



上海师范大学
Shanghai Normal University

《算法设计与分析》

9-动态规划 (Dynamic Programming)

杨启哲

上海师范大学信机学院计算机系

2025 年 10 月 27 日



- › 什么是动态规划
- › 动态规划算法设计

► 什么是动态规划

让我们回顾一下，Floyd 算法是怎么解决的所有节点对的最短路径问题的？

$$d_{i,j}^k = \begin{cases} \omega(i,j) & \text{如果 } k = 0 \text{ 且 } (i,j) \in E \\ \infty & \text{如果 } k = 0 \text{ 且 } (i,j) \notin E \\ \min\{d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\} & \text{如果 } k \geq 1 \end{cases}$$

上述递推式做了这两件事：

- 定义了一系列的**子问题**：节点 i 到节点 j 只经过 $1 \sim k$ 个节点的最短路径。
- “较大”的子问题与“较小”的子问题之间**存在某种关系**。

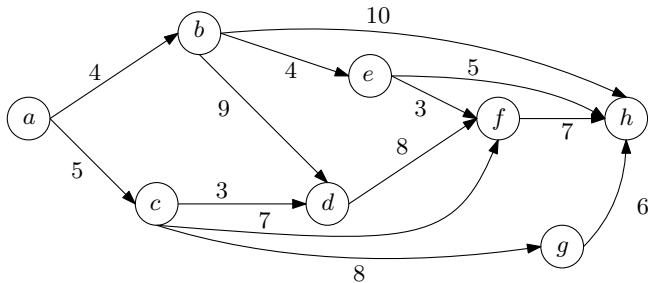
事实上，这是一种通用性很强的解法。我们再来看几个例子。

第一个例子跟图上的距离相关，只不过这次我们来研究一下 DAG 上的**最长路径**。

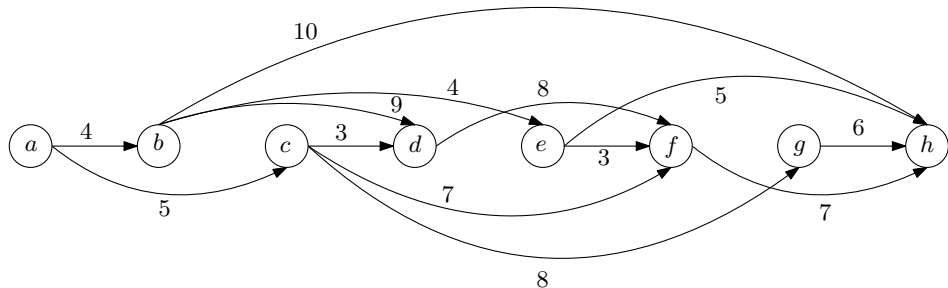
问题 1

[DAG 上的最长路径].

给定一个带权重的有向无环图 $G = (V, E, \omega)$ ，其中每条边 $(u, v) \in E$ 都有一个权重 $\omega(u, v)$ ，求 G 上的最长路径。



由之前我们可以知道，我们可以将一个 DAG 上的点按照拓扑序排列，如下图所示：



这里我们采取 $abcdefgh$ 的拓扑序，并且我们后续用 i 来表示第 i 个点。（**请注意，拓扑序并不一定是唯一的！**）

考察这样一个问题，用 L_i 来表示前 i 个点的子图里的最长路径， L_i 之间满足什么关系？

L_i 满足:

$$L_i = \max\{L_j + \omega(j, i), (j, i) \in E\}$$

从而算法如下:

算法: DAG 上的最长路径

输入: $G = (V, E, \omega)$

输出: G 上的最长路径

- 1: 计算 G 的拓扑序 $1, 2, \dots, n$
- 2: $L_0 \leftarrow 0$
- 3: **for** $i \leftarrow 1$ to n **do**
- 4: $L_i \leftarrow \max\{L_j + \omega(j, i), (j, i) \in E\}$
- 5: **return** $\max\{L_i, 1 \leq i \leq n\}$

时间复杂性 $O(n^2)$!

