背包問題九講

作者：dd_engi

資料來源：

http://www.oiers.cn/pack/Index.html

（version 1.1 build 20071115）

前言

本篇文章是我(dd_engi)正在進行中的一個雄心勃勃的寫作計劃的一部分，這個計劃的內容是寫作一份較爲完善的 NOIP 難度的動態規劃總結，名爲《解動態規劃題的基本思考方式》。現在你看到的是這個寫作計劃最先發佈的一部分。

背包問題是一個經典的動態規劃模型。它既簡單形象容易理解，又在某種程度上能夠揭示動態規劃的本質，故不少教材都把它作爲動態規劃部分的第一道例題，我也將它放在我的寫作計劃的第一部分。

讀本文最重要的是思考。因爲我的語言和寫作方式向來不以易於理解爲長，思路也偶有跳躍的地方，後面更有需要大量思考才能理解的比較抽象的內容。更重要的是：不大量思考，絕對不可能學好動態規劃這一信息學奧賽中最精緻的部分。

你現在看到的是本文的 v1.1 版，發佈於 2007 年 11 月 15 日。我會長期維護這份文本，把大家的意見和建議融入其中，也會不斷加入我在 OI 學習以及將來可能的 ACM-ICPC 的征程中得到的新的心得。但目前本文還沒有一個固定的發布頁面，想瞭解本文是否有更新版本發佈，可以在 OIBH 論壇中以「背包問題九講」爲關鍵字搜索貼子，每次比較重大的版本更新都會在這個論壇裡發貼公佈。也可以用「背包問題九講」爲關鍵字在搜索引擎中搜索以得到最新版本。

目錄

P01: 01 背包問題

題目

有 N 件物品和一個容量爲 V 的背包。第 i 件物品的費用是 c[i]，價值是 w[i]。求解將哪些物品裝入背包可使價值總和最大。

基本思路

這是最基礎的背包問題，特點是：每種物品僅有一件，可以選擇放或不放。

用子問題定義狀態：即 f[i][v]表示前 i 件物品恰放入一個容量爲 v 的背包可以獲得的最大價值。則其狀態轉移方程便是：

```
f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
```

這個方程非常重要，基本上所有跟背包相關的問題的方程都是由它衍生出來的。所以有必要將它詳細解釋一下：「將前 i 件物品放入容量爲 v 的背包中」這個子問題，若只考慮第 i 件物品的策略（放或不放），那麼就可以轉化爲一個只牽扯前 i-1 件物品的問題。如果不放第 i 件物品，那麼問題就轉化爲「前 i-1 件物品放入容量爲 v 的背包中」，價值爲 f[i-1][v]；如果放第 i 件物品，那麼問題就轉化爲「前 i-1 件物品放入剩下的容量爲 v-c[i]的背包中」，此時能獲得的最大價值就是 f[i-1][v-c[i]]再加上通過放入第 i 件物品獲得的價值 w[i]。

優化空間複雜度

以上方法的時間和空間複雜度均爲 O(VN)，其中時間複雜度應該已經不能再優化了，但空間複雜度卻可以優化到 O。

先考慮上面講的基本思路如何實現，肯定是有一個主循環 i=1..N，每次算出來二維數組 f[i][0..V]的所有值。那麼，如果只用一個數組 f[0..V]，能不能保證第 i 次循環結束後 f[v]中表示的就是我們定義的狀態 f[i][v]呢？f[i][v]是由 f[i-1][v]和 f[i-1][v-c[i]]兩個子問題遞推而來，能否保證在推 f[i][v]時（也即在第 i 次主循環中推 f[v]時）能夠得到 f[i-1][v]和 f[i-1][v-c[i]]的值呢？事實上，這要求在每次主循環中我們以 v=V..0 的順

序推 f[v]，這樣才能保證推 f[v]時 f[v-c[i]]保存的是狀態 f[i-1][v-c[i]]的值。僞代碼如下：

```
for i=1..N
    for v=V..0
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

其中的 f[v]=max{f[v],f[v-c[i]]}一句恰就相當於我們的轉移方程 f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因爲現在的 f[v-c[i]]就相當於原來的 f[i-1][v-c[i]]。如果將 v 的循環順序從上面的逆序改成順序的話，那麼則成了 f[i][v]由 f[i][v-c[i]]推知，與本題意不符，但它卻是另一個重要的背包問題 P02 最簡捷的解決方案，故學習只用一維數組解 01 背包問題是十分必要的。

事實上，使用一維數組解 01 背包的程序在後面會被多次用到，所以這裡抽象出一個處理一件 01 背包中的物品過程，以後的代碼中直接調用不加說明。

過程 ZeroOnePack，表示處理一件 01 背包中的物品，兩個參數 cost、weight 分別表明這件物品的費用和價值。

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

注意這個過程裡的處理與前面給出的僞代碼有所不同。前面的示例程序寫成 v=V..0 是爲了在程序中體現每個狀態都按照方程求解了，避免不必要的思維複雜度。而這裡既然已經抽象成看作黑箱的過程了，就可以加入優化。費用爲 cost 的物品不會影響狀態 f[0..cost-1]，這是顯然的。

有了這個過程以後，01 背包問題的僞代碼就可以這樣寫：

```
for i=1..N
    ZeroOnePack(c[i],w[i]);
```

初始化的細節問題

我們看到的求最優解的背包問題題目中，事實上有兩種不太相同的問法。有的題目要求「恰好裝滿背包」時的最優解，有的題目則並沒有要求必須把背包裝滿。一種區別這兩種問法的實現方法是在初始化的時候有所不同。

如果是第一種問法，要求恰好裝滿背包，那麼在初始化時除了 f[0] 爲 0 其它 f[1..V]均設爲-∞，這樣就可以保證最終得到的 f[N]是一種恰好裝滿背包的最優解。

如果並沒有要求必須把背包裝滿，而是只希望價格儘量大，初始化時應該將 f[0..V]全部設爲 0。

爲什麼呢？可以這樣理解：初始化的 f 數組事實上就是在沒有任何物品可以放入背包時的合法狀態。如果要求背包恰好裝滿，那麼此時只有容量爲 0 的背包可能被價值爲 0 的 nothing「恰好裝滿」，其它容量的背包均沒有合法的解，屬於未定義的狀態，它們的值就都應該是-∞了。如果背包並非必須被裝滿，那麼任何容量的背包都有一個合法解「什麼都不裝」，這個解的價值爲 0，所以初始時狀態的值也就全部爲 0 了。

這個小技巧完全可以推廣到其它類型的背包問題，後面也就不再對進行狀態轉移之前的初始化進行講解。

一個常數優化

前面的僞代碼中有 for v=V..1，可以將這個循環的下限進行改進。

由於只需要最後 f[v]的值，倒推前一個物品，其實只要知道 f[v-w[n]]即可。以此類推，對以第 j 個背包，其實只需要知道到 f[v-sum{w[j..n]}]即可，即代碼中的

