Ch8 動態規劃 Dynamic Programming

本章要介紹的是「動態規劃」的演算法精神。

首先，一樣會簡介動態規劃是什麼，以及哪些狀況下適用動態規劃。

接著，會用動態規劃的想法解答幾個之前已經探討過的問題：

A. 費波那契數列
B. 找錢問題
C. 最大子數列問題
D. 活動選擇問題

接下來，再看動態規劃的幾種常見應用：

A. 郵票問題
B. 切割問題
C. 背包問題

最後總結動態規劃後，再做幾題實戰練習。

1. 動態規劃簡介

與分治法類似，動態規劃會把原始問題，也就是「母問題」切割成許多「子問題」，之後，再利用子問題的答案組成母問題的答案。

跟分治法不同的地方在於動態規劃中，每個子問題通常是環環相扣的，因為大部分子問題的答案會用到其他子問題的答案，所以過程中，每次解完一個子問題，都會把得到的答案記錄下來（分治法則不進行紀錄）。

這樣的策略是「以空間換取時間」，因為要記錄每個子問題的答案，會耗費許多記憶體空間，但是一旦記錄，之後要用到該結果就不用重算，也就節省了相對更珍貴的 CPU 資源。「凡走過必留下痕跡」就是動態規劃的寫照。

（1）費波那契數列

以費波那契數列為例，比較動態規劃與分治法：

費波那契數列 [動態規劃]

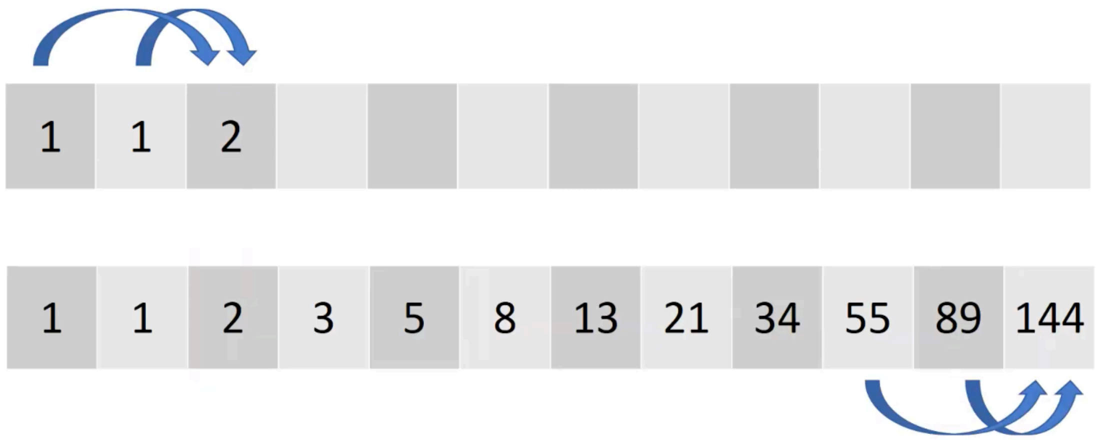
```
int fibo(int n){

	// 宣告儲存資料的陣列與初始化
	int *data = (int*) malloc(sizeof(int)*n);
	data[0] = 1;
	data[1] = 1;

	// 運用已儲存的資料得到數列中新的數
	for (int i=2 ; i<n ; i++){
		data[i] = data[i-1] + data[i-2];
	}

	int result = data[n-1];
	free(data);
	return result;
}
```

費波那契數列 [分治法（遞迴）]

```
int fibo(int n){
	if (n<=2){
		return 1;
	} else {
		return fibo(n-1)+fibo(n-2);
	}
}
```

因為分治法中不會記錄過程中計算出的各個費波那契數，所以存在重複運算的問題；動態規劃則會把每個子問題的答案都記錄下來，用這些答案去拼湊出其他子問題的答案，直到解答原問題。

用動態規劃求解特定費波那契數時，因為前面算過的費波那契數都被記錄下來了，所以任一個費波那契數從頭到尾都只會計算一次，且皆以已在記錄中的前兩個費波那契數求和就可以得到。



（2）修改版費波那契數列

已知有一函式 $f(n)$ 修改自費波那契數列，可以下式表示：

$$f(n) = \begin{cases} 1, if\ n <= 3 \\ f(n-1) + f(n-2) + f(n-3), if\ n > 3 \end{cases}$$

請找出 $f(30)$ 的值。

修改版費波那契數列

```
#include <iostream>
using namespace std;

int main()
{
        long long int triple_fibo[30];

```

```
	for (int i=0 ; i<30 ; i++){

		// 根據定義，前三項為 1
		if (i<3)
			triple_fibo[i] = 1;
		else
			triple_fibo[i] = triple_fibo[i-1] + triple_fibo[i-2] +
			triple_fibo[i-3];

	}

	cout << triple_fibo[29] << endl;

	return 0;
}
```

讀者可自行測試遞迴版的修改版費波那契數，會發現當數字較大時，因為存在重複運算，使用的時間會很長。

（3）LeetCode #746. 爬階梯的最小成本 Min Cost Climbing Stairs

A. 題目

給定一個整數陣列 $cost$，$cost[i]$ 是踏到階梯上第 i 階後需付的成本，付出該成本後，就可以選擇再往上爬一或二階。

你可以選擇從 $index\ 0$ 這階開始爬階梯，或者從 $index\ 1$ 這階開始。回傳要爬完整個階梯到達上方的最小成本。

B. 出處
https://leetcode.com/problems/min-cost-climbing-stairs/

C. 範例輸入與輸出
輸入：cost = [10,15,20]

輸出：15
最小成本的方法是從第一階（$cost[1]$）開始，付出 15 的成本，然後向上爬兩階到階梯上方。

輸入：cost = [1,100,1,1,1,100,1,1,100,1]
輸出：6
從第一階開始，並且只踏除了 $cost[3]$ 以外成本為 1 的那幾階。

D. 解題邏輯

先開出一個陣列，記錄前兩階的成本：

1	100	1	1	1	100	1	1	100	1

1	100								

要到達第三階的最小成本，會是「先踏第一階，然後直接踏到第三階」和「先踏第二階，然後踏到第三階」兩者中成本較低者，用數學式表示，就是：

$$Step[2]$$
$$= min(Step[0] + cost[2], Step[1] + cost[2])$$
$$= min(Step[0], Step[1]) + cost[2]$$
$$= min(1,100) + 1 = 2$$

1	100	1	1	1	100	1	1	100	1

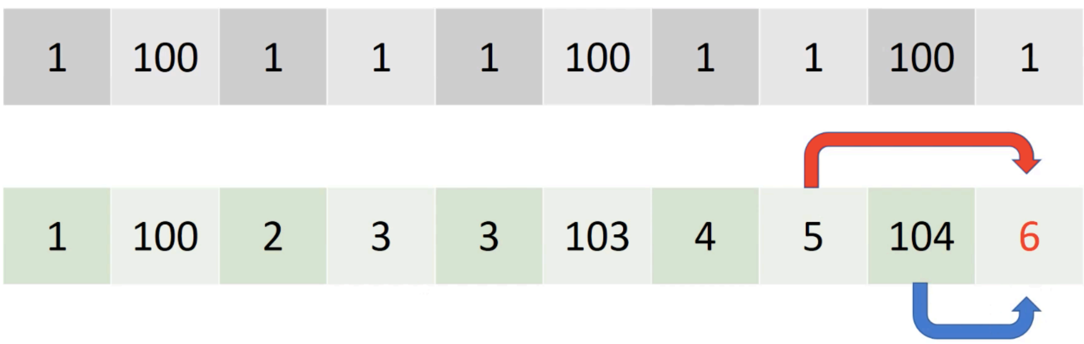
1	100	2							

要到達第四階的最小成本，是「到了第二階後，直接踏兩階到第四階」和「到

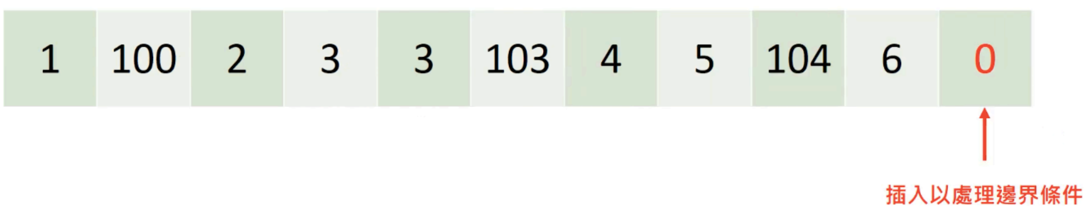
了第三階後，再踏一階到第四階」，因為前面已經得到了從起點開始，到第二階和到第三階分別的最小成本，因此可以列出：

$$
\begin{aligned}
&Step[3] \\
&= min(Step[1], Step[2]) + cost[3] \\
&= min(100,2) + 1 \\
&= 3
\end{aligned}
$$

