

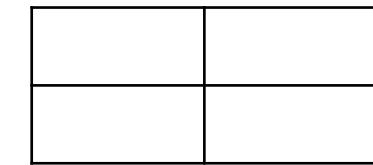
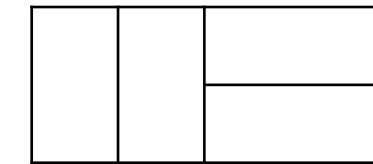
Dynamic Programming

日月卦長



骨牌覆蓋

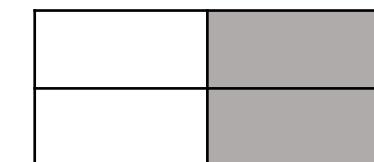
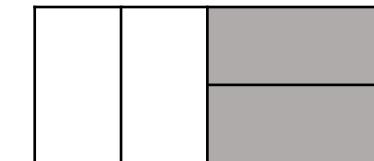
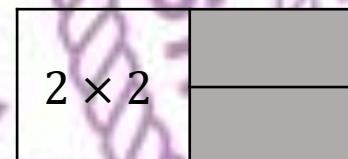
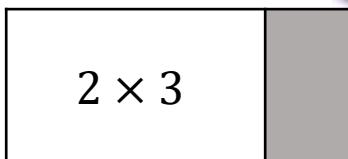
- 紿你 $2 \times n$ 的地板
- 你要用 2×1 的磁磚將其鋪滿
- 請問有幾種方法？



$n = 4$ 的所有方法

骨牌覆蓋

- 觀察這些方法

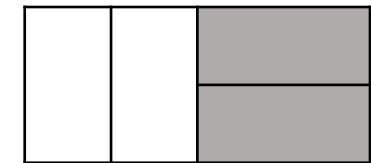


$n = 4$ 的所有方法

骨牌覆蓋

- 得到關係式

$$\bullet f_n = \begin{cases} 1 & , n \leq 1 \\ f_{n-1} + f_{n-2}, & n > 1 \end{cases}$$

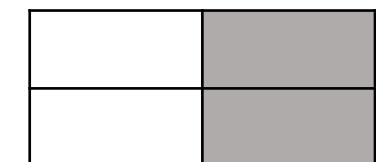
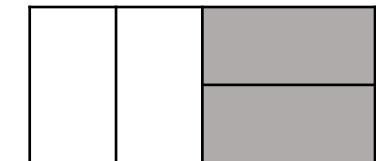


$n = 4$ 的所有方法

骨牌覆蓋

- 得到關係式

$$\bullet f_n = \begin{cases} 1 & , n \leq 1 \\ f_{n-1} + f_{n-2}, & n > 1 \end{cases}$$



$n = 4$ 的所有方法

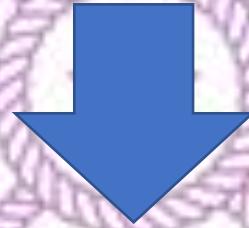
```
long long f(int n) {  
    if (n <= 1) return 1;  
    return f(n - 1) + f(n - 2);  
}
```

很慢 $O(2^n)$



DP的本質：避免重覆計算

```
long long f(int n) {  
    if (n <= 1) return 1;  
    return f(n - 1) + f(n - 2);  
}
```



```
map<int, long long> DP;  
long long f(int n) {  
    if (n <= 1) return 1;  
    if (DP.count(n)) return DP[n];  
    return DP[n] = f(n - 1) + f(n - 2);  
}
```

Dynamic Programming

- 無後效性：
相同的參數會得到相同的答案
- 重複子問題：
過程中相同的參數會重複出現
- 最佳子結構：
原問題的最佳解其子問題的解也是最佳的

Longest Increasing Subsequence

- <https://leetcode.com/problems/longest-increasing-subsequence/>
- 輸出最長遞增子序列長度
- Input: S = [0,3,1,6,2,2,7,-1]
- Output: 4
其中一個最長遞增子序列為：[0,1,2,7]

$\text{dfs}(i)$ 計算 $S[i]$ 為結尾的 LIS

```
vector<int> S;
int dfs(int i) {
    int ans = 0;
    for (int j = 0; j < i; ++j) {
        if (S[i] > S[j])
            ans = max(ans, dfs(j));
    }
    return ans + 1;
}
int solve() {
    int ans = 0;
    for (int i = 0; i < S.size(); ++i)
        ans = max(ans, dfs(i));
    return ans;
}
```

避免重複計算

```
vector<int> S;
vector<int> DP;
int dfs(int i) {
    if(DP[i]) return DP[i];
    int ans = 0;
    for (int j = 0; j < i; ++j) {
        if (S[i] > S[j])
            ans = max(ans, dfs(j));
    }
    return DP[i] = ans + 1;
}
int solve() {
    DP.assign(S.size(), 0);
    int ans = 0;
    for (int i = 0; i < S.size(); ++i)
        ans = max(ans, dfs(i));
    return ans;
}
```

時間複雜度

- 遞迴通常都不太好算時間複雜度
- 但因為我們避免重複計算
 $dfs(0) \sim dfs(n - 1)$ 只會各被計算一次
- 每次計算 dfs 的過程都是 $O(n)$
- 所以總共 $n \times O(n) = O(n^2)$

知道計算的順序就能用迴圈做

```
vector<int> S;

int solve() {
    vector<int> DP(S.size());
    int ans = 0;
    for (int i = 0; i < S.size(); ++i) {
        DP[i] = 0;
        for (int j = 0; j < i; ++j) {
            if (S[i] > S[j])
                DP[i] = max(DP[i], DP[j]);
        }
        DP[i] += 1;
        ans = max(ans, DP[i]);
    }
    return ans;
}
```

狀態、狀態轉移式 state transition equation

- 設 DP_i 表示以 $S[i]$ 為結尾時的 LIS 長度

狀態

$$DP_i = \max_{j < i, S[j] < S[i]} \{DP_j\} + 1$$

狀態轉移式

xD/yD 表示法

- 表示該 DP 的時間複雜度為 $O(n^{x+y})$
- 大多數 DP 時間複雜度 = 狀態數 \times 狀態轉移時間
 - 狀態數有 $O(n^x)$
 - 狀態轉移時間 $O(n^y)$
- EX: 剛剛的 LIS 是 $1D/1D$ 的 DP，複雜度 $O(n^{1+1}) = O(n^2)$

作法分類

- Top Down:

用遞迴，記錄所有狀態的計算結果避免重複計算

- Bottom Up:

知道每個狀態的計算順序，按造順序推出目標狀態
通常用迴圈就能搞定



一般思考流程



Longest Common Subsequence

- <https://leetcode.com/problems/longest-common-subsequence/>
- 輸出兩字串的最長共同子序列長度
- Input: A = "abcde", B = "aceb"
- Output: 3
最長共同子序列为："ace"

