infty\} = 4$

$t_{33}^{(2)} = \min\{\infty+8, 4+\infty, 0+0, \infty+(-5), \infty+\infty\} = 0$

$t_{34}^{(2)} = \min\{\infty+\infty, 4+1, 0+\infty, \infty+0, \infty+6\} = 5$

$t_{35}^{(2)} = \min\{\infty+(-4), 4+7, 0+\infty, \infty+\infty, \infty+0\} = 4$

$t_{11}^{(3)} = \{0+0, 3+\infty, 8+\infty, 2+2, -4+\infty\} = 0$

$t_{12}^{(3)} = \{0+3, 3+0, 8+4, 2+\infty, -4+\infty\} = 3$

$t_{21}^{(3)} = \{3+0, 0+\infty, -4+\infty, 1+2, 7+\infty\} = 3$

$t_{22}^{(3)} = \{3+3, 0+0, -4+4, 1+\infty, 7+\infty\} = 0$

$t_{31}^{(3)} = \{\infty+0, 4+\infty, 0+\infty, 5+2, 11+\infty\} = 7$

$t_{32}^{(3)} = \{\infty+3, 4+0, 0+4, 5+\infty, 11+\infty\} = 4$

$t_{11}^{(4)} = \{0+0, 3+\infty, -3+\infty, 2+2, -4+\infty\} = 0$

$t_{35}^{(4)} = \{7+(-4), 4+7, 0+\infty, 5+\infty, 11+0\} = 3$

根据输入矩阵 $W=(w_{ij})$，现在可以计算出矩阵序列 $L^{(1)}, L^{(2)}, \cdots, L^{(n-1)}$，其中对于 $m=1, 2, \cdots, n-1$，有 $L^{(m)}=(l_{ij}^{(m)})$。最后的矩阵 $L^{(n-1)}$ 包含的是最短路径的实际权重。注意，对于所有的结点 i 和 j，$L^{(1)}=(w_{ij})$，因此，$L^{(1)}=W$。

(3) 时间复杂度分析

```
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

$L^{(1)} = W$ → 乘法法则：$l_{ij}' = \min_k(l_{ik} + w_{kj})$

$L^{(2)} = L^{(1)} \cdot W = W^2$

$L^{(3)} = L^{(2)} \cdot W = W^3$

$O(n^3)$

⋮

$L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$

原算法从 $L^{(1)} \to L^{(n-1)}$

总时间复杂度 $O(n) \times O(n^3) = O(n^4)$

↓改进

改进算法的运行时间

我们的目标并不是要计算所有的 $L^{(m)}$ 矩阵，我们感兴趣的仅仅是矩阵 $L^{(n-1)}$。回忆本书前面的内容可知，在没有权重为负值的环路的情况下，式(25.3)意味着对于所有的 $m \geqslant n-1$，我们有 $L^{(m)}=L^{(n-1)}$。正如传统的矩阵乘法是相关的，由 EXTEND-SHORTEST-PATHS 过程所定义的矩阵乘法也是相关的(请参阅练习 25.1-4)。因此，可以仅用 $\lceil \lg(n-1) \rceil$ 个矩阵乘积来计算矩阵 $L^{(n-1)}$。计算的方法如下：

$$L^{(1)} = W$$
$$L^{(2)} = W^2 = W \cdot W$$
$$L^{(4)} = W^4 = W^2 \cdot W^2$$
$$L^{(8)} = W^8 = W^4 \cdot W^4$$
⋮ → 至少需要翻倍多少次，才能允许的边数达到或超过 n-1
$$L^{(2^{\lceil \lg(n-1) \rceil})} = W^{2^{\lceil \lg(n-1) \rceil}} = W^{2^{\lceil \lg(n-1) \rceil}-1} \cdot W^{2^{\lceil \lg(n-1) \rceil}-1}$$

由于 $2^{\lceil \lg(n-1) \rceil} \geqslant n-1$，最后的乘积 $L^{(2^{\lceil \lg(n-1) \rceil})}$ 等于 $L^{(n-1)}$。

下面的过程使用**重复平方**技术来计算上述矩阵序列。

FAST-APSP

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  m = 1
4  while m < n-1
5      let L^(2m) be a new n×n matrix
6      L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
7      m = 2m
8  return L^(m)
```

加速.

时间复杂度 $O(n^3 \lg n)$

示例：

$$L^{(1)} = W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \xrightarrow{m=2} L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

↓ $m=4$

$l_{11}^{(4)} = \min\{0+0, 3+3, 8+\infty, 2+2, -4+8\} = 0$

$l_{12}^{(4)} = \min\{0+3, 3+0, 8+4, 2+(-1), -4+\infty\} = 1$

$l_{13}^{(4)} = \min\{0+8, 3+(-4), 8+0, 2+(-5), -4+\infty\} = -3$

$t_{35}^{(4)} = \min\{\infty-4, 4+7, 0+11, 5+(-2), 11+0\} = 3$

$$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

二、Floyd-Warshall算法（问题定义同上）按点迭代

1)分析

①最优子结构

不再限制路径最多有几条边

而是限制路径中间允许经过哪些点。

Floyd-Warshall算法考虑的是一条最短路径上的中间结点

$P=\langle v_1, v_2, \cdots v_l\rangle$ 上除 v_1 和 v_l 之外的任意结点

$v_3, \cdots, v_{l-1}\}$ 中的结点。

Floyd-Warshall 算法依赖于下面的观察。假定图 G 的所有结点为 $V=\{1, 2, \cdots, n\}$，考虑其中的一个子集 $\{1, 2, \cdots, k\}$，这里 k 是某个小于 n 的整数。对于任意结点对 $i, j \in V$，考虑从结点 i 到结点 j 的所有中间结点均取自集合 $\{1, 2, \cdots, k\}$ 的路径，并且设 p 为其中权重最小的路径（路径 p 是简单路径）。Floyd-Warshall 算法利用了路径 p 和从 i 到 j 之间中间结点均取自集合 $\{1, 2, \cdots, k-1\}$ 的最短路径之间的关系。该关系依赖于结点 k 是否是路径 p 上的一个中间结点。

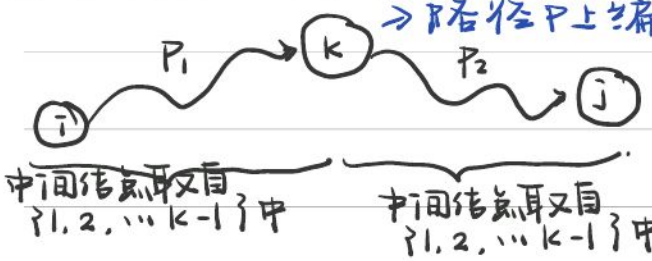
①若结点k不是路径P上的中间结点

则路径P上的所有中间结点都属于集合{1,2,…k-1}

∴从i⇝j的中间结点取自集合{1,2,…,k-1}的一条最短路径也是结点i⇝j取自集合{1,2,…k}的一条最短路径.

②若结点k是路径P上的中间结点，则将路径分解为 $i \overset{P_1}{\rightsquigarrow} k \overset{P_2}{\rightsquigarrow} j$

→路径P上编号最大的中间结点



中间结点取自{1,2,…k-1}中

中间结点取自{1,2,…k-1}中

②Bellman equation

Def.

设 d_{ij} = 从 i⇝j 的所有中间结点全部取自集合{1,2,…k}的一条最短路径的权重

Goal. $D^{(n)}=(d_{ij}^{(n)})$

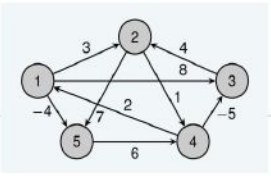
$$d_{ij}^{(k)}=\begin{cases} w_{ij} & \text{若 } k=0 \text{ 不包含中间结点} \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right) & \text{若 } k\geq 1 \end{cases}$$

③自底向上计算最短路径权重.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n        不断在可用集合中加点
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

填矩阵. → 时间复杂度 $O(n^3)$

(2) 示例.



可以简单画图梳理.

前驱结点(当前找到的最短路径倒数第2个)　　结点

$$D^{(0)}=\begin{pmatrix}0&3&8&\infty&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&\infty&\infty\\ 2&\infty&-5&0&\infty\\ \infty&\infty&\infty&6&0\end{pmatrix}\qquad \Pi^{(0)}=\begin{pmatrix}NIL&1&1&NIL&1\\ NIL&NIL&NIL&2&2\\ NIL&3&NIL&NIL&NIL\\ 4&NIL&4&NIL&NIL\\ NIL&NIL&NIL&5&NIL\end{pmatrix}.$$

$$D^{(1)}=\begin{pmatrix}0&3&8&\infty&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&\infty&\infty\\ 2&5&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\qquad \Pi^{(1)}=\begin{pmatrix}NIL&1&1&NIL&1\\ NIL&NIL&NIL&2&2\\ NIL&3&NIL&NIL&NIL\\ 4&1&4&NIL&1\\ NIL&NIL&NIL&5&NIL\end{pmatrix}$$

$d_{12}^{(1)}=\min\{d_{12}^{(0)},\ d_{11}^{(0)}+d_{12}^{(0)}\}=\min\{3,\ 0+3\}=3$

$d_{13}^{(1)}=\min\{d_{13}^{(0)},\ d_{11}^{(0)}+d_{13}^{(0)}\}=8$

$d_{21}^{(1)}=\min\{d_{21}^{(0)},\ d_{21}^{(0)}+d_{11}^{(0)}\}=\infty$

$d_{22}^{(1)}=\min\{d_{21}^{(0)},\ d_{21}^{(0)}+d_{12}^{(0)}\}=0$

$d_{23}^{(1)}=\min\{d_{23}^{(0)},\ d_{21}^{(0)}+d_{13}^{(0)}\}=\min\{\infty,\ \infty+8\}=\infty$

$d_{24}^{(1)}=\min\{d_{24}^{(0)},\ d_{21}^{(0)}+d_{14}^{(0)}\}=\min\{1,\ \infty+\infty\}=1$

$d_{25}^{(1)}=\min\{d_{25}^{(0)},\ d_{21}^{(0)}+d_{15}^{(0)}\}=7$

$d_{41}^{(1)}=\min\{d_{41}^{(0)},\ d_{41}^{(0)}+d_{11}^{(0)}\}=2$

$d_{42}^{(1)}=\min\{d_{42}^{(0)},\ d_{41}^{(0)}+d_{12}^{(0)}\}=\min\{\infty,\ 2+3\}=5$

$d_{45}^{(1)}=\min\{d_{45}^{(0)},\ d_{41}^{(0)}+d_{15}^{(0)}\}=\min\{\infty,\ 2+(-4)\}=-2$

$$D^{(2)}=\begin{pmatrix}0&3&8&4&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&5&11\\ 2&5&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\qquad \Pi^{(2)}=\begin{pmatrix}NIL&1&1&2&1\\ NIL&NIL&NIL&2&2\\ NIL&3&NIL&2&2\\ 4&1&4&NIL&1\\ NIL&NIL&NIL&5&NIL\end{pmatrix}$$

$d_{14}^{(2)}=\min\{d_{14}^{(1)},\ d_{12}^{(1)}+d_{24}^{(1)}\}=\min\{\infty,\ 3+1\}=4$

$$D^{(3)}=\begin{pmatrix}0&3&8&4&-4\\ \infty&0&\infty&1&7\\ \infty&4&0&5&11\\ 2&-1&-5&0&-2\\ \infty&\infty&\infty&6&0\end{pmatrix}\qquad \Pi^{(3)}=\begin{pmatrix}NIL&1&1&2&1\\ NIL&NIL&NIL&2&2\\ NIL&3&NIL&2&2\\ 4&3&4&NIL&1\\ NIL&NIL&NIL&5&NIL\end{pmatrix}$$

$$\pi_{ij}^{(k)}=\begin{cases}\pi_{ij}^{(k-1)} & \text{若 } d_{ij}^{(k-1)}\le d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\\ \pi_{kj}^{(k-1)} & \text{若 } d_{ij}^{(k-1)}>d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases}$$

未更新

$$D^{(4)}=\begin{pmatrix}0&3&-1&4&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\qquad \Pi^{(4)}=\begin{pmatrix}NIL&1&4&2&1\\ 4&NIL&4&2&1\\ 4&3&NIL&2&1\\ 4&3&4&NIL&1\\ 4&3&4&5&NIL\end{pmatrix}$$

$d_{52}^{(4)}=\min\{d_{52}^{(3)},\ d_{54}^{(3)}+d_{42}^{(3)}\}=\min\{\infty,\ 6+(-1)\}=5$

$d_{53}^{(4)}=\min\{d_{53}^{(3)},\ d_{54}^{(3)}+d_{43}^{(3)}\}=\min\{\infty,\ 6+(-5)\}=1$

$5\to3\to4\to3$

$$D^{(5)}=\begin{pmatrix}0&1&-3&2&-4\\ 3&0&-4&1&-1\\ 7&4&0&5&3\\ 2&-1&-5&0&-2\\ 8&5&1&6&0\end{pmatrix}\qquad \Pi^{(5)}=\begin{pmatrix}NIL&3&4&5&1\\ 4&NIL&4&2&1\\ 4&3&NIL&2&1\\ 4&3&4&NIL&1\\ 4&3&4&5&NIL\end{pmatrix}$$

固定.

