算法第二讲——矩阵连乘、01背包、Alignment

申世伟

2015-11-6

1 矩阵链式乘法

下面我们将从“矩阵链式乘法”这个简单的例子入手讲解动态规划。通过讲解这个例子，我们可以总结下如果要用动态规划的算法去解决实际问题，需要有哪些要素、解决问题的关键是什么以及怎样描述并定义子问题。

1.1 解决问题的一般思路：

首先我们先回忆下上一节提到的解决问题的三个基本思路。

碰到一个问题，如果这个问题太大了以致于搞不定，看能不能把它变小，把它规整成小问题去解决，这是考虑问题最基本的想法。

比如说给你一个长度为n的数组，n个数太多了我们不会做，我们可以先尝试一个数能不能做，然后两个数能不能做，这样一直进行下去。

另外一种对待问题的思路是，我们可以把n个数分成左一半和右一半，然后看左半部分会不会做，右半部分会不会做。将大问题分解为小问题去解决，这就是分治的思想。

动态规划与分治算法的联系：

(1)动态规划和分治是非常像的，都是要把大问题分解成子问题，然后将子问题的解进行合并起来求原问题的解。

(2)动态规划一般会枚举所有的子问题，要把所有的子问题都解决一遍，但是它避免了对同一个子问题的重复计算，那它是怎么避免重复的呢，这就是programming。programming的意思是说生成一张表，不断的向表中填数，当访问到表中单元时，如果表中有值则直接返回，没有则进行求解并将求到的值填在表中。

(3)动态规划和贪心一样，都可以典型的求解最优化问题，但是动态规划又不仅仅用于最优化问题的求解，例如 p-value 的计算问题。

通常来说，只要我们发现一个问题当中能存在一种递归的性质，我们就可以把它分解成子问题，就能找到一种递归的关系，这个时候就可以用动态规划进行求解。

当我们在计算一个原始问题的时候，我们需要把原问题进行扩展，试图发现有意义的递推关系，而确定递推关系的关键就在于确定子问题的一般形式。

矩阵链式乘法的形式化描述：

- **Input:**

  A sequence of n matrices $A_1, A_2, ..., A_n$; matrix A_i has dimension $p_{i-1} \times p_i$;

- **Output:**
  Fully parenthesizing the product $A_1A_2...A_n$ in a way to minimize the number of scalar multiplications.

我们的目标是给我们n个矩阵 $A_1, A_2, A_3..., A_n$，其中A_i大小为$P(i-1)*P(i)$,求最好的加括号方案使得整体的运算次数最少。

具体事例:

下面我们看下矩阵链式乘法这个例子:

比如说我们有如下四个矩阵A_1, A_2, A_3, A_4:

$$A_1 = \underset{1\times 2}{\begin{bmatrix} 1 & 2 \end{bmatrix}} \quad A_2 = \underset{2\times 3}{\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}} \quad A_3 = \underset{3\times 4}{\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}} \quad A_4 = \underset{4\times 5}{\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}}$$

Solutions:	$(((A_1)(A_2))(A_3))(A_4)$	$((A_1)(A_2))((A_3)(A_4))$
Cost:	$1\times 2\times 3$	$1\times 2\times 3$
	$+1\times 3\times 4$	$+3\times 4\times 5$
	$+1\times 4\times 5$	$+1\times 3\times 5$
	$=38$	$=81$

对这四个矩阵做运算，我们有很多种加括号的方案，比如：

方案一：先算$A_1 * A_2$，然后再乘以A_3，最后乘以A_4，此时总共运算次数为38次。

方案二：先算$A_1 * A_2$，然后算$A_3 * A_4$，最后将两者得到的矩阵相乘，此时总的运算次数为81次。

我们使用两种不同的求解顺序，得到了不同的运算次数且差异很大。现在我们想知道第一种方案是最优的吗？是否存在更好的加括号的方案使得总共的运算次数最少。

解空间：

总共有多少种加括号的方案呢，实际上加括号的方案可以描述成一颗二叉树，二叉树的每个节点对应一个子问题。总共n个节点，则有$\binom{2n}{n} - \binom{2n}{n-1}$ (Catalan number) 个从根节点到叶子节点的路径

即$\binom{2n}{n}-\binom{2n}{n-1}$个加括号的方案。

卡特兰数是指数级别的，解空间非常大，如果暴力枚举的话，速度会非常慢，所以暴力枚举这种策略是不可行的。

动态规划的一般思路：

下面我们研究下动态规划是怎么进行求解的。

问题的求解还是基于我们的观察，现在给我们n个矩阵，我们不会做，我们可以看能不能把它分解成小问题来求解。

我们的解就是加括号的方案，把求解过程想象成一系列的决策。每一步的决策是确定在哪个位置加括号，即确定先算哪些矩阵，再算哪些矩阵。

假如当前我们拿到了一个最优解，我们记做 O ，且假设我们将第一个括号加在第 k 个矩阵与第 $k+1$ 个矩阵之间，即 $(A_1...A_k)(A_{k+1}...A_n)$,也就是说我们要先算$(A_1...A_k)$，再算$(A_{k+1}...A_n)$,最后将两部分得到的矩阵乘起来。

这样我们就把原始问题分解成在 $(A_1...A_k)$ 里面加括号使得运算次数最小和在 $(A_{k+1}...A_n)$ 加括号使得运算次数最小两个子问题了。

我们可以发现左半部分的右下标需要改变，右半部分的左下标需要改变，因此该问题便是从A_i到A_j，这样问题的左右下标都可以变化。因此子问题一般形式表示为在 $(A_i,...,A_j)$ 加括号使得整体运算次数最少，我们记做 $OPT(i,j)$, 显然原始问题可以表示成 $OPT(1,n)$.