下一个例子则是跟字符串相关-最长公共子序列 (Longest Common Sequence)。

假设现在有两个序列:

- $X = x_1x_2 \dots x_m$
- $Y = y_1y_2 \dots y_n$

目标是求出 X 和 Y 的最长公共子序列, 这里 X 的子序列是形如 $x_{i_1}x_{i_2}, \dots, x_{i_n}$ 满足 $1 \leq i_1 < i_2 < \dots < i_n \leq m$ 的, Y 的子序列的定义类似。最长公共子序列即指两个序列中共有的长度最长的子序列。

例 2.

$X = \text{ABCB DAB}$, $Y = \text{BDCABA}$, 则 BCBA 是 X 和 Y 的一个最长公共子序列。

最简单的方式显然是暴力搜索。

暴力搜索

$$O(2^m \cdot n)!$$

1. X 一共有 2^m 个子序列。
2. 对于每个子序列，我们需要 $O(n)$ 的时间来判断它是否是 Y 的子序列。

我们尝试给出一个更快的算法，我们利用上述思想来考虑这个问题。

- 定义 $L[i][j]$ 表示 $x_1 \dots x_i$ 与 $y_1 \dots y_j$ 的最长公共子序列的长度。

$L[i][j]$ 之间存在什么联系？

当 $i, j > 0$ 时, $L[i][j]$ 满足:

- 如果 $x_i = y_j$, 则 $L[i][j] = L[i-1][j-1] + 1$ 。
- 如果 $x_i \neq y_j$, 则 $L[i][j] = \max\{L[i-1][j], L[i][j-1]\}$ 。

因此, 我们可以得到如下递推式:

$$L[i][j] = \begin{cases} 0 & \text{如果 } i = 0 \text{ 或者 } j = 0 \\ L[i-1][j-1] + 1 & \text{如果 } x_i = y_j \\ \max\{L[i][j-1], L[i-1][j]\} & \text{如果 } x_i \neq y_j \end{cases}$$

由上述递推式, 我们很容易写出一个算法:

算法 LCS

输入: 两个字符串 A, B, 长度分别为 m 和 n

输出: A 和 B 的最长公共子序列的长度

```
1: for i = 1 to m do
2:   L[i][0] ← 0
3: for j = 1 to n do
4:   L[0][j] ← 0
5: for i = 1 to m do
6:   for j = 1 to n do
7:     if A[i] = B[j] then
8:       L[i][j] ← L[i - 1][j - 1] + 1
9:     else
10:      L[i][j] ← max{L[i - 1][j], L[i][j - 1]}
11: return L[m][n]
```

时间复杂性: $O(mn)$!

L[i][j] 的更新特点

我们发现, $L[i][j]$ 的更新只与 $L[i-1][j-1]$, $L[i-1][j]$, $L[i][j-1]$ 有关。

	0	1	2	.	j	.	n
0	0	0	0	.	.	.	0
1	0	*	*	.	.	.	*
2	0	*	*	.	.	.	*
.	.	.	.	$L[i-1][j-1]$	$L[i-1][j]$.	.
i	.	.	.	$L[i][j-1]$	$L[i][j]$.	.
.
m	0	*	*	.	.	.	$L[m][n]$

这提示我们, 事实上, 我们并不需要保存整张表, 只需要保存两行就可以了。

算法的改进- $O(\min\{m, n\})$ 的空间!



事实上, 我们只需要**保存一行**即可。核心思想是用额外空间来临时保存 $L[i-1][j-1]$ 。

改进算法 LCS2

输入: 两个字符串 A, B , 长度分别为 m 和 n , 并且不妨令 $m \leq n$

输出: A 和 B 的最长公共子序列的长度

```
1: for  $i = 1$  to  $m$  do
2:    $L[i] \leftarrow 0$ 
3:  $tmp_1 \leftarrow 0, tmp_2 \leftarrow 0$ 
4: for  $i = 1$  to  $n$  do
5:   for  $j = 1$  to  $m$  do
6:      $tmp_1 \leftarrow tmp_2, tmp_2 \leftarrow L[j]$ 
7:     if  $A[j] = B[i]$  then
8:        $L[j] \leftarrow tmp_1 + 1$ 
9:     else
10:       $L[j] \leftarrow \max\{L[j], L[j-1]\}$ 
11: return  $L[m]$ 
```

