# 基礎動態規劃

暴力走不遠.....

by ray,青月喵

# 關於DP

動態規劃（Dynamic Programming, DP）是一種演算法的設計方式。

不具備演算法定義上固定的指令集和流程，也因此解題競賽經常出現。

# 先備知識&面向客群

- Big O-要會估計
- 爆搜-暴力解推回dp解
- recursive-直觀實作

適合不會DP的競程小白

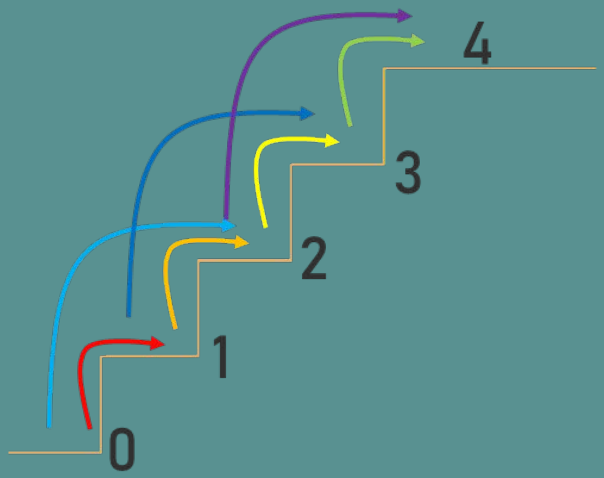
# Outline

- 爬樓梯問題
- What is dp?
- 經典問題：
  - 切棍子問題
  - extra:找零錢問題
  - 0/1 背包問題
  - 最長遞增子序列(LIS)
- 回溯法、滾動陣列
- 實作 by C++

# 小試身手-爬樓梯問題

# 簡單的問題

每一階樓梯可以往上跨1或是2步，請問跨到第n階有幾種方式？



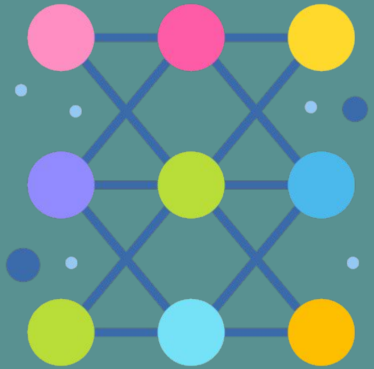

n=4有五種方法

# 走到第N階的方法數是第n-1階的方法數加上第n-2階的方法數!!

我們就可以........

1.從第n層慢慢遞迴下去(Top-down)
2.從第0階慢慢往上推(Bottom-up)

# What is DP?



# 將大問題轉成同性質小問題

則小問題便會因為性質相同，能變成更小的問題，直到規模足夠小的時候，能直接得知答案=>相當於求原問題的遞迴解

ex:前面提到的爬樓梯問題

# 不重複計算

ex:費氏數列,
f(n)=f(n-1)+f(n-2)
f(1)=1,f(2)=1

```
                       f(6)
                /                  \
            f(5)                  f(4)
        /          \          /          \
     f(4)          f(3)    f(3)          f(2)
    /     \      /     \   /    \
 f(3)    f(2) f(2) f(1) f(2) f(1)
 /    \
f(2) f(1)
```

好多重複計算....如果把算過的資訊都記錄起來呢？

# 只要算六次！

```
                    f(6)
             /                   \
          f(5)                   f(4)
        /       \
      f(4)       f(3)
    /     \
  f(3)    f(2)
 /    \
f(2) f(1)
```

# 狀態壓縮

不儲存不需要的狀態，壓縮掉資訊，以提高執行效率
ex:爬樓梯的dp[5]

```
1 1 1 1 1
1 1 1 2
1 1 2 1
1 2 1 1
2 1 1 1
1 2 2
2 1 2
2 2 1
```

