背包九讲学习笔记

> “失败不是什么丢人的事情， 从失败中全无收获才是。”
>
> ——崔添翼

1st_01背包问题

$f[i][j]$ 表示前 i 件物品在容量为 j 的背包中所能获取到的最大价值。状态 $f[i][j]$ 是由**先前状态+决策**得到的，**先前状态**是指我们已知前 $i-1$ 件物品在容量为 $0-j$ 的背包中取值的最优解(即得到最大价值)，**决策**是指我们从前 $i-1$ 件物品的最优解到前 i 件物品的最优解需要作出的选择。在这里，我们需要作出的选择有两种，一种是不将第 i 件物品加入背包，一种是将第 i 件物品加入背包，并从可选择的决策中选出最优的决策，其对应状态方程

$$f[i][j]=max(f[i-1][j],f[i-1][j-c[i]])$$

若我们能够保证 $f[i-1][0...j]$ 是最优解，则我们由此得到的 $f[i][j]$ 必然也是最优解。
C++实现代码如下

```
// c[i] 表示第 i 件物品所占用体积，w[i] 表示第 i 件物品的价值
for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= m; ++j)
        dp[i][j] = max(dp[i-1][j], dp[i-1][j-c[i]]+w[i]);
```

2nd_完全背包问题

与 01 背包问题 的不同，在完全背包问题中，每一种物品都有无数个。沿用01背包状态**先前状态+决策**的思想，这里 $dp[i][j]$ 可以由若干决策得到，第 k 个决策表示取 k 个第 i 种物品放入背包，我们要从这些决策中挑选出最优策略，只需添加一个循环语句即可。

```
for(int i = 1; i <= n; ++i)
    for(int j = 1; j <= m; ++j)
        for(int k = 1; k <= j/c[i]; ++k)
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-k*c[i]] + k*w[i]);
```

上述代码时间复杂度为 $O(n^3)$，可以通过推导公式得到一个优化，考虑：

$$dp[i][j]=max(dp[i-1][j],dp[i-1][j-k*c[i]]+k*w[i]),k\in(1,\lfloor\frac{j}{c[i]}\rfloor)$$

$$dp[i][j-c[i]]=max(dp[i-1][j-c[i]],dp[i-1][j-c[i]-k*c[i]]+k*w[i]),k\in(1,\lfloor\frac{j-w[i]}{c[i]}\rfloor)$$

可以发现

$$\begin{aligned}&max(dp[i-1][j-k*c[i]]+k*w[i])(k\in(1,\lfloor\frac{j}{c[i]}\rfloor)\\=&max(dp[i-1][j-c[i]]+w[i],dp[i-1][j-(k+1)*w[i]]+(k+1)*w[i])(k\in(1,\lfloor\frac{j}{c[i]}\rfloor-1)\\=&max(dp[i-1][j-c[i]]+w[i],dp[i-1][j-c[i]-k*c[i]]+k*w[i]+w[i]),k\in(1,\lfloor\frac{j-w[i]}{c[i]}\rfloor)=\quad dp[i][j-c[i]]+w[i]\end{aligned}$$

因此得到**优化公式**：

$$dp[i][j]=max(dp[i-1][j],dp[i-1][j-k*c[i]]+k*w[i])=max(dp[i-1][j],dp[i][j-c[i]]+w[i])$$

故化简后的递推方程为：

$$dp[i][j] = max(dp[i-1][j], dp[i][j-c[i]] + w[i])$$

该公式表示 $max(dp[i-1][j-k*c[i]] + k*w[i])$ 与 $dp[i][j-c[i]] + w[i]$ 的值是相同的，因此可将上述代码中 $O(n^3)$ 的时间复杂度减少为 $O(n^2)$ 的时间复杂度。
上述是数学上的证明，也可以通过递归状态来理解**优化公式**。若前 i 种物品可选，背包容量为 j，若想得到最优解，可以通过两种策略得到。一种是根本不选第 i 种物品，则 $dp[i][j] = dp[i-1][j]$，另一种是选择第 i 种物品，此时我们有最优状态 $dp[i][1...j-1]$(在这些状态中我们可能已经选择了第 i 件物品)，可以基于此来得到若再次选择第 i 件物品的价值 $dp[i][j] = dp[i][j-c[i]]$。由此思想也可以得到上述递推方程式。

