Dynamic Programming (1)

by music960633

Sprout

課程內容

- 1. 什麼是DP？
- 2. Top down / Bottom up
- 3. 「狀態」、「狀態轉移」
- 4. 取餘數

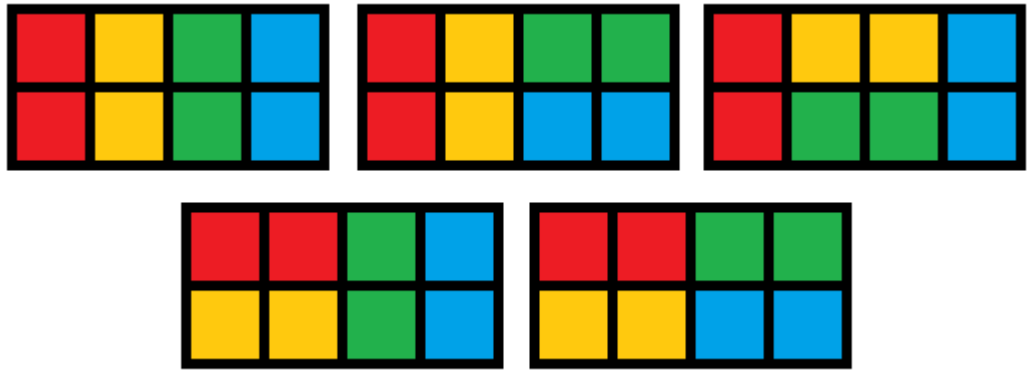
Sprout

什麼是DP

- Dynamic programming，動態規劃
- 將一個問題分成許多子問題，遞迴求解，最後再由子問題的答案得到原本問題的答案(類似D&C)
- 相同的子問題會出現不只一次(與D&C不同)
- 將子問題的答案記錄起來
  - 避免對相同的問題再遞迴一次
  - 用空間換取時間

