海量资源：666java.com

# (更多)动态规划

海量资源：666java.com

# 上节课讲到

- 介绍了一些基于动态规划的图算法
- **Bellman-Ford**: 解决单源最短路径问题 (带负权重边)
  - 子问题: $\mathbf{d^{(k)}[b]}$ = 从 s 至 b 经过至多 k 条边的最短路径代价
  - 递归关系: $\mathbf{d^{(k)}[b] \leftarrow \min\{\, d^{(k-1)}[b],\ \min_a \{d^{(k-1)}[a] + w(a,b)\}\,\}}$
- **Floyd-Warshall**: 解决所有结点对间的最短路径问题 (带负权重边)
  - 子问题: $\mathbf{D^{(k)}}$ = 所有结点对间仅通过结点 {1, …, k} 的最短路径代价
  - 递归关系: $\mathbf{D^{(k)}[v,w] = \min\{\, D^{(k-1)}[v,w],\ D^{(k-1)}[v,k] + D^{(k-1)}[k,w]\,\}}$

海量资源：666java.com

# 这节课的内容

- 更多动态规划问题
  - 最长公共子序列 Longest Common Subsequence
  - 无界限背包问题 Unbounded knapsack problem
  - 0/1 背包问题 0/1 knapsack problem

海量资源：666java.com

# 使用动态规划

使用 DP 设计算法的一般步骤

海量资源：666java.com

# (接上回)

## 动态规划的要点:

**最优子结构:** 一个问题的最优解包含其子问题的最优解.

e.g. $\mathbf{d^{(k)}[b] = \min\{ d^{(k-1)}[b], \min_a\{d^{(k-1)}[a] + w(a, b)\} \}}$

**重叠子问题:** 子问题大量重叠!

可以通过缓存已求解子问题的解来减少算法运行时间.

(也被称为 “记忆化”)

e.g. **$d^{(k)}$ 中很多项的计算都需要知道 $d^{(k-1)}[v]$**

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# 最长公共子序列

给定序列 X 和 Y, 寻找其最长公共子序列的长度

海量资源：666java.com

# 最长公共子序列

序列 **Z** 是 **X** 的 **子序列** 当且仅当 可以通过删除 **X** 中的项来得到 **Z**

**BDFH** 是 **ABCDEFGH** 的子序列

**C** 是 **ABCDEFGH** 的子序列

**ABCDEFGH** 是 **ABCDEFGH** 的子序列

称序列 **Z** 是 **X** 和 **Y** 的**最长公共子序列 LONGEST COMMON SUBSEQUENCE (LCS)**
当且仅当 **Z** 同时是 **X** 和 **Y** 的子序列
且所有长度大于 **Z** 的序列不是**X** 或 **Y** 的子序列

**ABDFGH** 是

**ABCDEFGH** 和 **ABDFGHI** 的 LCS

海量资源：666java.com

# 最长公共子序列

序列 **Z** 是 **X** 的 **子序列** 当且仅当 可以通过删除 **X** 中的项来得到 **Z**

**BDFH** 是 **ABCDEFGH** 的子序列

**C** 是 **ABCDEFGH** 的子序列

称序列

**E (LCS)**

**题目**: 给定序列 X 和 Y, 求其 LCS, Z 的长度.

(稍后我们在讨论如何得到序列 Z 本身)

**ABDFGH** 是

**ABCDEFGH** 和 **ABDFGHI** 的 LCS

海量资源：666java.com

# LCS 的应用

### 生物信息学

判断某些 DNA 与特定蛋白质序列的相似度

### 计算语言学

提取单词/单词组的相似性，并确定单词之间的关系

### Unix 中的 diff 命令

判断 2 个输入文件内容的差异

等等...

海量资源：666java.com

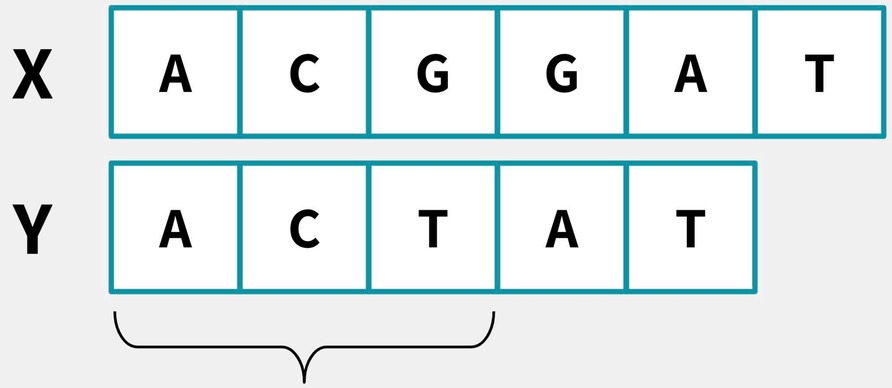
# LCS: 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 1: 最优子结构

**子问题:** 寻找 X 和 Y *前缀* 中的 LCS 长度.




将前缀 **ACT** 记为 $Y_3$

**例:**

**C[2,3] = 2** ($X_2$ 与 $Y_3$ 的 LCS 为 AC)
**C[5,4] = 3** ($X_5$ 与 $Y_4$ 的 LCS 为 ACA)

令 **C[i, j] = length_of_LCS($X_i$, $Y_j$)**

**为什么选择这样的子问题?**

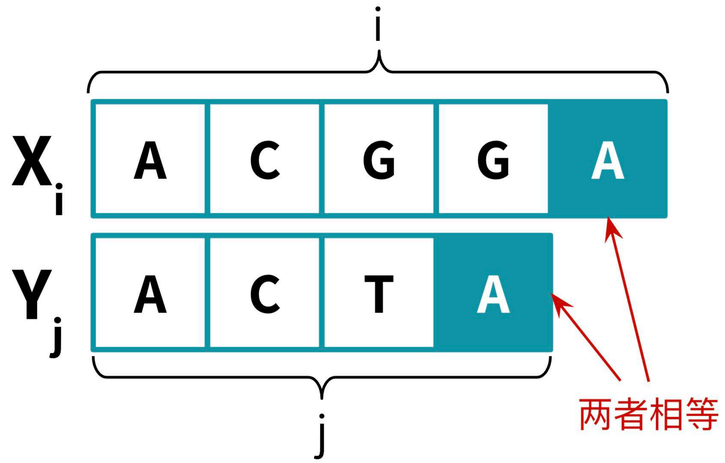
海量资源：666java.com

# STEP 1: 最优子结构

令 C[i, j] = length_of_LCS($X_i$, $Y_j$)

考虑前缀的末位, X[i] 与 Y[j]. 有 2 种情况:

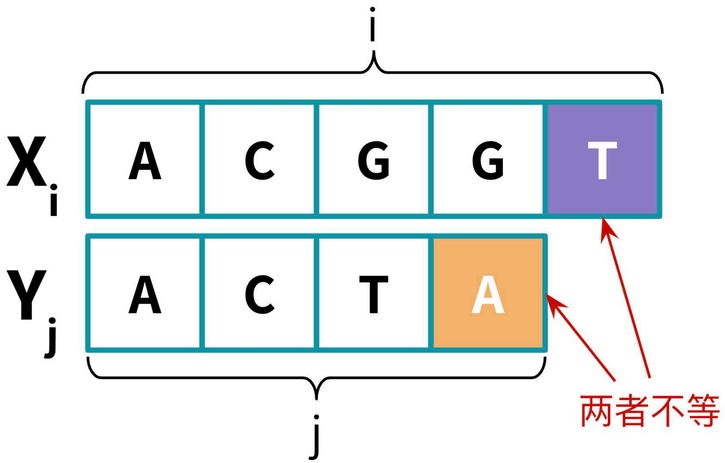
## Case 1: X[i] = Y[j]




则, **C[i, j] = 1 + C[i–1, j–1]**

因为 LCS($X_i$, $Y_j$) = LCS($X_{i-1}$, $Y_{j-1}$) 在加上新字符 **A**.

## Case 2: X[i] ≠ Y[j]




则, **C[i, j] = max{ C[i–1, j], C[i, j–1] }**

选择 **A** 有可能得到 “匹配”: LCS($X_i$, $Y_j$) = LCS($X_{i-1}$, Y)

选择 **T** 有可能得到 “匹配”: LCS($X_i$, $Y_j$) = LCS($X_i$, $Y_{j-1}$)

海量资源：666java.com

# LCS: 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

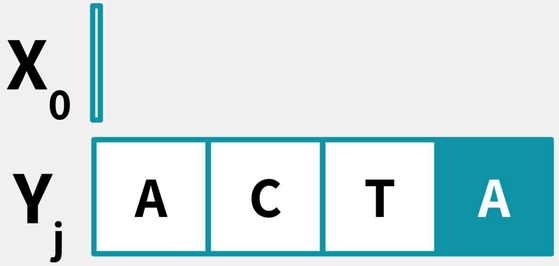
海量资源：666java.com

# STEP 2: 递归求解公式

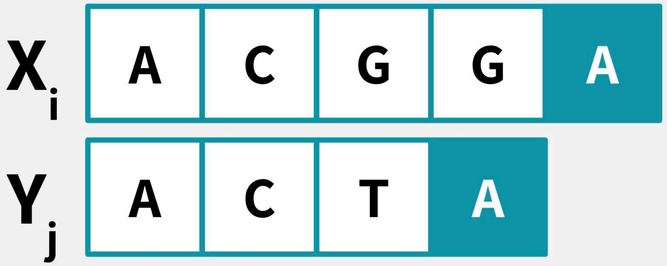
上述方法的递归求解公式为:

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i-1, j], C[i, j-1] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

