Dynamic Programming

Lecture by Colten (Jyun-An Chen)

Credit by yp155136, howard41436, boook, anj226,Colten (2023 updated)

Dynamic Programming(DP)

1. 回顧一下課前影片
2. 什麼是 DP？
3. DP 的一些細節
4. 練習

什麼是 DP？

- 大家分享一下看完影片教學後，覺得什麼是 DP 吧！

什麼是 DP？

簡單來說，就是把大問題分成小問題：

- 用小問題**推出**大問題的答案

所以建 DP 時我們要想兩件事：

1. 這個大問題**如何改成**比較簡單的小問題？（怎麼訂狀態）
2. 大問題的答案**如何**從小問題的答案**推得**？（狀態間怎麼轉移）

如果想不到 DP 的作法時，可以考慮**最後一次/最後一格**會發生什麼事，通常作法都會和這個有關。

什麼是 DP？

DP 的幾個特性：

1. 重複子問題(Overlapping subproblems)
   - 一格會用到很多次，因為 DP 是一個用空間換取時間的演算法！
2. 最佳子結構(Optimal Substructure)
   - 講白話就是最佳解一定在所有考慮的子問題範圍內
   - 最佳化問題中，做了某個決定以後，變成一個比較小的問題
   - 計數問題中，我們考慮**所有**可能的最終決定，變成很多個小問題

什麼是 DP？

可是...這和前幾周學到的演算法有什麼不一樣？

1. 和分治的差異

- 通常分治的子問題只會出現一次

2. 和 Greedy 的差異

- Greedy 同樣也具有最佳子結構，但是 Greedy 可以透過**推理**排除絕對不可能的分支！

什麼是 DP？

哪些題目可能是 DP？

DP 通常拿來解決兩種問題:

1. 最優解問題(通常是最大最小化問題)
2. 計數問題

//大家想想，這兩種問題是不是很適合從小答案算出大答案呢？

DP 的一些細節

1. 怎麼估計 DP 的時間複雜度？

- 狀態複雜度：
  - 簡單來說就是陣列開了多少格
  - 每格都是一個狀態，都需要算出答案
- 轉移複雜度：
  - 轉移複雜度則是算出某一格的答案需要的時間複雜度
  - 可以觀察轉移式來得知
- DP 的總複雜度，就是**總共有幾格**乘上**一格需要計算的時間**！

舉個栗子

費氏數列(假設要算 n 個):

- 定義**狀態**:DP[i] 代表數列裡第 i 個數字。
- 狀態**轉移**:DP[i] = DP[i-1] + DP[i-2]。
- **狀態**複雜度:O(n)
- **轉移**複雜度:O(1)
- 總共:O(n)*O(1) = O(n)

舉個栗子

LCS (假設兩個字串的長度都是 n):

- 定義**狀態**:DP[i][j] 代表 s1[1 : i] 跟 s2[1 : j] 的解
- 狀態**轉移**:DP[i] = max(DP[i-1][j], DP[i][j-1]) + 1

  DP[i-1][j-1] + 1 (s1[i]==s2[j])
- **狀態**複雜度:??? (大家可以想想看！)
- **轉移**複雜度:??? (大家可以想想看！)

DP 的一些細節

因為 DP 總是分成這兩個部分，而壓複雜度有可能是從狀態下手，也有可能是從轉移下手，因此這兩件事是要分開討論的。

也就是「我的 DP 是n^3」這句話本身不夠表示你的 DP 演算法，必須要說「我的 DP 狀態 n^2，轉移 n」才夠精確。

我們通常用 nD/mD 來表示一個狀態 $O(N^n)$，轉移 $O(N^m)$ 的 DP 演算法。

在做每題 DP 時，都一定要好好寫出你的時間複雜度！

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

- 這樣子從底層往上慢慢推出後面資訊的方式稱為 Buttom-Up
- 與 Buttom-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)

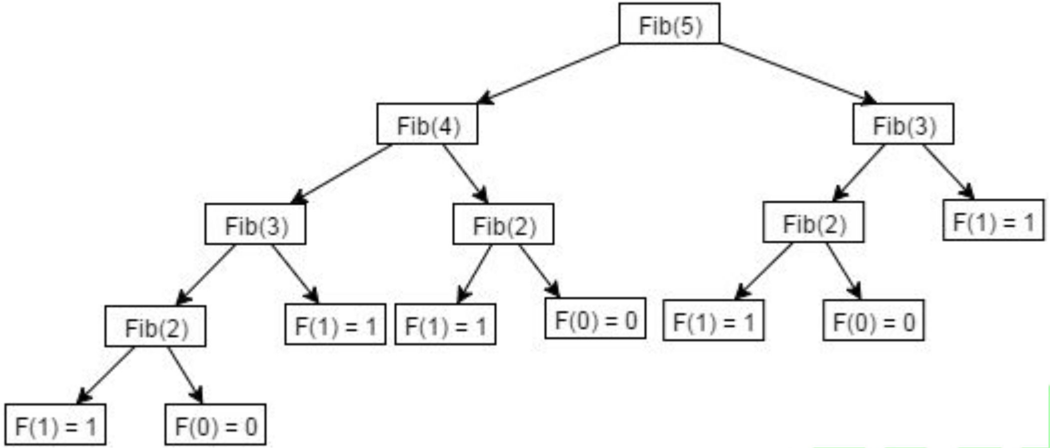
- 我們嘗試使用遞迴求得 f(n) 是多少
- 求得 f(n) 需要 f(n-1) 與 f(n-2) 的資訊
- 遞迴的終止條件則為 f(0) 與 f(1)