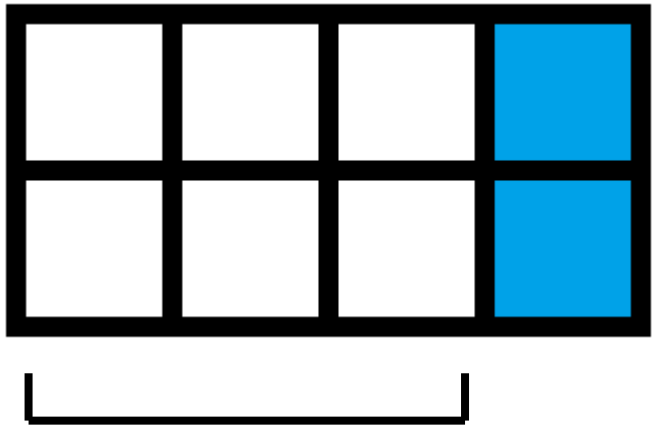
Example 1

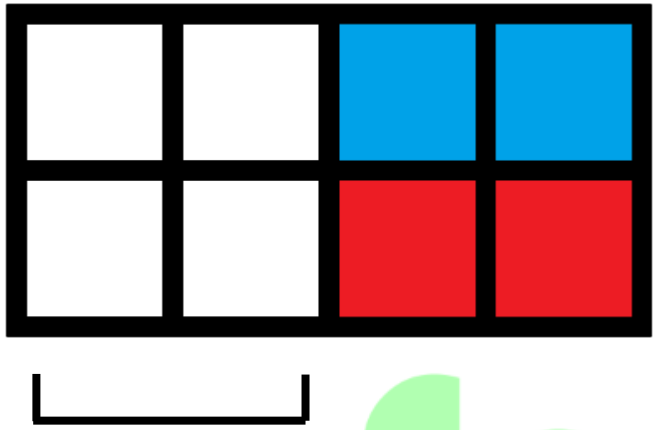
- 用1*2的骨牌填滿2*n的格子，共有幾種排法？



Example 1

- 用1*2的骨牌填滿2*n的格子，共有幾種排法？
- Sol:
  - 以f(n)表示填滿2*n格子的方法數
  - 觀察最後一格放置骨牌的情形：





Example 1

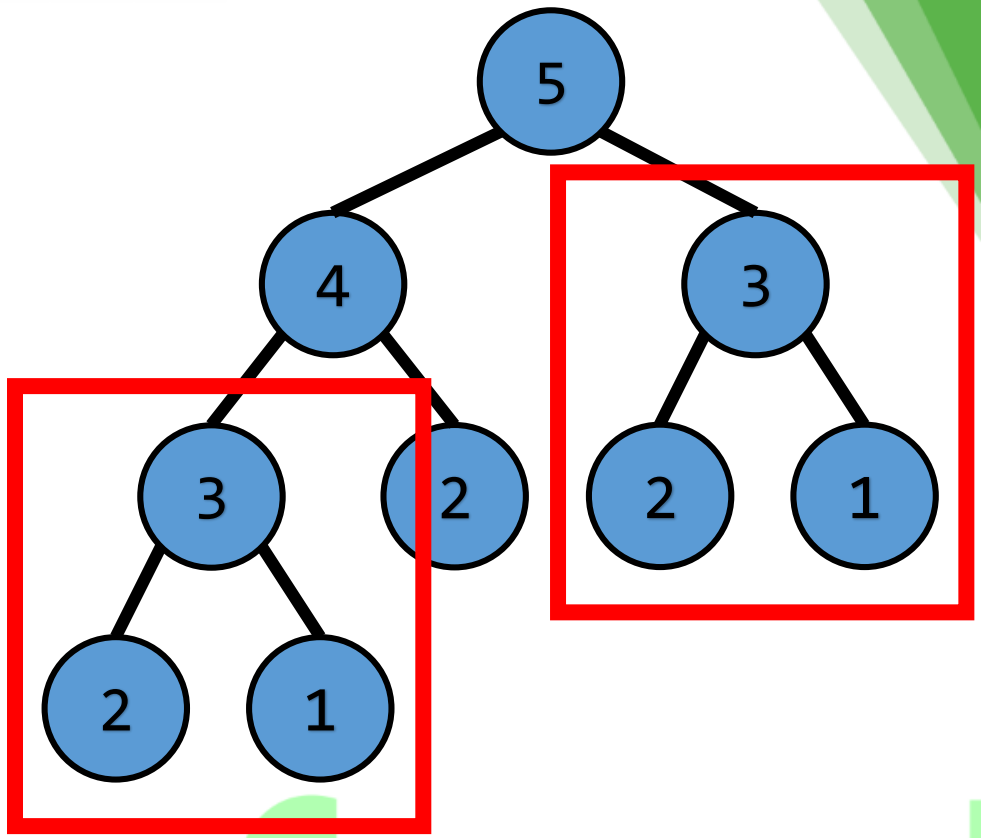
- 用1*2的骨牌填滿2*n的格子，共有幾種排法？
- Sol:
  - 以f(n)表示填滿2*n格子的方法數
  - 觀察最後一格放置骨牌的情形：
  - f(n)=f(n-1)+f(n-2)
  - 初始條件：f(1)=1, f(2)=2
  - 費氏數列

Example 1

- 實作
- 最簡單遞迴版本

```
3  int f(int n){
4      // initial values
5      if(n==1) return 1;
6      if(n==2) return 2;
7      // recursion
8      return f(n-1) + f(n-2);
9  }
```

