

# 動態規劃

## Introduction to Dynamic Programming

2024 南九校聯合寒訓 - 演算法組

講師：陳俊安 Colten

2024.02.06

# 動態規劃 Dynamic Programming

- 是一個把陣列名字叫做 dp 的技巧
  - 沒錯
- 是動態？還是規劃？
  - 他既非動態也非規劃
- 動態規劃是一個透過小的子問題解決大問題的技巧
  - 很像分治 (Divide and Conquer) 大事化小，小事化無
- 那為什麼叫動態規劃？

# 動態規劃 Dynamic Programming

- 我特別去找了一下資料，結果發明動態規劃的人的自傳有寫名子由來
- 發明動態規劃的人是 Bellman
- 他在他的自傳 [《Eye of the Hurricane: An Autobiography》](#) 有提到
- 有興趣的可以參考 [這個連結](#)
- 為了給大家一點期待感我現在不想公布答案：P

# 動態規劃 Dynamic Programming

- 動態規劃是什麼？
  - 在這邊我用一句話解釋動態規劃是什麼
    - 長江後浪推前浪，一替新人換舊人
  - 這句話呼應了動態規劃最核心的想法
  - 用以前的資訊來幫助我們得到當前最新的資訊

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

- 可以把動態規劃想像成是一個公式
- 只要我們把公式需要的資訊先取得了，就可以慢慢往後推得後面的資訊
- $f(0) = 0$  ,  $f(1) = 1$ 
  - 有了以上這兩個資訊之後我們就可以依序算出  $f(2)$ ,  $f(3)$   $\cdots$   $f(n)$
  - 求出  $f(n)$  的時間複雜度為  $O(n)$

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

- 這樣子從底層往上慢慢推出後面資訊的方式稱為 Bottom-Up
- 與 Bottom-Up 相反的方式稱為 Top-Down
- 我們一樣來看這一個例子

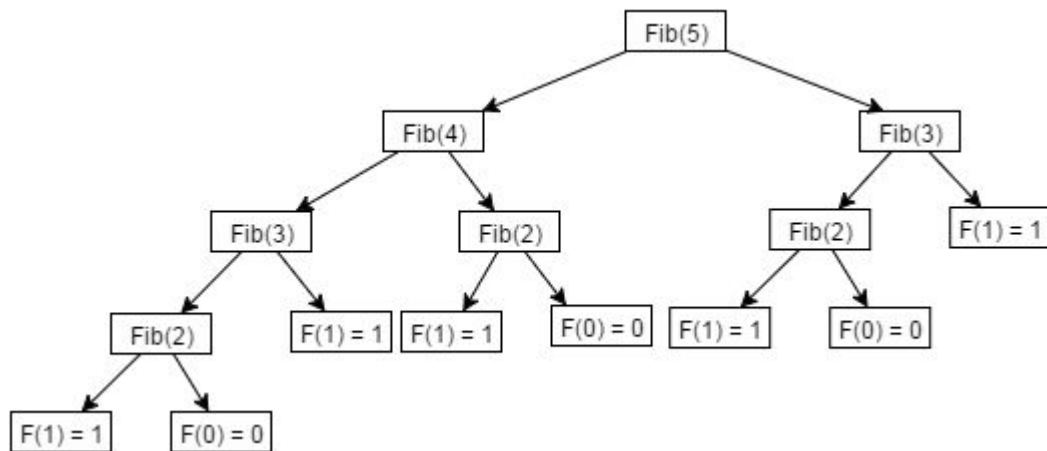
$$f(n) = f(n-1) + f(n-2)$$

- 我們嘗試使用遞迴求得  $f(n)$  是多少
- 求得  $f(n)$  需要  $f(n-1)$  與  $f(n-2)$  的資訊
- 遞迴的終止條件則為  $f(0)$  與  $f(1)$

```
6
7 int f(int n)
8 {
9     if( n == 0 ) return 0;
10    if( n == 1 ) return 1;
11
12    return f(n-1) + f(n-2);
13 }
```