#如果某些狀況在未來擁有相同發展，便可以被壓成相同狀態來看待

# 轉移

轉移就是如何將原本問題與子問題的解產生關聯，進而從原本的解，求得原問題的解

也可以這樣想：假如你已經知道所有的子問題的答案，如何拿來求原問題的答案？

# 複雜度估計

複雜度基本上就是狀態數量乘以狀態轉移的複雜度

爬樓梯問題需要計算的狀態量為n個，空間複雜度為O(n)
時間複雜度為狀態數量乘以2，共是2n，時間複雜度為O(n)

# 總結

1.當你看到一個題目能切成相同性質的子問題時=>往DP的方向去思考

2.試著去定義狀態，並盡量壓縮掉無意義的資訊(避免陣列開太大)

3.考慮狀態轉移式，注意邊界問題

4.複雜度是否合理，如果太大，試著優化狀態轉移式，若還是不行就得重新定義狀態

# 最難的地方-定義狀態轉移式

只能靠多刷題累積觀察經驗

# 一 : 計數型

一般會叫你計算出總共有多少情況
ex:前面提到的爬樓梯問題,爬格子問題.....

計數型的轉移必須包含所有可能情形，才不致於少算。
各種情形必須完全獨立不重覆，才不致於多算。

# 二:最佳化型

如果將爬樓梯問題改成最佳化型:

走上第i階階梯要花cost[i](cost[i]>0)元，從第0階開始往上，每次可以跨一階或是兩階，請問走到第n階花費最少要付多少錢？

# 其實也很像.....

第n階的最小花費要不來自第n-1階或n-2階再加上走到第n階的費用

dp[n]=min(dp[n-1],dp[n-2])+cost[n]

# 注意！

使用DP的前提是必須證明「子問題的最佳解能得到原問題的最佳解」

如果踏上第 5 階時，最佳解是付 10 塊錢。
假設存在一組踏上第 n （n > 5） 階的最佳解，在第 5 階付了 12 塊錢。

那麼我們改採用踏上第 5 階時，只要付 10 塊錢的走法，
跟著最佳解的走法走，則第 5 階之後所付的錢會跟假設的最佳解完全相同。
第 5 階之後付的錢完全相同，但在踏上第 5 階時少付了 2 塊錢，比最佳解更佳，因此矛盾。

故不存在任何最佳解，其過程的子問題並非是最佳的。

# 舉個反例

假設我們現在尋求走上第n階時，個位數最小的花費

則我們在第 5 階若有個位數 6 和 9 兩種可能，此時最佳花費應為 6；
但踏上第 6 階時若花 2 塊錢，則最佳花費應是 9+2 得到個位數 1，而非 6+2 的 8。

這題就不能用DP解嗎？

# 錯誤原因&改善

只存最小花費=>未必使之後的解是最小花費

要多存什麼資訊？

# 增維

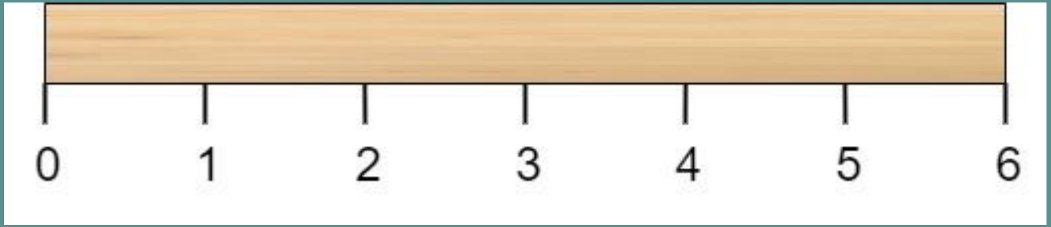
除了把狀態完全換掉之外，對狀態追加描述、進行增維，也可能解決這種問題。

```
將狀態改為 dp[i][j] 代表踏上第 i 階時，花費個位數 j 是否可能達成。
dp[i][j] = (dp[i-1][k] || dp[i-2][k])  // k 滿足 k+cost[i] 個位數為 j
```

