算法设计与分析

动态规划 (III)

算法设计与分析

动态规划 (III)

投资回报问题

问题：给定 m 枚硬币，n 个项目，函数 $f_i(x)$ 表示在第 i 个项目上投资 x 的利润。

- 找到使利润最大化的最优投资方案。

解：向量 $(x_1, x_2, \ldots, x_n)$，x_i：在项目 i 上的投资

优化目标函数：$\max \sum_{i=1}^{n} f_i(x_i)$

约束条件：$x_1 + x_2 + \cdots + x_n = m$，$x_i \in \mathbb{N}$

表：5 枚硬币投资 4 个项目

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

子问题与计算顺序

子问题：由两个参数 k 和 x 定义

- k：投资前 $1, 2, \ldots, k$ 个项目
- 总投资不超过 x

矩阵链乘法中的参数是同类型的下标元组
(k, x) 是不同类型的 $\rightsquigarrow$ 二维动态规划

原问题：$k = n$，$x = m$
计算顺序：$k = 1, 2, \ldots, n$；对于任意 k，$x = 1, 2, \ldots, m$

- 可以用两层循环实现

优化函数的递推关系

优化函数 $\mathrm{OPT}_k(x)$：将 x 枚硬币投资在前 k 个项目上的最大利润
递推关系：从所有 $y \le x$ 的 $\mathrm{OPT}_{k-1}(y)$ 推导出 $\mathrm{OPT}_k(x)$

$$\mathrm{OPT}_k(x) = \max_{0 \le x_k \le x} \{f_k(x_k) + \mathrm{OPT}_{k-1}(x - x_k)\}, \quad k > 1$$

$$\mathrm{OPT}_1(x) = f_1(x), \quad k = 1$$

$k = 2$ 的演示

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	20
2	12	5	10	21
3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

$k = 1$ 对应初始值：$\mathsf{OPT}_1(1) = 11$，$\mathsf{OPT}_1(2) = 12$，$\mathsf{OPT}_1(3) = 13$，$\mathsf{OPT}_1(4) = 14$，$\mathsf{OPT}_1(5) = 15$

$\mathsf{OPT}_2(1) = \max\{f_1(1), f_2(1)\} = 11$

$\mathsf{OPT}_2(2) = \max\{f_2(2), \mathsf{OPT}_1(1) + f_2(1), \mathsf{OPT}_1(2)\} = 12$

$\mathsf{OPT}_2(3) = \max\{f_2(3), \mathsf{OPT}_1(1) + f_2(2), \mathsf{OPT}_1(2) + f_2(1), \mathsf{OPT}_1(3)\} = 16$

类似地，可以计算 $\mathsf{OPT}_2(4) = 21$，$\mathsf{OPT}_2(5) = 26$

备忘录与解

x	$\mathrm{OPT}_1(\cdot)$	$s_1(\cdot)$	$\mathrm{OPT}_2(\cdot)$	$s_2(\cdot)$	$\mathrm{OPT}_3(\cdot)$	$s_3(\cdot)$	$\mathrm{OPT}_4(\cdot)$	$s_4(\cdot)$
1	11	1	11	0	11	0	20	1
2	12	2	12	0	13	1	31	1
3	13	3	16	2	30	3	33	1
4	14	4	21	3	41	3	50	1
5	15	5	26	4	43	4	61	1

- $\mathrm{OPT}_k(x)$ 记录将 x 枚硬币投资在前 k 个项目上的最大利润
- $s_k(x)$ 记录在第 k 个项目上的投资额

$$
\begin{aligned}
&s_4(5) = 1 \Rightarrow x_4 = 1, s_3(5-1) = s_3(4)\\
&s_3(4) = 3 \Rightarrow x_3 = 3, s_2(4-3) = s_2(1)\\
&s_2(1) = 0 \Rightarrow x_2 = 0, s_1(1-0) = s_1(1)\\
&s_1(1) = 1 \Rightarrow x_1 = 1
\end{aligned}
$$

解：$(x_1 = 1, x_2 = 0, x_3 = 3, x_4 = 1)$，$\mathrm{OPT}_4(5) = 61$

复杂度分析

备忘录表是一个 m 行（硬币总数）n 列（项目总数）的矩阵，共 mn 项：

$$\mathsf{OPT}_k(x) = \max_{0 \le x_k \le x} \{f_k(x_k) + \mathsf{OPT}_{k-1}(x - x_k)\}, \quad k > 1$$

$$\mathsf{OPT}_1(x) = f_1(x), \quad k = 1 \quad //$$

计算 $\mathsf{OPT}_k(x)$ 的代价：x_k 有 $x+1$ 种不同选择 $\Rightarrow$ $x+1$ 次加法 $+$ x 次比较

- 加法总次数：$\sum_{k=2}^{n} \sum_{x=1}^{m} (x+1) = \frac{1}{2}(n-1)m(m+3)$
- 比较总次数：$\sum_{k=2}^{n} \sum_{x=1}^{m} x = \frac{1}{2}(n-1)m(m+1)$