以此類推，可以得到接下來每一階的最小成本，只要比較它的前兩階，得到較小者後，再加上該階自己的成本即可。



為了避免處理邊界條件，可以在階梯最上方插入成本為 0 的一階，用上面方法算出到達該格的最小成本，就相當於得到「走到整個階梯上方」的最小成本。



爬階梯的最小成本 Min Cost Climbing Stairs

```
class Solution{
public:
```

```
    int minCostClimbingStairs(vector<int>& cost){

        int len = cost.size();

        // 記錄到達每一階的最小成本
        // 因為在最後面加上一階，所以長度是 len+1
        vector<int> Step(len+1);

        // 前兩階的最小成本直接取該階需付的成本
        Step[0] = cost[0];
        Step[1] = cost[1];

        // 把加在最後面的一階成本設為 0
        cost.push_back(0);

        for (int i=2 ; i<len ; i++){
            // 選擇前兩階中成本較小者，再加上該階自己的成本
            Step[i] = (Step[i-1]<Step[i-2] ? Step[i-1] : Step[i-2]) + cost[i];
        }

        return Step[len];

    } // end of minCostClimbingStairs

}; // end of Solution
```

上面這種做法的空間複雜度是 $O(n)$，因為需要記錄每一階的資料，不過觀察到「每次只需要前兩階的成本即可算出新的一階的成本」，因此也可以改寫程式碼為：

爬階梯的最小成本 [改良版]

```
class Solution{
public:
    int minCostClimbingStairs(vector<int>& cost){
```

```
        int len = cost.size();
        int step_1 = cost[0];
        int step_2 = cost[1];
        int step_3;
        cost.push_back(0);

        for (int i=2 ; i<=len ; i++){
            step_3 = step_1<step_2 ? step_1 : step_2;
            step_3 += cost[i];

            // 接下來，更新 step_1 和 step_2 的值
            // 使得算下一階時，能用到新的 step_1 和 step_2
            // step_2 -> step_1
            step_1 = step_2;
            // step_3 -> step_2
            step_2 = step_3;
        }

        // 回傳的是迴圈執行完後 step_3 的值
        return step_3;

    } // end of minCostClimbingStairs

}; // end of Solution
```

這樣一來，只需要三個變數就可以解決整個問題，成功降低了空間複雜度。

2. 動態規劃解析

做完上一節的兩題例題後，對動態規劃應該有了基本的認識。接著，來檢視一下哪些問題適用動態規劃。

（1）動態規劃的適用條件

首先，這些問題應該有「最佳化子結構」，也就是說，母問題能夠被切割成數個子問題，子問題的答案分別都能夠被推算出來，且能拼湊出母問題的答案。

再來，這些子問題間通常是「重疊的」。在求解一個子問題時，會用到其他子問題的答案，所以每算出一個子問題的答案，都要記錄下來才能避免重複運算。

最後，還必須符合「無後效性」：每次解決子問題時，該子問題的解答只跟之前求解過的子問題有關，而不會用到還未求解的子問題答案。比如費波那契數列中，每個數只和前兩個數有關，而與後面的數無關，符合「不使用後面子問題答案」的要求。

一些動態規劃中常用的名詞：

A. 狀態：切割後，每個子問題的特性、資料或解答
B. 階段：把性質類似且可同時處理的狀態集合在一起
C. 決策：每個階段下的選擇
D. 狀態轉移方程式：由若干子問題的狀態（答案），求出另一子問題的狀態（答案）

（2）動態規劃的複雜度

A. 時間複雜度：通常為狀態總數 × 狀態轉移方程式的複雜度
B. 空間複雜度：未來可能被使用，而需記錄下的所有狀態數目

（3）狀態轉移方程式

狀態轉移方程式代表如何「透過若干子問題的答案」來得到「另一個子問題的答案」。

比如在前面的兩個例題中，費波那契數列中的一個數的答案，是由前兩個數的答案加總而成；爬階梯的例題中，爬到某一階的最小成本則是前兩階中較小者，加上該階成本而得。

A. 費波那契數列

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

B. 爬階梯的最小成本

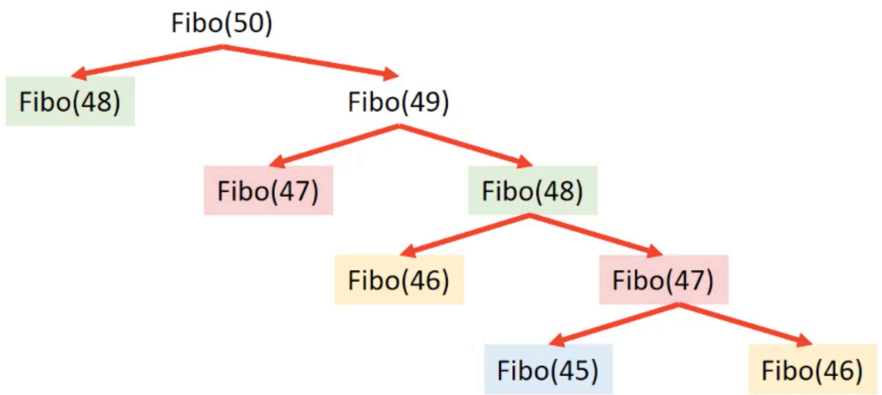
$$Step[i] = min(Step[i-2], Step[i-1]) + cost[i]$$

在解動態規劃的題目時，最重要的就是找出狀態轉移方程式，找到後，配合給定或可以很容易推導出的前幾個子問題的答案，就很容易解決整個問題。

（4）動態規劃求解的兩種方式

A. Top-down with memorization：從母問題開始切割出子問題後往下解，但每次得到子問題的答案後需記錄下來。

a. 通常用遞迴來解
b. 每次呼叫函式需佔用記憶體
c. 較耗費記憶體空間

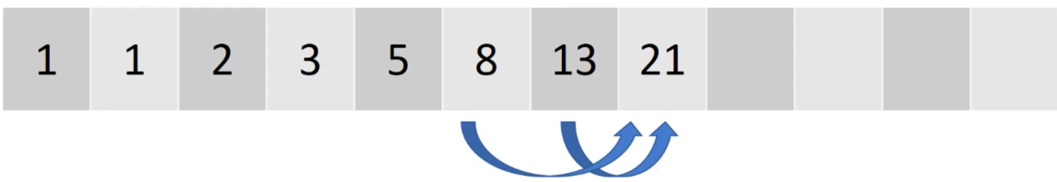


Top-down 動態規劃解費波那契數列

```
int Fibo_Top_Down(int n){
    if (n<=2){
        // n-1 是為了調整為索引值
        Fibo[n-1] = 1;
    }
    // Fibo[n-1] 值為 0，代表該索引值還沒被算過
    // 只有在值為 0 時才計算，避免重複進行
    else if (Fibo[n-1]==0){
        Fibo[n-1] = Fibo_Top_Down(n-1) + Fibo_Top_Down(n-2);
    }

    // Fibo[n-1] 之前已經有算過時，不需計算就能直接回傳
    return Fibo[n-1];
}
```

B. Bottom-up method：從小的子問題開始算到大的，過程中用陣列記錄每個子問題的答案，是比較常使用到的方式。

a. 通常用迭代來解決
b. 把狀態存在陣列中



Bottom-up 的做法利用陣列，從最小的子問題開始計算：

Bottom-up 動態規劃解費波那契數列

```
int Fibo_Bottom_Up(int n){
    Fibo[0] = Fibo[1] = 1;
    for (int i=2 ; i<n ; i++)
        Fibo[i] = Fibo[i-1] + Fibo[i-2];
    return Fibo[n-1];
}
```

3. 找錢問題

接下來，就可以試著解答講解貪婪演算法時並沒有順利解出來的「找錢問題」:

需要找給某個顧客 20 元，而目前有三種硬幣：

a. 1 元、b. 5 元、c. 8 元

如何找錢可以讓找的硬幣「個數」最少？

上面這個例題使用貪婪演算法會得到錯誤的結果（2 個 8 元和 4 個 1 元，共 6 個硬幣），而無法得到正確結果「使用 4 個 5 元」，這是因為 8 元不像 10 元一樣總是可以用若干個 5 元替代。

（1）動態規劃求解找錢問題

那麼，動態規劃會如何著手解決這個問題呢？假設需要找 x 元時，至少需要 $Coin(x)$ 枚硬幣。

已知：

$Coin(1) = 1$ // 一枚 1 元

$Coin(2) = 2$ // 兩枚 1 元

$Coin(3) = 3$ // 三枚 1 元

$Coin(4) = 4$ // 四枚 1 元

$Coin(5) = 1$ // 一枚 5 元

$Coin(6) =$?

