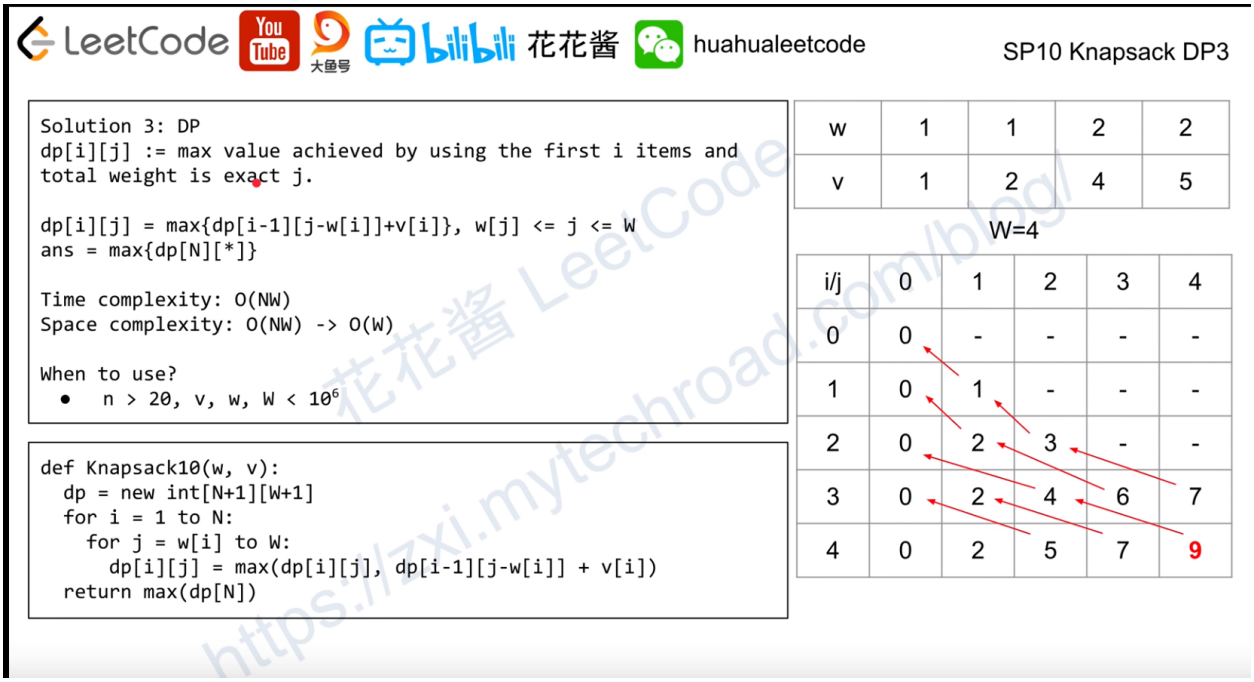
Knapsack Problem



Java 代码如下：

```
for (int i = 1; i <= N; ++i) {
    for (int j = 0; j <= V; ++j) {
        backpack[i][j] = backpack[i - 1][j];
        if (j >= cap[i]) {
            backpack[i][j] = Math.max(backpack[i][j], backpack[i - 1][j - cap[i]] + val[i]);
        }
    }
}
```

LeetCode 花花酱 huahualeetcode

SP11 Knapsack 2

0-1 Knapsack Problem

Problem definition: Given N items, w[i] is the weight of the i-th item and v[i] is value of the i-th item. Given a knapsack with capacity W. Maximize the total value. Each item can be used **0 or 1 time**.

dp[i][j] := **max value** of using **first i items** and **total weight is exact j**.

```
dp[i][j] = max(dp[i-1][j],           # i-th item not used
               dp[i-1][j-w[i]]+v[i]) # i-th item used


Ans = max(dp[N][*])
Time complexity: O(NW)
Space complexity: O(NW) -> O(W)

Space complexity reduction:
Iterate j in reverse order to prevent using i-th item
multiple times.
```

```
for i = 1 to N:
  for j = 0 to W:
    dp[i][j] = max(dp[i-1][j],
                   dp[i-1][j-w[i]]+v[i])
return max(dp[N])
```

```
def knapsack01(w, v):
  for j = W to w:
    dp[j] = max(dp[j], dp[j-w] + v)

for i = 1 to N:
  knapsack01(w[i], v[i])
return man(dp[N])
```


