物品 ID /剩余容量	0	1	2	3	4	5
1	0	3	3	3	3	3
2	0	3	7	10	10	10
3	0	3	7	12	15	19

> 直接来分析能放三种物品的情况，也就是最后一行

- 当容量少于 3 时， 只取上一行对应的数据， 因为当前容量不能容纳物品 3
- 当容量 为 3 时，考虑两种情况，分别为放入物品 3 和不放物品 3
  - 不放物品 3 的情况下， 总价值为 10
  - 放入物品 3 的情况下， 总价值为 12，所以应该放入物品 3
- 当容量 为 4 时，考虑两种情况，分别为放入物品 3 和不放物品 3
  - 不放物品 3 的情况下， 总价值为 10
  - 放入物品 3 的情况下， 和放入物品 1 的价值相加，得出总价值为 15，所以应该放入物品 3
- 当容量 为 5 时，考虑两种情况，分别为放入物品 3 和不放物品 3
  - 不放物品 3 的情况下， 总价值为 10
  - 放入物品 3的情况下， 和放入物品 2 的价值相加，得出总价值为 19，所以应该放入物品 3

以下代码对照上表更容易理解

```js
/**
 * @param {*} w 物品重量
 * @param {*} v 物品价值
 * @param {*} C 总容量
 * @returns
 */
function knapsack(w, v, C) {
  let length = w.length
  if (length === 0) return 0

  // 对照表格，生成的二维数组，第一维代表物品，第二维代表背包剩余容量
  // 第二维中的元素代表背包物品总价值
  let array = new Array(length).fill(new Array(C + 1).fill(null))

  // 完成底部子问题的解
  for (let i = 0; i <= C; i++) {
    // 对照表格第一行，  array[0] 代表物品 1
    // i 代表剩余总容量
```


-
-
  -
  -
-
  -
  -
-
  -
  -

```
    // 当剩余总容量大于物品 1 的重量时，记录下背包物品总价值，否则价值为 0
    array[0] [i] = i >= w[0] ? v[0] : 0
  }

  // 自底向上开始解决子问题，从物品 2 开始
  for (let i = 1; i < length; i++) {
    for (let j = 0; j <= C; j++) {
      // 这里求解子问题，分别为不放当前物品和放当前物品
      // 先求不放当前物品的背包总价值， 这里的值也就是对应表格中上一行对应的值
      array[i] [j] = array[i - 1] [j]
      // 判断当前剩余容量是否可以放入当前物品
      if (j >= w[i]) {
        // 可以放入的话，就比大小
        // 放入当前物品和不放入当前物品， 哪个背包总价值大
        array[i] [j] = Math.max(array[i] [j], v[i] + array[i - 1] [j - w[i]])
      }
    }
  }
  return array[length - 1] [C]
}
```

最长递增子序列

> 最长递增子序列意思是在一组数字中，找出最长一串递增的数字， 比如

```
0, 3, 4, 17, 2, 8, 6, 10
```

> 对于以上这串数字来说， 最长递增子序列就是 `0, 3, 4, 8, 10` ， 可以通过以下表格更清晰的理解

数字	0	3	4	17	2	8	6	10
长度	1	2	3	4	2	4 4	5	

通过以上表格可以很清晰的发现一个规律，找出刚好比当前数字小的数， 并且在小的数组成的长度基础上加一 。

这个问题的动态思路解法很简单， 直接上代码