因為要找 6 元必定是用 1 元和 5 元來找，所以可以假定最後一枚使用的是 1 元或是 5 元兩種情形：

A. 先找到找 5 元的方法，最後加上一枚 1 元來找給顧客 6 元：

$Coin(5) + 1$（枚）

B. 先找到找 1 元的方法，最後加上一枚 5 元來找給顧客 6 元：

$Coin(1) + 1$（枚）

除了這兩種情形以外，沒有其他方法可以找出 6 元，所以找 6 元的最低枚數一定是 $Coin(1)$ 和 $Coin(5)$ 兩者中枚數較少者，再加上 1（最後那枚硬幣）。

也就是說，$Coin(6) = min(Coin(5)+1, Coin(1)+1)$

一般化的情形，要找出 x 元，當 x 大於 8 時，最後加上的一枚硬幣可能是 1 元、5 元或 8 元：

A. 先找 $x-1$ 元，再加上一枚 1 元：$Coin(x-1)+1$（枚）
B. 先找 $x-5$ 元，再加上一枚 5 元：$Coin(x-5)+1$（枚）
C. 先找 $x-8$ 元，再加上一枚 8 元：$Coin(x-8)+1$（枚）

因此，可以列出狀態轉移方程式：

$$
\begin{aligned}
&Coin(x) \\
&= min(Coin(x-1)+1, Coin(x-5)+1, Coin(x-8)+1) \\
&= 1 + min(Coin(x-1), Coin(x-5), Coin(x-8))
\end{aligned}
$$

也就是說，當已經得到 $Coin(x-1)$、$Coin(x-5)$、$Coin(x-8)$ 三者的值時，就可以立刻決定出 $Coin(x)$ 的值。

實作上，首先開出一個陣列，每筆資料代表 index+1 元需要幾枚硬幣（index 0 代表一元、index 1 代表兩元），並且填入 1 到 5 元需要的硬幣枚數，如前所述。

接下來，使用狀態轉移方程式來得到 6 元以上需要的硬幣枚數，注意 $Coin(6) = 1 + min(Coin(5), Coin(1), Coin(-2))$，其中 $Coin(x-8)$ 產生的 $Coin(-2)$ 未定義，因此只考慮另外兩項何者較小。

$Coin(8) = 1 + min(Coin(7), Coin(3), Coin(0))$，其中 $Coin(0)$ 代表「沒有錢需要找時」要使用到的硬幣枚數，其值為「0」。

$\boldsymbol{Coin(8) = 1 + min(Coin(7), Coin(3), Coin(0))}$

1	2	3	4	1	2	3	1												

接下來，從 $Coin(9)$ 開始，要比較的三個值都已經在儲存結果的陣列中，因此可以透過狀態轉移方程式直接得到所求值。

$\boldsymbol{Coin(9) = 1 + min(Coin(8), Coin(4), Coin(1))}$

1	2	3	4	1	2	4	3	2											

（2）Try LeetCode #322. 找錢問題 Coin Change

A. 題目

給定一個整數陣列 $coins$ 代表不同面額的硬幣，與一個整數 $amount$ 代表要找的金額。回傳要組成該金額最少需要的硬幣枚數，如果給定的硬幣面額無法組成該要找的金額，回傳 -1。假設每種硬幣使用的數量不受限制。

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

使用兩枚 5 元與一枚 1 元，就可以找出 11 元。

找錢問題 Coin Change

```
class Solution{
public:

    int coinChange(vector<int>& coins, int amount){
```

```

    // 儲存 n 元需要最少硬幣枚數的陣列
    // 初始值設定為 -1，用來判斷是否算過
    vector<int> min_coin(amount+1, -1);
    min_coin[0] = 0;

    // 從 n=1 開始往後算到 n=amount
    for (int i=1 ; i<=amount ; i++){
        // min_coin[i] =
        // min(
        //     min_coin[i-coins[0]],
        //     min_coin[i-coins[1]],
        //     min_coin[i-coins[2]],
        //     ...
        // ) + 1

        // int_max
        int min = 2147483647;

        // 目前這個 min_coin[current] 的值
        int coin_now;

        for (int coin:coins){
            // 除了最後一枚外，要找的金額
            int current = i-coin;

            // i-coin < 0
            // min_coin[current] 中 current 是一個負數時
            // coin_now 設為最大值，確保 min 不被改變
            if (current<0){
                coin_now = 2147483647;
            }

            // i-coin >=0
```

```
                else {

                    // 如果除去最後那枚硬幣後要找的金額
                    // 也沒辦法被找開
                    if (min_coin[current]==-1){
                        coin_now = 2147483647;
                    }
                    // 除去最後那枚硬幣後要找的金額可以被找開時
                    else{
                        coin_now = min_coin[current];
                    }
                }

                min = coin_now<min ? coin_now : min;
            }

            // 如果 min 還是初始值，代表無法找開，回傳 -1
            if (min == 2147483647){
                min_coin[i]=-1;
            } else {
                // 否則，min_coin[i] 是 min 枚硬幣加上最後那枚
                min_coin[i] = min+1;
            }
        } // end of for

        return min_coin[amount];

    } // end of coinChange

}; // end of Solution
```

4. 最大子數列

接下來是在分治法章節中曾經提到過的最大子數列問題：

給定一個陣列，當中的值有正有負，請找出一區間 [a,b]，使得區間內的元素總和最大，並回傳該元素總和。

之前提過，暴力解複雜度高達 $O(n^3)$，使用分治法後，複雜度降到 $O(n\log_2 n)$。

（1）動態規劃求解最大子數列

使用動態規劃來解這個問題，可以進一步改善時間複雜度：

從陣列的開頭一個一個元素處理，在新開出的陣列當中，索引值 index 的這個位置要放的是「原陣列中，從這個元素開始往左擴張，最大的連續子數列可以達到的值」。

每次處理到原陣列中一個新的元素時，根據策略，該位置得到的值一定會包含自己，因此有兩種情形：

A. 只取目前這個元素自己的值
B. 取目前元素自己的值，加上往左延伸可以產生的最大連續子數列和

其中，「往左延伸可以產生的最大連續子數列和」其實就是「處理上一個元素時算出的值」，狀態轉移方程式為：

$$DP[i] = max(DP[i-1] + Data[i], Data[i])$$

可以觀察到，如果往左延伸只能得到一個負的和，那麼乾脆不要往左延伸，只取現在這個新處理元素自己的值就好了。

（2）求解最大子數列的進行過程

-8	-2	3	4	-3	6	-3	4	-3	1

上圖中，第一個元素是 -8，這時只有一種選擇，必須選 -8。處理到第二個元素時，有兩種選擇：只取自己的值 -2，和延伸到左邊加上 -8，顯然因為 -8 是負值，對於總和沒有幫助，所以此時只取 -2。

處理到第三個元素時，同樣有兩種選擇：只取自己的值 3，和往左邊去延伸出「連續」數列，既然要往左延伸，必定會包含 -2，而由剛才處理 -2 的結果，發現包含 -2 以左的子數列中，最好的值是 $-2 < 0$，對總和沒有幫助，所以也取 3 即可。

處理到第四個元素時，有兩種選擇：只取自己的值 4，和往左延伸出連續數列，往左延伸時必定會包含 3，而由處理 3 時的結果，發現 3 以左的子數列中，「最好」的值是 $3 > 0$，會使總和增加，因此這時選擇從 4 往左延伸（而不只是取自己的值 4），可以得到最大值 7。

依此類推，處理 -3 時，得到 4。處理 6 時，得到 10。接下來依序得到：7、11、8、9。

-8	-2	3	4	-3	6	-3	4	-3	1

$$\boldsymbol{DP}[0] = \boldsymbol{max}(\boldsymbol{DP}[-1] + \boldsymbol{Data}[0], \boldsymbol{Data}[0])$$

-8									

$$\boldsymbol{DP}[1] = \boldsymbol{max}(\boldsymbol{DP}[0] + \boldsymbol{Data}[1], \boldsymbol{Data}[1])$$

-8	-2								

$$\boldsymbol{DP}[2] = \boldsymbol{max}(\boldsymbol{DP}[1] + \boldsymbol{Data}[2], \boldsymbol{Data}[2])$$