因为左下标可以从1变化到n,而右下标可以从 i 变化到n，因此原问题的解空间为$\sum_{i=1}^{n}(n-i+1)=\frac{n(n+1)}{2}$,即原问题包含 $O(n^2)$ 个子问题空间。

假如我们是在 A_k 和 A_{k+1} 矩阵之间加的括号，则在A_i到A_j所使用的运算次数就是

$$Cost(i,j) = Cost(i,k) + Cost(k+1,j) + p_i p_{k+1} p_{j+1}$$

该表达式对于任意的解都有这个性质。因此我们考虑最优解这个特殊情况有：

$$OPT(i,j) = OPT(i,k) + OPT(k+1,j) + p_i p_{k+1} p_{j+1}$$

即 $(A_i, ..., A_j)$ 最少的运算次数，等于$(A_i, ..., A_k)$最少的运算次数加上$(A_k, ..., A_j)$最少的运算次数加上最后两个矩阵相乘的运算次数。即原问题包含子问题的最优解，这就是最优子结构性质。

求解一个问题，如果这个问题可以规约，则分治大概可以解决问题。如果他还有最优子结构的性质，则动态规划大概可以解决问题。因此在碰到一个问题的时候，先观察该问题具有哪些性质，根据不同的性质我们使用不同的策略。

1.2 最优子结构性质：

那我们如何证明最优子结构呢：

如果对$(Ai, ..., Ak)$有另外一种代价更小的加括号方案 $OPT`(i,k) < OPT(i,k)$,那将它替换到 $(A_i, ...A_j)$ 的最优加括号的策略中，就会产生另外一种加括号的方案，且代价小于最优代价 $OPT(i,j)$ ，这与原始定义$OPT(i,j)$ 是最优的矛盾。因此原问题一定包含子问题的最优解。

本问题隐含了独立性假设即左边的求解与右边的求解是彼此不影响的。

但是现在我们还不能知道k具体在哪个位置，因此需要在i和j区间内对k的所有情况进行枚举，则我们有如下递推表达式：

$$OPT(i,j) = \begin{cases} 0 & i = j \\ \mathbf{min_{i \le k < j}}\{OPT(i,k) + OPT(k+1,j) + p_i p_{k+1} p_{j+1}\} & otherwise \end{cases}$$

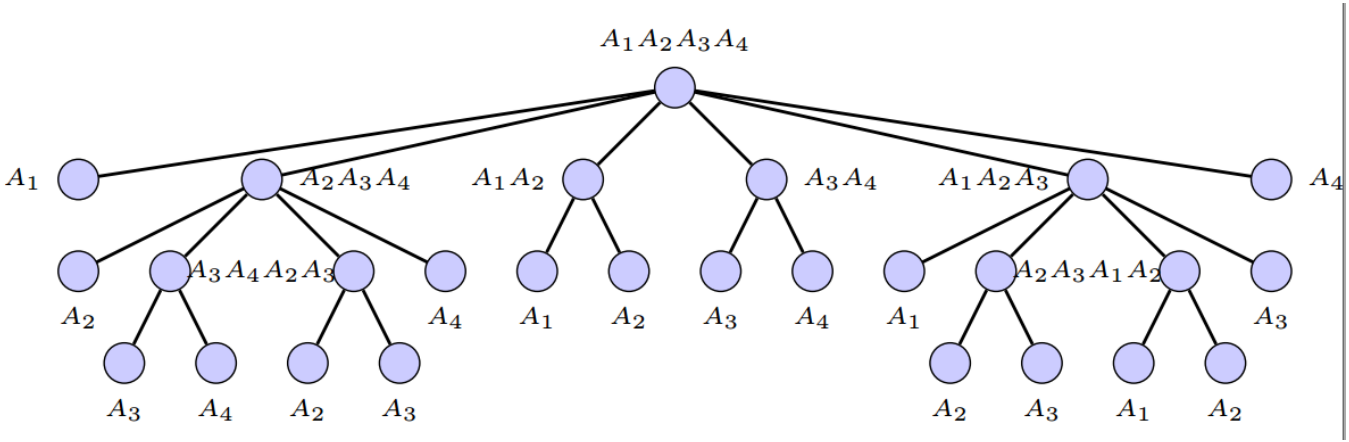
现在我们可以利用得到的递归表达式，写出如下伪代码：

RECURSIVE_MATRIX_CHAIN(i,j)

if $i == j$ **then**
  return 0;
end if
$OPT(i,j) = +\infty$;
for $k = i$ to $j-1$ **do**
  $q =$ RECURSIVE_MATRIX_CHAIN(i,k)
    + RECURSIVE_MATRIX_CHAIN$(k+1,j)$
    $+p_i p_{k+1} p_{j+1}$;
  if $q < OPT(i,j)$ **then**
    $OPT(i,j) = q$;
  end if
end for
return $OPT(i,j)$;

注意最优解可以通过调用 RECURSIVE_MATRIX_CHAIN$(1,n)$ 得到.