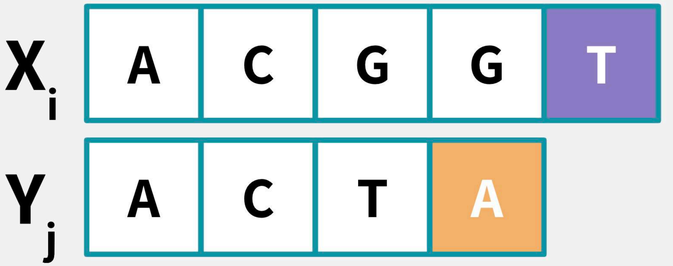
**CASE 0** (基本情况)




**CASE 1**




**CASE 2**



海量资源：666java.com

# LCS: 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 3: 实现动态规划

将子问题的解存在表 C[i, j] 中 (缓存)

根据递归求解公式, 编写代码: 初始化基本情况 & 推导中间项

**采用自底向上 bottom-up 的实现方式. 为什么?**

我们需要先解决短前缀上的情况, 再不断推广到更长的前缀上 (最终的答案储存在 C[m, n] 其中 |X| = m, |Y| = n)

海量资源：666java.com

# STEP 3: 实现动态规划

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i-1, j], C[i, j-1] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

len(X) = m & len(Y) = n

```
LCS(X,Y):
    Initialize an (m+1) x (n+1) 0-indexed array C
    C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
    for i = 1,...,m  and j = 1,...,n:
        if X[i] = Y[j]:
            C[i,j] = C[i-1,j-1] + 1
        else:
            C[i,j] = max{ C[i,j-1], C[i-1,j] }
    return C[m,n]
```

确保基本情况被初始化

Case 1

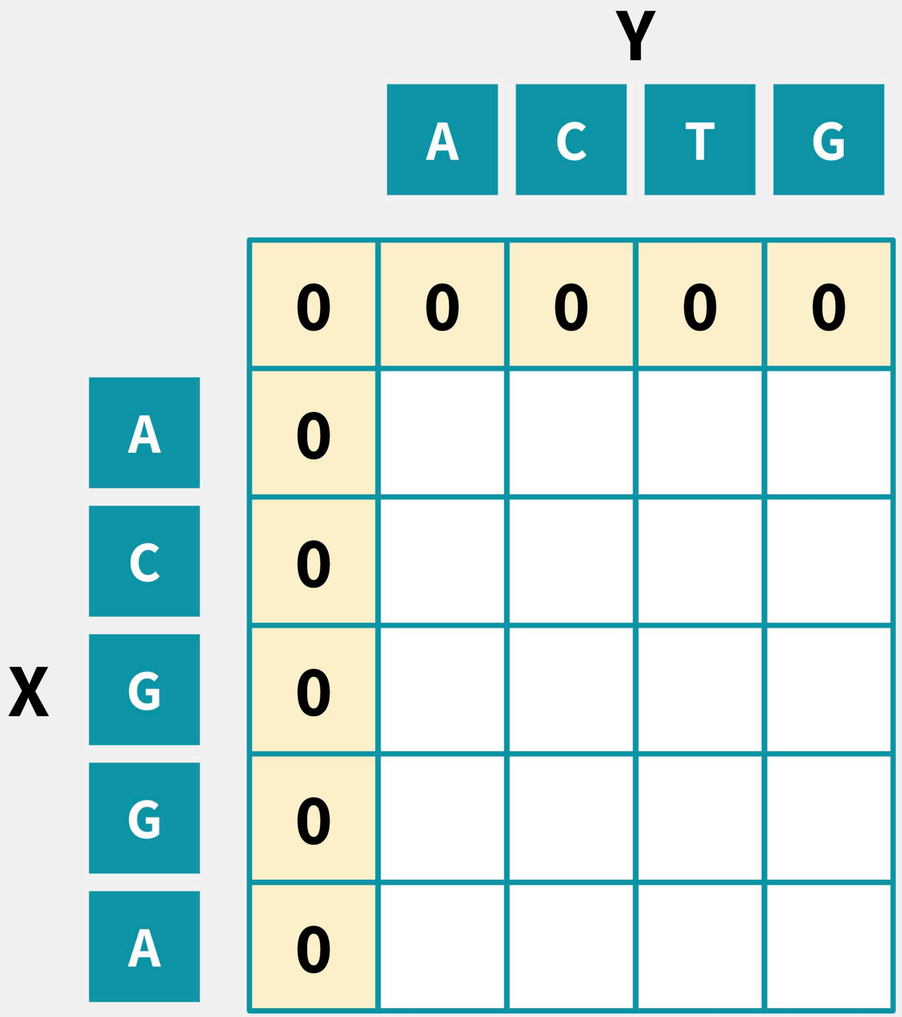
Case 2

**运行时间: O(mn)**

填充 C 中 mn 项每一项用时为常数

海量资源：666java.com

# 例

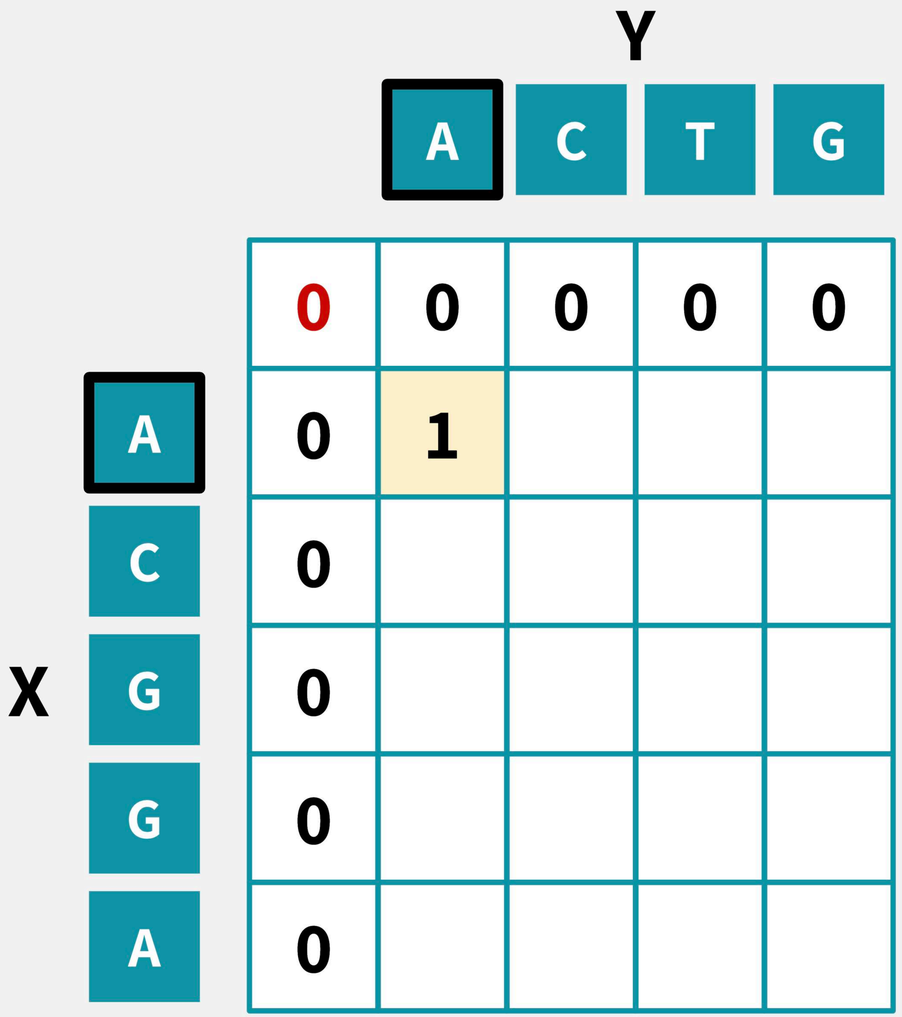



```
Initialize an (m+1) x (n+1) 0-indexed array C
   C[i,0] = C[0,j] = 0 for all i=0,...,m and j=0,...,n
```