# 切棍子問題

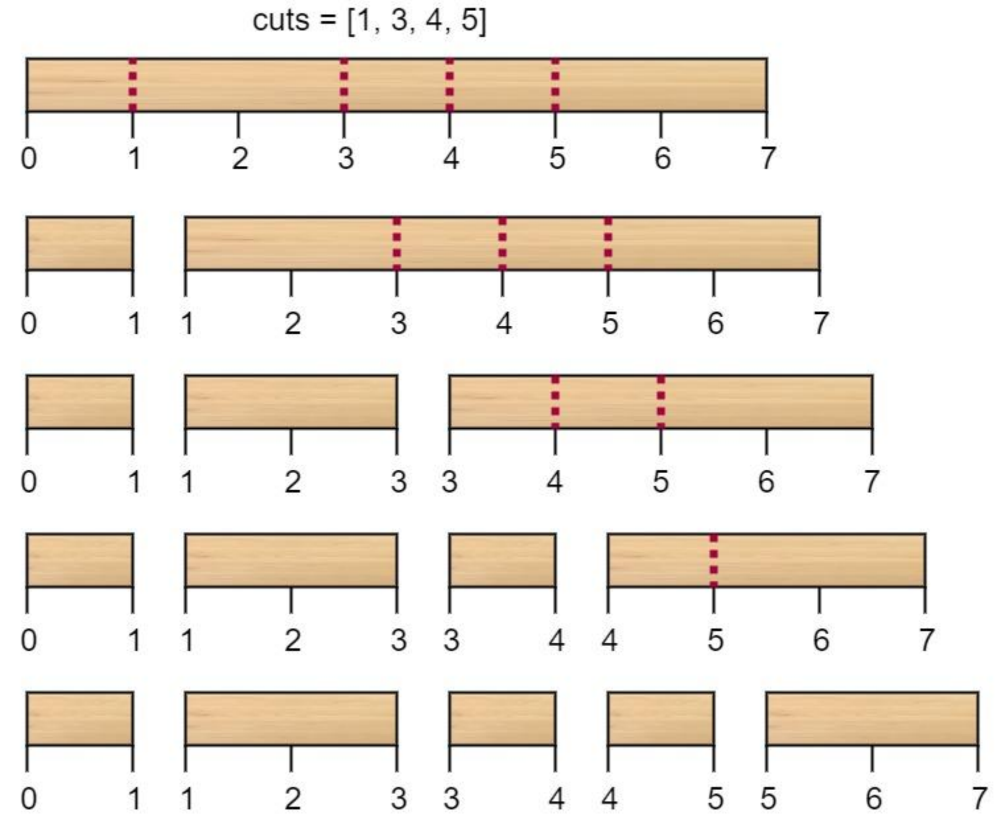
# 切棍子的最小成本（leetcode 1547）

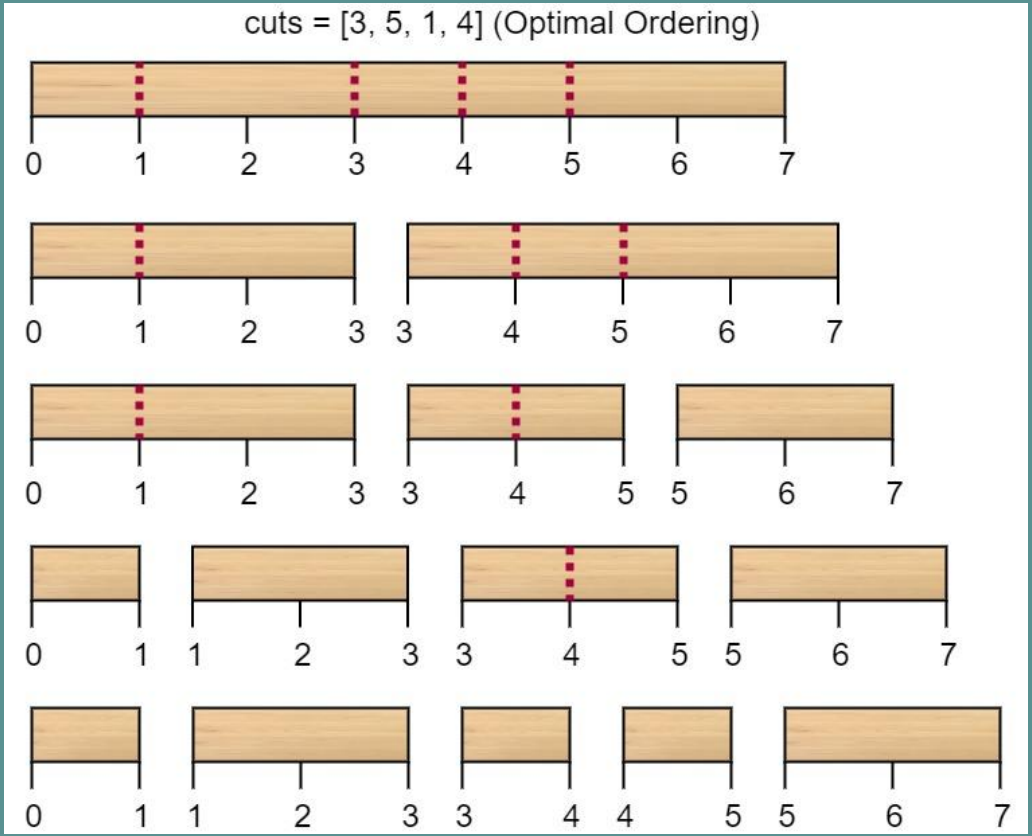
有一根長度為n的木棍，cuts數組標記棍子上一些需要被切開的位置，每次切開所花費的花費是那條木棍的長度，求最佳切割的順序下的最小成本



input: n=7,cuts=[1,3,4,5]

7+6+4+3=20




cuts = [3, 5, 1, 4] (Optimal Ordering)
0 1 2 3 4 5 6 7
0 1 2 3 3 4 5 6 7
0 1 2 3 3 4 5 5 6 7
0 1 1 2 3 3 4 5 5 6 7
0 1 1 2 3 3 4 4 5 5 6 7

$7+4+3+2=16$

# 有Greedy解嗎？

# 暴力解？ O(m!)

# 假設我們已經知道所有子問題的解了.....

$$dp(i,j)=\min(dp(i,k)+dp(k,j))+cuts[j]-cut[i] \quad ,k=i+1,.....,j-1$$

棍子的最小花費就是枚舉第一刀的切法後，兩條子木棍的最小花費總和加上該木棍的長度！

# Top-down實作

```
int stick(int st,int en)
{
    if(en-st==1)return 0;
    if(dp[st][en]!=0)return dp[st][en];
    for(int i=st+1;i<en;i++)
    {
        if(dp[st][en]==0)
        {
