算法设计与分析-复习

考点汇总

基于课堂笔记。

1. **时间复杂度**、**NP完全性理论**
   - O、o、Ω、ω、Θ 的**含义**与**分析**（$\frac{1}{3}n^3 \in \omega(n^2)$）
   - NPC证明：**3SAT（3合取范式可满足性）**以及相关问题的归约
2. **分治法**
   - **strassen 矩阵乘法**
   - **递归式时间分析（递归树）**
   - **主方法（要注意三种Case之间存在的Gap）**
3. **DP**（选择题为主）
   - 最优子问题证明
   - **BU、TP（备忘录版本的算法实现）**（具体实现可参考 **矩阵链乘**）
   - **矩阵链乘**
4. **贪心**
   - **背包（算法题）**
   - **拟阵（小题）**（具体证明参考 **分数背包问题分析**）
   - 活动选择（选择题）
5. **近似**
   - **01背包：FPTAS**
   - **01背包：2近似算法（PTAS）**
   - **多机调度（知道问题描述）**

时间复杂度

符号含义

- O：渐进上界，最坏情况下的时间复杂度
- o：渐进非紧确上界
- Ω：渐进下界，最好情况下的时间复杂度
- ω：渐进非紧确下界
- Θ：渐进紧确界

紧确界与非紧确界

- 定义上的区别（以O、o为例）：

  > $O(g(n)) = \{f(n)：存在正常量c和n_0，使得所有n \geq n_0，有0 \leq f(n) \leq cg(n)\}$
  >
  > $o(g(n)) = \{f(n)：对任意正常量c > 0，存在常量n_0 > 0，使得所有n \geq n_0，有0 \leq f(n) < cg(n)\}$

  主要区别在常数 c 的限制以及 $f(n)$ 与 $cg(n)$ 的关系上。

- 数学关系上的区别：

  紧确界要求两个函数的最高阶项阶数相同（比值的倒数为非0常数），非紧确界无此要求。

  - $2n^2 = O(n^2)$：$\lim_{n\to\infty} \frac{2n^2}{n^2} = 2$，紧确界。
  - $2n = O(n^2)$：$\lim_{n\to\infty} \frac{2n}{n^2} = 0$，非紧确界。

练习

1. **证明** $\frac{1}{3}n^3 \in \omega(n^2)$ **。**

   > 即要找到常数 c 和 n_0 使得在 $n \geq n_0$ 时 $0 \leq c \cdot n^2 < \frac{1}{3}n^3$ 成立。
   >
   > 对不等式 $c \cdot n^2 < \frac{1}{3}n^3$ 两边除以 n^2，得到 $c < \frac{1}{3}n$。
   >
   > 对 $\frac{1}{3}n$ 进行缩放得到 $n > \frac{1}{3}n > c$，即 $n > c$。
   >
   > 取 n_0 为大于 c 的数即可使得不等式成立。
   >
   > 对于任意的正常量 c 选择 $n_0 = c$，可以证明 $0 \leq c \cdot n^2 < \frac{1}{3}n^3$ 。

NP完全性理论

P、NP、NPC、NP-Hard

- `P`：多项式时间内可解的问题。
- `NP`：多项式时间内可验证问题（指验证其解的正确性）。
- `NPC`：属于`NP`，且`NP`中任一问题均可被多一归约得到的问题。
- `NP-Hard`：可以被`NP`中的问题多一归约得到，但却不一定属于`NP`的问题

> 如果所有`NP`问题都可以**多项式时间归约**到某个问题，则称该问题为NP困难。
>
> 因为NP困难问题未必可以在**多项式时间**内验证一个解的正确性（即不一定是NP问题），因此即使**NP完全**问题有多项式时间的解（**P=NP**），NP困难问题依然可能没有多项式时间的解。因此NP困难问题“至少与NP完全问题一样难”。
>
> ——摘自**维基百科：NP困难**

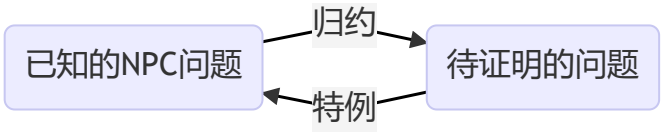
归约

在多项式时间内将问题 A 的一个实例 α 转化为问题 B 的一个实例 β。

NPC证明

步骤

1. 证明问题是`NP`的；
2. 证明一个已知的`NPC`问题可以在多项式时间内被多一归约到该问题（证明是已知NPC问题的特例）。



方法

1. 简单等价规约：最大独立集 $\equiv_p$ 最小顶点覆盖。

   图 $G=(V,E)$ 中的最大独立集 S 与最小顶点覆盖 C 满足 $S\cap C=\emptyset$ 、 $S+C=V$ 。

2. 特殊到一般：顶点覆盖 $\leq_p$ 集合覆盖。

   顶点覆盖问题中每一个点所连接的边对应这集合覆盖问题中的一个集合。

3. 编码：3-SAT $\leq_p$ 顶点覆盖。

   对3-SAT问题中每一个子句中的三个文字互相连边（构成三角形）、不同子句中对同一个文字 x_i 和 $\neg x_i$ 进行连边，构成的图的顶点覆盖对应着3SAT的解。