Step 1: 定義狀態

- 設 $DP_{x,y}$ 表示 $A[0 \sim x], B[0 \sim y]$ 的 LCS 長度



Step 2: 想狀態轉移式

A[0]	A[1]	A[2]	...	A[x]
------	------	------	-----	------

B[0]	B[1]	B[2]	...	B[y]
------	------	------	-----	------



Step 2: 想狀態轉移式

A[0]	A[1]	A[2]	...	A[x]
------	------	------	-----	------

B[0]	B[1]	B[2]	...	B[y]
------	------	------	-----	------

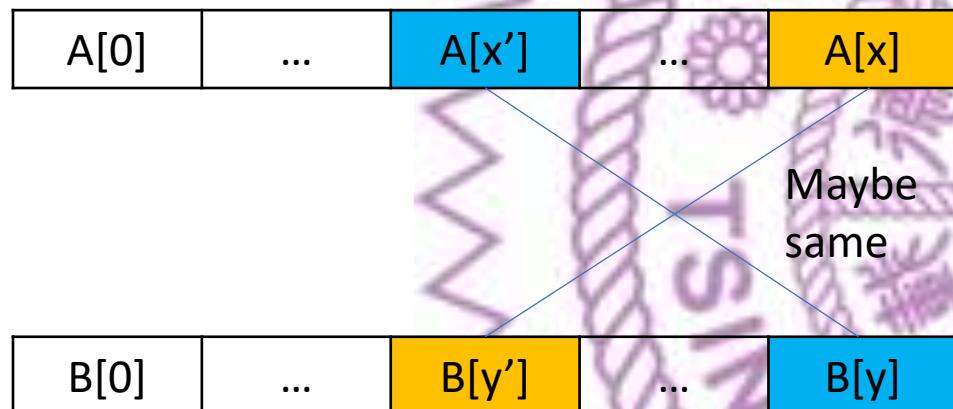
same

Case $A[x] == B[y]$

這個例子中 x,y 必然是LCS的一部分
所以

$$DP_{x,y} = DP_{x-1,y-1} + 1$$

Step 2: 想狀態轉移式



Case $A[x] \neq B[y]$

這個例子中 x, y 都各自可能是 LCS 的一部分
所以

$$DP_{x,y} = \max\{DP_{x-1,y}, DP_{x,y-1}\}$$

Step 2: 想狀態轉移式

$$DP_{x,y} = \begin{cases} 0 & , x < 0, y < 0 \\ DP_{x-1,y-1} + 1 & , A[x] = B[y] \\ \max\{DP_{x-1,y}, DP_{x,y-1}\} & , A[x] \neq B[y] \end{cases}$$

時間複雜度

- $2D/0D$
- 狀態數 \times 轉移時間
- $|A| \times |B| \times O(1) = O(|A||B|)$



Longest Palindromic Subsequence

- <https://leetcode.com/problems/longest-palindromic-subsequence/>
- 輸出最長迴文子序列長度
- Input: S = "bbbab"
- Output: 4
最長迴文子序列为："bbbb"

Step 1: 定義狀態

- 設 $DP_{l,r}$ 表示 $S[l \sim r]$ 的最長迴文子序列長度



Step 2: 想狀態轉移式

$s[l]$	$s[l+1]$	$s[l+2]$...	$s[r]$
--------	----------	----------	-----	--------



Step 2: 想狀態轉移式

$s[l]$	$s[l+1]$	$s[l+2]$...	$s[r]$
--------	----------	----------	-----	--------

Case $s[l] == s[r]$

這個例子中 x, y 必然是最長迴文的一部分
所以

$$DP_{l,r} = DP_{l+1,r-1} + 2$$

Step 2: 想狀態轉移式

$s[l]$	$s[l+1]$	$s[l+2]$...	$s[r]$
--------	----------	----------	-----	--------

Case $s[l] \neq s[r]$

這個例子中 x, y 都各自可能最長迴文的一部分
所以

$$DP_{l,r} = \max\{DP_{l+1,r}, DP_{l,r-1}\}$$

Step 2: 想狀態轉移式

$$DP_{l,r} = \begin{cases} r - l + 1 & , r - l + 1 \leq 1 \\ DP_{l+1,r-1} + 2 & , S[l] = S[r] \\ \max\{DP_{l+1,r}, DP_{l,r-1}\} & , S[l] \neq S[r] \end{cases}$$

Botton Up: 由小到大枚舉區間

```
int solve(string S) {
    int n = S.size();
    vector<vector<int>> DP(n, vector<int>(n));
    for (int i = 0; i < n; ++i) DP[i][i] = 1;
    for (int len = 2; len <= n; ++len) {
        for (int l = 0; l + len <= n; ++l) {
            int r = l + len - 1;
            if (S[l] == S[r]) DP[l][r] = DP[l + 1][r - 1] + 2;
            else DP[l][r] = max(DP[l + 1][r], DP[l][r - 1]);
        }
    }
    return DP[0][n - 1];
}
```

時間複雜度

- $2D/0D$
- 狀態數 \times 轉移時間
- $|S|^2 \times O(1) = O(|S|^2)$





特殊優化

資料結構優化

矩陣快速幕



Longest Increasing Subsequence

- <https://leetcode.com/problems/longest-increasing-subsequence/>
- 輸出最長遞增子序列長度
- Input: S = [0,3,1,6,2,2,7,-1] $O(n \log n)$
- Output: 4
其中一個最長遞增子序列為：[0,1,2,7]

這裡計算太慢了

```
vector<int> S;

int solve() {
    vector<int> DP(S.size());
    int ans = 0;
    for (int i = 0; i < S.size(); ++i) {
        DP[i] = 0;
        for (int j = 0; j < i; ++j) {
            if (S[i] > S[j])
                DP[i] = max(DP[i], DP[j]);
        }
        DP[i] += 1;
        ans = max(ans, DP[i]);
    }
    return ans;
}
```

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7

V	1	2	3	4	5

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1							

V	1	2	3	4	5	
	-7					

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2						

V	1	2	3	4	5	
	-7	10				

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2					

V	1	2	3	4	5	
	-7	9				

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2	2				

V	1	2	3	4	5
	-7	2			

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2	2	3			

V	1	2	3	4	5
	-7	2	3		

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2	2	3	4		

V	1	2	3	4	5
	-7	2	3	8	

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2	2	3	4	4	4

V	1	2	3	4	5
	-7	2	3	8	

$V[i]$: 紀錄滿足 $DP[x] = i$ 的最小值

S	0	1	2	3	4	5	6	7
	-7	10	9	2	3	8	8	1

DP	0	1	2	3	4	5	6	7
	1	2	2	2	3	4	4	2

V	1	2	3	4	5
	-7	1	3	8	

V 陣列永遠遞增，利用二分搜！

LIS 資料結構優化

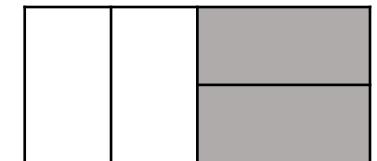
```
vector<int> S;

int solve() {
    if (S.size() == 0) return 0;
    vector<int> V;
    V.emplace_back(S[0]);
    for (size_t i = 1; i < S.size(); ++i) {
        if (S[i] > V.back())
            V.emplace_back(s[i]);
        else
            *lower_bound(V.begin(), V.end(), S[i]) = S[i];
    }
    return V.size(); // V 陣列的長度等同於答案
}
```

骨牌覆蓋

- 得到關係式

$$\bullet f_n = \begin{cases} 1 & , n \leq 1 \\ f_{n-1} + f_{n-2}, & n > 1 \end{cases}$$



$n = 4$ 的所有方法

```
long long f(int n) {  
    if (n <= 1) return 1;  
    return f(n - 1) + f(n - 2);  
}
```

$O(\log n)$

矩陣乘法

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e \\ g \end{bmatrix} = \begin{bmatrix} ae + bg \\ ce + dg \end{bmatrix}$$

觀察 – 使用矩陣快速幕

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} f_0 + f_1 \\ f_1 \end{bmatrix} = \begin{bmatrix} f_2 \\ f_1 \end{bmatrix}$$

a^b 我們有教過 $O(\log b)$ 的快速幕

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} f_1 \\ f_0 \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix}$$

