Ch7. 貪婪演算法 Greedy Algorithm

本章一開始，一樣會簡介貪婪演算法是什麼，以及它在什麼情況下適用。

接下來，會介紹貪婪演算法常見的應用：

A. 找錢問題 Coin Changing
B. 中途休息 Breakpoint Selection
C. 活動選擇問題 Activity-Selection Problem
D. 背包問題 Fractional Knapsack Problem
E. 工作排程 Task Scheduling

最後，一樣會選擇 LeetCode 上幾題相關的題目練習。

1. 貪婪演算法簡介

貪婪演算法可以被視為是一種「短視近利 / 偷懶 / 貪婪」的想法，這也是它名稱的由來，根本精神是在每一步要做選擇時，選擇當下的「局部最佳解」。

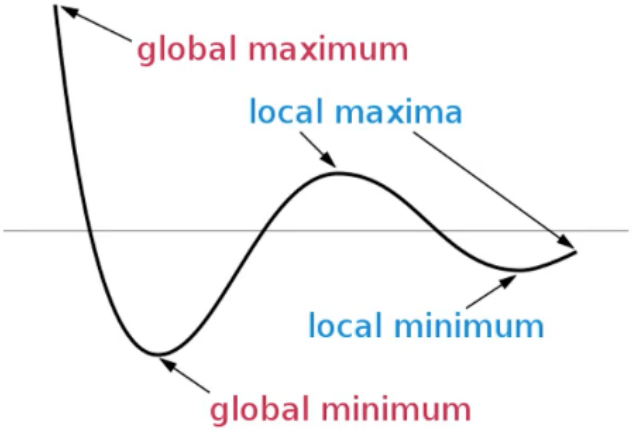
比如踏出家門後想要去學校，就每一步都選擇「一旦踏出就可以距離學校近一點」的方向走，希望能夠藉此到達學校。

舉另一個例子，如果一個非本科系的學生想在最短時間內當上「有名的律師」，那麼他就應該選擇能夠最快考到律師執照的方式：去念法律學分班（因為沒有學分不能報考），這樣所花的時間會比回頭念大學法律系來得短。

（1）不適用貪婪演算法的問題

如果登山時，迷路而想回到山下的城市，這時一種想法是環顧四周後，「往下坡的方向走」，希望這樣能帶我們回到山下。因為貪婪演算法的精神是在每一刻都採取當下的「局部最佳解」，所以會希望每一步都一定要往下方走一點，而不能走了某一步後，反而海拔變高了。

但是很明顯的，採取這種策略不一定能夠順利回到山下的城市，也可能反而被困在四周都較高的山谷（局部最低點）之中。有時候，從現在地出發，必須先跨越一個山峰後，才能回到海拔較低的城市，這種情況下貪婪演算法就因為「短視近利」的特性，而無法找到正確的道路。



就像上面的函數中，如果從「比右邊局部最低點 local minimum 還要右側」的某處出發，那麼根據「只往下走」的策略，就會陷在右邊的局部最低點，因為從該處往左就是「上坡」，所以不會選擇再往左走。

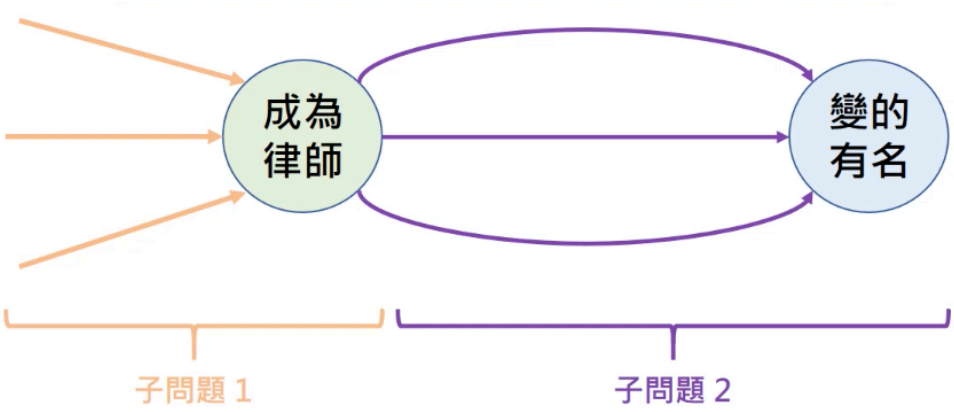
尋找道路回到山下的過程中，每一步做的選擇都會影響到下一步的選擇，所以選擇間的次序不同，就可能產生不同的結果。這種掉到「局部最低點」，而非真正想要抵達的「全域最低點 global minimum」的情形，也是機器學習中面臨到的一個很大的挑戰。

（2）大律師問題

那麼大律師問題為什麼可以使用貪婪解法呢？

這是因為要「成為大律師」，需要先後解決兩個子問題：

A. 成為律師
B. 變得有名



而這兩個子問題基本上是互相獨立的，不管是念學分班還是大學法律系，考到律師後的資格都相同，因此都會再從一樣的出發點接續去解決「變得有名」這個子問題。

因為兩個子問題獨立，所以可以先找到最快成為律師的方式，完成這步後，再去找最快變得有名的方式，一般來說，不用擔心採取某個方法快速取得律師資格後，影響到之後能否變得有名。

簡單來說，如果子問題間不互相獨立、會彼此影響，就不適合使用貪婪解，就像城市可能在左手邊跨過山峰的地方，可是現在站的地方卻是右邊比左邊低，這樣一直採用貪婪解往右走後，沒辦法達到原本的目的；相對的，子問題間相互獨立時，可以使用貪婪解，就像不管用什麼方法當上律師或考到某個證照，取得的資格都是相同的，不會影響到之後執業能不能變得有名、成功。

（3）賺大錢問題

若目標是「賺最多錢」，而需要做下方的兩種決定：

A. 工作選擇
    a. 跑 Ubxx 賺 250 元 / hr
    b. 跑 熊 X 賺 220 元 / hr

B. 行程選擇
    a. 好好讀書考國考 0 元 / hr
    b. 去打工賺 200 元 / hr

在第一個選擇中，不管今天是跑 Ubxx 還是熊 X，因為做的事情性質差不多，所以除了賺到的錢以外，不會對做選擇的人明天的狀態造成什麼差別。

反之，第二個選擇中，如果選了唸書這項，明天腦袋裡的知識就會增加，也會稍微提升考過國考的機率，反之，如果去打工，則不會達成國考知識的累積。

也就是說，因為第一個選擇中不管做哪個，對之後的狀態、未來能做的後續選擇並沒有什麼直接影響，所以可以用貪婪演算法選擇「時薪最高」者就好。

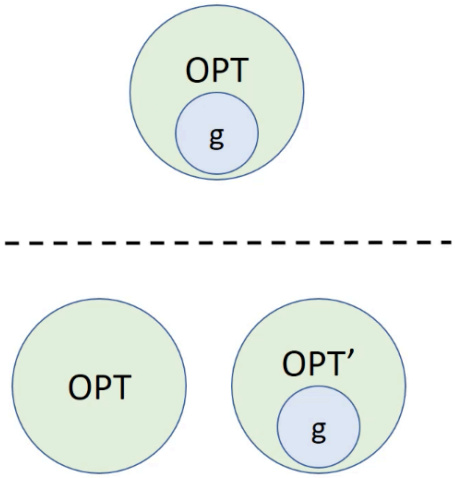
反之，第二個選擇中，做出的選擇會對未來的狀態產生影響：如果去念國考，之後會產生能考過國考取得資格的狀態，選擇去打工（貪婪演算法會得到的解），則之後無法考過國考，也就無法選擇去做國考的工作，不一定能夠達成「賺最多錢」的目標。因此第二個選擇中，貪婪解就不是一個適合的解法。