```
for i=1..N
    for v=V..0
```

可以改成

```
for i=1..n
    bound=max{V-sum{w[i..n]},c[i]}
    for v=V..bound
```

這對於 V 比較大時是有用的。

小結

01 背包問題是最基本的背包問題，它包含了背包問題中設計狀態、方程的最基本思想，另外，別的類型的背包問題往往也可以轉換成 01 背包問題求解。故一定要仔細體會上面基本思路的得出方法，狀態轉移方程的意義，以及最後怎樣優化的空間複雜度。

P02: 完全背包問題

題目

有 N 種物品和一個容量爲 V 的背包，每種物品都有無限件可用。 第 i 種物品的費用是 c[i]，價值是 w[i]。 求解將哪些物品裝入背包可使這些物品的費用總和不超過背包容量，且價值總和最大。

基本思路

這個問題非常類似於 01 背包問題，所不同的是每種物品有無限件。 也就是從每種物品的角度考慮，與它相關的策略已並非取或不取兩種，而是有取 0 件、取 1 件、取 2 件……等很多種。 如果仍然按照解 01 背包時的思路，令 f[i][v]表示前 i 種物品恰放入一個容量爲 v 的背包的最大權值。仍然可以按照每種物品不同的策略寫出狀態轉移方程，像這樣：

```
f[i][v]=max{f[i-1][vk*c[i]]+k*w[i]|0<=k*c[i]<=v}
```

這跟 01 背包問題一樣有 O(VN)個狀態需要求解，但求解每個狀態的時間已經不是常數了，求解狀態 f[i][v]的時間是 O(v/c[i])，總的複雜度可以認爲是 O(V*Σ(V/c[i]))，是比較大的。

將 01 背包問題的基本思路加以改進，得到了這樣一個清晰的方法。 這說明 01 背包問題的方程的確是很重要，可以推及其它類型的背包問題。 但我們還是試圖改進這個複雜度。

一個簡單有效的優化

完全背包問題有一個很簡單有效的優化，是這樣的：若兩件物品 i、j 滿足 c[i]<=c[j]且 w[i]>=w[j]，則將物品 j 去掉，不用考慮。 這個優化的正確性顯然：任何情況下都可將價值小費用高得 j 換成物美價廉的 i，得到至少不會更差的方案。 對於隨機生成的數據，這個方法往往會大大減少物品的件數，從而加快速度。 然而這個並不能改善最壞情況的複雜度，因爲有可能特別設計的數據可以一件物品也去不掉。

這個優化可以簡單的 O(N^2)地實現，一般都可以承受。 另外，針對背包問題而言，比較不錯的一種方法是：首先將費用大於 V 的物品去掉，然後使用類似計數排序的做法，計算出費用相同的物品中價值最高的是哪個，可以 O (V+N)地完成這個優化。 這個不太重要的過程就不給出僞代碼了，希望你能獨立思考寫出僞代碼或程序。

轉化爲 01 背包問題求解

既然 01 背包問題是最基本的背包問題，那麼我們可以考慮把完全背包問題轉化爲 01 背包問題來解。最簡單的想法是，考慮到第 i 種物品最多選 V/c[i]件，於是可以把第 i 種物品轉化爲 V/c[i]件費用及價值均不變的物品，然後求解這個 01 背包問題。這樣完全沒有改進基本思路的時間複雜度，但這畢竟給了我們將完全背包問題轉化爲 01 背包問題的思路：將一種物品拆成多件物品。

更高效的轉化方法是：把第 i 種物品拆成費用爲 c[i]*2^k、價值爲 w[i]*2^k 的若干件物品，其中 k 滿足 c[i]*2^k<=V。這是二進制的思想，因爲不管最優策略選幾件第 i 種物品，總可以表示成若干個 2^k 件物品的和。這樣把每種物品拆成 O(log V/c[i])件物品，是一個很大的改進。

但我們有更優的 O(VN)的算法。

O(VN)的算法

這個算法使用一維數組，先看僞代碼：

```
for i=1..N
    for v=0..V
        f[v]=max{f[v],f[v-cost]+weight}
```

你會發現，這個僞代碼與 P01 的僞代碼只有 v 的循環次序不同而已。爲什麼這樣一改就可行呢？首先想想爲什麼 P01 中要按照 v=V..0 的逆序來循環。這是因爲要保證第 i 次循環中的狀態 f[i][v]是由狀態 f[i-1][v-c[i]]遞推而來。換句話說，這正是爲了保證每件物品只選一次，保證在考慮「選入第 i 件物品」這件策略時，依據的是一個絕無已經選入第 i 件物品的子結果 f[i-1][v-c[i]]。而現在完全背包的特點恰是每種物品可選無限件，所以在考慮「加選一件第 i 種物品」這種策略時，卻正需要一個可能已選入第 i 種物品的子結果 f[i][v-c[i]]，所以就可以並且必須採用 v=0..V 的順序循環。這就是這個簡單的程序爲何成立的道理。

值得一提的是，上面的僞代碼中兩層 for 循環的次序可以顛倒。這個結論有可能會帶來算法時間常數上的優化。

這個算法也可以以另外的思路得出。例如，將基本思路中求解 f[i][v-c[i]]的狀態轉移方程顯式地寫出來，代入原方程中，會發現該方程可以等價地變形成這種形式：