练习

1. **证明：3-CNF-SAT问题是NPC的。（参考 P.636）**

   证明3-CNF-SAT（后续简称3SAT）是 NP 的：

   当知道3SAT问题的一个解时，可以通过直接将解代入计算得到是否为正确解，代入后的计算可以在多项式时间内完成，所以3SAT问题是 NP 的。

   将SAT多一归约到3SAT：

   > SAT：布尔可满足性问题。
   >
   > “询问给定布尔公式的变量是否可以一致地用值TRUE或FALSE替换，公式计算结果为TRUE。如果是这种情况，公式称为可满足。另一方面，如果不存在这样的赋值，则对于所有可能的变量赋值，公式表示的函数为FALSE，并且公式不可满足。例如，公式“a AND NOT b”是可以满足的，因为可以找到值a = TRUE且b = FALSE，这使得（a AND NOT b）= TRUE。相反，“a AND NOT a”是不可满足的。” —— **百度百科**

   SAT问题可以被描述为一系列子句 S_i 的计算：

   $$\begin{cases} S=S_0\wedge S_1\wedge\cdots\wedge S_n \\ S_i=x_0^i\vee x_1^i\vee\cdots\vee x_m^i \end{cases}$$

   判断给定的解集合 $\left\{x_j^i:0\leq i\leq n,0\leq j\leq m,x_j^i\in\{true,false\}\right\}$ 是否能使得最终的计算结果为 `true` 。

   在SAT问题中，每一个子句中的文字的数量是不定的，而3SAT问题中要求每一个子句中包含三个不同文字，所以需要将SAT问题中的子句转换成3SAT中子句的形式（将不定长度的子句转换为长度为3的子句）。

文字数量	原子句（SAT）	转换后子句（3SAT）	变换说明
1	x_0	$x_0 \vee x_0 \vee x_0$	重复某一个文字，使得子句包含三个文字
2	$x_0 \vee x_1$	$x_0 \vee x_0 \vee x_1$	同上
3	$x_0 \vee x_1 \vee x_2$	$x_0 \vee x_1 \vee x_2$	不需要变换
4	$x_0 \vee x_1 \vee x_2 \vee x_3$	$(x_0 \vee x_1 \vee y_0) \wedge (\neg y_0 \vee x_2 \vee x_3)$	引入一个新的文字 y_0，设计新的子句形式使最终的结果不受新文字的影响
...	...	...	类似的，文字数量超过3个时，引入新的文字进行转换

显然，这个转换过程是多项式时间复杂度的。

如此SAT问题可以通过多项式时间归约到3SAT问题，3SAT问题是SAT问题的一个特殊实例，已知SAT问题是`NPC`的，所以3SAT问题也是`NPC`的。

> 参考：
>
> 1. **3-SAT NP-Complete 证明**

分治法

主要步骤

1. 分解（Divide）
2. 求解（Conquer）
3. 组合（Combine）

基于这三个步骤，计算出使用分治法求解问题的时间复杂度。

时间复杂度分析

代入法

类似于 《组合数学》 中使用迭代法求解递推关系，将递推式不断展开，直到能分析出其规律，**需要用数学归纳法进行证明。**

当表达式了包含`ceil`、`floor`时，在推导过程中可以假定参数 n 是整数次幂，整理结果时可以通过无穷级数进行缩放，得到通用的表达式。

递归树

思路与代入法相似，通过画出递归树，便于更好地确定**树的深度**与**每一层的计算开销**。

最终的时间为递归树中每一层时间开销之和。

> “递归树最是很用来生成好的猜测，然后即可用代入法来验证猜测是否正确” ——《算法导论》 P. 50

主方法

公式

$$T(n) = aT(n/b) + f(n)$$

表示将一个问题分成 a 个子问题，每个子问题的规模是 $\frac{n}{b}$ ，问题分解与组合的时间为 $f(n)$ 。

子问题数量 a 与子问题规模比例 b 不一定相等：

- 不同子问题所使用的数据之间可能有重叠，此时 $a \geq b$ （Strassen 矩阵乘法，$a = 7$，$b = 2$，子问题基于拆分出来的 $A_{11}, A_{12}, B_{11}, B_{12}$ 进行 7 次加减法运算再用于乘法）；
- 子问题在原问题的基础上可能只选择处理一部分数据，此时 $a \leq b$ （二分搜索，$a = 1$，$b = 2$，子问题只需要选择性地处理一半的数据）。