(3) 空间优化版本

```
FLOYD-WARSHALL'(W, n)
1  D = W
2  for k = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              d_ij = min{d_ij, d_ik + d_kj}
6  return D
```

只是把原有上标去掉 → 原地更新

proof: 原地更新不会破坏正确性

case 1: when i=k or j=k

$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} = d_{ij}^{(k-1)}$

第k轮中，第k行和第k列不会改变

更新 d_{ij} 需要用到 d_{ik}, d_{kj}

case 2: when i, j ≠ k

$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$ → 更新 d_{ij} 后面其他元素更新不会再依赖旧值 $d_{ij}^{(k-1)}$

只会用到 d_{ik} 和 d_{kj}.

∴ 可以只用一个矩阵 D 原地更新.

空间复杂度: $O(n^2)$

(4) 有向图的传递闭包

有向图的传递闭包

给定有向图 $G=(V, E)$，结点集合为 $V=\{1, 2, \cdots, n\}$，我们希望判断对于所有的结点对 i 和 j，图 G 是否包含一条从结点 i 到结点 j 的路径。我们定义图 G 的**传递闭包**为图 $G^*=(V, E^*)$，其中 $E^*=\{(i, j)$：如果图 G 中包含一条从结点 i 到结点 j 的路径$\}$。

法一：

一种时间复杂度为 $\Theta(n^3)$ 的计算图 G 的传递闭包的办法是给 E 中的每条边赋予权重 1，然后运行 Floyd-Warshall 算法。如果存在一条从结点 i 到结点 j 的路径，则有 $d_{ij}<n$；否则，$d_{ij}=\infty$。

法二：

还有另一种类似的办法，其时间复杂度也是 $\Theta(n^3)$，但在实际场景中能够节省时间和空间。该办法以逻辑或操作（∨）和逻辑与操作（∧）来替换 Floyd-Warshall 算法中的算术操作 min 和 +。对于 $i, j, k=1, 2, \cdots, n$，我们定义：如果图 G 中存在一条从结点 i 到结点 j 的所有中间结点都取自集合 $\{1, 2, \cdots, k\}$ 的路径，则 $t_{ij}^{(k)}$ 为 1；否则，$t_{ij}^{(k)}$ 为 0。我们构建传递闭包 $G^*=(V, E^*)$ 的方法为：将边 (i, j) 置于集合 E^* 当且仅当 $t_{ij}^{(n)}$ 为 1。一种与公式(25.5)相似的 $t_{ij}^{(k)}$ 递归定义如下：

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{若 } i \neq j \text{ 且 } (i,j) \notin E \\ 1 & \text{若 } i = j \text{ 或 } (i,j) \in E \end{cases}$$

对于 $k \geqslant 1$，

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) \qquad (25.8)$$

不经过k / 可达性上一轮可求 / i可达k / 图中直接有边 / k可达j / 经过k

如 Floyd-Warshall 算法一样，我们以 k 递增的次序来计算矩阵 $T^{(k)}=(t_{ij}^{(k)})$。

$t_{12}^{(1)} = t_{11}^{(0)} \wedge t_{12}^{(0)} = 0$

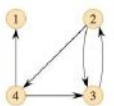
$t_{21}^{(1)} = t_{21}^{(0)} \wedge t_{11}^{(0)} = 0$

TRANSITIVE-CLOSURE(G)

Time complexity. $O(n^3)$

```
TRANSITIVE-CLOSURE(G, n)
1  let T^(0) = (t_ij^(0)) be a new n × n matrix
2  for i = 1 to n
3      for j = 1 to n
4          if i == j or (i, j) ∈ G.E
5              t_ij^(0) = 1
6          else t_ij^(0) = 0
7  for k = 1 to n
8      let T^(k) = (t_ij^(k)) be a new n × n matrix
9      for i = 1 to n
10         for j = 1 to n
11             t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
12 return T^(n)
```

初始化



$T^{(0)} = \begin{pmatrix} 1&0&0&0 \\ 0&1&1&1 \\ 0&1&1&0 \\ 1&0&1&1 \end{pmatrix}$ $T^{(1)} = \begin{pmatrix} 1&0&0&0 \\ 0&1&1&1 \\ 0&1&1&0 \\ 1&0&1&1 \end{pmatrix}$

$t_{42}^{(1)} = t_{41}^{(0)} \wedge t_{12}^{(0)}$

$t_{21}^{(2)} = t_{22}^{(1)} \wedge t_{21}^{(1)}$

$t_{31}^{(2)} = t_{32}^{(1)} \wedge t_{21}^{(1)}$

$t_{34}^{(2)} = t_{32}^{(1)} \wedge t_{24}^{(1)} = 1$

$T^{(2)} = \begin{pmatrix} 1&0&0&0 \\ 0&1&1&1 \\ 0&1&1&1 \\ 1&0&1&1 \end{pmatrix}$ $T^{(3)} = \begin{pmatrix} 1&0&0&0 \\ 0&1&1&1 \\ 0&1&1&1 \\ 1&1&1&1 \end{pmatrix}$

$t_{21}^{(3)} = t_{23}^{(2)} \wedge t_{31}^{(2)}$
$t_{12}^{(3)} = t_{13}^{(2)} \wedge t_{23}^{(2)}$
$t_{31}^{(3)} = t_{33}^{(2)} \wedge t_{31}^{(2)}$

$t_{42}^{(3)} = t_{43}^{(2)} \wedge t_{32}^{(2)} = 1 \wedge 1 = 1$
$t_{12}^{(4)} = t_{14}^{(3)} \wedge t_{42}^{(3)}$
$t_{21}^{(4)} = t_{24}^{(3)} \wedge t_{41}^{(3)}$
$t_{31}^{(4)} = t_{34}^{(3)} \wedge t_{41}^{(3)}$

$T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$

ch 26 最大流

1. 基本概念

(1) 流网络 flow network

$G = (V, E, s, t, c)$

其中：它是一个有向图 (V, E)，包含一个源点 $s \in V$ 和汇点 $t \in V$

对于每一条边 $e \in E$，都有一个容量 $c(e) \geq 0$

G 中的流是一个实值函数 $f: V \times V \to R$.
（输入是一对顶点 (u, v) 输出是一个实数）

满足：

① 容量限制：对于所有 $e \in E$，$0 \leq f(e) \leq c(e)$

② 流量守恒：对所有结点 $u \in V - \{s, t\}$：$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$

(2) 流 f 的值 $|f|$

$|f| = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e)$ ⇒ 从源结点流出的总流量减去流入源结点总流量.

(3) 最大流问题 Max-flow problem

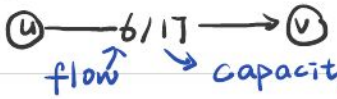
给定一个流网络 G，一个源结点 s，一个汇点 t，我们希望找到值最大的流.

2. Max-flow problem ⇒ Ford-Fulkerson 方法

greedy algorithm fail × ⟶ need some mechanism to undo a bad decision

(1) Residual network 残差网络/残存网络.

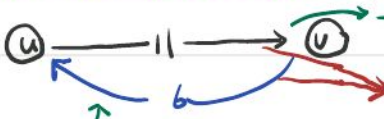
Orignal flow network G



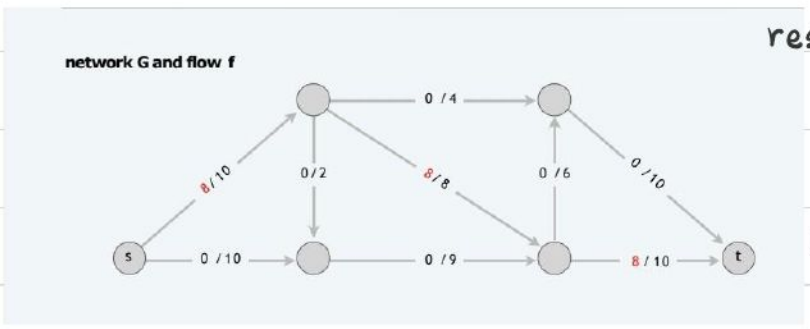
$G_f = (V, E_f, s, t, c_f)$

$E_f = \{e : f(e) < c(e)\} \cup \{e^{reverse} : f(e) > 0\}$

residual network G_f.

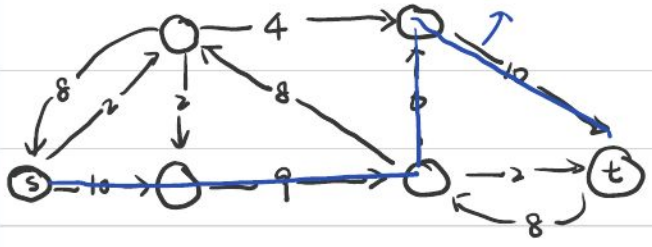


⇒ 可撤回上限.



residual network

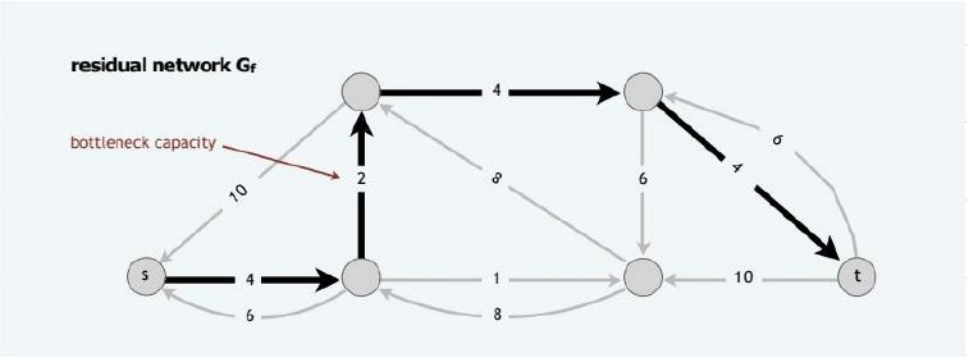
Augmenting path.



(2) Augmenting path t增广路径.

① Def. 在残差网络Gf中，从源点s到汇点t的一条简单路径.

Def. bottleneck capacity: t增广路径P中所有边的残存容量最小值.



② 关键性质

设f是当前的一个流，P是残存网络中一条t增广路径.

AUGMENT(f, c, P)

δ ← bottleneck capacity of augmenting path P.

FOREACH edge e ∈ P:

IF (e ∈ E) f(e) ← f(e) + δ. e为原网络中的正向边

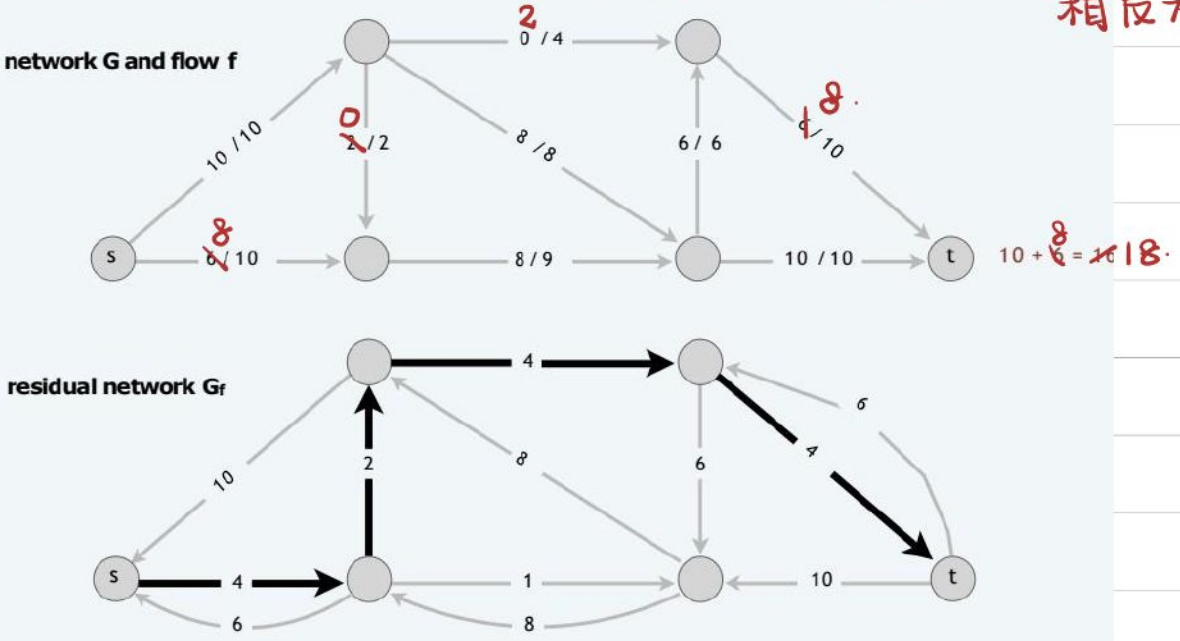
ELSEIF (e ∉ E ∧ e ∈ E^r) f(e) ← f(e) − δ. e为残差网络中的反向边.

RETURN f.

沿着t增广路径P增加流量之后，总流量会增加这条路径的瓶颈容量.

eg:

残存网络和原网络边相同方向+
相反方向−



13) ford-fulkerson 增广路径算法

① 对于每一条边 $e \in E$, 初始令 $f(e)=0$

② 在残存网络 G_f 中任意找到一条 $s \rightsquigarrow t$ 的路径 P

③ 沿着路径 P 增广流量 { 找到 P 上的 bottleneck capacity δ ; 对路径上正向边增加 δ ; 对路径上反向边减少 δ } > 相对 residual network.