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

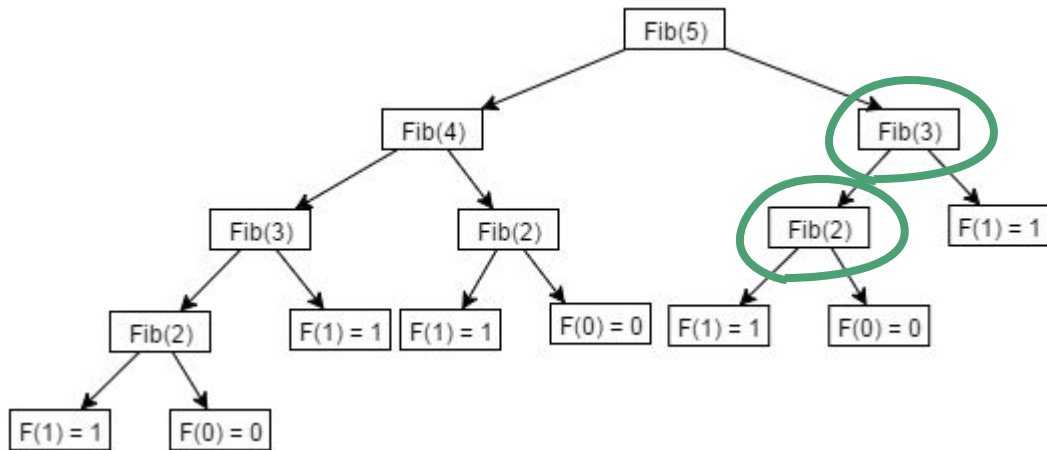
- 這樣子的時間複雜度是  $O(n)$  ㄐ ?
- 我們畫出遞迴樹來看看





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

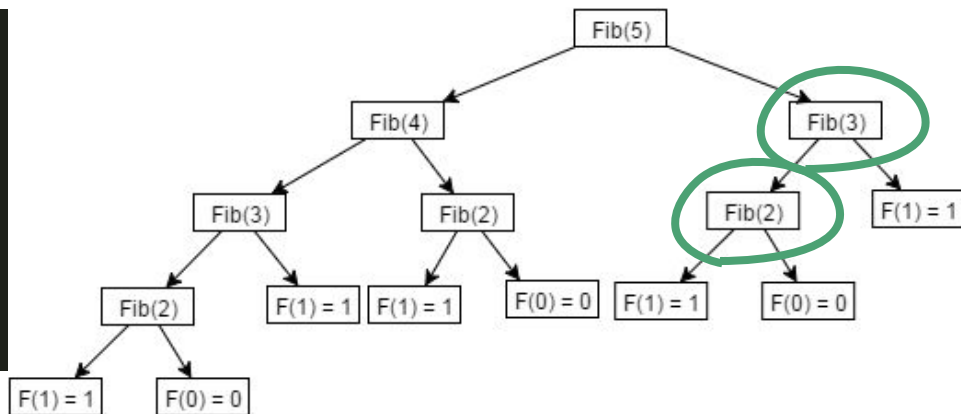
- 你會發現有地方我們重複計算了
- 先前已經求過  $f(2)$  與  $f(3)$  了



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

- 因此如果我們把之前算過的存起來
- 如果某次突然要使用到之前算過的資訊就可以直接拿出來用了
- 這就是動態規劃 Top-Down 的精神

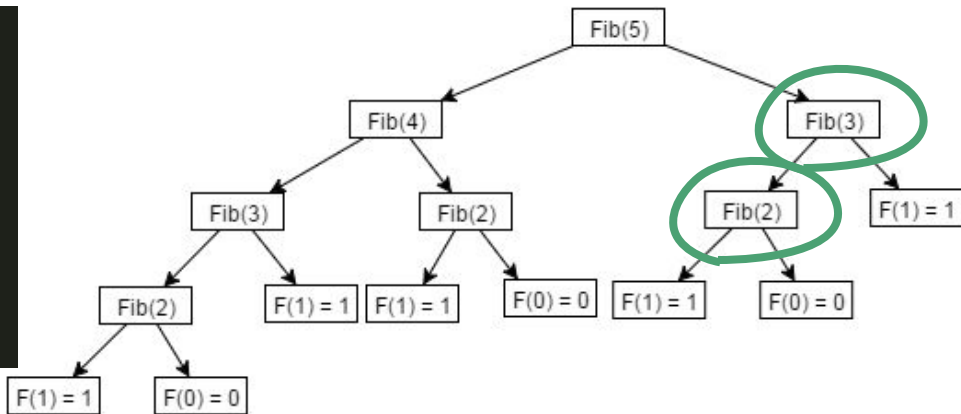
```
6
7 int dp[101];
8
9 int f(int n)
10 {
11     if( n == 0 ) return 0;
12     if( n == 1 ) return 1;
13
14     if( dp[n] != 0 ) return dp[n]; // 算過了，直接拿以前算出來的東西
15     else
16     {
17         dp[n] = f(n-1) + f(n-2);
18         return dp[n];
19     }
20 }
```



# 時間複雜度？

- 因為這樣子我們可以保證這  $n$  個東西我們只會算過 1 次
- 時間複雜度又變回乾淨的  $O(n)$  了

```
7 int dp[101];
8
9 int f(int n)
10 {
11     if( n == 0 ) return 0;
12     if( n == 1 ) return 1;
13
14     if( dp[n] != 0 ) return dp[n]; // 算過了，直接拿以前算出來的東西
15     else
16     {
17         dp[n] = f(n-1) + f(n-2);
18         return dp[n];
19     }
20 }
```



# Top-Down v.s Bottom-Up

- Top-Down 的缺點
  - 時間複雜度常數大
    - pass by value & pass by reference
    - 遞迴太深會導致 stack overflow
- Top-Down 的優點
  - 轉移式很直覺，只要記得把算過的存起來就好