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

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

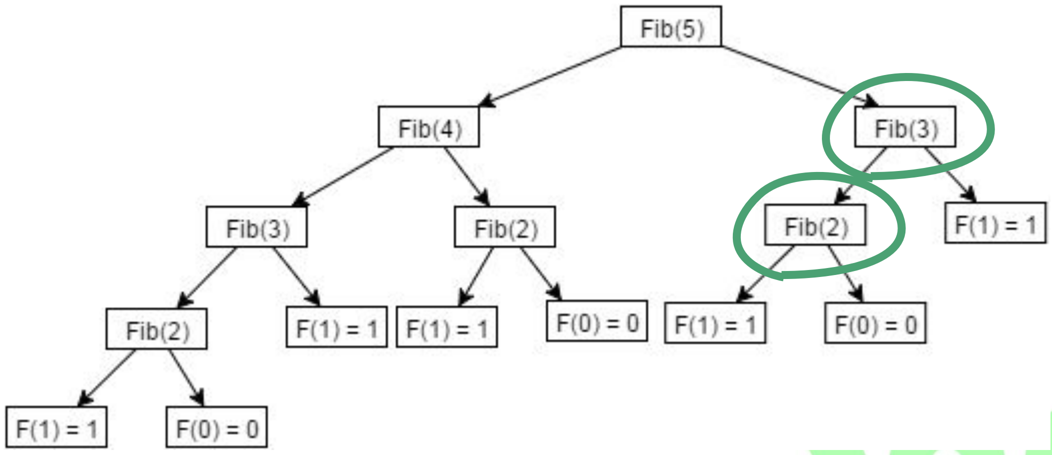
f(n) = f(n-1) + f(n-2)

- 這樣子的時間複雜度是 O(n) ㄇ？
- 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



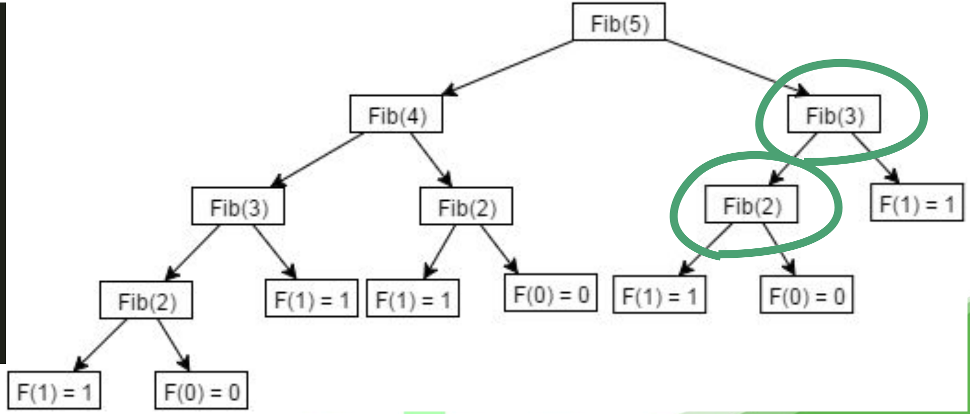
f(n) = f(n-1) + f(n-2)

- 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

```cpp
int dp[101];

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

    if( dp[n] != 0 ) return dp[n]; // 算過了，直接拿以前算出來的東西
    else
    {
        dp[n] = f(n-1) + f(n-2);
        return dp[n];
    }
}
```



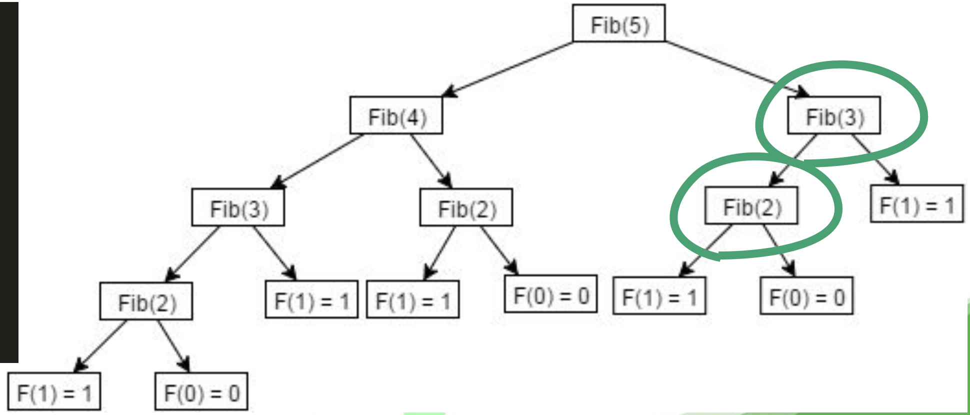
時間複雜度？

- 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

```cpp
int dp[101];

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

    if( dp[n] != 0 ) return dp[n]; // 算過了，直接拿以前算出來的東西
    else
    {
        dp[n] = f(n-1) + f(n-2);
        return dp[n];
    }
}
```