-8	-2	3							

$$\boldsymbol{DP}[3] = \boldsymbol{max}(\boldsymbol{DP}[2] + \boldsymbol{Data}[3], \boldsymbol{Data}[3])$$

-8	-2	3	7						

$$\boldsymbol{DP}[4] = \boldsymbol{max}(\boldsymbol{DP}[3] + \boldsymbol{Data}[4], \boldsymbol{Data}[4])$$

-8	-2	3	7	4					

-8	-2	3	4	-3	6	-3	4	-3	1

$$\boldsymbol{DP}[5] = \boldsymbol{max}(\boldsymbol{DP}[4] + \boldsymbol{Data}[5], \boldsymbol{Data}[5])$$

-8	-2	3	7	4	10				

$$\boldsymbol{DP}[6] = \boldsymbol{max}(\boldsymbol{DP}[5] + \boldsymbol{Data}[6], \boldsymbol{Data}[6])$$

-8	-2	3	7	4	10	7			

$$\boldsymbol{DP}[7] = \boldsymbol{max}(\boldsymbol{DP}[6] + \boldsymbol{Data}[7], \boldsymbol{Data}[7])$$

-8	-2	3	7	4	10	7	11		

$$\boldsymbol{DP}[8] = \boldsymbol{max}(\boldsymbol{DP}[7] + \boldsymbol{Data}[8], \boldsymbol{Data}[8])$$

-8	-2	3	7	4	10	7	11	8	

$$\boldsymbol{DP}[9] = \boldsymbol{max}(\boldsymbol{DP}[8] + \boldsymbol{Data}[9], \boldsymbol{Data}[9])$$

-8	-2	3	7	4	10	7	11	8	9

上面算出的新陣列中，最大值為 11，出現在處理第 8 個元素 4 的時候，這代表最大的子數列是從 4 開始往左延伸，一直延伸到第三個元素 3，而不再往左，因為在處理 3 時，當初是選擇「只取自己的值，不往左延伸」。

因為要算出新陣列中的每個值只需要 $O(n)$ 時間，而找出新陣列中最大值同樣也需要 $O(n)$，因此時間複雜度為：

$$O(n) + O(n) = O(2n) = O(n)。$$

（3）LeetCode #53. 最大子數列 Maximum Subarray

A. 題目

給定一個整數陣列 $nums$，找到最大子數列（至少包含一個元素）使得其和為最大，並回傳該和。

B. 出處

https://leetcode.com/problems/maximum-subarray/

C. 範例輸入與輸出

輸入：nums = [-2,1,-3,4,-1,2,1,-5,4]
輸出：6

子數列 [4,-1,2,1] 有最大和 6

最大子數列 Maximum Subarray

```
class Solution{
public

    int maxSubArray(vector<int>& nums){

        int len = nums.size();
        // 例外處理：只有一個元素時，回傳該元素值
        if (len==1)
            return nums[0];

        // 開出新陣列 DP
        vector<int> DP(len);
        // DP[0] 與第一個元素的值相同
        DP[0] = nums[0];

        // 算出 DP 陣列中的值
        for (int i=1 ; i<len ; i++){
            // DP 陣列中前一個值
            int before_now = DP[i-1];
            // 目前處理的新元素值
            int now = nums[i];
            // 目前處理的元素往左延伸可以得到的最大值
            int add_before = now + before_now;
            // 決定是否往左延伸
            DP[i] = add_before>now ? add_before : now;

        }

```

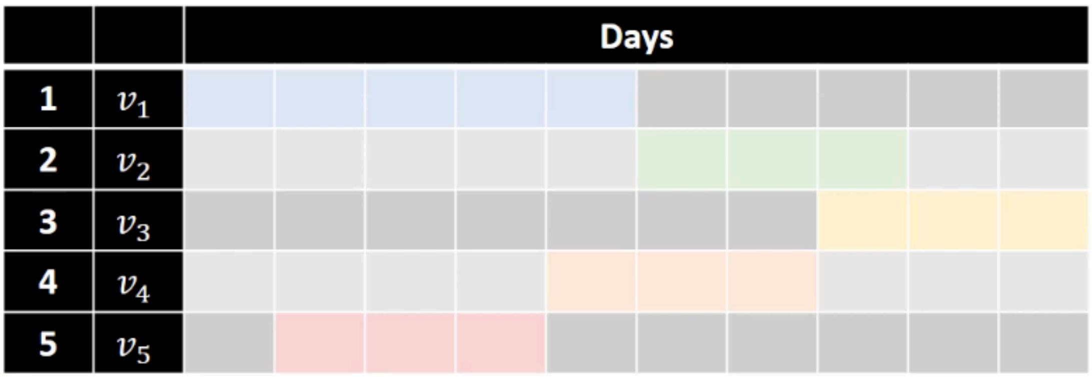
```
        // min_int
        int max = -2147483648

        // 找出 DP 陣列中的最大值
        for (int i=0 ; i<len ; i++){
            max = max>DP[i]?max:DP[i];
        }

        return max;

    } // end of maxSubArray

}; // end of Solution
```

5. 活動選擇問題

接下來是介紹貪婪演算法時曾經探討過的「活動選擇問題」:

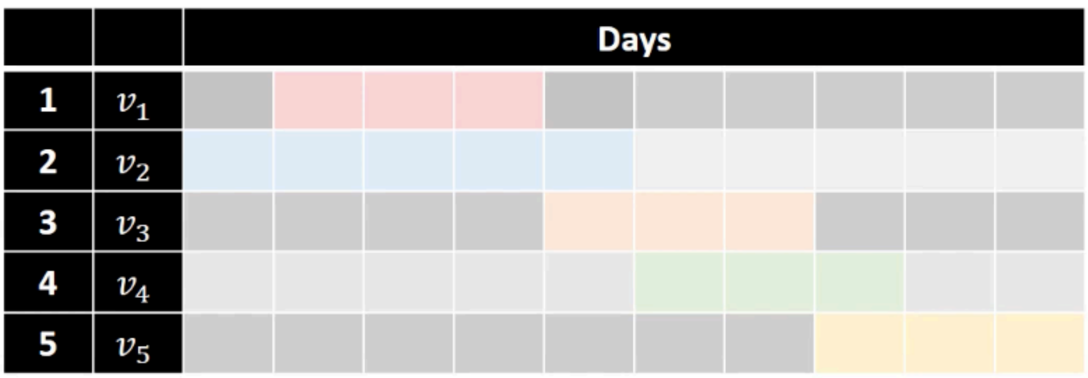
學校只有一個音響，但是 n 個活動每個都需要使用它，給定每個活動的時間長度，如何選擇舉辦哪些活動，可以使舉辦的活動「數目」最多？

利用貪婪演算法設計出的解法，是比較每個活動的結束時間，當兩個活動時間重疊而必須擇一時，總是選擇較早結束的那個。但是當活動的權重（重要性）有別時，貪婪演算法就不能得到正確答案（使得選擇出的活動權重和最高），必須改用動態規劃。



（1）用動態規劃解「有權重的」活動選擇問題

首先，仍然要把活動依照「結束日期」進行排序。



接下來要對每個活動進行處理，以算出第 1 到 n 天之間可以得到的最大權重總和 $wis(n)$。針對活動 5（第 8 天開始，第 10 天結束），有兩種做法：

A. 選擇活動 5

$$wis(10) = wis(7) + v_5$$

B. 不選擇活動 5

$$wis(10) = wis(9)$$

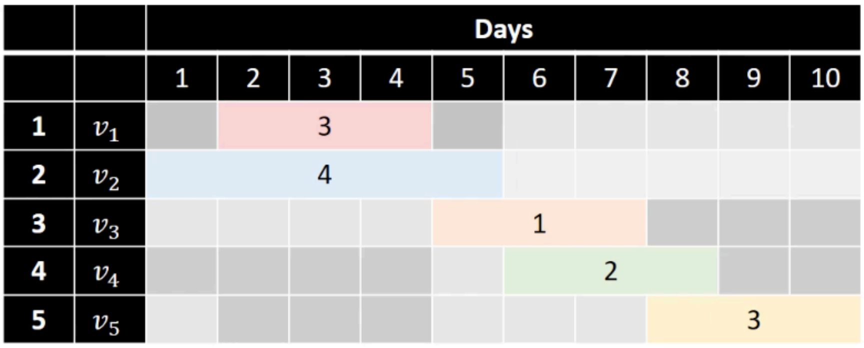
如果選擇活動 5，那麼第 1 到 10 天間能夠得到的最大權重和就是第 1 到 7 天間能夠得到的最大權重和（因為第 8、9、10 天被活動 5 佔用了），再加上舉辦活動 5 而增加的權重。