# Top-Down v.s Bottom-Up

- Bottom-Up 的缺點
  - 轉移式比較不直覺
- Bottom-Up 的優點
  - 寫起來很乾淨，時間複雜度常數小
- Bottom-Up 是動態規劃最常見的使用方法
- Top-Down 一個不小心遞迴太深就出大事了

## 排列組合動態規劃：[CSES Problem Set Dice Combinations](#)

- 骰子有 1 ~ 6 點，現在可以骰無限顆骰子，依序骰每一個骰子
- 求最後共有幾種骰法會使所有骰子的點數和為  $n$
- Example :
  - $n = 3$  ,  $\text{answer} = 4$
  - $1 + 1 + 1$
  - $2 + 1$
  - $1 + 2$
  - $3$

## 排列組合動態規劃：[CSES Problem Set Dice Combinations](#)

- 動態規劃的第一個步驟都是 定義狀態
- 有點類似定義一個 Function 的概念
- 像是這題我們會定義  $dp[i] =$  骰出點數為  $i$  的組合數

## 排列組合動態規劃：[CSES Problem Set Dice Combinations](#)

- 定義完轉移式之後接下來就可以開始把公式推出來了
- 對於點數  $i$  來說，要使骰出的點數總和為  $i$ ，會有 6 種可能
  - 點數  $i - 6$  時再骰出 1 個 6 點
  - 點數  $i - 5$  時再骰出 1 個 5 點
  - 點數  $i - 4$  時再骰出 1 個 4 點
  - and so on...
- 因此轉移式為  $dp[i] = dp[i-1] + dp[i-2] + \dots + dp[i-6]$ 
  - 加法原理



## 排列組合動態規劃：[CSES Problem Set Dice Combinations](#)

- 如此一來，很簡單的就可以用迴圈解決了，時間複雜度  $O(n)$
- 題目有說答案可能很大，只要輸出  $\text{mod } 10^9 + 7$  的結果就好

```
23 const int mod = 1e9 + 7;
24
25 signed main(void)
26 {
27     int n;
28     cin >> n;
29
30     dp[0] = 1;
31
32     for(int i=1;i<=n;i++)
33     {
34         for(int k=1;k<=6;k++)
35         {
36             if( i - k >= 0 ) dp[i] += dp[i-k] , dp[i] %= mod;
37         }
38     }
39
40     cout << dp[n] << "\n";
41
42     return 0;
43 }
```

## 排列組合動態規劃：[CSES Problem Set Coin Combinations I](#)

- 有  $n$  種硬幣，每種硬幣的面額分別是  $c_i$
- 接下來每一次你可以選擇其中一種硬幣 (可以重複拿一樣的)
- 求最後湊出總金額  $x$  的選法有幾種

For example, if the coins are  $\{2, 3, 5\}$  and the desired sum is 9, there are 8 ways:

- $2 + 2 + 5$
- $2 + 5 + 2$
- $5 + 2 + 2$
- $3 + 3 + 3$
- $2 + 2 + 2 + 3$
- $2 + 2 + 3 + 2$
- $2 + 3 + 2 + 2$
- $3 + 2 + 2 + 2$

## 排列組合動態規劃：[CSES Problem Set Coin Combinations I](#)

- 定義轉移式： $dp[i]$  = 湊出總和為  $i$  的湊法有幾種
- 對於總和  $i$  來說，你有可能是透過：
  - $i - c_1$  再拿 1 個  $c_1$  硬幣得來的
  - $i - c_2$  再拿 1 個  $c_2$  硬幣得來的
  - $i - c_3$  再拿 1 個  $c_3$  硬幣得來的
  - and so on...
- 因此你會發現轉移式跟骰子那一題一樣
- 差別只在於骰子固定  $1 \sim 6$ ，硬幣是  $c_1 \sim c_n$

## 排列組合動態規劃：[CSES Problem Set Coin Combinations I](#)

- 對於每一個總和  $i$  我們都需要去枚舉  $c_1 \sim c_n$
- 因此整體時間複雜度為  $O(n \times m)$
- 我自己變數  $x$  是取名叫做  $m$
- 因為我比較叛逆一點
  - $m$  剛好在  $n$  旁邊
  - 打字會比較快

```
17  int n,m;
18  cin >> n >> m;
19  vector<int> a(n);
20
21  for(int i=0;i<n;i++)
22  {
23      cin >> a[i];
24  }
25
26  dp[0] = 1;
27
28  for(int i=1;i<=m;i++)
29  {
30      for(int k=0;k<n;k++)
31      {
32          if(i - a[k] >= 0)
33          {
34              dp[i] += dp[i-a[k]];
35              dp[i] %= mod;
36          }
37      }
38  }
39
40  cout << dp[m] << "\n";
```