（4）貪婪演算法的適用範圍

一個問題要適用貪婪演算法，要有下列條件：
    A. 最佳化子結構：子問題的最佳解在原問題的最佳解內
    B. 適用貪婪選擇：局部最佳解能夠組成全域最佳解

如果目標是「退休前賺最多錢」，要解決的原問題是退休前的所有行程規劃，而子問題是「今天下午的行程規劃」，那麼今天下午選擇去做可以賺最多錢的行程，顯然不一定會與「真正可以在退休前賺到最多錢所必需的行程」中「今天下午該做的事情」相符，所以像這樣的行程規劃問題就不符合上面的「最佳化子結構」條件。

（5）利用 Exchange Arguments 證明貪婪演算法的適用



A. 目前子問題的貪婪最佳解為 g
B. 任意（全域）最佳解為 OPT
C. 產生兩種情形
    a. g 在 OPT 中 -> 沒問題
    b. g 不在 OPT 中

若 g 不在 OPT 中，修改 OPT 成為 OPT'，使 OPT' 包含 g
    a. 如果 OPT' 比 OPT 還要好，反證，因為全域最佳解不是 OPT
        （跟假設矛盾，不適用貪婪演算法）
    b. 如果 OPT' 跟 OPT 正好一樣好，得證（可適用貪婪演算法）

綜合來說，貪婪演算法的步驟就是從原問題中切出至少一個子問題，並使用「貪婪」的方法來解決這個子問題。不斷切割出子問題並解決後，把所有子問題的答案拼湊起來，如果這些子問題的答案會共同構成全域最佳解，貪婪演算法就解決了這個問題。

要記得貪婪演算法雖然通常有效率且簡單，只要一步步找出目前最佳解，但是貪婪演算法並「不能」解決所有最佳化問題，當每一步的選擇會影響到下一步時，就不能適用，而應改用下一章介紹的動態規劃。

後面會學到尋找「最短路徑」的 Dijkstra 演算法和資料結構中會教到的「霍夫曼編碼」，都是貪婪演算法的應用。

2. 找錢問題

接下來，開始來看幾個貪婪演算法的常見應用，首先是「找錢問題」:

如果需要找給顧客 n 元，而目前有四種硬幣：1 元、5 元、10 元和 50 元，如何找錢可以找的硬幣「個數」最少呢？

舉例來說，如果要找給顧客 128 元，應該使用 50 元硬幣兩枚，10 元硬幣 2 枚，5 元硬幣 1 枚，與 1 元硬幣 3 枚，因為以下等式成立：

$$50 \times 2 + 10 \times 2 + 5 \times 1 + 1 \times 3 = 128$$

上面這種做法總共使用了 8 枚硬幣，想法是先用 50 元盡量貼近 128 元，等到使用了兩個 50 元，還差 28 元時，轉為使用 10 元繼續貼近，依此類推。

（1）證明貪婪演算法適用找錢問題

怎麼證明這個問題可以適用貪婪演算法呢？以決定「應使用兩個 50 元」這步為例，利用反證法證明：

A. 假設與前述貪婪演算法所得之解「不同」，要找 128 元時最佳解「應使用 0 枚或 1 枚 50 元」
B. 另外，又有 10 元硬幣只應使用 0~4 枚的限制（否則可以用 1 枚 50 元來替代 5 枚 10 元，使用到的硬幣個數就能更少，才會是最佳解）
C. 同樣的，知道 5 元硬幣只應使用 0~1 枚
D. 1 元硬幣只應使用 0~4 枚
E. 這樣一來，在「只應使用 0 枚或 1 枚 50 元」的前提下，最多只能找出 $50 \times 1 + 10 \times 4 + 5 \times 1 + 1 \times 4 = 99$，不符合要找 128 元的要求，反證。

因為成功反證，代表「應使用兩個 50 元」是正確的（也可以說，證明了要找 100 元以上時一定都會用到兩個或以上的 50 元），同樣的方法也可以被用來依序證明 10 元、5 元、1 元應使用的枚數都與貪婪解相同，代表從最大面額的硬幣開始「能用到幾個就選擇用到幾個」，本題適用貪婪演算法。

（2）LeetCode #860 檸檬水找零 Lemonade Change

A. 題目

有個賣檸檬水的攤位，每杯檸檬水賣 5 元。有一些顧客排隊來買，而他們一次都只會點「一杯」。每個顧客點了他們的一杯檸檬水後，會用一枚面額 5 元、10 元或 20 元的硬幣來結帳。你必須正確找零，使得找零後的結果是每個顧客總共只付給你 5 元（顧客付 10 元時找 5 元，付 20 元時找 15 元）。

注意一開始手上並沒有任何零錢，所以有時會遇到無法找零的情形。只有在從頭到尾都能正確找零給每個顧客時，回傳 true。

B. 出處
https://leetcode.com/problems/lemonade-change/

C. 範例輸入與輸出

輸入：[5,5,5,10,20]
輸出：true

因為第四個顧客付 10 元時，手上已經有 5 元可以找，第五個顧客付 20 元時，手上也能找出 15 元，所以回傳 true。

輸入：[10,10]
輸出：false

因為第一個顧客付 10 元時，手上還沒有任何零錢可以找，所以回傳 false。

檸檬水找零 Lemonade Change	
1	class Solution{
2	public:
3	bool lemonadeChange(vector<int>& bills){
4	

```
5           // 紀錄手上的不同硬幣個數
6           // 開始時每種硬幣個數都是 0
7           int five = 0;
8           int ten = 0;
9           int twenty = 0;
10
11          int len = bills.size();
12
13          // 依序處理每位顧客的找錢流程
14          for (int i=0 ; i<len ; i++){
15
16              // 第 i 個顧客使用的硬幣
17              int coin = bills[i];
18
19              // 顧客使用 5 元時，手上增加一枚 5 元
20              if (coin==5){
21                  five++;
22              }
23
24              // 顧客使用 10 元時
25              else if (coin==10){
26                  // 手上有 5 元硬幣，能夠成功找錢
27                  if (five){
28                      five--;   // 找給顧客一枚 5 元
29                      ten++;   // 手上增加一枚 10 元
30                  }
31                  // 不能成功找錢
32                  else {
33                      return false;
34                  }
35              }
36
37              // 顧客使用 20 元時
38              else if (coin==20){
```

```
39
40                      // 5 元比較珍貴，優先把 10 元找掉
41                      // 但也要同時至少有一枚 5 元才能找
42                      if (ten&&five){
43                          ten--;
44                          five--;
45                          twenty++;
46                      }
47
48                      // 沒有 10 元時，才試著用 3 枚 5 元找錢
49                      else if (five){
50                          five-=3;
51                          twenty++;
52                      }
53
54                      // 不能成功找錢
55                      else {
56                          return false;
57                      }
58
59                  } // end of outer if
60
61              } // end of for
62
63          } // end of lemonadeChange
64
65  }; // end of Solution
```

