研究生算法课课堂笔记

上课日期：2016.10.27　　第(1)节课

组长学号及姓名：1601214493　曾鑫璐

组员学号及姓名：1601214505　陈佳棋

背包问题

给定一个重量限制为 W 的背包，以及 n 个物品，每个物品有重量 w_i 和价值 v_i，w_i 是正整数。将物品装入背包，求在不超过背包重量限制的情况下，所能取得的最大价值。根据物品能取一次、无限次、有限 k_i 次可分为 01 背包问题、完全背包问题、多重背包问题。

一、01 背包问题

例如 n=5，W=11，每个物品的重量 wi 和价值 vi 如下表：

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

方案(1)装物品{1, 2, 5} 价值 35 重量 10

方案(2)装物品{3, 4} 价值 40 重量 11

方案(3)装物品{3, 5} 价值 46 但重量超 1 个单位，不是合法方案

贪心法对于背包问题不适用

策略 1 每次选择价值最大的物品

反例：三个物品，价值分别为 7、5、5 ，重量分别为 51%，50%，50%。按照价值最大，首先选择价值为 7，重量为 51%的物品 1，剩余重量为 49%，不能放下其他物品，取得的价值为 7。而最优方案为选择价值为 5，重量为 50%的物品 2、3，背包恰好装满，取得的价值为 10。

策略 2 每次选择重量最小的物品

反例：三个物品，价值分别为 1、5、5，重量分别为 49%，50%，50%。按照重量最小，首先选择价值为 1，重量为 49%的物品 1，剩余重量为 51%，还能放下价值为 5，重量为 50%的物品 2 或 3，取得的价值为 1+5=6。而最优方案为选择价值为 5，重量为 50%的物品 2、3，背包恰好装满，取得的价值为 10。