DP 的一些細節

3. DP “bottom up” 的種類
以費氏數列舉例:

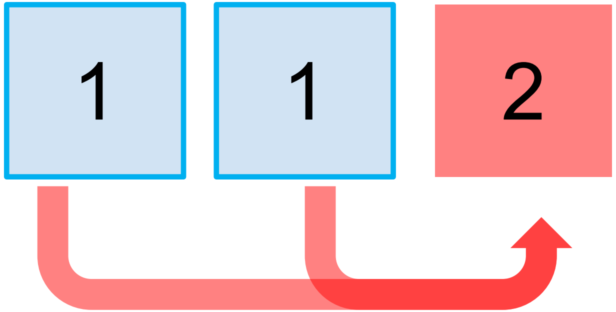
- 用「拉」的:
  - for (int i = 2; i <= n; ++ i)
    - dp[i] = dp[i - 1] + dp[i - 2];

- 用「推」的:
  - for (int i = 0; i <= n; ++ i) {
        if (i + 1 <= n) dp[i + 1] += dp[i];
        if (i + 2 <= n) dp[i + 2] += dp[i];
    }

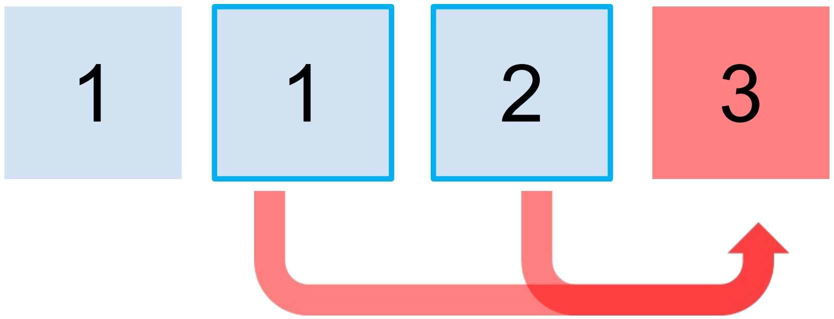
- 用「拉」的：

1	1

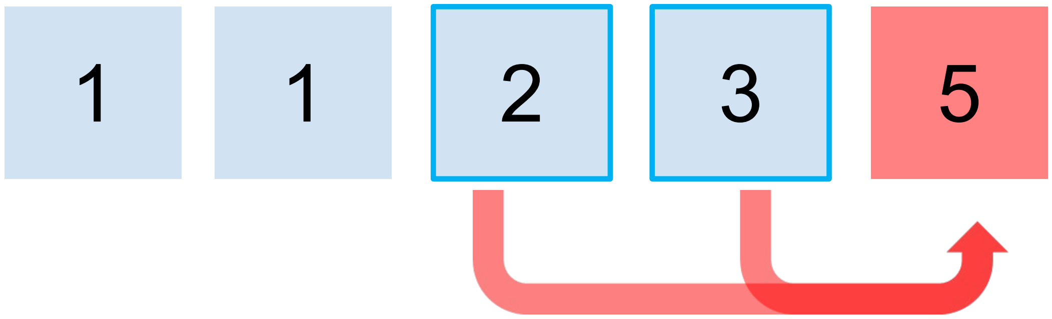
- 用「拉」的：



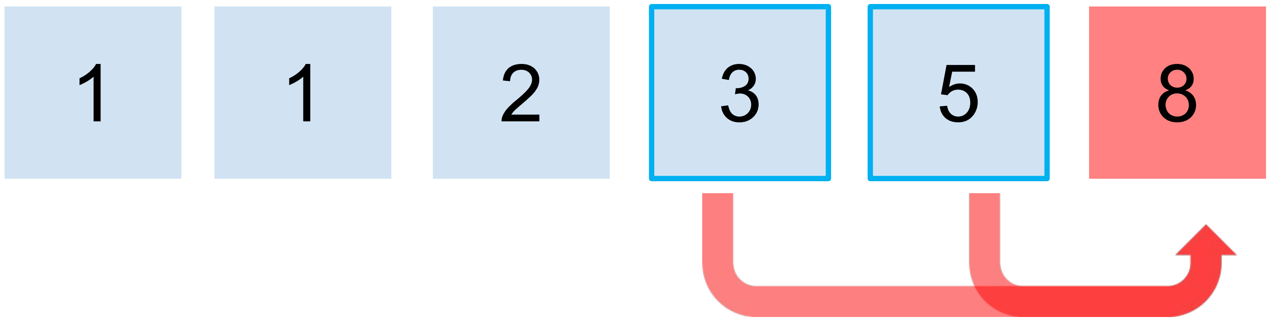
- 用「拉」的：



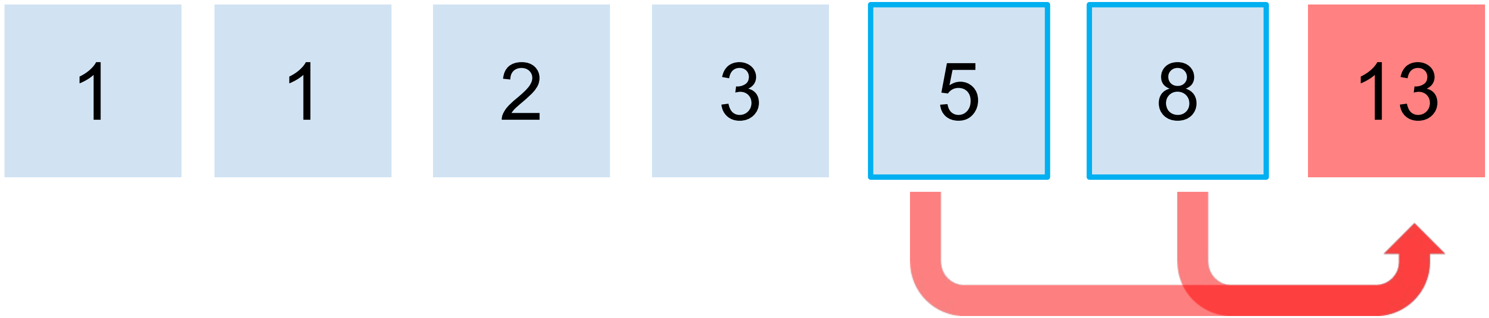
- 用「拉」的：



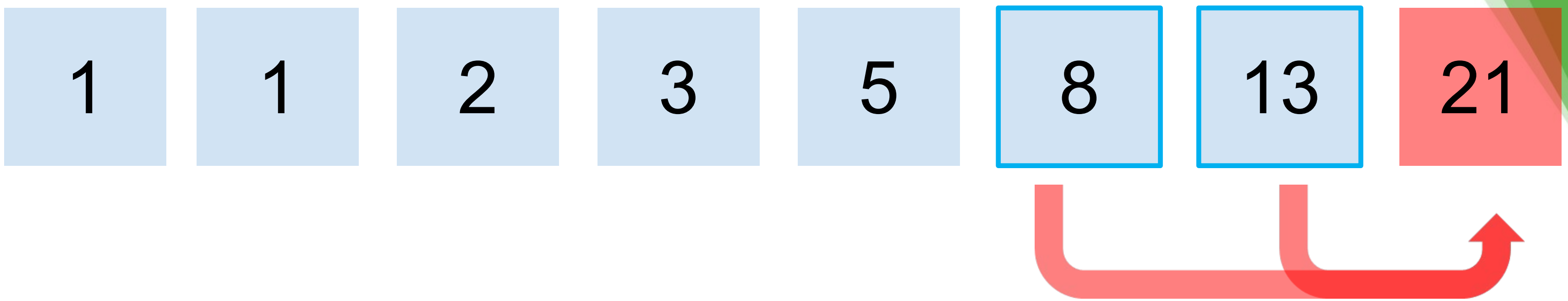
- 用「拉」的:



- 用「拉」的：



- 用「拉」的:



- 用「拉」的:

1	1	2	3	5	8	13	21

- 用「推」的:

1	0	0	0	0	0	0	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	1	0	0	0	0	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	1	0	0	0	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	3	2	0	0	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	3	5	3	0	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	3	5	8	5	0

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	3	5	8	13	8

- 用「拉」的：

1	1	2	3	5	8	13	21

- 用「推」的：

1	1	2	3	5	8	13	21

- 用「拉」的:

1	1	2	3	5	8	13	21

- 用「推」的:

1	1	2	3	5	8	13	21

跑步問題

[Zerojudge b589]

- 有 n 段路，每段路有一個分數 a_i，你每段路可以用其中一種速度
  1.用走的：你不會得到任何分數
  2.用跑的：你會得到 a_i 的分數
  3.用衝的：你會得到 $2a_i$ 的分數，但你下一段路得用走的
- 請問你最多能得到多少分？

跑步問題

Zerojudge b589

- 有 n 段路，每段路有一個分數 a_i，你每段路可以用其中一種速度