```
f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}
```

將這個方程用一維數組實現，便得到了上面的僞代碼。

最後抽象出處理一件完全背包類物品的過程僞代碼：

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

小結

完全背包問題也是一個相當基礎的背包問題，它有兩個狀態轉移方程，分別在「基本思路」以及「O(VN)的算法」的小節中給出。希望你能夠對這兩個狀態轉移方程都仔細地體會，不僅記住，也要弄明白它們是怎麼得出來的，最好能夠自己想一種得到這些方程的方法。事實上，對每一道動態規劃題目都思考其方程的意義以及如何得來，是加深對動態規劃的理解、提高動態規劃功力的好方法。

P03: 多重背包問題

題目

有 N 種物品和一個容量爲 V 的背包。第 i 種物品最多有 n[i]件可用，每件費用是 c[i]，價值是 w[i]。求解將哪些物品裝入背包可使這些物品的費用總和不超過背包容量，且價值總和最大。

基本算法

這題目和完全背包問題很類似。基本的方程只需將完全背包問題的方程略微一改即可，因爲對於第 i 種物品有 n[i]+1 種策略：取 0 件，取 1 件……取 n[i]件。令 f[i][v]表示前 i 種物品恰放入一個容量爲 v 的背包的最大權值，則有狀態轉移方程：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k<=n[i]}

複雜度是 O(V*Σn[i])。

轉化爲 01 背包問題

另一種好想好寫的基本方法是轉化爲 01 背包求解：把第 i 種物品換成 n[i]件 01 背包中的物品，則得到了物品數爲 Σn[i]的 01 背包問題，直接求解，複雜度仍然是 O(V*Σn[i])。

但是我們期望將它轉化爲 01 背包問題之後能夠像完全背包一樣降低複雜度。仍然考慮二進制的思想，我們考慮把第 i 種物品換成若干件物品，使得原問題中第 i 種物品可取的每種策略——取 0..n[i]件——均能等價於取若干件代換以後的物品。另外，取超過 n[i]件的策略必不能出現。

方法是：將第 i 種物品分成若干件物品，其中每件物品有一個係數，這件物品的費用和價值均是原來的費用和價值乘以這個係數。使這些係數分別爲 1,2,4,...,2^(k-1),n[i]-2^k+1，且 k 是滿足 n[i]-2^k+1>0 的最大整數。例如，如果 n[i]爲 13，就將這種物品分成係數分別爲 1,2,4,6 的四件物品。

分成的這幾件物品的係數和爲 n[i]，表明不可能取多於 n[i]件的第 i 種物品。另外這種方法也能保證對於 0..n[i]間的每一個整數，均可以用

若干個係數的和表示，這個證明可以分 0..2^k-1 和 2^k..n[i]兩段來分別討論得出，並不難，希望你自己思考嘗試一下。

這樣就將第 i 種物品分成了 O(log n[i])種物品，將原問題轉化爲了複雜度爲<math>O(V*Σlog n[i])的 01 背包問題，是很大的改進。

下面給出 O(log amount)時間處理一件多重背包中物品的過程，其中 amount 表示物品的數量：

```
procedure MultiplePack(cost,weight,amount)
    if cost*amount>=V
        CompletePack(cost,weight)
        return
    integer k=1
    while k<amount
        ZeroOnePack(k*cost,k*weight)
        amount=amount-k
        k=k*2
    ZeroOnePack(amount*cost,amount*weight)
```

希望你仔細體會這個僞代碼，如果不太理解的話，不妨翻譯成程序代碼以後，單步執行幾次，或者頭腦加紙筆模擬一下，也許就會慢慢理解了。

O(VN)的算法

多重背包問題同樣有 O(VN)的算法。這個算法基於基本算法的狀態轉移方程，但應用單調隊列的方法使每個狀態的值可以以均攤 O(1)的時間求解。由於用單調隊列優化的 DP 已超出了 NOIP 的範圍，故本文不再展開講解。我最初瞭解到這個方法是在樓天成的「男人八題」幻燈片上。

小結

這裡我們看到了將一個算法的複雜度由 O(V*Σn[i])改進到 O(V*Σlog n[i])的過程，還知道了存在應用超出 NOIP 範圍的知識的 O(VN)算法。希望你特別注意「拆分物品」的思想和方法，自己證明一下它的正確性，並將完整的程序代碼寫出來。

P04: 混合三種背包問題

問題

如果將 P01、P02、P03 混合起來。也就是說，有的物品只可以取一次（01 背包），有的物品可以取無限次（完全背包），有的物品可以取的次數有一個上限（多重背包）。應該怎麼求解呢？

01 背包與完全背包的混合

考慮到在 P01 和 P02 中給出的偽代碼只有一處不同，故如果只有兩類物品：一類物品只能取一次，另一類物品可以取無限次，那麼只需在對每個物品應用轉移方程時，根據物品的類別選用順序或逆序的循環即可，複雜度是 O(VN)。偽代碼如下：

```
for i=1..N
    if 第 i 件物品屬於 01 背包
        for v=V..0
            f[v]=max{f[v],f[v-c[i]]+w[i]};
    else if 第 i 件物品屬於完全背包
        for v=0..V
            f[v]=max{f[v],f[v-c[i]]+w[i]};
```

再加上多重背包

如果再加上有的物品最多可以取有限次，那麼原則上也可以給出 O(VN)的解法：遇到多重背包類型的物品用單調隊列解即可。但如果不考慮超過 NOIP 範圍的算法的話，用 P03 中將每個這類物品分成 O(log n[i])個 01 背包的物品的方法也已經很優了。

當然，更清晰的寫法是調用我們前面給出的三個相關過程。

```
for i=1..N
    if 第 i 件物品屬於 01 背包
        ZeroOnePack(c[i],w[i])
    else if 第 i 件物品屬於完全背包
        CompletePack(c[i],w[i])
    else if 第 i 件物品屬於多重背包
```