通过伪代码的递归形式，我们可以画出如下递归树：



递归树中的每个节点都代表一个子问题，我们可以发现上面的递归树中有很多节点是重复的。

时间复杂度：

现在我们求下该程序的时间复杂度：

假设$T(n)$ 为计算 n 个矩阵乘积的复杂度，则有

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

下面我们证明$T(n) \geq 2^{n-1}$,即原问题是指数时间的复杂度：

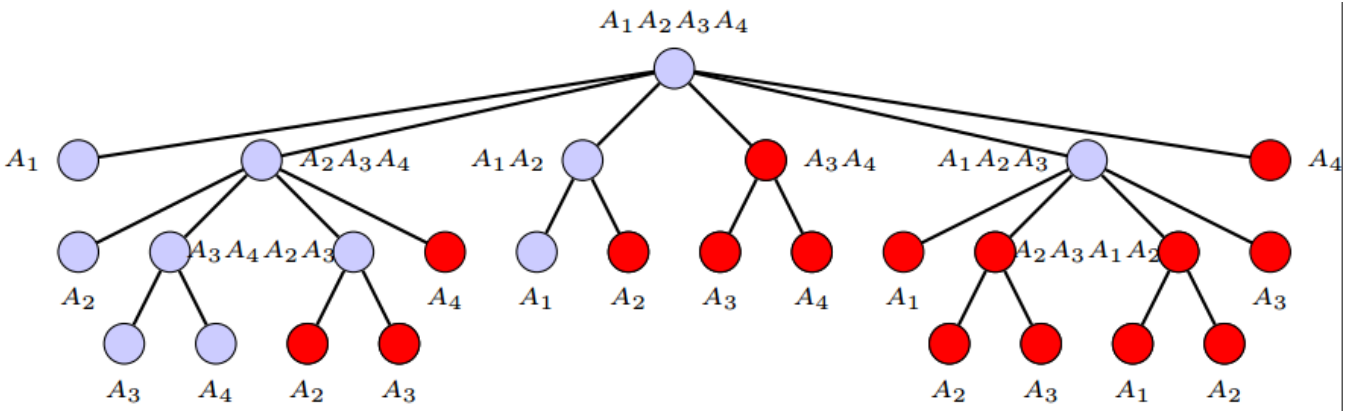
首先我们有$T(1) \geq 1 = 2^{1-1}$ 然后针对$n > 1$ 的情况，我们有：

$$\begin{aligned} T(n) &\geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) && (1)\\ &= n + 2\sum_{k=1}^{n-1} T(k) && (2)\\ &\geq n + 2\sum_{k=1}^{n-1} 2^{k-1} && (3)\\ &\geq n + 2(2^{n-1} - 1) && (4)\\ &\geq n + 2^n - 2 && (5)\\ &\geq 2^{n-1} && (6) \end{aligned}$$

1.3 记忆化搜索优化时间复杂度：

现在有个问题，我们总共只有 $O(n^2)$ 个子问题，但是现在我们的程序竟然花费了 2^n 时间，表明这里面肯定有子问题被重复计算了。

比如下图的红色节点：



我们直观的想法是可以将已经计算过的子问题保存在一张表里面。下次如果再求解该子问题则直接将表中存放的值返回即可，这样便避免了重复计算。

因此我们有下面伪代码：

MEMORIZE_MATRIX_CHAIN(i, j)

1: **if** $OPT[i, j] \neq NULL$ **then**
2: **return** $OPT(i, j)$;
3: **end if**
4: **if** $i == j$ **then**
5: $OPT[i, j] = 0$;
6: **else**

```
 7:    for k = i to j − 1 do
 8:       q = MEMORIZE_MATRIX_CHAIN(i, k)
 9:            +MEMORIZE_MATRIX_CHAIN(k + 1, j)
10:            +p_i p_{k+1} p_{j+1};
11:       if  q < OPT[i, j] then
12:          OPT[i, j] = q;
13:       end if
14:    end for
15: end if
16: return   OPT[i, j];
```

其实就是在原有伪代码的基础上添加了对表格中该单元的判断，如果该单元有值，则表明已经算过该子问题则直接返回，否则则对该子问题进行求解并保存在表格中。

因为原问题有 $O(n^2)$ 个子问题，每个子问题有 $O(n)$ 种选择，因此原问题的时间复杂度为O（n^3）。

由于递归需要使用内存中的栈结构，运算速度稍微慢些，因此我们可以尝试使用非递归的形式自底向上进行计算。因此有以下伪代码:

```
MATRIX_CHAIN_MULTIPLICATION(P)
1: for i = 1 to n  do
2:    OPT(i, i) = 0;
3: end for
```

```
4: for l = 2 to n do
5:    for i = 1 to n − l + 1 do
6:       j = i + l − 1;
7:       OPT(i, j) = +∞;
8:       for k = i to j − 1 do
9:          q = OPT(i, k) + OPT(k + 1, j) + p_i p_{k+1} p_{j+1};
10:         if q < OPT(i, j) then
11:            OPT(i, j) = q;
12:            S(i, j) = k;
13:         end if
14:      end for
15:   end for
16: end for
17: return OPT(1, n);
```

因此我们有以下的运算过程:

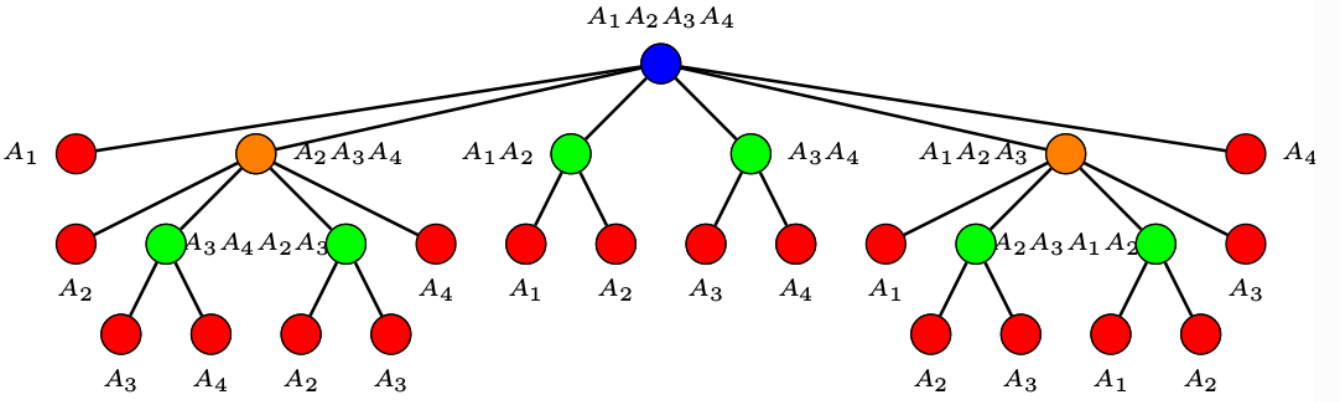
(1)求解红色(叶子)节点上的子问题。

(2)求解绿色节点上的子问题。

(3)求解橘黄色节点上的子问题。

(4)得到原问题的最优解。

我们可以看到动态规划是自低向上求解最优解问题的。



1.4 具体事例运算过程：

下面我们详细看下动态规划求解矩阵链式乘法的运算过程：

OPT

1	2	3	4	
0	6			1
	0	24		2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1			1
		2		2
			3	3
				4