$n^{\log_b a}$：表示求解子问题所花费所有时间。

> 通过递归树进行理解，树的最底层深度为 $\log_b n$ ，最底层共有 $a^{log_b n}$ 个叶子节点，每个叶子节点的代价为**常数** $T(1)$，通过 **换底公式** 可得**子问题的求解时间**为： $a^{log_b n} \cdot T(1) = a^{log_b n} = n^{\log_b a}$ 。
>
> > 通过换底自然对数进行公式证明：
> >
> > $$a^{\log_b n} = e^{\ln a \cdot \log_b n} = e^{\ln a \cdot \ln n / \ln b} = n^{\ln a / \ln b} = n^{\log_b a}$$
>
> 参考《算法导论》P. 89。

三种情况

将求解子问题的时间与分解组合的时间进行比较，对于一个大于 0 的常数 ϵ ：

1. $f(n) = O(n^{(\log_b a) - \epsilon})$ ： $T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$ ： $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$
3. $f(n) = \Omega(n^{(\log_b a) + \epsilon})$ 且对于 $c < 1$ 和足够大的 n 有 $af(n/b) \leq cf(n)$ ： $T(n) = \Theta(f(n))$

> `Case 3` 中的 $af(n/b) \leq cf(n)$ 约束，表明要求满足“正则”条件。
>
> 参考《算法导论》P. 54。

分别表示 $f(n)$ 与 $n^{\log_b a}$ 在数量级上的小于、等于、大于关系时的时间复杂度。

三种情况存在间隙

需要注意 ϵ 的存在： `Case 1` 和 `Case 3` 的条件存在间隙，相差了一个 n^{ϵ} 因子，该因子可通过求 $f(n)$ 与 $n^{\log_b a}$ 的比值与 n^{ϵ} 比较进行判断是否存在（参考练习2）。

函数 $f(n)$ 落在这些间隙之间时，不能采用主方法进行求解。

练习

1. **分析Strassen算法的时间复杂度。**

   对于 $n \times n$ 的矩阵算法的递归式为 $T(n) = \begin{cases} \Theta(1), n = 1 \\ 7T(n/2) + \Theta(n^2), n > 1 \end{cases}$。

   在该问题中，有 $a = 7, b = 2, f(n) = n^2$，求解子问题的时间为 $n^{\log_2 7} \approx n^{2.8}$，$n^{\log_b a}/f(n) \approx n^{0.8}$，未落入 `Case1` 的间隙，属于第一种情况，其中 $\epsilon \approx 0.8$。

   所以其时间复杂度 $T(n) = \Theta(n^{\log_2 7})$。

   > 这里的做法没有使用 $\lg 7$ 进行缩放，相关做法可参考《算法导论》P. 55。

2. **分析递归式 $T(n) = 2T(n/2) + n \lg n$ 是否可用主方法求解。**

   $a = 2, b = 2, f(n) = n \lg n, n^{\log_2 2} = n$，但是 $f(n)/n^{\log_b a} = \lg n$，$\lg n$ 渐进小于 n^ϵ，**即无法找到 $\epsilon > 0$ 使得 `Case 3` 条件成立**，所以该递归式属于落入了 `Case 3` 之间的间隙的情况，无法使用主方法进行求解。

   > 可参考《算法导论》P. 54。

DP

主要步骤

1. 描述最优解的结构特征
2. 递归地定义一个最优解的**值**
3. 自底向上计算一个最优解的**值**
4. 从已计算的信息中构造一个最优解

> 摘自 lec6.pdf。

最优子结构

原问题的最优解包含子问题的最优解。

> 这里需要注意的是：不是所有的子问题的最优解都能用于计算出原问题的最优解。
>
> 譬如在矩阵链乘问题中，对于某一个规模为 n 的原问题，一共可以分解出 $\binom{n}{2}$ 个子问题，这些子问题的最优解可能会**相互比较用于选择**，但**不是都会用于原问题最优解的计算**。
>
> 可参考《算法导论》P. 243

Cut and Paste + 反证法

用于证明最优子结构的正确性。

即如果不采用“最优子结构”所选择的解，那么可以将这个解“cut”掉，并将通过最优子结构所得到的最优解“paste”上去，从而得到一个更优的解。由此，不使用最优子结构的解不会比使用最优子结构的解更好，即使用最优子结构可以得到最优解。