时间复杂度 $W(n) = O(nm^2)$，空间复杂度 $O(mn)$

目录

问题动机

在一次抢劫中，窃贼发现的赃物比预期的多得多，他必须决定拿走什么。

- 他的包（即" 背包"）最多能承载总重量 W 磅。
- 他想快速找出能装进背包的最有价值的物品组合。

这个问题有两个版本：

- **可重复**：每种物品的数量无限
- **不可重复**：每种物品只有一个（窃贼闯入了一个美术馆）

两个版本都不太可能有多项式时间算法。

正式动机

如果上述动机看起来有些荒谬

- 将" 重量" 替换为"CPU 时间"
- 将" 只能拿走 W 磅" 替换为" 只有 W 单位的 CPU 时间可用"

CPU 时间也可以替换为带宽

背包问题概括了各种资源受限的选择任务。

目录

可重复背包

问题：给定 n 个物品和一个背包，物品 i 的重量为 $w_i > 0$，价值为 $v_i > 0$，背包容量为 W
目标：装满背包使总价值最大化。

表：背包实例，$W = 11$

i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7

- 按价值贪心（最大 v_i 优先）：$\{28 \times 1, 6 \times 2\}$ 价值为 40
- 按重量贪心（最小 w_i 优先）：$\{1 \times 11\}$ 价值为 11
- 按比率贪心（最大比率 v_i/w_i 优先）：$\{28 \times 1, 6 \times 2\}$ 价值为 40

观察：以上贪心算法都不是最优的。

建模

解向量：$x = (x_1, x_2, \ldots, x_n) \in (\mathbb{N})^n$，$x_i$ 是物品 i 的数量
优化目标：$\max \sum_{i=1}^{n} v_i x_i$
约束条件：$\sum_{i=1}^{n} w_i x_i \leq W$

- **线性规划**：在线性约束下寻找优化函数的最小值或最大值
  - **整数规划**：当 x_i 为非负整数时的线性规划

寻找子问题

一如既往，动态规划的主要问题是：

什么是子问题

- 通常需要一些实验来确定什么方法有效。

对于可重复背包问题，我们可以用三种方式缩小原问题：

- 更小的背包容量 $w \leq W$
- 更少的物品（例如，物品 $1, 2, \ldots, j$，其中 $j \leq n$）
- 以上两者的组合

动态规划：失败的尝试

初始方案：限制物品编号，定义 $\mathrm{OPT}(j) =$ 在重量限制 W 下使用物品 $1, \ldots, j$ 能达到的最大价值。

情况 1：$\mathrm{OPT}(j)$ 不选择物品 j。

- 选择 $\{1, 2, \ldots, j-1\}$ 中的最优解 $\rightsquigarrow$ 满足最优子结构性质（通过交换论证证明）

情况 2：$\mathrm{OPT}(j)$ 选择物品 j

- 我们不知道选择物品 j 的后果，因为它会改变子问题的重量限制 $\rightsquigarrow$ 无法做出决策

我们还需要考虑容量的限制来引入更细粒度的子问题！

动态规划：添加新变量

定义 $\mathrm{OPT}_j(w) =$ 从物品 $\{1, \ldots, j\}$ 中选择且重量限制为 w 时的最大价值。

情况 1：$\mathrm{OPT}_j(w)$ 不选择物品 j

- $\mathrm{OPT}_j(w)$ 在重量限制 w 下选择 $\{1, 2, \ldots, j-1\}$ 中的最优解。

情况 2：$\mathrm{OPT}_j(w)$ 选择物品 j（至少 1 个）

- 新重量限制 $= w - w_j$。
- $\mathrm{OPT}_j(w)$ 在新重量限制下选择 $\{1, 2, \ldots, j\}$ 中的最优解（因为允许重复）

两种情况都满足最优子结构性质

整合

子问题：由两个变量 j 和 w 定义

- j：从 $\{1, 2, \ldots, j\}$ 的子集中选择
- w：容量（重量）限制

$\mathsf{OPT}_j(w)$：在重量限制 w 下从前 j 个物品中选择能达到的最大价值

计算顺序：$j = 1 \to n$；对于任意 j，$w = 1 \to W$

$$\begin{cases} \mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\} \\ \mathsf{OPT}_0(w) = 0, 0 \le w \le W, \quad \mathsf{OPT}_j(0) = 0, 0 \le j \le n \\ \text{定义} \mathsf{OPT}_j(w) = -\infty, 1 \le j \le n, w < 0 \end{cases}$$

- $\mathsf{OPT}_j(w - w_j) + v_j$：选择至少一个第 j 个物品时的最大价值
- 设置 $\mathsf{OPT}_j(w) = -\infty$ 当 $w < 0 \rightsquigarrow$ 不需要显式检查 $w - w_j \ge 0$

背包问题伪代码

Algorithm 1 Knapsack(n, W, $\{w_i\}_{i\in n}$, $\{v_i\}_{i\in n}$)

for $w = 0$ **to** W **do**
  $\mathsf{OPT}_0(w) \leftarrow 0$
end
for $j = 1$ **to** n **do**
  $\mathsf{OPT}_j(0) \leftarrow 0$
end
设置 $\mathsf{OPT}_j(w) \leftarrow -\infty$，对于 $1 \le j \le n$，$w < 0$
for $j = 1$ **to** n **do**
  for $w = 1$ **to** W **do**
    $\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\}$
  end
end