我们现在讲了两个算法，展示了动态规划的一些威力。我们来总结一下这种思想的特点：

- 首先要定义相应的子问题。
- 子问题之间存在某种联系，即较大的子问题的最优解一定是来源于某些较小的子问题的最优解。

我们将上述两个特点称为**最优子结构性质**。

贪心算法也具有最优子结构性质！

这个时候让我们再思考一下贪心算法，我们发现，贪心算法也具有最优子结构性质！

- 当前的选择一定是最优解的一部分！

与动态规划不同的是，贪心算法不仅具有最优子结构，而且还具有所谓的**贪心选择性质**，也就是说其最优解不仅来自于较小的子问题，其还知道最终的最优解来源自哪个子问题的最优解-当前最好的那个！

我们再通过一个例子来说明**最优子结构**的重要性。

之前我们考察了在 DAG 上的最长路径问题，现在让我们来考虑在一般图上的最长路径问题，特别的我们考察无权的情况，即求出两个点之间的边数最长的简单路径，注意简单路径是不允许包含圈的。

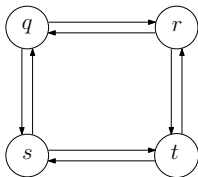
递归解？

仿照 Floyd 算法，我们可以写出如下的递推式：

$$d_{i,j}^k = \begin{cases} \omega(i,j) & \text{如果 } k = 0 \text{ 且 } (i,j) \in E \\ \infty & \text{如果 } k = 0 \text{ 且 } (i,j) \notin E \\ \max\{d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\} & \text{如果 } k \geq 1 \end{cases}$$

但很遗憾，上述递推式是**错误的**。

我们考察如下的一个图：



q 到 t 的最长路径

- q 到 t 的最长路径显然是 $q \rightarrow s \rightarrow t$.
- $q \rightarrow s$ 和 $s \rightarrow t$ 显然都不是 q 到 s 和 s 到 t 的最长路径。

一种理解是，要保证利用到的两个子问题是无关的。

通过上述讨论，我们可以总结出动态规划算法的设计方法：

1. 定义子问题, 寻找最优子结构。
2. 给出基于最优子结构的递推式，这一式子也被称作状态转移方程，或者动态规划范式。
3. 写出相应的算法，再针对性的进行优化。

► 动态规划算法设计

第一个问题还是跟字符串相关，当拼写检查工具遇到一个可能拼写的错误时，它将在自己的词典中查找与之相近的单词。那么，这里所谓的**距离相近**怎么表示？

匹配程度

一个自然的想法是，看两个字符串能在多大程度上进行对齐。比如考虑 SNOWY 和 SUNNY 这两个词，其可能有如下的对齐方式：

S	—	N	O	W	Y		—	S	N	O	W	—	Y
S	U	N	N	—	Y		S	U	N	—	—	N	Y

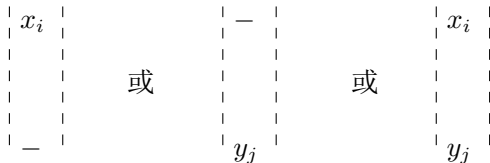
其中 — 表示一个空隙，我们可以将其随意的插入到每个字符串中。对于一种对齐方式，我们可以定义其**代价**为上下字符串对应字母不相同的个数，而**编辑距离**指的是两个字符串对齐方式中的最小代价。

可以看到，上述左边对齐方式的代价为 3，右边为 5。特别的，对于 SNOWY 和 SUNNY。我们不可能找到代价比 3 还小的对齐方式了，因此其**编辑距离**为 3。

当我们的目标聚焦于两个字符串 $x_1x_2\ldots x_m$ 和 $y_1y_2\ldots y_n$ 的编辑距离时，自然其前缀之间的编辑距离成为了子问题：