④ 重复上述步骤直到无法继续

FORD–FULKERSON(G)

FOREACH edge $e \in E : f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an $s \rightsquigarrow t$ path P in G_f)

$f \leftarrow$ AUGMENT(f, c, P).

Update G_f.

RETURN f.

AUGMENT(f, c, P)

$\delta \leftarrow$ bottleneck capacity of augmenting path P.

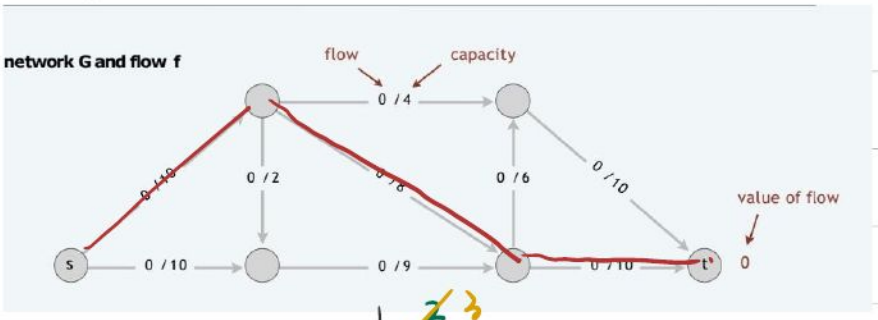
FOREACH edge $e \in P$:

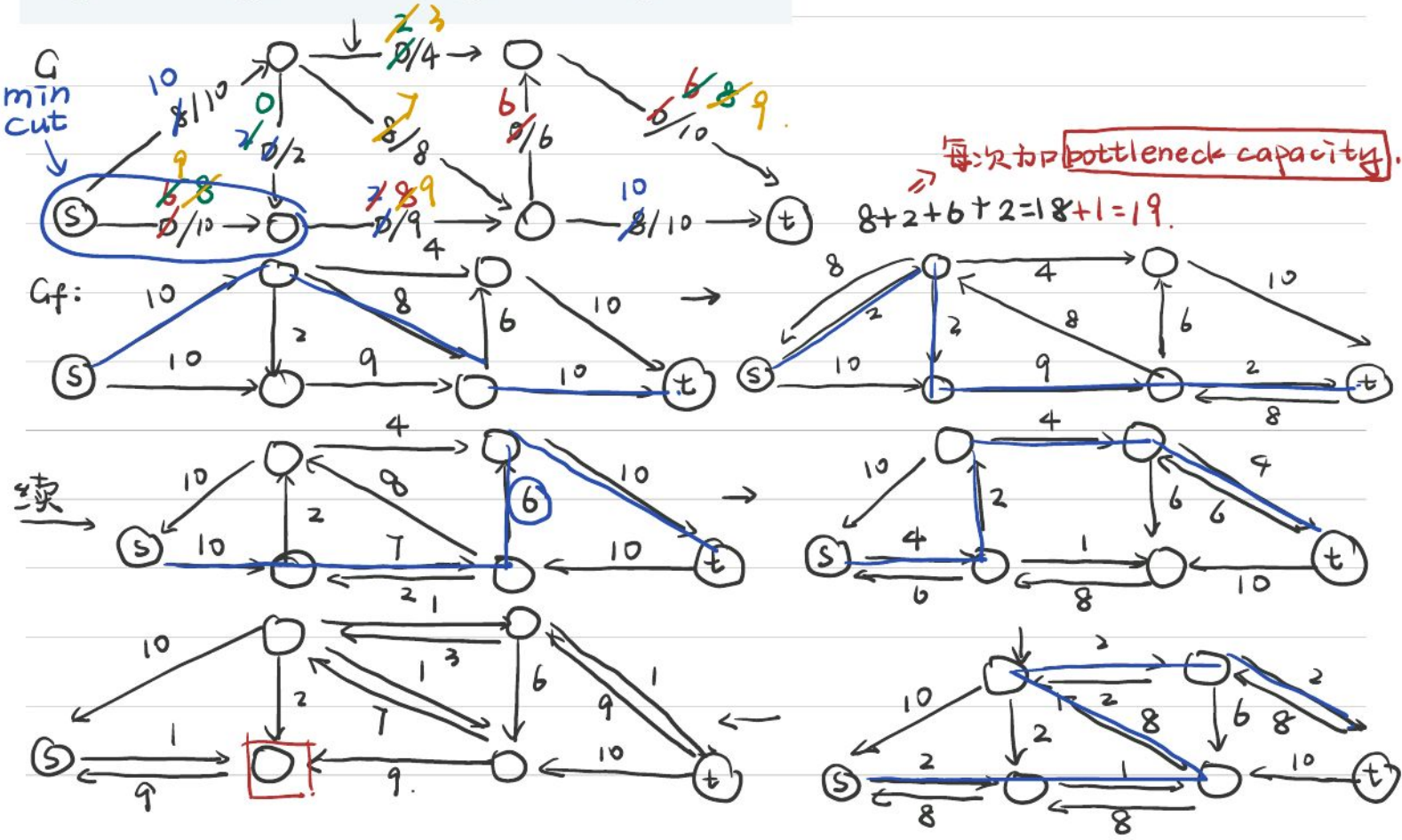
IF ($e \in E$) $f(e) \leftarrow f(e) + \delta$.

ELSEIF ($e \notin E \wedge e \in E_f$) $f(e) \leftarrow f(e) - \delta$.

RETURN f.

示例.

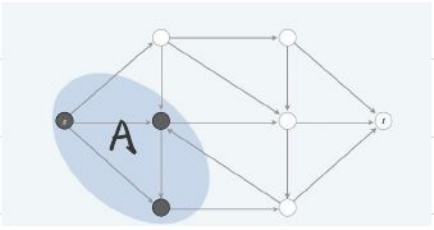




3. Max-flow Min-cut Theorem 最大流最小割定理

基本定义

(1) 割 (cut): 一个割是对顶点集合V的一个划分 (A, B)，其中源点 s 在集合A中，汇点 t 在集合B中.



★ A→B，不包含B→A!!

(2) 割的容量：所有从集合A指向集合B的边的容量之和

$$cap(A,B) = \sum_{e \text{ out of } A} c(e)$$

(3) Min-cut problem：求一个容量最小的割

流量与割的关系

①

Flow value lemma：设 f 是任意流，(A,B) 任意一个割，那么流 f 的总流量等于流过割 (A,B) 的净流量

⇒ 流从源点到汇点的总流量等于集合A指向集合B净流量.

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

$A \to B$　　$B \to A$

proof:

$$|f| = val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ into } s} f(e)$$

$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \right)$$

→ 除s外其余作点，满足流量守恒

A内部的边会被抵消 →

$$= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

把集合 A 中所有点的净流出量加起来时，除了源点 s 贡献 $val(f)$，其他点因为流量守恒贡献 0；同时 A 内部的边一正一负相互抵消，最后只剩下从 A 到 B 的流量减去从 B 到 A 的流量。

eg: (u,v) → 对u 流出边 / 对v 流入边 > A内部边抵消

② weak duality 弱对偶性

f 为任意流，(A,B) 为任意割，那么 $val(f) \le cap(A,B)$

任何流的值不会超过任意割的容量

proof

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \le \sum_{e \text{ out of } A} f(e) \le \sum_{e \text{ out of } A} c(e) = cap(A,B)$$

③ 等于论：

- 如果 **某个流值刚好等于某个割的容量**：

$$val(f) = cap(A,B)$$

那么：

1. f 是 **最大流** (max flow)
2. (A,B) 是 **最小割** (min cut)

proof.

对于任意流 f'

$val(f') \le cap(A,B) = val(f)$

对于任意割 (A',B')

$cap(A',B') \ge val(f) = cap(A,B)$

↓

④ 最大流-最小割定理

→ strong duality.

Value of a max flow = capacity of a min cut

定理 26.6(最大流最小切割定理) 设 f 为流网络 $G=(V, E)$ 中的一个流，该流网络的源结点为 s，汇点为 t，则下面的条件是等价的：

1. f 是 G 的一个最大流。
2. 残存网络 G_f 不包括任何增广路径。
3. $|f|=c(S, T)$，其中 (S, T) 是流网络 G 的某个切割。

c(A,B) (A,B)

1→2 反证法

假设关于当前流f仍存在一条增广路径(Gf中.仍有一条从s→t的路径)

那么我们可以沿着这条路径继续发送流量.从而增加f

从与f是G的一个最大流矛盾

2→3

令f是一个没有增广路径的流

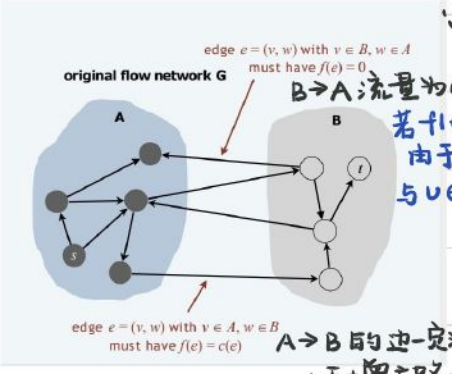
A为在残存网络 Gf中.从源点s能够到达的所有顶点组成的集合.

从根据A定义 s∈A

从f没有增广路径 从t∉A

定义 B=V\A 易知(A,B)为一个割

s∈A.t∈B



B→A流量为0

若f(e)>0 Gf中有w→v

由于w∈A ⇒ s可以到达w $\xrightarrow{w\to v}$ s可达v ⇒ v∈A

与v∈B矛盾

A→B的边一定满流

(无增广路径)

若不满流f(e)<c(e) → 残存网络会存在边 $v\to w$

由于s可达v, v可达w ⇒ s可达w ⇒ w∈A

与w∈B矛盾

从 $val(f)=\underset{e \text{ out of } A}{\sum} f(e)-\underset{e \text{ into } A}{\sum} f(e)=\underset{e \text{ out of } A}{\sum} c(e)-0=cap(A,B)$

A→B满流　B→A流量为0

3→1 弱对偶性的推论.

整理证明步骤：

1. **假设**
   - f 是一个流，并且在残差网络 G_f 中没有增广路径。
   - 定义集合 A 为 **从源点 s 出发在残差网络中可达的所有节点。**

$$A=\{v \in V \mid v \text{ 在 } G_f \text{ 中可从 } s \text{ 达到}\}$$

2. **根据集合定义**
   - $s \in A$（因为从源点可以到自身）。
   - $t \notin A$（因为如果 $t \in A$，则存在一条增广路径，从而与假设矛盾）。
3. **分析割 (A, B)**
   - 定义 $B=V \setminus A$，那么 (A, B) 就是一个割，且 $s \in A$，$t \in B$。
   - 对割 (A, B) 上的边分类：
     1. 从 A 到 B 的边 (v, w)，此时 $v \in A, w \in B$：
        - 因为 $w \notin A$（不可达），在残差网络中 **没有容量** 可以增加流，即

$$f(e)=c(e)$$

     2. 从 B 到 A 的边 (v, w)，此时 $v \in B, w \in A$：
        - 因为 $v \notin A$，流不能从 v "返回"到 A，所以

$$f(e)=0$$

4. **计算流的值**
   - 根据 **流值引理 (Flow Value Lemma)**：

$$val(f)=\sum_{e \text{ out of } A} f(e)-\sum_{e \text{ into } A} f(e)$$

   - 对于割 (A, B)：

$$\sum_{e \text{ out of } A} f(e)=\sum_{e \text{ out of } A} c(e), \quad \sum_{e \text{ into } A} f(e)=0$$

   - 因此：

$$val(f)=\sum_{e \text{ out of } A} c(e)=cap(A, B)$$

4. Ford-Fulkerson 算法的进一步分析

(1) 局限性　整数容量假设 ⟶ 当边容量为无理数时，Ford-Fulkerson 可能无法保证终止/无法收敛到最大流

因为 **Ford-Fulkerson 每次增广多少流量，取决于你选的增广路径的瓶颈容量。**如果容量是整数，每次至少增加 1；但如果容量是无理数，瓶颈可能越来越小，小到无限接近 0，于是算法可能一直"有进步"，但永远走不到最大流。

(2) 整数又不变量

在 Ford-Fulkerson 算法运行过程中，每条边流量 $f(e)$ 和残差容量 $c_f(e)$ 都保持为整数

- **证明方法**：用增广路径数量进行归纳法证明：
  1. 初始每条边流量为 0，是整数。
  2. 每次沿增广路径增加流量，增加量是残量容量 $c_f(e)$，也是整数。
  3. 因此每次增广后流量仍为整数。

(3) 增广路径定理：Ford-Fulkerson 算法在最多 $val(f^*)$ 条增广路径后终止，其中 f^* 是最大流

proof. 每次增广至少增加1单位流量，因此总流量最多增加到最大流量 $val(f^*)$

(4) **整数最大流定理 (Integrality theorem)**

结论：存在整数最大流 f^*。

解释：因为算法终止且每次增广的流量是整数，最终得到的最大流也是整数。

推论/分析

Ford-Fulkerson 算法的时间复杂度 $O(mf^*)$ ⇒ 边数

① 每次增广路径可用 BFS/DFS 在 $O(m)$ 时间内找到

② 最多增广 f^* 次，每次 $O(m)$ ⇒ $O(mf^*)$

5. 最大二分匹配问题

(1) 匹配 Matching.