（3）LeetCode #322 硬幣找零 Coin Change

注意本題使用貪婪演算法來解決，會得到「錯誤」的答案，但建議讀者仍可以試著先寫出對應的貪婪解法，並思考為何會出錯，下一章介紹動態規劃方法就可以順利解決這個問題。

A. 題目

給定一個整數陣列 $coins$ 代表各種硬幣的面額，與一個整數陣列 $amount$ 代表要達到的總額。

回傳「要達到該總額最少需要的硬幣個數」，如果根據給定的各種硬幣面額無法剛好達到該總額，回傳 -1。每種硬幣的可用個數沒有上限。

B. 出處
https://leetcode.com/problems/coin-change/

C. 範例輸入與輸出
輸入：coins = [1,2,5], amount = 11
輸出：3

最少只要使用兩枚 5 元硬幣和一枚 1 元硬幣就可以得到 11 元。

輸入：coins = [1,8,10], amount = 25
輸出：4

只要使用三枚 8 元硬幣和一枚 1 元硬幣就可以得到 25 元。注意貪婪解會得到使用「兩枚 10 元硬幣和五枚 1 元硬幣」的結果，並不是最佳解。

硬幣找零 Coin Change	
1	class Solution{
2	public:
3	int coinChange(vector<int>& coins, int amount){
4	

```
5           // 例外處理：amount 為 0
6           if (amount == 0)
7               return 0;
8
9           // 一般情形
10
11          // 把硬幣面額從大到小排
12          sort(coins.begin(), coins.end(), greater<int>());
13
14          int coin_count = 0;
15          int len = coins.size();
16
17          // 從面額最大的硬幣開始盡量貼近總額
18          for (int i=0 ; i<len ; i++){
19
20              // 當前處理的面額
21              int value = coins[i];
22
23              // 目前面額硬幣最多可以使用的個數
24              // 例如總額為 240 時，可以使用四個 50 塊
25              coin_count += amount/value;
26
27              // 更新 amount
28              // 例如用了四個 50 塊後，amount 從 240 更新為 40
29              amount %= value;
30
31              // 可以找開時，回傳
32              if (amount == 0)
33                  return coin_count;
34              else
35                  return -1;
36
37          } // end of for
38
```

```
39          } // end of coinChange
40
41  }; // end of Solution
```

不過上面的程式碼送出後，會得到 Wrong Answer 的結果，需留待下一章解決。

考慮下面兩種情況：

A. 四種硬幣：1、5、10、50 元
B. 三種硬幣：1、8、10 元

為何第一種情況可以用貪婪演算法得到正確的解，第二種卻不行呢？這是因為第一種狀況中，各個面額間是整數倍數，多枚小硬幣可以用一枚大硬幣替代，所以子問題間不相關。

相反的，第二種情況下，面額間不是整數倍數，會有不能替換的情形發生。另外，讀者也可以把第二種情況代入前面針對第一種情況的反證法，思考一下反證是否還成立。

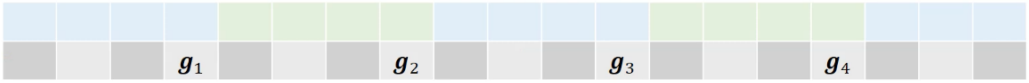
3. 中途休息

接下來，討論中途休息的問題：

有一段距離為 d 公里的旅途需要行駛，油箱的容量每次只能連續行走 c 公里，給定一連串加油站位置 g_i，每次加油後都能夠繼續行走 c 公里，該如何安排加油的地點，使整個旅途中停下來的次數最少？

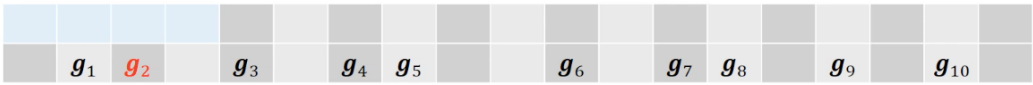
（1）利用貪婪演算法解中途休息問題

根據貪婪演算法的精神，訂出的策略會是「每次加油之前，能走越遠越好」。

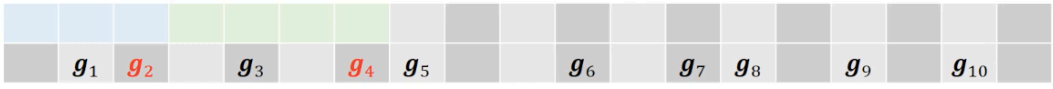


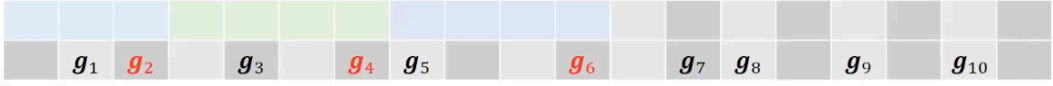
把題目用上圖表示，假設 $c = 4$，每次加油後只能往右走 4 個方格。理想的狀況下，每次油耗盡時正好都有加油站，但是現實中，這不一定成立，所以在油耗盡之前，必須提前找到一個加油站加油才能繼續前進。

貪婪演算法會去找每次可以開到的範圍內「最後一個加油站」，即「每次加油之前，能走越遠越好」的策略。

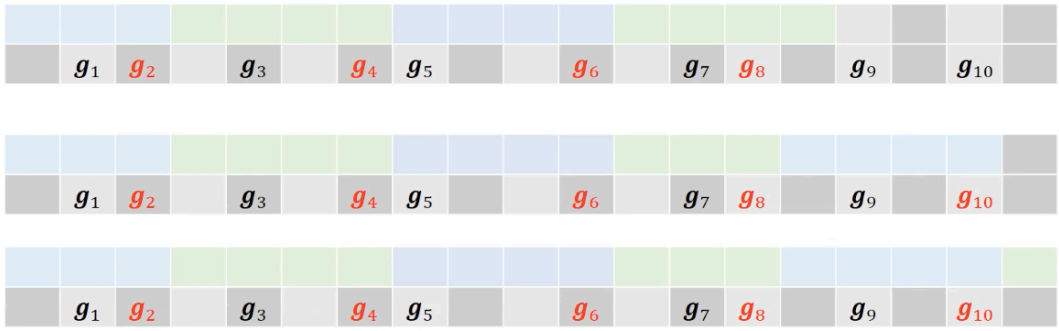


上圖中，一開始可以走到左邊數過來第四格，所以會選擇在 g_2 加油，加完油後，最遠可以再向右走到第七格，所以選擇在第七格以左（包含第七格）的最後一個加油站 g_4 再加一次油。