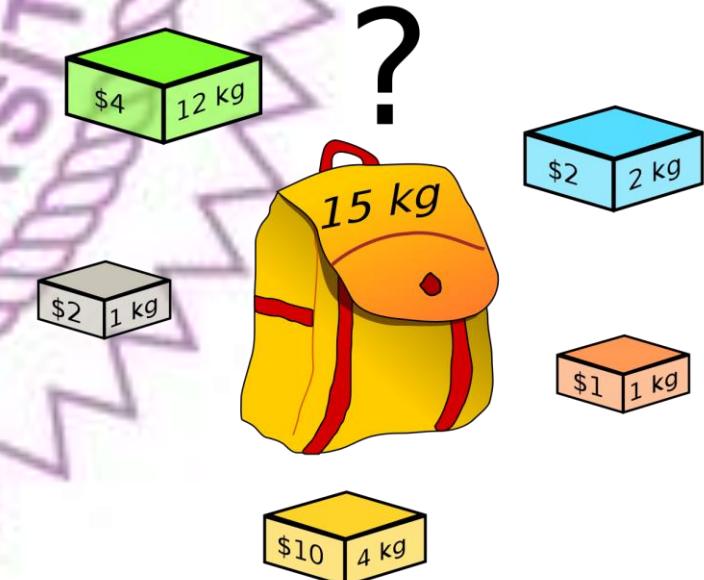


Knapsack Problem

背包問題

0-1 背包問題

- 你有一個限重為 W 的背包
- 有 n 個物品，第 i 個物品的重量和價格分別為 w_i, c_i
- 每個物品你可以選擇放或不放
請問在不超過背包限重的情況下，能拿到的價格總和最多為多少
- $n \leq 1000, 0 \leq W \leq 1000, \sum c_i \leq 1000$



枚舉法

- 每個物品只有選或不選兩種
- 有 2^n 種可能性 → $O(2^n)$ 枚舉所有可能
- 太慢



Step 1: 定義狀態 1

- 設 $DP_{x,y}$ 表示：
只考慮編號 $1 \sim x$ 的物品
總價值是 y 時最少要花費的重量

- 目標答案就是

$$\max_{0 \leq y \leq \sum c_i} \{y: DP_{n,y} \leq W\}$$

Step 1: 定義狀態 2

- 設 $DP_{x,y}$ 表示：
只考慮編號 $1 \sim x$ 的物品
且背包中物品總重量剛好是 y 時的最佳總價值
- 目標答案就是

$$\max_{0 \leq y \leq W} \{DP_{n,y}\}$$

Step 1: 定義狀態 3

- 設 $DP_{x,y}$ 表示：
只考慮編號 $1 \sim x$ 的物品
且背包限重是 y 時的最佳總價值
- 目標答案就是

$$DP_{n,W}$$

Step 2: 想狀態轉移式

- 考慮要不要放第 x 物品



Step 2: 想狀態轉移式

- 考慮要不要放第 x 物品
- 不放第 x 物品
- $DP_{x,y} = DP_{x-1,y}$



TSING HUA UNIVERSITY

Step 2: 想狀態轉移式

- 考慮要不要放第 x 物品
- 放第 x 物品
- $DP_{x,y} = DP_{x-1,y-w_x} + c_x$



Step 2: 想狀態轉移式

$$\cdot DP_{x,y} = \begin{cases} 0, & x = 0 \\ \max\{DP_{x-1,y}, DP_{x-1,y-w_x} + c_x\}, & x > 0 \end{cases}$$

```
int DP[MAXN][MAXW] = {};  
  
int solve() {  
    for (int x = 1; x <= n; ++x)  
        for (int y = 0; y <= w; ++y) {  
            DP[x][y] = DP[x - 1][y];  
            if (y >= w[x])  
                DP[x][y] = max(DP[x][y], DP[x - 1][y - w[x]] + c[x]);  
        }  
    return DP[n][w];  
}
```

時間複雜度

- 狀態數 × 轉移時間
- $nW \times O(1) = O(nW)$



觀察狀態轉移式

$$\cdot DP_{x,y} = \begin{cases} 0, & x = 0 \\ \max\{DP_{x-1,y}, DP_{x-1,y-w_x} + c_x\}, & x > 0 \end{cases}$$

- 計算 DP_x 時只會用到 DP_{x-1} 的資訊

DP_{x-1}	DP_x		

省略沒用到的空間

```
int DP[2][MAXW] = {};
int solve() {
    for (int x = 1; x <= n; ++x)
        for (int y = 0; y <= W; ++y) {
            DP[x & 1][y] = DP[(x & 1) ^ 1][y];
            if (y >= w[x])
                DP[x & 1][y] = max(DP[x & 1][y], DP[(x & 1) ^ 1][y - w[x]] + c[x]);
        }
    return DP[n & 1][W];
}
```

更加細緻的觀察

$$\cdot DP_{x,y} = \begin{cases} 0 & , x = 0 \\ \max\{DP_{x-1,y}, DP_{x-1,y-w_x} + c_x\} & , x > 0 \end{cases}$$

- 計算 $DP_{x,y}$ 時只會用到 $DP_{x-1,i}, i \leq y$ 的資訊

DP_{x-1}

DP_x

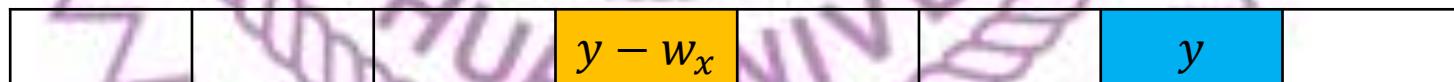
			$y - w_x$			y	
						y	

滾動陣列 – 只用一個維度記錄答案

$$\cdot DP_{x,y} = \begin{cases} 0, & x = 0 \\ \max\{DP_{x-1,y}, DP_{x-1,y-w_x} + c_x\}, & x > 0 \end{cases}$$

- 計算 $DP_{x,y}$ 時只會用到 $DP_{x-1,i}, i \leq y$ 的資訊
- 讓 y 由大到小計算可以把陣列壓成一維

DP



滾動陣列 – 只用一個維度記錄答案

DP_{x-1} 的資料:



DP_x 的資料:



DP_{x-1}

			$y - w_x$			y	
						y	

DP_x

DP

			$y - w_x$			y	
						y	

滾動陣列 – 只用一個維度記錄答案

DP_{x-1} 的資料:



DP_x 的資料:



DP_{x-1}

		$y - w_x$			y		
					y		

DP_x

DP

		$y - w_x$			y		
					y		

滾動陣列 – 只用一個維度記錄答案

DP_{x-1} 的資料:



DP_x 的資料:

DP_{x-1}

	$y - w_x$			y			
				y			

DP_x

DP

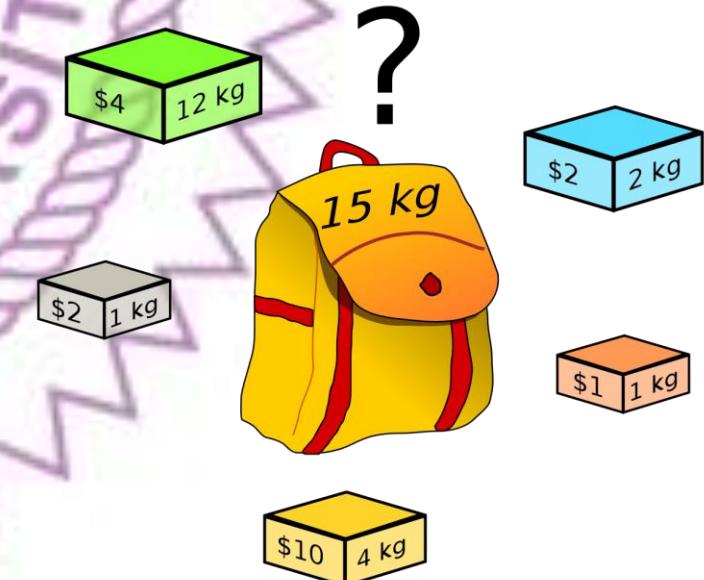
	$y - w_x$			y			
				y			

滾動陣列 – 只用一個維度記錄答案

```
int DP[MAXW] = {};
int solve() {
    for (int x = 1; x <= n; ++x)
        for (int y = W; y >= w[x]; --y)
            DP[y] = max(DP[y], DP[y - w[x]] + c[x]);
    return DP[W];
}
```