**先填充基本情况**

海量资源：666java.com

# 例



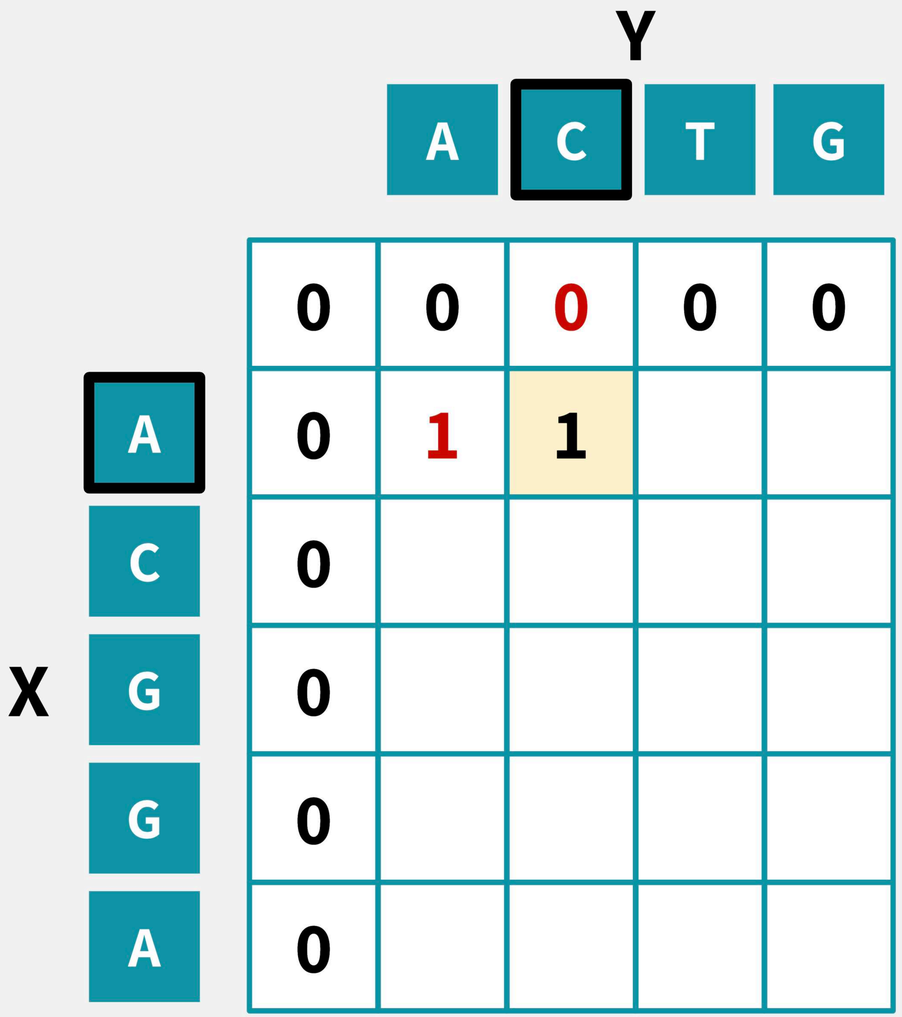

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] = Y[j]**

C[i,j] = C[i-1,j-1] + 1

海量资源：666java.com

# 例



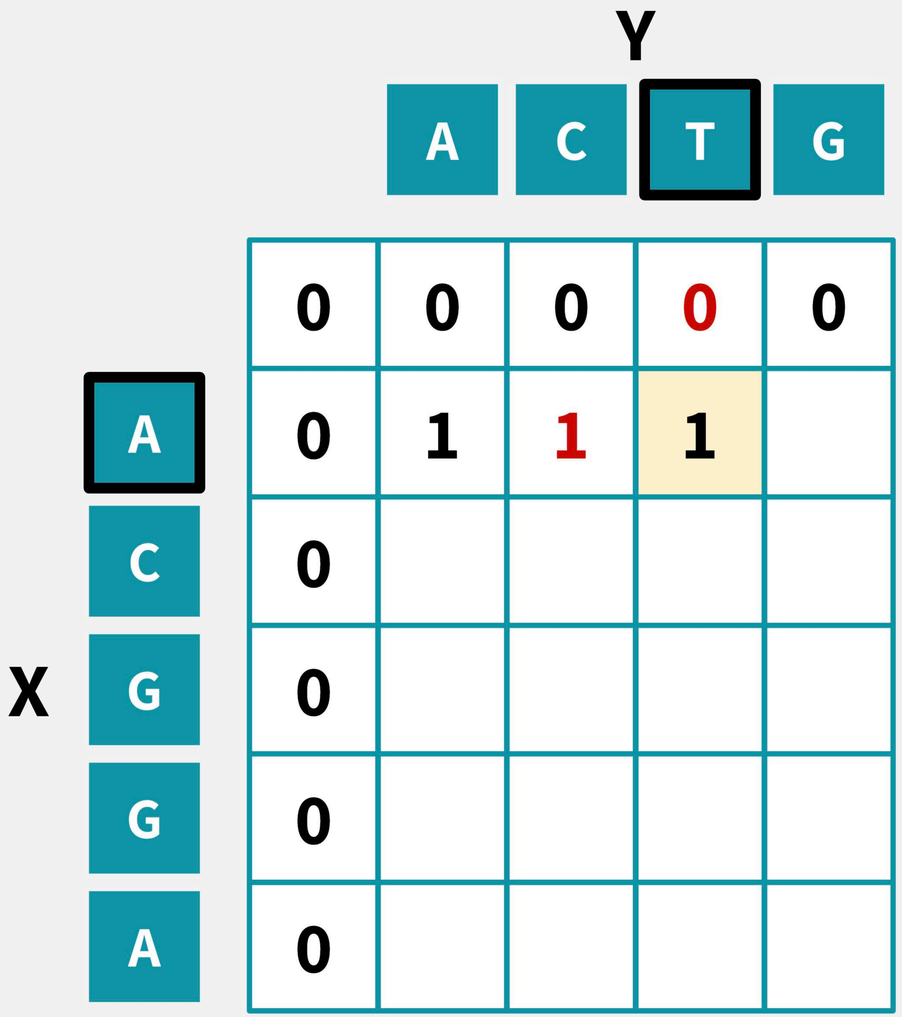

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例



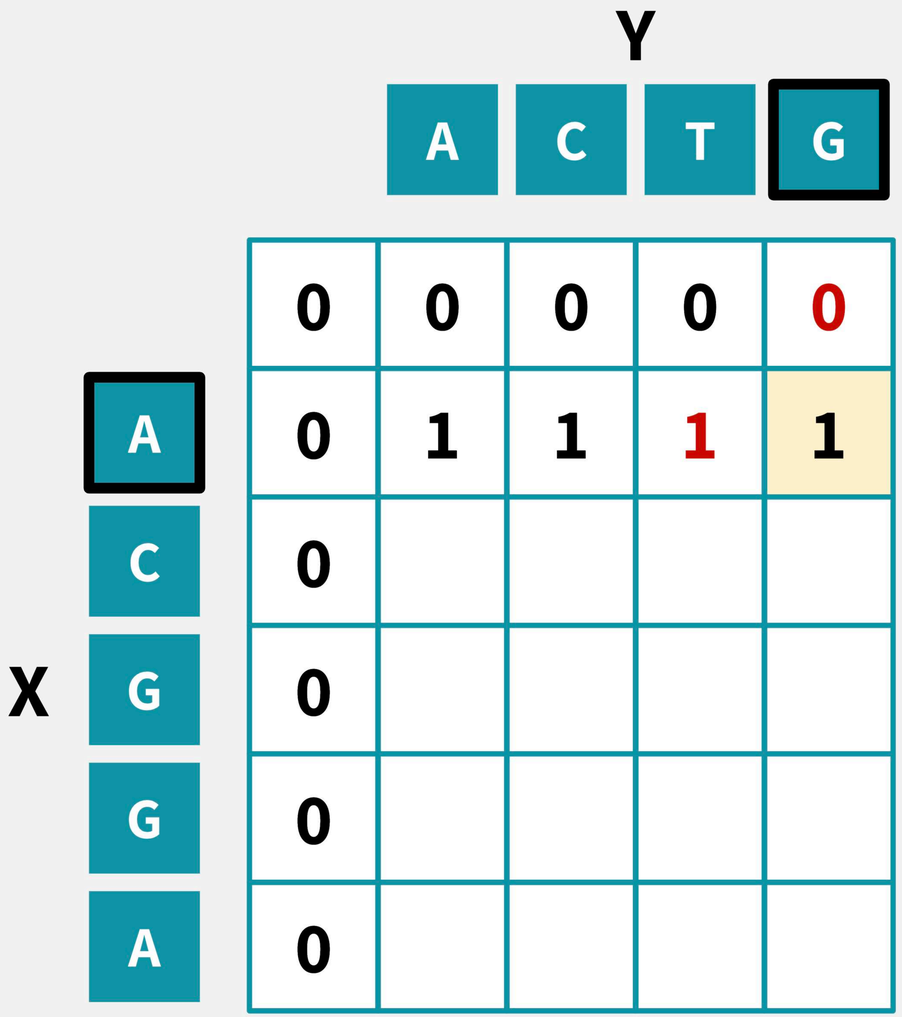

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例




```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | | | |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

Y

X

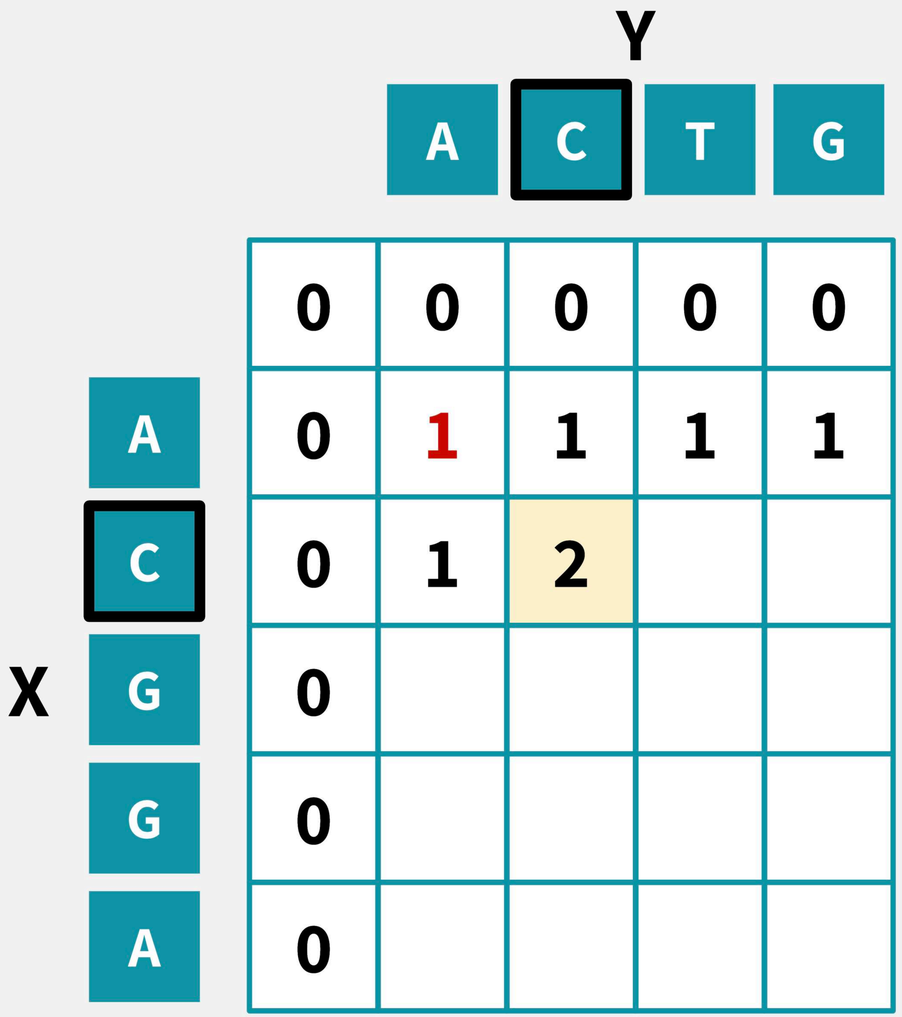
```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例



