4 貪婪法

章節大綱

4.1　何謂貪婪法？

4.2　最小成本生成樹

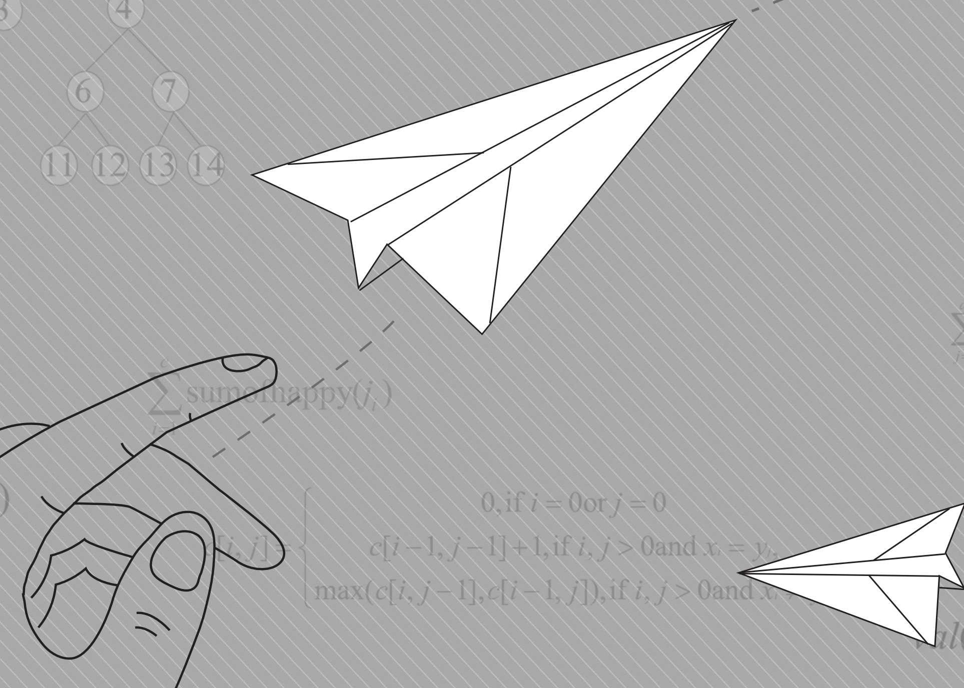
4.3　霍夫曼編碼樹

4.4　貪婪法的陷阱：0-1 背包問題

4.5　單位時間工作排程問題

4.6　如何證明貪婪法是正確的?
簡介 Matroid 理論

4.7　貪婪法的技巧



4.1 何謂貪婪法？

「什麼是貪婪法 (greedy method)？」

簡言之：「重複地(或貪婪地)根據一個法則來挑選解的一部份。當挑選完畢時，最佳解也出現了。」

「印度有一位農夫在他田地的河邊撿到一顆很漂亮的石頭。他將石頭帶回家給小孩玩，小孩玩膩後，就將那顆石頭隨手丟到雜物堆的角落。有一天，一位珠寶商路過他家，告訴農夫在此附近有一條河，河裡盛產鑽石。農夫心想，種了一輩子的田太辛苦又賺不到錢，就決心把農地賣掉，去尋找那條產鑽石的河。農夫找了好多年，無功而返。有天閒來無事整理家裡時，在雜物堆裡發現了那顆以前在河邊撿來的石頭，這才發現那竟是顆價值連城的大鑽石。而他多年來苦心尋找的鑽石河，正好位於他賣掉的田地內。」

前述農夫尋找寶石的故事暗示：眼前的東西有時是最珍貴的。貪婪法一旦使用得當，會是一個有效率的策略。雖然，也許貪婪法無法解決所有問題。

4.2 最小成本生成樹

第一個例子是一個使用最小成本佈建網路的問題。

表 4.1 最小成本網路佈建問題

問題	有一位網路工程師替一家公司規劃佈建一個網路。他想將所有網路設備連接成一個完全連通的網路。但是，希望所使用的網路成本可以降到最低。請替他設計一個演算法解決此問題
輸入	一個網路的可能建置圖(graph) $G=(V, E)$，其中點(vertex)代表網路設備，點和點的線(edge)代表兩網路設備可以連通。線上的整數，代表連接此兩個網路設備所需的佈建成本
輸出	將整網路設備連接成一個連通的網路時，所需最小成本的連接方式

「這個問題是在輸入的圖(graph)中，找怎樣的答案？」

「應該是尋找此圖的一部分，即其子圖(sub-graph)。」

「任意的子圖都可以嗎？」

「不！需要將所有的點連接在一起的子圖。」

「還有其他的條件嗎？」

「還有這個子圖的成本總和(sum)必須最小。」

以上描述了***最小成本生成樹***(minimum cost spanning tree)問題：在一個有權重(weight)的圖中，找尋一個子圖(即此圖的子集合)符合下列條件：

(1) 連接在一起， [此為樹(tree)的性質]

(2) 經過每一個點， [此為生成(spanning)的性質]

(3) 線的成本總和最小。 [此為最小成本(minimum cost)的性質]

如圖 4.1 中，粗線的子圖為整個圖的其中一個最小成本生成樹。

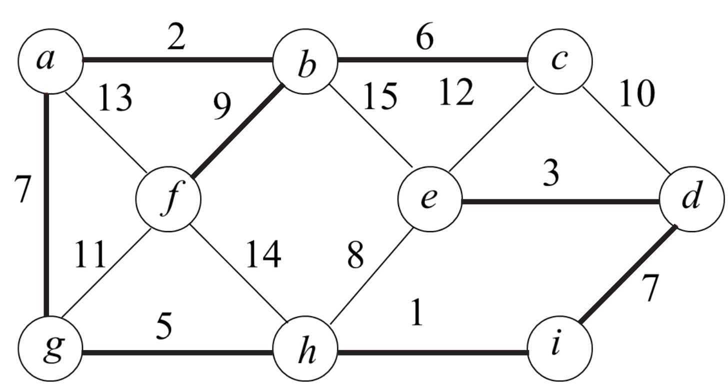


圖 4.1 一個最小成本生成樹(即粗線的子圖)

在任意一連接圖(connected graph)上，找其最小成本生成樹後，會發現此樹不含迴圈；而且任意地額外加入一條新的線於此樹上，會產生一個唯一的迴圈。如在圖 4.1 加入線(e, h)即產生迴圈{dehi}。相反地，一旦自這一棵生成樹上，任意地刪除一條樹上的線，也會造成整棵樹不連接。用建構一個連接網路的問題來做比喻，每一棵最小成本生成樹是將網路連在一起的最精簡(最省線材)的方式。

