AlgoSeacow

動態規劃 I
(Dynamic Programming I)

動態規劃介紹
(Dynamic Programming)

狀態與轉移

- 設計狀態轉移 SOP
  - Step1：設計 狀態
  - Step2：找到 轉移式 (狀態與狀態之間的關係) 和 邊界條件
  - Step3：找邊界條件與查找 答案

狀態與轉移 - LeetCode 70. Climbing Stairs

- 樓梯有 n 階，可以跨 1 步或 2 步
- 請問有多少種不同的走法？

狀態與轉移 - LeetCode 70. Climbing Stairs

- 樓梯有 n 階，可以跨 1 步或 2 步
- 請問有多少種不同的走法？

- 定義狀態
  - F(n) : 從第 0 層走到第 n 層的方法數
- 轉移式和邊界條件
  - $F(n) = F(n-1) + F(n-2)$
  - $F(0) = 1, F(1) = 1$
- 答案
  - F(n)

狀態與轉移 - LeetCode 70. Climbing Stairs

- 樓梯有 n 階，可以跨 1 步或 2 步
- 請問有多少種不同的走法？

- 定義狀態
  - F(n) : 從第 0 層走到第 n 層的方法數
- 轉移式和邊界條件 (白話文版)
  - (到第 n 層的方法數) = (到第 n-1 層的方法數) + (到第 n-2 層的方法數)
  - 到第 0 層的方法數是 1，到第 1 層的方法數是 1
- 答案
  - F(n)

狀態與轉移 - 前綴和

- 輸入一個陣列 a[1] … a[n]
- 定義長度 k 的前綴和為 a[1] + a[2] + … + a[k]
- 對於 k = 1~n 輸出每一個前綴和數值

狀態與轉移 - 前綴和

- 輸入一個陣列 a[1] ... a[n]
- 定義長度 k 的前綴和為 a[1] + a[2] + ... + a[k]
- 對於 k = 1~n，輸出每一個前綴和數值

- 定義狀態
  - F(n) : 長度 k 的前綴和
- 轉移式和邊界條件
  - F(i) = a[1] + a[2] + ... + a[i] or F(i) = F(i-1) + a[i]
  - F(0) = 0
- 答案
  - F(1), F(2), F(3), ..., F(n)

狀態與轉移 - 二分最糟情況

- 二分搜尋每個回可以把答案範圍變成本來的大約一半
- [L, R] 變成 [L, mid] 或是 [mid + 1, R]，其中 mid = (L+R) / 2
- 給定 L, R，求最糟情況要比較幾個回合

- 定義狀態
  - F(L, R) : 答案範圍在 L, R，二分搜尋回合數的最大可能
- 轉移式和邊界條件
  - F(L, R) = 1 + max{ F(L, mid), F(mid+1, R) }
  - F(i, i) = 0
- 答案
  - F(L, R)

狀態與轉移 - 二分最糟情況

- 二分搜尋每個回可以把答案範圍變成本來的大約一半
- [L, R] 變成 [L, mid] 或是 [mid + 1, R]，其中 mid = (L+R) / 2
- 給定 L, R，求最糟情況要比較幾個回合

- 定義狀態
  - F(n) : 答案範圍長度是 n，二分搜尋回合數的最大可能
- 轉移式和邊界條件
  - $F(n) = 1 + \max\{ F(n/2), F(n - n/2) \}$
  - $F(0) = 1, F(1) = 1$
- 答案
  - $F(R - L)$