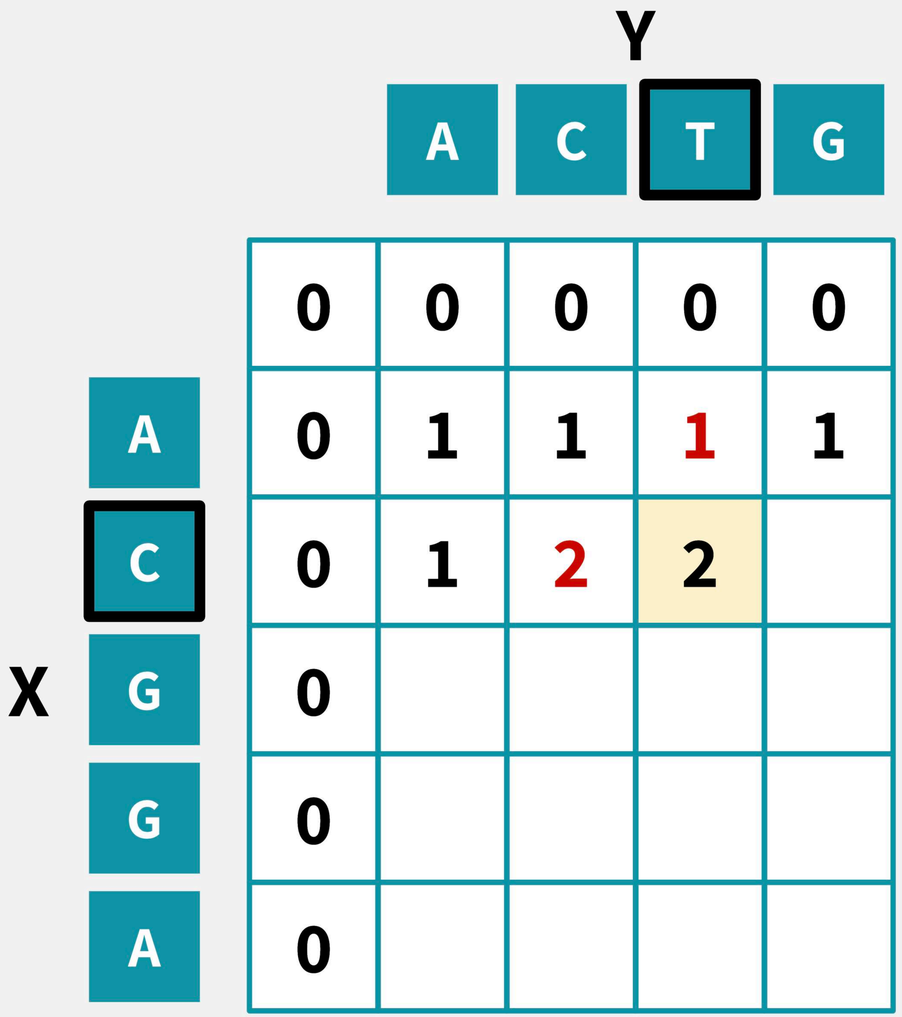

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] = Y[j]**

C[i,j] = C[i-1,j-1] + 1

海量资源：666java.com

# 例




```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | | | | |
| G | 0 | | | | |
| A | 0 | | | | |

Y

X

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | | | |
| G | 0 | | | | |
| A | 0 | | | | |

Y

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | 2 | | |
| G | 0 | | | | |
| A | 0 | | | | |

Y

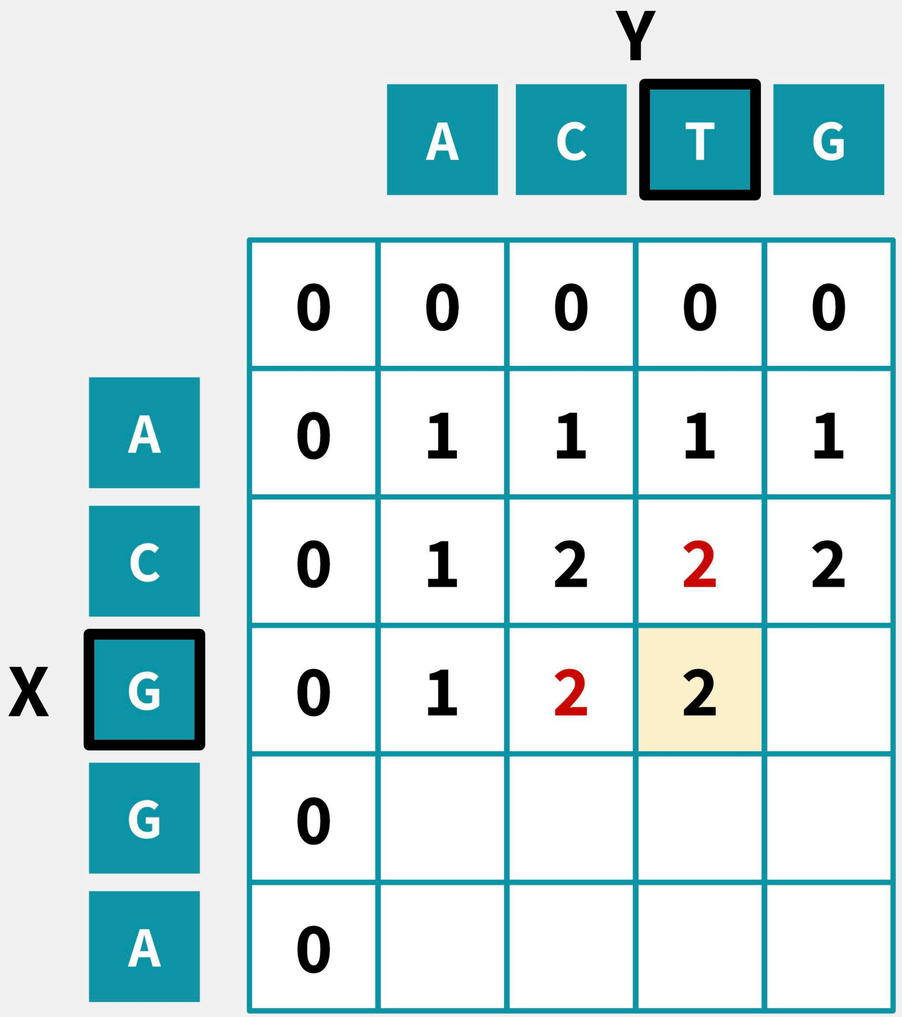
```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例




```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | | | | |
| A | 0 | | | | |

Y

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] = Y[j]**

C[i,j] = C[i-1,j-1] + 1

海量资源：666java.com

# 例

| | | | A | C | T | G |
|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | | | |
| | A | 0 | | | | |

Y

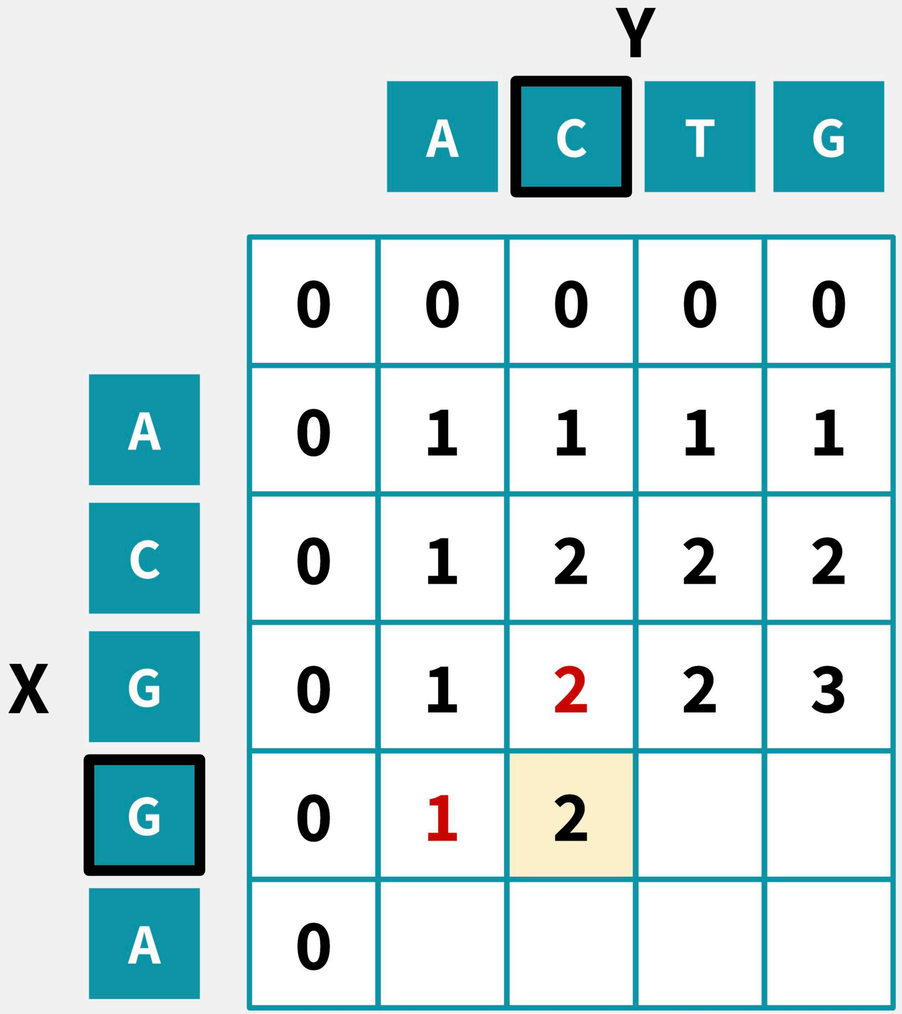
```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例




```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | |
| A | 0 | | | | |

Y

X

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | Y | | | |
|---|---|---|---|---|---|
| | | A | C | T | G |
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | | | | |

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] = Y[j]**

C[i,j] = C[i-1,j-1] + 1

海量资源：666java.com

# 例

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G | |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | | | |

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] = Y[j]**

C[i,j] = C[i-1,j-1] + 1

海量资源：666java.com

# 例

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G | |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | | |

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | | A | C | T | G |
|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | |

Y

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | | A | C | T | G |
|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

Y

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**X[i] ≠ Y[j]**

C[i,j] = max{ C[i,j-1], C[i-1,j] }

海量资源：666java.com

# 例

| | | A | C | T | G |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

Y

X