「如何在一個圖中，找到一棵最小成本生成樹？」

「嗯！不知道。」

「觀察最小成本生成樹的例子(如圖 4.1)，試著找這些最佳解答的特點？」

「一棵生成樹是 通過每一個點 且 擁有連接、成本最小及無迴圈的特點。」

「利用這些特色可以找到最小成本生成樹嗎？」

「嗯！好像不行。」

「試著比較 落在最小成本生成樹中的線 和 不在其中的線 的差別？」

「看不出來有何差別。除了，任意地額外加入一條樹外的線在生成樹上會產生一個唯一的迴圈」

「注視此迴圈，試著找出 生成樹上的線 和 新加入的線 之間有何差別？」

「似乎，在此迴圈中，新加入(即不在最小成本生成樹上)線的成本，都比迴圈中其他(即在最小成本生成樹上)的線大。」

「這個發現有助於找到一棵最小成本生成樹？」

「也許可以先不要考慮成本較大的線，或按照線的成本由小到大的順序來建構一個連接網路。」

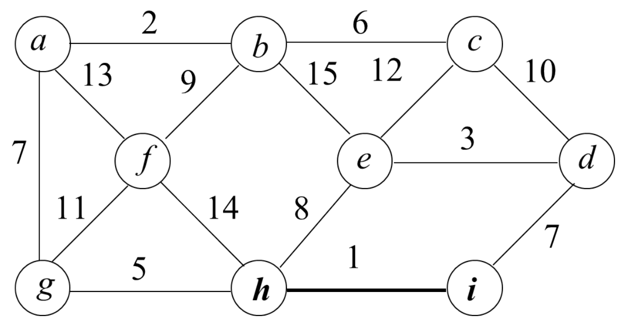
「試幾個例子看看，可否找到一棵最小成本生成樹？」

「有些例子在連接的過程中，會發生迴圈。這種狀況需要避免，因為目標是要找到一棵樹，而樹是無迴圈的。」

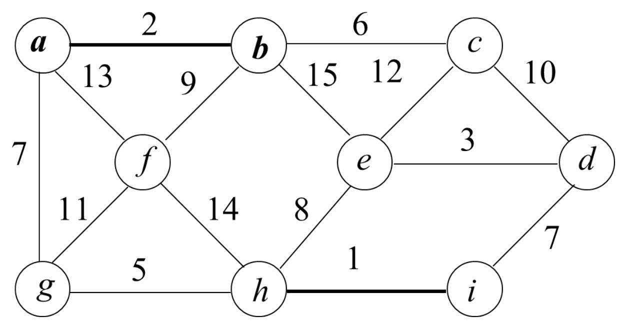
「目前您有什麼想法？」

「按照線的成本由小到大的順序來連接整個網路，同時需要避免發生迴圈。」

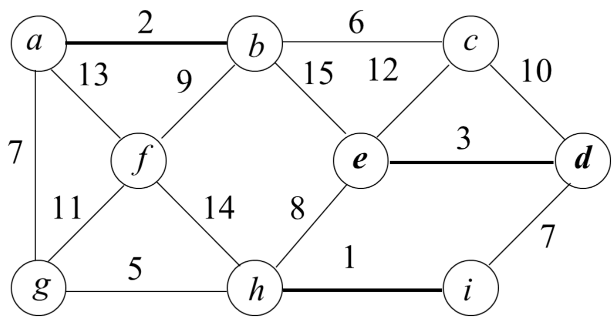
Kruskal 的最小成本生成樹演算法就是：「按照線的成本，由小到大依序地選擇線來連接整個網路，但是在此過程中，需避免發生迴圈」。以圖 4.1 為例，此演算法的執行過程如下。



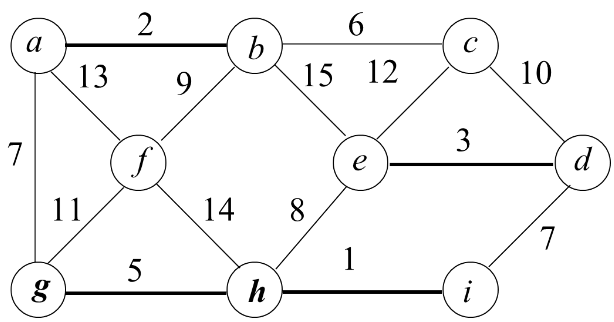
選取線 (*h*, *i*)



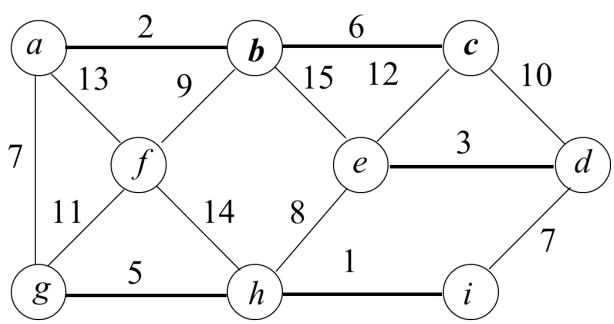
選取線 (*a*, *b*)



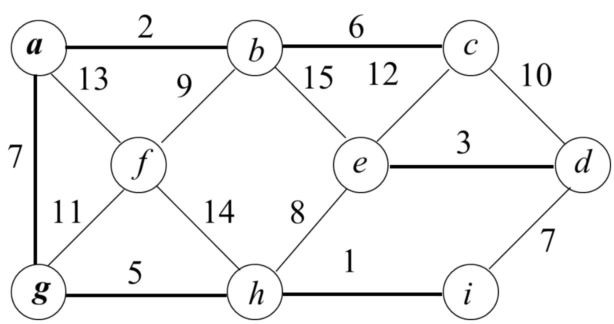
選取線(*e*, *d*)



選取線 (*g*, *h*)



選取線 (*b*, *c*)



選取線 (*a*, *g*)