  1.用走的：你不會得到任何分數

  2.用跑的：你會得到 a_i 的分數

  3.用衝的：你會得到 $2a_i$ 的分數，但你下一段路得用走的

  dp[i][0]：沒限制

  dp[i][1]：下一段路得用走的

跑步問題

Zerojudge b589

1.用走的:你不會得到任何分數

2.用跑的:你會得到 a_i 的分數

dp[i][0]
= max(dp[i - 1][0], dp[i – 1][0]+ a_i, dp[i
– 1][1])

3.用衝的:你會得到 $2a_i$ 的分數, 但你下一段路得用走的

dp[i][1] = dp[i – 1][0] + $2a_i$

pastebin.com/raw/DTY0cwzh

最大連續和問題

[Zerojudge d784]

- 有 n 個數字，每個數字可正可負。
- 請你選連續的一段數字，問最大總和可以是多少？

最大連續和問題

[Zerojudge d784]

- dp[i] : 以 i 為結尾的最大總和是多少
- 考慮要不要跟前面的接起來:
  `dp[i] = max(dp[i - 1] + a[i], a[i]);`
- 最後的答案為 dp 陣列的最大值
- 當然, 這題也有 greedy / 分治的作法!

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 , 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 定義轉移式
- 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] + … + 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 mod 10^9 + 7 的結果就好

```
const int mod = 1e9 + 7;

signed main(void)
{
    int n;
    cin >> n;

    dp[0] = 1;

    for(int i=1;i<=n;i++)
    {
        for(int k=1;k<=6;k++)
        {
            if( i - k >= 0 ) dp[i] += dp[i-k] , dp[i] %= mod;
        }
    }

    cout << dp[n] << "\n";

    return 0;
}
```

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 ci
- 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 - c1 再拿 1 個 c1 硬幣得來的
  - i - c2 再拿 1 個 c2 硬幣得來的
  - i - c3 再拿 1 個 c3 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 ~ 6，硬幣是 c1 ~ cn