策略 3 每次选择性价比最高的物品

反例同策略 1，物品 1 的性价比 7/51%最高。

但如果是 Gold Rust 模型，即每个物品可以无限细分，此时使用性价比策略从高到低选择是最优的。

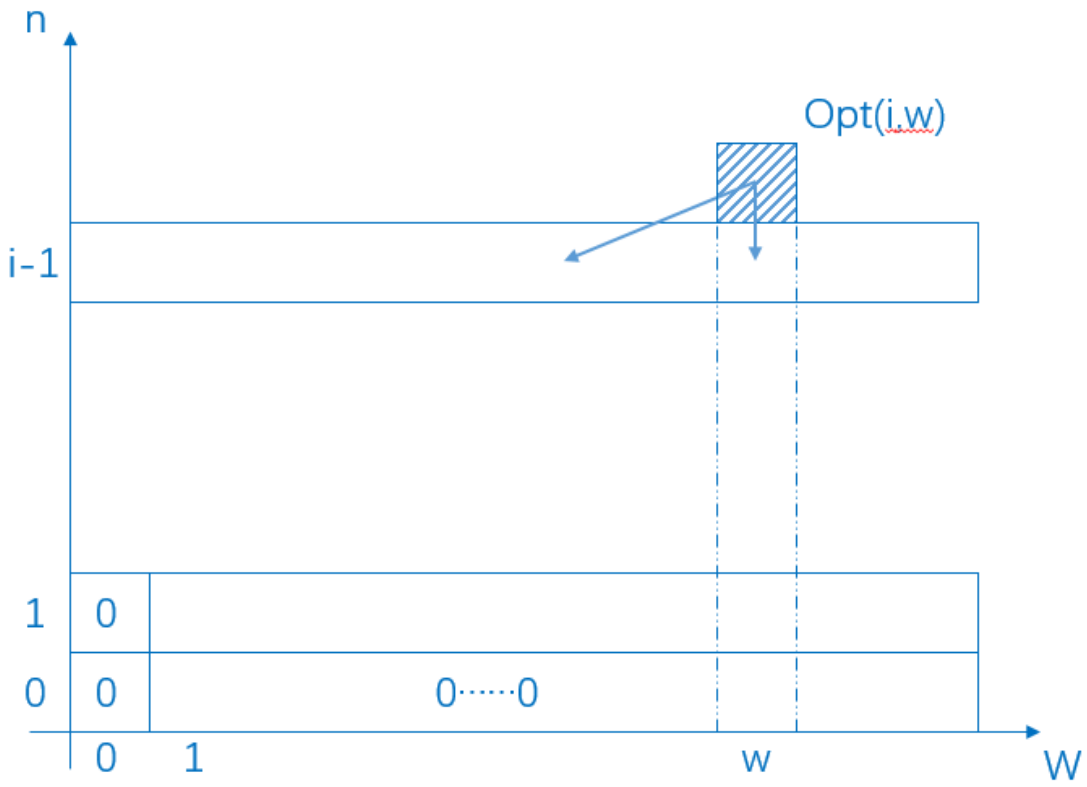
可以使用二维动态规划解决 01 背包问题，

状态设计：Opt[i, w]表示前 i 个物品装进重量限制为 w 的背包内能获得的最大价值。

状态转移：考虑第 i 件物品是否放入背包。

伪代码如下：

```
For w = 0 To W
    Opt[0, w] ← 0
For i = 1 To n
    For w = 0 To W
        If (wᵢ > w)     Opt[i, w] ← Opt[i-1, w]
        Else            Opt[i, w] ← max( Opt[i-1, w], vᵢ + Opt[i-1, w-wᵢ] )
Return Opt[n, W]
```



二维动态规划

图 1　01 背包二维动态规划状态依赖

动态规划的状态依赖如图 1，初始值对任意 w 值 Opt[0, w]为 0，计算到 Opt[i, w]时，依赖 Opt[i-1, w]和 Opt[i-1, w-w_i]，只依赖于第 i-1 行的状态。

时间复杂度

动态规划求解 01 背包的时间复杂度为 Θ(nW)，但这是一个伪多项式的算法，理由是表示 W 只需要 logW 位，Θ(nW)里的 W 相对于表示 W 的 logW 位是指数。背包问题是 NP-complete 问题。

在题设中假定背包重量限制 W 和物品重量 w_i 都是正整数，使得重量维的状态共有 0…W 这(W+1)个状态，因此可以使用动态规划算法求解。若物品重量 w_i 可以是小数，重量维的状态将增加到 2^n 个。

反向追踪

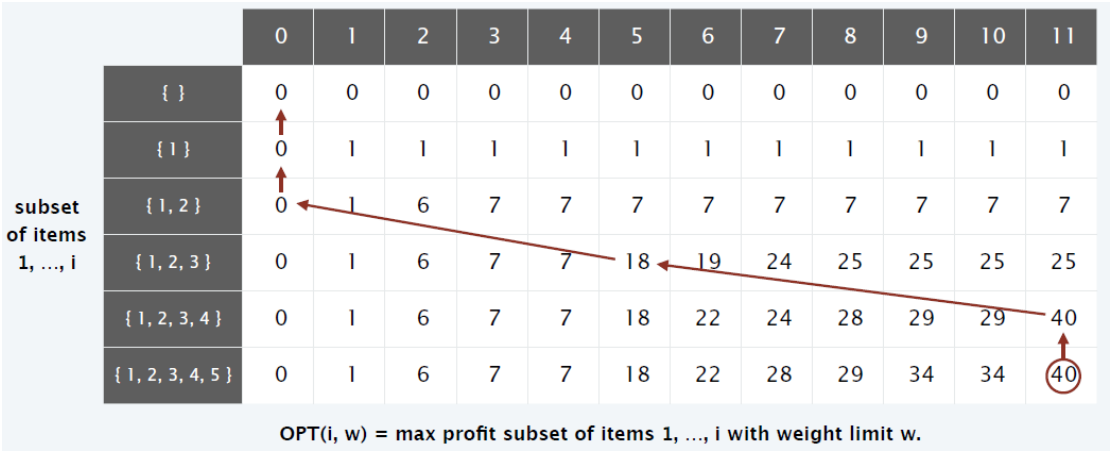


图 2　反向追踪过程

二维动态规划反向追踪输出最优解包含的物品方案不需要辅助的数据结构，初始设置方案集合为空，i 设为 n，w 设为 W。每次比较 Opt[i, w]和 Opt[i-1, w]，有两种情况：

(1)Opt[i, w] = Opt[i-1, w]，说明没有选物品 i，将 i 减少 1；

(2)Opt[i, w] > Opt[i-1, w]，说明选了物品 i，将物品 i 加入方案集合，将 i 减少 1，将 w 减少 w_i。

空间优化

考虑到 Opt[i, w]依赖 Opt[i-1, w]和 Opt[i-1, w-w_i]，即第 i 行的状态依赖于第 i-1 行的状态，可以只保存两行的状态，使用滚动数组求解。在这种实现下，没有保留足够的信息，不能实现反向追踪。