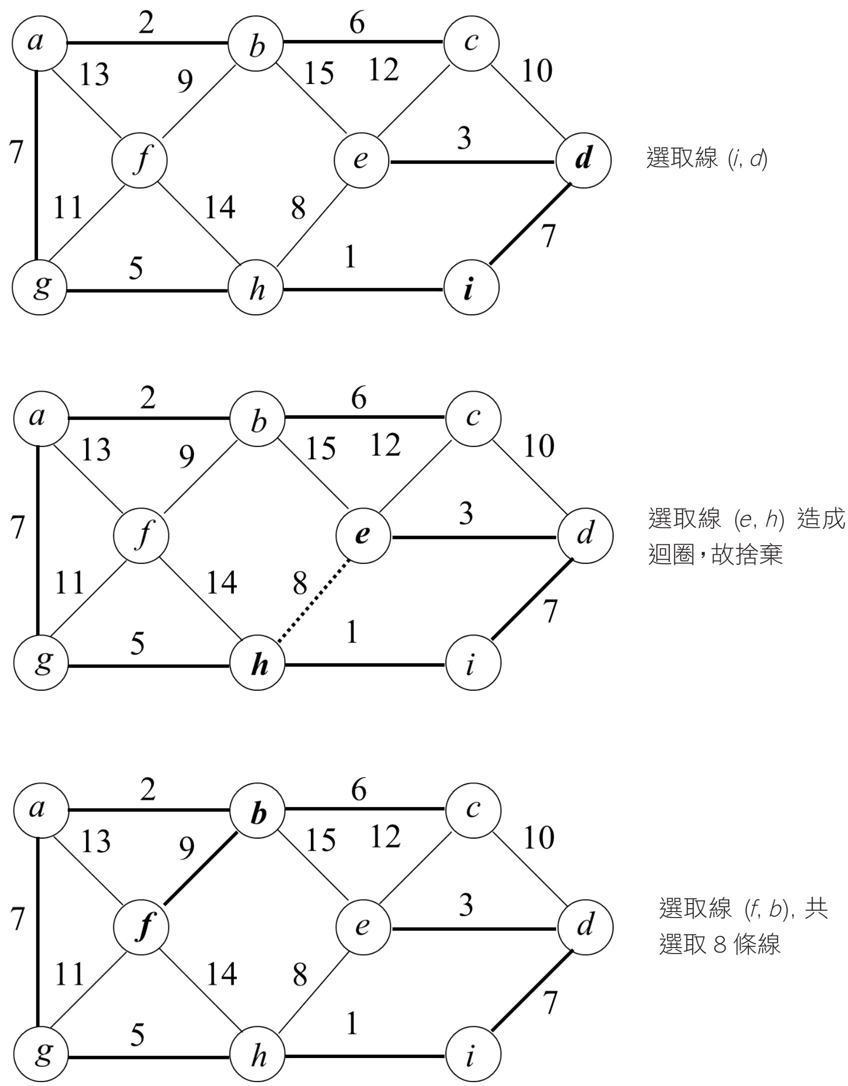


圖 4.2 Kruskal 最小成本生成樹演算法的執行範例

詳細的 Kruskal 最小成本生成樹演算法列於表 4.2 中。

表 4.2 Kruskal 最小成本生成樹的演算法

輸入	一連接圖 $G=(V, E)$，其中 V 為點集合而 E 為有權重的線集合
輸出	圖 G 的最小成本生成樹 T

步驟	Algorithm MST-Kruskal
	{
	Step 1: 令 T 為空集合，並令 V 中的每一個點 v 為一個集合。
	Step 2: 將 E 中的線依其成本(權重)由小到大排列。
	Step 3: 依照由小到大的成本，自 E 中選取一線(u, v)，並執行下列指令:
	{if $T \cup \{(u, v)\}$不會形成迴圈 Then 將(u, v)加入 T 中。}直到選中$\|V\|-1$條線為止。
	}

Kruskal 的最小成本生成樹演算法，還有一個步驟需要進一步討論。請看下面的圖例及對話。

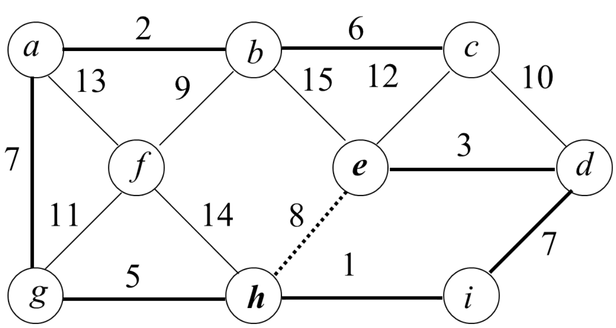


圖 4.3 當線(e, h)加入時會造成迴圈

「當一線(e, h)加入時，如何判斷 $T \cup \{(e, h)\}$會不會形成迴圈?」「試著觀察一個例子。」

「好像是判斷兩個欲選的線之兩端點，是否已經相連了。」

「如何判斷兩端點已經相連了?」

「只要檢查兩端點是否已經有一條路連通即可。」

「如何檢查兩端點是否有一條路連通?」

「嗯…」

「所有連通的點有何共同性質?」

「所有連通的點彼此互相連接。」

「如何記錄這種關係?」

「將所有相連的點可用一個集合來記錄。」

「這個關係何時會被改變?」

「當加入的新線連接兩個不同的集合時。」

「此時關係做怎樣的改變?」

「此兩個原來不同的集合，因為此新加入的線，導致此兩集合的所有點都連接在一起，因此可以將此兩集合聯集成一個大的集合。」

「如何儲存一個集合?」

「可用矩陣。」

「還有更好的方式嗎?」

「嗯…」

「為何選中 $|V|$-1 條線後，此演算法即可停止?」

「因為已經找到一棵生成樹了。」

「$|V|$ 個點的生成樹有多少的線?」

「好像是 $|V|$-1。」

「這個演算法是對的嗎?其時間複雜度為多少?」

「嗯…」

Kruskal 的最小成本生成樹演算法的時間複雜度為 $O(|E|log|E|)$。注意 Step 3 需應用較有效率的集合聯集(union)及查詢(find)的資料結構。

表 4.3 Kruskal 的最小成本生成樹演算法的時間複雜度

指令	執行次數
Step 1: 令 T 為空集合，並令 V 中的每一個點 v 為一個集合。	$O(\|V\|)$
Step 2: 將 E 中的線依其成本由小到大排列。	$O(\|E\| \log \|E\|)$
Step 3: 依照由小到大的成本，自 E 中選取一線 (u, v)，並執行下列指令: {if $T \cup \{(u,v)\}$ 不會形成迴圈 Then 將 (u, v) 加入 T 中。}直到選中 $\|V\|$-1 條線為止。	$O(\|E\|)$
時間複雜度	$O(\|E\| \log \|E\|)$

4.3 霍夫曼編碼樹

若將一份文字檔先進行資料壓縮(data compression)後，再於網路上傳送，可以減少傳輸時間及成本。編碼樹利用一棵二元樹(binary tree)表示編碼的方法。如下圖，每個被編碼的符號，被置於此二元樹的樹葉(leaf)上，而且樹上的每一個線上被標上一個位元的 0(向左的線)或 1(向右的線)。