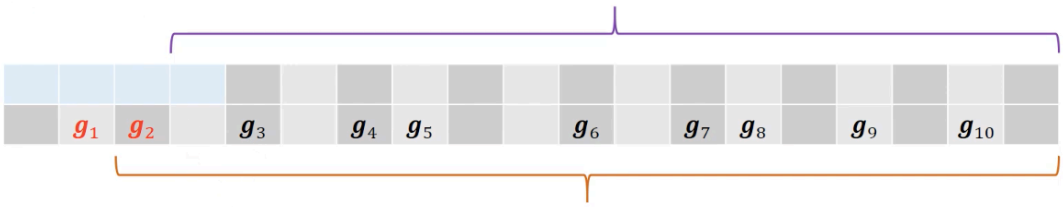


以此類推，整趟旅程中會在 g_2、g_4、g_6、g_8、g_{10} 加油，總共加了 5 次油。



（2）貪婪演算法的適用性

那麼使用貪婪演算法能不能得到這題的最佳解答呢？這裡給出一個大致的證明：



不妨假設某次加油後，剩下的路途如上圖所示，貪婪解告訴我們應在 g_2 加油。若選擇這次不在 g_2 加油，則有兩種情形：

A. 選擇 g_2 後的加油站（如 g_3）：油會在到達加油站前用完，不合

B. 選擇 g_2 前的加油站（如 g_1）：用 g_2 來取代，不會使得加油次數增加

針對「選擇 g_2 前面的加油站」說明：如果在 g_2 加油，那麼接下來可以選擇 4~7 格間的加油站來加油，包含 g_3 和 g_4，而貪婪解會選擇其中的 g_4。

如果 g_4 在全域最佳解（真正的最佳解）當中，那麼一開始一定要選擇 g_2 而非 g_1 加油，才能下一次就到達 g_4 那格；相反的，如果 g_3 才在全域最佳解當中，那麼乍看之下似乎一開始就要選擇 g_1 而非 g_2 來加油，才會在第二次選擇 g_3。

然而如果一開始在 g_1 加油，那麼接下來可以選擇的範圍是 3~6 格之間的加油站來加油，這個範圍內的加油站都可以在選擇 g_2 的情況下走到，也就是說，在 g_2 加油後的選項只會比在 g_1 加油後多，而不會較少，所以選擇 g_2 相較 g_1 只有優點而沒有缺點。同樣的道理，之後的過程中也都應該在每趟路

程中盡量能走到哪裡就走到哪裡，在可行範圍內的最後一個加油站加油即可。

也就是說，中途休息的問題可以適用「貪婪演算法」。

（3）實作貪婪演算法解中途休息問題

題目：讓使用者輸入每次加滿油之後可以行走的距離 c、旅途總長度 d、加油站數目 n 與每個加油站的座標，輸出這趟旅途中至少要停下來加油幾次，以及每個停靠加油站的座標。

A. 輸入：

>> 4 19 10

>> 2 3 5 7 8 11 13 14 16 18

B. 輸出

Stop @3

Stop @7

Stop @11

Stop @14

Stop @18

Stops = 5

中途休息	
1	#include <iostream>
2	#include <vector>
3	#include <algorithm> // for sort
4	using namespace std;
5	
6	int main(){
7	
8	int c, d, n;
9	cout << "Please enter c,d,n:" << endl;
10	cin >> c >> d >> n;
11	vector<int> gas_station(n);

```
12      // 獲取每個加油站的位置
13      for (int i=0 ; i<n ; i++){
14          cin >> gas_station[i];
15      }
16
17      // 將加油站依預設規則由小到大排序
18      sort(gas_station.begin(), gas_station.end());
19
20      // 記錄停靠的加油站數量
21      int stops = 0;
22
23      cout << "Stops = " << stops << endl;
24
25      // 在終點（最後一格右邊）放一個加油站
26      // 只要檢查是否有到這個加油站，就知道是否有走到終點
27      gas_station.push_back(d);
28
29      // 上一個有停靠的加油站是所有加油站中「第幾個」
30      int last_stop_index = -1;
31      // 上一個有停靠的加油站座標
32      int last_stop_position = 0;
33
34      // 尋找「可行範圍內最後面一個加油站」的邏輯是
35      // 先找到「超過 c 距離外」，最近的一個加油站
36      // 再往回退一個加油站（退回 c 的距離內）
37
38      // 因為在終點處多加了一個加油站，總共要檢查 n+1 次
39      for (int i=0 ; i<n+1 ; i++){
40
41          // 檢查和上次停靠的距離超過 c 後，第一個遇到的加油站
42          if (gas_station[i] - last_stop_position > c){
43
44              // 如果上次停靠的加油站正好是第 i-1 個（i 的上一個）
45              // 那麼 i-1 和 i 兩個相鄰的加油站間距離超過 c
```

```
46                  // 沒有可行解
47                  if (last_stop_index == i-1){
48                      cout << "No solution!" << endl;
49                      stops = -1;
50                      break;
51                  }
52
53                  // 如果上次停的加油站和第 i 個加油站間還有一個加油站
54                  // 就退回這個加油站（可行距離內的最後一個加油站）
55                  last_stop_index = i-1;
56                  last_stop_position = gas_station[i-1];
57
58                  // 停靠的加油站增加一個
59                  stops++;
60                  cout << "Stops @" << last_stop_position << endl;
61
62                  // 因為要停第 i-1 個加油站
63                  // 停完後重新從第 i 個加油站檢查起
64                  i--;
65
66              } // end of if
67          } // end of for
68
69          cout << "Stops = " << stops << endl;
70
71          return 0;
72  }
```

注意到，如果終點加上的那個加油站和題目中給的最後一個加油站之間距離超過 c，一樣會輸出沒有可行解；反之，距離沒有超過 c 時，不會再進入「for 中的第一個 if 迴圈」而多處理一次，也就不會多輸出一次。

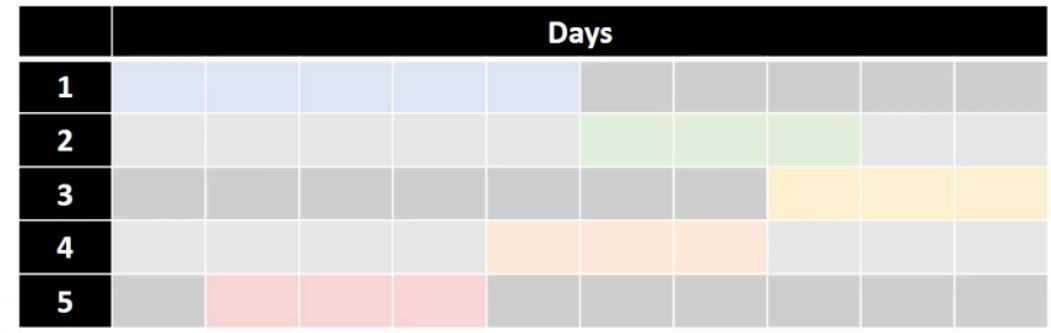
因此，在終點加上一個加油站後不需另行處理邊界情況，維持了程式的簡潔。

4. 活動選擇

接下來，來看活動選擇問題：

學校只有一個音響，但舉辦 n 個活動中任一個都需要使用到它，給定每個活動的開始時間與長度，如何選擇可以讓舉辦的活動總數最多？

（1）利用貪婪演算法解活動選擇問題



上圖中，假設第一個活動舉辦時間（橫軸）是第 1 到第 5 天，第二個活動是第 6 到第 8 天，其它活動依此類推。因為「音響」這個資源有限，所以每一天同時只能舉辦一個活動，也就是說，選擇的活動之間，期間不能相互重疊。