## 選擇困難的動態規劃：[Atcoder DP Contest Frog 1](#)

- 現在有一隻青蛙在第一個石頭， $n$  個石頭，每一個石頭的高度數字  $h_i$
- 這一隻青蛙每一次只能跳 1 格或 2 格
- 如果原本在高度  $a$  的石頭，跳到高度  $b$  的石頭需要花費  $|a - b|$
- 求青蛙最後跳到第  $n$  個石頭所需要的最少花費

## 選擇困難的動態規劃：[Atcoder DP Contest Frog 1](#)

- 對於青蛙第  $i$  個石頭來說只有兩種可能
  - 從第  $i - 1$  個石頭跳過來
  - 從第  $i - 2$  個石頭跳過來
- 所以如果我們能算出跳到  $i - 1$  與  $i - 2$  的最佳答案，就可以求得跳到  $i$  的最佳答案

## 選擇困難的動態規劃：[Atcoder DP Contest Frog 1](#)

- 定義  $dp[i]$  = 跳到  $i$  所需要的最小花費
- $dp[1] = 0$  ,  $dp[2] = |h_1 - h_2|$
- 如果第  $i - 2$  個石頭的高度是  $a$  , 第  $i - 1$  個石頭的高度是  $b$
- 從第  $i - 2$  個石頭跳過來所需花費為  $dp[i-2] + |a - h_i|$
- 從第  $i - 1$  個石頭跳過來所需花費為  $dp[i-1] + |b - h_i|$

## 選擇困難的動態規劃：[Atcoder DP Contest Frog 1](#)

- 要選出最好的方案，因此整個轉移式合併在一起就會變成：
- $dp[i] = \min( dp[i-2] + |a - h_i| , dp[i-1] + |b - h_i| )$
- 整體時間複雜度  $O(n)$

```
24
25     vector<int>a(n+1),dp(n+1,(int)1e9);
26
27     for(int i=1;i<=n;i++) cin >> a[i];
28
29     dp[1] = 0 , dp[2] = abs(a[2]-a[1]);
30
31     for(int i=3;i<=n;i++)
32     {
33         dp[i] = min( dp[i-1] + abs(a[i]-a[i-1]) , dp[i-2] + abs(a[i]-a[i-2]) );
34     }
35
36     cout << dp[n] << "\n";
```



## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- Colten 放暑假只會做 3 件事情
  - 寫程式
  - 水餃
  - 睡覺
- 如果在第  $i$  天做第 1 件事情 Colten 會得到  $a_i$  的快樂度
- 如果在第  $i$  天做第 2 件事情 Colten 會得到  $b_i$  的快樂度
- 如果在第  $i$  天做第 3 件事情 Colten 會得到  $c_i$  的快樂度

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- Colten 不會連續 2 天做同一件事情
- 求如果有  $n$  天, Colten 在最佳規劃下, 快樂度最大可以是多少？

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- 對於每一天會有 3 種選擇
- 定義  $dp[i][k]$  = 如果第  $i$  天做第  $k$  件事情能得到的最大快樂度
- $dp[1][1] = a1$  ,  $dp[1][2] = a2$  ,  $dp[1][3] = a3$

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- 對於  $dp[i][1]$  來說
  - 前一天（第  $i - 1$  天）不能也做第 1 件事情
- 對於  $dp[i][2]$  來說
  - 前一天（第  $i - 1$  天）不能也做第 2 件事情
- 對於  $dp[i][3]$  來說
  - 前一天（第  $i - 1$  天）不能也做第 3 件事情

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- 如果第  $i$  天要做第 1 件事情
  - 第  $i - 1$  天只能做第 2、3 件事情
  - 可以列出轉移式
    - $dp[i][1] = \max( dp[i-1][2] , dp[i-1][3] ) + a_i$
- 如果第  $i$  天要做第 2 件事情
  - 第  $i - 1$  天只能做第 1、3 件事情
  - 可以列出轉移式
    - $dp[i][2] = \max( dp[i-1][1] , dp[i-1][3] ) + b_i$

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- 如果第  $i$  天要做第 3 件事情
  - 第  $i - 1$  天只能做第 1、2 件事情
  - 可以列出轉移式
    - $dp[i][3] = \max( dp[i-1][1] , dp[i-1][2] ) + c_i$

## 選擇困難的動態規劃：[Atcoder DP Contest Vacation](#)

- 因此只要把每一天的所有選擇的最佳答案計算出來，就可以一直往後推出最佳的答案
- 最後答案為  $\max(dp[n][1], dp[n][2], dp[n][3])$
- 時間複雜度： $O(n)$

```
23  int n;  
24  cin >> n;  
25  
26  for(int i=1;i<=n;i++)  
27  {  
28      for(int k=1;k<=3;k++)  
29      {  
30          cin >> a[i][k];  
31      }  
32  }  
33  
34  dp[1][1] = a[1][1], dp[1][2] = a[1][2], dp[1][3] = a[1][3];  
35  
36  for(int i=2;i<=n;i++)  
37  {  
38      dp[i][1] = max(dp[i-1][2], dp[i-1][3]) + a[i][1];  
39      dp[i][2] = max(dp[i-1][1], dp[i-1][3]) + a[i][2];  
40      dp[i][3] = max(dp[i-1][1], dp[i-1][2]) + a[i][3];  
41  }  
42  
43  cout << max({dp[n][1], dp[n][2], dp[n][3]}) << "\n";  
44 }
```