- 自底向上方法填充备忘录表
- 类似投资回报问题中减少备忘录表大小的技巧在这里也适用，但计算顺序略有不同

演示

表：背包实例，$n = 4$，$W = 10$

i	1	2	3	4
v_i	1	3	5	9
w_i	2	3	4	7

$\mathrm{OPT}_j(w)$ 的计算过程（提示：如何填充矩阵）

- 从左到右，从上到下
- 从上到下，从左到右

$j \setminus w$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	2	2	3	3	4	4	5
2	0	1	3	3	4	6	6	7	9	9
3	0	1	3	5	5	6	8	10	10	11
4	0	1	3	5	5	6	9	10	10	12

一点说明

替代优化函数：类似投资回报问题

$$\mathrm{OPT}_j(w) = \max_{0 \le x_j \le \lfloor w/w_j \rfloor} \{\mathrm{OPT}_{j-1}(w - x_j \cdot w_j) + x_j \cdot v_j\}$$

- 优点：更直观，易于理解
- 缺点：计算 $\mathrm{OPT}_j(w)$ 的复杂度依赖于 w，即需要 $\lfloor w/w_j \rfloor$ 次比较，而原优化函数只需要一次比较。

警告：可重复背包问题的改进优化函数不适用于投资回报问题，因为利润函数不是线性的。

经验

优化函数的设计至关重要

追踪函数

$s_j(w)$：解 $\mathrm{OPT}_j(w)$ 中最大的物品编号

$$s_j(w) = \begin{cases} s_{j-1}(w) & \mathrm{OPT}_{j-1}(w) > \mathrm{OPT}_j(w - w_k) + v_k \\ j & \mathrm{OPT}_{j-1}(w) \le \mathrm{OPT}_j(w - w_k) + v_k \end{cases}$$

$$s_1(w) = \begin{cases} 0 & w < w_1 \\ 1 & w \ge w_1 \end{cases}$$

追踪函数用于追踪解并输出详细信息

追踪解的伪代码

追踪解

表：$s_j(w)$

$j \setminus w$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	2	2	2	2	2	2	2	2
3	0	1	2	3	3	3	3	3	3	3
4	0	1	2	3	3	3	4	3	4	4

- $s_4(10) = 4 \Rightarrow x_4 = 1$
- $s_4(10 - w_4) = s_4(3) = 2 \Rightarrow x_4 = 1, x_3 = 0, x_2 = 1$
- $s_2(3 - w_2) = s_2(0) = 0 \Rightarrow x_2 = 1, x_1 = 0$

解：$x_1 = 0$，$x_2 = 1$，$x_3 = 0$，$x_4 = 1$，最大利润为 12。

复杂度分析

上述算法在 $\Theta(nW)$ 时间和 $\Theta(nW)$ 空间内解决具有 n 个物品和最大重量 W 的背包问题。

根据优化函数

$$\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w), \mathsf{OPT}_j(w - w_j) + v_j\}$$

- 备忘录计算：每个表项需要 $O(1)$ 时间，共有 $\Theta(nW)$ 个表项
- 回溯：最多 $\Theta(n + W)$ 步（想想为什么?）

总时间复杂度和空间复杂度为 $O(nW)$

备注

- 不是输入规模的多项式，因为对于整数 W，二进制表示需要 $\log W$ 位，因此输入规模是 n 和 $\log W \leftarrow$ 超多项式

再思考

我们真的必须使用二维动态规划吗?

[见原文第 27 页图片]

再思考

我们真的必须使用二维动态规划吗?
[见原文第 27 页图片]

考虑只对容量进行限制，定义：

$\mathrm{OPT}(w) =$ 容量为 w 的背包能达到的最大价值

再思考

我们真的必须使用二维动态规划吗?
[见原文第 27 页图片]

考虑只对容量进行限制，定义：

$$\mathrm{OPT}(w) = \text{容量为 } w \text{ 的背包能达到的最大价值}$$

如何用更小的子问题来表示?

- 如果 $\mathrm{OPT}(w)$ 的最优解包含物品 i，那么从背包中移除这个物品后剩下的是 $\mathrm{OPT}(w - w_i)$ 的最优解（得益于可重复性质）。
- 换句话说，$\mathrm{OPT}(w) = \mathrm{OPT}(w - w_i) + v_i$，对于某个 i。
  - 我们不知道是哪个 i，所以需要尝试所有可能。
  - 如果 $\mathrm{OPT}(w)$ 包含物品 i 和 j，则必有 $\mathrm{OPT}(w - w_i) + v_i = \mathrm{OPT}(w) = \mathrm{OPT}(w - w_j) + v_j$。

递推关系

算法现在呼之欲出 $\rightsquigarrow$ 极其简洁优雅

Algorithm 2 Knapsack(n, W, $\{w_i\}_{i\in n}$, $\{v_i\}_{i\in n}$)

1: $\mathsf{OPT}(0) \leftarrow 0$ // 空背包的最大值为 0
2: **for** $w = 1$ **to** W **do**
3: $\quad \mathsf{OPT}(w) = \max_{i:w_i\le w}\{\mathsf{OPT}(w - w_i) + v_i\}$
4: **end**
5: **return** $\mathsf{OPT}(W)$

该算法从左到右填充长度为 $W+1$ 的一维表

- 空间复杂度：$O(W)$
- 时间复杂度：每个表项最多需要 $O(n)$ 时间计算 $\Rightarrow$ 总运行时间为 $O(nW)$。

仔细思考

一如既往，存在一个底层的 DAG。尝试构造它，你会得到一个惊人的洞见

- 背包问题的这个特定变体归结为在 DAG 中寻找最长路径

[见原文第 29 页图片]

目录

不可重复背包

如果不允许重复呢?