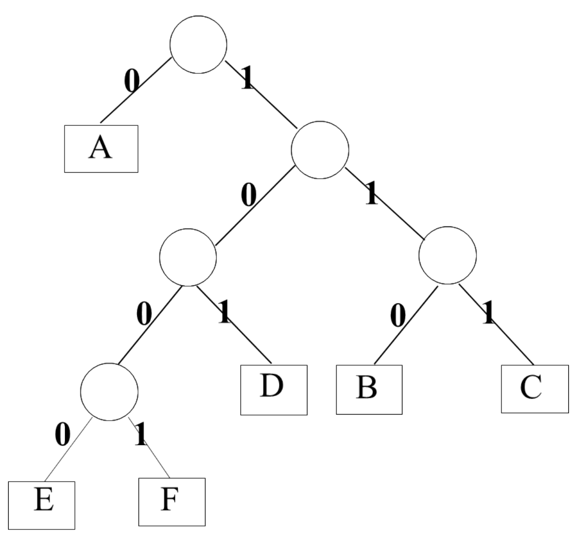
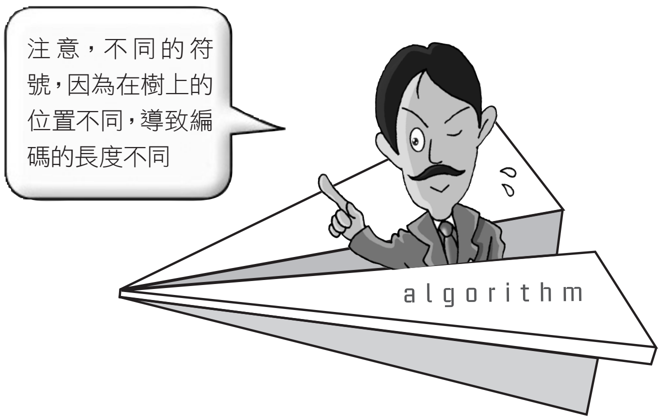


圖 4.4 編碼樹

在一棵編碼樹上，自樹根(root)走到一個特定的樹葉(leaf)，會形成一條唯一的路徑。收集此路徑線上的 0 與 1 字串，即是此樹葉上的符號所對應到的編碼。例如，圖 4.4 中的 F 即對應到 1001，而 D 對應到 101。



不同的編碼樹會有不同的編碼方式，也會產生不同的資料壓縮效果。如果知道每個符號在傳送資料中出現的次數，如何建造出一棵最佳的編碼樹將傳送的資料壓縮成最少的位元，就成為一個值得探討的問題。霍夫曼編碼(Huffman code)即是其中的一個方法。

表 4.4 建造最佳的編碼樹

問題	已知每個符號在傳送資料中出現的次數。請設計一個程式找出一棵最佳的編碼樹，使得利用此樹的編碼方式，可以將傳送的資料壓縮成最少的位元
輸入	每個符號在傳送資料中出現的次數
輸出	可將傳送的資料壓縮成最少位元的一棵最佳的編碼樹

「編碼樹的資料壓縮效果和什麼有關係?」

「應該和符號出現的次數及符號編碼有關。」

「符號出現的次數，對此符號在最佳的編碼樹上出現的位置，有何影響?」

「嗯…」

「出現最多次數的符號，應該放在編碼樹之何處?」

「當然是，越靠近樹根(root)越好囉！因為如此對應到位元數總和會越少。」

「出現最少次數的符號，應該放在編碼樹之何處?」

「越靠近樹葉(leaf)越好，如此可把靠近樹根的位置，留給出現較多次數的符號。」

「依此想法，如何建造一棵最佳的編碼樹?」

「根據符號出現的頻率，來安排出現在樹上的位置：也就是，在樹上層放置常出現的符號，在樹下層放置少出現的符號。」

「一開始要怎麼做?」

「也許可以，將最少出現的兩個符號，放在樹的最下層。」

霍夫曼編碼樹，就是貪婪地以「出現最少次數的兩個符號，放在樹的最下層」的方法，所建造出的一棵最佳的編碼樹。以下利用一個範例來說明。

Step 1 一開始整個集合包含所有符號，將每個符號的權重設定成其出現的次數。並且依照權重由小到大排列。

E: 1	F:2	D:5	B:6	C:7	A:14

圖 4.5 建造霍夫曼編碼樹演算法的過程之 1

Step 2 將集合中出現最小權重的兩個符號 E(權重 1) 及 F(權重 2) 取出，合併成一棵二元樹(binary tree)後，將此樹的權重設為 E 及 F 權重之和(即 3=1+2)。並依照權重由小到大插入原順序中。

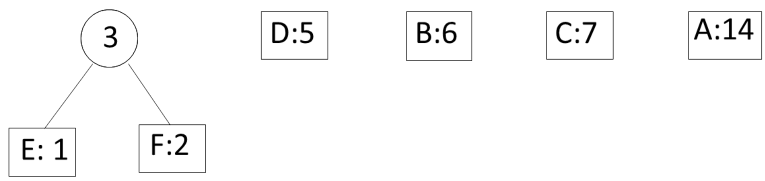


圖 4.6 建造霍夫曼編碼樹演算法的過程之 2

Step 3 再將集合中出現最小權重的樹 (權重 3) 及 D(權重 5) 取出，合併成一棵二元樹。將此新樹的權重設為 8(=3+5)，並依照權重由小到大插入原順序中。

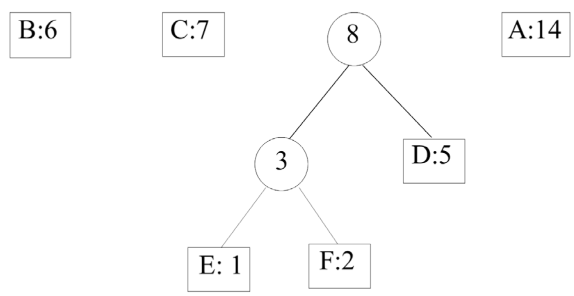


圖 4.7 建造霍夫曼編碼樹演算法的過程之 3

Step 4 接下來，將集合中出現最小權重的 B(權重 6) 及 C(權重 7) 取出，合併成一棵樹後，並將此新樹的權重設為 13(=6+7)，再次置回原順序中。