完全背包問題

- 你有一個限重為 W 的背包
- 有 n 種物品，第 i 個物品的重量和價格分別為 w_i, c_i
- 每種物品你可以選擇放任意多個或不放
請問在不超過背包限重的情況下，能拿到的價格總和最多為多少
- $n \leq 1000, 0 \leq W \leq 1000, \sum c_i \leq 1000$



Step 1: 定義狀態

- 設 $DP_{x,y}$ 表示：
只考慮編號 $1 \sim x$ 種類的物品(無論每個物品選多少個)
且背包限重是 y 時的最佳總價值
- 目標答案就是

$$DP_{n,W}$$

Step 2: 想狀態轉移式

- 考慮放第 x 物品的方法



Step 2: 想狀態轉移式

- 考慮放第 x 物品的方法
- 不放第 x 物品
- $DP_{x,y} = DP_{x-1,y}$

Step 2: 想狀態轉移式

- 考慮放第 x 物品的方法
 - 放第 x 物品(不論放多少個)
 - $DP_{x,y} = DP_{x,y-w_x} + c_x$
- 比較：只能放一個的方法
 $DP_{x,y} = DP_{x-1,y-w_x} + c_x$

Step 2: 想狀態轉移式

$$\cdot DP_{x,y} = \begin{cases} 0 & , x = 0 \\ \max\{DP_{x-1,y}, DP_{x,y-w_x} + c_x\} & , x > 0 \end{cases}$$



時間複雜度

- 狀態數 × 轉移時間
- $nW \times O(1) = O(nW)$



必讀參考書籍

- 背包九講



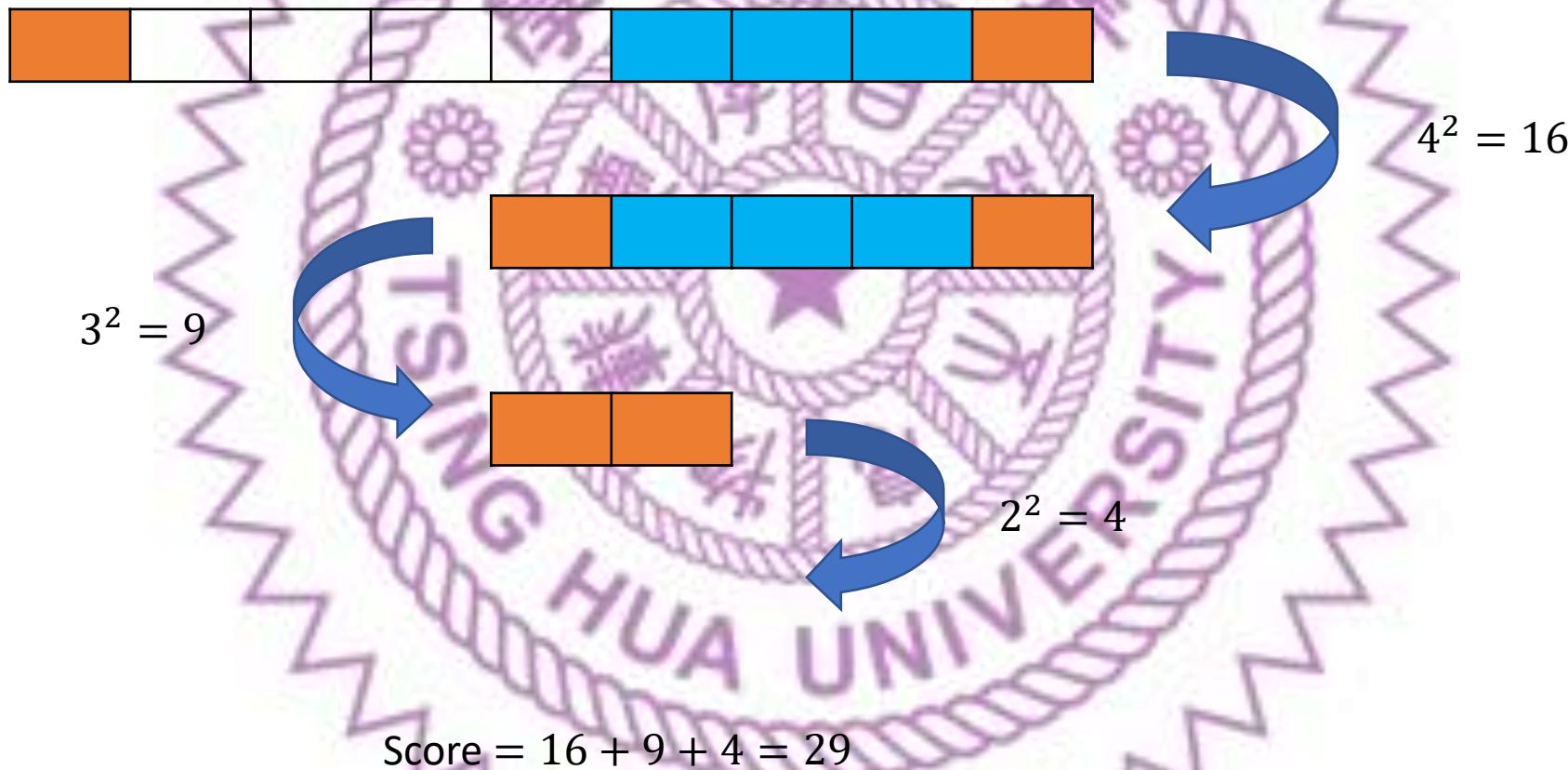


不直觀的狀態

UVA 10559 Blocks

- 有 $N(1 \leq N \leq 200)$ 個方塊排成一列，每個方塊有自己的顏色
每個顏色有不同的編號，第 i 個方塊的顏色編號是 $c_i(1 \leq c_i \leq n)$
- 同一個顏色的區間可以消除
若消除的區間長度為 k ，則可以得到 k^2 的分數，初始分數為 0
- 連續一段區間被消除後，剩餘左右兩端會合併起來
- 問最多能得到多少分數

UVA 10559 Blocks



Step 1: 定義狀態

- 設 $DP_{l,r}$ 表示消除 $c_l \sim c_r$ 得到的最佳分數

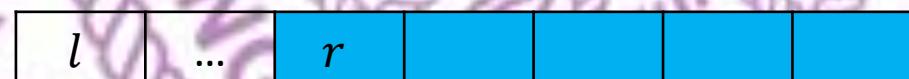


Step 2: 想狀態轉移式

$$DP_{l,r} = \max_{c_i=c_{i+1}=\dots=c_j} \{DP_{l,i-1} + DP_{j+1,r} + (j - i + 1)^2\}$$