- 定义 $E[i][j]$ 表示 $x_1x_2\ldots x_i$ 和 $y_1y_2\ldots y_j$ 的编辑距离。

注意到对于任何一个关于 $x_1x_2\ldots x_i$ 和 $y_1y_2\ldots y_j$ 的对齐方式，其最右端一定是如下的形式：



由上述观察可得, $E[i][j]$ 满足:

- 或者等于 $1 + E[i - 1][j]$
- 或者等于 $1 + E[i][j - 1]$
- 或者等于 $\text{diff}(i, j) + E[i - 1][j - 1]$, 这里 $\text{diff}(i, j)$ 表示 x_i 与 $y[j]$ 是否相同, 相同为 0, 否则为 1。

从而 $E[i][j]$ 遵循如下递推式:

$$E[i][j] = \begin{cases} 0 & \text{如果 } i = 0 \text{ 或者 } j = 0 \\ \min\{1 + E[i][j - 1], 1 + E[i - 1][j], E[i - 1][j - 1] + \text{diff}(i, j)\} & \text{如果 } i, j \neq 0 \end{cases}$$

算法 EditDistance

输入: 两个字符串 A, B, 长度分别为 m 和 n

输出: A 和 B 的编辑距离

```
1: for i = 1 to m do
2:   E[i][0] ← 0
3: for j = 1 to n do
4:   E[0][j] ← 0
5: for i = 1 to m do
6:   for j = 1 to n do
7:     E[i][j] ← min{1 + E[i][j - 1], 1 + E[i - 1][j], E[i - 1][j - 1] + diff(i, j)}
8: return E[m][n]
```

1. 显然这是个 $O(mn)$ 的算法。
2. 如同最长公共子序列, 我们可以将其优化至 $O(\min\{m, n\})$ 的空间。

在分治算法中，我们介绍过两个矩阵的相乘，现在我们来考虑一下多个矩阵的相乘。注意到矩阵相乘是满足结合律的，考察如下的一个矩阵链相乘：

$$M_1 M_2 M_3 = \begin{bmatrix} a_{1,1} & \cdots & a_{1,100} \\ a_{2,1} & \cdots & a_{2,100} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} \\ \vdots & \vdots \\ b_{100,1} & b_{100,2} \end{bmatrix} \cdot \begin{bmatrix} c_{1,1} & \cdots & c_{1,100} \\ c_{2,1} & \cdots & c_{2,100} \end{bmatrix}$$

- 如果我们按照 $((M_1 M_2) M_3)$ 的顺序进行计算，那么我们需要 $2 \times 100 \times 2 + 2 \times 2 \times 100 = 800$ 次乘法。
- 如果我们按照 $(M_1 (M_2 M_3))$ 的顺序进行计算，那么我们需要 $100 \times 2 \times 100 + 2 \times 100 \times 100 = 40000$ 次乘法。

$(M_1 (M_2 M_3))$ 所需的乘法次数足足是 $((M_1 M_2) M_3)$ 的 50 倍！

问题 3.

给定 n 个矩阵 A_1, A_2, \dots, A_n , 其中 A_i 的规模为 $p_{i-1} \times p_i$, 求一这 n 个矩阵相乘的所需的最少乘法次数。

我们可以看到, 这其中存在着指数多个不同的次数。

事实上，对于 n 个矩阵的相乘，不同的运算顺序会导致不同的乘法次数，比如当计算 4 个矩阵相乘时，我们有如下的 5 种运算顺序：

- $((M_1M_2)M_3)M_4$
- $(M_1(M_2M_3))M_4$
- $(M_1M_2)(M_3M_4)$
- $M_1((M_2M_3)M_4)$
- $M_1(M_2(M_3M_4))$

指数种运算顺序！

事实上，对于 n 个矩阵的相乘，令 $P(n)$ 表示其运算顺序的个数，那么 $P(n)$ 满足：
$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)。$$
这是著名的**卡特兰数**，其通项为 $P(n) = \frac{1}{n} \binom{2n-2}{n-1} = \Omega(\frac{4^n}{n^{1.5}})$ 。