第一步:
$OPT[1,2] = p_0 \times p_1 \times p_2 = 1 \times 2 \times 3 = 6;$
$OPT[2,3] = p_1 \times p_2 \times p_3 = 2 \times 3 \times 4 = 24;$
$OPT[3,4] = p_2 \times p_3 \times p_4 = 3 \times 4 \times 5 = 60;$

OPT

1	2	3	4	
0	6	18		1
	0	24	64	2
		0	60	3
			0	4

SPLITTER

1	2	3	4	
	1	2		1
		2	3	2
			3	3
				4

第二步:

$$OPT[1,3] = \min \begin{cases} OPT[1,2] + OPT[3,3] + p_0 \times p_3 \times p_4 (= 18) \\ OPT[1,1] + OPT[2,3] + p_0 \times p_2 \times p_4 (= 32) \end{cases} \quad \text{Thus, } SPLITTER[1,2] = 2.$$

$$OPT[2,4] = \min \begin{cases} OPT[2,2] + OPT[3,4] + p_1 \times p_2 \times p_4 (= 90) \\ OPT[2,3] + OPT[4,4] + p_1 \times p_3 \times p_4 (= 64) \end{cases} \quad \text{Thus, } SPLITTER[2,4] = 3.$$

OPT

	1	2	3	4	
1	0	6	18	38	1
2		0	24	64	2
3			0	60	3
4				0	4

SPLITTER

	1	2	3	4	
1		1	2	3	1
2			2	3	2
3				3	3
4					4

第三步:

$$OPT[1,4] = \min \begin{cases} OPT[1,1] + OPT[2,4] + p_0 \times p_1 \times p_4 (= 74) \\ OPT[1,2] + OPT[3,4] + p_0 \times p_2 \times p_4 (= 81) \\ OPT[1,3] + OPT[4,4] + p_0 \times p_3 \times p_4 (= 38) \end{cases} \quad \text{Thus, } SPLITTER[1,4] = 3.$$

经过我们一步步的计算最后得到结果是 38 ，即我们运用动态规划得到的第一种方案是最好的方案。

1.5 构建最优解方案:

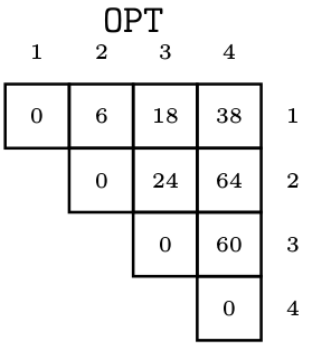
虽然我们找到了最优解的值，但是怎样求得与该最优解对应的解方案呢。

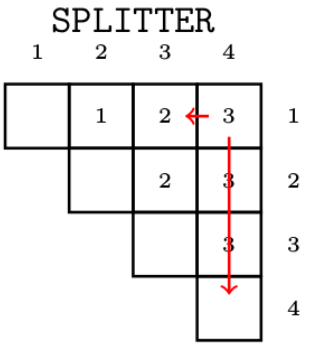
我们的解决方案是进行回溯，需要我们另外使用一个数组S保存当前节点最优解的来源。比如$S[i,j]$ 是记录 了对乘积$A_i,...,A_j$ 在A_k 与A_{k+1} 之间进行分开以取得最优加括号方案的 k 值。

因此原问题$A_1,...,A_n$的最优解方案就是$(A_1,...,A_{S[1,n]})(A_{S[1,n]+1},...,A_n)$.

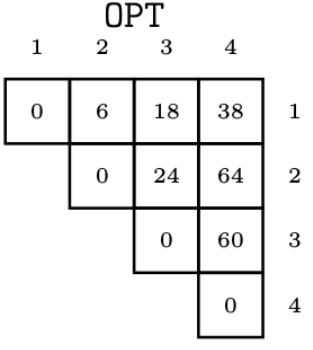
这里我们需要注意：原问题的最优解只有当所有的子问题都被算出来后才能得到。

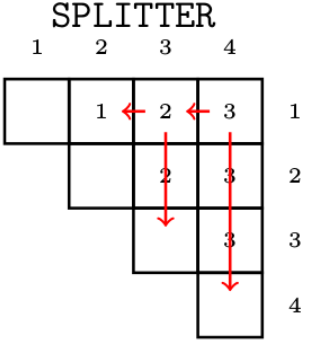
下面我们尝试回溯找到最优解方案:



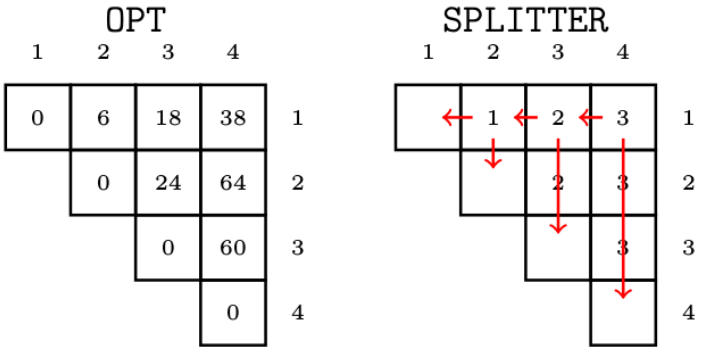


第一步： $(A_1A_2A_3)(A_4)$





第二步： $((A_1A_2)(A_3))(A_4)$



第三步： $(((A_1)(A_2))(A_3))(A_4)$

根据我们的回溯，知道了最优解方案就是先算A_1A_2 然后结果乘以A_3，最后乘以A_4.