> "假定在最优解对应的子问题的解非最优，删去它换上最优解，得到原问题的解非最优，矛盾！"
> —— lec6.pdf

BU、TD

BU（Bottom-up，自底向上）、TD（Top-down，自顶向下），两种DP方法在时间上是相似的。

BU

- 直接从已经划分好的最小子问题开始求解。
- 在编程时表现为**对于一个已经划分好且完成初始化的多维数组进行填充**，数组中某一项往往依赖于对应几个子问题所在的数组项进行比较选择，然后计算。
- 原问题的解通常是最后计算的。

TD

- 也称为带备忘的DP。
- **编程时类似于DFS**，从原问题开始，通过递归与回溯进行求解。只不过**对于已求解过的问题，会将该问题的解进行保存**，在后续再次遇到该问题时便不需要再次递归求解而是直接查表（处理重叠子问题）。
- 最先接触的是原问题，但是解也是最后计算出来的，中间会进行各种子问题的递归求解。

练习

1. **分析矩阵链乘问题，并构造最优解。**

   问题描述：

   给定n个矩阵的大小 $p_0, p_1, \cdots, p_n$ （第 i 个矩阵的列数和第 $i+1$ 的行数是相同的，所以用同一个符号表示，第 i 个矩阵 A_i 大小为 $p_{i-1} \times p_i$），给定其最优的括号化方案使得最终的计算时间最小。

   最优子结构分析：

   子问题共有 $\binom{n}{2}$ 个，即在这 $n+1$ 个数中任意选取两个 $i, j\,(0 < i < j)$，表示 $[i, j]$ 区间的矩阵 $A_i A_{i+1} \cdots A_j$ 链乘的子问题。用 $p(i, j)$ 表示区间 $[i, j]$ 内的矩阵链乘问题的时间。

   假设 $[i, j]$ 子问题的**最优**分割点为 k，则产生了两个子问题 $[i, k]$、$[k+1, j]$ 对这两个子问题如果不独立使用最优括号化的方案进行求解，那么则可以将使用最优求解方案计算的解代入 $[i-1, j]$ 的最优解中（cut and paste），如此便会得到比原来的“最优解”更优的解，与代入前的解是最优解相矛盾。

   所以，矩阵链乘问题具有最优子结构。

   算法分析：

   每一个长度不为 1 的矩阵链乘区间，都可以划分为两个更小的子问题，直至区间大小为 1。所以对于区间 $[i, j]$ 分析以下两种情况：

   1. 区间大小为1：此时只有一个矩阵，无需计算，即 $p(i, j) = 0$。
   2. 区间大小不为1：此时需要找到一个分割点 $k, i \le k \le j$，使得 $p(i, k) + p(k+1, j) + p_i p_k p_j$ 最小，即：

      $$\arg\min_k (p(i, k) + p(k+1, j) + p_i p_k p_j)$$

      找到最优的分割点 k 以后，有：

      $$p(i, j) = (p(i, k) + p(k+1, j) + p_i p_k p_j)$$

算法需要做的，就是按照 $p(i,j)=\min_{i\le k\le j}\left(p(i,k)+p(k+1,j)+p_ip_kp_j\right)$ 不断计算和比较子问题，并组合得到最终的解。

BU算法实现：

定义一个 $n\times n$ 的二维数组 m，$m[i][j]$ 表示矩阵区间 $[i,j]$ 链乘的时间。

1. 初始化 $m[i][i]=0$；
2. 递推：对于 $i<j$ 有 $m[i][j]=\min_{i\le k\le j}\left(m[i,k]+p[k+1,j]+p_ip_kp_j\right)$；
3. 最终解为 $m[0][n]$。

编程实现时，主要就是 `for` 循环填表。

TD算法实现：

定义一个函数 $f(i,j)$ 表示求解矩阵区间 $[i,j]$ 链乘的时间，并在其中递归求解子问题，子问题求解以后回溯计算当前问题的解：

```
dp = dict() # 备忘录
INF = 1e10
p = [.....] # 矩阵规模数组
def f(i, j):
    # 二者相等
    if i == j:
        return 0
    # 查找备忘录
    if (i,j) in dp.values():
        return dp[(i,j)]
    # 递归求解子问题
    result = INF # 初始化为一个不可能取到的极大值，用于比较（具体要视问题规模而定）
    for k in range(i, j+1):
        result = min(result, f(i,k) + f(k+1, j) + p[i] * p[k] * p[j])
    dp[(i,j)] = result # 写入备忘录
    return result
```