之前那个巧妙的子问题定义现在完全没用了。

- 例如，知道 OPT$(w - w_j)$ 形式的值并不能帮助做进一步决策，因为我们不知道物品 j 在这个部分解中是否已经被使用。

不可重复背包

如果不允许重复呢?

之前那个巧妙的子问题定义现在完全没用了。

- 例如，知道 $\mathrm{OPT}(w - w_j)$ 形式的值并不能帮助做进一步决策，因为我们不知道物品 j 在这个部分解中是否已经被使用。

我们必须细化子问题以携带关于已使用物品的额外信息 $\rightsquigarrow$ 像之前可重复背包问题的第一种方法一样，添加另一个参数 $0 \le j \le n$:

$$\mathrm{OPT}_j(w) = \text{使用物品 } \{1, \ldots, j\} \text{ 且重量限制为 } w \text{ 时的最大价值}$$

我们要求的答案是 $\mathrm{OPT}_n(W)$。

递推关系

如何用更小的子问题来表示 $\mathsf{OPT}_j(w)$?
很简单：物品 j 要么需要以达到最优值，要么不需要。

$$\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w - w_j) + v_j, \mathsf{OPT}_{j-1}(w)\}$$

换句话说，我们用子问题 $\mathsf{OPT}_{j-1}(\cdot)$ 来表示 $\mathsf{OPT}_j(w)$。

伪代码

Algorithm 3 Knapsack$(n, W, \{w_i\}_{i\in[n]}, \{v_i\}_{i\in n})$

1: $\mathsf{OPT}_0(w) \leftarrow 0$，对于 $w \in [0, W]$；$\mathsf{OPT}_j(0) = 0$，对于 $j \in [0, n]$
2: 设置 $\mathsf{OPT}_j(w) = -\infty$，对于 $w < 0$
3: **for** $j = 1$ **to** n **do**
4: **for** $w = 1$ **to** W **do**
5: $\mathsf{OPT}_j(w) = \max\{\mathsf{OPT}_{j-1}(w - w_j) + v_j, \mathsf{OPT}_{j-1}(w)\}$
6: **end**
7: **end**
8: **return** $\mathsf{OPT}_n(W)$

- 该算法填充一个二维表，有 $W+1$ 行和 $n+1$ 列。每个表项只需要常数时间。运行时间保持不变：$O(nW)$。

记忆化

在动态规划中，我们写出一个递归公式，用更小的问题表示较大的问题，然后用它以自底向上的方式填充解值表，从较小的子问题到最大的。
该公式也提示了一个递归算法。
正如我们之前所见，朴素递归可能极其低效，因为它一遍又一遍地解决相同的子问题。
那么更智能的递归实现呢？一个能记住之前调用从而避免重复的实现？

记忆化

对于可重复背包问题，递归算法可以使用键值映射来存储已经计算过的 $\mathrm{OPT}(\cdot)$。

- 在每次请求某个 $\mathrm{OPT}(w)$ 的递归调用时，算法首先检查答案是否已经在 KV 映射中，只有不在时才继续计算。
- 这个技巧叫做**记忆化**。
- 注意：KV 映射可以用数组或哈希表实现，取决于键的数据类型和分布。

复杂度：递归算法从不重复子问题 $\rightsquigarrow$ 运行时间为 $O(nW)$，与动态规划相同。

- 然而，大 O 符号中的常数因子明显更大，因为递归的开销。

动态规划的缺点

在某些情况下，记忆化更有优势。

- 动态规划自动解决每个可能需要的子问题，而记忆化只解决那些实际需要的。

动态规划的缺点

在某些情况下，记忆化更有优势。

- 动态规划自动解决每个可能需要的子问题，而记忆化只解决那些实际需要的。

例如，假设 W 和所有重量 w_i 都是 100 的倍数。那么如果 100 不能整除 w，子问题 $\text{OPT}_j(w)$ 就是无用的

- DP 总是计算所有表项。
- 记忆化递归算法永远不会查看这些多余的表项。

动态规划的缺点

在某些情况下，记忆化更有优势。

- 动态规划自动解决每个可能需要的子问题，而记忆化只解决那些实际需要的。

例如，假设 W 和所有重量 w_i 都是 100 的倍数。那么如果 100 不能整除 w，子问题 $\mathrm{OPT}_j(w)$ 就是无用的

- DP 总是计算所有表项。
- 记忆化递归算法永远不会查看这些多余的表项。

DP 和递归算法的最坏情况复杂度仍然相同，但后者在某些实例上可能表现更好，因为它的执行依赖于实例。

背包问题的扩展

背包问题的判定版本是 $\mathcal{NP}$-完全的。
存在一个多项式时间算法，能产生一个可行解，其值在最优值的 1% 以内。

背包问题的变体

- 带物品数量约束的背包：第 i 个物品的最大数量是 n_i
  - 0-1 背包：$x_i = 0, 1$；$i \in [n]$
- 多背包：m 个背包，背包 i 的重量限制是 W_i，$i \in [m]$。
- 二维背包：每个物品有重量 w_i 和体积 t_i，$i \in [n]$，重量限制是 W，体积限制是 V