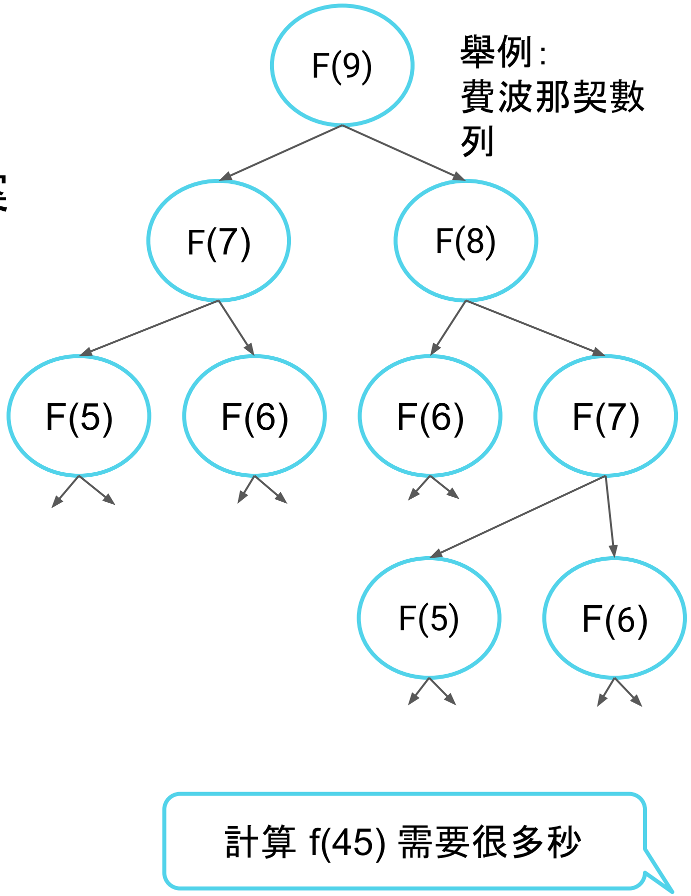
不同的狀態定義，轉移大不同

避免重複計算

- 有了狀態轉移關係，可以直接跑遞迴計算答案
- 但是有很多一樣的狀態會重複計算到

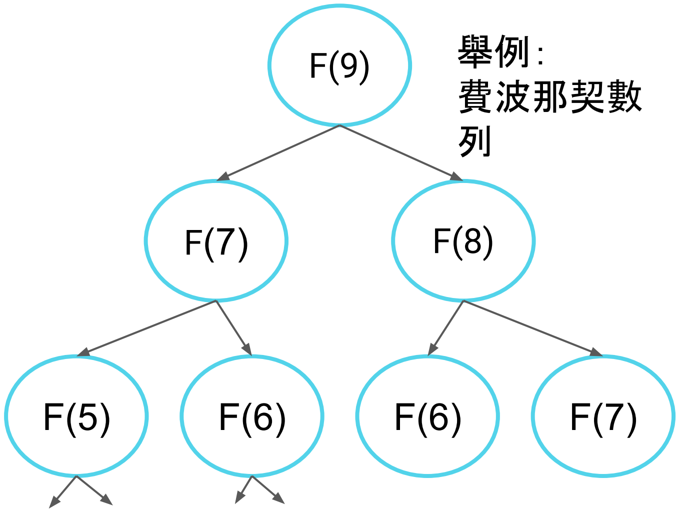
```
int f(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }

    int ans = f(n - 1) + f(n - 2);
    return ans;
}
```



避免重複計算

- 把算過的東西計算下來，之後再遇到就可以直接使用，不需要反覆計算求解



F(6) 和 F(7) 已經記錄過所以不往下遞迴

避免重複計算 - Top-down / 記錄

- 開一個陣列，算過的紀錄下來，下次遇到不用再遞迴

```
int dp[MAXN];

int f(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    if (dp[n] != 0) {
        return dp[n];
    }

    dp[n] = f(n - 1) + f(n - 2);
    return dp[n];
}
```

避免重複計算 - Bottom-up / 預處理

- 直接把所有狀態的答案按照某個順序計算出來

```
int dp[MAXN];

void build() {
    dp[0] = 1;
    dp[1] = 1;
    for (int i = 2; i < MAXN; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
}

int f(int n) {
    return dp[n];
}
```

動態規劃

= 狀態轉移關係 + 避免重複計算

動態規劃 - 時間複雜度分析

- 時間複雜度 = 每個狀態都計算一次所需的時間

- 若每一個狀態的轉移時間都相同
    - 狀態數量 * 每個狀態所需要的轉移時間

- 費氏數列
    - 費氏數列有 n 個狀態，每個狀態 O(1) 時間轉移到別的狀態
    - 總時間 n * O(1) = O(n)

動態規劃 - 1D/0D !? 1D/1D !? 2D/0D !?