也可以只存一行状态，将 w 的循环反向从 W 到 wi。空间复杂度降为 Θ(W)，时间复杂度仍为 Θ(nW)。

```
For i = 0 To W
    Opt[w] ← 0
For i = 1 To n
    For w = W To wi
        Opt[w] ← max( Opt[w], vi + Opt[w - wi] )
Return Opt[W]
```

二、完全背包问题

不同于 01 背包，一个物品只能取一次；完全背包问题下，一个物品可以取任意次。

```
For w = 0 To W
    Opt[0, w] ← 0
For i = 1 To n
    For w = 0 To W
        Opt[i, w] ← Opt[i-1, w]
        If (wᵢ ≤ w)    Opt[i, w] ← max( Opt[i, w], vᵢ + Opt[i, w-wᵢ] )
Return Opt[n, W]
```

二维动态规划

时间复杂度降为 $\Theta(nW)$，空间复杂度仍为 $\Theta(nW)$。

```
For i = 0 To W
    Opt[w] ← 0
For i = 1 To n
    For w = wᵢ To W
        Opt[w] ← max( Opt[w], vᵢ+ Opt[w - wᵢ] )
Return Opt[W]
```

空间优化

空间复杂度降为 $\Theta(W)$，时间复杂度仍为 $\Theta(nW)$。

图 3　一维空间完全背包状态依赖图

Opt[w]依赖 Opt[w-w_i]，w 循环只能从小到大。

三、恰好装满的背包问题

Piggy-bank

现有一个装满硬币的小猪储钱罐，已知空的储钱罐重量，满的储钱罐重量，每种硬币的重量和价值，为避免储钱罐被砸碎后硬币价值太少，求储钱罐里最少有多少钱？

和完全背包问题的区别：

(1)背包要求不超过限重；储钱罐要求恰好装满

(2)背包要求的最优解是最大值；储钱罐要求的最优解是最小值，如果不要求恰好，最优解价值是 0 即什么都不装。

先看另一个较简单的恰好装满的问题，

```
Opt[0] ← 0
For i = 1 To W
    Opt[w] ← -∞
For i = 1 To n
    For w = W To wi
        If (Opt[w - wi] ≠ -∞)   Opt[w] ← max( Opt[w], vi+ Opt[w - wi] )
Return Opt[W]
```

恰好装满的最大 01 背包问题

时间复杂度为Θ(nW)，空间复杂度为Θ(W)。

恰好装满的完全背包问题修改 w 循环为 w_i 到 W；

Piggy-bank 该题修改初值为正无穷，转移用 min 函数替换 max 函数。

关于恰好装满的背包的一些结论

(1)即使不看负无穷的项，Opt 值也不是单调递增。

例：重量为 3、4、8 的三个物品，价值依次是 2、2、1

Opt[7] = 4

Opt[8] = 1

Opt[7] > Opt[8]

(2)如不要求恰好装满，是单调递增。

(3)已经求了非恰好装满的背包，要求恰好装满的背包，没有办法。

(4)已经求了恰好装满的背包，要求非恰好装满的背包，可以扫描找最大值。

四、多重背包问题

不同于完全背包，每个物品能取任意多件；在多重背包问题下，物品 i 至多能取 k_i 件。类似于完全背包的三维动态规划，直接修改 01 背包的动态规划算法，增加一维枚举每种物品放置个数。

```
For w = 0 To W
    Opt[0, w] ← 0
For i = 1 To n
    For w = 0 To W
        For j = 1 To k_i
            If (w_i*j > w)  Opt[i, w] ← Opt[i-1, w]
            Else            Opt[i, w] ← max( Opt[i-1, w], v_i*j + Opt[i-1, w-w_i*j] )
Return Opt[n, W]
```

三维动态规划

时间复杂度为 $\Theta(nW^2)$，空间复杂度为 $\Theta(nW)$。

二进制改进

例如物品 1 至多可取 13 件，将物品 1 拆分为 4 个物品，重量和价值分别为 (w_i, v_i)，$(2w_i, 2v_i)$，$(4w_i, 4v_i)$，$(6w_i, 6v_i)$，使用 01 背包算法求解，时间复杂度为 $\Theta(nWlogW)$，空间复杂度为 $\Theta(W)$。

另有一个时间为 $\Theta(nW)$ 的算法，需要使用单调队列优化，这里不展开。