```
for i = 1,...,m  and j = 1,...,n:
   if X[i] = Y[j]:
      C[i,j] = C[i-1,j-1] + 1
   else:
      C[i,j] = max{ C[i,j-1], C[i-1,j] }
```

**所以 X 和 Y 的最长公共子序列的长度为 3.**

海量资源：666java.com

# LCS: 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的值/代价, 但是问题本身需要求解最优的路径/子集/子数组/等等. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G |
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

**假设我们需要得到实际的 LCS.**

我们如何通过刚才填充的表 C 来复原实际的 LCS?

**从 C[m,n] 开始倒着检查**

**我们是如何得到 3 这个最终结果的!**

如果发现 X 中的字符与 Y 中的字符相等, 那么该字符属于 LCS 并且我们需要按*对角线方向*往左上方移动一格.

否则, 如果字符不相等, 我们按照更大值的方向移动

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | | A | C | T | G |
|---|---|---|---|---|---|---|
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

Y

```
RECOVER_LCS(X, Y, C):
    // C is already filled out
    L = []
    i = m
    j = n
    while i > 0 and j > 0:
        if X[i] = X[j]:
            append X[i] to the beginning of L
            i = i-1
            j = j-1
        else if C[i,j] = C[i,j-1]:
            j = j-1
        else:
            i = i-1
    return L
```

这一部分的运行时间为 O(m+n)

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

LCS of X and Y:

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

**X[5] ≠ Y[4]**

无需更新 LCS.

向上移动 → C[4,4]

**LCS of X and Y:** [ ] [ ] [ ]

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

**X[4] = Y[4]**

将 “G” 添加至 LCS

朝左上方移动 → C[3,3]

**LCS of X and Y:** [ ] [ ] G

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | |
|---|---|---|---|---|---|
| | | A | C | T | G |
| | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 2 | 2 | 2 |
| X G | 0 | 1 | 2 | 2 | 3 |
| G | 0 | 1 | 2 | 2 | 3 |
| A | 0 | 1 | 2 | 2 | 3 |

**X[3] ≠ Y[3]**

无需更新 LCS.

向上移动 → C[2,3]

(由于表中数值相等, 朝左走也是可以的. 由于 LCS 不唯一, 不同的走法**可能**会得到不同的 LCS. 在这个例子中, 只有一种 LCS)

**LCS of X and Y:** [ ] [ ] G

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G | |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

**X[2] ≠ Y[3]**

无需更新 LCS.

向左移动 → C[2,2]

**LCS of X and Y:** [ ] [ ] G

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G | |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

**X[2] = Y[2]**

将 “C” 添加至 LCS.

朝左上方移动 → C[1,1]

**LCS of X and Y:** [ ] C G

海量资源：666java.com

# STEP 4: 寻找实际的 LCS

| | | Y | | | | |
|---|---|---|---|---|---|---|
| | | A | C | T | G |
| | | 0 | 0 | 0 | 0 | 0 |
| | A | 0 | 1 | 1 | 1 | 1 |
| | C | 0 | 1 | 2 | 2 | 2 |
| X | G | 0 | 1 | 2 | 2 | 3 |
| | G | 0 | 1 | 2 | 2 | 3 |
| | A | 0 | 1 | 2 | 2 | 3 |

**X[1] = Y[1]**

将 “A” 添加至 LCS.

结束!

**LCS of X and Y:** A C G

海量资源：666java.com

# LCS: 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.
5. **还能更好吗?** 是否浪费了空间? 其他可以优化的地方?
   (无论能否优化, 这都是很重要且值得考虑的一步)!

海量资源：666java.com

# STEP 5: 还能更好吗?

- 如果我们只关心 LCS 的长度, 那么没有必要储存整张表 (只需要储存 2 行).
- 如果我们需要得到完整的 LCS, 那么需要储存整张表.
- 如何优化运行时间 O(mn)?
  - 如果字符取值范围有限, 运行时间可以优化至对数级别 (使用 Method of Four Russians).
  - LCS 问题是 *NP-hard*, 所以如何优化是一个开放问题

海量资源：666java.com

# 题外话: NP 完全性

**P 类问题** 多项式时间内可以解决的问题 (算得很快的问题)

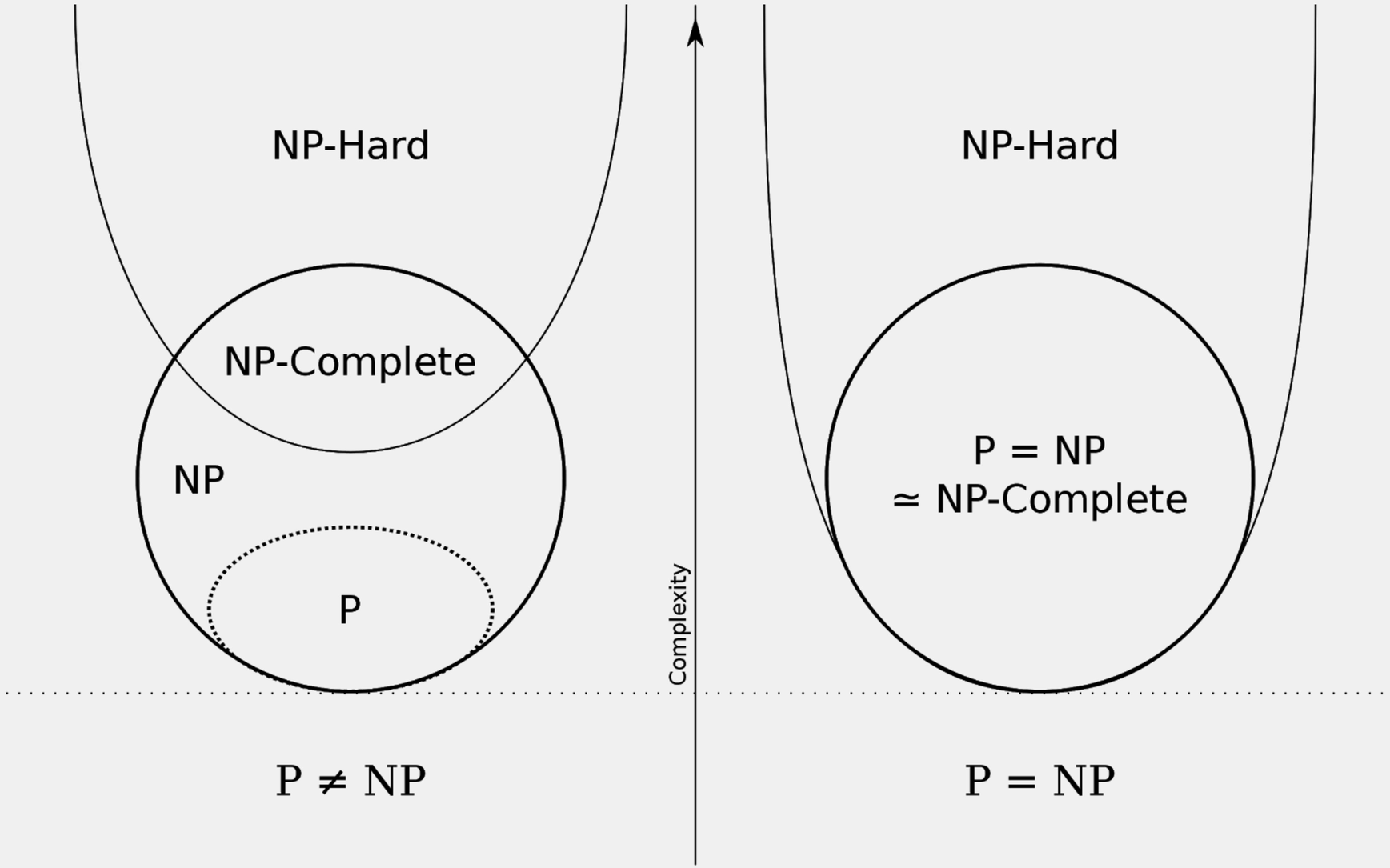
**NP 类问题** 多项式时间内可以被证明的问题 (算起来不确定快不快的问题, 但是我们可以快速验证这个问题的解)

**NP-complete 类问题** 属于 NP 问题, 且属于 NP-hard 问题

**NP-hard 类问题** 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 (比 NP 问题都要难的问题)

海量资源：666java.com

# 题外话: NP 完全性



海量资源：666java.com

# 背包问题

如何将物品装入背包中使得总价值最大?

海量资源：666java.com

# 背包问题

**如何将物品装入背包中使得总价值最大?**

共有 n 件物品, 物品由重量和价值.

| Item: | 🥑 | 🧲 | 🍇 | 🐨 | 🚙 |
|---|---|---|---|---|---|