反之，如果不選擇活動 5，那麼第 1 到 10 天間能夠得到的最大權重和與第 1 到 9 天間能得到的最大權重和相同，因為上面給定的例子中，並不存在單單只花第 10 天一天就可以舉辦的活動。

由此，可以列出狀態轉移方程式如下：

$$wis(f_i) = max(wis(s_i) + v_i, wis(f_i - 1))$$

其中，s_i 是第 i 個活動的開始時間，f_i 是第 i 個活動的結束時間。

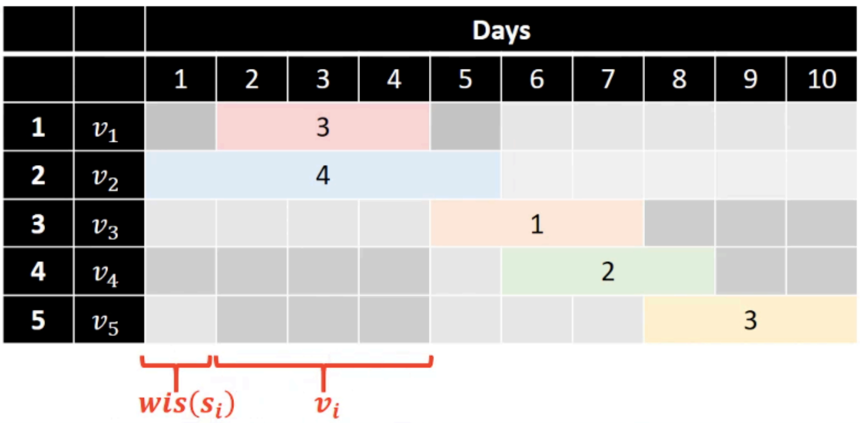


Days	1	2	3	4	5	6	7	8	9	10
Value	0	0	0							

針對上例，動態規劃的過程如下，首先，「只有第 1 天」、「只有第 1~2 天」、「只有第 1~3 天」都不能舉辦任何活動，因此開出的陣列 wis 中，前三個值都為 0。

接下來，加上第 4 天後，就有「舉辦活動 1（2~4 天的那個活動）」和「不舉辦活動 1」兩個選項，得到的權重和分別為 $wis(1)+3$ 和 $wis(3)$：

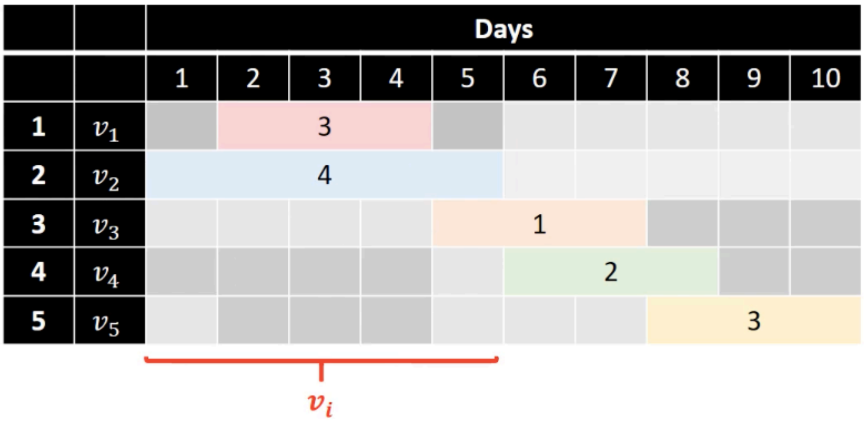
$$\begin{aligned} wis(4) &= max(wis(1)+3, wis(3)) \\ &= max(0+3, 3) = 3 \end{aligned}$$



Days	1	2	3	4	5	6	7	8	9	10
Value	0	0	0	3						

加上第 5 天後，同樣有「舉辦活動 2（1~5 天）」和「不舉辦活動 2」兩種選擇，得到的結果為：

$$wis(5) = max(wis(0) + 4, wis(4))$$
$$= 4$$

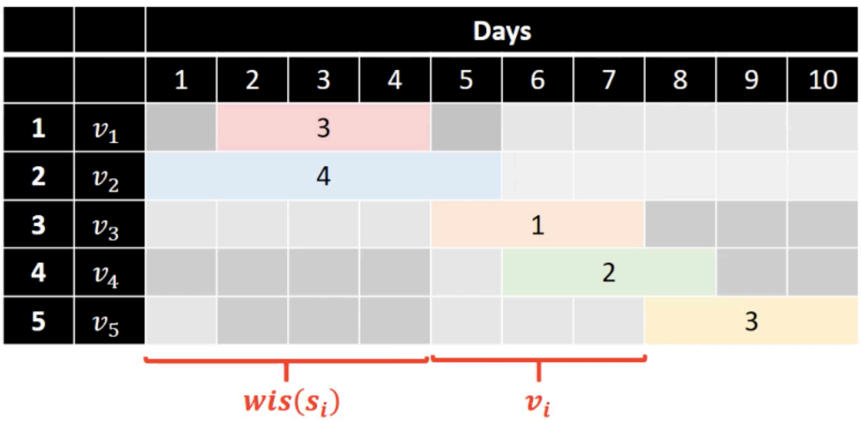


Days	1	2	3	4	5	6	7	8	9	10
Value	0	0	0	3	4					

因為沒有活動在第 6 天結束，所以 $wis(6) = wis(5) = 4$。

第 7 天則要選擇是否舉辦活動 3：

$$wis(7) = max(wis(4) + 1, wis(6))$$
$$= max(3 + 1, 4) = 4$$



Days	1	2	3	4	5	6	7	8	9	10
Value	0	0	0	3	4	4	4			

依此類推，可以得到 $wis(8) = 6$（舉辦活動 4）、$wis(9) = wis(8) = 6$、$wis(10) = max(4+3,6) = 7$（舉辦活動 5）。