## 0 / 1 背包問題

- 動態規劃當中最具代表性的問題之一
- 題目會給  $n$  個物品，容量  $m$  的背包
- 第  $i$  個物品會佔據背包  $w_i$  的容量，價值為  $v_i$
- 你的目標是最後讓背包裡的所有物品價值越高越好



## 0 / 1 背包問題

- 定義  $dp[i][k]$  表示考慮前  $i$  個物品的情況下，背包容量為  $k$  時所能得到的最大價值
- 對於  $dp[i][k]$  來說，只有兩種選擇
  - 拿第  $i$  個物品
  - 不拿第  $i$  個物品

## 0 / 1 背包問題

- 如果我們要拿第  $i$  個物品，且當前背包只有  $k$  的容量
- 那在我們考慮前  $i - 1$  個物品時，背包容量只能有  $k - w_i$ 
  - 因為拿第  $i$  個物品會佔據掉  $w_i$  的容量
  - 如果在考慮前  $i - 1$  個物品時就用掉了超出  $k - w_i$  的容量，第  $i$  個物品是裝不下容量只有  $k$  的背包的
- 因此如果我們要拿第  $i$  個物品，轉移式為：
  - $dp[i][k] = dp[i-1][k-w_i] + v_i$

## 0 / 1 背包問題

- 如果我們不拿第  $i$  個物品，且當前背包只有  $k$  的容量
- 轉移式很簡單：
  - $dp[i][k] = dp[i-1][k]$
- 把這兩種可能的轉移式合併在一起就會變成
- $dp[i][k] = \max( dp[i-1][k] , dp[i-1][k-w_i] + v_i )$
- 而我們最後要求的答案是  $dp[n][m]$  (  $n$  個物品、背包容量  $m$  )
- 因此我們就必須依序求出  $dp[1][0\sim m]$ ,  $dp[2][0\sim m]$ ,  $\cdots dp[3][0\sim m]$

## 0 / 1 背包問題

- 求出  $dp[i][0\sim m]$  之前必須先把  $dp[i-1][0\sim m]$  求得
- 而我們已知  $dp[0][0\sim m]$  的結果都是 0
- 因此我們可以從底部開始推答案，推出  $dp[n][m]$  的結果
- 這就是動態規劃最重要的核心精神
- 整體時間複雜度： $O(nm)$

## 0 / 1 背包問題

```
28
29     for(int i=1;i<=n;i++)
30     {
31         for(int k=0;k<=m;k++)
32         {
33             if( k - w[i] >= 0 ) dp[i][k] = max( dp[i-1][k] , dp[i-1][ k - w[i] ] + v[i] );
34             else dp[i][k] = dp[i-1][k];
35         }
36     }
37
38     cout << dp[n][m] << "\n";
```

# 無限背包問題

- 跟 0 / 1 背包問題要求的東西一樣
- 只是每一個物品的數量是無限的

# 無限背包問題

- 由於物品可以重複拿，我們的狀態就不用特別註明當前是考慮前幾種物品，因此我們重新定義轉移式：
  - $dp[i] =$  背包容量為  $i$  時，所可以得到的最大價值

# 無限背包問題

- 對於容量  $i$  來說有  $m$  種可能
  - 在最大容量  $i - w_1$  時再拿一個第 1 種物品
  - 在最大容量  $i - w_2$  時再拿一個第 2 種物品
  - 在最大容量  $i - w_3$  時再拿一個第 3 種物品
  - and so on...
- 你有發現  $\square$ ？是不是跟前面骰子還有硬幣那一題一樣了！
- 只差在最後要求的東西不一樣而已



## 無限背包問題

- 對於容量  $i$  來說拿第  $k$  個物品的話：
  - $dp[i] = \max( dp[i] , dp[i-w_k] + v_k )$
- 整體時間複雜度： $O(nm)$

```
29     for(int i=1;i<=m;i++)
30     {
31         for(int k=1;k<=n;k++)
32         {
33             dp[i] = max(dp[i],dp[i-w[i]]+v[i]);
34         }
35     }
36
37     cout << dp[m] << "\n";
38 }
```

## 背包問題的瓶頸

- 時間複雜度我們可能無法改變，目前找不到什麼好方法
- 那我們來看看空間複雜度

## 背包問題的空間複雜度

- 需要開  $n * m$  的 dp 表格去紀錄 (除了無限背包有  $O(m)$  的作法), 因此空間複雜度為  $O(nm)$
- DP 是一個用 空間 換取 時間 的技巧
- 也就是說  $n * m$  如果太大, 我們是完成不了 0/1 背包問題的
- 但其實 0/1 背包問題也有空間複雜度  $O(m)$  的作法
- 所以接下來我們來講 DP 當中的第一個優化技巧 滾動陣列