            dp[st][en]=stick(st,i)+stick(i,en)+(cuts[en]-cuts[st]);
        }
        else
        {
            dp[st][en]=min(dp[st][en],stick(st,i)+stick(i,en)+(cuts[en]-cuts[st]));
        }
    }
    return dp[st][en];
}
```

# 複雜度

假設cuts裏面有m個元素，每次狀態轉移要花O(m)，總共有m^2種狀態，故複雜度是O(m^3)

# 複雜度

假設cuts裏面有m個元素，每次狀態轉移要花O(m)，總共有m^2種狀態，故複雜度是O(m^3)

# extra:找零錢問題

# 題敘(from Lucky cat)

給你一個金額（ n cents），請你回答共有多少種硬幣組合的方式。例如：n=11，那麼你可以有以下4種硬幣的組合：

1. 1個 10 cent的硬幣加上1個 1 cent的硬幣
2. 2個 5 cent的硬幣加上1個 1 cent的硬幣
3. 1個 5 cent的硬幣加上6個 1 cent的硬幣
4. 11個 1 cent的硬幣

**p.s 美國的零錢共有以下5種硬幣以及其面值：**

- **penny, 1 cent**
- **nickel, 5 cents**
- **dime, 10 cents**
- **quarter, 25 cents**
- **half-dollar, 50 cents**

請注意：n=0 我們算他是有一種方式。

# 直觀想法

用最常見的切尾巴去思考......