1.6 问题总结：

经过对矩阵链式乘法的整个求解过程，我们有如下总结：

如果大问题搞不定，我们可以将其分解成更小的子问题。在动态规划求解问题时关键是如何定义子问题， 我们可以将求解过程想象成多步决策的过程，再假设已经拿到了最优解，然后考察第一个决策是做什么的，此时可能会有多种情况，那我们就枚举所有情况，然后观察子问题的所有形式，并进行总结便会得到子问题的一般形式即递归表达式。

最后看该问题是否满足最优子结构的性质。如果满足则根据递归表达式，我们便可以写出源代

码。

2 0/1背包问题：

现在有个物品集，每个物品都有重量和价值，现在希望选择一个物品子集，使得总的重量小于给定的重量并且总的价值最大。

问题的形式化描述：

该问题的形式化描述如下：

- **Input:**
  A set of items. Item i has weight w_i and value v_i, and a total weight limit W;
- **Output:**
  A sub-set of items to maximize the total value with a total weight below W.

这里 0/1 的意思是只能装或者不装这个物品，装记做1，不装记做0.

2.1 具体事例：

下面我们看下一个生活中的例子：

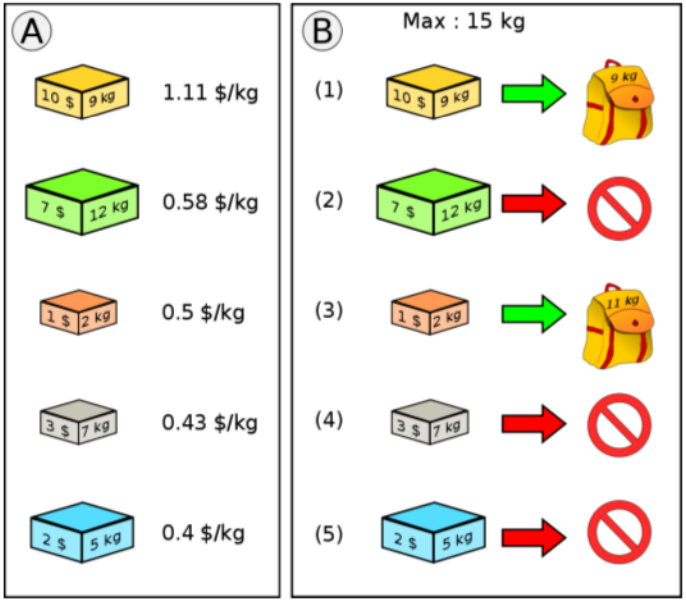


Figure 1: 物品从上到下依次为金银铜铁锡

我们直观的想法可能是先装单价贵重的物品，比如图中的物品，我们可能会先装金块，因为最大只有15kg，装完金块后还剩6 kg，此时还能装铜块和 x ｉ 块。按照我们先装单价贵重的物品则会装铜块，此时还剩下4 kg，则没有物品可以装的下了。按照我们启发式的装东西想法得到的总价为11$，可是我们启发式得到的一定是最优的吗？下面我们看下动态规划是怎么求解的。

2.2 动态规划求解思路：

现在我们还是按照之前解决矩阵链式乘法的求解思路来：

当物品集是n个物品的时候，我们不知道怎么解决，可以先考虑$n-1$个物品，或者$n/2$个物品，看可不可以将物品数目变少。

我们解方案是选哪些物品，是物品的子集。我们可以把问题的求解想象成一系列的决策，在第 i 步，我们决定第 i 个物品是装还是不装。

现在假设我们已经得到子问题的最优解，当前考虑第一个决策即考虑最后一个物品是装还是不装。

如果装，则原问题变成从前$n-1$个物品中选择限重 $W-w_n$ 的物品子集使得价值最大。

如果不装，则问题变成在前$n-1$个物品中选择限重 W 的物品子集使得价值最大。

此时对子问题的两种情况进行总结，我们可以得到子问题的一般形式，在前 i 个物品中选择物品，选物品的价值越大越好，将子问题的最优解记做$OPT(i,w)$.

我们对 (i,w) 的两种情况进行分析，并取最大值，则有如下递归表达式：

$$OPT(i,w)=\max\{OPT(i-1,w),OPT(i-1,w-w_n)+v_n\}$$

根据递归表达式，我们可以写出如下伪代码：

KNAPSACK(n,W)

1: **for** $w=1$ to W **do**

2: $\quad OPT[0,w]=0;$

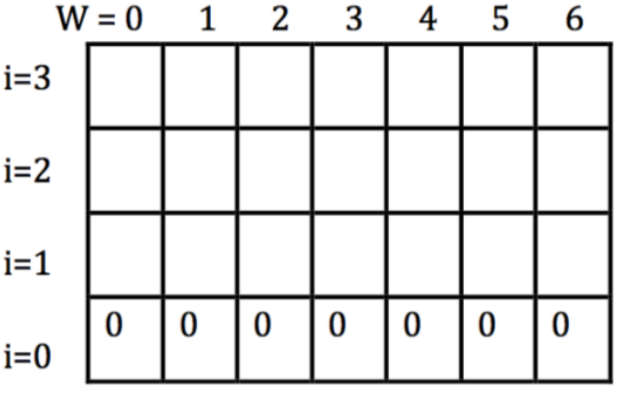
3: **end for**
4: **for** $i = 1$ to n **do**
5: **for** $w = 1$ to W **do**
6: $OPT[i, w] = \max\{OPT[i-1, w], v_i + OPT[i-1, w-w_i]\}$;
7: **end for**
8: **end for**

我们列了一张4行7列的一张表格， $dp[i][j]$ 表示前 i 个物品，容量为 j 的时候最多可以装多少价值的物品。

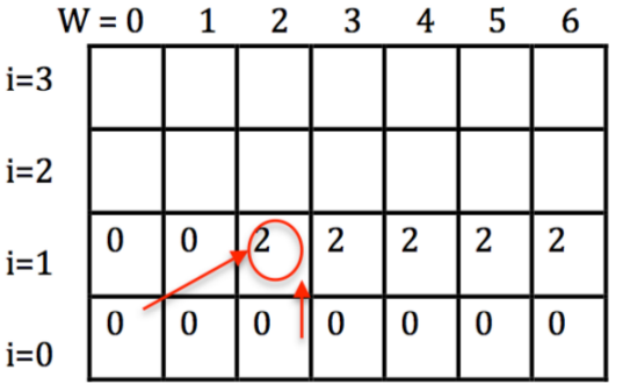
程序的开始需要对数组进行初始化，对前 0 个物品装，无论容量多大，最大效益都是0。