Days	1	2	3	4	5	6	7	8	9	10
Value	0	0	0	3	4	4	4	6	6	7

（2）LeetCode #1235. 最大收益的工作排程 Maximum Profit in Job Scheduling

A. 題目

給定 n 項工作，每個工作開始時間與結束時間分別為 $startTime[i]$ 到 $endTime[i]$，收益為 $profit[i]$。

給定三個陣列 $startTime$、$endTime$ 與 $profit$，回傳工作間佔用時間不重疊的前提下，能夠達到的最大收益。

如果一個工作結束時間是 X，允許同時選擇一個開始時間是 X 的工作。

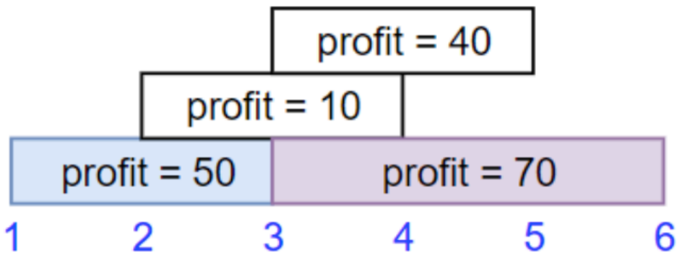
B. 出處

https://leetcode.com/problems/maximum-profit-in-job-scheduling/

C. 範例輸入與輸出

輸入：startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

輸出：120



選擇活動 1（[1,3]）和活動 4（[3,6]）時，有最大收益 $50 + 70 = 120$。

D. 解題邏輯

注意本題的測資中，有時會有數個工作結束時間相同，因此狀態轉移方程式中要加上「同樣結束時間的其他工作」：

$$wis(f_i) = max(wis(s_i) + v_i, wis(f_i - 1), wis(f_i))$$

最大收益的工作排程 Maximum Profit in Job Scheduling

```
// 存放工作資訊的結構
typedef struct{
    int start;
    int finish;
    int profit;
} job;

// 根據工作時間排序的函式
bool cmp(job* a, job* b){
    return a->finish < b->finish;
}

class Solution{
public:
    int jobScheduling(vector<int>& startTime, vector<int>& endTime,
    vector<int>& profit){
```

```

    int len = startTime.size();

    // 存放工作資訊的向量
    // 使用指標，排序時會較快
    vector<job*> jobs(len);

    // 把工作資訊存放入向量中
    for (int i=0 ; i<len ; i++){
        jobs[i] = new job{startTime[i], endTime[i], profit[i]};
    }

    // 把工作根據結束時間排序
    sort(jobs.begin(),jobs.end(),cmp);

    // 到每一天為止的最大利潤
    // 只需要計算到最後一個工作的結束時間
    vector<int> max_profit(jobs[len-1]->finish,0);

    // 記錄上一個結束的工作其結束時間
    int last_finish;

    for (int i=0 ; i<len ; i++){
        // 取出當前這筆工作的資料
        int start = jobs[i]->start;
        int finish = jobs[i]->finish;
        int profit = jobs[i]->profit;

        // 如果不是第一個活動
        // 從上一個活動結束到這個活動結束之間的 profit
        // 都設為與前一個活動結束時的 max_profit 相同
        // 因為這個區間內都不可能舉行額外的活動
        //（結束時間是 finish-1，前一天是 finish-2）
        if (i>0){
```

```
                for(int j=last_finish ; j<finish-1 ; j++){
                    max_profit[j] = max_profit[last_finish-1];
                }
            }

            // 若選擇當前這個工作可得的收益
            int choose = max_profit[start-1] + profit;

            // 若不選擇當前這個工作可得的收益
            // 根據題目，finish 那天該工作不佔用時間
            // 所以要採的是 finish-1 的前一天
            int not_choose = max_profit[finish-2];

            // 先比較選與不選兩個選項
            int max_choose = choose>not_choose ? choose : not_choose;


            // 再來要比較同一時間結束的不同工作中有較大收益者
            max_profit[finish-1] = max_profit[finish-1] >
            max_choose ? max_profit[finish-1] : max_choose;

            last_finish = finish;
        }

        // 回傳最後一個工作結束時間的 max_profit
        return max_profit[max_profit.size()-1];

    } // end of jobScheduling

}; // end of Solution
```

目前為止，已經把之前講解分治法與貪婪演算法時做過的題目，用動態規劃再次求解。接下來，另外來看其他一些動態規劃的常見應用。

6. 其它動態規劃問題

（1）郵票問題

v_i	1	3	10	12	18

要付 n 元的郵資，已知每種郵票對應的郵資為 1、3、10、12、18。如何安排郵票的貼法，可以讓使用的郵票張數最少？

這題的解法和找錢問題一樣。狀態轉移方程式為：

$$
\begin{aligned}
&Stamp(n) \\
&= 1 + \min\,(Stamp(n-1), Stamp(n-3), \\
&\qquad\qquad Stamp(n-10), Stamp(n-12), Stamp(n-18))
\end{aligned}
$$

很多時候，不同的演算法問題其實有同樣的結構，可以用同樣的方法解決，只是問法不同而已。大部分考試中出的題目，都屬於舊瓶裝新酒，可以簡單轉換為經典的演算法問題。

（2）木頭切割問題

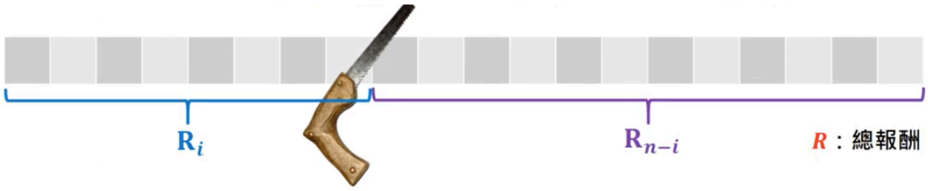
有一長度為 n 的木頭，且市場上不同長度的木頭對應價格如下：

l_i	1	2	3	4	5
p_i	10	22	35	45	56

該如何切割這個長度為 n 的木頭，使其各部分出售價格之和最高？

如果用暴力解來處理，一個長度為 n 的木頭共有 $n-1$ 個切割點（假設只切在整數長度處），每個切割點都可以選擇切或不切，複雜度為 $O(2^{n-1})$。

A. 分治法解木頭切割問題



利用分治法，可以寫出下列關係：

$$R_n = max(p_n, R_1 + R_{n-1}, R_2 + R_{n-2}, \ \ldots, R_{n-1} + R_1)$$

其中 p_n 是整段長度為 n 的木頭不切割的售價
R_i 是長度為 i 的木頭透過切割可達到的最高總價格

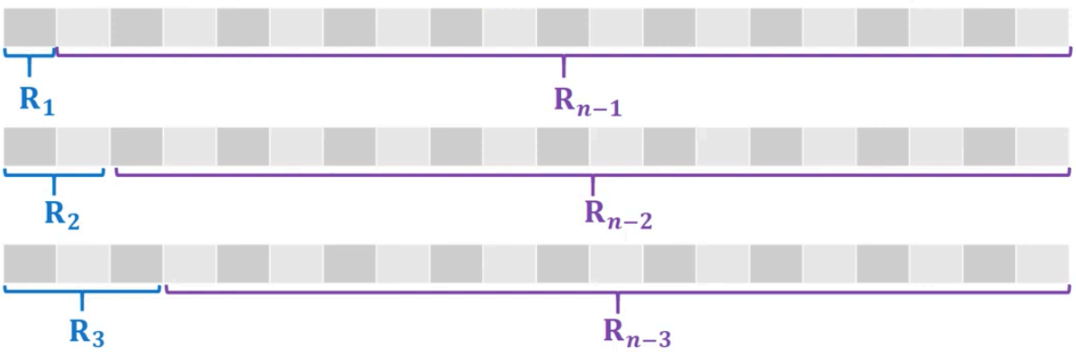
上式也可以寫成

$$R_n = max_{0 \le i \le n-1}(R_i + R_{n-i})$$

代表一個問題會被切割成 n 個子問題，當一個問題會被分成很多個子問題，且彼此相關時，就會導致分治法的運算量過高而不實用。

要減少運算量，首先要減少切割出的子問題數目，比如題目只給了 5 種長度木頭的售價，因此分治法也可以改為每次都只從左邊切一段下來，再去處理右邊剩下的木頭，寫成：

$$R_n = max_{1 \le i \le len_p}(R_{p_i} + R_{n-p_i})$$

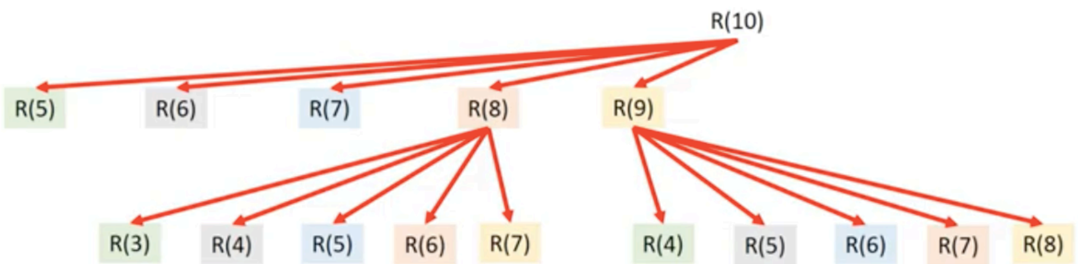


這樣一來，每個問題都對應五個子問題，也就是給定售價的木頭長度數量。

分治法解木頭切割問題

```
// *p：價格陣列
// p_len：p 的長度（即木頭種類）
// n：木頭長度
// p[0]：長度 1 的價格、p[1]：長度 2 的價格、...

int Cut_Rod(int* p, int p_len, int n){

        if (n==0)
                return 0;

        int revenue = -2147483648;

        for (int i=0 ; i<p_len ; i++){
                if (n>=i+1){
                        // 注意 p[0] 對應長度 1、p[1] 對應長度 2
                        int revenue_i = p[i] + Cut_Rod(p, p_len, n-i-1);
                        revenue = revenue>revenue_i ? revenue : revenue_i;
                }
        }
        return revenue;
}
```

上面這個程式碼符合動態規劃的精神嗎？因為中間並沒有儲存得到的結果，因此仍然存在重複計算的問題，呼叫 Cut_Rod 時，要算 R_{10} 就要算到 R_9、R_8、...，但是算 R_9 時，又要去算 R_8、R_7、...，產生了許多重複的計算。



B. 用動態規劃解決木頭切割問題

如果要發揮動態規劃的精神，就要使用一個陣列把每次算出特定長度木頭的最大價值記錄下來：

動態規劃解木頭切割問題

```
int Cut_Rod(int* p, int p_len, int n){

    if (n==0)
        return 0;

    // 儲存每種長度木頭的最大價值
    int revenue_array[n+1] = {0};

    for (int i=1 ; i<=n ; i++){
        int max_revenue = -2147483648;
        for (int j=0 ; j<p_len ; j++){

            // 要切割的長度大於目前有的木頭長度時，不處理
            if (i<=j) break;

            // 一般情形，注意 p[0] 對應長度 1 的價格
            int revenue_j = p[j] + revenue_array[i-j-1];
            max_revenue =
            max_revenue>revenue_j ? max_revenue : revenue_j;
        }

        // 每次都記錄下算出的（長度 i 的木頭）結果
        revenue_array[i] = max_revenue;
    }
    return revenue_array[n];
}
```

