作业：动态规划

- 一个有用的函数：collections.namedtuple

```
import collections
Case = collections.namedtuple('Case', ['value', 'items'])
my_case = Case(5, [1, 3, 6])
print(my_case.value)
print(my_case.items)
```

- 用于保存动态规划的中间状态
- 也可以用普通的元组或其它结构保存

完全背包

- 背包大小为*w*时总价值最高为*m*[*w*]

$$m[0] = 0$$

$$m[w] = \max_{w_i \le w}(v_i + m[w - w_i])$$

背包大小	0	1	2	…	w	…
最大价值	0	…	…	…	?	…

```python
def knapsack_unbound(treasure, maxw):
    Case = collections.namedtuple('Case', ['value', 'items'])
    res = [Case(0, [])]
    for Y in range(1, maxw + 1):
        cur_max, cur_items = 0, []
        for t in treasure:
            if t['w'] <= Y:
                new_max = t['v'] + res[Y - t['w']].value
                if new_max > cur_max:
                    cur_items = res[Y - t['w']].items + [t]
                    cur_max = new_max
        res.append(Case(cur_max, cur_items))
    return res[maxw]
```

0/1背包

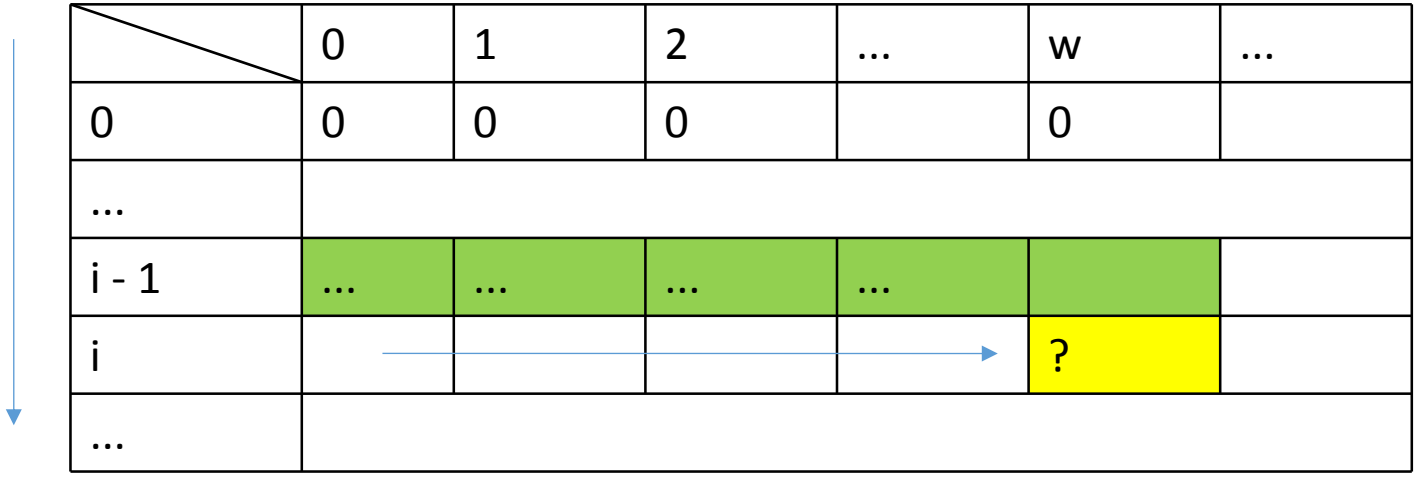
- 一种思路：背包大小为w，最多装i个物品时，最大价值为$m[i, w]$
  - 记录哪些物品已经使用过？（标记、使用set）
  - 复杂度？

0/1背包

- 背包大小为*w*，使用前i个物品时，最大价值为*m*[*i*, *w*]

$$m[i,\ w] = m[i-1,\ w] \qquad w_i > w$$

$$m[i,\ w] = \max(m[i-1,\ w],\ m[i-1, w-w_i] + v_i) \qquad w_i \leqslant w$$



- 可以只保留上一行的结果节省空间

```
def knapsack_01(treasure, maxw):
    Case = collections.namedtuple('Case', ['value', 'items'])
    res = [Case(0, []) for _ in range(maxw + 1)]
    for t in treasure:
        new_res = res[:]
        for Y in range(1, maxw + 1):
            if t['w'] <= Y:
                pre_case = res[Y - t['w']]
                new_max = t['v'] + pre_case.value
                if new_max >= new_res[Y].value:
                    new_res[Y] = Case(new_max, pre_case.items + [t])
        res = new_res
    return res[maxw]
```

最短编辑距离

- D(i, j)：分别取前i个字符和前j个字符的最短编辑距离

$$D(0, 0) = 0$$
$$D(i, 0) = insertCost * i$$
$$D(0, j) = deleteCost * j$$

i, 目标串字符位置

j, 原始串字符位置

$$D(i, j) = \min \begin{cases} D(i-1, j) + insertCost(target_i) \\ D(i-1, j-1) + substituteCost(source_j, target_i) \\ D(i, j-1) + deleteCost(source_j) \end{cases}$$

$$substituteCost \begin{cases} = 0 & \text{if } target[i] = source[j] \\ = 2 & \text{otherwise} \end{cases}$$

$$insertCost = 1$$
$$deleteCost = 1$$

	0	1	…	i - 1	i	…
0	0					
1						
…						
j - 1						
j					?	
…						

```python
def levenshtein(src, dst, op = {'copy': 5, 'ins': 20, 'del': 20}):
    Case = collections.namedtuple('Case', ['dis', 'op'])
    D = [[Case(0, [])] * (len(dst) + 1) for _ in range(len(src) + 1)]
    D[0][0] = Case(0, [])
    for i in range(1, len(src) + 1):
        D[i][0] = Case(op['del'] * i, D[i - 1][0].op + ['del ' + src[i - 1]])
    for j in range(1, len(dst) + 1):
        D[0][j] = Case(op['ins'] * j, D[0][j - 1].op + ['ins ' + dst[j - 1]])

    for i in range(1, len(src) + 1):
        for j in range(1, len(dst) + 1):
            del_from, ins_from, copy_from = D[i - 1][j], D[i][j - 1], D[i - 1][j - 1]
            del_op = 'del ' + src[i - 1]
            ins_op = 'ins ' + dst[j - 1]
            copy_op = 'copy ' + src[i - 1]

            del_case = Case(del_from.dis + op['del'],
                            del_from.op + [del_op])
            ins_case = Case(ins_from.dis + op['ins'],
                            ins_from.op + [ins_op])
            if src[i - 1] == dst[j - 1]:
                copy_case = Case(copy_from.dis + op['copy'],
                                 copy_from.op + [copy_op])
            else:
                copy_case = Case(copy_from.dis + op['del'] + op['ins'],
                                 copy_from.op + [del_op, ins_op])
            new_case = del_case if del_case.dis < ins_case.dis else ins_case
            if copy_case.dis < new_case.dis:
                new_case = copy_case
            D[i][j] = new_case
    return D[len(src)][len(dst)]
```