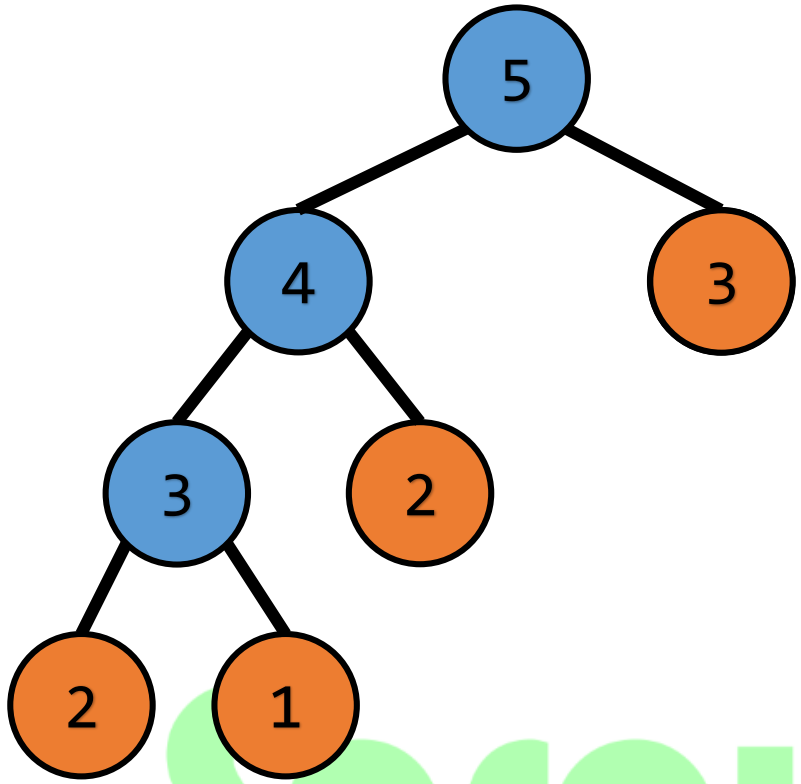
- 缺點：太慢
- 時間複雜度：O(f(n))
- 解決方法：使用陣列記錄答案



Top down

- 將答案記錄起來，如果遇到相同的問題，則直接查表

```
3  int dp[101]={1, 1, 2};
4
5  int f(int n){
6      // calculated
7      if(dp[n] != 0) return dp[n];
8      // not calculated
9      dp[n] = f(n-1) + f(n-2);
10     return dp[n];
11 }
```



- 時間複雜度：O(n)

Bottom up

- 用迴圈先把答案全部算出來

```
3  int dp[101];
4
5  void build(){
6      // initial values
7      dp[1] = 1;
8      dp[2] = 2;
9      // recursion
10     for(int i=3; i<101; ++i)
11         dp[i] = dp[i-1] + dp[i-2];
12 }
13
14 int f(int n){ return dp[n]; }
```

- 時間複雜度：O(n)

