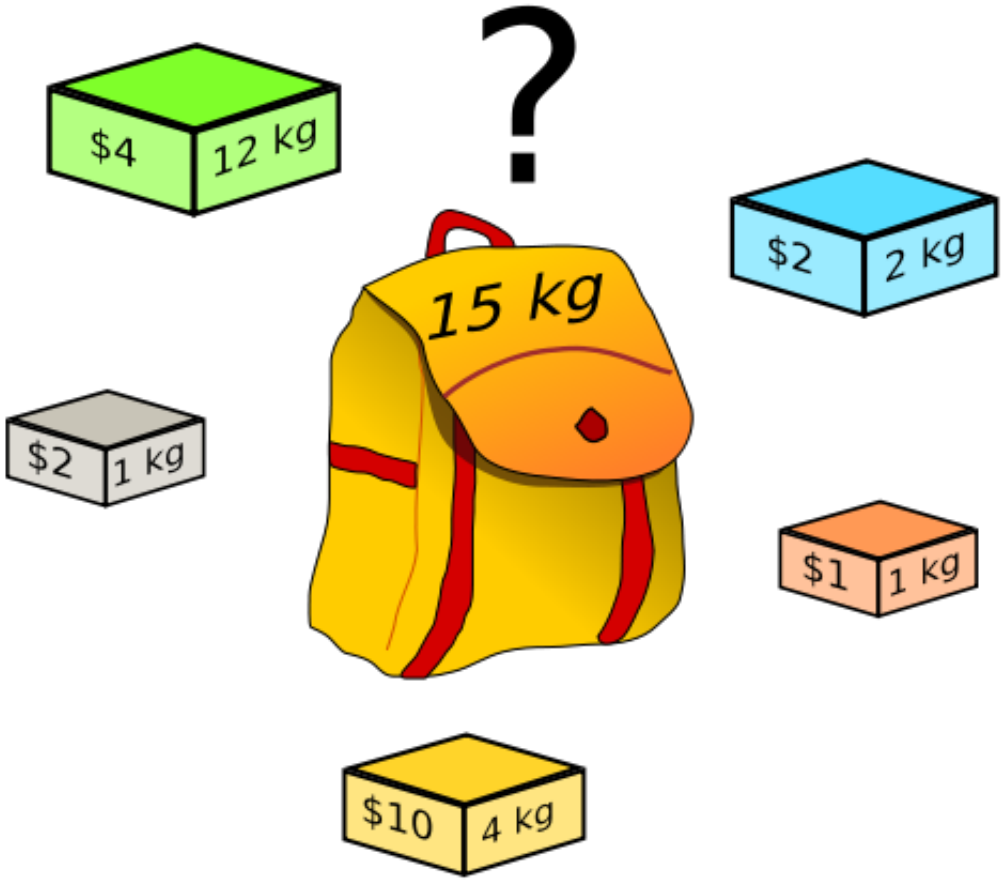
数据结构与算法题解（10）：0-1背包问题与部分背包问题

假设我们有n件物品，分别编号为1, 2...n。其中编号为i的物品价值为v_i，它的重量为w_i。为了简化问题，假定价值和重量都是整数值。现在，假设我们有一个背包，它能够承载的重量是W。现在，我们希望往包里装这些物品，使得包里装的物品价值最大化，那么我们该如何来选择装的东西呢？问题结构如下图所示：



这个问题其实根据不同的情况可以归结为不同的解决方法。假定我们这里选取的物品每个都是独立的，不能选取部分。也就是说我们要么选取某个物品，要么不能选取，不能只选取一个物品的一部分。这种情况，我们称之为0-1背包问题。而如果我们可以使用部分的物品的话，这个问题则成为部分背包(fractional knapsack)问题。下面我们针对每种情况具体分析一下。