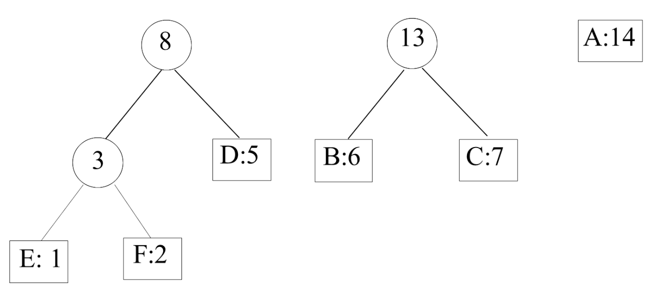


圖 4.8 造霍夫曼編碼樹演算法的過程之 4

Step 5 再將集合中出現最小權重的樹 (權重 8) 及 (權重 13) 取出，合併成一棵樹 (其權重設為 21) 後，依次置回原集合。

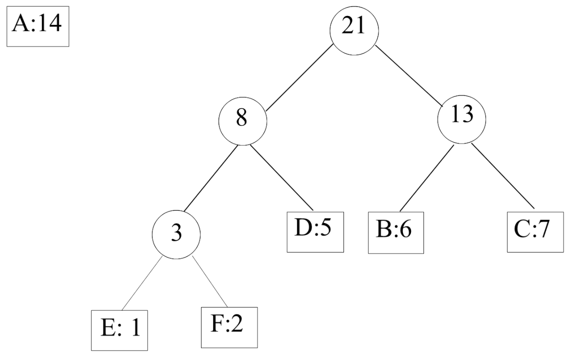


圖 4.9 建造霍夫曼編碼樹演算法的過程之 5

Step 6 最後將集合中剩下的兩個，合併成一棵樹後，置回原集合。此新樹的權重為 35(=14+21)。

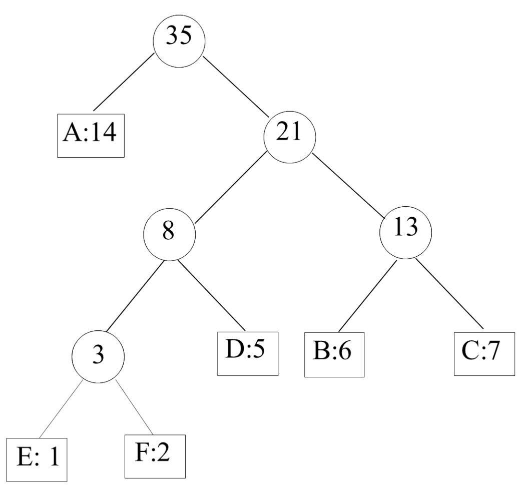


圖 4.10 建造霍夫曼編碼樹演算法的建立結果

建造霍夫曼編碼樹的貪婪演算法如下表所示。

表 4.5 建造霍夫曼編碼樹的演算法

輸入	*n* 個符號及其權重(每個符號的出現次數)
輸出	一棵最佳的編碼樹 *T*

步驟	Algorithm Huffman_code
	{
	Step 1: 將每個 *n* 個符號依其權重進行由小到大排列。
	Step 2: 執行以下步驟 *n*-1 次。
	    Step 2.1: 找到目前集合中最小權重的元素 *A*，並自集合中刪去元素 *A*。
	    Step 2.2: 找到目前集合中最小權重的元素 *B*，並自集合中刪去元素 *B*。
	    Step 2.3: 將 *A* 及 *B* 設定為一新節點 *C* 的左、右兒子。將新樹加入原排列中，並將其權重設為兩樹權重之和。
	}

表 4.6 建造霍夫曼編碼樹的時間複雜度分析

指令	執行次數
Step 1：	$O(n \log n)$
Step 2：執行以下步驟 n-1 次	n-1
Step 2.1	$O(1)$
Step 2.2	$O(1)$
Step 2.3	$O(\log n)$
時間複雜度	$O(n \log n)$

4.4 貪婪法的陷阱：0-1 背包問題

當貪婪法使用得當(如前兩節所述)，會是一個有效率的策略。但是，目前並非所有問題都可用貪婪法解決。下列介紹的 0-1 背包問題(0-1 Knapsack problem)就是一個例子。

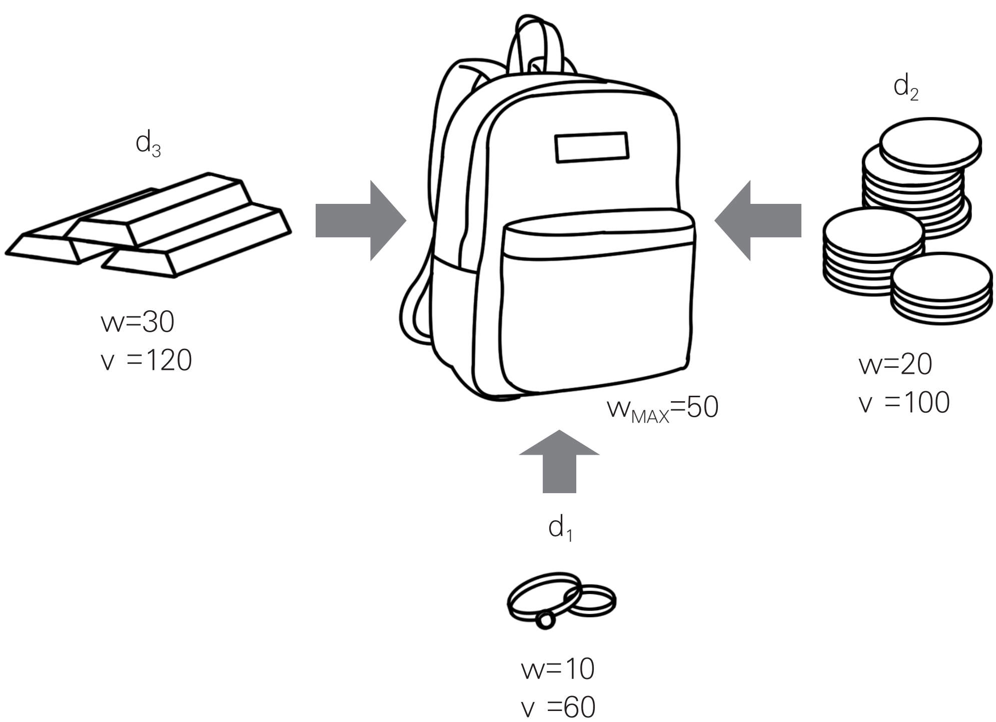
表 4.7 0-1 背包問題

問題	有一位小偷闖入一戶人家，看見一些值錢及不值錢的家當。他想把值錢的東西都帶走，但是裝東西的袋子有重量的限制。請問，在不超過重量的限制之下，他該帶走那些家當使得總價值最高? 注意，每件家當需整個被取走或整個留下，如同只能從 0 及 1 中作選擇一般。
輸入	(1) n 個家當$\{d_1, d_2 \cdots, d_n\}$及 d_i 對應的價值 v_i 及重量 w_i (2) 袋子的重量限制 W
輸出	一個家當清單，使得清單內的物件的總重量和小於或等於 W，且其價值的和為最大

直覺上，貪婪地挑目前最有價值且輕的家當，會有最佳解；也就是，由大到小依照{價值/重量}的值來挑。表 4.8 是一個例子。

表 4.8 0-1 背包問題的一個範例

	d_1	d_2	d_3
重量 w_i	10	20	30
價值 v_i	60	100	120
價值/重量	6	5	4



令袋子的重量限制 W=50。若由大到小依照{價值/重量}的值來挑，則選入袋子的家當為 d_1 及 d_2 而總價值為 60+100=160。注意此刻 d_3 不可同時納入，否則總重量為 60，就超過袋子的負荷(W=50)。但可惜的是，此解並非最佳。若取 d_2 及 d_3 則總價值為 100+120=220 且總重量為 20+30=50，並未超過袋子的負荷。