目录

最长公共子序列

设 $X = (x_1, x_2, \ldots, x_m)$ 和 $Z = (z_1, z_2, \ldots, z_n)$ 是两个字符串。如果存在一个严格递增的下标序列 $(i_1, \ldots, i_k)$ 使得对所有 $k \in [n]$ 都有 $z_k = x_{i_k}$，则称 Z 是 X 的子序列。
公共子序列：既是 X 的子序列，也是 Y 的子序列。
问题：找出 $X = (x_1, x_2, \ldots, x_m)$ 和 $Y = (y_1, y_2, \ldots, y_n)$ 的最长公共子序列。

示例

- X: A B C B D A B
- Y: B D C A B A

LCS: B C B A，长度为 4

暴力算法

假设 $m \leq n$，$|X| = m$，$|Y| = n$
暴力算法：对于 X 的每个子序列，检查该子序列是否出现在 Y 中

复杂度分析

- 检查一个候选子序列是否是给定字符串的子序列需要 $O(n)$ 时间
  - 思考如何实现？提示：使用两个指针顺序扫描两个字符串（每次比较后至少有一个指针向前移动，因此最大比较次数为 $2n$）
- X 中共有 2^m 个子序列

复杂度：$O(n2^m)$

动态规划：子问题

引入 i 和 j 来定义子问题

- X 的右边界是 i，Y 的右边界是 j
- $X_i = (x_1, x_2, \ldots, x_i)$，$Y_j = (y_1, y_2, \ldots, y_j)$

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$， $Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$， $Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

- $x_m = y_n \Rightarrow z_k = x_m = y_n$， $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$，$Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

- $x_m = y_n \Rightarrow z_k = x_m = y_n$，$Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$（以下一种或两种情况发生）

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$，$Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

- $x_m = y_n \Rightarrow z_k = x_m = y_n$，$Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$（以下一种或两种情况发生）
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$， $Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

- $x_m = y_n \Rightarrow z_k = x_m = y_n$， $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$（以下一种或两种情况发生）
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$
  - $z_k \neq y_n \Rightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$

问题与子问题的关系

$X_m = (x_1, x_2, \ldots, x_m)$， $Y_n = (y_1, y_2, \ldots, y_n)$
设 $Z_k = (z_1, z_2, \ldots, z_k) = \mathsf{LCS}(X_m, Y_n)$
考虑以下情况：

- $x_m = y_n \Rightarrow z_k = x_m = y_n$， $Z_{k-1} = \mathsf{LCS}(X_{m-1}, Y_{n-1})$
- $x_m \neq y_n$（以下一种或两种情况发生）
  - $z_k \neq x_m \Rightarrow Z_k = \mathsf{LCS}(X_{m-1}, Y_n)$
  - $z_k \neq y_n \Rightarrow Z_k = \mathsf{LCS}(X_m, Y_{n-1})$

满足最优子结构

优化函数与递推关系

优化函数：$L(i,j)$

- $X_i = (x_1, x_2, \ldots, x_i)$ 和 $Y_j = (y_1, y_2, \ldots, y_j)$ 的 LCS 长度

递推关系

$$L(i,j) = \begin{cases} 0 & i = 0 \vee j = 0 \\ L(i-1, j-1) + 1 & i, j > 0 \wedge x_i = y_j \\ \max\{L(i, j-1), L(i-1, j)\} & i, j > 0 \wedge x_i \neq y_j \end{cases}$$

注意：在最后一种情况下，我们不知道哪种情况发生，因此选择最大值（如果两个值相等，则两种情况都构成解）。

指示函数

指示函数 $s(i,j)$ 的取值：$\nwarrow$，$\leftarrow$，$\uparrow$

指示函数

指示函数 $s(i,j)$ 的取值：$\nwarrow$，$\leftarrow$，$\uparrow$

- $L(i,j) = L(i-1,j-1)+1$：$\nwarrow$

指示函数

指示函数 $s(i,j)$ 的取值：$\nwarrow$，$\leftarrow$，$\uparrow$

- $L(i,j) = L(i-1,j-1)+1$：$\nwarrow$
- $L(i,j) = L(i,j-1)$：$\leftarrow$

指示函数

指示函数 $s(i,j)$ 的取值：$\nwarrow$，$\leftarrow$，$\uparrow$

- $L(i,j) = L(i-1,j-1)+1$：$\nwarrow$
- $L(i,j) = L(i,j-1)$：$\leftarrow$
- $L(i,j) = L(i-1,j)$：$\uparrow$