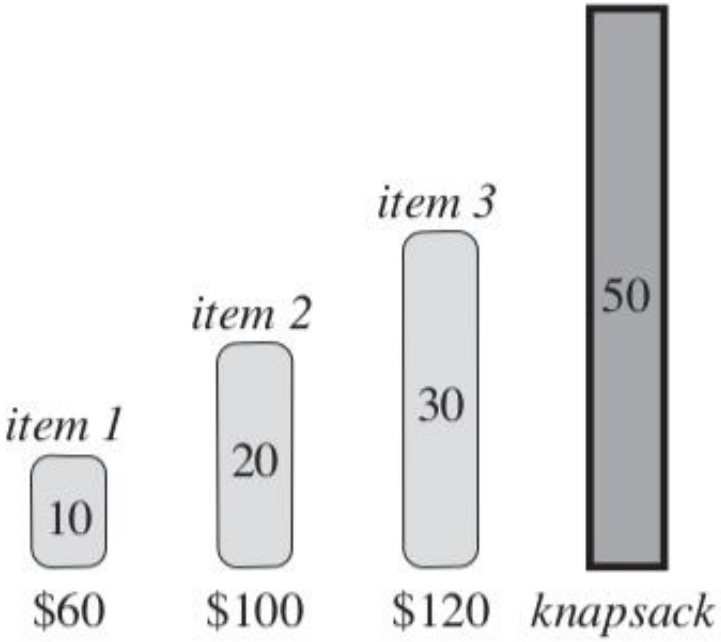
一、0-1背包

1.1 初步分析

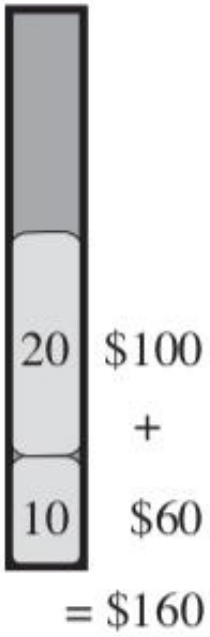
对于这个问题，一开始确实有点不太好入手。一堆的物品，每一个都有一定的质量和价值，我们

能够装入的总重量有限制，该怎么来装使得价值最大呢？对于这n个物品，每个物品我们可能会选，也可能不选，那么我们总共就可能有2^n种组合选择方式。如果我们采用这种办法来硬算的话，则整体的时间复杂度就达到指数级别的，肯定不可行。

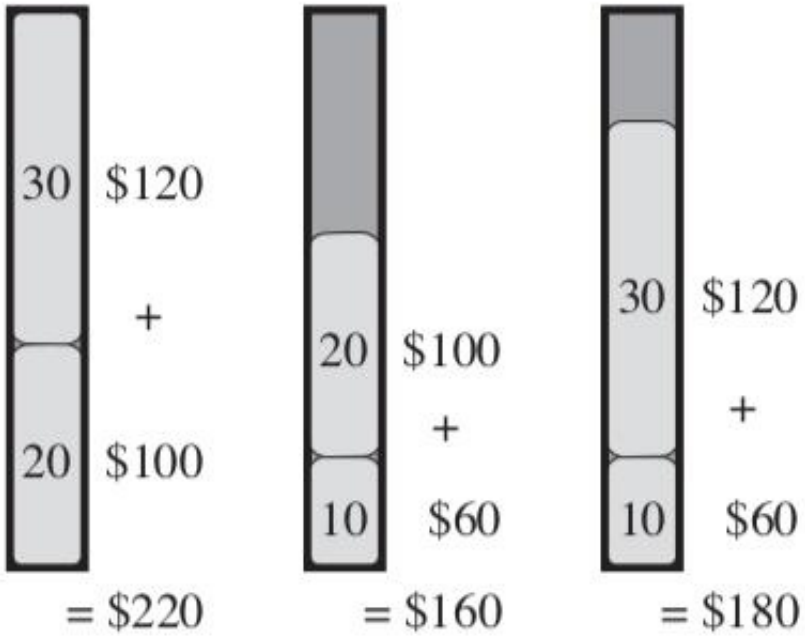
现在我们换一种思路。既然每一种物品都有价格和重量，我们优先挑选那些单位价格最高的是否可行呢？比如在下图中，我们有3种物品，他们的重量和价格分别是10, 20, 30 kg和60, 100, 120。



那么按照单位价格来算的话，我们最先应该挑选的是价格为60的元素，选择它之后，背包还剩下50 - 10 = 40kg。再继续前面的选择，我们应该挑选价格为100的元素，这样背包里的总价值为60 + 100 = 160。所占用的重量为30, 剩下20kg。因为后面需要挑选的物品为30kg已经超出背包的容量了。我们按照这种思路能选择到的最多就是前面两个物品。如下图：



按照我们前面的期望，这样选择得到的价值应该是最大的。可是由于有一个背包重量的限制，这里只用了30kg，还有剩下20kg浪费了。这会是最优的选择吗？我们看看所有的选择情况：



很遗憾，在这几种选择情况中，我们前面的选择反而是带来价值最低的。而选择重量分别为20kg和30kg的物品带来了最大的价值。看来，我们刚才这种选择最佳单位价格的方式也行不通。

1.2 动态规划

既然前面两种办法都不可行，我们再来看看有没有别的方法。我们再来看这个问题。我们需要选择n个元素中的若干个来形成最优解，假定为k个。那么对于这k个元素a1, a2, ...ak来说，它们组成的物品组合必然满足总重量<=背包重量限制，而且它们的价值必然是最大的。因为它们是我们假定的最优选择嘛，肯定价值应该是最大的。假定a_k是我们按照前面顺序放入的最后一个物品。它的重量为w_k，它的价值为v_k。既然我们前面选择的这k个元素构成了最优选择，如果我们把这个a_k物品拿走，对应于$k-1$个物品来说，它们所涵盖的重量范围为$0-(W-w_k)$。假定W为背包允许承重的量。假定最终的价值是V，剩下的物品所构成的价值为$V-v_k$。这剩下的$k-1$个元素是不是构成了一个这种$W-w_k$的最优解呢?