由上例可知，利用貪婪法未能成功地解答 0-1 背包問題。當每件家當可任意被取走一部份時(如取走 35% 的家當 d_1)，此種的背包問題被稱為***部份背包問題***(fractional knapsack problem)。有趣的是，部份背包問題是可以利用上述的貪婪法找到最佳解。注意，當最後取一完整家當會超過袋子的負荷時，此法只取其一部份，使得整個袋子裝滿即可。

4.5 單位時間工作排程問題

目前知道貪婪法可解的問題有最小成本生成樹、建造最佳的編碼樹，而貪婪法尚未成功解決的問題有 0-1 背包問題。本節再介紹一個貪婪法可解的問題，也就是單位時間工作排程問題，以突顯判斷一個問題是否可用貪婪法來解，是一個極需思考的議題。

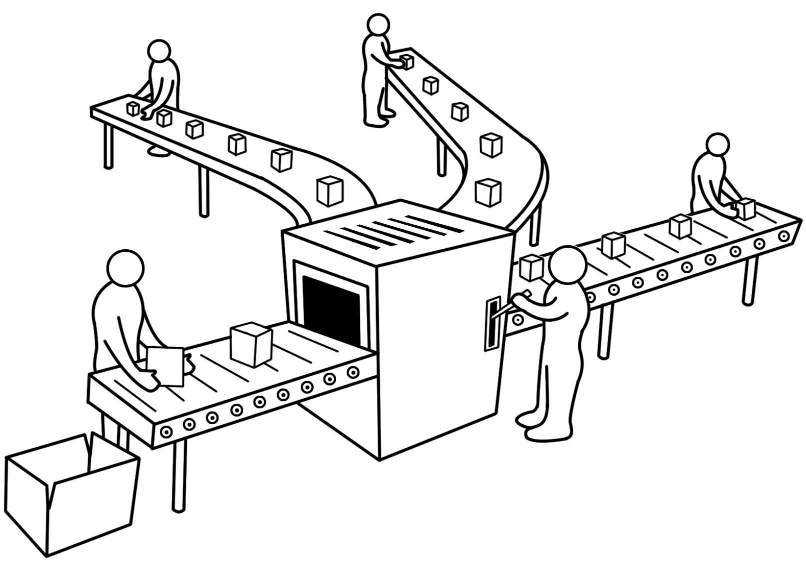


表 4.9 單位時間工作排程問題

<table>
<tr><td>問題</td><td>老王的工廠只有一台機器但需完成的工作有 n 件。每份工作需佔用這台機器一日工作時間。每份工作都有完成的期限(deadline)，一旦未在期限前完成需繳納罰金(penalty)。請幫老王寫個程式完成工作排程，使他繳納最少的罰金</td></tr>
<tr><td>輸入</td><td>n 工作$\{J_1, J_2, \cdots, J_n\}$及 J_i 對應的完成期限為 D_i 及未完成時需繳的罰金為 p_i
<table>
<tr><td>工作編號</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>期限</td><td>4</td><td>2</td><td>4</td><td>3</td><td>1</td><td>4</td><td>6</td></tr>
<tr><td>罰金</td><td>70</td><td>60</td><td>50</td><td>40</td><td>30</td><td>20</td><td>10</td></tr>
</table>
</td></tr>
<tr><td>輸出</td><td>找出需繳納罰金最少工作排程
<table>
<tr><td>工作編號</td><td>2</td><td>4</td><td>1</td><td>3</td><td>7</td><td>5*</td><td>6*</td></tr>
<tr><td>排程日期</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>期限</td><td>2</td><td>3</td><td>4</td><td>4</td><td>6</td><td>1</td><td>4</td></tr>
<tr><td>罰金</td><td>60</td><td>40</td><td>70</td><td>50</td><td>10</td><td>30*</td><td>20*</td></tr>
</table>
＊代表超過期限的工作及所付罰金
罰金=30(工作 5)+20(工作 6)=50</td></tr>
</table>

首先，我們透過下列的討論，希望可以構思這個問題的解決方法。

「若要繳納最少罰金，怎樣的工作需要先排?」

「應該是期限早到期的工作或罰金較多的工作。」

「這兩類的工作，那一種需要先排?」

「嗯…」

「如果期限早到期的工作先排，會有什麼缺點?」

「應該是，當罰金較多的工作想要排入，卻被期限早到期的工作先佔用了機器。」

「相反的，如果罰金較多的工作先排，會有什麼缺點?」

「應該是，當期限早到期的工作想要排入，卻被罰金較多的工作先佔用了機器。」

「回到原來的問題，這兩類的工作，那一種需要先排?」

「嗯…」

「排程的目的何在?」

「找到繳納罰金最少的排程。」

「那一種排程最有可能繳較少罰金?」

「嗯…」

「每一種排程下，被處罰的工作其罰金的情況如何?」

「我想一下。第一種排程，當罰金較多的工作想要排入，卻被期限早到期的工作先佔用了機器，罰的是罰金較多的工作; 相對地，第二種排程，當期限早到期的工作想要排入，卻被罰金較多的工作先佔用了機器，罰的是期限早到期的工作。」

「那一種排程最有可能繳較少罰金?」

「嗯，也許應先不用第一種排程。犯不著先承受罰金較多的處罰，使用第二種排程的話，罰的是期限早到期的工作，也許繳的罰金會比較少。」

將罰金較多的工作先排入的方法，可以設計出一個貪婪演算法。以下利用一個範例來說明。

Step 1　將所有工作按照罰金的大小排列好。

工作編號	1	2	3	4	5	6	7
罰金	70	60	50	40	30	20	10

Step 2　依照 Step 1 所得的順序，將每件工作一一利用下列步驟判斷是否加入目前的排程中。

Step 2.1　將工作 1 排入行程。

工作編號	1						
排程日期	1						
期限	4						
罰金	70						

Step 2.2　試著將工作 2 排入行程時，因其期限為 2(較工作 1 的期限 4 早)，故將工作 2 排於工作 1 之前。

工作編號	2	1					
排程日期	1	2					
期限	2	4					
罰金	60	70					

Step 2.3　工作 3 及工作 4 排入行程時，也依其期限先後排入。

工作編號	2	4	1	3			
排程日期	1	2	3	4			
期限	2	3	4	4			
罰金	60	40	70	50			

Step 2.4 若要將工作 5 排入行程時，因其期限為 1，需將前面的工作 2，4，1 及 3 的排程日期都向後退一日。但如此會造成工作 3 超過期限。如此工作 5 則先不排入(預定被罰)。