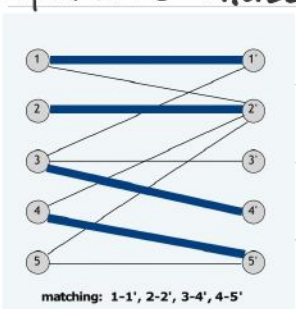
Def. 给定一个无向图 $G=(V,E)$，一个匹配是边的子集 $M \subseteq E$，使得对于所有结点 $v \in V$，子集 M 中最多有1条边与结点 v 相连

(每个点最多只出现在 M 中的一条边)

Max Matching：给定一个图 G，找到边数最多的匹配

Def. 二分图：所以结点可以被划分成两个子集 L 和 R，并且每一条边都连接 L 中的一个结点和 R 中的一个结点 (不相交)

Bipartite Matching：给定一个二分图 $G=(L \cup R, E)$，找到最大基数(max-cardinality)匹配 → 边集合 E 中所有边横跨 L 和 R.



找到一个无向二分图 $G=(V,E)$ 的最大匹配：

(1) 建模分析 → 构建一个流网络（点集、边集、s、t、c容量）

① 创建一个有向图 $G'=(L\cup R\cup\{s,t\},E')$

$G=(V,E)$的最大匹配。解决这一问题的关键技巧是构建一个流网络，其中的流对应于匹配，如图 26-8(c)所示。我们将二分图 G 所对应的流网络 $G'=(V',E')$定义如下：设源结点 s 和汇点 t 为不属于结点集 V 的新结点，并设 $V'=V\cup\{s,t\}$。如果图 G 的结点集划分为 $V=L\cup R$，则 E 中从 L 指向 R 的边都是流网络 G'的边。此外，G'中的边还包括如下的 $|V|$ 条新有向边：

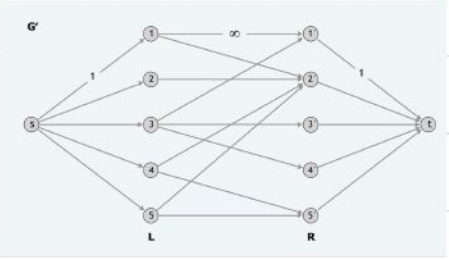
$$E'=\{(s,u):u\in L\}\cup\{(u,v):(u,v)\in E\}\cup\{(v,t):v\in R\}$$

要完成流网络的构建，需要给 E'中的每条边赋予单位容量。由于结点集 V 中的每个结点至少有一条相连的边，$|E|\geqslant|V|/2$。因此，$|E|\leqslant|E'|=|E|+|V|\leqslant 3|E|$，所以 $|E'|=\Theta(E)$。

② 将二分图所有边定向为 L→R，并把这些边的容量设置为无限大

③ 从源点 s 向 L 中的每个结点添加容量为 1 的边

④ 从 R 中每个结点向汇点 t 添加容量为 1 的边



Theorem. Max cardinality of a matching in G = value of max flow in G'.

最大二分匹配　　最大流

proof.

① ≤ 最大匹配基数 K ≤ 网络 G' 最大流

给定一个基数为 K 的最大匹配

流 f 在 G' 中的每一条对应匹配的边发送 1 单位流（每条匹配对应 s→L→R→t 的流量）

则此流 f 的总流量为 K

在这个网络中，边 $s\to u$ 和 $v\to t$ 的容量都是 1，因此它们的流量只能是 0 或 1。

同时，由于每个左侧点 $u\in L$ 从源点 s 接收的流量最多为 1，所以从 u 流向右侧点的总流量也最多为 1。因此中间边 $u\to v$ 上的流量也只能是 0 或 1。

② ≥ 最大匹配基数 K ≥ 网络 G' 最大流

Step 1:

令 f 是网络 G' 中最大流，总流量为 K 即 val(f) = K

Step 2:

根据流整数性原理 ⇒ K 是整数，且每条边流量 f(e) = 0 或 1

Step 3:

构造集合 M：由所有在最大流中从 L→R f(e)=1 的边组成

$M=\{(u,v)\in E\mid u\in L,v\in R,f(u,v)=1\}$

Step 4:

为原图 $(L\cup\{s\},R\cup\{t\})$

由 flow value lemma 可知 $|M|=K$

Flow value lemma：设 f 是任意流，(A,B) 任意一个割 ⇒ 流从源点到汇点的总流量等于集合 A 指向集合 B 净流量

$$val(f)=\sum_{e\ out\ of\ A} f(e)-\sum_{e\ into\ A} f(e)$$

A→B　　B→A

proof:

$$|f|=val(f)=\sum_{e\ out\ of\ s} f(e)-\sum_{e\ into\ s} f(e)$$

A 内部的边会被抵消 $=\sum_{v\in A}\left(\sum_{e\ out\ of\ v} f(e)-\sum_{e\ into\ v} f(e)\right)$ → 除源外其余结点满足流量守恒

$$=\sum_{e\ out\ of\ A} f(e)-\sum_{e\ into\ A} f(e)$$

Step 5:

接下来证明 M 中没有两条边共享同一个端点。

对于任意左侧点 $u\in L$，边 $s\to u$ 的容量为 1，所以 u 最多只能接收 1 单位流。

由流量守恒可知，u 向右侧点流出的总流量最多也是 1。因此，u 最多只能参与一条流量为 1 的边。

同理，对于任意右侧点 $v\in R$，边 $v\to t$ 的容量为 1，所以 v 最多只能向 t 发送 1 单位流。因此，v 最多也只能参与一条流量为 1 的边。

所以，集合 M 中每个点最多只出现在一条边里，因此 M 是一个匹配。

（从最大流出发，把所有流量为 1 的 L→R 取出来作为匹配）

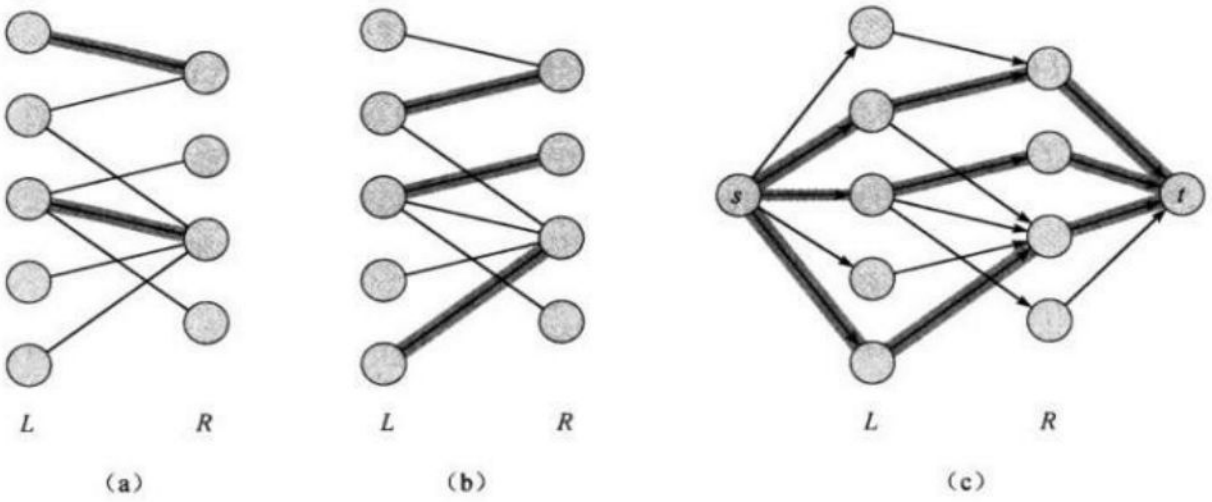


图 26-8 一个二分图 $G=(V, E)$，结点集划分为 $V=L\cup R$。(a)基数为 2 的匹配，由覆盖阴影的边所表示。(b)基数为 3 的最大匹配。(c)对应的流网络 G'，图中显示的是最大流。每条边的容量为单位容量。覆盖阴影的边的流量为 1，其他所有的边没有流量。从子集 L 指向子集 R 的覆盖阴影的边对应的是图(b)中最大匹配所用到的边

在二分图中寻找最大匹配问题有着许多的实际应用。例如，把一个机器集合 L 和要同时执行的任务集合 R 相匹配。E 中有边 (u, v) 就说明一台特定的机器 $u\in L$ 能够完成一项特定的任务 $v\in R$。最大匹配能够让尽可能多的机器运行起来。

寻找最大二分匹配

使用 Ford-Fulkerson 方法可以在关于 $|V|$ 和 $|E|$ 的多项式时间内，找出无向二分图 $G=(V, E)$ 的最大匹配。解决这一问题的关键技巧是构建一个流网络，其中的流对应于匹配，如图 26-8(c)所示。我们将二分图 G 所对应的流网络 $G'=(V', E')$ 定义如下：设源结点 s 和汇点 t 为不属于结点集 V 的新结点，并设 $V'=V\cup\{s, t\}$。如果图 G 的结点集划分为 $V=L\cup R$，则 E 中从 L 指向 R 的边都是流网络 G' 的边。此外，G' 中的边还包括如下的 $|V|$ 条新有向边：

$$E'=\{(s,u):u\in L\}\cup\{(u,v):(u,v)\in E\}\cup\{(v,t):v\in R\}$$

要完成流网络的构建，需要给 E' 中的每条边赋予单位容量。由于结点集 V 中的每个结点至少有一条相连的边，$|E|\geqslant|V|/2$。因此，$|E|\leqslant|E'|=|E|+|V|\leqslant 3|E|$，所以 $|E'|=\Theta(E)$。

下面的引理证明了图 G 中的一个匹配直接对应 G 所对应的流网络 G' 中的一个流。对于流网络 $G=(V, E)$ 中的一个流 f 来说，如果对于所有的边 $(u, v)\in V\times V$，$f(u, v)$ 都是整数值，则称流 f 是**整数值的**。

引理 26.9 设 $G=(V, E)$ 为一个二分图，其结点划分为 $V=L\cup R$，设 $G'=(V', E')$ 是图 G 所对应的流网络。如果 M 是 G 中的一个匹配，则流网络 G' 中存在一个整数值的流 f，使得 $|f|=|M|$。相反，如果 f 是 G' 中的一个整数值的流，则 G 中存在一个匹配 M，使得 $|M|=|f|$。

证明 首先证明图 G 中的一个匹配 M 对应流网络 G' 中一个整数值的流 f。定义流 f 如下：如果边 $(u, v)\in M$，则 $f(s, u)=f(u, v)=f(v, t)=1$。对于所有其他属于 E' 的边 (u, v)，定义 $f(u, v)=0$。读者可以很容易地验证这样所定义的流 f 满足容量限制和流量守恒性质。

从直观上看，每条边 $(u, v)\in M$ 对应流网络 G' 中流经路径 $s\to u\to v\to t$ 的一个单位的流。而且，除了源结点 s 和汇点 t 之外，由子集 M 中的边所诱导的路径都是结点不相交的[⊖]。横跨切割 $(L\cup\{s\}, R\cup\{t\})$ 的净流量等于 $|M|$。因此，根据引理 26.4，流 f 的值为 $|f|=|M|$。

要证明反向的论断，设 f 为 G' 中的一个整数值的流，并设

732
~
733

⊖ 除 s 和 t 之外，两条不同的路径中不存在相同的结点。——译者注

$$M=\{(u,v):u\in L,v\in R,\text{并且 } f(u,v)>0\}$$

每个结点 $u\in L$ 只有一条进入的边，即(s,u)，其容量为 1。因此，每个结点 $u\in L$ 至多有 1 个单位的流进入，而如果有 1 个单位的流进入，根据流量守恒性质，离开该结点的流也必须有 1 个单位。此外，由于 f 是整数值的流，对于每个结点 $u\in L$，流入的这 1 个单位的流只能最多从一条边进入，也只能最多从一条边流出。因此，1 个单位的流进入结点 u 当且仅当恰好存在一个结点 $v\in R$，使得 $f(u,v)=1$，并且在离开每个结点 $u\in L$ 的边中至多有一条出边带有正值的流。对每个结点 $v\in R$ 也有一个对称的结论。因此，集合 M 是一个匹配。

要证明 $|M|=|f|$，只要注意到对于每个匹配的结点 $u\in L$，有 $f(s,u)=1$，并且对于每条边 $(u,v)\in E-M$，有 $f(u,v)=0$。因此，横跨切割$(L\cup\{s\},R\cup\{t\})$的净流量 $f(L\cup\{s\},R\cup\{t\})$等于 $|M|$。因此，根据引理 26.4，流的值为 $|f|=|M|$。■

基于引理 26.9，我们希望得出结论，二分图 G 中的一个最大匹配对应于流网络 G' 中的一个最大流，并且，因此可以通过在对应流网络 G' 上运行一个最大流算法来计算图 G 中的最大匹配。这一推理过程中存在的唯一障碍是最大流算法可能返回流网络 G' 中一个非整数的流 $f(u,v)$，即使流的值 $|f|$ 本身必须是整数。不过，下面的定理将说明，如果使用 Ford-Fulkerson 方法，则这个问题不会发生。