我们可以用反证法来推导。假定拿走a_k这个物品后，剩下的这些物品没有构成$W-w_k$重量范围的最佳价值选择。那么我们肯定有另外$k-1$个元素，他们在$W-w_k$重量范围内构成的价值更大。如果这样的话，我们用这$k-1$个物品再加上第k个，他们构成的最终W重量范围内的价值就是最优的。这岂不是和我们前面假设的k个元素构成最佳矛盾了吗？所以我们可以肯定，在这k个元素里拿掉最后那个元素，前面剩下的元素依然构成一个最佳解。

现在我们经过前面的推理已经得到了一个基本的递推关系，就是一个最优解的子解集也是最优的。可是，我们该怎么来求得这个最优解呢？我们这样来看。假定我们定义一个函数$c[i, w]$表示到第i个元素为止，在限制总重量为w的情况下我们所能选择到的最优解。那么这个最优解要么包

含有i这个物品，要么不包含，肯定是这两种情况中的一种。如果我们选择了第i个物品，那么实际上这个最优解是c[i - 1, w-wi] + vi。而如果我们没有选择第i个物品，这个最优解是c[i-1, w]。这样，实际上对于到底要不要取第i个物品，我们只要比较这两种情况，哪个的结果值更大不就是最优的么？

在前面讨论的关系里，还有一个情况我们需要考虑的就是，我们这个最优解是基于选择物品i时总重量还是在w范围内的，如果超出了呢？我们肯定不能选择它，这就和c[i-1, w]一样。

另外，对于初始的情况呢？很明显c[0, w]里不管w是多少，肯定为0。因为它表示我们一个物品都不选择的情况。c[i, 0]也一样，当我们总重量限制为0时，肯定价值为0。

这样，基于我们前面讨论的这3个部分，我们可以得到一个如下的递推公式：

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0\,, \\ c[i-1,w] & \text{if } w_i > w\,, \\ \max(v_i + c[i-1, w-w_i], c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i\,. \end{cases}$$

有了这个关系，我们可以更进一步的来考虑代码实现了。我们有这么一个递归的关系，其中，后面的函数结果其实是依赖于前面的结果的。我们只要按照前面求出来最基础的最优条件，然后往后面一步步递推，就可以找到结果了。

我们再来考虑一下具体实现的细节。这一组物品分别有价值和重量，我们可以定义两个数组int[] v, int[] w。v[i]表示第i个物品的价值，w[i]表示第i个物品的重量。为了表示c[i, w]，我们可以使用一个int[i][w]的矩阵。其中i的最大值为物品的数量，而w表示最大的重量限制。按照前面的递推关系，c[i][0]和c[0][w]都是0。而我们所要求的最终结果是c[n][w]。所以我们实际中创建的矩阵是(n + 1) x (w + 1)的规格。下面是该过程的一个代码参考实现：

```
public class DynamicKnapSack {
    private int[] v;
    private int[] w;
    private int[][] c;
    private int weight;

    public DynamicKnapSack(int length, int weight, int[] vin, int[] win) {
        v = new int[length + 1];
        w = new int[length + 1];
        c = new int[length + 1][weight + 1];
        this.weight = weight;
        for(int i = 0; i < length + 1; i++) {
            v[i] = vin[i];
            w[i] = win[i];
        }
```

```java
    }

    public void solve() {
       for(int i = 1; i < v.length; i++) {
            for(int k = 1; k <= weight; k++) {
                if(w[i] <= k) {
                    if(v[i] + c[i - 1][k - w[i]] > c[i - 1][k])
                        c[i][k] = v[i] + c[i - 1][k - w[i]];
                    else
                        c[i][k] = c[i - 1][k];
                } else
                    c[i][k] = c[i - 1][k];
            }
        }
    }

    public void printResult() {
        for(int i = 0; i < v. length; i++) {
            for(int j = 0; j <= weight; j++)
                System.out.print(c[i][j] + " ");
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int[] v = {0, 60, 100, 120};
        int[] w = {0, 10, 20, 30};
        int weight = 50;
        DynamicKnapSack knapsack = new DynamicKnapSack(3, weight, v, w);
        knapsack.solve();
        knapsack.printResult();
    }
}
```