```
MultiplePack(c[i],w[i],n[i])
```

在最初寫出這三個過程的時候，可能完全沒有想到它們會在這裡混合應用。我想這體現了編程中抽象的威力。如果你一直就是以這種「抽象出過程」的方式寫每一類背包問題的，也非常清楚它們的實現中細微的不同，那麼在遇到混合三種背包問題的題目時，一定能很快想到上面簡潔的解法，對嗎？

小結

有人說，困難的題目都是由簡單的題目疊加而來的。這句話是否公理暫且存之不論，但它在本講中已經得到了充分的體現。本來 01 背包、完全背包、多重背包都不是什麼難題，但將它們簡單地組合起來以後就得到了這樣一道一定能嚇倒不少人的題目。但只要基礎紮實，領會三種基本背包問題的思想，就可以做到把困難的題目拆分成簡單的題目來解決。

P05: 二維費用的背包問題

問題

二維費用的背包問題是指：對於每件物品，具有兩種不同的費用；選擇這件物品必須同時付出這兩種代價；對於每種代價都有一個可付出的最大值（背包容量）。問怎樣選擇物品可以得到最大的價值。設這兩種代價分別爲代價 1 和代價 2，第 i 件物品所需的兩種代價分別爲 a[i] 和 b[i]。兩種代價可付出的最大值（兩種背包容量）分別爲 V 和 U。物品的價值爲 w[i]。

算法

費用加了一維，只需狀態也加一維即可。設 f[i][v][u]表示前 i 件物品付出兩種代價分別爲 v 和 u 時可獲得的最大價值。狀態轉移方程就是：

```
f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}
```

如前述方法，可以只使用二維的數組：當每件物品只可以取一次時變量 v 和 u 採用逆序的循環，當物品有如完全背包問題時採用順序的循環。當物品有如多重背包問題時拆分物品。這裡就不再給出僞代碼了，相信有了前面的基礎，你能夠自己實現出這個問題的程序。

物品總個數的限制

有時，「二維費用」的條件是以這樣一種隱含的方式給出的：最多只能取 M 件物品。這事實上相當於每件物品多了一種「件數」的費用，每個物品的件數費用均爲 1，可以付出的最大件數費用爲 M。換句話說，設 f[v][m]表示付出費用 v、最多選 m 件時可得到的最大價值，則根據物品的類型（01、完全、多重）用不同的方法循環更新，最後在 f[0..V][0..M] 範圍內尋找答案。

複數域上的背包問題

另一種看待二維背包問題的思路是：將它看待成複數域上的背包問題。也就是說，背包的容量以及每件物品的費用都是一個複數。而常見的一維背包問題則是實數域上的背包問題。（注意：上面的話其實不嚴謹，因爲事實上我們處理的都只是整數而已。）所以說，一維背包的種

種思想方法，往往可以應用於二位背包問題的求解中，因爲只是數域擴大了而已。

作爲這種思想的練習，你可以嘗試將 P11 中提到的「子集和問題」擴展到複數域（即二維），並試圖用同樣的複雜度解決。

小結

當發現由熟悉的動態規劃題目變形得來的題目時，在原來的狀態中加一緯以滿足新的限制是一種比較通用的方法。希望你能從本講中初步體會到這種方法。

P06: 分組的背包問題

問題

有 N 件物品和一個容量爲 V 的背包。第 i 件物品的費用是 c[i]，價值是 w[i]。這些物品被劃分爲若干組，每組中的物品互相衝突，最多選一件。求解將哪些物品裝入背包可使這些物品的費用總和不超過背包容量，且價值總和最大。

算法

這個問題變成了每組物品有若干種策略：是選擇本組的某一件，還是一件都不選。也就是說設 f[k][v]表示前 k 組物品花費費用 v 能取得的最大權值，則有：

```
f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品 i 屬於組 k}
```

使用一維數組的僞代碼如下：