我们现在继续尝试用动态规划来解决这个问题。我们考虑如下的子问题：

- 定义 $C[i][j]$ 表示 $A_i \cdot A_{i+1} \cdot A[j]$ 所需的最小乘法次数。

注意到计算 $A_i \cdot A_{i+1} \times \cdots \times A_j$ 时，最后一步一定是如下的样子：

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

因此 $C[i][j]$ 一定满足：

$$C[i][j] = \min_{i \leq k < j} \{C[i][k] + C[k+1][j] + p_i p_k p_j\}$$

由此我们可以快速得到计算矩阵链相乘最少乘法次数的 $O(n^3)$ 的算法。

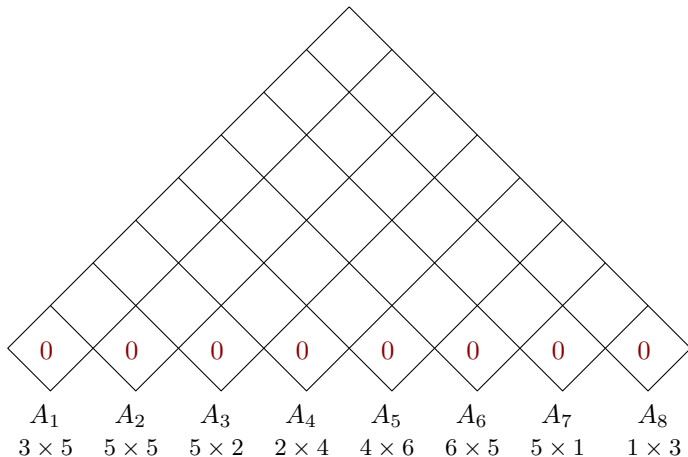
算法: Matchain

输入: 一个 $n + 1$ 维数组 $r[1, \dots, n]$, 其中 $r[i] (i \leq n)$ 表示第 i 个矩阵的行数, r_{n+1} 第 n 个矩阵的列数。