Step 1: 定義狀態

- 設 $DP_{l,r,k}$ 表示：
消除 $c_l \sim c_r$ 以及在 r 後面接上的 k 個與 c_r 同色的方塊的最佳分數



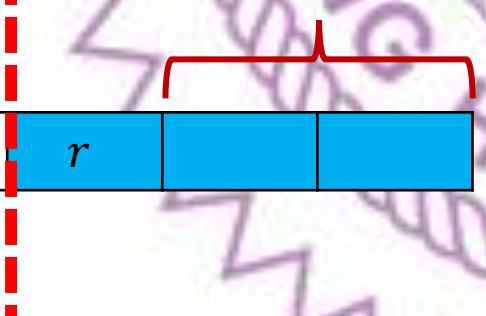
Step 2: 想狀態轉移式

切法1: 把 r 切出來處理

$$DP_{l,r,k} = DP_{l,r-1,0} + (k+1)^2$$

k 個與 c_r 同色的方塊

l	\dots	$r-1$	r		
-----	---------	-------	-----	--	--



Step 2: 想狀態轉移式

切法2: 若 $c_i = c_r$, 可以先消去 $[i + 1, r - 1]$

$$DP_{l,r,k} = \max_{c_i=c_r} \{ DP_{l,i,k+1} + DP_{i+1,r-1,0} \}$$

消掉這段

k 個與 c_r 同色的方塊

l	\dots	$i - 1$	i	$i + 1$	\dots	$r - 1$	r		
-----	---------	---------	-----	---------	---------	---------	-----	--	--

l	\dots	$i - 1$	i			
-----	---------	---------	-----	--	--	--

程式碼

```
const int MAXN = 201;
int c[MAXN], n; // input
int dp[MAXN][MAXN][MAXN];

int dfs(int l, int r, int k) {
    if (l > r) return 0;
    if (dp[l][r][k]) return dp[l][r][k];
    int ans = dfs(l, r - 1, 0) + (1 + k) * (1 + k);
    for (int i = r - 1; i >= l; --i)
        if (c[i] == c[r])
            ans = max(ans, dfs(l, i, k + 1) + dfs(i + 1, r - 1, 0));
    return dp[l][r][k] = ans;
}
```

時間複雜度

- $3D/1D$
- 狀態數 \times 轉移時間
- $n^3 \times O(n) = O(n^4)$



Dynamic Programming 2

日月卦長





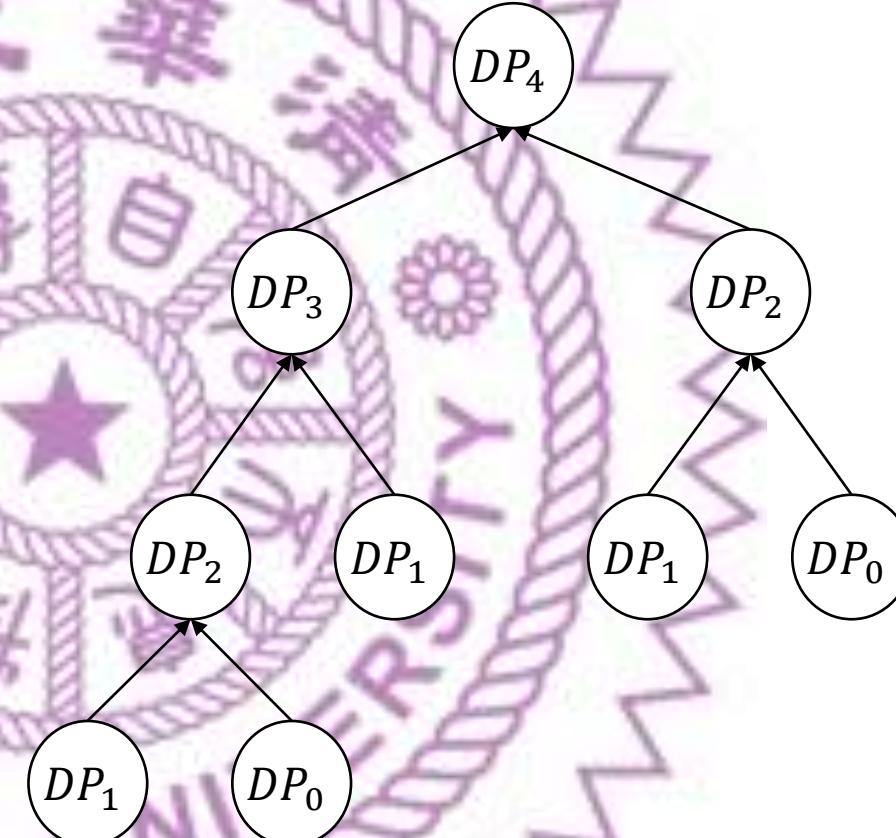
動態規劃 & 有向無環圖

DP & DAG



費式數列

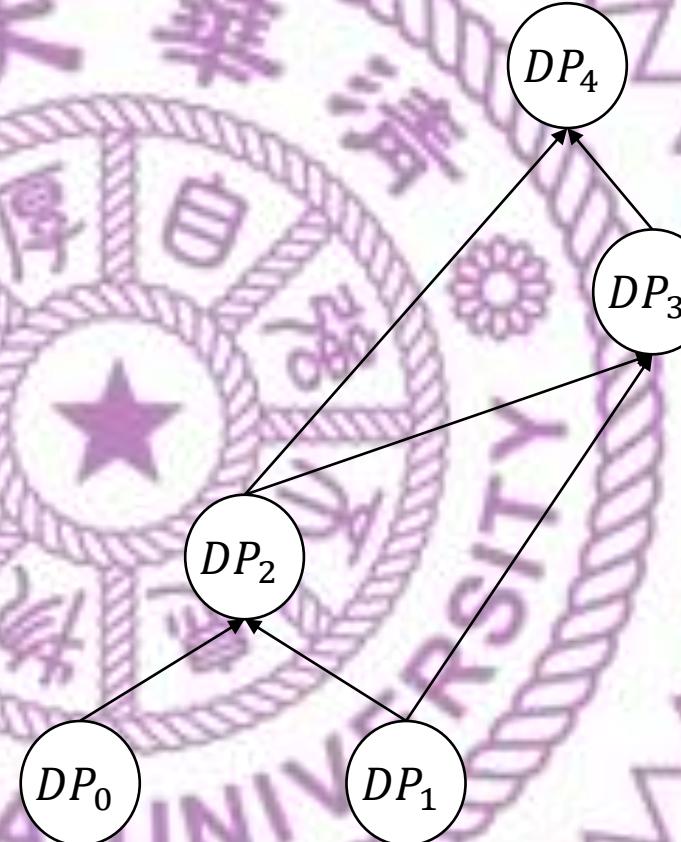
$$DP_n = \begin{cases} 1 & , n \leq 1 \\ DP_{n-1} + DP_{n-2}, & n > 1 \end{cases}$$