一个简单的优化

在解决问题之前，有一个 $O(n^2)$ 的算法可以对其进行优化。即对所有物品进行两两比较，若存在 $c[i] < c[j]$ 且 $w[i] > w[j]$，则可以抛弃物品 j。

3rd_多重背包问题

多重背包的 $O(n\sum logm)$ 解法

多重背包问题介于 01 背包问题和完全背包问题，其规定每一种物体最多可用 M_i 个。考虑将其转化为 01 背包问题，则其相当于有 $\sum M_i$ 个物品的 01 背包问题(相当于将 1 种物品看做 M_i 个不同的物品)。如果这样的话，则其时间复杂度为$O(n * \sum M_i)$，时间复杂度较大，应考虑优化。

我们使用二进制的思想进行优化。考虑这样一个问题，对于数字 10，我们可以通过 $a[1]*1 + a[2]*1 + a[3]*1 + \ldots + a[10]*1$的方式取得$1-10$以内的所有值，其中$a[i] \in \{0,1\}$，我们也有另一种方式取到$1-10$以内的值，考虑 $a[1]*1 + a[2]*2 + a[3]*2^2 + a[4]*(10-1-2-2^2)$，同样可以达到目标。使用相同的思想，对于第 i 种物品，可以进行相关的划分，不再将其看成 M_i 个独立的物品，而是将其划分成若干组，这些组中分别含有 $\{2^0, 2^1, 2^2, \ldots 2^{k-1}, M_i - 2^k + 1\}$ 个第 i 种物品，通过这些组的组合，我们可以得到范围内任意数量的第 i 种物品，而我们将每一组看成一个独立的物品，其容量为 $m_k * c[i]$，价值为 $m_k * w[i]$，$m[1...k+1] = \{2^0, 2^1, 2^2, \ldots 2^{k-1}, M_i - 2^k + 1\}$。

使用上述思想求解问题，问题的时间复杂度就降为$O(n * \sum logm_i)$，大大降低了时间复杂度。貌似可以使用优先队列优化，但好像考题很少，此处略过~~~。

实现代码：

```
void OneZeroPack(int * dp, int n, int c, int w){
    for(int i = n; i >= c; --i)
        dp[i] = max(dp[i], dp[i-c]+w);
}
void CompletePack(int * dp, int n, int c, int w){
    for(int i = c; i <= n; ++i)
        dp[i] = max(dp[i], dp[i-c]+w);
}
// m 表示该物品最多的个数
void MultiplePack(int * dp, int n, int c, int w, int m){
    if(c*m >= n){
        // 转化为完全背包问题
        CompletePack(dp, n, c, w);
        return;
    }
    int k = 1;
    while(k < m){
        OneZeroPack(dp, n, k*c, k*w);
```

```
        m -= k;
        k <<= 1;
    }
    OneZeroPack(dp, n, m*c, m*w);
}
```

多重背包可行性问题的 $O(VN)$ 解法

若只考虑“每种有若干件的物品能否恰好填满背包”，而不考虑每一件物品的价值，那么有着 $O(N)$ 的算法。使用 $f[i][j]$ 表示“使用前 i 件物品恰好填满了容量为 j 的背包后，最多还剩下第 i 件物品的件数”，若 $F[i][j]=-1$ 表示前 i 件物品无法恰好填满容量为 j 的背包。可以写出以下代码：

```
for(int i = 1; i <= m; ++i)
    dp[i] = -1;
dp[0] = 0;
for(int i = 1; i <= n; ++i){
    // 从前 i-1 件物品转移到前 i 件物品
    for(int j = 1; j <= m; ++j)
        dp[j] = dp[j] == -1 ? -1 : num[i];
    // 试图填满之前无法填满的背包
    for(int j = c[i]; j <= m; ++j)
        // 状态为 j 时是否使用第 i 件物品
        if(dp[j-c[i]] > 0)
            dp[j] = max(dp[j], dp[j-c[i]] - 1);
}
```