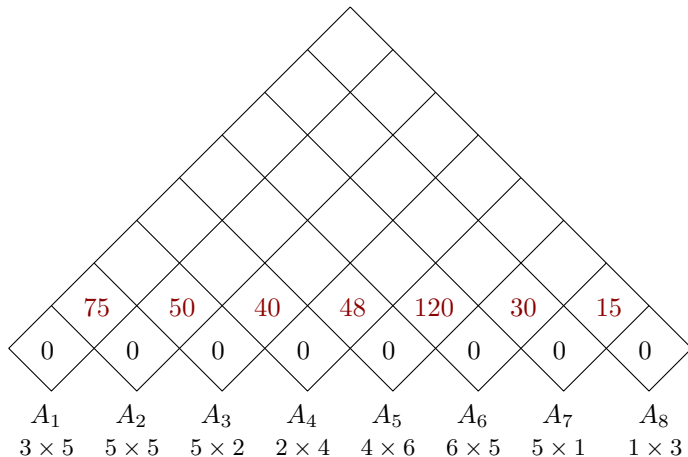
输出: 这 n 个矩阵相乘所需的最少乘法次数

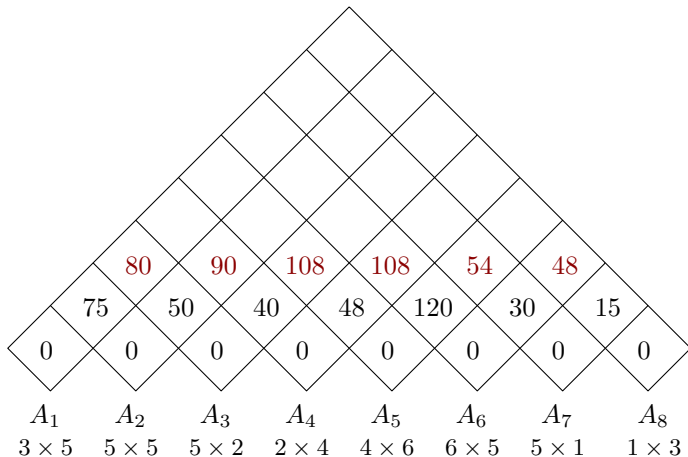
```
1: for  $i \leftarrow 1$  to  $n$  do
2:    $C[i][i] \leftarrow 0$ 
3: for  $l \leftarrow 2$  to  $n$  do
4:   for  $i \leftarrow 1$  to  $n - l + 1$  do
5:      $j \leftarrow i + l - 1$ 
6:      $C[i][j] \leftarrow \infty$ 
7:     for  $k \leftarrow i$  to  $j - 1$  do
8:        $C[i][j] \leftarrow \min\{C[i][j], C[i][k] + C[k + 1][j] + r[i]r[k + 1]r[j + 1]\}$ 
9: return  $C[1][n]$ 
```

我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：

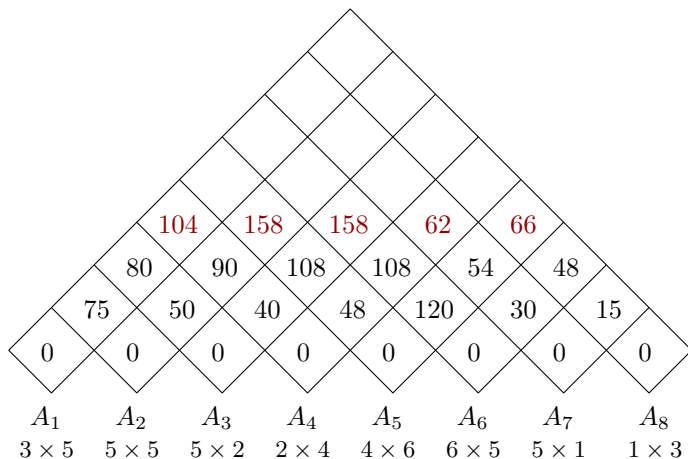


我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：

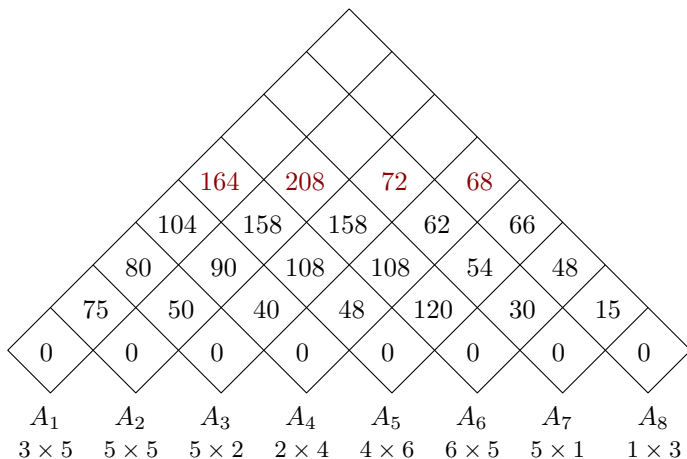




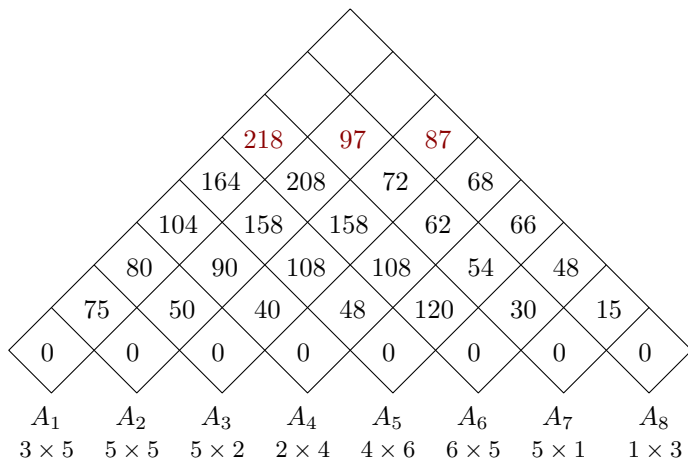
我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：