定理 26.10(完整性定理) *如果容量函数 c 只能取整数值，则 Ford-Fulkerson 方法所生成的最大流 f 满足 $|f|$ 是整数值的性质。而且，对于所有的结点 u 和 v，$f(u,v)$ 的值都是整数。*

734 **证明** 通过对迭代次数进行归纳来进行证明，具体证明留作练习 26.3-2。■

下面来证明引理 26.9 的一个推论。

推论 26.11 *二分图 G 中的一个最大匹配 M 的基数等于其对应的流网络 G' 中某一最大流 f 的值。*

证明 下面的证明使用引理 26.9 中的术语。假定 M 是图 G 中的一个最大匹配，且其相应的流网络 G' 中的流 f 不是最大流。那么 G' 中存在一个最大流 f'，满足 $|f'|>|f|$。由于 G' 的容量都是整数值，因此，根据定理 26.10，可以假设 f' 是整数值。因此，f' 对应 G 中的一个匹配 M'，且其基数为 $|M'|=|f'|>|f|=|M|$，这与 M 是最大匹配这一假设相矛盾。用类似的方法可以证明：如果 f 是 G' 中的一个最大流，则其对应的匹配是 G 的一个最大匹配。■

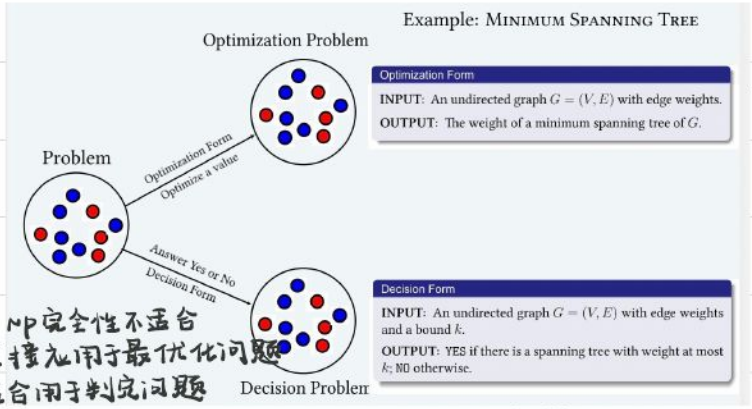
因此，给定一个二分无向图 G，可以通过创建流网络 G'，在其上运行 Ford-Fulkerson 方法来找到一个最大匹配。这个最大匹配 M 可以直接从找到的整数最大流 f 获得。由于二分图中的任何匹配的基数的最大值为 $\min(L,R)=O(V)$，G' 中的最大流的值为 $O(V)$。因此，可以在 $O(VE')=O(VE)$ 时间内找到一个二分图的最大匹配，因为 $|E'|=\Theta(E)$。

练习

Ch 34 NP完全性

Decision Optimization

(1) 判定问题与最优化问题



一个优化问题可以通过给要优化的目标值加上一个界限，转化为一个相关的判定问题

NP完全性不适合直接应用于最优化问题

适合用于判定问题

proof. 优化问题与判定问题难度等价

① A solve B opt ⟶ A solve B' dec

② A solve B' dec ⟶ A solve B opt

假设最小值在区间 $[L, U]$ 内：

1. 取中间值 $mid = (L + U)/2$
2. 问判定问题：

   是否存在解使得 $f(x) \leq mid$？
3. 根据回答调整范围：

- **Yes** → 存在解 ≤ mid → 最小值 ≤ mid → 可以**调整上界** $U = mid$
- **No** → 不存在解 ≤ mid → 最小值 > mid → 可以**调整下界** $L = mid$

重复这个过程，直到收敛到最小值。

(2) complexity classes 复杂类 ⇒ 根据已知的最快算法进行划分 + decision problem

① P class (P类)：在多项式时间内可以解决的问题 ⇒ 对于X的每一个实例I，算法A(1)输出Yes当且仅当I本身是一个Yes实例

② NP class (NP类)：(1) 对于每一个答案为Yes的实例，其解可以在多项式时间内被验证的判定问题的集合

(2) 可以被一个非确定性图灵机在多项式时间内解决的判定问题集合

上面两个定义是等价的，因为基于图灵机的算法可以看成包含两个阶段：

第一阶段是对问题的解进行一次"猜测"，这个解是以非确定性的方式生成的；

第二阶段是用一个确定性的算法来验证这个猜测是否真的是该问题的一个解。

① $P \subseteq NP$ (定义)

② $P = NP$?

③ co-NP：对于一个判定问题X，当且仅当它的补问题又属于NP类时，X属于co-NP

$P \subseteq$ co-NP

④ NP-Complete class：一个问题Y是NP完全问题满足：

① $Y \in NP$

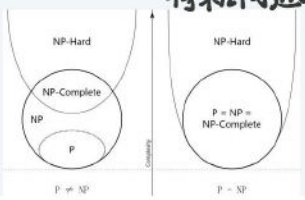
② 对于任何问题 $X \in NP$，都有 $X \leq_p Y$ (通过多项式时间归约把X转化为Y)

⑤ NP-hard：一个问题Y是NP-Hard满足：对于任何问题 $X \in NP$，都有 $X \leq_p Y$

通俗理解：NP-Hard问题至少和NP中最难的问题一样难
它可能不是判定问题，也可能是搜索问题/优化问题

- Interestingly, there are also decision problems that are NP-hard but not NP-complete such as the halting problem.

停机问题→不可判定的



(3) Polynomial-time reduction 多项式时间归约

定义

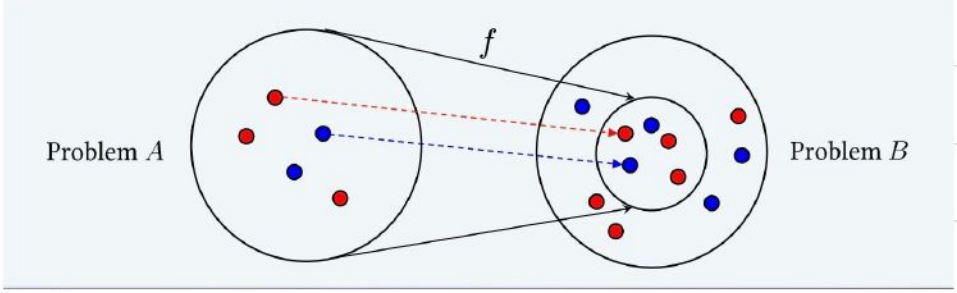
(any-to-one)

假设有一个过程，它可以将A的任何实例α转化成B的具有以下特征的某个实例β：

① Transformation 转换性：转换操作需要多项式时间

② Equivalence 等价性：两个实例的解是相同的，即α的解是Yes当且仅当β的解也是Yes

符号表示：$A \leq_p B$ A可归约到B



B至少不比A简单

定理

如果B在多项式时间内被解决，那么A也可以多项式时间内被解决

我们称这一过程为多项式时间归约算法(reduction algorithm)，并且如图34-1所示，它提供了一种在多项式时间内解决问题A的方法：

1. 给定问题A的实例α，利用多项式时间归约算法，将它转化为问题B的一个实例β。
2. 在实例β上，运行B的多项式时间判定算法。
3. 将β的解作为α的解。

只要上述过程中的每一步只需要多项式时间，则所有三步合起来也只需要多项式时间，这样，我

反之，如果A是困难的，那么B也是困难的

特点 传递性

if $X \leq_p Y$ and $Y \leq_p Z$，那么 $X \leq_p Z$

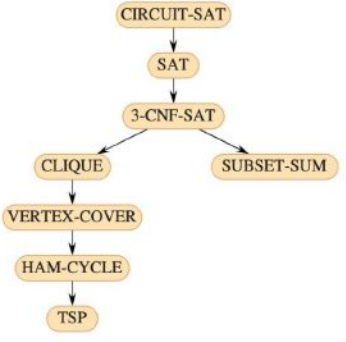
(4) 证明NPC问题(定义出发)(必考)

步骤(prove $Y \in$ NP-complete)

Step1：证明 $Y \in NP$

Step2：choose an NP-complete problem X

Step3：证明 $X \leq_p Y$



3SAT $\leq_p$ CLIQUE

问题定义

3SAT:

INPUT: Given a CNF (Conjunctive Normal Form) $\Phi = C_1 \wedge C_2 ... \wedge C_k$.

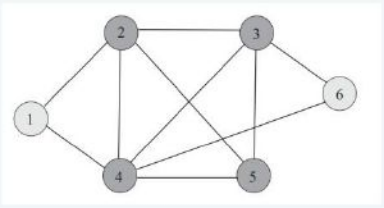
OUTPUT: Is there an assignment of all x_i such that all clauses C_j are satisfied?

CLIQUE:

INPUT: Given a graph $G = (V, E)$, and an integer k.

OUTPUT: Is there a subset of vertices $V' \subseteq V$, $|V'| = k$, such that each pair of vertices in V' are connected by an edge?

V'中任意两个顶点之间都有一条边相连.



TSP

无向图 $G=(V, E)$ 中的**团**(clique)是一个顶点子集 $V' \subseteq V$，其中每一对顶点之间都由 E 中的一条边来连接。换句话说，一个团是 G 的一个完全子图。团的**规模**是指它所包含的顶点数。**团问题**是关于寻找图中规模最大的团的最优化问题。作为判定问题，我们仅仅要考虑的是：在图中是否存在一个给定规模为 k 的团？其形式定义为：

CLIQUE=$\{\langle G, k\rangle : G$ 是一个包含规模为 k 的团的图$\}$

图 34-13　34.4 节和 34.5 节中 NP 完全性证明的结构，所有的证明最终都是通过对 CIRCUIT-SAT 的 NP 完全性归约而得到的

多项式变换

思路：　3 SAT　等价性　Clique

$\phi = C_1 \wedge C_2 \cdots \wedge C_k$　　$|V'| = k$

① $\forall C_i$ 中有一个变量为 True.　　↓

② k个语句同时取True　→　$G(V')$ complete graph 任意两点之间有边.

x_i 和 $\neg x_i$ 不能同时取True

同一个语句中可以不建立起联系.

Proof. CLIQUE 问题是NP完全的

(1) CLIQUE $\in$ NP

对于 $G=(V, E)$，团中顶点集 $V' \subseteq V$，只对于 $\forall u, v \in V'$，通过检查 (u, v) 是否属于 E，可在多项式时间内确定 V' 是否是团.

(2) 3SAT $\leq_p$ CLIQUE

给定一个3-SAT的实例

设 $\Phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ 是3-CNF形式中一个具有 k 个子句的布尔公式

对 $r=1, 2, \cdots k$ 每个子句 C_r 中恰好有3个不同的文字 l_1^r, l_2^r, l_3^r

我们将构造一个图 G 使得 Φ 是可满足的当且仅当 G 中包含一个规模为 k 的团

按以下要求构造 G

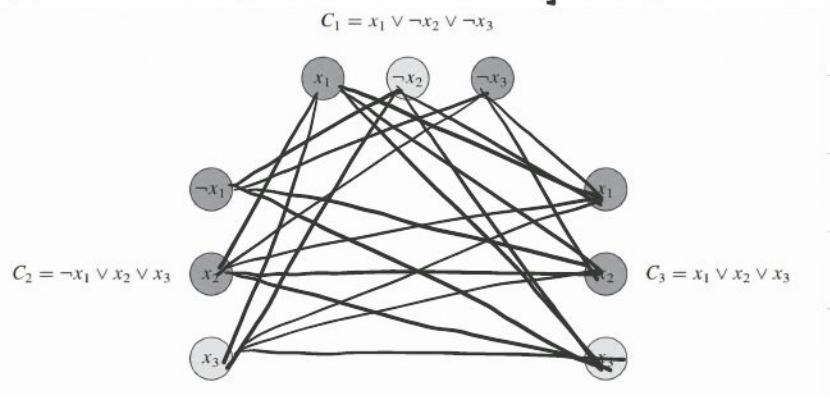
① 对 Φ 中的每个子句 $C_r = (l_1^r \vee l_2^r \vee l_3^r)$，把3个顶点 v_1^r、v_2^r、v_3^r 组成三元组放入 V 中.

② 如果下列两个条件同时满足，用一条边连接 v_i^r 和 v_j^s

③ v_i^r 和 v_j^s 处于不同的三元组中，$r \neq s$

④ 相应文字 literal 是一致的，l_i^r 不是 l_j^s 的非（只要不是对应的互为反的两个点都连）

eg:



1° ⇒

假定Φ有一个可满足性赋值

那么每个子句 C_r 至少包含一个文字 l_i^r，将此文字赋值为1，并且把每个这样的文字对应于一个顶点 v_i^r

从上述每个子句(clause)中挑选出一个这样"true"文字(literal)，得到k个顶点的集合 V'，可以断言 V' 是一个clique 且规模为k

∀ $v_i^r, v_j^s \in V'(r \neq s)$ 根据可满足性赋值，l_i^r 和 l_j^s 者P被赋为1，不可能是互补关系

∴ $(v_i^r, v_j^s) \in E$

2° ⇐

假设G有一个规模为k的团 V'

∵ G中没有连接同一个三元组中顶点的边

∴ V' 中恰好包含每个三元组的一个顶点

∴ 我们可以把满足 $v_i^r \in V'$ 的文字 l_i^r 赋为1且不会出现一个文字与其补一同时为1的情况(互补文字之间不存在连线)

∴ 每个子句可满足的

∴ Φ可满足的

CLIQUE ≤p VERTEX COVER

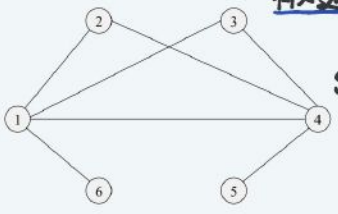
问题定义

VERTEX COVER:

INPUT: Given a graph $G = (V, E)$, and an integer k.