狀態轉移方程式同樣是：

$$R_n = max_{1 \le i \le len_p}(R_{p_i} + R_{n-p_i})$$，

根據這個式子：

l_i	1	2	3	4	5
p_i	10	22	35	45	56

n	1																		
R	10																		

$$R_1 = 10$$

長度為 1 時，只能直接按照長度 1 的價格賣掉。

l_i	1	2	3	4	5
p_i	10	22	35	45	56

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
R	10	22																	

$$\begin{aligned} R_2 &= max(R_1 + R_1, R_2 + R_0) \\ &= max(10 + 10, 22 + 0) \\ &= 22 \end{aligned}$$

長度為 2 時，可以選擇切割一段長度為 1 或者一段長度為 2 的下來。

$$\begin{aligned} R_3 &= max(R_1 + R_2, R_2 + R_1, R_3 + R_0) \\ &= max(10 + 22, 22 + 10, 35 + 0) \\ &= 35 \end{aligned}$$

長度為 3 時，可以選擇從最左邊切割一段長度為 1、一段長度為 2，或者一段長度為 3 的木頭下來。

依此類推，就可以漸次得到各個長度木頭的最高價格，這就是 buttom-up 的解決方式。比如：

$$R_7 = max(R_1 + R_6, R_2 + R_5, R_3 + R_4, R_4 + R_3, R_5 + R_2)$$
$$= (10 + 70, 22 + 57, 35 + 45, 45 + 35, 57 + 22)$$

長度為 7 的木頭的最高價值，可以從已經解決的長度較小的木頭的最高價值拼湊而成。

（3）LeetCode #343. 拆分整數 Integer Break

A. 題目

給定一個整數 n，將它拆分為 k 個正整數的和，其中 $k \geq 2$。找到可使這些拆分出的整數乘積最大的拆分方法，並回傳該乘積。

B. 出處
https://leetcode.com/problems/integer-break/

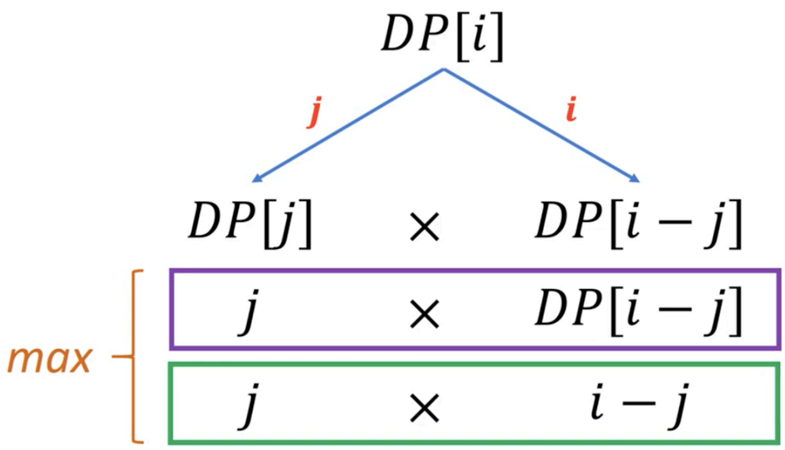
C. 範例輸入與輸出
輸入：n = 2
輸出：1

2 只能拆分出 $1 + 1$，回傳 $1 \times 1 = 1$。

輸入：n = 10
輸出：36

最佳拆分方法為 $10 = 3 + 3 + 4$，回傳 $3 \times 3 \times 4 = 36$。

D. 解題邏輯



根據題目規定，一個整數 i 至少要被拆分成兩個整數的和，把兩個整數分別表示為 j 與 $i-j$，可以選擇不再將兩者繼續往下拆分，或選擇繼續拆分。

因此，$DP[i]$ 是四個值中最大者（$DP[i]$ 代表把 i 拆分開來）：

a. j 和 $i-j$ 都不再拆分：$j \times (i-j)$
b. 只繼續拆分 $i-j$：$j \times DP[i-j]$
c. 只繼續拆分 j：$DP[j] \times (i-j)$
d. 兩個都繼續拆分：$DP[j] \times DP[i-j]$

不過，可以把 j 指定為「不再拆分」的部分，這樣一來，只要比較 $j \times DP[i-j]$ 和 $j \times (i-j)$ 兩者中何者較大即可。

拆分整數 Integer Break

```
class Solution{
public:
    int integerBreak(int n){

        // 儲存中間結果的陣列
        vector<int> DP(n+1,1);

        for (int i=2 ; i<n+1 ; i++){

            int max = -2147483648;
```

```

            for (int j=1 ; j<i ; j++){
                // i-j  繼續拆分與不拆分兩者間，取較大的
                int now = j*DP[i-j] > j*(i-j) ? j*DP[i-j] : j*(i-j);
                max = now>max ? now : max;
            }

            //  儲存得到的結果
            DP[i] = max;
        }
        return DP[n];

    } // end of integerBreak

}; // end of Solution
```

7. 背包問題

接下來是之前曾經提過的背包問題：

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前提下，讓背包中裝的物品總價值最大？

w_i	4	5	2	6	3	7
v_i	6	7	5	12	5	18

W 是一個正數，每個物品的重量 w_i 和價值 v_i 分別被儲存在相應的陣列當中。

不同於之前用貪婪演算法解決的「Fractional Knapsack Problem」中，每個物品可以只拿部分，這裡要解決的則是「0 / 1 背包問題」，只能選擇「拿」或「不拿」每項物品，會使用到二維的動態規劃。

這是一個典型的 NP-complete 問題，無法快速求得最佳解，但「驗證」一個解則很快，只要確定某個組合可以被放進背包裡，沒有超過容量。

若使用暴力解，當物品數量為 N 時，每個物品可以拿或不拿，因此選擇的方法有 $O(2^N)$ 種。

（1）用二維動態規劃解決背包問題

用 $DP[i][j]$ 來記錄「前 i 件物品放入容量 j 的背包所能產生的最大價值」。

用迴圈依序處理每個物品，處理到第 i 件時，有兩種選擇：

A. 不放入第 i 件物品：那麼同樣容量 j 的背包能產生的最大價值與只考慮前 $i-1$ 件物品時相同，為 $DP[i-1][j]$

B. 放入第 i 件物品：因為必須空出空間來放入這個新的物品，所以前面 $i-1$ 件物品可能就會有一些放不下。這也就是說，前 $i-1$ 件物品只能

選擇一些放到剩下的 $j-w_i$ 的容量中，得到的最大價值為 $DP[i-1][j-w_i]+v_i$，其中 v_i 是新加入的第 i 件物品的價值

由上述，可以列出狀態轉移方程式：

$$DP[i][j] = \begin{cases} DP[i-1][j], j < w_i \\ max(DP[i-1][j], DP[i-1][j-w_i]+v_i),\ j \geq w_i \end{cases}$$

其中 $j < w_i$ 的情形代表新處理到的第 i 件物品自己就太重而無法放入背包，這時只需考慮前 $i-1$ 件物品能達到的最高價值即可。

（2）二維動態規劃的進行過程

w_i	1	2	3
v_i	2	5	8

$$DP[i][j] = \begin{cases} DP[i-1][j], j < w_i \\ max(DP[i-1][j], DP[i-1][j-w_i] + v_i), j \geq w_i \end{cases}$$

w_i	v_i	0	1	2	3	4	5
0	0	0	0	0	0	0	0
1	2	0					
2	5	0					
3	8	0					

上例中，有三個物品，重量分別為 1、2、3，價值分別為 2、5、8。

首先，畫出的表中最上方的 row 和最左邊的 column 都會填上 0（兩個維度都加上 0 值來處理邊界條件），最上方的 row 代表「不考慮放任一種物品」，而最左邊的 column 則代表「背包重量為 0」。