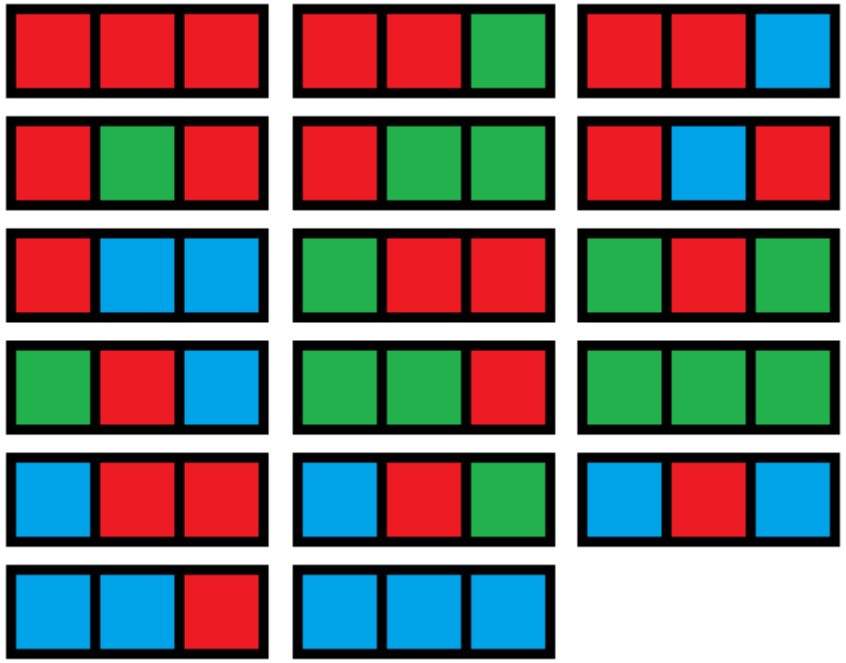
Top down vs Bottom up

- Top down:
  - 只要知道遞迴式就可以了，剩下交給遞迴跑
  - code一般為遞迴函式
  - 注意遞迴過深

- Bottom up
  - 子問題一定要比母問題先跑到(注意迴圈跑法)
  - code一般為for迴圈

Example 2

- 將n個排成一列的格子塗上紅、綠、藍三種顏色，且藍綠不可相鄰，問有幾種塗法？



Example 2

- 將n個排成一列的格子塗上紅、綠、藍三種顏色，且藍綠不可相鄰，問有幾種塗法？
- Sol:
  - 設f(n)為塗n格的方法數
  - f(n)=.....
  - 糟糕，遞迴式不知道怎麼寫

Example 2

- 將n個排成一列的格子塗上紅、綠、藍三種顏色，且藍綠不可相鄰，問有幾種塗法？
- Sol:
  - 設f(n,0)為塗n格，且最後一格為紅色的方法數
  - 設f(n,1)為塗n格，且最後一格為綠色的方法數
  - 設f(n,2)為塗n格，且最後一格為藍色的方法數
  - 遞迴式
    - f(n,0)=f(n-1,0)+f(n-1,1)+f(n-1,2)
    - f(n,1)=f(n-1,0)+f(n-1,1)
    - f(n,2)=f(n-1,0)+f(n-1,2)
    - 初始條件：f(1,0)=f(1,1)=f(1,2)=1
  - 最後答案：f(n,0)+f(n,1)+f(n,2)

Example 2

- Top down

```
3  int dp[101][3] = {};
4
5  // initialize
6  void init(){ dp[1][0] = dp[1][1] = dp[1][2] = 1; }
7
8  int f(int n, int m){
9      // calculated
10     if(dp[n][m] != 0) return dp[n][m];
11     // not calculated
12     if(m==0) dp[n][m] = f(n-1,0)+f(n-1,1)+f(n-1,2);
13     if(m==1) dp[n][m] = f(n-1,0)+f(n-1,1);
14     if(m==2) dp[n][m] = f(n-1,0)+f(n-1,2);
15     return dp[n][m];
16 }
```

Example 2

- Bottom up

```
3  int dp[101][3];
4
5  void build(){
6      // initial values
7      dp[1][0] = dp[1][1] = dp[1][2] = 1;
8      // recursion
9      for(int i=2; i<101; ++i){
10         dp[i][0] = dp[i-1][0]+dp[i-1][1]+dp[i-1][2];
11         dp[i][1] = dp[i-1][0]+dp[i-1][1];
12         dp[i][2] = dp[i-1][0]+dp[i-1][2];
13     }
14 }
15
16 int f(int n){ return dp[n][0]+dp[n][1]+dp[n][2]; }
```

「狀態」、「狀態轉移」

- 狀態
  - Example 1: f(n)表示填滿2*n格子的方法數
  - Example 2:
    - f(n,0)為塗n格，且最後一格為紅色的方法數
    - f(n,1)為塗n格，且最後一格為綠色的方法數
    - f(n,2)為塗n格，且最後一格為藍色的方法數
  - 在以上兩個例子中，我們都定義了函式參數所代表的意義，或是陣列索引值所代表的意義
    - f(n,m) 中的 n,m
    - dp[n][m] 中的 n,m
- 狀態：用一些數字來「唯一」表示一個子問題