工作編號	5*	2	4	1	3		
排程日期	1	2	3	4	5		
期限	1	2	3	4	4		
罰金	30*	60	40	70	50		

Step 2.5 若要將工作 6 排入行程時，因其期限為 4，可排在工作 3 之後。但如此會造成工作 6 超過期限。如此工作 6 也暫時不排入(預定被罰)。

工作編號	2	4	1	3	6*		
排程日期	1	2	3	4	5		
期限	2	3	4	4	4		
罰金	60	40	70	50	20*		

Step 2.6 工作 7 順利排在工作 3 之後。

工作編號	2	4	1	3	7		
排程日期	1	2	3	4	5		
期限	2	3	4	4	6		
罰金	60	40	70	50	10		

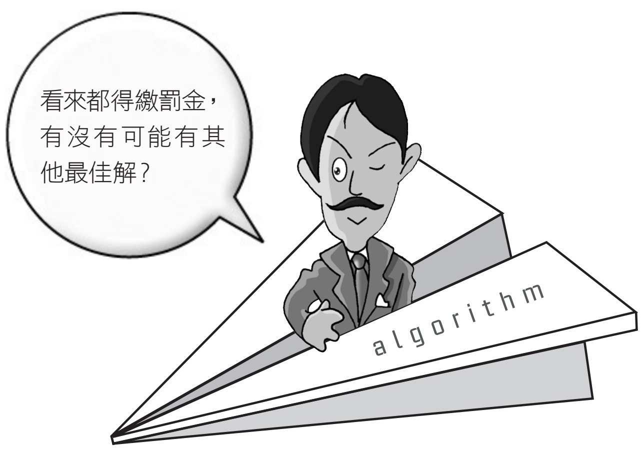
Step 2.7 最後將先前未排的工作 5 及 6 任意排入行程，並接受處罰。

工作編號	2	4	1	3	7	5*	6*
排程日期	1	2	3	4	5	6	7
期限	2	3	4	4	6	1	4
罰金	60	40	70	50	10	30*	20*

表 4.10 單位時間工作排程問題的貪婪演算法

輸入	n 個工作$\{J_1, J_2, \cdots, J_n\}$及 J_i 對應的完成期限 D_i 及未完成時需繳的罰金 P_i
輸出	找出需繳納罰金最少工作排程

步驟	Algorithm Unit_Time_Job_Scheduling
	{
	Step 1: 依照罰金由大到小，將所有工作排序：$\{J_1, J_2, \cdots, J_n\}$。
	Step 2: J_1 納入排程 $S=\{J_1\}$。
	Step 3: for i=2 to n do
	{
	if 所有的工作 $S \cup \{J_i\}$ 因為 J_i 的加入後，都還可以在期限前完成，
	then $S = S \cup \{J_i\}$ (上述的判斷可將工作 $S \cup \{J_i\}$ 依照其期限由早到晚排列，並檢查是否有任何工作超出期限)。
	}
	Step 4: 將在 Step 3 未被排入的工作，任意地安排在排程的後面。
	}



4.6 如何證明貪婪法是正確的？簡介 Matroid 理論

「單位時間工作排程問題的貪婪演算法，是對的嗎？」

「老師說的一定正確。」

「Kruskal 的最小成本生成樹演算法，是對的嗎？」

「嗯，這個老外挺有名的，他設計的演算法應差不了多少。」

「有沒有方法證明這些貪婪法是正確的呢？」

「嗯…」

本節所介紹的理論，即是可用於證明一些貪婪法是正確的。貪婪演算法的設計常需要提供證明。而證明需要嚴格的論證，總是令人不易接受。以下是在 4.2 節中介紹 Kruskal 的最小成本生成樹演算法的證明。聰明如您，可以試著看看此證明正確否？

表 4.11 Kruskal 的最小成本生成樹演算法之證明

Kruskal 的演算法在一連接圖中找到最小成本生成樹

證明：

假設由 Kruskal 演算法找到的樹是 t，而 t' 是一棵最小成本生成樹。我們要證明的是這兩棵樹的成本相同。如此也證明 Kruskal 演算法找到了最小成本生成樹。令 $E(t)$ 及 $E(t')$ 為分別為兩棵樹的線集合 (edge set)，此時有兩種可能:

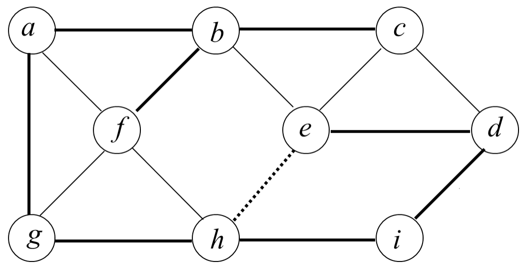
(1) 兩棵樹的線集合 (edge set) 相同，則這兩棵樹的成本必相同。得證。

(2) 兩棵樹的線集合 (edge set) 不相同，則令 q 為落在 t 中(即 $q \in t$)，但不在 t' 中的最小成本的一條線。

(2.1) 此 q 必然存在，否則兩棵樹的線集合相同。

(2.2) 將 q 加入 t' 中必產生唯一的一個迴圈，因為在任何一棵生成樹 (spanning tree) (下圖粗線) 上加入任意一條非樹上的新線 (下圖虛線) 必產生唯一的迴圈 (cycle)。

next



(2.3) 此唯一的一個迴圈中，必然至少存有一線不在 t 中，令 q' 為此線 ($q' \in t'$)。否則，此迴圈存在此樹 t 中(但 t 為一棵樹不應有迴圈)，產生矛盾。

(2.4) 線 $q' \in (t')$ 的成本必高於或等於線 $q(\in t)$ 的成本。否則 q' 的成本小於線 q 的成本，則 kruskal 的演算法會先考慮將 q' (而非 q)納入 t 中。(注意將 q' 納入 t 中不會產生迴圈的原因是，小於 q 而落於 t 中的線也必落於樹 t' 中，因為當初選擇 q 是落在 t 中但不在 t' 中的一條最小成本線。)

(2.5) 將 t' 中的 q' 換成 q 即($E(t') - \{q'\} \cup \{q\}$)會產生新的一棵生成樹，且此樹的成本不會高於原先的樹 t'。也是一棵最小成本生成樹，而其線集合比 t' 更靠近 t。

重複以上步驟，t' 會逐漸轉變成 t。因此得證。

看起來，想證明一個貪婪演算法是正確的，有時不太容易。以下介紹一種離散結構稱為 Matroid，可能協助證明一些貪婪演算法。Matroid 的定義如下表。

表 4.12 Matroid 的定義

Matroid

Matroid 是符合下列條件的一個系統 $M=(S, I)$:

(1) S 是一個有限元素的集合(非空集合)。

(2) I 是 S 的子集合所構成的集合(非空集合)，稱為 S 的*獨立集合*(independent set)需擁有以下兩個性質。

2.1 繼承性質(hereditary property): 如果 $B \in I$ 且 $A \subseteq B$ ，則 $A \in I$ 。

2.2 交換性質(exchange property): 如果 $A \in I$，$B \in I$，且 $|A| < |B|$ ，則存在 $x \in B - A$ 使得 $A \cup \{x\} \in I$ 。