```
for 所有的組 k
    for v=V..0
        for 所有的 i 屬於組 k
            f[v]=max{f[v],f[v-c[i]]+w[i]}
```

注意這裡的三層循環的順序，甚至在本文的第一個 beta 版中我自己都寫錯了。「for v=V..0」這一層循環必須在「for 所有的 i 屬於組 k」之外。這樣才能保證每一組內的物品最多只有一個會被添加到背包中。

另外，顯然可以對每組內的物品應用 P02 中「一個簡單有效的優化」。

小結

分組的背包問題將彼此互斥的若干物品稱爲一個組，這建立了一個很好的模型。不少背包問題的變形都可以轉化爲分組的背包問題（例如 P07），由分組的背包問題進一步可定義「泛化物品」的概念，十分有利於解題。

P07: 有依賴的背包問題

簡化的問題

這種背包問題的物品間存在某種「依賴」的關係。也就是說，i 依賴於 j，表示若選物品 i，則必須選物品 j。爲了簡化起見，我們先設沒有某個物品既依賴於別的物品，又被別的物品所依賴；另外，沒有某件物品同時依賴多件物品。

算法

這個問題由 NOIP2006 金明的預算方案一題擴展而來。遵從該題的提法，將不依賴於別的物品的物品稱爲「主件」，依賴於某主件的物品稱爲「附件」。由這個問題的簡化條件可知所有的物品由若干主件和依賴於每個主件的一個附件集合組成。

按照背包問題的一般思路，僅考慮一個主件和它的附件集合。可是，可用的策略非常多，包括：一個也不選，僅選擇主件，選擇主件後再選擇一個附件，選擇主件後再選擇兩個附件……無法用狀態轉移方程來表示如此多的策略。（事實上，設有 n 個附件，則策略有 2^n+1 個，爲指數級。）

考慮到所有這些策略都是互斥的（也就是說，你只能選擇一種策略），所以一個主件和它的附件集合實際上對應於 P06 中的一個物品組，每個選擇了主件又選擇了若幹個附件的策略對應於這個物品組中的一個物品，其費用和價值都是這個策略中的物品的值的和。但僅僅是這一步轉化並不能給出一個好的算法，因爲物品組中的物品還是像原問題的策略一樣多。

再考慮 P06 中的一句話：*可以對每組中的物品應用 P02 中「一個簡單有效的優化」*。這提示我們，對於一個物品組中的物品，所有費用相同的物品只留一個價值最大的，不影響結果。所以，我們可以對主件 i 的「附件集合」先進行一次 01 背包，得到費用依次爲 0..V-c[i]所有這些值時相應的最大價值 f'[0..V-c[i]]。那麼這個主件及它的附件集合相當於 V-c[i]+1 個物品的物品組，其中費用爲 c[i]+k 的物品的價值爲 f'[k]+w[i]。也就是說原來指數級的策略中有很多策略都是冗餘的，通過一次 01 背包後，將主件 i 轉化爲 V-c[i]+1 個物品的物品組，就可以直接應用 P06 的算法解決問題了。

較一般的問題

更一般的問題是：依賴關係以圖論中「森林」的形式給出（森林即多叉樹的集合），也就是說，主件的附件仍然可以具有自己的附件集合，限制只是每個物品最多只依賴於一個物品（只有一個主件）且不出現循環依賴。

解決這個問題仍然可以用將每個主件及其附件集合轉化爲物品組的方式。唯一不同的是，由於附件可能還有附件，就不能將每個附件都看作一個一般的 01 背包中的物品了。若這個附件也有附件集合，則它必定要被先轉化爲物品組，然後用分組的背包問題解出主件及其附件集合所對應的附件組中各個費用的附件所對應的價值。

事實上，這是一種樹形 DP，其特點是每個父節點都需要對它的各個兒子的屬性進行一次 DP 以求得自己的相關屬性。這已經觸及到了「泛化物品」的思想。看完 P08 後，你會發現這個「依賴關係樹」每一個子樹都等價於一件泛化物品，求某節點爲根的子樹對應的泛化物品相當於求其所有兒子的對應的泛化物品之和。

小結

NOIP2006 的那道背包問題我做得很失敗，寫了上百行的代碼，卻一分未得。後來我通過思考發現通過引入「物品組」和「依賴」的概念可以加深對這題的理解，還可以解決它的推廣問題。用物品組的思想考慮那題中極其特殊的依賴關係：物品不能既作主件又作附件，每個主件最多有兩個附件，可以發現一個主件和它的兩個附件等價於一個由四個物品組成的物品組，這便揭示了問題的某種本質。

我想說：失敗不是什麼丟人的事情，從失敗中全無收穫才是。

P08: 泛化物品

定義

考慮這樣一種物品，它並沒有固定的費用和價值，而是它的價值隨著你分配給它的費用而變化。這就是泛化物品的概念。

更嚴格的定義之。在背包容量爲 V 的背包問題中，泛化物品是一個定義域爲 0..V 中的整數的函數 h，當分配給它的費用爲 v 時，能得到的價值就是 h(v)。

這個定義有一點點抽象，另一種理解是一個泛化物品就是一個數組 h[0..V]，給它費用 v，可得到價值 h[V]。

一個費用爲 c 價值爲 w 的物品，如果它是 01 背包中的物品，那麼把它看成泛化物品，它就是除了 h(c)=w 其它函數值都爲 0 的一個函數。如果它是完全背包中的物品，那麼它可以看成這樣一個函數，僅當 v 被 c 整除時有 h(v)=v/c*w，其它函數值均爲 0。如果它是多重背包中重複次數最多爲 n 的物品，那麼它對應的泛化物品的函數有 h(v)=v/c*w 僅當 v 被 c 整除且 v/c<=n，其它情況函數值均爲 0。

一個物品組可以看作一個泛化物品 h。對於一個 0..V 中的 v，若物品組中不存在費用爲 v 的的物品，則 h(v)=0，否則 h(v)爲所有費用爲 v 的物品的最大價值。P07 中每個主件及其附件集合等價於一個物品組，自然也可看作一個泛化物品。

泛化物品的和

如果面對兩個泛化物品 h 和 l，要用給定的費用從這兩個泛化物品中得到最大的價值，怎麼求呢？事實上，對於一個給定的費用 v，只需枚舉將這個費用如何分配給兩個泛化物品就可以了。同樣的，對於 0..V 的每一個整數 v，可以求得費用 v 分配到 h 和 l 中的最大價值 f(v)。也即

$$f(v)=\max\{h(k)+l(v-k)\mid 0<=k<=v\}$$

可以看到，f 也是一個由泛化物品 h 和 l 決定的定義域爲 0..V 的函數，也就是說，f 是一個由泛化物品 h 和 l 決定的泛化物品。

由此可以定義泛化物品的和：h、l 都是泛化物品，若泛化物品 f 滿足以上關係式，則稱 f 是 h 與 l 的和。這個運算的時間複雜度取決於背包的容量，是 O(V^2)。

泛化物品的定義表明：在一個背包問題中，若將兩個泛化物品代以它們的和，不影響問題的答案。事實上，對於其中的物品都是泛化物品的背包問題，求它的答案的過程也就是求所有這些泛化物品之和的過程。設此和爲 s，則答案就是 s[0..V]中的最大值。

背包問題的泛化物品

一個背包問題中，可能會給出很多條件，包括每種物品的費用、價值等屬性，物品之間的分組、依賴等關係等。但肯定能將問題對應於某個泛化物品。也就是說，給定了所有條件以後，就可以對每個非負整數 v 求得：若背包容量爲 v，將物品裝入背包可得到的最大價值是多少，這可以認爲是定義在非負整數集上的一件泛化物品。這個泛化物品——或者說問題所對應的一個定義域爲非負整數的函數——包含了關於問題本身的高度濃縮的信息。一般而言，求得這個泛化物品的一個子域（例如 0..V）的值之後，就可以根據這個函數的取值得到背包問題的最終答案。

綜上所述，一般而言，求解背包問題，即求解這個問題所對應的一個函數，即該問題的泛化物品。而求解某個泛化物品的一種方法就是將它表示爲若干泛化物品的和然後求之。

小結

本講可以說都是我自己的原創思想。具體來說，是我在學習函數式編程的 Scheme 語言時，用函數編程的眼光審視各類背包問題得出的理論。這一講真的很抽象，也許在「模型的抽象程度」這一方面已經超出了 NOIP 的要求，所以暫且看不懂也沒關係。相信隨著你的 OI 之路逐漸延伸，有一天你會理解的。

我想說：「思考」是一個 OIer 最重要的品質。簡單的問題，深入思考以後，也能發現更多。

P09: 背包問題問法的變化

以上涉及的各種背包問題都是要求在背包容量（費用）的限制下求可以取到的最大價值，但背包問題還有很多種靈活的問法，在這裡值得提一下。但是我認爲，只要深入理解了求背包問題最大價值的方法，即使問法變化了，也是不難想出算法的。

例如，求解最多可以放多少件物品或者最多可以裝滿多少背包的空間。這都可以根據具體問題利用前面的方程求出所有狀態的值（f 數組）之後得到。

還有，如果要求的是「總價值最小」「總件數最小」，只需簡單的將上面的狀態轉移方程中的 max 改成 min 即可。

下面說一些變化更大的問法。

輸出方案

一般而言，背包問題是要求一個最優值，如果要求輸出這個最優值的方案，可以參照一般動態規劃問題輸出方案的方法：記錄下每個狀態的最優值是由狀態轉移方程的哪一項推出來的，換句話說，記錄下它是由哪一個策略推出來的。便可根據這條策略找到上一個狀態，從上一個狀態接著向前推即可。

還是以 01 背包爲例，方程爲 f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。再用一個數組 g[i][v]，設 g[i][v]=0 表示推出 f[i][v]的值時是採用了方程的前一項（也即 f[i][v]=f[i-1][v]），g[i][v]表示採用了方程的後一項。注意這兩項分別表示了兩種策略：未選第 i 個物品及選了第 i 個物品。那麼輸出方案的僞代碼可以這樣寫（設最終狀態爲 f[N][V]）：