「狀態」、「狀態轉移」

- 狀態轉移
  - Example 1: f(n)=f(n-1)+f(n-2)
  - Example 2:
    - f(n,0)=f(n-1,0)+f(n-1,1)+f(n-1,2)
    - f(n,1)=f(n-1,0)+f(n-1,1)
    - f(n,2)=f(n-1,0)+f(n-1,2)
  - 在以上兩個例子中，我們可以用一些方程式表示如何由其他的子問題結果得到一個問題的答案
- 狀態轉移：如何由其他的狀態得到某個狀態的值

「狀態」、「狀態轉移」

- 定義狀態時該注意的事
  - 1. 狀態是否太多？ (太多陣列開不下)
  - 2. 是否能導出狀態轉移式？ (如Example 2)

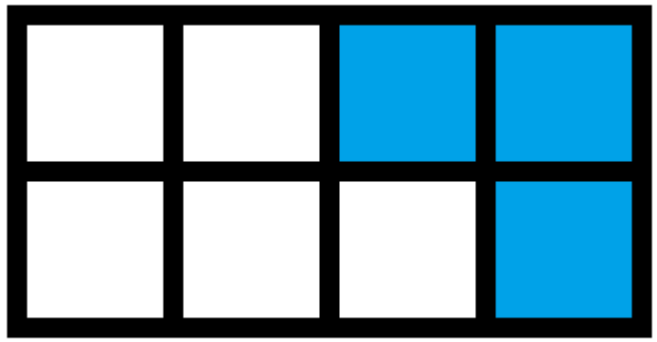
- 定出狀態且找出狀態轉移式之後
  - 1. 時間複雜度是否合理？
  - 2. 若不合理，能不能對狀態轉移做優化？
  - 3. 若還是不行，試試看用其他方法定義狀態