优化：

LeetCode 花花酱 huahualeetcode

SP10 Knapsack DP3

Reduce space complexity

Approach 1: Use a tmp array

```
def Knapsack10(w, v):
  dp = new int[W+1]
  for i = 1 to N:
    tmp = new int[W+1]
    for j = w[i] to W:
      tmp[j] = max(tmp[j],
                   dp[j-w[i]]+v[i])
    dp = tmp
  return max(dp)
```

Approach 2: iterator j in reverse order

```
Push:
def Knapsack10(w, v):
  dp = new int[W+1]
  for i = 1 to N:
    for j = W-w[i] to 0:
      dp[j+w[i]] = max(dp[j+w[i]],
                       dp[j]+v[i])
  return max(dp)
```

```
Pull:
def Knapsack10(w, v):
  dp = new int[W+1]
  for i = 1 to N:
    for j = W to w[i]:
      dp[j] = max(dp[j],
                  dp[j-w[i]]+v[i])
  return max(dp)
```

w	1	1	2	2
v	1	2	4	5

花花酱 LeetCode Knapsack - 刷... Watch later Share

i/j	0	1	2	3	4
4	0	2	0+	7	9

1:13 / 1:17 YouTube

1.滚动数组优化

2.一维数组优化：把第二层循环倒序即可。

滚动数组优化

目前我们得到了一个空间复杂度为 $O(NV)$ 的算法来解决 0-1 背包问题。

思考 1 分钟，空间复杂度上我们是否可以继续优化？

答案：能！

观察到在计算 $backpack[i][j]$ 时，我们只用到了 $backpack[i-1][j]$ 和 $backpack[i-1][j-cap[i]]$ 这两个数，并没有用到 $backpack[i-1][], backpack[i-2][], ...$ 的信息，也就是 $backpack[i][]$ 只和 $backpack[i-1][]$ 有关，那么我们只需要两个数组就够了。

根据以上思路可以得到 Core Code。

Java 代码如下：

```
for (int i = 1; i <= N; ++i) {
    for (int j = 0; j <= V; ++j) {
        int index = i & 1;
        backpack[index][j] = backpack[index ^ 1][j];
        if (j >= cap[i]) {
            backpack[index][j] = Math.max(backpack[index][j], backpack[index ^ 1][j - cap[i]] + val[i]);
        }
    }
}
```

一维数组优化

虽然空间复杂度从 $O(NV)$ 降到了 $O(V)$，但是我们仍然需要 `backpack[2][V]` 这样的二维数组。有没有只通过一维数组就可以解决 0-1 背包问题呢？

答案：有！

事实上我们只需要把第二层循环倒序即可。

Java 代码如下：

```
for (int i = 1; i <= N; ++i) {
    for (int j = V; j >= cap[i]; --j) {
        f[j] = Math.max(f[j], f[j - cap[i]] + val[i]);
    }
}
```

怎么理解呢？

在第 i 层循环初 $f[j]$ 存的相当于 $backpack[i-1][j]$ 的值。

在更新 $f[j]$ 时，我们用到了 $f[j-cap[i]]$，由于第二层循环倒序，所以 $f[j-cap[i]]$ 未被更新，此时它代表 $backpack[i-1][j-cap[i]]$。所以 `f[j] = Math.max(f[j], f[j - cap[i]] + val[i])` 等价于 `backpack[i][j] = Math.max(backpack[i - 1][j], backpack[i - 1][j - cap[i]] + val[i])`。

在第 i 层循环末 $f[j]$ 存的相当于 $backpack[i][j]$ 的值。