費式數列

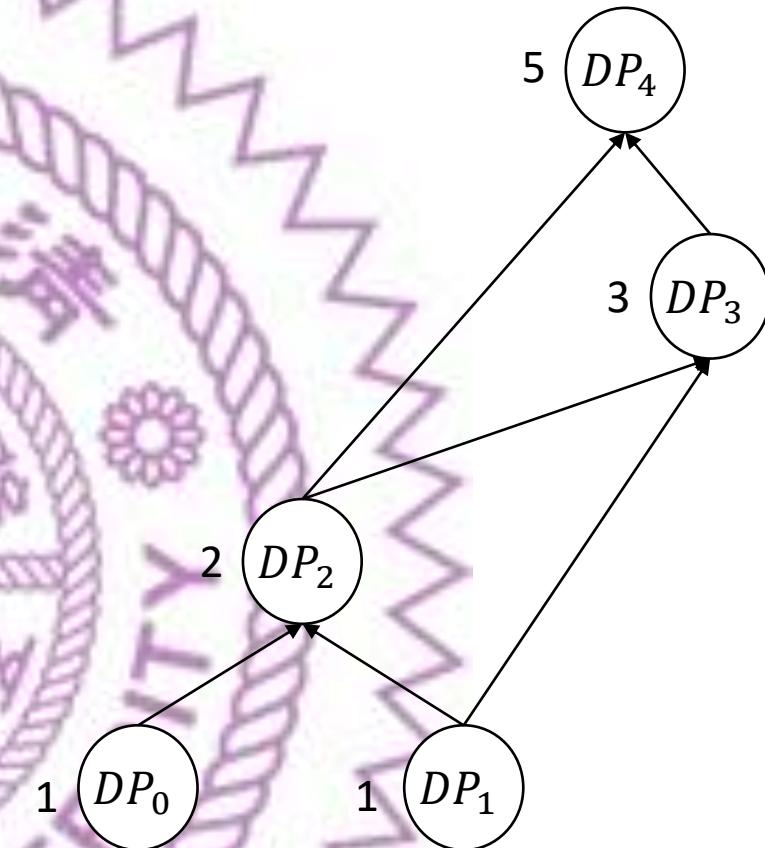
$$DP_n = \begin{cases} 1 & , n \leq 1 \\ DP_{n-1} + DP_{n-2}, & n > 1 \end{cases}$$

```
map<int, long long> DP;
long long f(int n) {
    if (n <= 1) return 1;
    if (DP.count(n)) return DP[n];
    return DP[n] = f(n - 1) + f(n - 2);
}
```



DP 與 DAG 的關係

- 將狀態看成點
- 狀態轉移式定義了有向邊
- 會變出一張 DAG
- 狀態的計算順序就是拓樸排序

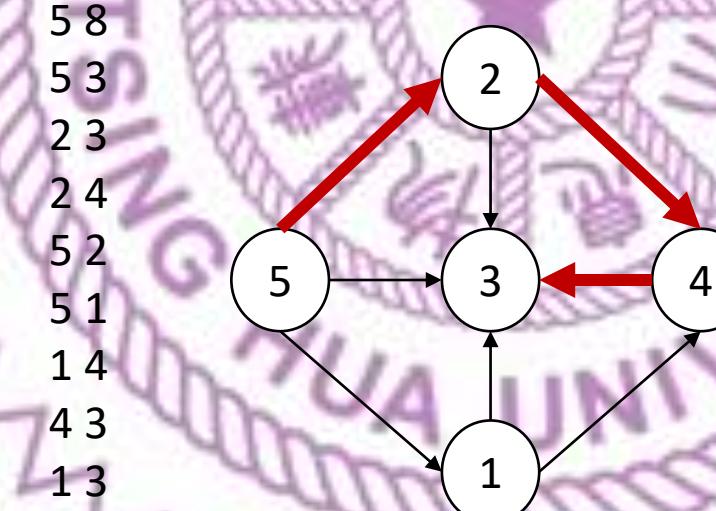


$$DP_n = \begin{cases} 1 & , n \leq 1 \\ DP_{n-1} + DP_{n-2}, & n > 1 \end{cases}$$

Atcoder Edu Dp Contest – G. Longest Path

- https://atcoder.jp/contests/dp/tasks/dp_g
- 紿你一個有向無環圖，問你最長路徑的長度

這個問題在一般圖上是 NP complete



狀態轉移式

$$DP_v = \begin{cases} 0 & , \text{degree}^{in}(v) = 0 \\ \max_{(u,v) \in E} \{DP_u\} + 1, & \text{degree}^{in}(v) > 0 \end{cases}$$

Diagram illustrating the dynamic programming formula for state transfer. A graph node v has incoming edges from nodes a , b , and c . Node a has $DP_a = 5$, node b has $DP_b = 2$, and node c has $DP_c = 3$. The formula calculates $DP_v = \max(DP_a, DP_b, DP_c) + 1 = 6$.

Top Down – 反著存圖

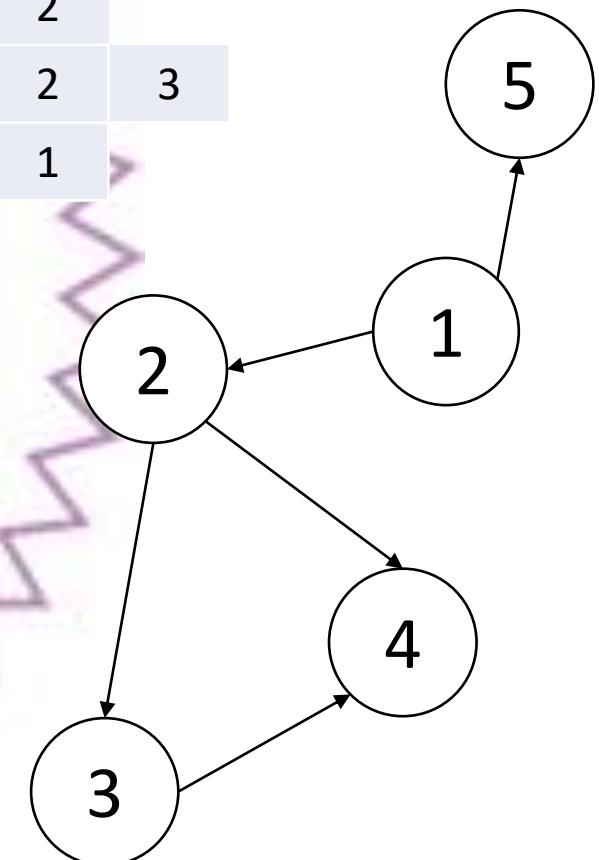
n 個點, m 條邊

5	6
1	2
2	3
2	4
1	5
3	4

m 條邊

```
vector<vector<int>> rG;
int n, m;
cin >> n >> m;
rG.assign(n + 1, {});
while (m--) {
    int u, v;
    cin >> u >> v;
    rG[v].emplace_back(u);
}
```

1		
2	1	
3	2	
4	2	3
5	1	



Top Down – 照著公式寫

```
vector<int> DP;
int dfs(int v) {
    if (DP[v] != -1) return DP[v];
    for (int u : rG[v])
        DP[v] = max(DP[v], dfs(u));
    return DP[v] += 1;
}
int solve(int n) {
    DP.assign(rG.size(), -1);
    int ans = 0;
    for (int v = 1; v <= n; ++v)
        ans = max(ans, dfs(v));
    return ans;
}
```

Bottom Up - 紀錄 in-degree

n 個點, m 條邊

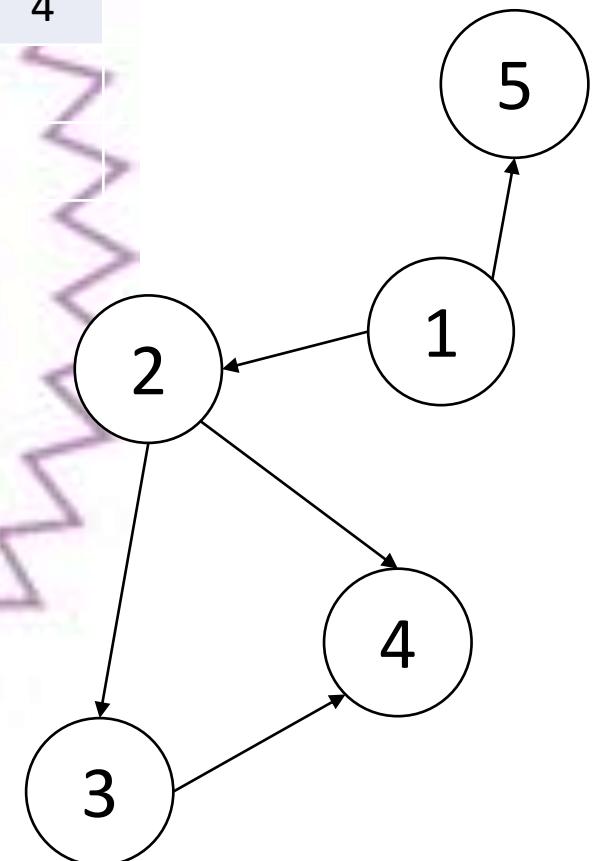
5	6
1	2
2	3
2	4
1	5
3	4

m 條邊

```
vector<vector<int>> G;
vector<int> in;
int n, m;
cin >> n >> m;
G.assign(n + 1, {});
in.assign(n + 1, 0);
while (m--) {
    int u, v;
    cin >> u >> v;
    G[u].emplace_back(v);
    ++in[v];
}
```