| Weight: | **6** | **2** | **4** | **3** | **11** |
| Value: | **20** | **8** | **14** | **13** | **35** |

这是我们的背包, 可容纳的物品重量有限:



Capacity: **10**

海量资源：666java.com

# 背包问题: 两个版本



Capacity: **10**

| Item: | 🥑 | 🧲 | 🍇 | 🐨 | 🚗 |
|---|---|---|---|---|---|
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

## 无界限背包

所有物品均有无限个.

如何使得包中物品总价值最大?

🧲 🧲 🐨 🐨

Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

## 0/1 背包

每个物品只有一份.

如何使得包中物品总价值最大?

🧲 🍇 🐨

Total weight: **2 + 4 + 3 = 9**

Total value: **8 + 14 + 13 = 35**

海量资源：666java.com

# 无界限背包问题

每个物品有无数个

海量资源：666java.com

# 问题描述

## 无界限背包

所有物品均有无限个.

如何使得包中物品总价值最大?



Capacity: **W**

| Item: |  |  |  | ... |  |
|---|---|---|---|---|---|
| Weight: | $w_1$ | $w_2$ | $w_3$ | | $w_n$ |
| Value: | $v_1$ | $v_2$ | $v_3$ | | $v_n$ |

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 1: 最优子结构

**子问题:**

小背包上的无界限背包问题

**K[x]** = 容量为 **x** 的背包中可以装入的最大价值



首先解决小背包上的问题



然后考虑更大的背包



更大的背包

海量资源：666java.com

# STEP 1: 最优子结构

**子问题:**

小背包上的无界限背包问题

**K[x]** = 容量为 **x** 的背包中可以装入的最大价值

为什么可以这样选取子问题, 以及如何通过子问题的解得到原问题的最优解 K[x]?
我们希望通过穷举所有可能情况来得到最优解:

**背包容量为 x 时. 我们应该放入哪件物品?**

如果放入重量为 $w_i$ 的物品 **i**, 包中最大价值为物品 **i** 的价值 $v_i$, *加上容量为* **x** - $w_i$ *的背包装入的最大价值* (即装入物品 **i** 后剩余的容量).

| Item: |  |  |  | ... |  |
|---|---|---|---|---|---|
| Weight: | $w_1$ | $w_2$ | $w_3$ | | $w_n$ |
| Value: | $v_1$ | $v_2$ | $v_3$ | | $v_n$ |

海量资源：666java.com

# STEP 1: 最优子结构

**子问题:**

小背包上的无界限背包问题

K[x] = 容量为 x 的背包中可以装入的最大价值

为什 K[x]?

如果放入重 入的最大价

**基本思路:**

对于每一个可以放入背包的物品 i,
考虑该选择能得到的最大价值:

**该选择能得到的最大价值 = [$v_i$] + [容量 x–$w_i$背包能装入的最大价值]**

选择所有可能中的最大值

| | | | | | |
|---|---|---|---|---|---|
| Weight: | $w_1$ | $w_2$ | $w_3$ | ••• | $w_n$ |
| Value: | $v_1$ | $v_2$ | $v_3$ | | $v_n$ |

海量资源：666java.com

# STEP 1: 最优子结构

**子问题:**

小背包上的无界限背包问题

**另一种表达:**

令 K[x] 为容量为 x 的背包能装入的最大价值,
并假设包含至少一个物品 i.

**背包中其他物品的重量不能超过 $x-w_i$, 且剩余物品的价值构成子问题的最优解**

(否则, 可以通过替换剩余物品中的某些来得到更大的 K[x])

如果

大价

| | | | | | |
|---|---|---|---|---|---|
| Weight: | $w_1$ | $w_2$ | $w_3$ | ... | $w_n$ |
| Value: | $v_1$ | $v_2$ | $v_3$ | | $v_n$ |

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 2: 递归求解公式

**K[x]** = 容量为 **x** 的背包中可以装入的最大价值

递归求解公式为:

$$K[x] = \begin{cases} 0 & \text{if there are no i where } w_i \le x \\ \max_i \{ K[x-w_i] + v_i \} & \text{otherwise} \end{cases}$$

穷举所有物品 i
($w_i \le x$) 得到的最大值

小背包中的最优解



物品 i 的价值



海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 3: 实现动态规划

使用一个一维数组来储存子问题的解 K[x]

根据递归求解公式, 编写代码: 初始化基本情况 & 推导中间项

**采用自底向上 bottom-up 的实现方式. 为什么?**

与前一个问题类似, 我么需要先计算更小背包中的问题才能得到原问题的解,
所以需要从 K[0] 开始不断扩展问题规模直至得到最终答案 K[W].

海量资源：666java.com

# STEP 3: 实现动态规划

$$K[x] = \begin{cases} 0 & \text{if there are no i where } w_i \le x \\ \max_i \{ K[x-w_i] + v_i \} & \text{otherwise} \end{cases}$$

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize a size W+1 array, K
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

初始化基本情况(容量为 0 意味着价值也为 0)

按照背包容量大小从小到大进行迭代

穷举每个可以选择的物品 & 只考虑能能放入容量 x 的背包中的那些物品

**运行时间: O(nW)**

数组中每一项需要时间 O(n)

海量资源：666java.com

# 评估 O(nW)?

我们希望运行时间相对于输入规模的增长不要太快,
而输入规模与背包容量 W 有关.

输入规模是 O(nlogW) 的, 因为需要 logW 位来储存 W, 以及 nlogW 位来储存所有 n 个重量
(假设每件物品的价值不是主要影响因素).

所以, O(nW) 并不是输入大小的多项式形式. 这一类算法的复杂程度成为 “pseudo-polynomial”.

**多项式时间内解决背包问题是一个开放问题**
**背包问题的复杂程度为 NP-hard.**

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的值/代价, 但是问题本身需要求解最优的路径/子集/子数组/等等. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 4: 记录放入的物品