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 c1 ~ cn
- 因此整體時間複雜度為 O(nx)
- 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

```
    int n,m;
    cin >> n >> m;
    vector <int> a(n);

    for(int i=0;i<n;i++)
    {
        cin >> a[i];
    }

    dp[0] = 1;

    for(int i=1;i<=m;i++)
    {
        for(int k=0;k<n;k++)
        {
            if( i - a[k] >= 0 )
            {
                dp[i] += dp[i-a[k]];
                dp[i] %= mod;
            }
        }
    }

    cout << dp[m] << "\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] = $a_{1,1}$, dp[1][2] = $a_{1,2}$, dp[1][3] = $a_{1,3}$

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]) + ai
- 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]) + bi

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]) + ci

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)

```
int n;
cin >> n;

for(int i=1;i<=n;i++)
{
    for(int k=1;k<=3;k++)
    {
        cin >> a[i][k];
    }
}

dp[1][1] = a[1][1] , dp[1][2] = a[1][2] , dp[1][3] = a[1][3];

for(int i=2;i<=n;i++)
{
    dp[i][1] = max(dp[i-1][2],dp[i-1][3]) + a[i][1];
    dp[i][2] = max(dp[i-1][1],dp[i-1][3]) + a[i][2];
    dp[i][3] = max(dp[i-1][1],dp[i-1][2]) + a[i][3];
}

cout << max({dp[n][1],dp[n][2],dp[n][3]}) << "\n";
```

最大和矩陣問題

經典題：給你一個 $n \times m$ 的矩陣，所有的子矩陣中，最大的數字和是多少？（$1 \le n, m \le 500$）

- 最裸最裸的做是 $O(n^3m^3)$
- 合理的裸做是 $O(n^2m^2)$

還能不能做得更好？

Hint: 跟區間最大連續和的作法有關係。

What is 矩陣？

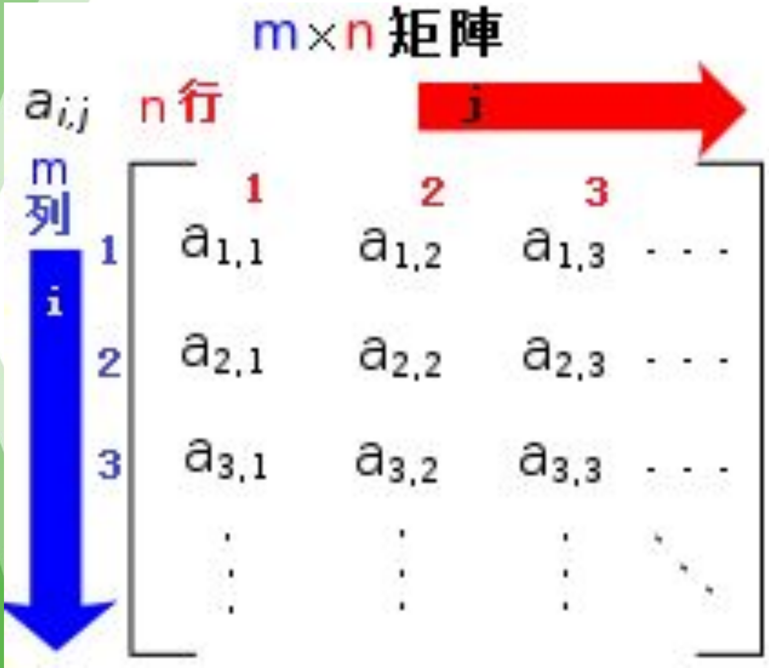
$m \times n$ 矩陣

$a_{i,j}$ n行 $\xrightarrow{\quad j \quad}$

m列 $\downarrow i$

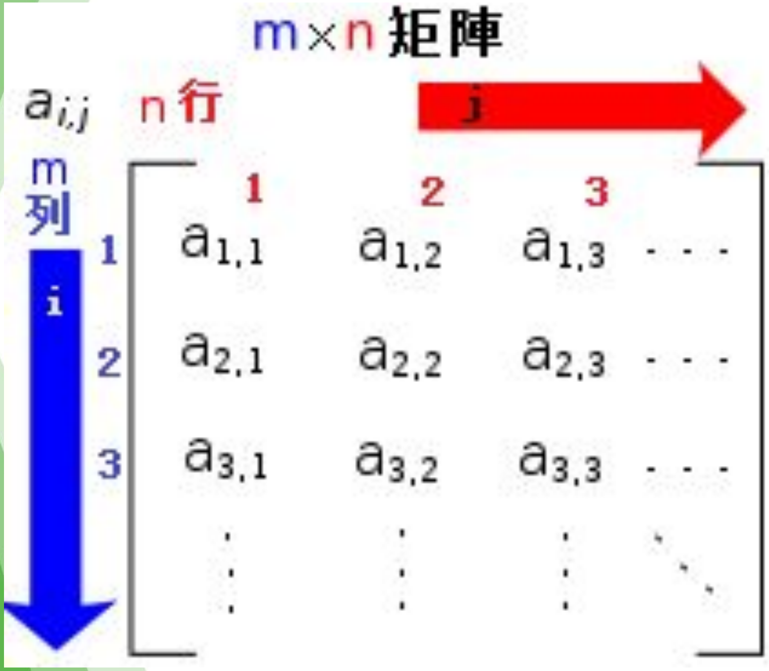
$$\begin{array}{cc}
 & \begin{matrix} 1 & \quad 2 & \quad 3 & \quad \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \\ \\ \end{matrix} &
\begin{bmatrix}
a_{1,1} & a_{1,2} & a_{1,3} & \cdots \\
a_{2,1} & a_{2,2} & a_{2,3} & \cdots \\
a_{3,1} & a_{3,2} & a_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}
\end{array}$$

What is 矩陣？



1	-1	3	4	5
7	4	-5	7	19
21	13	8	0	0
-10	13	-19	-21	0

What is 矩陣？



1	-1	3	4	5
7	4	-5	7	19
21	13	8	0	0
-10	13	-19	-21	0

把他壓扁！！

1	-1	3	4	5
7	4	-5	7	19
21	13	8	0	0
-10	13	-19	-21	0

假設我已經知道要取兩列了。

把他們兩列壓成一列！

把他壓扁！！

1	-1	3	4	5
7	4	-5	7	19
21	13	8	0	[illegible]
-10	13	-19	-21	0

8	3	-2	11	24
28	17	3	7	19
11	26	-11	-21	0

假設我已經知道要取兩列了。
把他們兩列壓成一列！

把他壓扁！！

8	3	-2	11	24
28	17	3	7	19
11	26	-11	-21	0

每一列做最大連續和。

這樣就等於做完所有 2×? 的子矩陣了！

對於高度 m、寬度 n 的矩陣：

- 枚舉高度，O(m)
- 對於壓完的每一行，O(m)
- 做一次最大連續和，O(n)
- $O(m^2n)$（嗎？）

最大和矩陣問題

- 當然也可以把行換成列、列換成行做事，所以實際時間複雜度是 $O(\min(n^2m, m^2n))$。
- 問題在於，怎麼快速的算把很多行壓扁以後的結果？

Sprout

區間和

CSES 1646

- 給你一個陣列，並且有 q 次詢問，每次問某段區間 [L,R] 的數字和。
- 要求每次詢問時間複雜度 O(1)。

區間和

- 用 sum[i] 紀錄第 1 格至第 i 格的數字和，這樣要求區間和時可以用 sum[R]-sum[L-1]
- 這個 sum 叫做**前綴和**陣列