贪心

使用条件：具有最优子结构。

主要步骤

- 一般直接求解思路是一个分解+两个证明：
    1. 将优化问题分解为做出一种选择及留下一个待解的子问题。
    2. 证明对于原问题总是存在一个最优解会做出贪心选择，从而保证贪心选择是安全的。
    3. 验证当做出贪心选择之后，它和剩余的一个子问题的最优解组合在一起，构成了原问题的最优解。

> 摘自 lec7.pdf。

第二步可以使用 Cut and Paste 思路进行证明：对于一个没使用贪心策略的最优解，将其中没使用贪心的那一部分替换成贪心的选择，如果替换后没有和原来剩下的一部分解冲突，则证明贪心是安全的。

第三步需要证明已有的解和另一个子问题的最优解不冲突。

- 另一种方法是使用拟阵求解，重点在于拟阵的构造与证明，基于**拟阵具有贪心选择的性质**。

> 拟阵具有贪心选择性质的证明可参考《算法导论》P. 252-253。

拟阵

定义

满足遗传性、扩充性的二元组 $\mathcal{M}=(S,I)$， S 是一个有限集合， I 是 S 的非空子集簇（集族）。

通俗来说 I 就是**满足某一类条件**的子集的集合，其中**每一个元素都是一个集合**，集合的收集“collection”。

> 在MST（最小生成树）问题中，对于图 $G=(V,E)$，可以构造拟阵 $\mathcal{M}=(S,I)$，其中 $S=E$，而 I 为 E 中**不包含环的边集簇**。具体的性质证明可以看 lec7.pdf 。

也就是说，对于拟阵的构造，I 中每一个元素所拥有的性质很重要。

加权拟阵

对于 $\mathcal{M}=(S,I)$，当 S 中的每一个元素 x 都包含一个**正的权值** $w(x)>0$ 时，就称 $\mathcal{M}=(S,I)$ 是加权拟阵。

S 中某个子集 A 的权值 $w(A)$ 即为 A 中所有元素的权值之和：$w(A)=\sum_{x\in A}w(x)$。

性质

- **遗传性：** I 中元素的子集一定也在 I 中。

  对于 $A\in I$，若有 $B\subseteq A$，则 $B\in I$。
- **扩充性：** I 中的某一个元素（I 中每个元素是一个集合）被 I 中其他集合扩充后依然在 I 中。

  对于 I 中的两个元素 A,B，满足 $|B|>|A|$，A 加入了 $B-A$ 中的元素后依然在 I 中。
- **最大独立子集：** I 中无法再被扩充的集合，所有最大独立子集大小相同。

证明

证明一个二元组是拟阵，只需**结合问题特征**来证明其满足**遗传性**和**扩充性**即可。

> 还是MST问题，这里的问题特征就是边不包含环。

应用

通过**加权拟阵的最大独立子集**来进行贪心求解。

求解思路：构造加权拟阵的最大独立子集时，每次选择权值最大的元素加入其中。

练习

1. **分数背包问题的算法设计与分析（《算法导论》练习 16.2-1）。**

   问题描述：

   给定背包容量 C，以及 n 个有重量 w_i 和价值 v_i 两个属性的物品集合 $(w_0,v_0),(w_1,v_1),\cdots,(w_{n-1},v_{n-1})$，每个物品可以拿取任意部分，要求给出背包能装下的物品分配方案的最大价值。

   贪心性质证明：

   - 直接证明（不一定正确）：

- 单一子问题：问题空间可以用 $P=(C,I)$ 进行描述，C 表示背包可用容量，I 表示剩余物品列表。对于选择了 i 物品后，子问题可被描述为 $P_i=(C-w_i,I-i)$，子问题对应选择方案的价值和重量分别为 V_i,W_i。
- 最优子结构（贪心安全）：假设 i 为 I 中单位价值 v_i/w_i 最大的物品，假设最优解中，没有选择物品 i，得到的价值为 V'，则可将最优解的选择方案中 $\min(C,w_i)$ 容量的最大单位价值物品替换为 i，替换后的价值 V。因为 i 单位价值最大，所以替换后的价值 $V\geq V'$（当有多个最大单位价值的物品且背包只能装下这些物品时可能会相等）。如果 $V>V'$ 则与此前的方案是最优解相矛盾，说明具有最优子结构；若 $V=V'$ 则说明替换后的依然为最优解，由此也证明了贪心算法是安全的。
- 组合得到最优解：在选取了单位价值最大的物品 i 后，可得到 v_i 的价值，对于剩余的子问题 P_i 的候选的物品不再包含 i，与贪心选择的物品 i 不矛盾。且子问题所选择的物品总重量 $W_i\leq C-w_i$，加上贪心选择物品 i 的重量 w_i，最终重量不会超过背包容量 C。所以贪心选择的方案与剩下子问题的最优解是相容的。