窮舉硬幣面額1,5,10,25,50作為尾巴.....

# 狀態轉移式(？

dp[n] = dp[n-1] + dp[n-5] + dp[n-10] + dp[n-25] + dp[n-50]

喔耶~AC~~~

# 好像不太對...

dp[1]=1,dp[5]=2

假設用剛剛的式子算n=6, 則

dp[6]=dp[6-1]+dp[6-5]=3

1 1 1 1 1 1

5 1

明明只有兩種……

# 重複計算了QQ

剛剛的式子會重複計算1+5跟5+1

1 1 1 1 1 1

1 5

5 1

# 如何避免？

為了避免這種排列順序不同，組合卻相同的情形，我們希望讓小的先放、大的後放，
這樣保證了順序必定由小至大，不會反過來，不會出現僅排列不同的問題。

# 思考轉移式

考慮前 n 種硬幣組成金額 m 的情形，可以分成使用第 n 種硬幣，以及不使用，共兩種可能。
不使用的話就只靠前 n-1 種硬幣；使用的話就必須前 n 種組成 m-value[n]，
加上第 n 種硬幣剛好金額 m。

# 轉移式

```
dp[n][m] = k 代表以面額前 n 大的硬幣，組成總金額 m 的方法數為 k
dp[n][m] = dp[n-1][m] + dp[n][m-value[n]]
```

dp[n-1][m] 定義上必不包含第 n 種硬幣;
而 dp[n][m-value[n]] 則包含 0 至多個第 n 種硬幣，加上一個第 n 種硬幣，則至少有 1 個。
於是兩邊獨立不交集，又互相補完0 和多個第 n 種硬幣的可能情形。

# 滾動陣列

考慮第 n 種硬幣時，從轉移來看，實際上只需要 dp[n] 和 dp[n-1] 兩條陣列。
因此，可用兩條陣列 p, q 分別代表 dp[n-1] 和 dp[n]，之後交替代表 dp[n] 和 dp[n-1]，
即為滾動數組，可將空間複雜度自 nm 降至 2n。

# 滾動陣列

考慮第 n 種硬幣時，從轉移來看，實際上只需要 dp[n] 和 dp[n-1] 兩條陣列。
因此，可用兩條陣列 p, q 分別代表 dp[n-1] 和 dp[n]，之後交替代表 dp[n] 和 dp[n-1]，
即為滾動數組，可將空間複雜度自 nm 降至 2n。

# 兩個陣列的實作方式

1.視奇偶互相交替使用

2.將新算的結果複製到舊的

3.宣告 int dp[2][M]; 用 dp[n&1] 和 dp[(n-1)&1] 分別代表

4.宣告兩條陣列和兩個指標 p, q 並透過交換 p, q 來輪替

聽起來已經不錯了


但是我拒絕！

# 我只想用一條陣列啦！

考慮到方向性，m一定是從金額較小的m-value[n]轉移落來的，方向單一，故可用一條陣列

而剛進到第n個硬幣時，陣列存的是n-1時的內容，便可寫成：

```
dp[m] += dp[m-value[n]];
```

```
for (int i=value[n]; i<=M; i++)
{
    dp[i] += dp[i-value[n]];
}
```

注意for迴圈的順序，如果寫反，意義將完全不同，答案就是錯的

# 背包問題

# 題目敘述

有 n 個物品, 每個物品有自己的價值 : c[i] 和 體積 : w[i]
還有一個容量為 m 的背包

求最佳情況下, 可以放入背包的最高價值

# Greedy?

如果我們直接根據每個物品的價值除以體積來挑呢?