这部分代码里关键的就是solve方法。里面两个遍历循环，i表示从1到n的范围，对应于我们递归方法里描述的c(i, w)中到第i位。而k表示的是当前的重量限制。下面是程序运行的输出结果：

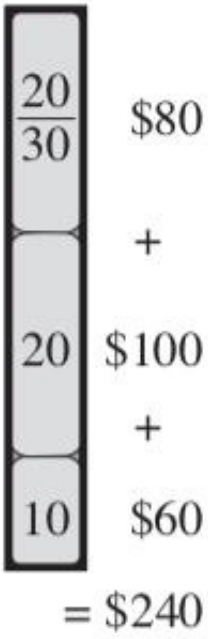
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60 60
0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 100 100 100 100 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160 160
0 0 0 0 0 0 0 0 0 0 60 60 60 60 60 60 60 60 60 60 100 100 100 100 100 100 100 100 100 100 160 160 160 160 160 160 160 160 160 160 180 180 180 180 180 180 180 180 180 180 220
```

最右下角的数值220就是c[3, 50]的解。
至此，我们对于这种问题的解决方法已经分析出来了。它的总体时间复杂度为O(nw)，其中w是设定的一个重量范围，因此也可以说它的时间复杂度为O(n)。

二、部分背包问题

和前面使用动态规划方法解决问题不一样。因为这里是部分背包问题，我们可以采用前面讨论过的一个思路。就是每次选择最优单位价格的物品，直到达到背包重量限制要求。

以前面的示例来看，我们按照这种方式选择的物品结果应该如下图：



现在，我们从实现的角度再来考虑一下。我们这里的最优解是每次挑选性价比最高的物品。对于这一组物品来说，我们需要将他们按照性价比从最高到最低的顺序来取。我们可能需要将他们进行排序。然后再依次取出来放入背包中。假定我们已经有数组v，w，他们已经按照性价比排好序了。一个参考代码的实现如下：

```
public double selectMax() {
        double maxValue = 0.0;
        int sum = 0;
        int i;
        for(i = 0; i < v.length; i++) {
            if(sum + w[i] < weight) {
                sum += w[i];
                maxValue += v[i];
            } else
                break;
```

```
        }
        if(i < v.length && sum < weight) {
            maxValue += (double)(weight - sum) / w[i] * v[i];
        }
        return maxValue;
    }
```

这里省略了对数组v, w的定义。关键点在于我们选择了若干了物品后要判断是否装满了背包重量。如果没有，还要从后面的里面挑选一部分。所以有一个if(i < v.length && sum < weight)的判断。

在实现后我们来看该问题这种解法的时间复杂度，因为需要将数组排序，我们的时间复杂度为O(nlgn)。

在前面我们挑选按照性价比排好序的物品时，排序消耗了主要的时间。在这里，我们是否真的需要去把这些物品排序呢？在某些情况下，我们只要选择一堆物品，保证他们物品重量在指定范围内。如果我们一次挑出来一批这样的物品，而且他们满足这样的条件是不是更好呢？这一种思路是借鉴快速排序里对元素进行划分的思路。主要过程如下：

1. 求每个元素的单位价值，pi = vi /wi。然后数组按照pi进行划分，这样会被分成3个部分，L, M, N。其中L < M < N。这里L表示单位价值小于某个指定值的集合，M是等于这个值的集合，而N是大于这个值的集合。
2. 我们可以首先看N的集合，因为这里都是单位价值高的集合。我们将他们的重量累加，如果WN的重量等于我们期望的值W，则N中间的结果就是我们找到的结果。
3. 如果WN的重量大于W，我们需要在N集合里做进一步划分。
4. 如果WN的重量小于W，我们需要在N的基础上再去L的集合里划分，找里面大的一部分。

这样重复步骤1到4.

这里和快速排序的思路基本上差不多，只是需要将一个分割的集合给记录下来。其时间复杂度也更好一点，为O(N)。这里就简单的描述下思路，等后续再将具体的实现代码给补上。

三、总结

我们这里讨论的两种背包问题因为问题的不同其本质解决方法也不同。对于0-1背包来说，他们构成了一个最优解问题的基础。我们可以通过从最小的结果集递推出最终最优结果。他们之间构成了一个递归的关系。而对于部分背包问题来说，我们可以考虑用贪婪算法，每次选择当前看来最优的结果。最终也构成了一个最优的结果。一个小小的前提变化，问题解决的思路却大不同。里面的思想值得反复体会。