## DP 優化：滾動陣列

- 如果一般的動態規劃是：
  - 長江後浪推前浪，一替新人換舊人
- 那麼被 滾動陣列 優化後的動態規劃就是：
  - 長江後浪推前浪，前浪死在沙灘上

## DP 優化：滾動陣列

- 不是把陣列拿起來滾
- 滾動陣列的核心精神是：
  - 把沒有用到的陣列拿來繼續重複使用
- 其實就是有點資源回收的概念

## DP 優化：滾動陣列

- 我們來看看 0/1 背包問題：
  - $dp[i][k] = \max( dp[i-1][k] , dp[i-1][k-w_i] + v_i )$
- 有發現什麼事情□？
- 如果我們現在正在算  $dp[5][0\sim m]$ 
  - 那麼  $dp[1][0\sim m]$ ,  $dp[2][0\sim m]$ ,  $dp[3][0\sim m]$  以後都用不到了
  - 因為每一次轉移都只要知道前一次的結果
  - 這個時候滾動陣列這個技巧就可以派上用場了

## DP 優化：滾動陣列

- 對於背包問題來說只需要記錄前 1 次的結果
- 我們就可以開兩組長度為  $m$  的陣列就好
  - 其中一組紀錄上一次的結果
  - 其中一組紀錄這一次轉移的結果
- 這樣的話空間複雜度就從  $O(nm)$  被我們優化成  $O(m)$

## DP 優化：滾動陣列

- 我自己習慣把陣列開成  $dp[2][m]$
- 然後假設  $i$  是奇數時就表示  $i - 1$  是偶數，因此可以寫成
  - $dp[i \bmod 2][k] = \max( dp[(i - 1) \bmod 2][k] , dp[(i - 1) \bmod 2][k - w_i] + v_i )$

```
38
39     for(int i=1;i<=n;i++)
40     {
41         for(int k=0;k<=m;k++)
42         {
43             if( k - w[i] >= 0 ) dp[i][k] = max( dp[(i-1)%2][k] , dp[(i-1)%2][ k - w[i] ] + v[i] );
44             else dp[i][k] = dp[(i-1)%2][k];
45         }
46     }
47
48     cout << dp[n%2][m] << "\n";
49
```



## 0/1 背包甚至可以不使用二維陣列

- 因為每一次都是拿前 1 次的結果，而且結果是可以一直使用的不用清空，所以其實 0/1 背包問題可以只開一維陣列解決
- 但是有個超級大的陷阱

## 0/1 背包甚至可以不使用二維陣列

- 如果我們寫成這樣會發生什麼事情？
- 反正之前的 dp 記錄留著就好，那我就用一維陣列

```
38  
39     for(int i=1;i<=n;i++)  
40     {  
41         for(int k=0;k<=m;k++)  
42         {  
43             if( k - w[i] >= 0 ) dp[k] = max(dp[k],dp[k-w[i]]+v[i]);  
44         }  
45     }  
46  
47     cout << dp[m] << "\n";
```

## 0/1 背包甚至可以不使用二維陣列

- 如果我們寫成這樣會發生什麼事情？
- 假設  $dp[10]$  我們拿了第 1 個物品，這個物品的容量是 10
- 那如果我在轉移  $dp[20]$  的時候： $dp[20] = dp[20-10] + v_1$
- 我們重複拿了第 1 個物品 2 次！這是不合法的！

```
38
39     for(int i=1;i<=n;i++)
40     {
41         for(int k=0;k<=m;k++)
42         {
43             if( k - w[i] >= 0 ) dp[k] = max(dp[k],dp[k-w[i]]+v[i]);
44         }
45     }
46
47     cout << dp[m] << "\n";
```

## 0/1 背包甚至可以不使用二維陣列

- 但是如果我們如果把第二個迴圈倒著回來？
- 我們每一次拿的資訊都是前一次的資訊，不會重複拿
- 這樣就完成一維陣列版本的 0/1 背包了

```
39     for(int i=1;i<=n;i++)
40     {
41         for(int k=m;k>=w[i];k++)
42         {
43             dp[k] = max(dp[k],dp[k-w[i]]+v[i]);
44         }
45     }
46
47     cout << dp[m] << "\n";
```

## 那這個是什麼背包？

- 我們剛剛說這樣子因為會重複拿同一個物品，不符合 0/1 背包
- 那什麼樣的背包問題可以重複拿同一個物品？

```
38  
39     for(int i=1;i<=n;i++)  
40     {  
41         for(int k=0;k<=m;k++)  
42         {  
43             if( k - w[i] >= 0 ) dp[k] = max(dp[k],dp[k-w[i]]+v[i]);  
44         }  
45     }  
46  
47     cout << dp[m] << "\n";
```