- 狀態數量有 n^a 個，每個狀態的轉移時間是 n^b
- 我們就稱這個問題是屬於 aD/bD dp 問題

- 費氏數列
  - 費氏數列有 n^1 個狀態，每個狀態 $O(1) = O(n^0)$ 時間轉移到別的狀態
  - 1D / 0D dp
- 前綴和
  - 前綴和有 n^1 個狀態，每個狀態 $O(1) = O(n^0)$ 時間轉移到別的狀態
  - 1D / 0D dp

動態規劃 - 學習技巧

- 動態規劃技巧屬於演算法設計方法，是比較抽象的演算法設計精神，不是指特定的演算法

- 學習 DP 的幾個階段
    - 知道狀態轉移及記錄化的概念
    - 能看懂別人定義的狀態及轉移式
    - 知道狀態定義可以自己推出轉移式
    - 可以自己想到狀態定義
    - 優化

動態規劃 - 學習技巧

- 需要自己想到新的狀態定義方式屬於比較困難的題目
- 多數中易 DP 題目都是從經典問題變化而來, 不太需要自己設計新的狀態
- 多刷題, 掌握常用的狀態轉移設計方式
- 思考經典問題可以如何變化, 變成新的題目, 能不能用類似技巧解決?

1D / 0D DP

LeetCode 53. Maximum Subarray

- 題目敘述
  - 輸入一個長度 n 的陣列 a[0], a[1], … a[n-1]
  - 選一段連續的區間出來，使得總和愈大愈好，輸出總和
- 輸入範圍
  - $n \le 30000$

Input

```
2 1 -4 7 -4 8 3 -6
```

Output

```
14
```

LeetCode 53. Maximum Subarray - 方法一

- 最後一個東西一定要選
- 狀態
  - dp(i) 表示只看 a[0] ~ a[i] 的最好答案，第 i 項一定要選
- 轉移
  - dp(i) = max{ dp(i-1) + a[i], a[i] }
- 答案
  - max{ dp(0), dp(1), dp(2), ..., dp(n-1) }

LeetCode 53. Maximum Subarray - 方法二

- 連續區間總和就是兩個前綴和的差，先找出前綴和陣列 S[ ]
- 找出滿足 i < j 的最大 S[j] - S[i]

LeetCode 198. House Robber

- 題目敘述
  - 輸入一個長度 n 的陣列 a[0], a[1], … a[n-1]
  - 選一些數字出來，讓總和盡量大
  - 相鄰兩個數字不能同時選
- 輸入範圍
  - $n \le 100$

Input

```
2 7 9 3 1
```

Output

```
12
```

LeetCode 198. House Robber - 方法一

- 最後一個東西是否有選都考慮
- 狀態
  - dp(i, 0/1) 表示只看 a[0] ~ a[i]，第 i 項有選 / 沒選，的最好答案
- 轉移
  - dp(i, 0) = max{ dp(i-1, 0), dp(i-1, 1) }
  - dp(i, 1) = dp(i-1, 0) + a[i]
- 答案
  - max{ dp(n-1, 0), dp(n-1, 1) }

LeetCode 198. House Robber - 方法二

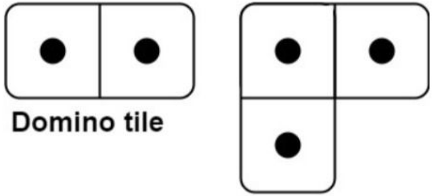
- 考慮最後一個東西是否有選沒選都可以
- 狀態
    - dp(i) 表示只看 a[0] ~ a[i] 的最好答案
- 轉移
    - dp(i) = max{ a[i] 有選, a[i] 沒選 }
    - dp(i) = max{ a[i] +dp(i-2), dp(i-1) }
- 答案
    - max{ dp(0), dp(1), dp(2), ..., dp(n-1) }

LeetCode 198. House Robber

- 考慮最後一個東西一定要選
- 狀態
    - dp(i) 表示只看 a[0] ~ a[i]，第 i 項一定要選的最好答案
- 轉移
    - dp(i, 0) = max{ dp(i-1, 0), dp(i-1, 1) }
    - dp(i, 1) = dp(i-1, 0) + a[i]
- 答案
    - max{ dp(n-1, 0), dp(n-1, 1) }

LeetCode 790. Domino and Tromino Tiling

- 題目敘述
  - 大小是 2 * n 的棋盤格，有多少種方法可以用下面兩種形狀填滿
  - 形狀可以旋轉
  - 答案模 $10^9 + 7$
- 輸入範圍
  - $n \leq 1000$