因此我们有如下求解过程:

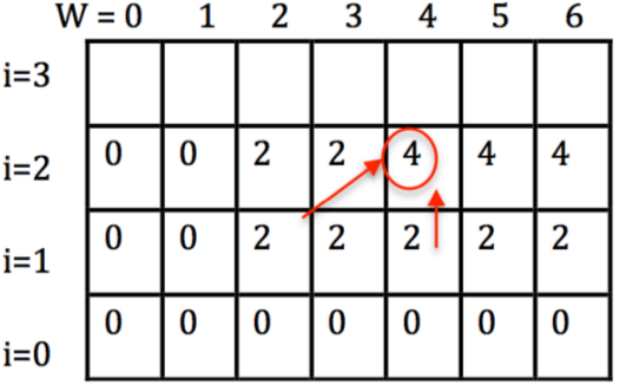
Initially all OPT[0,w] = 0



OPT[1,2] = max{
OPT[0,2] (=0),
OPT[0,0] + V1 (=0+2) }
= 2



OPT[2,4] = max{
OPT[1,4] (=2),
OPT[1,2] + V2 (=2+2) }
= 4



OPT[3,3] = max{
OPT[2,3] (=2),
OPT[2,0] + V3 (=0+3) }
= 3

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

Backtracking

OPT[3,6] = max{
OPT[2,6] (=4),
OPT[2,3] + V3 (=2+3) }
= 5
Decision: Select item 3

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

最后我们得到$dp[3][6] = 5$,那么如何找到最优解方案呢。

根据之前我们做矩阵链式乘法的经验知道可以通过对中间过程进行记录，一步步进行回溯找到该最优解方案。

2.3 时间复杂度的讨论：

该问题的时间复杂度为 $O(nW)$。因为原问题有 nW 个子问题，每个子问题进行2次比较，则共有$O(2nW)$ 运算次数，因此原问题的时间复杂度就是 $O(nW)$。

如果 W 过大，则该算法将不是很高效。因为数值是用2进制表示的，则 W 将使用 $logW$ 个 bit 表示，则原问题的时间复杂度为$O(n2^{logW}) = O(n2^{input\ length})$, 是与输入长度相关的指数表达式， 这种形式的表达式叫做伪多项式时间的算法。

2.4 另外一个子问题表示：

开始我们假设所有的问题给了一个排序。现在我们假设对 n 个物品不排序，则子问题可以定义成$OPT(s, W)$ 即在 S 这个物品集合里，当包的容量最大是 W 时如何使装的物品的价值最大。

现在考察包里面的任何一个物品，则该物品有装和不装两种情况，因此有递归表达式

$$OPT(S, W) = \max\{OPT(S - \{i\}, W - w_i) + v_i\ ,\ OPT(S - \{i\}, W)\}$$

这种思路在理论上是ok的，但是时间复杂度会非常高。因为任何一个物品的子集我们都要作为 S 进行求解，如果有 n 个物品，则有2^n个子集，则问题的时间复杂度将是指数级的。

而我们之前定义的 $OPT(i,w)$ 是指在前 i 个物品中选且包的容量最大为 w 时的装的物品价值最大。此时只考虑前 i 个物品，并不考虑前 i 个物品到底是哪 i 个物品，则该子问题只有 $O(n)$ 个子问题，可见之前对问题的表示比上述表示要简洁的多。

可见子问题的表示对动态规划的时间复杂度有很重要的影响，如果定义的不够好将导致时间复杂度特别高。

3 序列的连配问题

3.1 问题描述:

关于序列连配问题的实际需求比较多，比如生物信息领域中DNA序列的匹配问题。生活中常见的例子就是我们在word里面打英文单词，如果有错误，它会划红线进行提醒，有时甚至可以自动修改。

关于判断输入的英文单词是否错误，我们可以在系统后台存储一个词典。如果打出的英文单词不在该词典中，则该词就判断为错误，但是如何自动化修改呢？如何知道该词与词典中的哪个词最近呢？比如我们敲入的是teh这个单词，如何自动修改成the这个单词呢?

关键问题在于如何评判键盘敲入的单词与词典背后的单词的相似程度呢？比如我们敲入的是OCURRANCE这个单词，我们通过插入一个字母C和修改A这个字母为E变成词典中的OCCURRENCE这个单词，即我们可以经过有限次的修改，删除，添加操作，可以将敲出的单词变成词典中某个正确的单词。

3.2 形式化定义：

因此我们引出序列连配问题的形式化定义：

- **Input:**
  Two sequence S and T, $|S| = m$, and $|T| = n$;
- **Output:**
  To identify an alignment of S and T that maximizes a scoring function.

Note: for the sake of simplicity, the following indexing schema is used: $S = S_1S_2...S_m$.

这里Alignment是左右对齐的意思，经常表示产生式过程，表示上面的单词S是怎样通过下面的单词T变成的。

我们的目标就是通过添加空格使上下序列一样长，例如在S中添加一些空格变成S'，T中添加一些空格变成T'。

如果变了之后T'的第i个是空格，则表示S该位置上的字母是T通过插入操作得到的。

如果变了之后S'的第i个是空格，则表示S是将T中该位置的字母删除了。

因此Alignment就是记录了T是经过怎样一系列的变化变成S的。

现在假设我们有两个序列： S : teh 和 T: the

我们认为敲的S与字典里面的T是最相似的。为什么最相似呢，肯定是我们对单词之间的相似度进行了打分，那怎么打分的呢?