```
UNBOUNDED_KNAPSACK(W, n, weights, values):
    Initialize a size W+1 array, K
    K[0] = 0
    for x = 1,...,W:
        K[x] = 0
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
    return K[W]
```

**假设现在我们需要知道实际放入的物品.**

如何修改现有的伪代码来记录放入的物品?

考虑所有背包容量, 从 0 到 W, 我们储存当前容量背包所能储存的价值最大的物品集合.

海量资源：666java.com

# STEP 4: 记录放入的物品

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | | | | |
| ITEMS | | | | | |

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] U {item i}
   return ITEMS[W]
```



Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 0 | | | |
| ITEMS | | | | | |

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] U {item i}
   return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 0 | | | |
| ITEMS | | | | | |

K[1] = max{ K[1], K[0] + 1 }

**ITEMS[1] = ITEMS[0] +** 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```

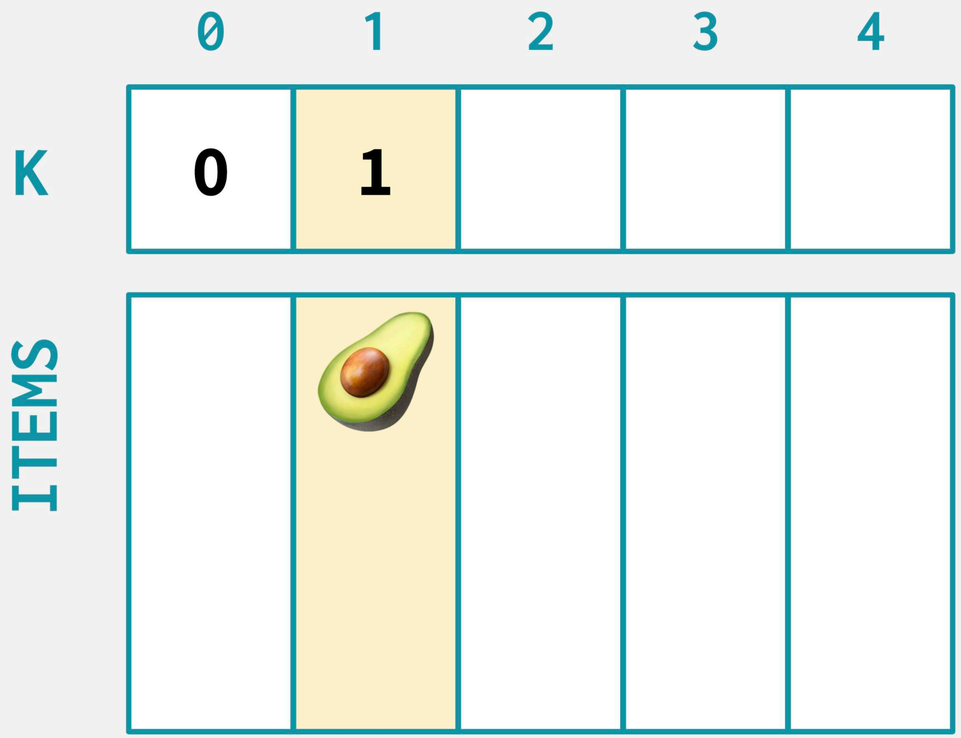


Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例




K[1] = max{ K[1], K[0] + 1 }

**ITEMS[1] = ITEMS[0] +** 🥑

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] U {item i}
   return ITEMS[W]
```

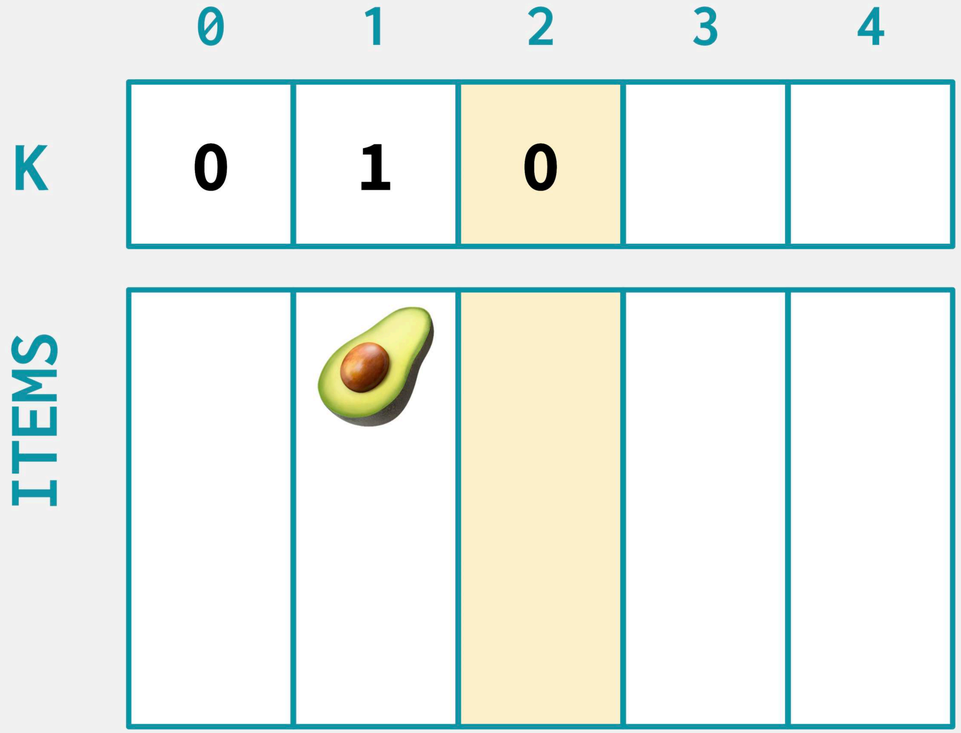


Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | **1** | **2** | **3** |
| Value: | **1** | **4** | **6** |

海量资源：666java.com

# 例




```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] U {item i}
   return ITEMS[W]
```



Capacity: **4**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 0 | | |
| ITEMS | |  | | | |

K[2] = max{ K[2], K[1] + 1 }

**ITEMS[2] = ITEMS[1] +** 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```

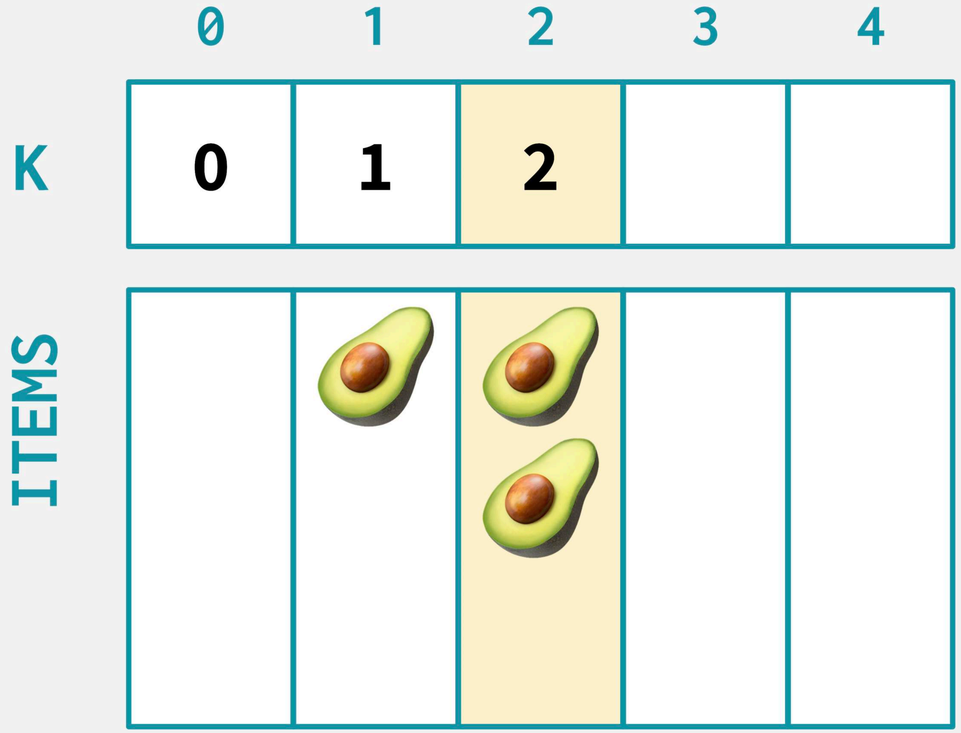


Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例




K[2] = max{ K[2], K[1] + 1 }

**ITEMS[2] = ITEMS[1] +** 🥑

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```

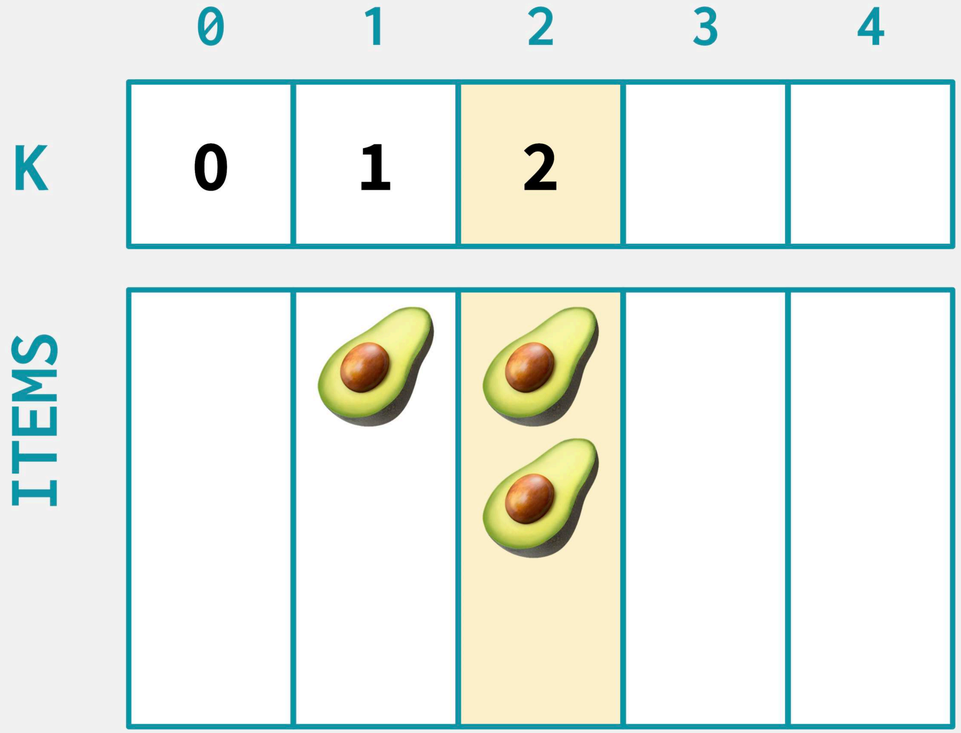


Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例




K[2] = max{ K[2], K[0] + 4 }

**ITEMS[2] = ITEMS[0] +** 🧲

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | **1** | **2** | **3** |
| Value: | **1** | **4** | **6** |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | | |
| ITEMS | |  |  | | |

K[2] = max{ K[2], K[0] + 4 }

**ITEMS[2] = ITEMS[0] +** 

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 0 | |
| ITEMS | |  |  | | |

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 0 | |
| ITEMS | |  |  | | |

K[3] = max{ K[3], K[2] + 1 }

**ITEMS[3] = ITEMS[2] +** 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 5 | |
| ITEMS | |  |  |   | |

K[3] = max{ K[3], K[2] + 1 }

**ITEMS[3] = ITEMS[2] +** 

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 5 | |
| ITEMS | | 🥑 | 🧲 | 🧲 🥑 | |

K[3] = max{ K[3], K[1] + 4 }

**ITEMS[3] = ITEMS[1] +** 🧲

*(并没有触发更新)*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | **1** | **2** | **3** |
| Value: | **1** | **4** | **6** |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 5 | |
| ITEMS | | 🥑 | 🧲 | 🧲 🥑 | |

K[3] = max{ K[3], K[0] + 6 }

**ITEMS[3] = ITEMS[0] +** 🐨

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | |
| ITEMS | |  |  |  | |

K[3] = max{ K[3], K[0] + 6 }

**ITEMS[3] = ITEMS[0] +** 

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 0 |
| ITEMS | |  |  |  | |

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 0 |
| ITEMS | |  |  |  | |

K[4] = max{ K[4], K[3] + 1 }

**ITEMS[4] = ITEMS[3] +** 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
   Initialize size W+1 arrays, K and ITEMS
   K[0] = 0, ITEMS[0] = { }
   for x = 1,...,W:
      K[x] = 0, ITEMS[x] = { }
      for i = 1,...,n:
         if w_i ≤ x:
            K[x] = max{ K[x], K[x-w_i] + v_i }
            if K[x] was updated:
               ITEMS[x] = ITEMS[x-w_i] U {item i}
   return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 7 |
| ITEMS | |  |  |  |  |

K[4] = max{ K[4], K[3] + 1 }

**ITEMS[4] = ITEMS[3] +** 

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 7 |
| ITEMS | |  |  |  |  |

K[4] = max{ K[4], K[2] + 4 }

**ITEMS[4] = ITEMS[2] +** 

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 8 |
| ITEMS | |  |  |  |  |

K[4] = max{ K[4], K[2] + 4 }

**ITEMS[4] = ITEMS[2] +** 

*更新!*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 8 |
| ITEMS | |  |  |  |  |

K[4] = max{ K[4], K[1] + 6 }

**ITEMS[4] = ITEMS[1] +** 

*(并没有触发更新)*

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| K | 0 | 1 | 4 | 6 | 8 |
| ITEMS | |  |  |  |   |

**结束!**