Domino tile

Tromino tile

Input

3

Output

5

LeetCode 790. Domino and Tromino Tiling

- 枚舉最後一排的兩個格子目前是否填滿
- 狀態
  - dp(n, 0/1, 0/1) 表示到第 n 排，n-1 排以前都填滿了，a[n][1] 是否有放東西，a[n][2] 是否有放東西，的方法數
- 轉移
  - dp(n, 1, 1) = dp(n-1, 1, 1) + dp(n-1, 1, 0) + dp(n-1, 0, 1)
  - dp(n, 0, 1) = dp(n-1, 0, 0) + dp(n-1, 1, 0)
  - dp(n, 1, 0) = dp(n, 0, 1)
  - dp(n, 0, 0) = dp(n-1, 1, 1)
- 答案
  - dp(n, 1, 1)

1D / 1D DP

LeetCode 300. 最長遞增子序列 (LIS)

- 題目敘述
  - 給一個長度 n 的陣列 a[0], a[1], a[2], ..., a[n-1]
  - 找出一個最長的子序列，裡面的值是遞增的
- 測資範圍
  - $n \le 2500$

2	1	4	7	4	8	3	6	4	7

LeetCode 300. 最長遞增子序列 (LIS)

- 考慮最後一個東西一定要選
- 狀態
    - dp(i) 表示只看 a[0] ~ a[i]，第 i 項一定要選的最好答案
- 轉移
    - dp(i) = a[i] + max{ dp(j) | j < i 且 a[j] < a[i] }
- 答案
    - max{ dp(0), dp(1), dp(2), ..., dp(n-1) }

LIS 其實還有 o(n log n) 的演算法

LeetCode 300. 最長遞增子序列 (LIS)

- 可能的變化問題:
  - 最長連續子陣列
  - 輸出 LIS 的個數
  - 有權重的 LIS
  - 最長非嚴格遞增子序列
  - 輸出一個 LIS
  - 輸出最小字典順序的LIS
  - 輸出最大字典順序的LIS

LeetCode 279. Perfect Squares

- 題目敘述
  - 給一個數字 n，問 n 最少可以用多少個完全平方數相加組成
- 測資範圍
  - n ≤ 10000

Input

```
12
```

Output

```
3
```

```
12 = 4 + 4 + 4
```

LeetCode 279. Perfect Squares

- 考慮最後一個拿的數字是什麼
- 狀態
  - dp(i) 表示輸入 i 的答案，也就是最少可以表示成幾個平方數相加
- 轉移
  - dp(i) = min{ dp(i-k*k) + 1 | k*k 不超過 i }
- 答案
  - max{ dp(n) }

自編題：分組最大值總和最小化

- 題目敘述
  - 給一個整數 K 以及長度 n 的陣列 a[0], a[1], a[2], ..., a[n-1]
  - 把陣列元素分組，每個東西都要被分到某個組別
  - 每個組別的東西要是連續的，且最多包涵K 個數字
  - 每個組別的最大值相加後要愈小愈好，輸出這個最小可能的值
- 測資範圍
  - n, K ≤ 1000

Input

```
8 3
2 1 4 8 3 2 5 2
```

Output

```
15
```

2 + 8 + 5

自編題：分組最大值總和最小化

- 考慮最後一段拿的區間是什麼
- 狀態
  - dp(i) 表示只看陣列 a[0] ~ a[i] 的答案
- 轉移
  - dp(i) = max{ dp(j) + max{a[j+1], a[j+2], ..., a[i] ｜ j < i }
- 答案
  - dp(n-1)

題單 - 1D/0D

- leetcode 70. Climbing Stairs
- leetcode 53. Maximum Subarray
- leetcode 790. Domino and Tromino Tiling
- leetcode 152. Maximum Product Subarray
- leetcode 978. Longest Turbulent Subarray
- leetcode 1191. K-Concatenation Maximum Sum

題單 - 1D/1D

- leetcode 300. Longest Increasing Subsequence
- leetcode 354. Russian Doll Envelopes
- leetcode 646. Maximum Length of Pair Chain
- leetcode 673. Number of Longest Increasing Subsequence
- leetcode 960. Delete Columns to Make Sorted III