根據貪婪演算法的精神，可以訂出一個策略：「活動日程完全包含另一活動的，優先選擇該被包含的活動」。比如上例中活動 1 的時間完全包含了活動 5 的時間，兩者間必定只能擇一，選擇活動 5 而不選擇活動 1 時，不但一樣算有辦「一個活動」，又能空出前後的若干時間段（比如可以舉辦活動 4），所以是較好的選擇。

要執行上面的策略，首先把活動依照開始日期進行排序。接著，檢查兩個相鄰活動的區間，假設開始時間分別是 s_1 與 s_2，結束時間分別是 f_1 與 f_2。

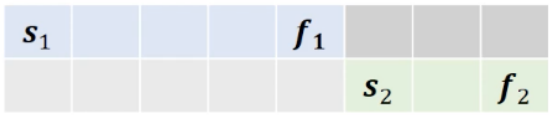
假設 s_1 不晚於 s_2，這兩個相鄰活動的區間共可形成三種情況：

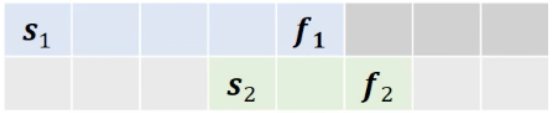
A. $s_2 > f_1$：兩個活動時間沒有任何重疊，採納活動 1

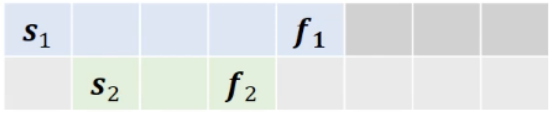
B. $s_2 \leq f_1$

a. $f_2 > f_1$：選擇較早結束者（為什麼？），即採納活動 1

b. $f_2 \leq f_1$：活動 2 被包含在活動 1 的時間內，採納活動 2







針對上面三種情況中第二種進行說明：活動 1 比活動 2 早開始也早結束（$s_1 \leq s_2$、$f_1 < f_2$），那麼這時選擇舉辦活動 1 一定較有利，為什麼呢？

程式執行時，會從第一天（圖的左方）開始依序處理，試著反證：

第一種情形

假設 1：存在「某個活動 0」，它的舉辦時間完全被包含在 s_1 到 s_2 這個區間內，或者它的開始時間比 s_1 早，而結束在 s_1 到 s_2 區間內。

假設 2：應選擇結束在後的活動（選擇活動 2 而非活動 1）

假設 3-1：選擇「活動 0」

- 與假設 2 矛盾，因為根據假設 2，會選擇活動 1 而非活動 0

假設 3-2：不選擇「活動 0」

- 因為不選擇任何這種活動 0，所以選擇活動 2 而不選活動 1 只會往後多佔空間，往前空出的空間沒有用處，所以一定不會較佳
- 與假設 2 矛盾

第二種情形

假設 1：不存在「某個活動 0」，它的舉辦時間完全被包含在 s_1 到 s_2 這個區間內，或者它的開始時間比 s_1 早，而結束在 s_1 到 s_2 區間內。
假設 2：應選擇結束在後的活動（選擇活動 2 而非活動 1）

- 因為不存在這種活動 0，所以選擇活動 2 而不選活動 1 只會往後多佔空間，往前空出的空間沒有用處，所以一定不會較佳
- 與假設 2 矛盾

綜合上面兩種情形，無論是否存在這種「活動 0」，假設 2 都會產生矛盾，因此假設 2 不成立，也就是說，前面三種情況中第二種，應該選擇「結束在前」的活動 1。

上面三種情況的結論，正好可以被簡化為：在有重疊的兩個活動中擇一時，總是選「結束時間較早者」。根據這個邏輯，來練習 LeetCode 上的活動選擇問題。

（2）LeetCode #435. 不重疊的區間 Non-overlapping Intervals

A. 題目

給定一個含有多個區間的陣列 $intervals$，其中 $intervals[i] = [start_i, end_i]$，回傳至少需要移除多少個區間，才能使剩下的區間不互相重疊。

B. 出處
https://leetcode.com/problems/non-overlapping-intervals/

C. 範例輸入與輸出
輸入：intervals = [[1,2],[2,3],[3,4],[1,3]]
輸出：1
只要移除 [1,3] 這個區間，就可以使剩下的區間不互相重疊。

D. 解題邏輯

這題雖然與剛才活動選擇的問題問法不同，但是背後的邏輯是相同的：最小化「移除」區間的數目，也就是最大化「保留」區間的數目，且根據

全部活動數目 − 保留區間數目 ＝ 移除區間數目

就可以得到本題答案，因此同樣根據剛才訂出的策略，選擇活動時保留先結束者即可。

不重疊的區間 Non-overlapping Intervals

```
1  class Solution{
2
3  /*
4  // 排序根據活動的「開始時間」
5  // 不過預設排序方式正好與此相同，因此可以不特別加上此函式
6  bool cmp(vector<int> interval_1, vector<int> interval_2){
7        return interval_1[0] > interval_2[0];
8  }
9  */
10
11 public:
12       int eraseOverlapIntervals(vector<vector<int>>& intervals){
13
14             // 排序活動
15             sort(intervals.begin(), intervals.end());
16
17             // 紀錄刪除了多少個區間
18             int delete_number = 0;
19             // 把結束時間 finish 初始化成第一個活動的結束時間
20             int finish = intervals[0][1];
21
22             for (vector<int>& interval:intervals){
23
```

```
24                  int next_start = interval[0];
25                  int next_finish = interval[1];
26
27                  // 兩相鄰活動沒有重疊，不需刪除活動
28                  if (next_start >= finish){
29                      // 更新 finish 為目前檢查活動的結束時間
30                      finish = next_finish;
31                  }
32
33                  // 兩相鄰活動有重疊
34                  else {
35
36                      // 一定需要刪除其中一個活動
37                      delete_number++;
38                      // 只保留兩者中「較早結束的活動」
39                      // 所以 finish 更新為「較早結束的活動」的結束時間
40                      // 即 finish = min(finish, next_finish)
41                      finish = finish<next_finish ? finish : next_finish;
42
43                  }
44
45                  // 迴圈第一次執行時，第一個活動會和自己比較到
46                  // 所以多計算了一次刪除，需要扣掉
47                  return delete_number--;
48              }
49
50          } // end of eraseOverlapIntervals
51
52  }; // end of Solution
```

本題之所以可以用貪婪演算法解決，是因為目標是讓舉辦的活動「數目」最多，也就是說，每個活動的重要性是相同的，但如果每個活動還有「權重」，相互間重要性有別，則必須使用動態規劃。

5. 背包問題 Knapsack Problem

接下來，要討論「背包問題」:

給定固定的背包負重 W 以及每個物品的重量 w_i 及價值 v_i。如何在不超過背包負重的情況下，讓背包裡的物品總價值最貴重？

背包問題還有很多種細分的類型：
A. 0/1 Knapsack Problem：每項物品最多只能拿一個
B. Unbounded Knapsack Problem：每項物品可以拿無限多個（但須是整數）
C. Multidimensional Knapsack Problem：背包體積（而不只是負重）有限
D. Multiple Choice Knapsack Problem：每一「類型」物品最多拿一個
E. Fractional Knapsack Problem：物品可以只拿分數個（比如 0.5 個）