反例：

當背包容量為 6
且有右邊四個物品

不會是最佳解

| | 價值 | 體積 | CP值 |
|---|---|---|---|
| 物品A | 2 | 3 | 0.66 |
| 物品B | 4 | 3 | 1.33 |
| 物品C | 2 | 4 | 0.5 |
| 物品D | 5 | 4 | 1.25 |

# 暴力?

直接遍歷所有組合呢？

時間拖太久

# 以DP觀點來思考

※壓縮狀態
　知道有更高價值的解，就不需要紀錄

※最簡單的狀況
　當沒有物品時，背包體積 0 ~ m 的最佳解？

※同性質小問題
　已知 i – 1 個物品，體積 0 ~ m 的解
　→ 前 i 個物品，背包體積 0 ~ m 的解？

# 初始條件及轉移式

初始化 :

for( j : 0 ~ m ) dp[ 0 ][ j ] = 0

轉移 :

dp[ i ][ j ] = max(
dp[ i-1 ][ j ],
dp[ i-1 ][ j-weight[ i ] ]+ price[ i ]    )

# 舉例來說

| | 價值 | 體積 | CP值 |
|---|---|---|---|
| 物品A | 2 | 3 | 0.66 |
| 物品B | 4 | 3 | 1.33 |
| 物品C | 2 | 4 | 0.5 |
| 物品D | 5 | 4 | 1.25 |

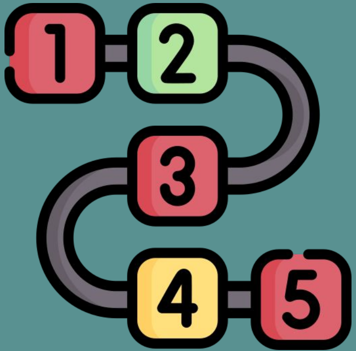
# 最長遞增子序列 (LIS)

# 題目敘述

給一個長度為 n 的陣列

挑出陣列中幾個遞增的元素,

找到所有可能中, 最長會是多長

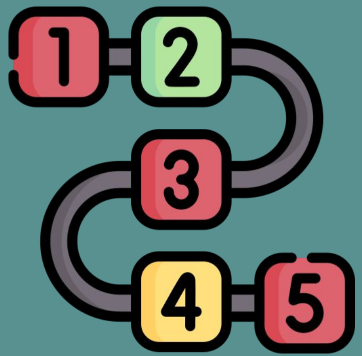


# 以DP觀點來思考

※壓縮狀態
知道有更長的解，就不需要紀錄

※最簡單的狀況
當陣列長度為 1 的時候，解為？

※同性質小問題
已知 0 ~ i – 1 前所有陣列的解
→ 0~i 陣列的解？



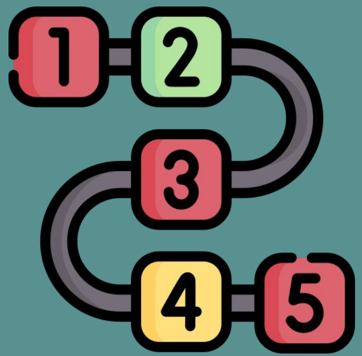
# 初始化條件及轉移式

初始化 :

for( i : 0 ~ n ) dp[ i ] = 1

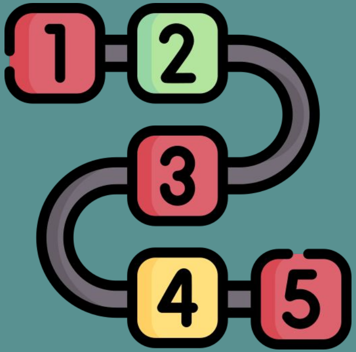
轉移 :

```
if( arr[ i ] > arr[ j ] )
  dp[ i ] = max(   dp[ i ],   dp[ j ] + 1   )
```



# 舉例來說

arr = {6, 1, 5, 2, 8, 4, 7, 3}



# 回溯法

# 用途

如果題目要求的不只是最佳解為多少

而是要如何得到最佳解呢?

# 滾動陣列

# 用途

動態規劃中, 有些資訊再也不會用到了

有辦法再進一步壓縮記憶體空間嗎?