4th_混合三种背包问题

如果在一个背包中，有的物体的个数有限、有的物体的个数仅有一个、有的物体的个数无限多个，这就是混合三种背包问题，对此，只需要将上述代码综合起来即可。伪代码如下：

```
for(int i = 1; i <= n; ++i)
    if 第 i 种物品有一个
        ZeroOnePack(dp, m, c[i], w[i]);
    else if 第 i 种物品有无限个
        CompletePack(dp, m, c[i], w[i]);
    else 第 i 种物品有 num 个
        MultiplePack(dp, m, c[i], w[i], num);
```

5th_二维费用的背包问题

二维费用的背包问题指对于每件物品有 C、D 两种容量，同时对于这两种费用背包各有一个容量上限，在此基础上求背包所能达到的最大价值。

设 $f[i][u][v]$ 表示前 i 件物品中背包的两种容量分别为 u 和 v 的情况下所能取得的最大价值，状态方程如下

$$f[i][u][v]=max(f[i-1][u][v],f[i-1][u-C_i][v-D_i]+w[i])$$

给出伪代码：

```
for(int i = 1; i <= n; ++i)
    for(int j = s; j >= c[i]; --j)
        for(int k = t; k >= d[i]; --k)
            f[j][k] = max(dp[s][t], dp[s-c[i]][t-d[i]] + w[i]);
```

6th_分组的背包问题

有 n 个物品，将其划分为 k 组，每一组内部的物品最多选取一件，试求出该背包问题的最大解法。设 $dp[k][j]$ 表示前 k 组在容量为 j 的背包中的最大价值，其与 01背包 的区别是：**前者的决策是决定在第 k 组中是否选出物品以及选出哪一个物品，后者是决定第 i 件物品是否选中。**递推方程如下：

$$dp[k][j] = max(dp[k-1][j], dp[k-1][j-c[k][i]] + w[k][i] | i \in (1, num[k]))$$

其中 $c[k][i]$ 和 $w[k][i]$ 表示第 k 组中第 i 个物品的容量和价值，$num[k]$ 表示第 k 组物品的数目。

给出代码：

```
for(int k = 1; k <= K; ++k)
    for(int j = m; j >= 1; ++j)
        for(int i = 1; i <= num[k]; ++i)
            if(j >= c[k][i])
                dp[j] = max(dp[j], dp[j-c[k][i]] + w[k][i]);
```

7th_有依赖的背包问题

若选择了物品 j，则必须先选择物品 i，我们称物品 j 依赖于物品 i。有依赖的背包问题指的就是当背包中的物品满足若干依赖关系式如何对该类问题进行求解。简化问题，我们仅假设依赖没有连续性，即不会出现 A 依赖于 B，而 B 依赖于 C 的情况；又假设一件物品只能依赖于一件物品。

下面的话完全引用自崔添翼的背包九讲(但是我添加了部分锚点)，~~--我实在无法找到更美妙的描述了--~~

> 按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件……无法用状态转移方程来表示如此多的策略。事实上，设有 n 个附件，则策略有 $2^n + 1$ 个，为指数级。
>
> 考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于分组背包中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。
>
> 再考虑对每组内的物品应用完全背包中的优化。我们可以想到，对于第 k 个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，可以对主件 k 的“附件集合”先进行一次 01 背包，得到费用依次为 $0...V - C_k$，所有这些值时相应的最大价值 $F_k[0...V - C_k]$。那么，这个主件及它的附件集合相当于 $V - C_k + 1$ 个物品的物品组，其中费用为 v 的物品的价值为 $F_k[v - C_k] + W_k$，v 的取值范围是 $C_k \le v \le V$。
>
> 也就是说，原来指数级的策略中，有很多策略都是冗余的，通过一次 01 背包后， 将主件 k 及其附件转化为 $V - C_k + 1$ 个物品的物品组，就可以直接应用完全背包的算法解决问题了。