## 那這個是什麼背包？

- 我們剛剛說這樣子因為會重複拿同一個物品，不符合 0/1 背包
- 那什麼樣的背包問題可以重複拿同一個物品？
- 無限背包！因此無限背包也可以這樣寫

```
38  
39     for(int i=1;i<=n;i++)  
40     {  
41         for(int k=0;k<=m;k++)  
42         {  
43             if( k - w[i] >= 0 ) dp[k] = max(dp[k],dp[k-w[i]]+v[i]);  
44         }  
45     }  
46  
47     cout << dp[m] << "\n";
```

## 背包問題的變化？

- 背包問題的變形其實也很多
- 通常比賽遇到背包問題的題目都會被包裝的很精美

## 背包問題 - 改：平分問題

- 這其實也很常見了
- 有  $n$  個物品，每個物品的精美程度為  $a_i$
- 現在必須把這  $n$  個物品分給某兩個人
- 你的目標是要讓最後這兩個人拿到的物品總精美程度的差越小越好
  - 講簡單一點：就是要盡可能讓兩個人的精美程度一樣
- 想想看這該怎麼做ㄅ



## 背包問題 - 改：平分問題

- 在最完美的情況下，兩個人的總精美程度都會是  $(a_1 + \cdots + a_n) / 2$
- 把題目轉成，給定  $n$  個物品，你的背包容量是  $(a_1 + \cdots + a_n) / 2$
- 每個物品的精美程度是  $a_i$ ，佔據的空間也是  $a_i$
- 求出這個背包最多能裝多少精美程度的物品
- 從這邊就可以發現，題目就變回一個很單純的 0/1 背包問題了
- 最後答案就會是  $(a_1 + \cdots + a_n) / 2 - dp[(a_1 + \cdots + a_n) / 2]$

## LIS: CSES Problem Set Increasing Subsequence

- 給定一個長度為  $n$  的序列  $a$
- 找出  $a$  序列的最長遞增子序列

## LIS: CSES Problem Set Increasing Subsequence

- 我們先想想  $O(n^2)$  的作法
- 定義  $dp[i]$  表示以  $i$  這個位置為結尾的 LIS 長度
- 我們從左到右求出  $dp[i]$

## LIS: CSES Problem Set Increasing Subsequence

- 假設我們在求  $dp[5]$
- 那麼我們就去看看前面的  $a[1]$  ,  $a[2]$  ,  $a[3]$  ,  $a[4]$ 
  - 如果  $a[1] < a[5]$  , 那就  $dp[5] = \max( dp[5] , dp[1] + 1 )$
  - 如果  $a[2] < a[5]$  , 那就  $dp[5] = \max( dp[5] , dp[2] + 1 )$
  - 如果  $a[3] < a[5]$  , 那就  $dp[5] = \max( dp[5] , dp[3] + 1 )$
  - 如果  $a[4] < a[5]$  , 那就  $dp[5] = \max( dp[5] , dp[4] + 1 )$
- 因為前面的 LIS 我們都已經求出來了, 就想像成多接一個  $a[5]$  進去就可以了

## LIS: CSES Problem Set Increasing Subsequence

- 最後全部求完後記得答案不是  $dp[n]$ ，不要衝動
- 我們轉移式： $dp[i]$  = 以  $i$  這個位置為結尾的 LIS 長度
- 因此答案為： $\max(dp[1], \dots, dp[n])$
- 時間複雜度： $O(n^2)$

```
26
27     for(int i=0;i<n;i++) cin >> a[i], dp[i] = 1;
28     for(int i=0;i<n;i++)
29     {
30         for(int k=i-1;k>=0;k--)
31         {
32             if( a[k] < a[i] ) dp[i] = max(dp[i],dp[k]+1);
33         }
34     }
35
36     int ans = 1;
37     for(int i=0;i<n;i++) ans = max(ans,dp[i]);
38
39     cout << ans << "\n";
```

## LIS: CSES Problem Set Increasing Subsequence

- 但這題的  $n$  範圍太大,  $O(n^2)$  會 TLE, 怎麼辦?
- 這個時候我們需要一個額外的演算法來輔助我們:
  - 二分搜尋 Binary Search
  - 貪婪演算法 Greedy Algorithm

## 補一些先備知識 - 二分搜尋 Binary Search

- 在搜尋演算法中最具有代表性的一個演算法
- 現在給你猜  $1 \sim 10^9$  其中一個數字
- 猜錯我會告訴你猜的太大還是太小
- 請你盡可能的使用最少次數猜到正確的數字

## 補一些先備知識 - 二分搜尋 Binary Search

- 每次都猜中間的那一個數字就可以讓範圍少掉一半
- 因此我們持續使用同一個策略猜到數字
- 就可以保證我們可以在  $\log(10^9) + 1$  次內找到答案
- 這個就是二分搜的精神
- 來實際挑戰看看？