接下來，$DP[1][1]$ 代表了「只用第一個物品來裝一個負重為 1 的背包」，這時根據狀態轉移方程式：

$DP[1][1]$
$= max(DP[0][1], DP[0][0] + 2)$
$= max(0,2) \ = \ 2$

max 中第一項 $DP[0][1]$ 代表「只用前 0 項物品來填負重為 1 的背包」，因為沒有物品可以選，自然得到的價值為 0，而第二項 $DP[0][0] + 2$ 是「選擇放入第一項物品」的情形，2 是第一項物品的價值，$DP[0][0]$ 則是用把第一項物品之前的物品放入容量為 $1 - 1 = 0$（容量 1，被第一項物品用掉 1）的背包中。

w_i	v_i	0	1	2	3	4	5
0	0	0	0	0	0	0	0
1	2	0	2				
2	5	0					
3	8	0					

類似的，$DP[2][1]$ 代表「只用前兩個物品來裝一個負重為 1 的背包」，根據狀態轉移方程式：

$DP[2][1]$
$= max(DP[1][1], DP[1][-1] + 5)$
$= max(2, X) \ = \ 2$

其中 $DP[1][1]$ 是不放第二件物品，只考慮第一件物品時的最大價值，$DP[0][-1] + 5$ 是放入第二件物品時可以得到的最大價值，但是因為第二件物品重量超過此時假設的背包負重 1（$j < w_i$），所以此項不考慮。

類似的：
$DP[3][1]$
$= max(DP[2][1], DP[2][-2] + 8)$
$= max(2, X) \ = \ 2$

w_i	v_i	0	1	2	3	4	5
0	0	0	0	0	0	0	0
1	2	0	2				
2	5	0	2				
3	8	0					

$DP[2][2]$
$= max(DP[1][2], DP[1][0] + 5)$
$= max(2, 0 + 5) \ = \ 5$

前項 $DP[1][2]$ 是不選擇第二件物品，只用第一件物品來填負重為 2 的背包時的最大價值，$DP[1][0] + 5$ 則是選擇放入價值為 5 的第二件物品後，編號在前面的物品還可以放入負重為 $2 - 2 = 0$ 的背包時的情形。

依此類推，可以填滿整個表，讀者可以利用狀態轉移方程式來一一檢視每個值的正確性：

例如
$DP[3][5]$
$= max(DP[2][5], DP[2][2] + 8)$
$= max(7, 5 + 8) \ = \ 13$

w_i	v_i	0	1	2	3	4	5
0	0	0	0	0	0	0	0
1	2	0	2	2	2	2	2
2	5	0	2	5	7	7	7
3	8	0	2	5	8	10	13

（3）Try LeetCode #518. 找錢問題 2 Coin Change 2

A. 題目

給定一個整數陣列 $coins$ 代表每種硬幣的面額，與一個整數 $amount$ 代表要找的金額。

回傳可以組成該金額的方法數，如果沒有任何方法可以組成該金額，回傳 0。假設每種面額的硬幣數不限。

B. 出處
https://leetcode.com/problems/coin-change-2/

C. 範例輸入與輸出
輸入：amount = 5, coins = [1,2,5]
輸出：4

有四種方法可以湊出 5 元：
5 = 5
5 = 2+2+1
5 = 2+1+1+1
5 = 1+1+1+1+1

D. 解題邏輯

本題狀態轉移方程式類似 0 / 1 背包問題。

$DP[i][j]$ 是用前 i 種硬幣去找 j 元

$$DP[i][j] = \begin{cases} DP[i-1][j] & \text{不使用第 i 種硬幣，只使用前 } i-1 \text{ 種} \\ & + \\ DP[i]\big[j - coin[i-1]\big] & \text{使用至少一個第 } i \text{ 種硬幣} \end{cases}$$

後面一項中，$coin[i-1]$ 代表的是「第 i 種硬幣的面額」，湊成的金額是用「包括 i 在內的前面種類的硬幣」去湊出「除了最後使用的一個第 i 種硬幣

外，還要找的 $j - coin[i-1]$ 元」。

v_i	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1										
2	1										
5	1										

不過還要注意到插入第一個 row 和第一個 column 後，第一個 row 的值是 0，因為「不考慮任何面額的硬幣」時一定無法找零；第一個 column 的值則是 1，因為「不找任何硬幣」就是找出零元的唯一一種方法。

v_i	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1									
2	1										
5	1										

表中，$DP[1][1]$ 可以表示為

$DP[0][1] + DP[1][0]$

$= 0 + 1 = 1$

v_i	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1									
2	1	1									
5	1	1									

$DP[3][1]$

$= DP[2][1] + DP[3][-4]$

$= 1 + X = 1$

v_i	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	1								
2	1	1	2								
5	1	1									

$DP[2][2]$

$= DP[1][2] + DP[2][0]$

$= 1 + 1 = 2$

在這個表中，最下方的 row 代表每種硬幣都可以使用的情形，因此對應了各個金額完整的找零方法，比如 8 這個 column 的最下方一個 row 這格中填的數字，就代表要找 8 元的所有方法數。

轉為一維動態規劃的方法：

$$DP[i][j] = \begin{cases} DP[i-1][j] \\ + \\ DP[i][j - coin[i-1]] \end{cases}$$

$$DP[i\text{-}1][j] = \begin{cases} DP[i-2][j] \\ + \\ DP[i\text{-}1][j - coin[i-2]] \end{cases}$$

$$DP[i\text{-}2][j] = \begin{cases} DP[i-3][j] \\ + \\ DP[i\text{-}2][j - coin[i-3]] \end{cases}$$

$$\cdots\cdots$$

$$DP[1][j] = \begin{cases} DP[0][j] = 0 \\ + \\ DP[1][j - coin[0]] \end{cases}$$

v_i	0	1	2	3	4	5	6	7	8	9	10
0	1	0	0	0	0	0	0	0	0	0	0
1	1	1									
2	1	1									
5	1	1									

如上圖中所示，因為每次拆出的 $DP[i-1][j]$ 都可以再行拆分，因此也可以表示成一維的陣列：

$$DP[len_{coin}][j]$$
$$= \sum_{i=1}^{len_{coin}} DP[i][j-coin[i-1]]$$

因此，下面的 pseudocode 也可以得到 $DP[len_{coin}][i]$ 的值：

一維動態規劃解找錢問題 2

$DP[0] = 1$

$for\ i = 0 \sim len_{coin} - 1$

    $for\ j = coin[i] \sim amount$

        $DP[j] += DP[j - coin[i]]$

利用這種方式，可以把「空間複雜度」從原先的二維陣列往下壓到 $O(amount + 1)$。

找錢問題 2 Coin Change 2 [二維陣列]

```
class Solution{
public:
    int change(int amount, vector<int>& coins){

        int len = coins.size();
        // 陣列大小為 len+1 個 row 乘上 amount+1 個 column
        vector<vector<int>> DP(len+1,vector<int>(amount+1,0));

```

```
        DP[0][0] = 1;

        // 用雙重迴圈逐一填上 DP 的值
        // 不用填第 0 個 row，因為值和預設值 0 相同
        for (int i=1 ; i<=len ; i++){
            DP[i][0] = 1;   // 第 0 個 column 填上 1

            for (int j=1 ; j<=amount ; j++){

                // DP[i][j] = DP[i-1][j] + DP[i][j-coins[i-1]]
                // 不使用第 i 種硬幣 + 使用第 i 種硬幣
                DP[i][j] += DP[i-1][j];
                if (j-coins[i-1]>=0){
                    DP[i][j] += DP[i][j-coins[i-1]];
                }

            } // end of inner for

        } // end of outer for

        // 回傳值中 len 代表考慮使用所有種類的硬幣
        // amount 代表要湊出 amount 元
        return DP[len][amount];

    } // end of change

}; // end of Solution
```

接下來，示範只需使用一維陣列，空間複雜度較低的解法：

找錢問題 2 Coin Change 2 [一維陣列]

```
class Solution{
public:
    int change(int amount, vector<int>& coins){
```

```
        int len = coins.size();

        vector<int> DP(amount+1,0);
        DP[0] = 1;

        for (int coin:coins){

            // i 要從 coin 開始，避免 i-coin<0 的情況
            for (int i=coin ; i<=amount ; i++){
                DP[i] += DP[i-coin];
            }

        } // end of for

        return DP[amount];

    } // end of change

}; // end of Solution
```

8. 總結三種常見的演算法精神

一般來說，遇到一個最佳化問題，先想想是否能使用貪婪解，因為貪婪解通常執行起來較快，空間複雜度也較低。

在題目不適用貪婪演算法時，再考慮動態規劃和分治法

A. 子問題間不相干時，使用分治法
B. 子問題間相干時，使用動態規劃

動態規劃因為會把每個子問題的解記錄下來，所以會耗費較大的記憶體空間，但也因此可以避免使用分治法時重複運算的問題，是一種以「空間換取時間」的策略。

	分治法	動態規劃
適用時機	子問題間不交疊（overlap）	子問題間交疊（overlap）
演算過程	Top-Down 居多	Bottom-Up 居多
記憶體空間	不需額外空間	需要額外空間