其中，貪婪演算法只適用在「Fractional Knapsack Problem」上，如果限定每一種物品必須拿「整數」個，貪婪演算法就不能正確解決。

Fractional Knapsack Problem 中，可以拿每個物品的「一部份」，但上限是拿「一個」該物品。如果每個物品的重量與價值如下所示，可計算出每種物品「每單位重量的價值」。接下來，從單位重量價值最高者開始取，直到把背包放滿即可。

$index$	1	2	3	4	5	6
w_i	4	5	2	6	3	7
v_i	6	7	5	12	5	18
$\frac{v_i}{w_i}$	1.5	1.4	2.5	2	1.67	2.57

上例中，假設背包負重為 16：
A. 編號 6 的物品拿滿重量 7，目前 value = 18，剩餘負重 = 9（16 − 7）
B. 編號 3 的物品拿滿重量 2，目前 value = 23，剩餘負重 = 7
C. 編號 4 的物品拿滿重量 6，目前 value = 35，剩餘負重 = 1

D. 編號 5 的物品拿滿重量 1，目前 value $= 36\frac{2}{3}$，剩餘負重 = 0

根據貪婪演算法的策略，最多可以拿到價值 $36\frac{2}{3}$ 的物品。

利用本章開頭介紹的反證法來證明本題適用貪婪演算法（單位價值最高的物品 $Obj_1(w_i, v_i)$ 必在最佳解中）：假設 $FoK(n)$ 是負重 n 的背包最高可拿的價值、OPT 是問題的最佳解

A. 單位價值最高的物品 $Obj_1(w_i, v_i)$ 若在 OPT 中：Obj_1 移走後的結果，仍然是 $FoK(n - w_i)$ 的最佳解

B. 假設單位價值最高的物品 $Obj_1(w_i, v_i)$ 不在 OPT 中：OPT 中的物品用 Obj_1 來替換，價值會增加，代表假設的 OPT 並非最佳解，反證。

也就是說，「Obj_1 不在 OPT 中」的情形不會出現，Obj_1 必定在 OPT 中。

但是其它種背包問題就不能這樣解決，上面的反證也不能成立，比如物品需要取整數個時，OPT 中的物品不一定可以用 Obj_1 來替換（單位價值較高的物品可能很重，取「一個」就會超過背包剩餘負重）。

6. 工作排程

再來是工作排程問題：

給定一連串回家作業 H_i，每份作業都有各自的繳交期限 D_i 與遲交時的扣分 P_i，已知每份作業都需要正好一整天的時間完成（一天只能完成一份作業），如何安排寫作業的順序可以讓「遲交造成的扣分」最少？

（1）用貪婪演算法求解工作排程

如下圖所示，用 S_i 來代表繳交第幾天繳交該份作業，如果這個 S_i 比當天交的作業的期限 D_i 來得大，就代表那份作業遲交了，會被扣 P_i 分。

S_i	1	2	3	4	5	6
D_i	4	5	4	2	3	2
P_i	8	3	7	6	5	2

比如 $S_i = 4$ 時，已經是第四天，但是繳交的作業期限卻是第二天，因此算作遲交，會被扣 6 分。上面這種任意的安排下，後面繳交的三份作業都遲交，總共被扣 $6 + 5 + 2 = 13$ 分。

其中一種「貪婪」的想法，是「以期限的順序來寫作業」：

S_i	1	2	3	4	5	6
D_i	2	2	3	4	4	5
P_i	6	2	5	8	7	3

這種做法中把作業依據 D_i 來排序，期限近的先寫完交出，只有最後兩份作業遲交，被扣 $7 + 3 = 10$ 分。

但是這種做法顯然不一定會得到最佳解，比如把第二天和第五天寫的作業對

調，就只會被扣 $2+3=5$ 分。

那麼改為「以扣分的大小順序」來寫作業，會得到最佳解嗎？

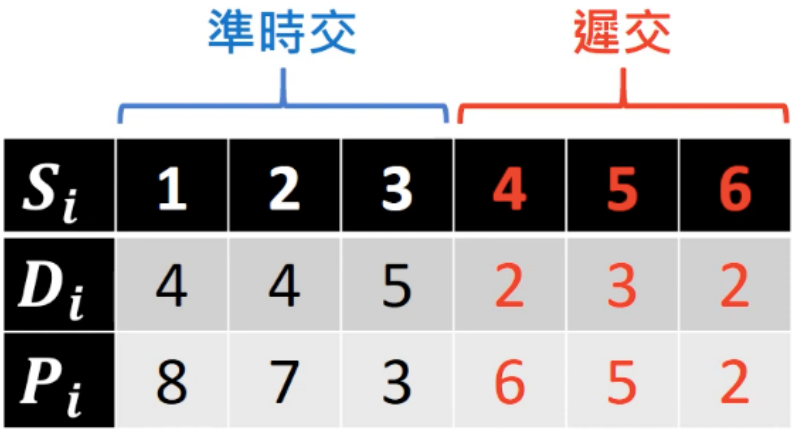
S_i	1	2	3	4	5	6
D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

上面的圖中可以看到，把作業完成順序依據「扣分由大到小」排列時，總扣分為 $6+5+2=13$，並非最佳解。

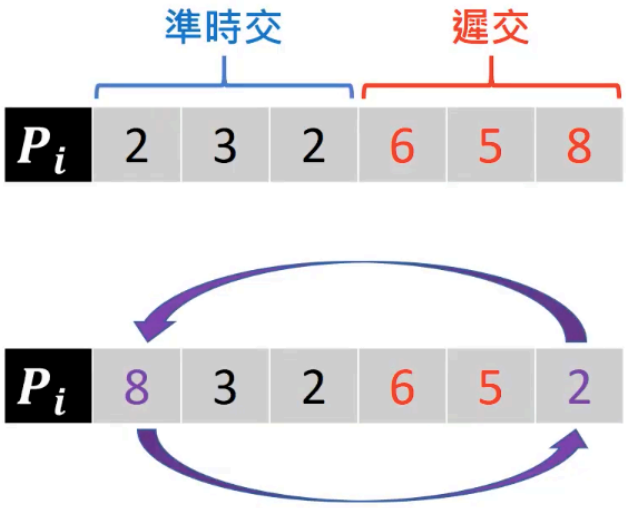
（2）更好的做法

既然這兩種做法各自都有瑕疵，就要另行找到更好的做法：觀察一下問題的特色，遲交就是遲交，一份作業一旦遲交後，不論何時交出，都是被扣固定的分數，因此若把作業分成「準時交」和「遲交」的兩組，各自組內的順序並不會影響總扣分。

這樣一來，重點就是分出哪些作業應該「準時交」，哪些應該選擇「遲交」。得到的貪婪想法可以是：從扣分最重的作業開始，盡可能準時交出。



證明這種做法能得到最佳解：假設作業 H_i 的扣分 P_i 最重，如果 H_i 不在最佳解中（不在準時交的組別中），那麼把它和目前放在期限 D_i 那一天交的作業交換，讓 H_i 進到準時交的組別中（即使可能會讓被換位置的那份作業進到遲交組中），必定可以減少總扣分。



上圖中，假設扣分最重（$P_i = 8$）的這份作業期限是第一天，那麼把它和第一天交的作業（$P_i = 2$）對調，即使會讓後面這份作業變成遲交，總扣分還是減少。