我们假设有如下打分原则：

$$d(S,T) = \sum_{i=1}^{|S'|} \delta(S'[i], T'[i])$$

这里的 $\delta(a,b)$ is:

1. Match: $+1$, e.g. $\delta(\text{`}C', \text{`}C') = 1$.

2. Mismatch: -1, e.g. $\delta(\text{`}E', \text{`}A') = -1$.

3. Ins/Del: -3, e.g. $\delta(\text{`}C', \text{`}-') = -3$.

$\delta(a,b)$ 表示如果当前S位置上的字母与T位置上的字母match到了，则+1分，如果没有match到则-1分，少敲则-3分。这里这样设计打分原则只是为了教学方便。

一个问题可以建模成组合优化或者统计问题，这两个问题是密切相关的。

使用Alignment，我们可以知道你最想敲的是什么。我们系统后台有个词典，词典中的每个单词都与你想敲的单词进行相似度计算，则找到与你敲的单词最相似的就是你最想敲的，也就是得分最高的那个单词。

具体事例：

例如下面一个例子：

1. $T =$ "OCCUPATION":

```
S': OC_URRA___NCE
   || |  |   |
T': OCCU_PATION__
```

$d(S', T') = 1+1-3+1-3-3-1+1-3-3-3+1-3-3 = -28.$

2. $T =$ "OCCURRENCE":

```
S': O_CURRANCE
     | |||| |||
T': OCCURRENCE
```

$d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4.$

我们对单词"OCCUPATION"和"OCURRANCE"通过加空格使得它们等长，然后通过计分的方案得到"OCCUPATION" 与"OCURRANCE"的得分是-28 分。

同样假设我们在词典中遇到了另外一个词"OCCURRENCE"，和上面一样操作，最后得到"OCCURRENCE"与"OCURRANCE" 的得分是4分。

因为单词“OCCURRENCE”得分比较高，所以我们猜测敲错的单词“OCURRANCE”极有可能是从“OCCURRENCE”这个单词过来的。

另外我们还能知道你是怎么敲的?

即使我们找到与“OCURRANCE”最相似的是“OCCURRENCE”这个单词，我们仍然有很多种加空格的方案使得他们等长。

1. Alignment 1:

```
S': O_CURRANCE
    |  ||||  |||
T': OCCURRENCE
```

$$d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4.$$

2. Alignment 2:

```
S': O_CURR_ANCE
    |  ||||   |||
T': OCCURRE-NCE
```

$$d(S', T') = 1 - 3 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1.$$

这两种加空格的方案都可以使得单词“OCURRANCE”与”OCCURRENCE“等长，但是得分却是不同的。通过Alignment 我们可以知道单词”OCURRANCE“中的字母A 最可能是把字母E 敲错了得到的，而不是少敲了一个E且多敲了一个A得到的。

因此我们知道Alignment十分重要，不仅可以识别出字典中哪个单词与你敲的单词相似度最大，而且可以推测出你是怎么敲错的。

3.3 动态规划求解：

现在我们对问题进行总结：

给我们两个字符串S与T，问如何通过加空格使得得分最高。

采用我们之前求解问题的一般思路：首先给我们的字符串很长，我们不好解决，我们可以将该字符串分成更小的字符串，我们的solution是通过加空格使两个字符串对齐的方案。我们把求解过程当做一系列的决策，对每个决策部分我们决定 $s[i]$ 是怎样通过 $T[j]$ 变来的，比如$OCURRANCE$是S，$OCCURRENCE$是T，我们尝试考虑 S 是经过什么样的操作变化到 T 的。

我们首先考虑S单词最后的字母E，那S的E是通过T怎么变化得到的呢?

(1)从T的最后一个单词直接过来的，即 match 到了。

(2)这个E是我们多敲的，即T通过插入操作变化过来的。

(3)这个E是我们少敲的，即T通过删除操作变化过来的。

我们的总体目标是给我们一个词，我们想知道该词是否与词典中的某个词相似，给我们的词太长不好解决，可以先考虑最后一个字母，看它有哪些来源。

如果是多敲的，则S除了最后一个字母剩余的部分，是由T的整体变化而来的。则问题变成假设我们找到一个最优解，现在我们只考虑最后一个决策，即S的最后一个字母是怎么过来的呢? 我们由上面的分析知道，其来源三种可能。

(1)如果S[m]与T[n]形成匹配，则子问题变成了 $S[1,...,m-1]$ 与 $T[1,..,n-1]$ 的对齐问题。

(2)如果S[m]与空格匹配，则表示S[m] 是T通过插入操作过来的，则子问题变成了 $S[1,...,m-1]$ 与 $T[1,..,n]$ 的对齐问题。

(3)如果T[n]与空格匹配，则表示S[m] 是T通过删除操作过来的，则子问题变成了 $S[1,...,m]$ 与 $T[1,..,n-1]$ 的对齐问题。

则我们可以总结得到子问题的一般形式就是S的前缀与T的前缀的对齐方案，我们将该问题的最优解记做$OPT(i,j) = S[1...i]\ alignment\ T[1...j]$.

因此我们可以得到如下递归表达式:

$$OPT(i,j) = \max \begin{cases} \delta(S_i, T_j) + OPT(i-1, j-1) \\ \delta(\text{`\_'}, T_j) + OPT(i, j-1) \\ \delta(S_i, \text{`\_'}) + OPT(i-1, j) \end{cases}$$

即枚举当前单元的三种可能来源，在其中取最大值就可以。

因此有如下伪代码:

NEEDLEMAN_WUNCH(S,T)

1: **for** $i = 0$ to m; **do**
2: $\quad OPT[i,0] = -3 * i$;
3: **end for**
4: **for** $j = 0$ to n; **do**
5: $\quad OPT[0,j] = -3 * j$;
6: **end for**

7: **for** $i = 1$ to m **do**
8: **for** $j = 1$ to n **do**
9: $OPT[i, j] = \max\{OPT[i-1, j-1] + \delta(S_i, T_j), OPT[i-1, j] - 3, OPT[i, j-1] - 3\}$;
10: **end for**
11: **end for**
12: **return** $OPT[m, n]$;

Note: the first row is introduced to describe the alignment of prefixes $T[1..i]$ with an empty sequence ϵ, so does the first column.