LCS 伪代码

Algorithm 4 LCS($X[m]$, $Y[n]$)

1: $L(i,0) \leftarrow 0$, $i \in [m]$; $L(0,j) \leftarrow 0$, $j \in [n]$
2: **for** $i \leftarrow 1$ **to** m **do**
3: **for** $j \leftarrow 1$ **to** n **do**
4: **if** $X[i] = Y[j]$ **then**
5: $L(i,j) = L(i-1,j-1) + 1$, $s(i,j) \leftarrow (\nwarrow)$
6: **else**
7: **if** $L(i-1,j) \geq L(i,j-1)$ **then**
8: $L(i,j) \leftarrow L(i-1,j)$, $s(i,j) \leftarrow (\uparrow)$
9: **else**
10: $L(i,j) \leftarrow L(i,j-1)$, $s(i,j) \leftarrow (\leftarrow)$
11: **end**
12: **end**
13: **end**
14: **end**

回溯解

Algorithm 5 TrackLCS(s, m, n)

输出: X 和 Y 的 LCS

```
1:  while m ≠ 0 ∧ n ≠ 0 do
2:      if s(m, n) = (↖) then
3:          输出 X[m]; m = m − 1, n = n − 1, continue
4:      end
5:      if s(m, n) = (↑) then
6:          m = m − 1, continue
7:      end
8:      if s(m, n) = (←) then
9:          n = n − 1, continue
10:     end
11: end
```

指示函数示例

$X = (A, B, C, B, D, A, B)$， $Y = (B, D, C, A, B, A)$

	1	2	3	4	5	6
1	↑	↑	↑	↖	←	↖
2	↖	←	←	↑	↖	←
3	↑	↑	↖	←	↑	↑
4	↑	↑	↑	↑	↖	←
5	↑	↑	↑	↖	↑	←
6	↑	↑	↑	↖	↑	↖
7	↑	↑	↑	↑	↑	↑

解：$\mathsf{LCS} = (X[2], X[3], X[4], X[6]) = (B, C, B, A)$

复杂度分析

优化函数的计算

- 初始化：$O(m+n)$
- 计算：每次循环需要 ≤ 2 次比较，复杂度为 $\Theta(mn)$

指示函数的计算

- 计算：$\Theta(mn)$
- 回溯解：$\Theta(m+n)$（每步将 X 或/和 Y 的规模减少 1）

总时间复杂度：$\Theta(mn)$
空间复杂度：$\Theta(mn)$

进一步讨论

标准 LCS 问题

- 动态规划：$\Theta(nm)$
- 广义后缀树：$\Theta(n+m)$

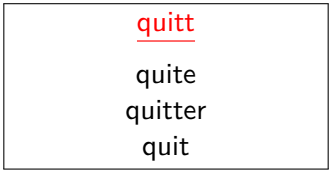
广义 LCS 问题：求 k 个字符串（长度分别为 $n_1, \ldots, n_k$）的 LCS

- k 维动态规划：$\Theta(n_1 \cdots n_k)$
- 广义后缀树：$\Theta(n_1 + \cdots + n_k)$

目录

字符串相似度的动机

当拼写检查器遇到可能的拼写错误时，它会在字典中查找相近的其他单词。



对于两个字符串，什么是合适的接近度或相似度概念?

编辑距离

编辑距离 [Levenshtein 1966, Needleman-Wunsch 1970]
给定两个字符串 x 和 y，经过一系列操作（替换、插入、删除）将 y 变为 x。最少的操作次数称为 x 和 y 之间的编辑距离，记为 $\Delta(x, y)$。

刻画两个字符串之间的相似度

定义的合理性：满足距离的三条规则

- 非负性：$\Delta(x, y) \geq 0$。$\Delta(x, y) = 0$ 当且仅当 $x = y$
- 对称性：$\Delta(x, y) = \Delta(y, x)$（只需逆转操作）
- 三角不等式：$\forall x, y, z$，$\Delta(x, z) + \Delta(z, y) \geq \Delta(x, y)$
  - $x \to z \to y$ 是从 x 到 y 的一条路径

如何计算编辑距离

编辑距离的理解

两个字符串之间的编辑距离是它们最佳对齐的代价。

- 找编辑距离等价于找最优对齐。

编辑距离之所以这样命名，是因为它也可以被认为是将第一个字符串转换为第二个字符串所需的最少编辑次数——插入、删除和替换。

- 上述示例：插入 ‘U’，将 ‘O’ 替换为 ‘N’，删除 ‘W’

编辑距离的理解

两个字符串之间的编辑距离是它们最佳对齐的代价。

- 找编辑距离等价于找最优对齐。

编辑距离之所以这样命名，是因为它也可以被认为是将第一个字符串转换为第二个字符串所需的最少编辑次数——插入、删除和替换。

- 上述示例：插入 ‘U’，将 ‘O’ 替换为 ‘N’，删除 ‘W’

一般来说，两个字符串之间有很多可能的对齐方式；逐一搜索所有对齐来找最优解是非常低效的。

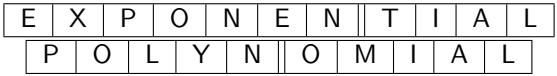
动态规划解法

用动态规划解决问题时，最关键的问题是：

什么是子问题?