```
i=N
v=V
while(i>0)
    if(g[i][v]==0)
        print "未選第 i 項物品"
    else if(g[i][v]==1)
        print "選了第 i 項物品"
        v=v-c[i]
```

另外，採用方程的前一項或後一項也可以在輸出方案的過程中根據f[i][v]的值實時地求出來，也即不須紀錄 g 數組，將上述代碼中的 g[i][v]==0 改成 f[i][v]==f[i-1][v]，g[i][v]==1 改成 f[i][v]==f[i-1][v-c[i]]+w[i] 也可。

輸出字典序最小的最優方案

這裡「字典序最小」的意思是 1..N 號物品的選擇方案排列出來以後字典序最小。以輸出 01 背包最小字典序的方案爲例。

一般而言，求一個字典序最小的最優方案，只需要在轉移時注意策略。首先，子問題的定義要略改一些。我們注意到，如果存在一個選了物品 1 的最優方案，那麼答案一定包含物品 1，原問題轉化爲一個背包容量爲 v-c[1]，物品爲 2..N 的子問題。反之，如果答案不包含物品 1，則轉化成背包容量仍爲 V，物品爲 2..N 的子問題。不管答案怎樣，子問題的物品都是以 i..N 而非前所述的 1..i 的形式來定義的，所以狀態的定義和轉移方程都需要改一下。但也許更簡易的方法是先把物品逆序排列一下，以下按物品已被逆序排列來敘述。

在這種情況下，可以按照前面經典的狀態轉移方程來求值，只是輸出方案的時候要注意：從 N 到 1 輸入時，如果 f[i][v]==f[i-1][i-v]及 f[i][v]==f[i-1][f-c[i]]+w[i]同時成立，應該按照後者（即選擇了物品 i）來輸出方案。

求方案總數

對於一個給定了背包容量、物品費用、物品間相互關係（分組、依賴等）的背包問題，除了再給定每個物品的價值後求可得到的最大價值外，還可以得到裝滿背包或將背包裝至某一指定容量的方案總數。

對於這類改變問法的問題，一般只需將狀態轉移方程中的 max 改成 sum 即可。例如若每件物品均是完全背包中的物品，轉移方程即爲

```
f[i][v]=sum{f[i-1][v],f[i][v-c[i]]}
```

初始條件 f[0][0]=1。

事實上，這樣做可行的原因在於狀態轉移方程已經考察了所有可能的背包組成方案。

最優方案的總數

這裡的最優方案是指物品總價值最大的方案。以 01 背包爲例。

結合求最大總價值和方案總數兩個問題的思路，最優方案的總數可以這樣求：f[i][v]意義同前述，g[i][v]表示這個子問題的最優方案的總數，則在求 f[i][v]的同時求 g[i][v]的僞代碼如下：