實際執行步驟如下：

A. 先把作業依照扣分 P_i 從大到小排序

B. 按照 P_i 大小，把 H_i 放到對應的 D_i 前最後一個空著的位置

- 如果從第一天到 D_i 都滿了，則代表只能選擇遲交，往最後面放

D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

S_i	1	2	3	4	5	6
D_i						
P_i						

首先，把 $P_i = 8$ 這份作業放到符合 D_i 的第四天交（更早交有可能會佔到其他作業的時間，沒有好處）。

D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

S_i	1	2	3	4	5	6
D_i				4		
P_i				8		

再來，要放 $P_i = 7$ 這份作業時，原本要放到符合它的 D_i 的第四天，但是因為那天已經被佔滿了，所以往前一天放到第三天。

D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

S_i	1	2	3	4	5	6
D_i			4	4		
P_i			7	8		

要放 $P_i = 6$ 這份作業時，因為其 $D_i = 2$，放到第二天；要放 $P_i = 5$ 的作業時，因為第三天已經被佔、第二天也被佔，因此放到第一天交。

D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

S_i	1	2	3	4	5	6
D_i	3	2	4	4		
P_i	5	6	7	8		

之後 $P_i = 3$ 的作業放到第五天，$P_i = 2$ 的作業，則因為都沒有辦法往前找到可以準時交的空位，所以從最後面開始填（既然已經遲交，那麼再早交也沒有好處）。

D_i	4	4	2	3	5	2
P_i	8	7	6	5	3	2

S_i	1	2	3	4	5	6
D_i	3	2	4	4	5	2
P_i	5	6	7	8	3	2

按照這種做法，上例中總共只需要被扣 2 分。

（3）LeetCode #621 工作排程 Task Scheduler

A. 題目

給定一個字元陣列 $tasks$ 代表 CPU 需要完成的所有任務，不同的字元代表不同類型的任務，這些任務可以被以任意順序完成。

每個任務需要一個單位時間來完成，而在任一單位時間內，CPU 可以完成某一個工作，或者什麼也不做。

另外考慮一個非負整數 n，代表兩個相同種類（用同一個字元來表示）的任務間最少要間隔多久才能進行，也就是說，同樣的兩個任務間必須至少空出 n 單位的時間。

回傳 CPU 完成所有給定任務最少需要花費的總時間。

B. 出處
https://leetcode.com/problems/task-scheduler/

C. 範例輸入與輸出
輸入：tasks = ["A","A","A","B","B","B"], n = 2
輸出：8

因為同類型的任務間至少需要間隔兩單位的時間，以下列順序進行：

$A \rightarrow B \rightarrow$ 閒置 $\rightarrow A \rightarrow B \rightarrow$ 閒置 $\rightarrow A \rightarrow B$

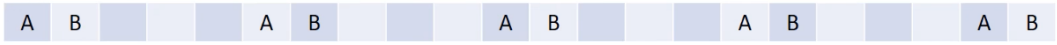
總共需要至少 8 單位的時間。

D. 解題邏輯

如果只有一種工作 A，出現次數為 5，$n = 4$

總時間
$= (4 + 1)(5 - 1) + 1$
$= (n + 1)(t - 1) + 1$
其中 t 為出現次數，$t - 1$ 代表間隔數目
$n + 1$ 是間隔長度加上執行所需的時間

如果有兩種工作 A、B，出現次數為 5，$n = 4$



總時間

$= (4 + 1)(5 - 1) + 2$

$= (n + 1)(t - 1) + 2$

$= (n + 1)(t - 1) + counts$

其中 $counts$ 代表共有幾種工作

另外一種情形，如果工作的種類很多，大於間隔時間 n，那麼中間沒有空閒，只要一直輪流進行工作即可：如果有六種工作 A、B、C、D、E、F，出現次數為 5，$n = 4$

總時間

$= 6 \times 5$

$= counts \times t$

綜合來說，需要的總時間是下列兩者間較大者

- 有空閒時：$(n + 1)(t - 1) + counts$
- 沒有空閒時：$taskAmount = counts \times t$

可以表示為 $Total = max(taskAmount, (n + 1)(t - 1) + counts)$

工作排程 Task Scheduler	
1	class Solution{
2	public:
3	int leastInterval(vector<char>& tasks, int n){
4	
5	// 最多可能有 26 種字母
6	int counts[26] = {0};
7	
8	// 記錄每個字母的出現次數

```
9         for (char c:tasks){
10              counts[c-'A']++;
11        }
12
13        // 找到出現次數最多的字母其出現次數
14        int max_frequency = -1;
15        for (int i=0 ; i<26 ; i++){
16              if (counts[i]>max_frequency){
17                    max_frequency = counts[i];
18              }
19        }
20
21        // 找到與該出現次數相同的字母總數
22        // 比如解題邏輯中，A、B 都出現 5 次的情形
23        int max_frequency_words = 0;
24        for (int i=0 ; i<26 ; i++){
25              if (counts[i]==max_frequency){
26                    max_frequency_words++;
27              }
28        }
29
30        // 即解題邏輯中的 taskAmount
31        int len = tasks.size();
32
33        // 即 (n+1)(t-1)+counts
34        int result = (n+1)*(max_frequency-1)+max_frequency_words;
35
36        return len>result ? len : result;
37
38        } // end of leastInverval
39
40  }; // end of Solution
```

7. 實戰練習

（1）LeetCode #1217. 移動籌碼到同一位置的最小成本 Minimum Cost to Move Chips to The Same Position

A. 題目

總共有 n 個籌碼，$position[i]$ 代表第 i 個籌碼的位置。

目標是移動所有籌碼到同一個位置上，每次動作可以把某個籌碼從 $position[i]$ 移動到其他位置：

a. 移動到 $position[i]+2$ 或 $position[i]-2$，成本為 0
b. 移動到 $position[i]+1$ 或 $position[i]-1$，成本為 1

回傳要把所有籌碼移動到同一個位置需要的最小成本。

B. 出處
https://leetcode.com/problems/minimum-cost-to-move-chips-to-the-same-position/

C. 範例輸入與輸出

輸入：position = [1,2,3]
輸出：1

首先，把位置 3 的籌碼移動到位置 1（cost = 0），接著，把位置 2 的籌碼移動到位置 1（cost = 1）。

D. 解題邏輯

因為移偶數格不需要任何成本，只有移奇數格時才需要，所以一開始可以將所有籌碼都集中在 1 和 2 兩個位置上（初始位置奇數者移到 1、偶數者移到 2），兩個位置的籌碼數量分別為 $num1$、$num2$。

接下來，再把籌碼較少的那個位置上的所有籌碼，都移到籌碼多的位置上即可，總成本為 $min(num1, num2)$。

也就是說，本題只需計算「初始位置在奇數的籌碼數」與「初始位置在偶數的籌碼數」，並回傳較小者即可。

移動籌碼到同一位置的最小成本 Minimum Cost to Move Chips to The Same Position	
1	class Solution{
2	public:
3	int minCostToMoveChips(vector<int>& position){
4	
5	int len = position.size();
6	int odd_sum = 0;
7	int even_sum = 0;
8	
9	// 計算奇數籌碼數與偶數籌碼數
10	// 例如 [2,3,5] 代表共有三個籌碼
11	// 第一個籌碼在 2、第二個籌碼在 3、第三個籌碼在 5
12	for (int i=0 ; i<len ; i++){
13	if (i%2==1){
14	odd_sum ++;
15	} else {
16	even_sum ++;
17	}
18	}
19	
20	// 回傳兩者中較小者

```
21              return odd_sum<even_sum ? odd_sum : even_sum;
22
23          } // end of minCostToMoveChips
24
25  }; // end of Solution
```