```
Sum[1] = a[1]                          = sum[0] + a[1]
Sum[2] = a[1] + a[2]                   = sum[1] + a[2]
Sum[3] = a[1] + a[2] + a[3]            = sum[2] + a[3]
Sum[4] = a[1] + a[2] + a[3] + a[4] = sum[3] + a[4]

a[3] + a[4] = Sum[4] – Sum[2]
```

區間和

- 這是非常非常常用的技巧，請大家一定要記得，看到跟區間和有關的東西時常常可以這樣轉換：

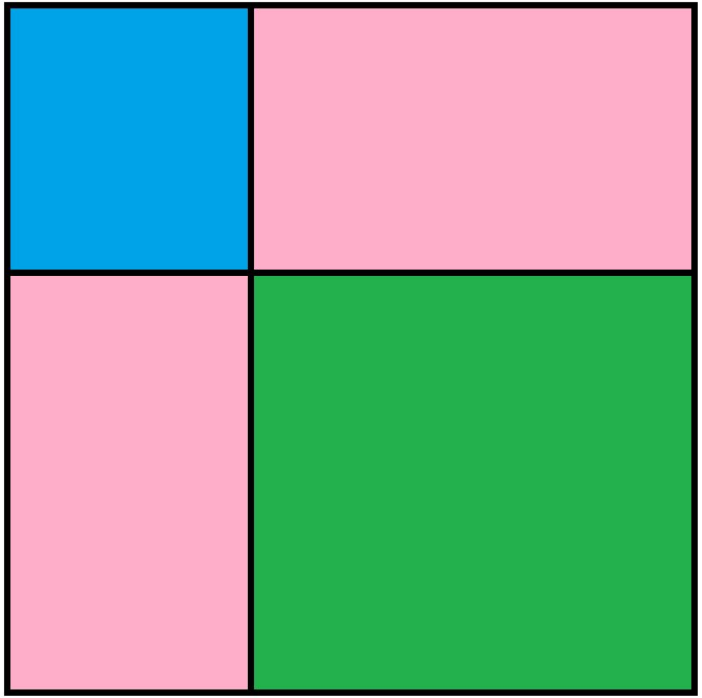
  區間和 ⇔ 兩數的差

- 如果是二維的呢？（每次問你一個矩陣區塊的和）