只要子问题的选择满足最优子结构，写算法就很容易：按规模递增的顺序迭代地一个接一个解决子问题。

目标：找出两个字符串 $x[1\ldots m]$ 和 $y[1\ldots n]$ 之间的编辑距离 $E(m,n)$。
子问题：考虑 $x[1\ldots i]$ 的某个前缀和 $y[1\ldots j]$ 的某个前缀之间的编辑距离，称这个子问题为 $E(i,j)$。



子问题 $E(7,5)$

问题结构

我们需要用更小的子问题来表达 $E(i,j)$。分析 $x[1\ldots i]$ 和 $y[1\ldots j]$ 之间最佳对齐的最右列只能是以下三种情况之一：

$$\begin{array}{ccc} x_i & \perp & x_i \\ \perp & y_j & y_j \end{array}$$

情况 1a：x_i 不匹配

- x_i 的空位代价 + 对齐 $x[1\ldots i-1]$ 和 $y[1\ldots j]$ 的最小代价

情况 1b：y_j 不匹配

- y_j 的空位代价 + 对齐 $x[1\ldots i]$ 和 $y[1\ldots j-1]$ 的最小代价

情况 2：M 将 x_i 与 y_j 匹配

- $x_i \sim y_j$ 的（不）匹配代价 + 对齐 $x[1\ldots i-1]$ 和 $y[1\ldots j-1]$ 的最小代价

满足最优子结构性质

优化函数的递推关系

优化函数：$E(i,j)$ ——$x[1,\ldots,i]$ 和 $y[1,\ldots,j]$ 之间的编辑距离
初始值：$E(i,0)=i$，$E(0,j)=j$
递推关系：我们将 $E(i,j)$ 表示为三个更小子问题 $E(i-1,j)$、$E(i,j-1)$、$E(i-1,j-1)$ 的函数。

- 我们不知道哪个是正确的，所以需要尝试所有可能并选择最优的

$$E(i,j)=\min\{1+E(i-1,j),1+E(i,j-1),\mathsf{diff}(i,j)+E(i-1,j-1)\}$$

$$\mathsf{diff}(i,j)=\begin{cases}0 & x_i=y_j\\ 1 & x_i\neq y_j\end{cases}$$

计算顺序

所有子问题 $E(i,j)$ 的答案构成一个二维表。

这些子问题应该以什么顺序求解?

任何顺序都可以，只要 $E(i-1,j)$、$E(i,j-1)$ 和 $E(i-1,j-1)$ 在 $E(i,j)$ 之前处理。

1. 逐行填表，从上到下，每行从左到右
2. 或者逐列填表

这两种方法都能确保在计算某个表项时，所有需要的其他表项都已填好。

计算顺序示意图

[见原文第 59 页的计算顺序示意图]

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$E(5,5) \leftarrow E(4,4) + 0 \quad \mathsf{Y} \leftrightarrow \mathsf{Y}$

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$E(5,5) \leftarrow E(4,4) + 0 \quad \mathsf{Y} \leftrightarrow \mathsf{Y}$

$E(4,4) \leftarrow E(4,3) + 1 \quad \mathsf{W} \leftrightarrow \perp$

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$E(5,5) \leftarrow E(4,4) + 0$ Y $\leftrightarrow$ Y

$E(4,4) \leftarrow E(4,3) + 1$ W $\leftrightarrow \perp$

$E(4,3) \leftarrow E(3,2) + 1$ O $\leftrightarrow$ N

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0 \quad \text{Y} \leftrightarrow \text{Y}$$
$$E(4,4) \leftarrow E(4,3) + 1 \quad \text{W} \leftrightarrow \perp$$
$$E(4,3) \leftarrow E(3,2) + 1 \quad \text{O} \leftrightarrow \text{N}$$
$$E(3,2) \leftarrow E(2,1) + 0 \quad \text{N} \leftrightarrow \text{N}$$

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0 \quad \text{Y} \leftrightarrow \text{Y}$$
$$E(4,4) \leftarrow E(4,3) + 1 \quad \text{W} \leftrightarrow \perp$$
$$E(4,3) \leftarrow E(3,2) + 1 \quad \text{O} \leftrightarrow \text{N}$$
$$E(3,2) \leftarrow E(2,1) + 0 \quad \text{N} \leftrightarrow \text{N}$$
$$E(2,1) \leftarrow E(1,1) + 1 \quad \perp \leftrightarrow \text{U}$$

回溯解

x		S	N	O	W	Y
y	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1	1	2	3	4
N	3	2	1	2	3	4
N	4	3	2	2	3	4
Y	5	4	3	3	3	3

$$E(5,5) \leftarrow E(4,4) + 0 \quad \mathsf{Y} \leftrightarrow \mathsf{Y}$$
$$E(4,4) \leftarrow E(4,3) + 1 \quad \mathsf{W} \leftrightarrow \perp$$
$$E(4,3) \leftarrow E(3,2) + 1 \quad \mathsf{O} \leftrightarrow \mathsf{N}$$
$$E(3,2) \leftarrow E(2,1) + 0 \quad \mathsf{N} \leftrightarrow \mathsf{N}$$
$$E(2,1) \leftarrow E(1,1) + 1 \quad \perp \leftrightarrow \mathsf{U}$$
$$E(1,1) \leftarrow E(0,0) + 0 \quad \mathsf{S} \leftrightarrow \mathsf{S}$$