1	2	3	4	5
0	1	1	2	1

1	2	5
2	3	4
3		4
4		
5		



Bottom Up – 拓樸排序時順便計算

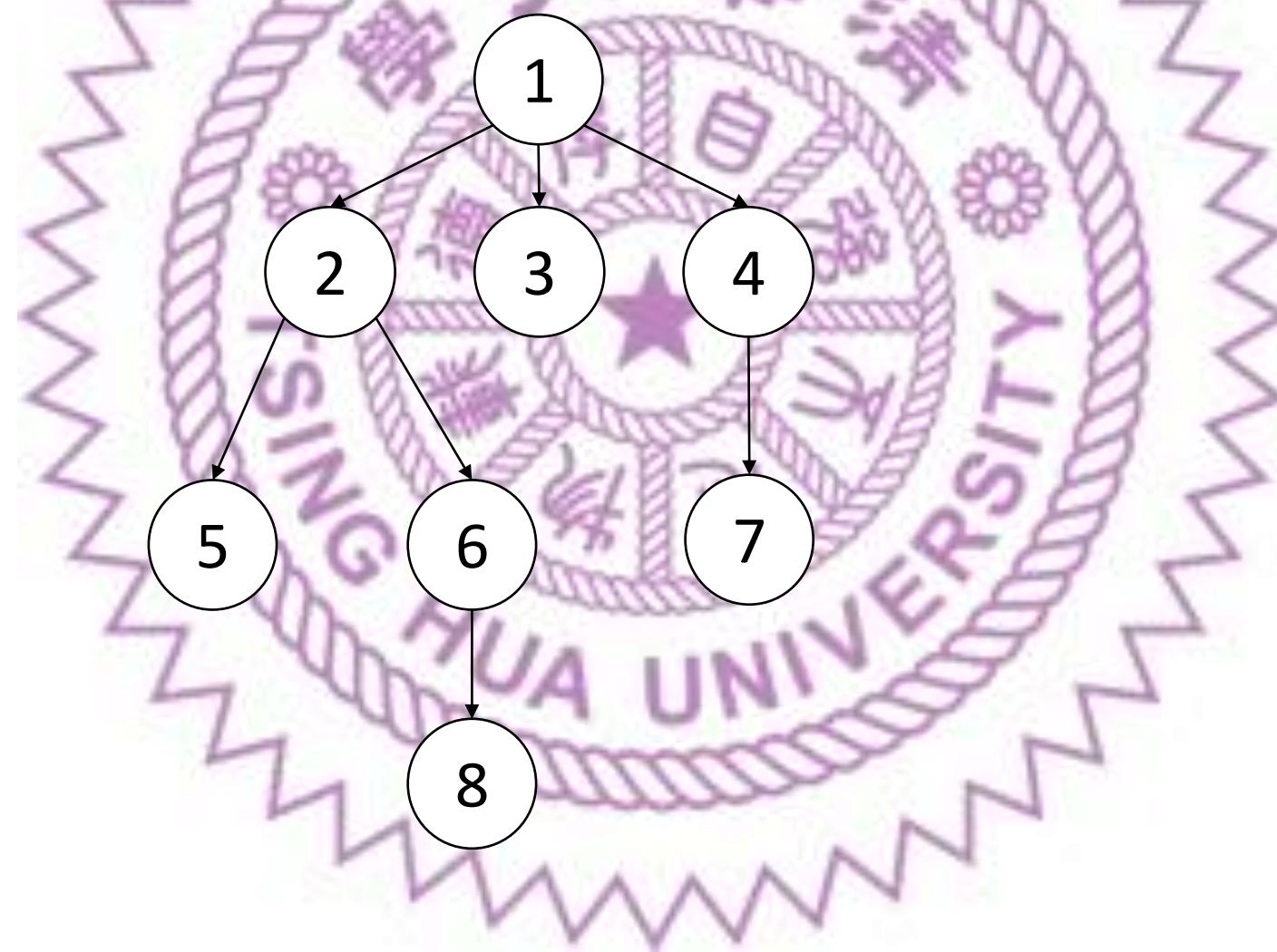
```
int solve(int n) {
    vector<int> DP(G.size(), 0);
    vector<int> Q;
    for (int u = 1; u <= n; ++u)
        if (in[u] == 0)
            Q.emplace_back(u);
    for (size_t i = 0; i < Q.size(); ++i) {
        int u = Q[i];
        for (auto v : G[u]) {
            DP[v] = max(DP[v], DP[u] + 1);
            if (--in[v] == 0)
                Q.emplace_back(v);
        }
    }
    return *max_element(DP.begin(), DP.end());
}
```