Sprout

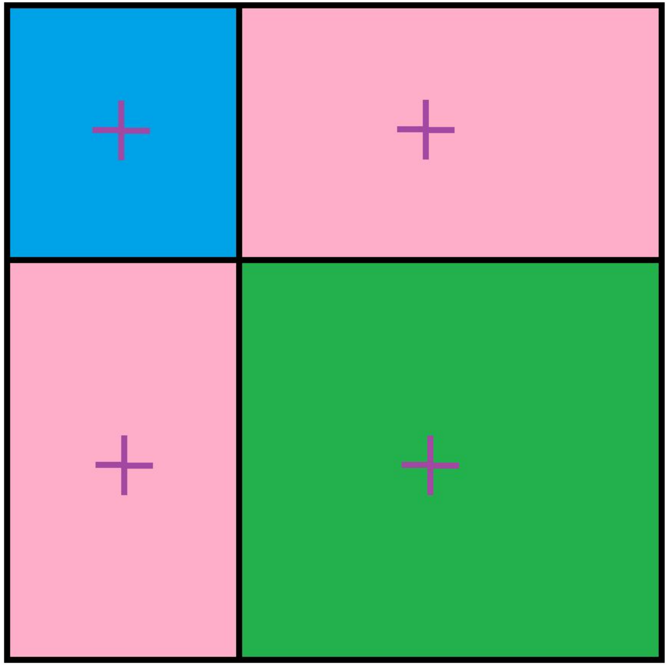
區段和

- 如果是二維的呢？(每次問你一個矩陣區塊的和)



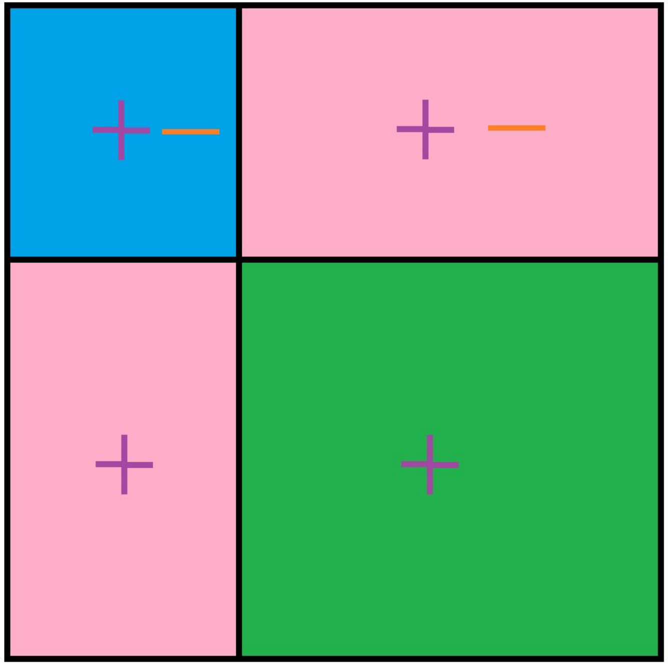
區段和

- 如果是二維的呢？(每次問你一個矩陣區塊的和)



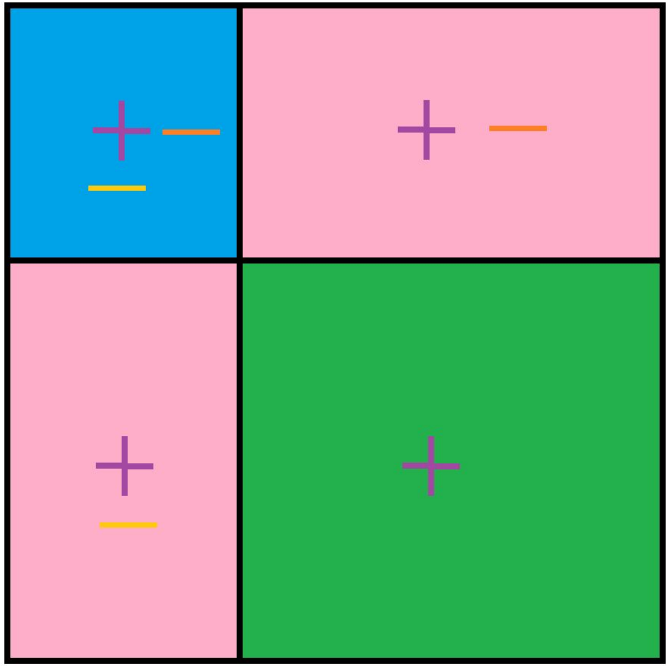
區段和

- 如果是二維的呢？(每次問你一個矩陣區塊的和)



區段和

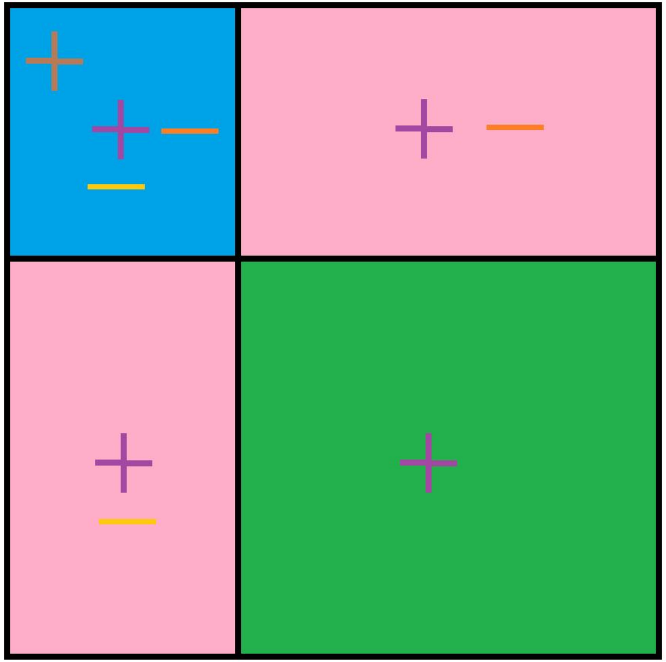
- 如果是二維的呢？(每次問你一個矩陣區塊的和)



區段和

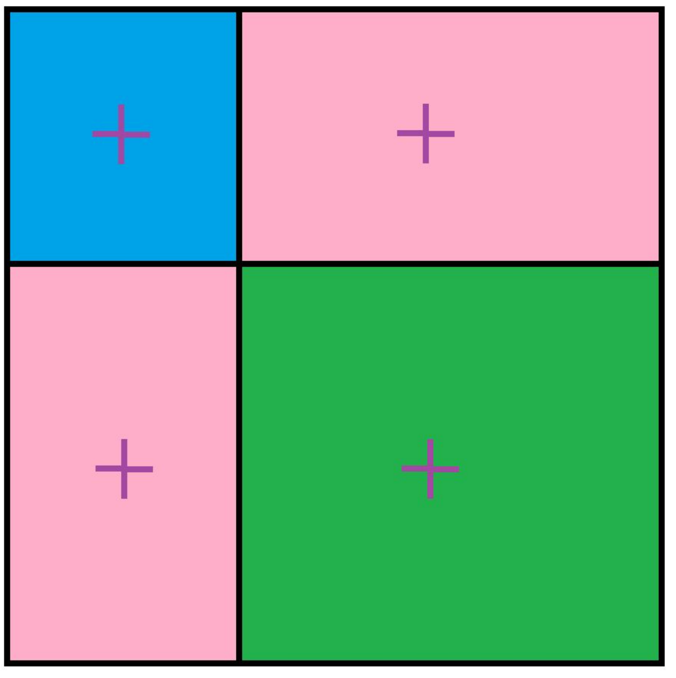
- 如果是二維的呢？(每次問你一個矩陣區塊的和)