（2）LeetCode #1221. 分割出平衡字串 Split a String in Balanced Strings

A. 題目

平衡字串是指字串當中 'L' 和 'R' 出現次數相同者。給定一個平衡字串 s，把它切割為若干個平衡子字串，使得切割出的子字串個數最多。回傳該切割出的個數。

B. 出處

https://leetcode.com/problems/split-a-string-in-balanced-strings/

C. 範例輸入與輸出

輸入：s = "RLRRLLRLRL"

輸出：4

s 最多可以被分割為 4 個平衡字串："RL","RRLL","RL","RL"

D. 解題邏輯

因為本題目標是要切出「最多」個平衡字串，且因為原字串就是平衡字串，不會有切不出來的情形，所以從左邊開始一個個字元檢查，每當形成了一個平衡字串的時候，就把要回傳的數加一，再繼續往後檢查即可。

分割出平衡字串 Split a String in Balanced Strings

```
1   class Solution{
2   public:
3       int balancedStringSplit(string s){
```

```
4
5           int L = 0;
6           int R = 0;
7           int counts = 0;    // 平衡字串的數目
8
9           // 逐一檢查 s 中的字元
10          for (char c:s){
11              if (c=='L'){
12                  L++;
13              } else {
14                  R++;
15              }
16
17              // 目前 L 與 R 個數相同時，counts 加一
18              if (L==R && L){
19                  L=R=0;
20                  counts++;
21              }
22          }
23
24          return counts;
25
26      } // end of balancedStringSplit
27
28  }; // end of Solution
```

（3）LeetCode #1710. 裝載最多單位 Maximum Units on a Truck

A. 題目

本題要把一些盒子放到卡車上，給定一個陣列 $boxTypes$，
$boxTypes[i] = [numberOfBoxes_i, numberOfUnitsPerBox_i]$，其中

a. $numberOfBoxes_i$ 是這種盒子的總數
b. $numberOfUnitsPerBox_i$ 是每個這種盒子中裝有的單位數

另外，有一個整數 $truckSize$ 是卡車上最多可以放的盒子數，只要還沒超過這個上限值，可以繼續選擇任一種盒子放到卡車上。

回傳可以放到卡車上的最大「單位」數。

B. 出處
https://leetcode.com/problems/maximum-units-on-a-truck/

C. 範例輸入與輸出
輸入：boxTypes = [[1,3],[2,2],[3,1]], truckSize = 4
輸出：8

可以拿第一種盒子一個、第二種盒子兩個、第三種盒子一個，總單位數為：
$3(\text{單位}) \times 1 + 2(\text{單位}) \times 2 + 1(\text{單位}) \times 1 = 8$

D. 解題邏輯

因為每種盒子一個都只佔盒子上限數中的「1」，所以本題解法同 Fractional Knapsack Problem，從內含單位數最多的盒子種類開始放即可。

裝載最多單位 Maximum Units on a Truck	
1	// 依照 $numberOfUnitsPerBox_i$ 從大到小排序
2	bool cmp(vector<int>& box_1, vector<int>& box_2){
3	return box_1[1] > box_2[1];
4	}

```
5
6   class Solution{
7   public:
8       int maximumUnits(vector<vector<int>>& boxTypes, int truckSize){
9
10          sort(boxTypes.begin(), boxTypes.end(), cmp);
11          int total = 0;
12
13          // 從內含單位數最多的箱子種類開始處理
14          for (vector<int>& box:boxTypes){
15
16              if (truckSize==0){ break; }
17
18              // 取出目前這種箱子的資料
19              int number = box[0];
20              int capacity = box[1];
21
22              // 可以把目前這種盒子全拿的情形
23              if (truckSize>=number){
24                  truckSize -= number;
25                  total += number*capacity;
26              }
27              // 剩餘空間不夠全拿的情形
28              else {
29                  total += truckSize*capacity;
30                  truckSize = 0;
31              }
32
33          } // end of for
34
35          return total;
36
37      } // end of maximumUnits
38  }; // end of Solution
```

（4）LeetCode #1094. 共享乘車 Car Pooling

A. 題目

有一輛共可搭載 $capacity$ 位乘客的車，這輛車只會往東邊開（不會轉彎，也不往西邊開）。

給定一個陣列 $trips$，其中

$$trips[i] = [num\_passengers, start\_location, end\_location]$$

代表第 i 個接駁處有幾位乘客搭車，以及這幾位乘客的上下車地點。地點是以距離出發地東邊的公里數表示。只有在可以成功搭載所有乘客並讓他們在目的地下車時，回傳 true。

B. 出處

https://leetcode.com/problems/car-pooling/

C. 範例輸入與輸出

輸入：trips = [[2,1,5],[3,3,7]], capacity = 4

輸出：false

因為在座標 3 時上車的乘客有三位，但是車上的空位只剩下兩個，所以回傳 false。

D. 解題邏輯

從起點出發，檢查整趟旅程中乘客數量是否有超過 capacity 的情況即可。

	Location									
1	5	5	5	5	5					
2						3	3	3		
3								4	4	4
4					2	2	2			
5		1	1	1						

舉個例子，可以把某個測資 $trips$ 中的每一筆資料可以用上圖表示，如 $trips[1]$ 的內容是有 5 名乘客在東邊 0 公里處上車，東邊 5 公里處下車。

	Location									
1	+5				-5					
2						+3		-3		
3								+4		-4
4					+2		-2			
5		+1		-1						

把資料改為上圖的形式處理，計算每個點的乘客「增減數」，並與到達該點時車上剩餘的座位數比較，以判斷是否所有乘客都能上車。

共享乘車 Car Pooling

```
1  class Solution{
2  public:
3      bool carPooling(vector<vector<int>>& trips, int capacity){
4
5          // 記錄每個座標位置的乘客「變動數」
6          // 1001 是 LeetCode 給的測資限制
7          vector<int> passenger_change(1001,0);
8
9          // 從每筆 trip 資料中，找到乘客會變動的座標
10         // 乘客上車時，用正數標記該點乘客會「增加」幾人
11         // 乘客下車時，用負數標記
12         for (vector<int>& trip:trips){
13             int num_passengers = trip[0];
14             int start_location = trip[1];
15             int end_location = trip[2];
16             passenger_change[start_location] += num_passenger;
17             passenger_change[end_location] -= num_passengers;
18         }
19
20         // 透過 passenger_change 向量
```

```
21              // 檢查過程中是否超過車的座位數 capacity
22              for (int num:passenger_change){
23                  // passenger_change 是正數時（乘客上車），capacity 減少
24                  // passenger_change 是負數時（乘客下車），capacity 增加
25                  capacity -= num;
26
27                  // 座位不夠時，回傳 false
28                  if (capacity<0){
29                      return false;
30                  }
31              }
32
33              return true;
34
35          } // end of carPooling
36
37  }; // end of Solution
```