```
for i=1..N
   for v=0..V
        f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
        g[i][v]=0
        if(f[i][v]==f[i-1][v])
            inc(g[i][v],g[i-1][v])
        if(f[i][v]==f[i-1][v-c[i]]+w[i])
            inc(g[i][v],g[i-1][v-c[i]])
```

如果你是第一次看到這樣的問題，請仔細體會上面的僞代碼。

求次優解、第 K 優解

對於求次優解、第 K 優解類的問題，如果相應的最優解問題能寫出狀態轉移方程、用動態規劃解決，那麼求次優解往往可以相同的複雜度解決，第 K 優解則比求最優解的複雜度上多一個係數 K。

其基本思想是將每個狀態都表示成有序隊列，將狀態轉移方程中的 max/min 轉化成有序隊列的合併。這裡仍然以 01 背包爲例講解一下。

首先看 01 背包求最優解的狀態轉移方程：
f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。如果要求第 K 優解，那麼狀態 f[i][v]就應該是一個大小爲 K 的數組 f[i][v][1..K]。其中 f[i][v][k]表示前 i 個物品、背包大小爲 v 時，第 k 優解的值。「f[i][v]是一個大小爲 K 的數組」這一句，熟悉 C 語言的同學可能比較好理解，或者也可以簡單地理解爲在原來的方程中加了一維。顯然 f[i][v][1..K]這 K 個數是由大到小排列的，所以我們把它認爲是一個有序隊列。

然後原方程就可以解釋爲：f[i][v]這個有序隊列是由 f[i-1][v]和 f[i-1][v-c[i]]+w[i]這兩個有序隊列合併得到的。有序隊列 f[i-1][v]即 f[i-1][v][1..K]，f[i-1][v-c[i]]+w[i]則理解爲在 f[i-1][v-c[i]][1..K]的每個數上加上 w[i]後得到的有序隊列。合併這兩個有序隊列並將結果的前 K 項儲存

到 f[i][v][1..K]中的複雜度是 O(K)。最後的答案是 f[N][V][K]。總的複雜度是 O(VNK)。

爲什麼這個方法正確呢？實際上，一個正確的狀態轉移方程的求解過程遍歷了所有可用的策略，也就覆蓋了問題的所有方案。只不過由於是求最優解，所以其它在任何一個策略上達不到最優的方案都被忽略了。如果把每個狀態表示成一個大小爲 K 的數組，並在這個數組中有序的保存該狀態可取到的前 K 個最優值。那麼，對於任兩個狀態的 max 運算等價於兩個由大到小的有序隊列的合併。

另外還要注意題目對於「第 K 優解」的定義，將策略不同但權值相同的兩個方案是看作同一個解還是不同的解。如果是前者，則維護有序隊列時要保證隊列裡的數沒有重複的。

小結

顯然，這裡不可能窮盡背包類動態規劃問題所有的問法。甚至還存在一類將背包類動態規劃問題與其它領域（例如數論、圖論）結合起來的問題，在這篇論背包問題的專文中也不會論及。但只要深刻領會前述所有類別的背包問題的思路和狀態轉移方程，遇到其它的變形問法，只要題目難度還屬於 NOIP，應該也不難想出算法。

觸類旁通、舉一反三，應該也是一個 OIer 應有的品質吧。

附錄一：USACO 中的背包問題

USACO 是 USA Computing Olympiad 的簡稱，它組織了很多面向全球的計算機競賽活動。

USACO Trainng 是一個很適合初學者的題庫，我認爲它的特色是題目質量高，循序漸進，還配有不錯的課文和題目分析。其中關於背包問題的那篇課文 (TEXT Knapsack Problems) 也值得一看。

另外，USACO Contest 是 USACO 常年組織的面向全球的競賽系列，在此也推薦 NOIP 選手參加。

我整理了 USACO Training 中涉及背包問題的題目，應該可以作爲不錯的習題。其中標加號的是我比較推薦的，標嘆號的是我認爲對 NOIP 選手比較有挑戰性的。

題目列表

- Inflate (+) （基本 01 背包）
- Stamps (+)(!) （對初學者有一定挑戰性）
- Money
- Nuggets
- Subsets
- Rockers (+) （另一類有趣的「二維」背包問題）
- Milk4 (!) （很怪的背包問題問法，較難用純 DP 求解）

題目簡解

以下文字來自我所撰的《USACO 心得》一文，該文的完整版本，包括我的程序，可在 DD 的 USACO 征程中找到。

Inflate 是加權 01 背包問題，也就是說：每種物品只有一件，只可以選擇放或者 不放；而且每種物品有對應的權值，目標是使總權值最大或最小。它最樸素的狀態 轉移方程是：f[k][i] = max{f[k-1][i] , f[k-1][i-v[k]]+w[k]}。f[k][i]表示前 k 件物品花費代 價 i 可以得到的最大權值。v[k]和 w[k]分別是第 k 件物品的花費和權值。可以看到， f[k]的求解過程就是使用第 k 件物品對 f[k-1]進行更新的過程。那麼事實上就不用使用 二維數組，只需要定義 f[i]，然後對於每件物品 k，順序地檢

查 f[i]與 f[i-v[k]]+w[k]的大 小，如果後者更大，就對前者進行更新。這是背包問題中典型的優化方法。

題目 stamps 中，每種物品的使用量沒有直接限制，但使用物品的總量有限制。 求第一個不能用這有限個物品組成的背包的大小。（可以這樣等價地認爲）設 f[k][i] 表示前 k 件物品組成大小爲 i 的背包， 最少需要物品的數量。則 f[k][i]= min{f[k-1][i],f[k-1][i-j*s[k]]+j}，其中 j 是選擇使用第 k 件物品的數目，這個方程運用時 可以用和上面一樣的方法處理成一維的。求解時先設置一個粗糙的循環上限，即最 大的物品乘最多物品數。

Money 是多重背包問題。也就是每個物品可以使用無限多次。要求解的是構成 一種背包的不同方案總數。基本上就是把一般的多重背包的方程中的 min 改成 sum 就行了。

Nuggets 的模型也是多重背包。要求求解所給的物品不能恰好放入的背包大小 的最大值（可能不存在）。只需要根據「若 i、j 互質，則關於 x、y 的不定方程 i*x+y*j=n 必有正整數解，其中 n>i*j」這一定理得出一個循環的上限。 Subsets 子集和問題相當於物品大小是前 N 個自然數時求大小爲 N*(N+1)/4 的 01 背包的方案數。

Rockers 可以利用求解背包問題的思想設計解法。我的狀態轉移方程如下： f[i][j][t]=max{f[i][j][t-1] , f[i-1][j][t] , f[i-1][j][t-time[i]]+1 , f[i-1][j-1][T]+(t>=time[i])}。其中 f[i][j][t]表示前 i 首歌用 j 張完整的盤和一張錄了 t 分鐘的盤可以放入的最多歌數，T 是 一張光盤的最大容量，t>=time[i]是一個 bool 值轉換成 int 取值爲 0 或 1。但我後來發 現我當時設計的狀態和方程效率有點低，如果換成這樣：f[i][j]=(a,b)表示前 i 首歌中 選了 j 首需要用到 a 張完整的光盤以及一張錄了 b 分鐘的光盤，會將時空複雜度都大 大降低。這種將狀態的值設爲二維的方法值得注意。

Milk4 是這些類背包問題中難度最大的一道了。很多人無法做到將它用純 DP 方 法求解，而是用迭代加深搜索枚舉使用的桶，將其轉換成多重背包問題再 DP。由於 USACO 的數據弱，迭代加深的深度很小，這樣也可以 AC，但我們還是可以用純 DP 方法將它完美解決的。設 f[k]爲稱量出 k 單位牛奶需要的最少的桶數。那麼可以用類 似多重背包的方法對 f 數組反覆更新以求得最小值。然而困難在於如何輸出字典序最 小的方案。我們可以對每個 i 記錄 pre_f[i]和 pre_v[i]。表示得到 i 單位牛奶的過程是 用 pre_f[i]單位牛奶加上若干個編號爲 pre_v[i]的桶的牛奶。這樣就可以一步步求得得 到 i 單位牛奶的完整方案。爲了使方案的字典序最小，我們在每次找到一個耗費桶數 相同的方案時

對已儲存的方案和新方案進行比較再決定是否更新方案。爲了使這種比較快捷，在使用各種大小的桶對 f 數組進行更新時先大後小地進行。USACO 的官 方題解正是這一思路。如果認爲以上文字比較難理解可以閱讀官方程序或我的程序。

附錄二：P11: 背包問題的搜索解法

《背包問題九講》的本意是將背包問題作爲動態規劃問題中的一類進行講解。但鑑於的確有一些背包問題只能用搜索來解，所以這裡也對用搜索解背包問題做簡單介紹。大部分以 01 背包爲例，其它的應該可以觸類旁通。

簡單的深搜

對於 01 背包問題，簡單的深搜的複雜度是 O(2^N)。就是枚舉出所有 2^N 種將物品放入背包的方案，然後找最優解。基本框架如下：