OUTPUT: Is there a set of vertices $S \subseteq V$, $|S| = k$, such that each edge has at least one of its endpoints in S?

能否选出恰好k个顶点，使每条边中恰好一个端点被选中



S={1,4}顶点覆盖.

顶点覆盖问题是指在一个给定的图中，找出具有最小规模的顶点覆盖。把这一最优化问题重新表述为一个判定问题，即确定一个图是否具有一个给定规模 k 的顶点覆盖。作为一种语言，我们定义

VERTEX-COVER=$\{\langle G, k\rangle$：图 G 有一个规模为 k 的顶点覆盖$\}$

proof. 顶点覆盖问题是NP完全的

① VERTEX-COVER ∈ NP

假定已知图 $G=(V,E)$ 和整数k，给定顶点覆盖 $V' \subseteq V$，易在多项式时间内验证 $|V'|=k$，且对于每条边 $(u,v) \in E$，检查是否 $u \in V'$ 或 $v \in V'$

② CLIQUE $\leq_p$ VERTEX-COVER

思路：Any clique $G(V,E)$ K ⟷ vertex cover $G'(V',E')$ K'

$\hat{V}=V'\setminus V''$ ⟹ $\hat{u},\hat{v}\in\hat{V}=V'\setminus V''$

① 找到团 $|V'|=k$

$\forall v_1,v_2\in V' \Rightarrow (v_1,v_2)\in E$

$\forall e(u,v)\in E'$ ⟹ $e\in(\hat{u},\hat{v})\notin E'$

u 或 $v\in V''$（V'' 为集）

$(\hat{u},\hat{v})$ 一定不存在一条边.

补图：完全图中把此图抠掉

为该否问题

过程是以“补图”的概念为基础的。给定一个无向图 $G=(V, E)$，定义 G 的补图 $\overline{G}=(V, \overline{E})$，其中 $\overline{E}=\{(u, v): u, v\in V, u\neq v, (u, v)\notin E\}$。换句话说，$\overline{G}$ 是正好包含不在 G 中的那些边的图。图 34-15 显示出了一个图与其补图，并说明了从 CLIQUE 到 VERTEX-COVER 的归约过程。

归约算法的输入是团问题的实例 $\langle G, k\rangle$。它计算出补图 $\overline{G}$，这很容易在多项式时间内完成。归约算法的输出是顶点覆盖问题的实例 $\langle\overline{G}, |V|-k\rangle$。为了完成证明，下面来说明该变换的确是一个归约过程：图 G 具有一个规模为 k 的团，当且仅当图 $\overline{G}$ 有一个规模为 $|V|-k$ 的顶点覆盖。

(any) (one) ⇒特定情况.

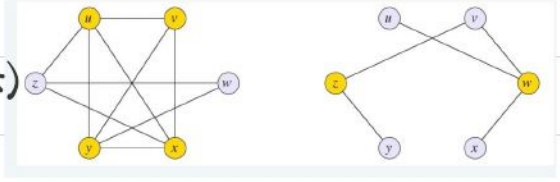
① G 包含一个团 $V'\subseteq V$，其中 $|V'|=K$ ⇒ $V-V'$ 是 $\overline{G}$ 中的一个顶点覆盖

设 (u,v) 是 $\overline{E}$ 中的任意边 ⇒ $(u,v)\notin E$

∴ u 或 v 中至少一个不属于 V'（每一对顶点间都至少有一条）

⇕ E 中的边与其相连

u 或 v 中至少有一个属于 $V-V'$



∴ (u,v) 被 $V-V'$ 覆盖

∵ (u,v) 是 E 中任意边

∴ E 中每条边都能被 $V-V'$ 覆盖

∴ 规模为 $|V|-K$ 的集合 $V-V'$ 形成了 $\overline{G}$ 的一个顶点覆盖.

② $\overline{G}$ 有一个顶点覆盖 $V'\subseteq V$ 其中 $|V'|=|V|-K$

对所有 $(u,v)\in V$ 如果 $(u,v)\in\overline{E}$，那么 $u\in V'$ 或 $v\in V'$ 或两者同时成立.

与此相对，对所有 $(u,v)\in V$，如果 $u\notin V'$ 且 $v\notin V'$，那么 $(u,v)\notin\overline{E}$ ⇒ $(u,v)\in E$

∴ $V-V'$ 是一个团，规模 $|V|-|V'|=K$

补充证明NPC问题

3SAT $\leq_p$ VERTEX COVER

3SAT实例

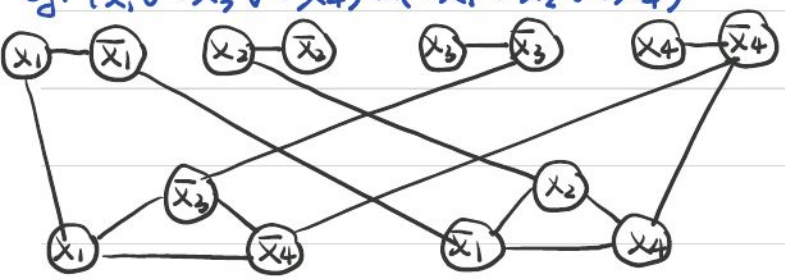
给定一个式子 $= C_1 \wedge C_2 \cdots \wedge C_m$

我们构造Vertex Cover图 $G=(V,E)$ 并设 $k=n+2m$ 其中n为文字的数量，m为句数量

顶点(V)：
- A类：3m nodes → 每个子句3个literal → 每个literal构建一个结点
- B类：2n nodes → 每个literal x_i 构造2个节点：x_i 和 $\neg x_i$

边(E)：
- A类内部：同一个子句内3个结点间都有边
- B类内部：每个literal自己正负结点互相连接
- A类和B类之间：同一个literal连接到A类中对应它的结点

eg: $(x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_4)$



要是可满足的当且仅当图G中存在一个大小规模为 $k=n+2m$ 的J顶点覆盖.

① ⇒

假设中有一个可满足性的赋值

覆盖B类内部+A/B之间

B类结点：对于每个literal，如果在赋值中为true，把对应的结点加入S

A类结点：对于每个子句
- 如果子句中有一个literal为true → 选剩下两个结点加入S
- 如果子句中有两个literal为true → 选剩下一个和任意一个为true的literal节点加入S
- 如果子句中三个literal均为true → 任选两个结点加入S

至少两边覆盖 <= A类内部边

易知此时构造集合S满足J顶点覆盖问题且 $k=n+2m$

② ⇐

假设存在一个J顶点覆盖S其规模 $k=n+2m$

A类中，对任何子句，至少需要选2个点覆盖
B类中，每对 x_i，$\neg x_i$ 中至少选1个点覆盖

$\Rightarrow$ ($k=n+2m$)

A类恰好选2m个点
B类恰好选n个点

对于B类中 被选入S的literal赋值为true，此时保证 x_i 和 $\neg x_i$ 不会同时为true.
(x_i 和 $\neg x_i$ 中一定有其中之一选入S+A与B

看某个子句三角形。

因为 A 类每个三角形恰好选 2 个点，所以一定剩下 1 个点没被选。

假设某个子句中 A 类没选的点是：

$$\ell$$

由于 A 中的 ℓ 没有进 S，而它和 B 中相同的 ℓ 有一条边：

$$(A_\ell, B_\ell)$$

为了覆盖这条边，B 中的 ℓ 必须进 S。

也就是：

$$A_\ell \notin S \Rightarrow B_\ell \in S$$

如果我们规定：

$$B \text{ 中在 } S \text{ 的 literal 为 TRUE}$$

那么就得到：

$$\ell = TRUE$$

所以这个子句至少有一个 literal 为 TRUE。

VERTEX COVER $\leq_p$ SET COVER

SET COVER

INPUT: A set $U = \{e_1, e_2, \ldots, e_n\}$ of n elements, a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, $S_i \subseteq U$ for all i, and an integer k.

OUTPUT: Does there exist a set $\mathcal{C} \subseteq \mathcal{S}$, such that $\bigcup_{S_i \in \mathcal{C}} = U$ and $|\mathcal{C}| \leq k$?

全集

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$　　$S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$　　$S_d = \{5\}$

$S_e = \{1\}$　　$S_f = \{1, 2, 6, 7\}$

是否存在一个集合族C，它是S的子集
使得C中所有集合的并集等于全集U
并且选出集合数量不超过K

① SET COVER $\in$ NP

给定 SET COVER 的一个实例 $\langle U, S, k\rangle$，假设有一个子集 $S' \subseteq S$ 作为候选解。我们可以在多项式时间内验证 $|S'| \leq k$ 和 $\bigcup_{s \in S'} s = U$

② VERTEX COVER $\leq_p$ SET COVER

给定一个 VERTEX COVER 实例 $\langle G=(V,E), k\rangle$

构造一个 SET COVER 实例 $\langle U, S, k\rangle$

令 $U = E$

对每个 $v \in V$ 构造一个集合 $S_v = \{e \in E : e \text{ is incident to } v\}$（原图中每一个顶点对应了一个集合）

证明图 G 有一个规模为 k 的顶点覆盖当且仅当 S 中存在 k 个集合覆盖 U

1° ⇒

令 $X \subseteq V$ 为 G 中一个规模为 k 的顶点覆盖

那么 $Y = \{S_v : v \in X\}$ 为规模为 k 集合覆盖

2° ⇐

令 $Y \subseteq S$ 为一个规模为 k 的集合覆盖

那么 $X = \{v : S_v \in Y\}$ 为规模为 k 的顶点覆盖.

※把集合覆盖解中选中的那些集合反过来找原图顶点

3SAT $\leq_p$ SUBSET SUM(3集和问题)

(1)问题定义

整数集合.

Subset Sum

INPUT: Given a collection of integers, $S = \{x_i\}_{i\in[n]}$ and a target integer t.
OUTPUT: Ask whether there exists a subset of $S' \subseteq S$ such that the summation of integers in S' is equal to t.

是否存在一个子集 $S' \subseteq S$, 其元素和为t

和通常一样，我们把该问题定义为一种语言：

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{存在一个子集 } S' \subseteq S, \text{使得 } t = \sum_{s\in S'} s\}$$

① SUBSET-SUM ∈ NP

候选解.

证明 为了说明 SUBSET-SUM 属于 NP，对该问题的实例$\langle S, t\rangle$，设子集 S' 是证书。我们使用验证算法可以在多项式时间内检查是否有 $t = \sum_{s\in S'} s$。

② 3SAT $\leq_p$ SUBSET-SUM

现在来证明 3-CNF-SAT$\leqslant_p$ SUBSET-SUM。给定变量 x_1，x_2，…，x_n 上的一个 3-CNF 公式 ϕ，它由子句 C_1，C_2，…，C_k 构成，每个子句恰好包含 3 个不同的文字，归约算法构造出子集和问题的一个实例$\langle S, t\rangle$使得 ϕ 是可满足的，当且仅当存在 S 的一个子集，其元素和恰为 t。不失一般性，下面对 ϕ 做两个简化性的假设。首先，ϕ 的任一子句都不会既包含某个变量，又包含该变量的非，因为这样的子句对变量的任何赋值来说恒成立。其次，每一个变量至少在一个子句中出现一次，否则，对没有出现在任何子句中的变量赋任意值都是没有影响的。

对每个变量 x_i，归约算法都要在 S 中生成两个数；对每个子句 C_j，也要在 S 中生成两个数。所生成的数都是十进制的，且每个数都包含 $n+k$ 个数位，每个数位对应于一个变量或一个子句。十进制(或其他进制)有着我们所需要的、可以避免从低位向高位进位的性质。

如图 34-19 所示，我们构造集合 S 和目标 t 如下：对于每一个数位，都用一个变量或一个子句来标记。最低有效 k 位用子句标记，最高有效 n 位用变量标记。

- 目标 t 在每个用变量标记的数位上都有个 1，在每个用子句标记的数位上都有个 4。
- 对于每个变量 x_i，集合包含两个整数 v_i 和 v'_i。每个 v_i 和 v'_i 在 x_i 所标记的数位上都是 1，其他变量位上都是 0。如果子句 C_j 中包含文字 x_i，那么 v_i 中被 C_j 所标记的数位包含一个 1。如果 C_j 中包含文字 $\neg x_i$，那么 v'_i 中被 C_j 所标记的数位包含一个 1。在 v_i 和 v'_i 中，所有其他由子句标记的数位都是 0。

在集合 S 中，所有 v_i 和 v'_i 的值都是唯一的。这是为什么呢？例如，对于 $l\neq i$，在最高有效的 n 个数位上，没有 v_l 或 v'_l 的值会与 v_i 和 v'_i 相等。此外，根据上面所做的简化性假设，在最低有效 k 位上也不存在 v_i 和 v'_i 的值相等。如果 v_i 和 v'_i 相等，那么 x_i 和 $\neg x_i$ 将不得不恰好出现于同一子句集中。但是我们又假设任何一个子句中都不能同时包含 x_i 和 $\neg x_i$，而且 x_i 和 $\neg x_i$ 其中之一出现在某一子句中，所以必定存在某一子句 C_j，其中 v_i 和 v'_i 是不同的。