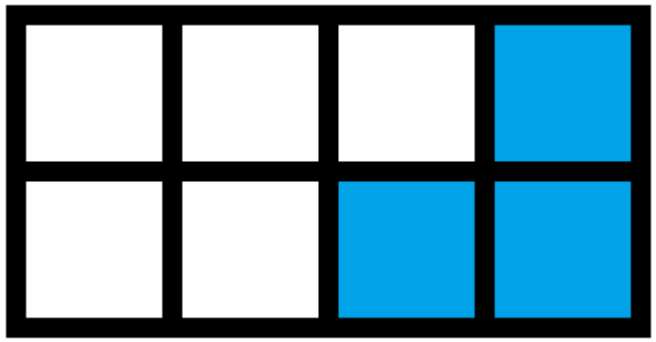
- 如何找到正確的狀態定義方式？
  - 靠經驗
  - 靠靈感

Sprout

Example 3

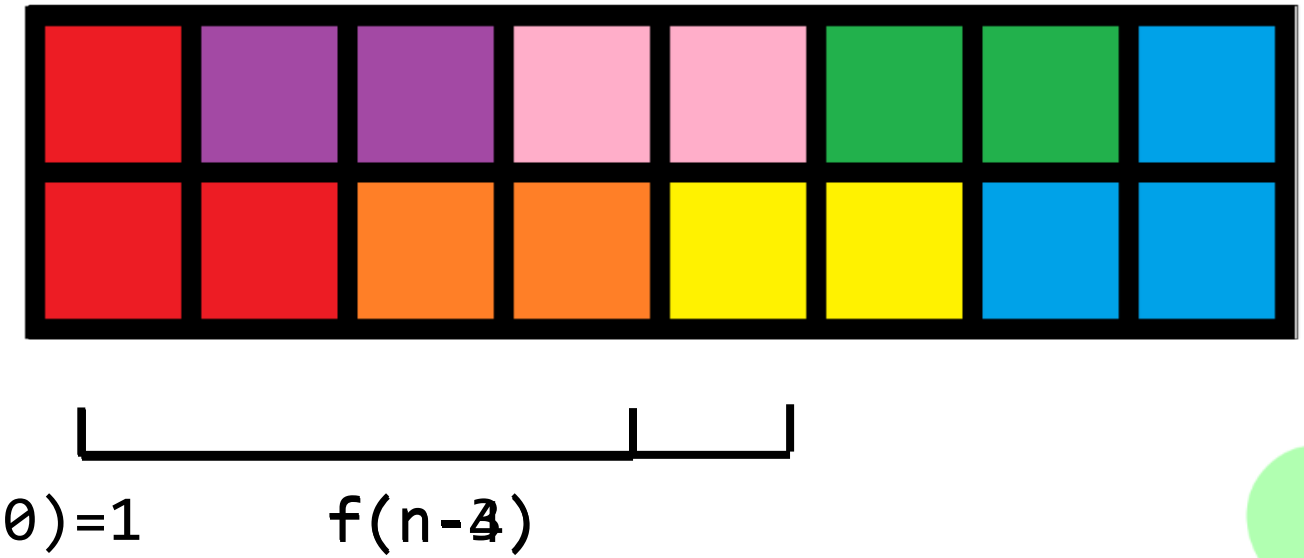
- 用1*2和1*3的L型骨牌填滿2*n的格子，共有幾種排法？
- 定義狀態
  - f(n)表示填滿2*n格子的方法數
  - 狀態是否太大？ No
- 狀態轉移
  - 同樣考慮最後一格的放法，放1*2的case已在Example 1討論過，因此只討論放L型骨牌的case





Example 3

- 因為兩種情況是上下對稱的，因此只考慮一種方向
  - 最後放1*2：f(n-1)+f(n-2)
  - 最後放L型：



Example 3

- 因為兩種情況是上下對稱的，因此只考慮一種方向
  - 最後放1*2：f(n-1)+f(n-2)
  - 最後放L型：2[f(n-3)+f(n-4)+...+f(1)+f(0)]

- 狀態轉移
  - f(n)=f(n-1)+f(n-2)+2[f(n-3)+f(n-4)+...+f(1)+f(0)]
  - O(n)時間轉移
  - 是否合理？ $O(n^2)$好像不太好