## 台南女中資訊研究社 38th 教學選幹上機考 第五題：社團評鑑

- 這是我某天走在路上想到的遊戲，然後我就把他出成題目了
- 這是一題互動題，我當初把這題設定好就花了我快 3 個小時：
  - 我都被他們 37th 的社長壓榨
- 現在有一個長度為  $n$  的序列，你一開始不知道這個序列長怎樣
- 你接下來可以做 18 次以內的查詢
- 每一次查詢你可以詢問  $[L, R]$  當中
  - $mid = (L + R) / 2$
  - $MAX[L, mid] - MAX[mid + 1, R]$  的數字是多少
- 請你在 18 次的查詢內，找出這一個序列最大值的位置在哪？

## 台南女中資訊研究社 38th 教學選幹上機考 第五題：社團評鑑

- N 的長度不超過  $2 * 10^5$ 
  - Hint :  $\log(2*10^5) + 1 = 18$
- 且保證每一個元素不會重複出現
- 大家想想看怎麼破解這一個遊戲ㄤ

## 台南女中資訊研究社 38th 教學選幹上機考 第五題：社團評鑑

- 我們每一次都查詢  $[L, R]$  就可以得到：
  - $MAX[L, (L + R) / 2] - MAX((L + R) / 2 + 1, R)$
- 假設上面這一個式子的結果是  $K$ ，則有兩種情況
  - $K > 0$ ：最大值的位置一定在左半部
  - $K < 0$ ：最大值的位置一定在右半部
- 因此每次查詢我們都可以縮小一半的區間
- 我們就可以保證在 18 次以內找到答案

## 台南女中資訊研究社 38th 教學選幹上機考 第五題：社團評鑑

- 大家應該都稍微懂二分搜的概念了，那我們回去原本那一題ㄟ

## LIS: CSES Problem Set Increasing Subsequence

- 我們額外開一個陣列  $b$
- 如果有一個長度為  $i$  的 LIS，那麼把  $b[i]$  放在此 LIS 的最後一個位置會是最小的數字
- 一開始  $b$  序列為  $\{ \}$

## LIS: CSES Problem Set Increasing Subsequence

- 我們從  $a[0] \sim a[n-1]$ , 每次看看  $a[i]$  可以被插入在  $b$  的什麼位置
  - 很像插入排序
- 假設  $b = \{ 10, 20, 30 \}$ , 這個時候  $a[i] = 13$
- 我們就要將  $b$  改為  $\{ 10, 13, 30 \}$ 
  - 因為在長度為 2 時, LIS 的最後一個數字選 13 是最小的
    - $\{ 10, 20 \}$  and  $\{ 10, 13 \}$
  - 只要讓選的數字越小越好, 我們後面要構造長度更長的 LIS 就會更容易構造

## LIS: CSES Problem Set Increasing Subsequence

- 那如果  $b = \{ 10, 13, 30 \}$ , 這個時候  $a[i] = 40$
- 就表示有一個當前最大的數字可以接在最後面
- 因此  $b = \{ 10, 13, 30, 40 \}$

## LIS: CSES Problem Set Increasing Subsequence

- 怎麼求出最後答案的長度？
- 我們回想看看我們  $b$  陣列的定義：
  - 如果有一個長度為  $i$  的 LIS，那麼把  $b[i]$  放在此 LIS 的最後一個位置會是最小的數字
- 那如果我們  $b$  陣列的長度是  $k$ 
  - 有一個長度  $k$  的 LIS，把  $b[k-1]$  放在這一個 LIS 會是最佳選擇
  - 換句話說存在這樣的長度為  $k$  的 LIS
  - 因此我們最後整個  $a$  序列的 LIS 為  $k$



## LIS: CSES Problem Set Increasing Subsequence

- 在尋找插入點的這一個過程我們可以使用 二分搜 來優化
- 整體的時間複雜度就會是  $O(N\log N)$
- 這一個求 LIS 的演算法稱為：Robinson-Schensted-Knuth Algorithm

```
5
6 int solve(vector<int> a)
7 {
8     vector<int> dp;
9
10    dp.push_back(-1e9);
11
12    for(int i=0;i<a.size();i++)
13    {
14        auto it = lower_bound(dp.begin(),dp.end(),a[i]);
15
16        if( it == dp.end() ) dp.push_back(a[i]);
17
18        else dp[it-dp.begin()] = a[i];
19    }
20
21    return dp.size() - 1;
22 }
23 signed main(void)
```

## 還有什麼經典問題？

- LIS 最長遞增子序列
- LCS 最長共同子序列
- 有限背包問題
- 賽局 dp
- and so on...
- 動態規劃在競程裡面有很噁心的題目：(
- 有機會再跟大家好好介紹！

## 下一次的主題？

- 繼續深入動態規劃？
  - 單調隊列優化
  - 位元 DP
  - 倍增法
- 矩陣快速冪？
- 二分搜尋？（浮點數二分搜、平均二分搜）
- 圖論？（MST、DSU、Shortest Path）