实现代码：

```
// 计算第 k 个物品组
// 得到重量为 0~m-c[k][0]，价值为 f[k][0]~f[k][m-c[k][0]] 的物品
// 这些物品代表同等重量中价值最高的物品组合
for(int i = 1; i < c[k].size(); ++i)
    for(int j = m-c[k][0]; j >= c[k][i]; --j)
        f[k][j] = max(f[k][j], f[k][j - c[k][i]] + w[k][i]);
for(int k = 1; k <= K; ++k)
    for(int j = m; j >= 0; --j)
        for(int i = 0; i <= m - c[k][0]; ++i)
            dp[j] = max(dp[j], dp[j - i - c[k][0]] + f[k][i] + c[k][0]);
```

现在考虑更加一般的情况。如果每个物品仍然只能依赖于一个物品，但是可以连续以来(不能循环依赖)，就构成了若干依赖树，那么该如何求解呢?可以采用相同的思路。仍然可以将一个主件即其附件集合看做一个物品组，但是由于其附件还有附件，因此需要先将其附件以及附件的附件看做一个物品组，以此类推。每当要求一个根节点的物品组时，先求得其子节点的物品组，这是一种树形动态规划的形式。

需要注意的是，我发现以上方法虽然通用但不代表是最优解。对组内进行朴素分组的时间复杂度为 $O(2^n)$，而通过上述方法进行分组的时间复杂度为 $O(k*C)$，其中 C 表示背包的容量。看似我们大大降低了时间复杂度，但是当面对 n 较小而 C 较大的情况时往往采用朴素分组的时间复杂度会更优。例子可以参考 金明的预算方案。虽然如此，我认为上述方法对于思维的开拓是极其具有意义和价值的。

8th_泛化物品

泛化物品是对背包内可装入的物品的一个更加抽象与广义的概念。其没有固定体积和价值，其价值随着被分配给它的体积而变化。即其价值与体积为一个函数关系，满足 $w=f(c)$，我们可称此方程为**泛化函数**。

对于一个容量为 c，价值为 w 的物品，在 01 背包问题中其**泛化函数**为

$$f(x)=\begin{cases} w, x=c \\ 0, x\neq c \end{cases}$$

在完全背包中其泛化函数为

$$f(x)=\begin{cases} \frac{x}{c}*w, c \mid x \\ 0, c \nmid x \end{cases}$$

多重背包同理。

一个物品组也可以看做一个泛化物品。若物品组中存在容量为 c 的物品，则 $f(c)$ 为所有容量为 c 的物品中的最大价值，若不存在容量为 c 的物品，则 $f(c)=0$。同样，对于依赖背包问题，我们可以按照之前所述的两种方式对其进行分组，得到一个依赖分组中的泛化函数。

现在有两个泛化物品 h 和 l，若给定他们以最大空间 v，通过分配 h 和 v 不同的空间，求得 v 所能够得到的最大价值 $f(v)$，$f(v)$ 满足

$$f(v)=max\{h(x)+l(v-x)|0\leq x\leq v\}$$

而通过 v 的不同取值，我们可以得到一个新的泛化物品 f，其代表泛化物品 h 和 l 的和。此时以 f 代替物品 h 和 l，结果不会受到任何影响。

求解背包问题，就是不断地合并泛化物品并求解泛化物品最大值的过程，其通用形式如下

```
// 集合 S 表示一个泛化物品的集合
for k in S
    for v <- C to 0
        // 合并泛化物品 obj 和 k，更新泛化物品 obj
        obj[v] <- max(obj[v - x] + k[x])
return obj[C]
```

9th_背包问题的变化

这里我会专门再写一篇博客来探究背包问题的不同问法，此处先留个坑。^_^