- 拟阵证明：
  - 预处理：将所有物品按照单位重量 e 拆分，得到新的物品集合。如单位 $e=1$ 时，一个物品为 $(3,6)$ 则拆分为 3 个 $(1,2)$ 的物品，如果无法整除则适当调整单位大小。此时得到新的背包容量为 $e\cdot C$。

    该“预处理”操作可在多项式时间内完成。
  - 构造带权拟阵：构造二元组 $\mathcal{M}=(S,I)$，S 为单位拆分后的新物品集合；$I=\{S_i:S_i\subseteq S,|S_i|\leq e\cdot C\}$，即为满足所有集合大小小于等于 $e\cdot C$ 的 S 非空子集簇；S 中元素的权值为其单位价值，$w(i)=e\cdot v_i/w_i$。
  - 证明遗传性：对于 $B\in I,A\subseteq B$，显然有 $|A|\leq|B|\leq e\cdot C$，所以 $A\in I$，满足遗传性。
  - 证明扩充性：对于 $B\in I,A\in I,|B|<|A|\leq e\cdot C$ 以及任意的 $i\in A-B$，将 i 扩容进 B 后有 $|B|+1\leq|A|\leq e\dot{C}$，即 $B\cup\{i\}\in I$，满足扩充性。
  - 由此，分数背包问题可用带权拟阵求解，满足贪心选择的性质。

算法设计：

```python
def f(items, cap):
    """
    items: [(w,v)] 物品列表
    cap：背包容量
    """
    items = sorted(lambda x: x[1]/x[0], items) # 按照单位价值排序
    value = 0 # 背包物品价值总和
    for i in items:
        # 可以全部装入背包
        if cap >= i[0]:
            value += i[1]
            cap -= i[0]
        # 只能装取部分
        else:
            value += cap*i[1]/i[0]
            cap = 0 # 背包装满了，结束循环
            break
    return value
```

近似

一些概念

近似比

设一个最优化问题的最优值为 C^*，该问题的近似算法求得了近似最优值 C，有近似比

$$\eta = max\left\{\frac{C^*}{C}, \frac{C}{C^*}\right\}\text{（最大化、最小化问题均适用）}$$

> 摘自 lec8.pdf。

k 近似算法：当一个算法的近似比为 k 时，也称这个算法为“ k 近似算法 ”。

PTAS、FPTAS

- `PTAS`：算法的时间取决于**问题的规模**。
- `FPTAS`：算法的时间取决于**问题的规模**与**近似比**。

> 参考《算法导论》P. 652。

二者相比，`FPTAS` 比 `PTAS` 更有实际应用意义，可以根据实际时间与近似比需求进行调整，比较灵活；而 `PTAS` 算法的时间与近似比是完全取决于问题本身的，比较“死板”。

伪多项式时间算法

算法的时间除了取决于问题的规模外，还取决于问题中每一项值的大小（如：数字位数）。

> 算法的复杂度与输入规模呈指数关系，与输入的数值大小呈多项式关系。
>
> ——摘自 **知乎：什么是伪多项式时间算法？- AaronHe的回答**（该回答前面的其他回答也值得参考）。

以01背包问题为例：

- 使用动态规划进行求解时，需要枚举背包在不同容量时的状态，当物品数量为 n、背包大小为 C 时，DP算法的数组大小为 $C \cdot n$。
- 且当物品的重量不是整数时，还要进行对背包大小和物品重量进行缩放，当整体放大 k 倍后使得所有物品重量为整数时，背包大小为 $k \cdot C$，算法的复杂度实际为 $k \cdot C \cdot n$。
- k 往往是 10 的幂（如物品重量精确到 10^{-3} 时，就有 $k = 10^3$），导致最终的时间也是一个指数级别的多项式。虽然是多项式，但计算的时间开销特别大，所以也叫“伪多项式时间算法”。

近似算法设计思路

对于给定的近似比 ϵ，`FPTAS` 的设计重点在于：

1. **对输入的值设计一个关于 ϵ 的近似策略**；
2. 再使用一种已知的**（伪）多项式算法**进行求解；
3. **需要证明算法的近似比。**

通过近似策略舍弃一部分精度，换取时间开销上的优化。

练习

1. **分析01背包问题的FPTAS。**

> 猜想：节点覆盖问题的近似算法是否可用来求解01背包?
>
> 理由：因为01背包问题是可以归约到节点覆盖问题的，从而意味着节点覆盖的 FPTAS 和 PTAS 可以直接用于处理01背包问题。