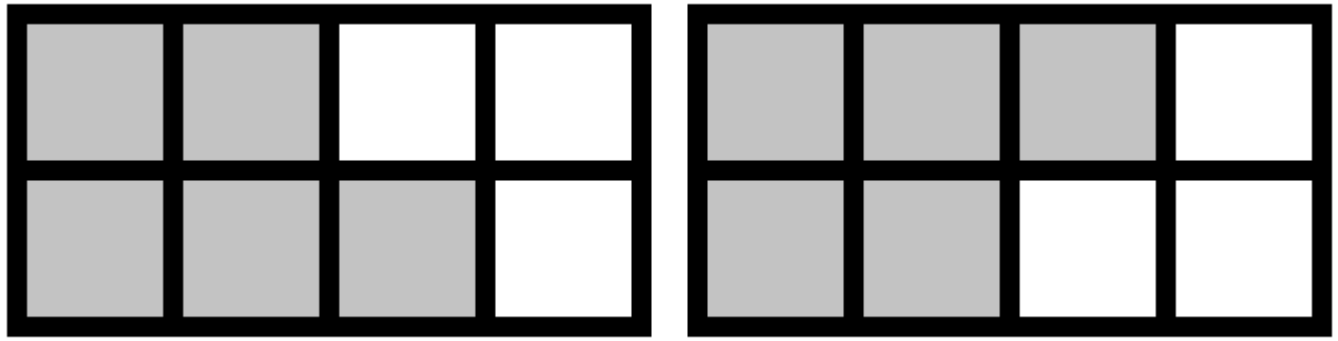
- 能不能優化呢？

Example 3

- 優化 f(n)=f(n-1)+f(n-2)+2[f(n-3)+f(n-4)+...+f(1)+f(0)]
- 使用變數記錄前綴和
  - 若使用bottom up，可以用一個變數(tmp)記錄f(0)+f(1)+...+f(n-3)，如此可讓轉移變成 f(n)=f(n-1)+f(n-2)+2tmp，O(1)轉移
- 化簡轉移式
  - f(n-1)=f(n-2)+f(n-3)+2[f(n-4)+f(n-5)+...+f(1)+f(0)]
  - f(n-1)+f(n-3)=f(n-2)+2[f(n-3)+f(n-4)+f(n-5)+...+f(1)+f(0)]
  - 得到 2[f(n-3)+f(n-4)+...+f(1)+f(0)]=f(n-1)-f(n-2)+f(n-3)
  - 轉移式變成 f(n)=2f(n-1)+f(n-3)，O(1)轉移

Example 3

- 另外一種狀態定義方式
  - f(n,0): 鋪滿2*n的方法數
  - f(n,1): 鋪滿2*n且下面多一格的方法數



2*f(2,1)

Example 3

- 另外一種狀態定義方式
  - f(n,0): 鋪滿2*n的方法數
  - f(n,1): 鋪滿2*n且下面多一格的方法數
- 狀態轉移
  - f(n,0)=f(n-1,0)+f(n-2,0)+2f(n-2,1)
  - f(n,1)=f(n-1,0)+f(n-1,1)
- 初始狀態
  - f(0,0)=1, f(0,1)=0
  - f(1,0)=1, f(1,1)=1
  - f(2,0)=2, f(2,1)=2

取餘數

- 在計算方法數的題目中，我們常看到「請輸出答案除以M的餘數」
- 以Example 3來說，f(28)=1914332891，如果n到很大(100萬)，則一定要用到大數，但大部分不想要那麼麻煩，因此會要求輸出餘數即可
- 加減乘法可以直接取餘數，除法則不行
- 小心在mod之前就overflow，因此建議開long long(尤其乘法)

取餘數

- 若Example 3要求輸出答案除1000007的餘數

```
3  int dp[101] = {};
4
5  void build(){
6      dp[0] = 1;
7      dp[1] = 1;
8      dp[2] = 2;
9      for(int i=3; i<101; ++i)
10         dp[i] = (2*dp[i-1]+dp[i-3]) % 1000007;
11 }
```

Example 4

- 給一個正整數陣列，現在要從裡面取出一些數，但兩數不能相鄰，求取出數字和的最大值
- Ex: arr[] = {1,4,2,3,5}，則取出4和5有最大總合9

Example 4

- 定義狀態
  - 假設索引值從1開始
  - f(n)為從arr[1]到arr[n]中取出數字，且有取到arr[n]的總合最大值
- 狀態轉移
  - 因為數字不為負的，因此選擇的兩個數字之間一定間格1或2
  - 如果取了arr[n]，則上一個取到的數字為arr[n-2]或arr[n-3]
  - f(n)=max(f(n-2),f(n-3))+arr[n]
- 最後答案
  - max(f(N), f(N-1))

Example 4

- 另外一個解法
- 定義狀態
  - f(n,0)為從arr[1]到arr[n]取數字，且沒有取到arr[n]的總合最大值
  - f(n,1)為從arr[1]到arr[n]取數字，且有取到arr[n]的總合最大值
- 狀態轉移
  - 如果沒有取到arr[n]，則arr[n-1]不論有沒有取都合法
  - 如果有取到arr[n]，則一定不能取arr[n-1]
  - f(n,0)=max(f(n-1,0), f(n-1,1))
  - f(n,1)=f(n-1,0)+arr[n]
- 最後答案
  - max(f(N,0), f(N,1))