- 对于每个子句 C_j，集合 S 包含两个整数 s_j 和 s'_j，除了由 C_j 标记的数位外，s_j 和 s'_j 的所有数位上都是 0。对于 s_j，在 C_j 的数位上为 1，而对于 s'_j，在 C_j 的数位上为 2。这些整数都是“松弛变量”，可以用其获得每个子句所标记的数位。从而将它们的值加到目标

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1

s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

图 34-19 从 3-CNF-SAT 到 SUBSET-SUM 的归约。3-CNF 的公式为 $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$，其中 $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$，$C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$，$C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$，$C_4 = (x_1 \vee x_2 \vee x_3)$。$\phi$ 的一个可满足性赋值为 $\langle x_1 = 0, x_2 = 0, x_3 = 1 \rangle$。归约过程所产生的集合 S 包含了一些十进制数；按照从上往下的顺序，$S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$。目标 t 为 1114444。子集 $S' \subseteq S$ 在图中用浅阴影表示，它包含了 v'_1、v'_2 和 v'_3，对应于可满足性赋值。它还包含了松弛变量 s_1、s'_1、s'_2、s'_3、s_4 和 s'_4，以便在由 C_1 到 C_4 所标记的数位上达到目标值 4

只要简单地观察一下图 34-19，就会发现所有 s_j 和 s'_j 的值在集合 S 中都是唯一的。

注意在任意数位上，数位和的最大值为 6，这个和出现在由子句所标记的数位上(来自 v_i 和 v'_i 的 3 个 1，加上来自 s_j 和 s'_j 的值 1 和 2)。因此，按照十进制来解释这些数，就不会出现由低位向高位进位的情况[⊖]。

以上的归约过程可以在多项式时间内完成。集合 S 包含了 $2n+2k$ 个值，其中每一个值都有 $n+k$ 个数位，产生每一个数位的时间都是 $n+k$ 的多项式。目标 t 有 $n+k$ 个数位，其中的每一个数位都可以由归约过程在常量时间内产生。

现在，我们来证明 3-CNF 公式 ϕ 是可满足的，当且仅当存在一个子集 $S' \subseteq S$，其元素之和为 t。首先，假设 ϕ 有一个可满足性赋值。对 $i=1, 2, \cdots, n$，如果在此赋值中存在 $x_i = 1$，就将 v_i 包含在集合 S' 中。否则，就将 v'_i 包含在集合 S' 中。换句话说，我们是把在可满足性赋值中，与值为 1 的文字对应的 v_i 和 v'_i 值包含在 S' 中。在对所有的 i 包含了 v_i 或 v'_i(两者取其一)之后，并且将所有由 s_j 和 s'_j 中的变量所标记的数位置 0 后，我们可以看出，对于每个由变量标记的数位，S' 中元素之和必为 1，这与目标 t 中的那些数位正好是匹配的。由于每个子句都是可满足的，故子句中必有某个文字的值为 1。于是，由一个子句所标记的每个数位都至少有一个 1。这一数据可以在 S' 元素的和值中，作为 v_i 或 v'_i 的值。事实上，在每个子句中，可能有 1、2 或 3 个文字的值为 1，因此，根据 S' 中所包含的 v_i 或 v'_i 值的情况，每个由子句所标记的数位和为 1、2 或 3。例如，在图 34-19 中，在某一可满足性赋值中，文字 $\neg x_1$、$\neg x_2$ 和 x_3 的值为 1。子句 C_1 和 C_4 都恰包含这三个文字中的一个，因此，v'_1、v'_2 和 v_3 共同为 C_1 和 C_4 中数位的和值贡献了一个 1。子句 C_2 包含这些文字中的两个，因而，v'_1、v'_2 和 v_3 共同为 C_2 中数位的和值贡献了一个 2。子句 C_3 包含所有这三个文字，因而，v'_1、v'_2 和 v_3 共同为 C_3 中数位的和值贡献了一个 3。在每个由子句 C_j 所标记的数位中，通过将松弛变量 $\{s_j, s'_j\}$ 的一个非空子集包含进 S'，即可达到目标值 4。在图 34-19 中，S' 包含了 s_1，s'_1，s'_2，s_3，s_4 和 s'_4，由于我们已经在和值的所有数位中匹配了目标值，且不会发生进位，因此，S' 中元素的和值为 t。

接下来，假设有一个子集 $S' \subseteq S$，其元素之和为 t。对每一个 $i=1, 2, \cdots, n$，子集 S' 必定恰好包含 v_i 和 v'_i 两者中的一个，否则，变量所标记的数位和就不会为 1。如果 $v_i \in S'$，就将 x_i 置 1。否则，$v'_i \subseteq S'$，就将 x_i 置 0。我们断言：对 $j=1, 2, \cdots, k$，每个子句 C_j 可以通过此赋值得到满足。为了证明这一断言，注意到由于松弛变量 s_j 和 s'_j 合起来的贡献至多为 3，因此为了在 C_j 标记的数位中达到和值 4，子集 S' 必须至少包含 v_i 或 v'_i 值中的一个，使得 C_j 在标记的数位上有个 1。如果 S' 包含一个 v_i，它在 C_j 位置上有个 1，则文字 x_i 会出现在子句 C_j 中。当 $v_i \in S'$ 时，我们已将 x_i 置 1，因此，子句 C_j 是可满足的。如果 S' 包含一个 v'_i 且它在该位置上有个 1，则文字 $\neg x_i$ 会出现在子句 C_j 中。当 $v'_i \in S'$ 时，我们已将 x_i 置 0，因此，子句 C_j 再次被证明是可满足的。于是，ϕ 的所有子句都是可满足的，这样就完成了整个证明。■

SUBSET SUM $\leq_P$ PARTITION

Partition

INPUT: Given a collection of integers, $S = \{y_i\}_{i\in[n]}$.
OUTPUT: Ask whether there exists a subset of $S' \subseteq S$ such that the summation of integers in S' is equal to summation of those in $S \backslash S'$.

$\sum_{x\in S'} x = \sum_{x\in S\backslash S'} x$ (把一堆数分成两组，使两组和一样)

① PARTITION ∈ NP

② **Theorem**

SUBSET SUM $\leq_P$ PARTITION

Proof

S中是否有一部分和为t

Given any instance of subset sum, $S = \{x_i\}_{i\in[n]}$ and a target integer t, we construct an instance of partition:

T是否能分成两边和相等

Define $C = \sum_{i=1}^{n} x_i$. Consider a set of integers $T = \{y_i\}_{i\in[n+2]}$ satisfying:

- $y_i = x_i$, for $i \in \{1,2,\cdots,n\}$;
- $y_{n+1} = t + 1$;
- $y_{n+2} = C - t + 1$.

→ T总和：$C+t+1+C-t+1 = 2C+2$

平分：$C+1$

"⇒" Assume that there exists a subset $S' \subseteq S$ such that the summation of integers in S' is equal to t.

- Let $T' = \{y_i | x_i \in S'\} \cup y_{n+2}$.
- Check that the summation is exactly to $C + 1$.

t → $C-t+1$

"⇐" Suppose that there exist a subset $T' \subseteq T$ such that the summation is $C + 1$.

- y_{n+1} and y_{n+2} must be in the different subsets. (Why?)
- The summation of elements in the subset including y_{n+2} is equal to t.

$y_{n+2} = C-t+1$

$(C+1)-(C-t+1) = t$

NPC问题答题模板

1. **NP**：候选解 + 多项式验证
2. **规约**：
   - 已知 NP-hard → 构造 gadget → 当且仅当关系
   - 保证多项式时间
3. **示例/图示**：验证 gadget 正确性
4. **反向证明**：确保构造解可还原原问题解

常用的 gadget 类型

Gadget 类型	用途	示例
顶点/节点	模拟布尔变量或元素	Vertex Cover, Clique
边/连接	模拟约束（子集/互斥/相邻关系）	顶点覆盖每条边、Clique 相连
辅助元素	排他或保证唯一分组	Subset Sum → Partition $t+1, C-t+1$
子集	模拟选择	Set Cover, Partition

ch 35 近似算法

(1) efficient algorithm: ① polynomial-time 多项式时间
② deterministic algorithm 确定性算法
③ optimal solution

⇓ deal with hard problems

Trade-off "quality" and "time"

- give up optimum restriction → 近似最优解 → 近似算法
  - 启发式
  - 局部搜索
- give up poloynomial-time restriction
- give up determinisitic restriction

(2) α-approximation algorithm α-近似算法

定义

对于一个优化问题，如果一个算法满足下面两个条件，就称其为一个α-近似算法

① 时间要求：多项式时间

② 质量要求：

算法输出一个解S，它的值和最优解OPT的值之间相差不超过一个因子α → 近似因子/近似比

最小化问题：$Value(S) \le \alpha OPT \ (\alpha>1) \rightarrow 1 \le \frac{Value(S)}{OPT} \le \alpha \ (\alpha>1) \leftarrow OPT \le Value(S) \le \alpha OPT$

最大化问题：$Value(S) \ge \alpha OPT \ (\alpha<1) \rightarrow 1 \ge \frac{Value(S)}{OPT} \ge \alpha \ (\alpha<1) \leftarrow \alpha OPT \le Value(S) \le OPT$

一个最优化问题的**近似模式**(approximation scheme)就是这样一种近似算法，它的输入除了该问题的实例外，还有一个值$\epsilon>0$，使得对任何固定的ϵ，该模式是一个$(1+\epsilon)$近似算法。对一个近似模式来说，如果对任何固定的$\epsilon>0$，该模式都以其输入实例规模n的多项式时间运行，则称此模式为**多项式时间近似模式**。(Polynomial-time approximation scheme) PTAS

随着ϵ的减小，多项式时间近似模式的运行时间可能会迅速增长。例如，一个多项式时间近似模式的运行时间复杂度可能达到$O(n^{2/\epsilon})$。在理想情况下，如果ϵ按一个常数因子减小，为了获得预期的近似效果，所增加的运行时间不应超过一个常数因子(尽管这两个常数因子不一定相同)。

Full Polynomial-time approximation scheme FPTAS

对一个近似模式来说，如果其运行时间表达式既为$1/\epsilon$的多项式，又为输入实例规模n的多项式，则称其为**完全多项式时间近似模式**。例如，近似模式的运行时间可能是$O((1/\epsilon)^2 n^3)$。对于这样的模式，ϵ的任意常数倍减少可以引起运行时间相应常数倍的增加。

(3) 设计近似算法的核心步骤

A dilemma: 如何在不获取最优解OPT的确切值的情况下与OPT建立联系

意思是说，在设计近似算法或分析最优解的情况下，有时候我们并不知道最优解OPT的具体数值，但我们仍然希望证明或者保证算法输出与OPT的比值在某个近似范围内，或者利用某种方法间接"建立联系"以进行分析。

最小化问题

策略：找到OPT的下界 lower bound，并将一个解的质量与这个下界进行比较而不是直接与OPT比较

$\min \frac{Value(S)}{OPT} \le \alpha$ → 未知

$OPT \ge LB$ → 下界

$\frac{Value(S)}{OPT} \le \frac{Value(S)}{LB} \le \alpha' \Rightarrow \frac{Value(S)}{OPT} \le \alpha'$

max $\frac{value(s)}{OPT} \geq \alpha$. $OPT \leq UB$ (UB → 上界) $\frac{value(s)}{OPT} \geq \frac{value(s)}{UB} \geq \alpha$.