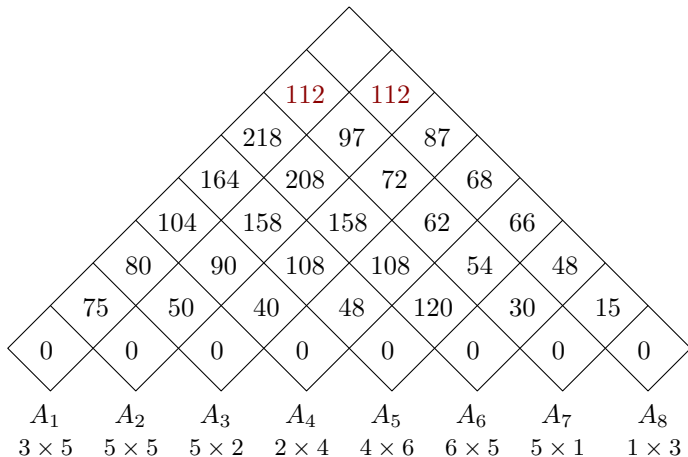


我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：

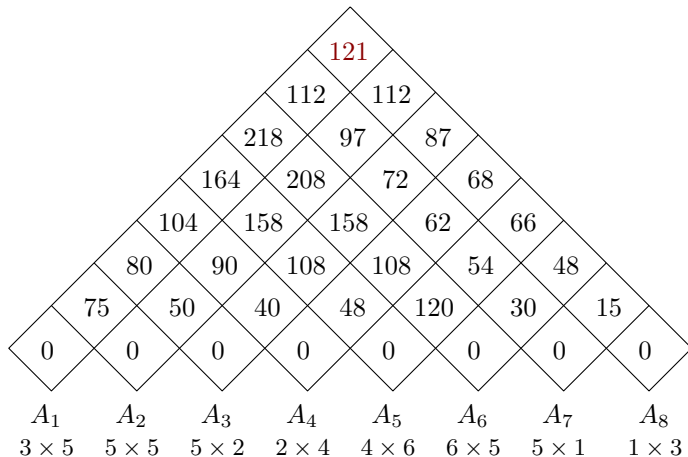


我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：





我们再用一个例子来解释一下，假设现在考虑 8 个矩阵相乘，其规模分别为 $3 \times 5, 5 \times 5, 5 \times 2, 2 \times 4, 4 \times 6, 6 \times 5, 5 \times 1, 1 \times 3$ ，则算法运行如下：



本节的最后，我们来考察动态规划里的一个经典问题-背包问题。

问题 4.

在一次打劫中，盗贼发现他的战利品太多，以至于无法全部带走，于是他必须决定究竟要把哪些战利品装入自己的背包。

假设他的背包一共能装重量为 W 的物品。他一共有 n 件物品，分别重 w_1, \dots, w_n ，每件价值分别为 v_1, \dots, v_n 。请问他应该选哪几件物品装入背包，才能使得背包中的物品总价值最大？

每件物品只能拿一次的背包问题，也被称为01 背包问题。



例 5.

假设 $W = 10$, 且:

物品	重量	价值
1	6	20
2	3	14
3	4	16
4	2	9

最佳方案是什么?

拿物品 2, 3, 4, 一共价值 39.

考虑如下的子问题:

- 定义 $V[i][j]$ 表示从前 i 个问题中选出总重量为 j 的物品的最大价值。
-

$v[i][j]$ 满足:

- 或者等于 $V[i-1][j]$ 。
- 或者等于 $V[i-1][j-w_i] + v_i$ 。

即:

$$V[i][j] = \max\{V[i-1][j], V[i-1][j-w_i] + v_i\}$$

算法 Knapsack

输入: n 个物品, 重量分别为 w_1, \dots, w_n , 价值分别为 v_1, \dots, v_n , 背包容量为 W

输出: 最大价值

```
1: for  $j = 0$  to  $W$  do
2:    $V[0][j] \leftarrow 0$ 
3: for  $i = 1$  to  $n$  do
4:   for  $j = 0$  to  $W$  do
5:     if  $j \geq w_i$  then
6:        $V[i][j] \leftarrow \max\{V[i-1][j], V[i-1][j-w_i] + v_i\}$ 
7:     else
8:        $V[i][j] \leftarrow V[i-1][j]$ 
9: return  $V[n][W]$ 
```

- Knapsack 需要 $O(nW)$ 的时间。
- Knapsack 需要 $O(nW)$ 的空间。

Knapsack 算法是多项式时间算法么？

不！

- 请注意，算法输入的 W 是按二进制输入的，因此输入大小是 $\log W$ ，不是 W ，从而这不是一个多项式时间算法。
- 事实上，背包问题是一个 NP 完全问题，这意味着其不太可能有多项式时间算法。

Knapsack 算法的空间还可以再优化么？ 可以！

算法 Knapsack2

输入: n 个物品, 重量分别为 w_1, \dots, w_n , 价值分别为 v_1, \dots, v_n , 背包容量为 W

输出: 最大价值

```
1: for  $j = 0$  to  $W$  do
2:    $V[j] \leftarrow 0$ 
3: for  $i = 1$  to  $n$  do
4:   for  $j = W$  to  $w_i$  do
5:     if  $j \geq w_i$  then
6:        $V[j] \leftarrow \max\{V[j], V[j - w_i] + v_i\}$ 
7: return  $V[W]$ 
```

Knapsack2 第 4 行为什么要逆着来?

因为我们要保证 $V[j - w_i]$ 是 $V[i - 1][j - w_i]$ 的值。

如果我们允许一件物品多可以重复拿，会发生什么？

例 6.

在上述的例子中 ($W = 10$),

物品	重量	价值
1	6	20
2	3	14
3	4	16
4	2	9

如果允许重复拿，最佳方案变成了5件物品 4, 总价值 45!

能无限次重复拿的背包问题也被称作完全背包问题。

事实上子问题依旧是一样的：

- 定义 $V[i][j]$ 表示从前 i 个问题中选出总重量为 j 的物品的最大价值。

但 $V[i][j]$ 满足：

$$V[i][j] = \max_l \{V[i-1][j], V[i][j-l \cdot w_i] + l \cdot v_i\}$$

- 利用上面的方式自然可以写出相应的算法。
- 注意到我们并不需要决定第 i 件物品拿或不拿，只需要考虑怎么拿价值最大即可。所以类似 Knapsack2 我们有个非常简单的优化方式。

算法 Knapsack3

输入: n 个物品, 重量分别为 w_1, \dots, w_n , 价值分别为 v_1, \dots, v_n , 背包容量为 W

输出: 最大价值

```
1: for  $j = 0$  to  $W$  do
2:    $V[j] \leftarrow 0$ 
3: for  $i = 1$  to  $n$  do
4:   for  $j = 0$  to  $W$  do
5:     if  $j \geq w_i$  then
6:        $V[j] \leftarrow \max\{V[j], V[j - w_i] + v_i\}$ 
7: return  $V[W]$ 
```

请注意与 Knapsack2 的区别!

只需要改变第四行循环的顺序, 我们就得到了一个多副本背包问题的算法!

事实上，背包问题还有很多变种，比如：

- 如果允许一件物品拿一部分，即允许分数，则算法是怎么样？
贪心算法就起作用了！
- 如果一件物品可以拿多次，但能拿的次数有次数限制，则算法是怎么样？（多重背包）
- 如果有的能无限拿，有的能重复拿有限多次，有的只能拿一次，那算法是怎么样？
（混合背包）
- ...

后面的问题都可以使用动态规划解决，大家有兴趣的话可以自己尝试一下。



本节内容

- 动态规划算法
 - 什么是动态规划
 - 动态规划算法的特点
- 贪心算法设计举例
 - 编辑距离
 - 矩阵链相乘
 - 背包问题