Matroid 中獨立集合的繼承性質，為任何此集合中元素的子集合皆落入獨立集合中。如在圖 4.11 中{ABC}落入獨立集合中，則{ABC}的子集合皆落入獨立集合中。Matroid 中獨立集合的交換性質，則保證在獨立集合中任何一個較小的集合，可以自任何一個較大的集合找到一元素，加入後擴大為另一個獨立集合。圖 4.11 為示意圖。

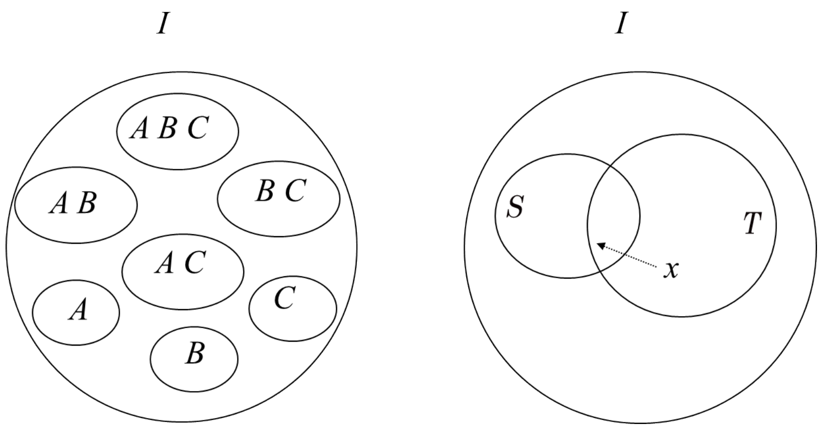


圖 4.11 (1) 繼承性質需要集合{ABC}的子集合皆落入獨立集合中。
(2) 交換性質存在 $x \in T - S$ 使得 $S \cup \{x\} \in I$。

前人發現有些問題可表示成，在有權重(正數)的 Matroid 中，尋找最大權重的獨立集合的問題。例如，最小成本生成樹的問題，是尋找最小成本的線得以連接所有的點。將{無迴圈的線集合}看做一個獨立集合，且將每條線的權重 w 轉換成 c-w，此處 c 是一個大於所有線權重的正值；則最小成本生成樹的問題，(可被證明)就是找出最大權重獨立集合的 Matroid。

此問題的繼承性質十分明顯，因為無迴圈的線集合的子集合必不含迴圈。而交換性質自表 4.10 的證明中隱約可見。本章<4.5 節>中的單位時間工作排程問題，也可表示成 Matroid。其中的獨立集合為{不被罰錢的工作集合}。

當有一個問題可以被表示成有權重(正數)的 Matroid 時，以下的貪婪演算法可以找到最大權重的獨立集合。

表 4.13 Matroid 的最佳貪婪演算法

輸入	$M=(S, I)$：一個有權重(weighted) Matroid
輸出	A： 最佳解

步驟	Algorithm Greedy
	{
	Step 1: $A=\phi$。
	Step 2: 將 S 依照其權重由大到小排列。
	Step 3: 依照上述的順序，將 S 的元素 x 試著加入 A 中：If $A\cup\{x\}\in I$ (即獨立集合) then $A=A\cup\{x\}$。
	Step 4: 回傳 A。
	}

4.7 貪婪法的技巧

貪婪法的一般設計步驟如下表所示，請參閱。

表 4.14 貪婪法的一般步驟

```
Algorithm Greedy(a, n)
// a[1:n] 包含 n 輸入；solution 為解集合
{
solution:=φ;
for i:=1 to n do
  {
  依照某法則，自 a 中選擇出 x。
  if 依照某法則判斷 x 可以被加入 solution 中
  then
  solution:=solution∪{x}。
  }
回傳 solution。
}
```

當您想用貪婪法解題時，注意您的方法，是不是針對所有輸入都可找到想要的答案或最佳解。若問題需要的答案是必須找出最佳解，而您的方法只是找到還不錯的解或偶爾找到最佳解，則可能不符需要。

另外，可表示成 Matroid 的問題可利用貪婪法解決或提供其正確性證明；但是，不代表當一個問題不被表示成 Matroid 時，就斷定此問題決不可為貪婪法所解。

總而言之，使用貪婪法時要小心設計並證明，以免失去找到最佳解的機會。最後，一般演算法書中提到，並可以被貪婪法解的尚有以下問題：

1. **磁帶上的最佳記憶空間**(optimal storage on tapes)：有 n 個程式想被儲存在一個長度為 L 的磁帶上。假設每一個程式被讀取時，磁帶都是在起始的位置。每個程式的長度可能不同。請找出一個 n 個程式的儲存排列方式，使得其平均的讀取時間為最短。

2. **最佳合併模式**(optimal merge patterns)：兩個排列好的檔案分別有 s 和 t 筆資料，共需要 $O(s+t)$時間，加以合併成一個排列好的檔案。輸入 n 個排列好的檔案(大小不一)，請找出一個花費最少比較次數的合併模式，可以兩兩合併成最終一個檔案。

學習評量

1. 請寫一個程式，將一連接圖 $G=(V, E)$的最小成本生樹的權重總和輸出，其中 V 為點集合，而 E 為有權重的線集合。

```
輸入：
9                (點 V 的個數)
15               (線 E 的個數)
a b 2            (以下是線及其權重)
a f 13
a g 7
b c 6
b e 15
b f 9
c d 10
c e 12
d e 3
d i 7
e h 8
f g 11
f h 14
g h 5
h i 1

輸出：
40               (最小成本生樹的權重總和)
```

2. 有 n 個程式想被儲存在一個長度為 L 的磁帶上。假設每一個程式被讀取時，磁帶都是在起始的位置。每個程式的長度可能不同。請找出一個 n 個程式的儲存排列方式，使得其平均的讀取時間為最短。假設每一個程式被讀取的機率是一樣的。例如，當 $n=3$，而三個程式的長度為(5, 3, 1)時，不同的儲存順序將會有不同讀取時間：

順序	讀取時間
5, 3, 1	5+(5+3)+(5+3+1)=22
5, 1, 3	5+(5+1)+(5+1+3)=20
3, 5, 1	3+(3+5)+(3+5+1)=20
3, 1, 5	3+(3+1)+(3+1+5)=16
1, 3, 5	1+(1+3)+(1+3+5)=14
1, 5, 3	1+(1+5)+(1+5+3)=16

輸入：
```
3                    (程式的個數)
5 3 1                (每個程式的長度)
```

輸出：
```
1 3 5                (最佳的儲存排列方式)
```

3. 兩個排列好的檔案分別有 s 和 t 筆資料，共需要 $O(s+t)$ 時間，加以合併成一個排列好的檔案。輸入 n 個排列好的檔案(長度不一)，請找出一個花費最少比較次數的合併模式，可以兩兩合併成最終一個檔案。

輸入：
```
20 30 10 5 30    (每個檔案的長度)
```

輸出：
```
205              (最少的比較次數)
```

Memo