我们通过伪代码可以得到下面的表格，S中每个字母一列，T中每个字母一行，其中单元 (i, j) 表示$S[1, ..., j]$ 与 $T[1, ..., i]$ 对齐所能得到的最大的分数，比如图中的单元(1,2):

T: \ S:	’’	O	C	U	R	R	A	N	C	E
’	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

Score: d("OC", "O") = max { d("OC","") −3 (=-9); d("O","") −1 (=-4); d("O","O") −3 (=-2) }

Alignment: S'= OC
T'= O-

单元(1,2)表示将T的前缀O敲成S的前缀OC时的Alignment情况，根据之前的分析我们知道有上述三个来源，即$(1,2)$单元仅与 $(1,1),(0,2),(0,1)$ 这三个单元有关。

S:	’’	O	C	U	R	R	A	N	C	E
T:’’	0	−3	−6	−9	−12	−15	−18	−21	−24	−27
O	−3	1	−2	−5	−8	−11	−14	−17	−20	−23
C	−6	−2	2	−1	−4	−7	−10	−13	−16	−19
C	−9	−5	−1	1	−2	−5	−8	−11	−12	−15
U	−12	−8	−4	0	0	−3	−6	−9	−12	13
R	−15	−11	−7	−3	1	1	−2	−5	−8	−11
R	−18	−14	−10	−6	−2	2	−	−3	−6	−9
E	−21	−17	−13	−9	−5	−1	1	−1	−4	−5
N	−24	−20	−16	−12	−8	−4	−2	2	−1	−4
C	−27	−23	−19	−15	−11	−7	−5	−1	3	0
E	−30	−26	−22	−18	−14	−10	−8	−4	0	4

同样的我们可以得到最后一个单元的来源情况。

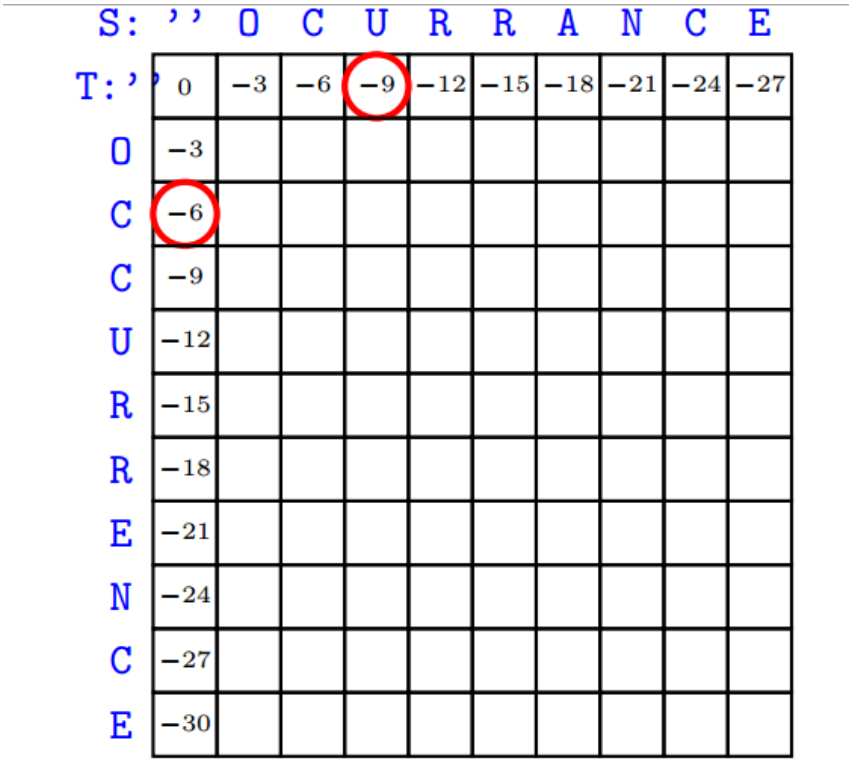
Score: `d("OCURRANCE", "OCCURRENCE") =` $\max \begin{cases} \text{d(“OCURRANCE“,“OCCURRENC“)} & -3 & (=\text{-3}) \\ \text{d(“OCURRANC“,“OCCURRENC“)} & +1 & (=4) \\ \text{d(“OCURRANC“,“OCCURRENCE“)} & -3 & (=\text{-3}) \end{cases}$

Alignment: `S'= O-CURRANCE`

从上面的例子我们可以知道中间的单元都是由相邻的三个单元变换而来的，那么第一行和第一列是怎么来的呢?

第一行是说T是空时，是怎么一步步从””变化到 $S[1,..,n]$ 的。相当于每次都需要插入一个字母变成与S 对应的字母，即每增加一个字母就扣三分。

第一列是说字典里有个词T，你敲成空字符串了，即表示T是怎么样变化到空字符串的，所以每次都少敲了，因此将T变成S需要每次都删除一个字母，即每删除一个字母就扣三分。



Score: d("OCU", "") = -9
Alignment: S'= OCU
T'= ---

Score: d("", "OC") = -6
Alignment: S'= --
T'= OC

3.4 构建最优解方案:

那么我们如何知道最后的得分是怎么来的呢? 常用的想法就是回溯，我们的最后的得分4本身有三个来源，我们开个表格记录该单元的得分是从三个来源中具体的哪个单元过来的，不断的向前回溯就可以找到一个对应的alignment，使得S变成T。

但是我们经常对分的评判不是很准确，这时候可以随机回溯多次，然后取平均，这样可以取一个比较常见的模式，使得我们最终得到的结果更稳定些。

现在对我们的问题再陈述下：我们在文档里面敲了一个词，该词不在词典中，这时候我们知道自己敲错了，但是我们想知道是怎么错的，我们想用这个分来衡量词典中的词经过最少的几次操作才能变成我们敲的单词S呢。