问题描述

给定 n 个物品 $a_1,\dots,a_n$ ，每个物品的重量和价值分别为 $weight(a_i),profit(a_i)$ ，背包容量为 C ，每种物品只有放入背包和不放入，设计一种策略使得放入背包的物品总重量不超过背包容量，且总价值最大。

近似策略

对于一个给定的近似比 ϵ ，用 $P=\max\{profit(a_i):i\in[1,n]\}$ 表示 n 个物品中最贵重物品的价值，令缩放的比例 K 为：

$$K=\frac{\epsilon\cdot P}{n}$$

对每一个物品的价值进行缩放和取整的近似处理：

$$profit'(a_i)=\left\lfloor\frac{profit(a_i)}{K}\right\rfloor$$

多项式算法

对于给定的 n 个物品，能取到的最大价值为 $n\cdot P$ 。

对于 $i\in[1,n],p\in[i,n\cdot P]$ 定义 $A(i,p)$ 表示前 i 个物品组合出价值为 p 时的重量，从前往后DP有：

$$A(i+1,p)=\begin{cases}\min\{A(i,p),A\left(i,p-profit(a_{i+1})\right)+weight(a_{i+1})\} & ,\text{if}\quad profit(a_{i+1})\le p\\ A(i,p) & ,\text{if}\quad profit(a_{i+1})>p\\ +\infty & ,\quad \text{不存在满足条件的方案}\end{cases}$$

最终的结果为：

$$\max\{A(i,p):p\le C,A(i,p)\neq+\infty\}$$

本算法只需填充 $n\times nP$ 的表格，故所需的时间为：$O(n^2P)$ 。

使用 $profit'(a_i)$ 作为物品的近似价值，并通过该伪多项式时间的DP算法进行求解。

证明

- 时间：将 $profit'(a_i)$ 代入 P 计算，$n^2\cdot\frac{P}{K}=n^2\cdot\lfloor\frac{n}{\epsilon}\rfloor$ ，时间为 $O(n^2\cdot\lfloor\frac{n}{\epsilon}\rfloor)$ 。
- 近似比：
  - 假设进行近似处理前的最优解方案为 O；
  - 由 $profit'(a)$ 的计算公式可知，对于任意物品 a 有 $K\cdot profit'(a)\le profit(a)$；

    以及 $profit'(a)>\frac{profit(a)}{K}-1$ （向下取整只是舍去小数部分，与原来的差距不会超过 1 ），即 $K\cdot profit'(a)>profit(a)-K$；
  - 整理上述两个公式得

$$profit(a) - K < K \cdot profit'(a) \leq profit(a)$$

- 对于单个物品 a 以及包含 n 个物品（扩大 n 倍）的最优方案 O，还能推导出

$$\begin{cases} K > profit(a) - K \cdot profit'(a) & ,对于单个物品\ a \\ n \cdot K > profit(O) - K \cdot profit'(O) & ,对于最优方案\ O \end{cases}$$

- 假设近似算法输出的选择方案为 S，经过一些列不等式的比较

$$\begin{cases} （在profit'的价值下DP求解的S是至少和O一样优的解） \\ profit'(S) \geq profit'(O) \\ \quad \Rightarrow K \cdot profit'(S) \geq K \cdot profit'(O) \\ \\ profit(S) \geq K \cdot profit'(S) > profit(S) - nK \end{cases}$$

$$\Rightarrow \quad profit(S) \geq K \cdot profit'(O)$$

- 再将有关 $profit'(O)$ 的不等式代入，得到

$$\begin{aligned} profit(S) \geq K \cdot profit'(O) > \\ profit(O) - n \cdot K = profit(O) - \epsilon \cdot P \\ = \mathbf{OPT} - \epsilon \cdot P \end{aligned}$$

- 因为 $\mathbf{OPT}$ 恒大于等于 P，即 $\mathbf{OPT} \geq P$，有

$$\mathbf{OPT} - \epsilon \cdot P \geq (1 - \epsilon)\,\mathbf{OPT}$$

- 代入 $profit(S)$ 得

$$profit'(S) \geq (1 - \epsilon)\mathbf{OPT}$$

- 证毕，该算法的近似比为 ϵ。

> 见 lec8.pdf P. 5。

2. **分析01背包问题的2近似算法（PTAS）。**

使用类似于分数背包的贪心算法进行求解。

算法思路

对 n 个物品（问题重量、价值、背包容量简记为 w_i, v_i, C）按照单位价值 v_i/w_i 排序，找到一个 k 使得前 k 个物品的价值最大的，且无法再将其他物品装入背包：$\sum_{i=1}^{k} w_i \leq C < \sum_{i=1}^{k+1} w_i$。