Step1：找到OPT的一个紧下界(tight lower bound)

Step2：设计一个过程得到一个可行解，并将这个解的值和OPT的下界进行比较

(紧下界 → 尽量接近OPT)

1. 顶点覆盖问题

方法一：

Step1：找到OPT的一个紧下界

lemma1：对于图G中任意一个匹配M以及任意一个顶点覆盖S，都有$|S| \geq |M|$

proof. 对于匹配M中的每一条边，为了覆盖这条边，顶点覆盖S至少包含这条边的一个端点，
∴$|S| \geq |M|$

推论：$\min_{S \text{ is a vertex cover}} |S| \geq \max_{M \text{ is a matching}} |M|$

即 $OPT_{vc} \geq |M^*|$

Step2：

VERTEX-COVER-APPRO-WITH-MM algorithm

1: Find a maximal matching M in G;　图G中找到一个最大匹配
2: $S \leftarrow \{u \in V : \exists (u,v) \in M\}$;　最大匹配M中每条边互不相交的
3: **return** S　任意未在匹配中的边都至少与匹配边共享一个顶点，加两个顶点保证所有边被覆盖

proof S是顶点覆盖

反证法

假设S不是顶点覆盖，那么$\exists (u,v) \in E$使得$u \notin S, v \notin S \rightarrow u \notin M, v \notin M$

∴我们可以将边(u,v)加入匹配M中得到更大匹配

∴与M为最大匹配矛盾

分析：基于最大匹配的顶点覆盖算法是顶点覆盖问题的2-近似算法

proof. $|S| = 2|M| \leq 2OPT$

方法二：

贪心思想

(6) 找它们的顶点覆盖。

APPROX-VERTEX-COVER(G)

```
1 C=∅
2 E'=G.E
3 while E'≠∅
4     let (u,v) be an arbitrary edge of E'
5     C=C∪{u,v}                    ⇒O(|E|)
6     remove from E' every edge incident on either u or v
7 return C
```

重复从E'中任选一条边(u,v)，将u和v加入到C并删掉E'中所有被u或v覆盖的边，直至E'为空！

证明上述算法是一个多项式时间的2-近似算法

① 证明多项式运行时间 O(|E|)
② 设A为算法第4行选出的边的集合

为了说明 APPROX-VERTEX-COVER 返回顶点覆盖的规模至多为最优覆盖的 2 倍，设 A 为 APPROX-VERTEX-COVER 算法中第 4 行选出的边集合。为了覆盖 A 中的边，任意一个顶点覆盖（特别是最优覆盖 C^*）都必须至少包含 A 中每条边的一个端点。如果一条边在第 4 行中被选中，那么在第 6 行就会从 E' 中删除所有与其端点关联的边。因此，A 中不存在两条边具有共同的端点，从而 A 中不会存在两条边由 C^* 中的同一顶点所覆盖。于是，最优顶点覆盖的规模下界如下：

$|C^*| \geq |A|$

$$|C^*| \geqslant |A| \qquad (35.2)$$

算法第 4 行的每一次执行都会挑选出一条边，其两个端点都不在 C 中，因此，所返回顶点覆盖的规模上界（实际上是一个上界）为：

$$|C| = 2|A| \qquad (35.3)$$

$|C| = 2|A| \leq 2|C^*|$

2. 集合覆盖问题

贪心思想：每一轮选择当前能包含覆盖最多"还没被覆盖元素"的集合。
如果多个集合覆盖的新元素数量一样多，随机选一个

```
1: procedure GREEDY SET COVER(U, (S_1, ..., S_m)):
2:     Initialize X ← U. ▷ X will denote the collection of uncovered elements.
3:     Initialize I ← ∅. ▷ I denotes the indices of the sets picked in our solution. // I ← ∅
4:     while X ≠ ∅ do:                                   (下标)
5:         Pick j ∈ [m] which maximizes |S_j ∩ X|.
6:         ▷ That is, the set which covers the maximum number of uncovered elements.
7:         I ← I ∪ j. // I = I ∪ {S_j} (加入集合)
8:         X ← X \ S_j.
9:     return I.
```

运行时间 O(|X||S|)

分析：基于贪心的集合覆盖问题是集合覆盖问题的log n近似算法

Optimal solution（令K为最优解）

如果K个集合能覆盖全部元素，那么这K个集合里，至少有一个集合覆盖不少于k元素

假设当前有 r 个未覆盖元素。

因为最优解用 k 个集合就能覆盖全部元素，所以这 r 个未覆盖元素也可以被这 k 个集合覆盖。

如果这 k 个集合每个都覆盖少于：

$$\frac{r}{k}$$

个未覆盖元素，那么它们加起来覆盖的元素就少于：

$$k \cdot \frac{r}{k} = r$$

这就不可能覆盖全部 r 个未覆盖元素。

所以至少有一个集合能覆盖：

$$\frac{r}{k}$$

个未覆盖元素。

也就是覆盖当前剩余元素的至少：

$$\frac{1}{k}$$

Greedy algorithm solution:

> 在每一轮中，贪心算法选择的集合至少能覆盖当前剩余元素的 $\frac{1}{k}$。为什么？

原因是：这样的集合一定存在，剩余元素也可以被最多 k 个集合覆盖 ⇒

既然最优解的 k 个集合中，至少有一个能覆盖当前剩余元素的 $\frac{1}{k}$，而贪心算法选的是"当前覆盖最多"的集合，所以贪心选的集合只会更好，不会更差。

所以贪心这一轮也至少覆盖：

$$\frac{1}{k}$$

比例的未覆盖元素。

因为最优解那 k 个集合本来能覆盖全集 U。

那它当然也能覆盖 U 的任意一部分，包括当前还没被覆盖的元素 X。

所以对于每一轮，剩余元素 X 都可以被那 k 个最优集合覆盖。

于是平均来看，总有一个集合覆盖至少：

$$\frac{|X|}{k}$$

个剩余元素。

令 E_t 为在 t 轮选代后仍然未被覆盖元素.

由于根据上述分析，贪心算法总是选择至少覆盖当前剩余元素的 $\frac{1}{k}$

∴ $|E_{t+1}| \le |E_t|(1-\frac{1}{k})$

∵ $E_0 = n$

∴ 归纳可得：

$|E_1| \le |E_0|(1-\frac{1}{k})$

$|E_2| \le |E_1|(1-\frac{1}{k}) \le |E_0|(1-\frac{1}{k})^2$

$\vdots$

$|E_t| \le |E_0|(1-\frac{1}{k})^t = n(1-\frac{1}{k})^t$

贪心算法停止当 $|E_t| < 1$

∴ 只要证明 $n(1-\frac{1}{k})^t < 1$

$(1-\frac{1}{x})^x \le \frac{1}{e}$ for $x > 0$

令 $t = k\ln n$

$n(1-\frac{1}{k})^t = n(1-\frac{1}{k})^{k\ln n} \le n(\frac{1}{e})^{\ln n} = n \cdot \frac{1}{n} = 1$

∴ 贪心算法最多运行 $k\ln(n+1)$ 轮

$$\frac{\text{Value(greedy)}}{OPT} \le \frac{k\ln(n+1)}{k} = \ln(n+1)$$

∴ 近似比为 $\ln n$

3. 旅行商问题 (Traveling-Salesperson problem) TSP

(1) 问题定义:

TRAVELING-SALESPERSON

INPUT: a complete undirected graph $G = (V, E)$ that has a non-negative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$.
OUTPUT: a hamiltonian cycle (a tour) of G with minimum cost.

完全无向图

希望找G的一条具有最小代价的哈密顿回路.

满足三角不等式旅行商问题

TRAVELING-SALESPERSON WITH THE TRIANGLE INEQUALITY

INPUT: a complete undirected graph $G = (V, E)$ that has a non-negative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$. Moreover, c satisfies the triangle inequality: for all vertices $u, v, w \in V$, $c(u, w) \le c(u, v) + c(v, w)$
OUTPUT: a hamiltonian cycle (a tour) of G with minimum cost.

直观理解: 一些情况, 从一个地方直接到另一个地方w花费的代价总是最小的

Hamiltonian tour. Given a strongly connected graph $G = (V, E)$ ask whether or not there exist a cycle that traverses each vertex of exactly once (called Hamiltonian tour).

(2) 算法思想

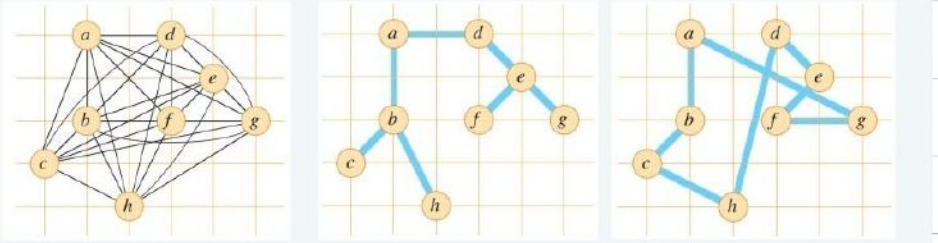
利用前一小节的方法，我们首先要计算出一个结构(即最小生成树)，其权值是最优旅行商路线长度的下界。接着，要利用这一最小生成树来生成一条遍历线路，其代价不大于最小生成树权值的2倍，只要代价函数满足三角不等式即可。下面的算法实现了这一过程，该算法将23.2节

伪代码:

APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in G.V$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r) 多项式时间 $O(E \log V)$
3 let H be a list of vertices, ordered according to when they are first visited in a preorder tree walk of T 对最小生成树T做一次先序遍历, 按照每个顶点第一次被访问到的顺序把顶点依次放进列表H $O(V)$
4 **return** the hamiltonian cycle H

(MST) 带权无向连通图



MST ALG: Preorder Tree Walk

APPROX-TSP-TOUR返回旅行路线

证明: 如果旅行商问题某一实例的代价函数满足三角不等式, 则APPROX-TSP-TOUR是其多项式时间的2-近似算法

① 运行时间多项式

② 2-近似

1° 寻找下界

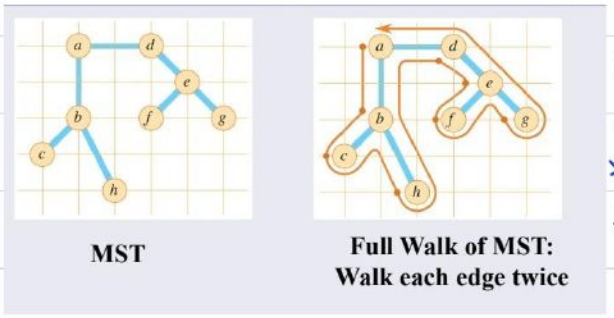
设OPT表示给定顶点集合上的一个最优旅行路线

通过删除一个旅行路线中的任一条边得到生成树 (一条走过所有顶点的路)

$c(MST) \le c(OPT)$

从 OPT 中删掉任意一条边，可以得到一棵生成树；而 MST 是所有生成树中权重最小的那一棵，所以 MST 的权重一定不超过这棵由 OPT 删除边得到的生成树。

2° 证明 α



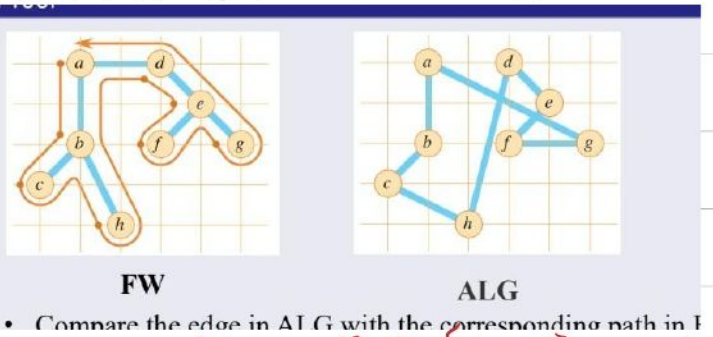
对T进行完全遍历时，在初次访问一个T顶点时输出该T顶点，并且在访问一棵子树返回后输出该顶点

次序：a b c b h b a d e f e g e d a

该遍历恰经过T MST的每条边两次

$c(FW) = 2c(MST)$

根据三角不等式，如果从 Full Walk of MST 中去掉一次对任意(重复)顶点访问代价不会增加，得到顺序与对 MST 先序遍历得到次序一致



- Compare the edge in ALG with the corresponding path in F

$[a \to b \to c \to b \to h \to b \to a \to d \to e \to f \to e \to g \to e \to d \to a]$

如何将树状结构拉回圈去想

$[a \to b \to c] [h \to d] e [f] [g]$

① $cb + bh \ge ch$

② $\begin{cases} hb + ba \ge ha \\ ha + ad \ge hd \end{cases}$

三角不等式

③ $fe + eg \ge fg$.

④ $ge + ed \ge gd$

$gd + ad \ge ag$.

设ALG为对应先序遍历的回路

$c(ALG) \le c(FW)$

$\therefore c(ALG) \le 2c(MST) \le 2c(OPT)$

$\therefore \frac{c(ALG)}{c(OPT)} \le 2$

$\therefore \alpha = 2$

(3) 一般旅行商问题 → 不可近似性.

如果 $P \ne NP$，则对任何常数 $\rho \ge 1$，一般旅行商问题不存在具有近似比为 ρ 的多项式时间近似算法

证明思路：寻找具有 ρ-近似比的多项式时间近似算法来解决一般旅行商问题

难度等同于NP-C问题

proof

证明 采用反证法证明。假设对某个数 $\rho \geqslant 1$，存在一个近似比为 ρ 的多项式时间近似算法 A。不失一般性，假定 ρ 是一个整数(必要的话，可以对其向上取整)。我们来说明如何在多项式时间内用 A 来解决哈密顿回路问题(其定义见 34.2 节)的各种实例。根据定理 34.13，哈密顿回路是 NP 完全问题，因而根据定理 34.4，如果能够在多项式时间内解决这个问题，必须满足 P=NP。

给定一个哈密顿回路问题的实例 $G=\langle V, E\rangle$ 我们可将G转化为如下的一个旅行商问题实例

设 $G'=(V, E')$ 为V上的完全图 $E'=\{(u,v): u,v \in V \text{ 且 } u \neq v\}$

再对 E' 中的每条边按如下方式赋以一个整数代价：

$$c(u,v)=\begin{cases}1 & (u,v)\in E\\ \rho|V|+1 & \text{其他}\end{cases}$$

其中 G' 与 c 的表示可在多项式时间内由G构造出来

- If there exists a Hamiltonian tour,
  ⟹ There exist a tour in TSP G' with cost $|V|$.
- If there exist no Hamiltonian tour,
  ⟹ The minimum cost in TSP is at least $\rho|V| + 1 + (|V| - 1) = (\rho + 1)|V|$

Suppose that there exists a polynomial-time approximation algorithm, ALG, with approximation ratio ρ for the general TSP, what will happen?

If there exists a Hamiltonian tour, ALG will return a tour with cost exactly $|V|$;
If there exist no Hamiltonian tour, ALG will return a tour with cost more than $\rho|V|$.

ALG can decide whether exists a Hamiltonian tour in polynomial time! Contradiction