```
UNBOUNDED_KNAPSACK_ITEMS(W, n, weights, values):
    Initialize size W+1 arrays, K and ITEMS
    K[0] = 0, ITEMS[0] = { }
    for x = 1,...,W:
        K[x] = 0, ITEMS[x] = { }
        for i = 1,...,n:
            if w_i ≤ x:
                K[x] = max{ K[x], K[x-w_i] + v_i }
                if K[x] was updated:
                    ITEMS[x] = ITEMS[x-w_i] U {item i}
    return ITEMS[W]
```



Capacity: **4**

| Item: |  |  |  |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 0/1 背包

每件物品只能取一次

海量资源：666java.com

# 背包问题: 两个版本



Capacity: **10**

| Item: | 🥑 | 🧲 | 🍇 | 🐨 | 🚗 |
|---|---|---|---|---|---|
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

## 无界限背包

所有物品均有无限个.

如何使得包中物品总价值最大?

🧲 🧲 🐨 🐨

Total weight: **2 + 2 + 3 + 3 = 10**

Total value: **8 + 8 + 13 + 13 = 42**

## 0/1 背包

每个物品只有一份.

如何使得包中物品总价值最大?

🧲 🍇 🐨

Total weight: **2 + 4 + 3 = 9**

Total value: **8 + 14 + 13 = 35**

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 1: 最优子结构

**子问题 (ATTEMPT #1):**

小背包上的 0/1 背包

**方法不可行. 我们只能选择一件物品一次**

子问题需要知道哪些物品已经被选取过了.



海量资源：666java.com

# STEP 1: 最优子结构

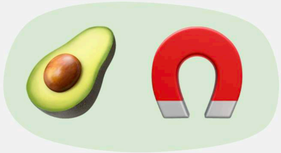
**子问题 (ATTEMPT #2):**

小背包上的 0/1 背包 & *物品的子集*

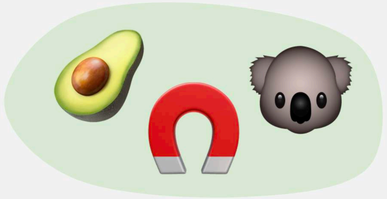
首先解决只有很少物品的子问题



添加物品



添加更多物品





解决小背包



更大的背包



更大的背包

**需要用到二维数组!**

海量资源：666java.com

# STEP 1: 最优子结构

**子问题 (ATTEMPT #2):**

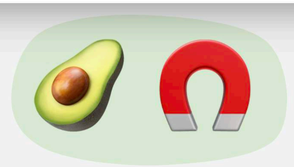
小背包上的 0/1 背包 & *物品的子集*

**子问题需要两个索引 x 与 j:**

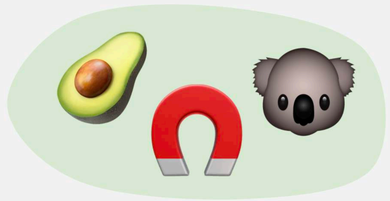
**K[x, j]** = 容量为 x 的背包、只考虑前 j 件物品的背包问题最优解

首先解
品

添加物品



添加更多物品



**需要用到二维数组!**

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 2: 递归求解公式

**K[x, j]** = 容量为 **x** 的背包、只考虑物品 1 至 **j** 能得到的最优解

递归公式为:

$$K[x,j] = \begin{cases} 0 & \text{if x or j are 0} \\ \max\{\ K[x, j-1],\ \ K[x-w_j, j-1] + v_j\ \} & \text{otherwise} \end{cases}$$

同样容量的背包不考虑物品 j

小背包且不考虑物品 j

放入物品 j 带来的价值

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的*值/代价*, 但是问题本身需要求解最优的*路径/子集/子数组/等等*. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# STEP 3: 实现动态规划

$$K[x,j] = \begin{cases} 0 & \text{if x or j are 0} \\ \max\{\, K[x,j-1],\ K[x-w_j,j-1] + v_j \,\} & \text{otherwise} \end{cases}$$

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
    Initialize a (n+1) x (W+1) table, K
    K[x,0] = 0 for all x = 0,...,W
    K[0,j] = 0 for all j = 0,...,n
    for x = 1,...,W:
        for j = 1,...,n:
            K[x,j] = K[x,j-1]
            if w_j ≤ x:
                K[x,j] = max{ K[x,j], K[x-w_j, j-1] + v_j }
    return K[W,n]
```

初始化基本情况(背包容量为 0 或者 0 件可选物品)

遍历背包容量

遍历可以考虑的物品

默认情况: 不考虑物品 j

如果物品 j 的重量可以放入背包, 考虑放入或者不放入

**运行时间: O(nW)** 填写 nW 项每一项用时 O(1)

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | | | |
| j=2 | 0 | | | |
| j=3 | 0 | | | |

首先初始化“基本情况”

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 0 | | |
| j=2 | 0 | | | |
| j=3 | 0 | | | |

默认值 = **K[x, j–1]**

如果能选择 🥑，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | | | |
| j=3 | 0 | | | |

默认值 = **K[x, j–1]**

如果能选择 🥑，是否能改善结果?

**0 + 1** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | | | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | | | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

背包中放不下

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 🐨，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

背包中放不下

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 0 | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 🥑，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

**0 + 1** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 1 | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

**0 + 4** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | **1** | **2** | **3** |
| Value: | **1** | **4** | **6** |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 🐨，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

背包中放不下

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 0 |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

**0 + 1** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 1 |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | |

默认值 = **K[x, j–1]**

如果能选择 🧲，是否能改善结果?

**1 + 4** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | 5 |

默认值 = **K[x, j–1]**

如果能选择 🐨，是否能改善结果?

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | 🥑 | 🧲 | 🐨 |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | 6 |

默认值 = **K[x, j–1]**

如果能选择 , 是否能改善结果?

**0 + 6** 更好!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 例

| | x=0 | x=1 | x=2 | x=3 |
|---|---|---|---|---|
| j=0 | 0 | 0 | 0 | 0 |
| j=1 | 0 | 1 | 1 | 1 |
| j=2 | 0 | 1 | 4 | 5 |
| j=3 | 0 | 1 | 4 | 6 |

计算结束! 最优解是在包中放入一只 此时有最大价值 6!

```
ZERO_ONE_KNAPSACK(W, n, weights, values):
   Initialize a (n+1) x (W+1) table, K
   K[x,0] = 0 for all x = 0,...,W
   K[0,j] = 0 for all j = 0,...,n
   for x = 1,...,W:
      for j = 1,...,n:
         K[x,j] = K[x,j-1]
         if w_j ≤ x:
            K[x,j] = max{ K[x,j], K[x-w_j,j-1] + v_j }
   return K[W,n]
```



Capacity: **3**

| Item: | | | |
|---|---|---|---|
| Weight: | 1 | 2 | 3 |
| Value: | 1 | 4 | 6 |

海量资源：666java.com

# 使用 DP 的一般步骤

1. **确定最优子结构.** 重叠的子问题是哪些?
2. **定义递归求解公式.** 通过子问题的解, 递归地定义最优解.
3. **实现动态规划.** 将递归公式转变为DP算法.
4. **必要时, 储存额外信息.** 可能需要解决一些相关问题, 例如第3步可以得到一个最优的值/代价, 但是问题本身需要求解最优的路径/子集/子数组/等等. 修改第3步重的算法来收集这些信息.

海量资源：666java.com

# 使用 DP 的一般步骤

**思考一下如何修改 ZERO_ONE_KNAPSACK 来得到最优解时包中放入的物品.**

参考刚才填的表

海量资源：666java.com

# 背包问题: 两个版本



Capacity: **10**

| Item: | 🥑 | 🧲 | 🍇 | 🐨 | 🚗 |
|---|---|---|---|---|---|
| Weight: | 6 | 2 | 4 | 3 | 11 |
| Value: | 20 | 8 | 14 | 13 | 35 |

## 无界限背包

所有物品均有无限个.

如何使得包中物品总价值最大?

**子问题**: 计算小背包上的最优解

(使用 1 维数组缓存结果)

## 0/1 背包

每个物品只有一份.

如何使得包中物品总价值最大?

**子问题**: 小背包上的最优解 & 考虑物品的子集

(使用 2 维数组缓存结果)

海量资源：666java.com

# 总结

- 介绍了三个经典的动态规划问题:
  - **最长公共子序列 Longest Common Subsequence**
  - **无界限背包问题 Unbounded Knapsack**
  - **0/1 背包问题 0/1 Knapsack**
- 前 3 步在解决动态规划问题中是必需的
  - 有时需要第 4 步 (第 5 步在实际应用中非常重要)

海量资源：666java.com

# 接下来

- 贪心算法