最终的近似解为：

$$\max\left\{\sum_{i=1}^{k} v_i, v_{k+1}\right\}$$

证明

假设最优解的价值为 $\mathbf{OPT}$，且第 $k+1$ 个物品是可以只拿取部分的（类似分数背包）。

将拿完前 k 个物品剩下的容量用来装部分 $k+1$ 个物品，此时的价值肯定是大于等于 $\mathbf{OPT}$，小于 $k+1$ 个物品全拿：

$$\mathbf{OPT} \leq \sum_{i=1}^{k} v_i + (C - \sum_{i=1}^{k} w_i)\frac{v_{k+1}}{w_{k+1}} < \sum_{i=1}^{k+1} v_i$$

这里的结论是：$\mathbf{OPT} < \sum_{i=1}^{k+1} v_i$ 。

下面需要用到一个数学定理：对于 $a > 0, b > 0$ 有 $\max\{a, b\} \geq \frac{1}{2}(a + b)$ 。

> 证明：对于 $a \geq 0, b \geq 0$ ，有 $2 \cdot \max\{a, b\} \geq a + b \geq \max\{a, b\}$ ，根据左边的不等式有 $\max\{a, b\} \geq \frac{1}{2}(a + b)$ 。

将算法得到的近似解应用于这个定理，有 $\max\left\{\sum_{i=1}^{k} v_i, v_{k+1}\right\} \geq \frac{1}{2}(\sum_{i=1}^{k} v_i + v_{k+1}) = \frac{1}{2}\sum_{i=1}^{k+1} v_i$ ，即：

$$\max\left\{\sum_{i=1}^{k} v_i, v_{k+1}\right\} \geq \frac{1}{2}\sum_{i=1}^{k+1} v_i$$

综合之前得到的 $\mathbf{OPT} < \sum_{i=1}^{k+1} v_i$ 有，$\max\left\{\sum_{i=1}^{k} v_i, v_{k+1}\right\} \geq \frac{1}{2}\sum_{i=1}^{k+1} v_i > \frac{1}{2}\mathbf{OPT}$ ，即：

$$\max\left\{\sum_{i=1}^{k} v_i, v_{k+1}\right\} > \frac{1}{2}\mathbf{OPT}$$

证毕，该算法为 2 近似算法。

3. **分析多机调度问题。**

问题描述

有 m 台处理机，n 个任务，每个任务有两个属性：任务到达时间、任务完成所需时间。类似于操作系统中的处理机调度问题。

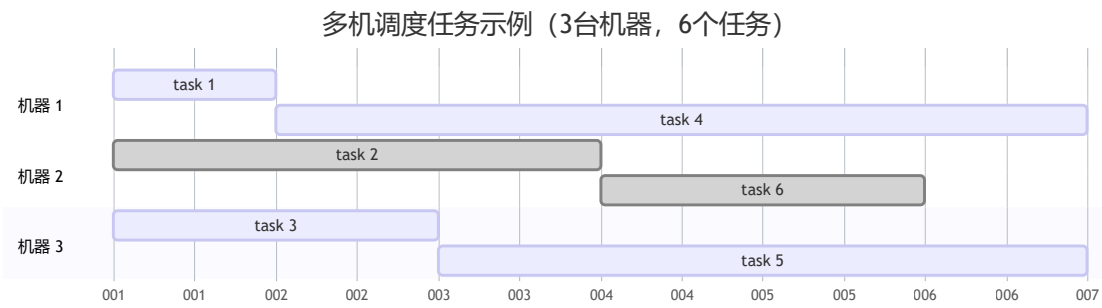
需要设计一个 在线算法 使得所有任务被处理完的总时间最小，需要注意的是无法在算法执行过程中知道后续到来的任务信息。

> 在计算机科学中，一个在线算法是指它可以以序列化的方式一个个的处理输入，也就是说在开始时并不需要已经知道所有的输入。相对的，对于一个 **离线算法** ，在开始时就需要知道问题的所有输入数据，而且在解决一个问题后就要立即输出结果。例如， **选择排序** 在排序前就需要知道所有待排序元素，然而 **插入排序** 就不必。——**百度百科：在线算法**

近似算法

使用贪心的策略，对于每一个到来的任务，分配给所有处理机中将最先完成已分配任务的一台。

如在下图例子中，假设任务到来的顺序是 $1, 2, 3, 4, 5, 6$ ， 则有如下图所示分配：



证明

> 见 lec8.pdf P. 6-7 。