```
A[l~L][r~R]
=   S[L][R]
  – S[L][r-1]
  - S[l-1][R]
  + S[l-1][r-1]
```



區段和

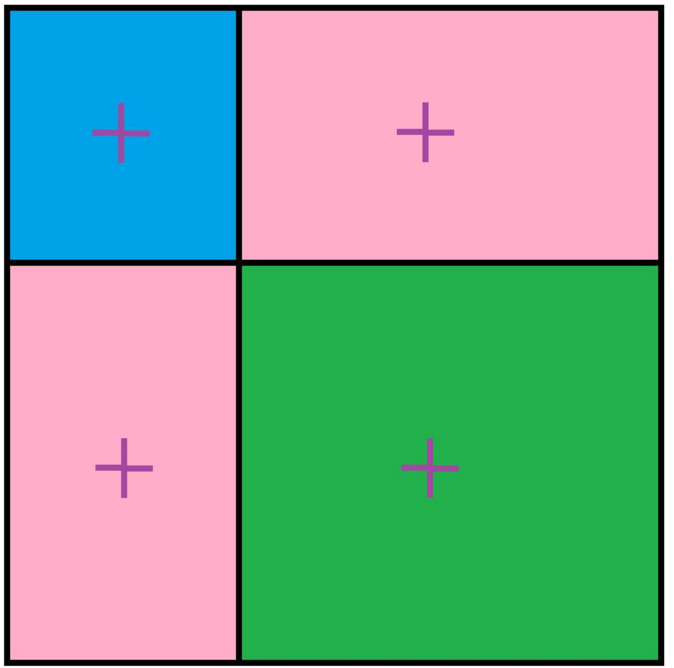
- 如果是二維的呢？(每次問你一個矩陣區塊的和)
- 要怎麼把 s[i][j] 蓋出來呢？



區段和

- 如果是二維的呢？(每次問你一個矩陣區塊的和)
- 要怎麼把 S[i][j] 蓋出來呢？

- S[i][j] = S[i - 1][j] +
  S[i][j - 1] -
  S[i - 1][j - 1] +
  A[i][j]



矩陣最大空方形問題

- 給你一個 01 矩陣，請問裡面最大的全部都是 0 的方形有多大？
- Hint: 令 dp[i][j] 代表以 (i,j) 為右下角時正方形邊長可以多長。

0	0	1	0	0
0	0	0	0	0
0	0	0	0	1
0	0	0	0	1

矩陣最大空方形問題

- 給你一個 01 矩陣，請問裡面最大的全部都是 0 的方形有多大？

```
if (a[i][j] == 1)
    dp[i][j] = 0
if (a[i][j] == 0)
    dp[i][j] = min(dp[i - 1][j - 1]
                 , dp[i][j - 1]
                 , dp[i - 1][j]) + 1
```

- 想想看為什麼(只)需要從 dp[i - 1][j - 1],dp[i][j - 1], dp[i - 1][j] 三個狀態轉移？

矩陣最大空方形問題

- 給你一個 01 矩陣，請問裡面最大的全部都是 0 的方形有多大？

- O(nm) 狀態
- O(1) 轉移

0	0	1	0	0
0	0	0	0	0
0	0	0	0	1
0	0	0	0	1

編輯距離 (edit distance)

- (CSES 1639) 給兩個字串 S, T，你希望把 S 變成 T。

  在 S 插入一個字元要花 ins 塊錢，刪除一個字元要花 del 塊錢，取代一個字元要花 sub 塊錢。請問最小的花費為何？

  |S|, |T| <= 1000

- 想想看**狀態**、**轉移**、**初始值**要怎麼訂？

關於 DP 大家要知道的

- 很多人覺得 DP 很難，但其實 DP 是簡單化問題的方法
- 看到 DP 題目時先建出可以轉移的狀態就好，先不管複雜度
- 找到不管時限會 AC 的 DP 算法，往往已經是成功的一半
  - 接下來的各種優化方式會在未來的 DP 課程教到
- DP 題目多練會進步得很快，因為從題目來建立狀態的方法在很多題目中是相似的，多接觸就會更加的熟悉