底层的 DAG

每个动态规划都有一个底层的 DAG。

- 每个节点代表一个子问题
- 每条边代表一个优先约束
- 除绿色边外，所有边长度设为 1
- 最终答案是从 $E(0,0)$ 到 $E(m,n)$ 的最短路径
  - 向下移动：删除
  - 向右移动：插入
  - 对角移动：匹配或替换

通过改变 DAG 上的权重，我们可以允许编辑距离的一般化形式：插入、删除和替换有不同的关联代价。

伪代码

Algorithm 6 SequenceAlignment($x[m], y[n]$)

for $i = 0$ **to** m **do**
  $E(i, 0) = i$
end
for $j = 0$ **to** n **do**
  $E(0, j) = j$
end
for $i = 1$ **to** m **do**
  for $j = 1$ **to** n **do**
    $E(i, j) \leftarrow \min\{1 + E(i-1, j), 1 + E(i, j-1), \mathsf{diff}(i, j) + E(i-1, j-1)\}$
  end
end
return $E(m, n)$

共有 mn 个子问题，每个子问题需要常数时间 $\Rightarrow$ 总共 $\Theta(mn)$ 时间和 $\Theta(mn)$ 空间。

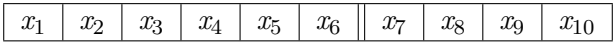
目录

如何找子问题

找到正确的子问题需要创造力和尝试。
但有一些标准的选择方法在动态规划中反复出现。

一维动态规划

输入是 $x_1, x_2, \ldots, x_n$。子问题是 $x_1, x_2, \ldots, x_i$



子问题的数量因此是 $O(n)$。

示例

- DAG 中的最短路径
- 最长递增子序列
- 最大区间和
- 图像压缩

二维动态规划：类型 1

输入是 $x_1, \ldots, x_n$。子问题是 $x_i, \ldots, x_j$

$$x_1 \quad x_2 \mid x_3 \mid x_4 \mid x_5 \mid x_6 \quad x_7 \mid x_8 \mid x_9 \mid x_{10}$$

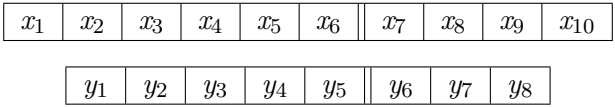
子问题的数量因此是 $O(n^2)$。

示例

- 矩阵链乘法
- 最优二叉搜索树

二维动态规划：类型 2

输入是 $x_1, \ldots, x_n$ 和 $y_1, \ldots, y_m$。子问题是 $x_1, \ldots, x_i$ 和 $y_1, \ldots, y_j$



子问题的数量因此是 $O(mn)$。

示例

- 投资回报
- 背包问题
- 最长公共子序列
- 编辑距离

动态规划的另一个重要特征

动态规划算法的计算复杂度不仅取决于子问题的数量，还取决于递推关系的复杂度，即一个问题与其子问题关联的程度。
我们用局部性来刻画这种依赖。

情况 1：依赖于线性数量的子问题

- DAG 最短路径、最长递增子序列、最大区间和、矩阵链乘法、最优二叉搜索树

情况 2：依赖于常数个子问题

- 背包问题、最长公共子序列、编辑距离

动态规划的本质

动态规划主要是对朴素递归的优化。

- 当我们看到一个递归解法对相同输入有重复调用时，可以用动态规划优化。
  - 简单地存储子问题的结果，这样在以后需要时就不必重新计算
- 这种简单的优化将时间复杂度从指数级降低到多项式级。
- 示例：Fibonacci 数的简单递归解导致指数时间复杂度。但如果通过存储子问题的解来优化，时间复杂度降为线性。

我们可以把动态规划看作是在 DAG 中找最短路径或计算路径（迭代方法），或者是带备忘录地遍历递归树（递归方法）。

贪心 vs. 动态规划

动态规划和贪心都是用于解决优化问题的算法范式。

DP vs Greedy Algorithms

贪心范式

理论思想：逐步解决问题；在每一步，算法根据某种启发式做出选择，以获得最明显和有益的收益。
适用性：满足贪心性质的问题：选择每个阶段的局部最优将导致形成全局最优
最优性：总是需要严格的证明
记忆化：可能需要维护一个数据结构来存储当前状态以做贪心选择
复杂度：通常更快
方式：以串行向前的方式计算解，从不回头或修改之前的选择。

动态规划范式

理论思想：找到子问题之间的顺序；使用已解决的子问题的解来解决当前子问题
适用性：可以通过递归/迭代方法解决，但会多次访问相同状态的问题
最优性：自动保证，因为动态规划实际上考虑了所有可能的情况然后选择最优
记忆化：需要 DP 表来存储已解决子问题的解
复杂度：通常较慢
方式：通过从更小的最优子解中综合来计算解：自底向上或自顶向下