```
procedure SearchPack(i,cur_v,cur_w)
    if(i>N)
        if(cur_w>best)
            best=cur_w
        return
    if(cur_v+v[i]<=V)
        SearchPack(i+1,cur_v+v[i],cur_w+w[i])
    SearchPack(i+1,cur_v,cur_w)
```

其中 cur_v 和 cur_w 表示當前解的費用和權值。主程序中調用 SearchPack(1,0,0)即可。

搜索的剪枝

基本的剪枝方法不外乎可行性剪枝或最優性剪枝。

可行性剪枝即判斷按照當前的搜索路徑搜下去能否找到一個可行解，例如：若將剩下所有物品都放入背包仍然無法將背包充滿（設題目要求必須將背包充滿），則剪枝。

最優性剪枝即判斷按照當前的搜索路徑搜下去能否找到一個最優解，例如：若加上剩下所有物品的權值也無法得到比當前得到的最優解更優的解，則剪枝。

搜索的順序

在搜索中，可以認爲順序靠前的物品會被優先考慮。所以利用貪心的思想，將更有可能出現在結果中的物品的順序提前，可以較快地得出貪心地較優解，更有利於最優性剪枝。所以，可以考慮將按照「性價比」（權值/費用）來排列搜索順序。

另一方面，若將費用較大的物品排列在前面，可以較快地填滿背包，有利於可行性剪枝。

最後一種可以考慮的方案是：在開始搜索前將輸入文件中給定的物品的順序隨機打亂。這樣可以避免命題人故意設置的陷阱。

以上三種決定搜索順序的方法很難說哪種更好，事實上每種方法都有適用的題目和數據，也有可能將它們在某種程度上混合使用。

子集和問題

子集和問題是一個 NP-Complete 問題，與前述的（加權的）01 背包問題並不相同。給定一個整數的集合 S 和一個整數 X，問是否存在 S 的一個子集滿足其中所有元素的和爲 X。

這個問題有一個時間複雜度爲 O(2^(N/2))的較高效的搜索算法，其中 N 是集合 S 的大小。

第一步思想是二分。將集合 S 劃分成兩個子集 S1 和 S2，它們的大小都是 N/2。對於 S1 和 S2，分別枚舉出它們所有的 2^(N/2)個子集和，保存到某種支持查找的數據結構中，例如 hash set。

然後就要將兩部分結果合併，尋找是否有和爲 X 的 S 的子集。事實上，對於 S1 的某個和爲 X1 的子集，只需尋找 S2 是否有和爲 X-X1 的子集。

假設採用的 hash set 是理想的，每次查找和插入都僅花費 O(1)的時間。兩步的時間複雜度顯然都是 O(2^(N/2))。

實踐中，往往可以先將第一步得到的兩組子集和分別排序，然後再用兩個指針掃瞄的方法查找是否有滿足要求的子集和。這樣的實現，在可接受的時間內可以解決的最大規模約爲 N=42。

搜索還是 DP?

在看到一道背包問題時，應該用搜索還是動態規劃呢？

首先，可以從數據範圍中得到命題人意圖的線索。如果一個背包問題可以用 DP 解，V 一定不能很大，否則 O(VN)的算法無法承受，而一般的搜索解法都是僅與 N 有關，與 V 無關的。所以，V 很大時（例如上百萬），命題人的意圖就應該是考察搜索。另一方面，N 較大時（例如上百），命題人的意圖就很有可能是考察動態規劃了。

另外，當想不出合適的動態規划算法時，就只能用搜索了。例如看到一個從未見過的背包中物品的限制條件，無法想出 DP 的方程，只好寫搜索以謀求一定的分數了。

聯繫方式

如果有任何意見和建議，特別是文章的錯誤和不足，或者希望爲文章添加新的材料，可以通過 http://kontactr.com/user/tianyi/這個網頁聯繫我。

值得說明的是，如果有 OI 方面的問題，例如不明白自己的程序爲什麼錯了或者索要某種算法的源代碼，使用這個聯繫方式可能得不到及時解答。請在 OIBH 論壇發問。

致謝

感謝以下名單：

- 阿坦
- jason911
- donglixp
- LeafDuo

他們每人都最先指出了本文曾經存在的某個並非無關緊要的錯誤。謝謝你們如此仔細地閱讀拙作並彌補我的疏漏。

感謝 XiaQ，它針對本文的第一個 beta 版發表了用詞嚴厲的六條建議，雖然我只認同並採納了其中的兩條。在所有讀者幾乎一邊倒的讚揚將我包圍的當時，你的貼子是我的一劑清醒劑，讓我能清醒起來並用更嚴厲的眼光審視自己的作品。

sfita 提供了 P01 中的「一個常數優化」。

當然，還有用各種方式對我表示鼓勵和支持的幾乎無法計數的同學。不管是當面讚揚，或是在論壇上回覆我的貼子，不管是發來熱情洋溢的郵件，或是在即時聊天的窗口裡豎起大拇指，你們的鼓勵和支持是支撐我的寫作計劃的強大動力，也鞭策著我不斷提高自身水平，謝謝你們！

最後，感謝 Emacs 這一世界最強大的編輯器的所有貢獻者，感謝它的插件 EmacsMuse 的開發者們，本文的所有編輯工作都借助這兩個卓越的自由軟件完成。謝謝你們——自由軟件社群——爲社會提供了如此有生產力的工具。我深深欽佩你們身上體現出的自由軟件的精神，沒

有你們的感召，我不能完成本文。在你們的影響下，採用自由文檔的方式發佈本文檔，也是我對自由社會事業的微薄努力。