樹上動態規劃

Dynamic Programming on Tree

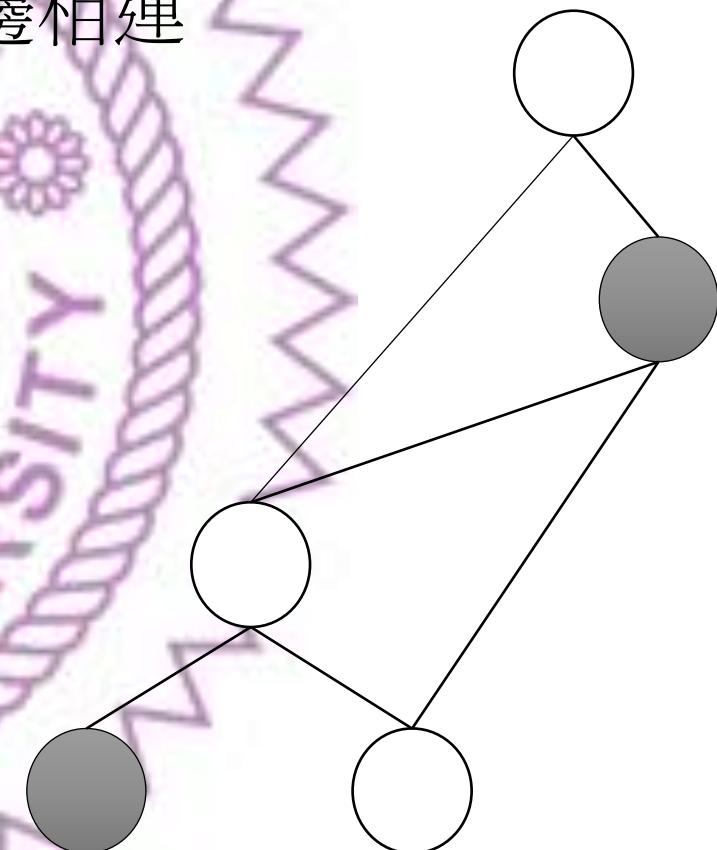


有根樹 ⊂ 有向無環圖



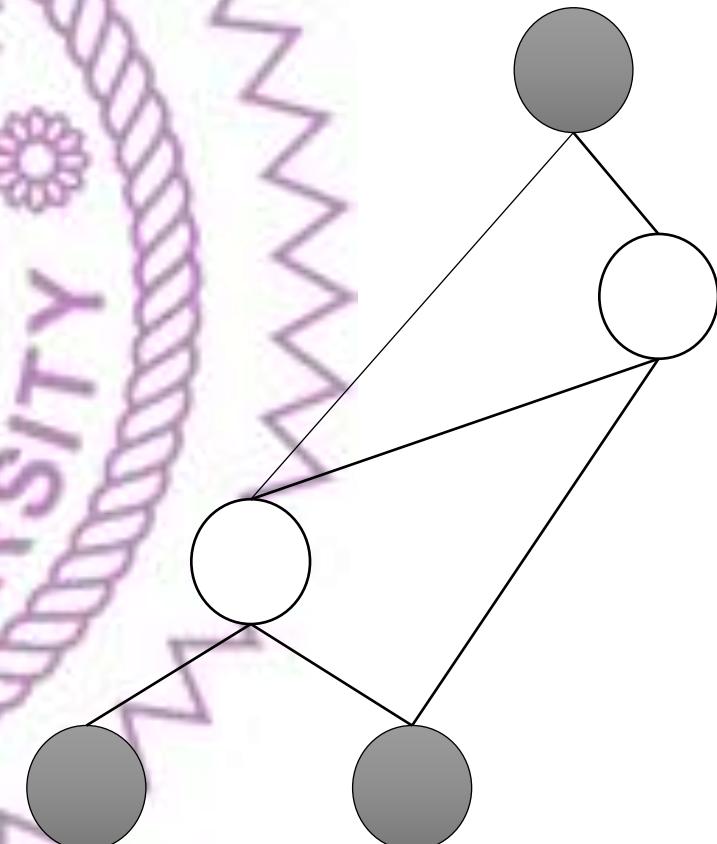
獨立集 Independent set

- 一張圖，選一些點，這些點彼此之間沒有邊相連



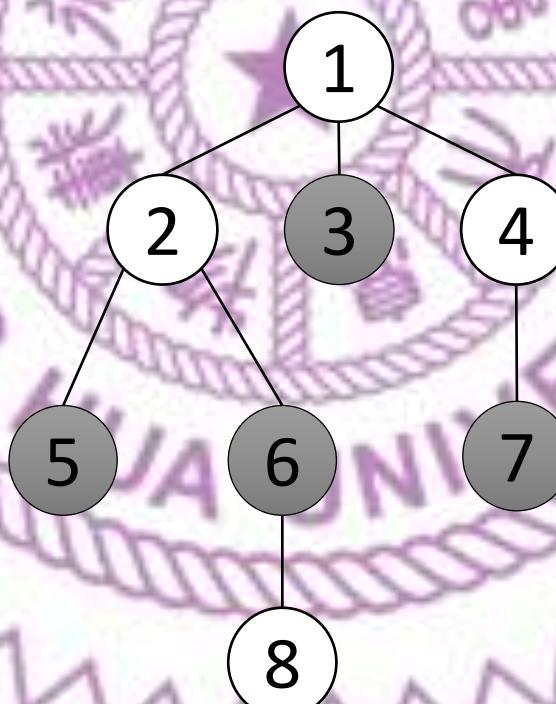
最大獨立集 Maximum independent set

- 一張圖中，點數量最多的獨立集
- 找出最大獨立集在一般圖上是 NP hard
- 但是在樹、二分圖等特殊圖上
存在高效演算法(方法不同)



樹上最大獨立集

- 輸入一棵有 n 個點的樹，要挑選一群彼此不相鄰的點，而且挑選的點越多點越好。請計算最多可以挑多少點。



定義狀態

- 假設是有根樹
- $DP_{u,0}$ 表示以 u 為根的子樹，不選 u 時的最佳值
- $DP_{u,1}$ 表示以 u 為根的子樹，選 u 時的最佳值



狀態轉移式

- 不選 u 時， u 的小孩要選不選都是可以的
- 選 u 時， u 的小孩都不能選



狀態轉移式

$$DP_{u,0} = \sum_{v \in \text{child}(u)} \max\{DP_{v,0}, DP_{v,1}\}$$
$$DP_{u,1} = \left(\sum_{v \in \text{child}(u)} DP_{v,0} \right) + 1$$



無根樹的輸入 (與圖的輸入相同)

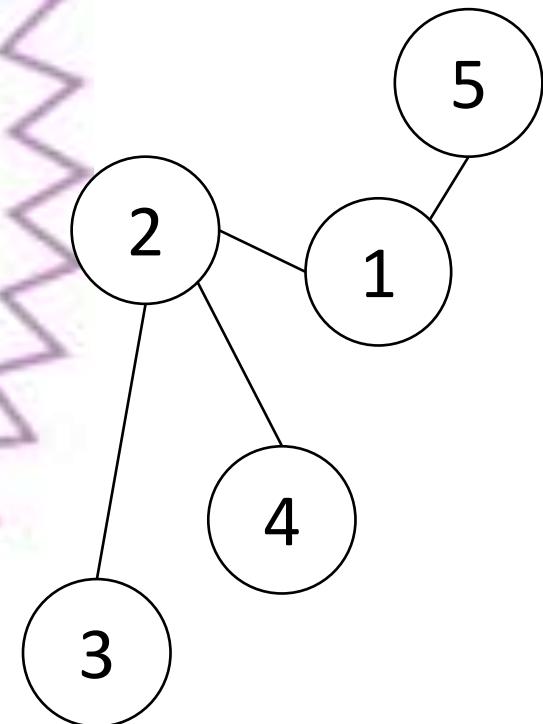
n 個點

$n - 1$ 條邊

5
1 2
2 3
2 4
1 5

1	2	5	
2	1	3	4
3	2		
4	2		
5	1		

```
vector<vector<int>> Tree;
int n;
cin >> n;
Tree.assign(n + 1, {});
for (int i = 0; i < n - 1; ++i) {
    int u, v;
    cin >> u >> v;
    Tree[u].emplace_back(v);
    Tree[v].emplace_back(u);
}
```



程式碼

```
vector<int> DP[2];
int dfs(int u, int pick, int parent = -1) {
    if (u == parent) return 0;
    if (DP[pick][u]) return DP[pick][u];
    if (Tree[u].size() == 1) return pick; // 葉子
    for (auto v : Tree[u]) {
        if (pick == 0) {
            DP[pick][u] += max(dfs(v, 0, u), dfs(v, 1, u));
        } else {
            DP[pick][u] += dfs(v, 0, u);
        }
    }
    return DP[pick][u] += pick;
}
int solve(int n) {
    DP[0] = DP[1] = vector<int>(n + 1, 0);
    return max(dfs(1, 0), dfs(1, 1));
}
```

Travelling salesman problem

旅行推銷員問題



Travelling salesman problem



Travelling salesman problem

示意圖



旅行推銷員問題

- 日日是聖地亞戈集團的推銷員
他要去美國的 $n(n \leq 15)$ 個城市中推銷金坷垃
城市的編號為 $0 \sim n - 1$
- 設 $dist(x, y)$ 表示城市 x 到城市 y 的距離
- 日日想從聖地亞戈 (城市 0) 出發，經過所有城市恰好各一次後回
到聖地亞戈，請你幫助伯爵為日日找出總距離最少的路徑

範例輸入輸出

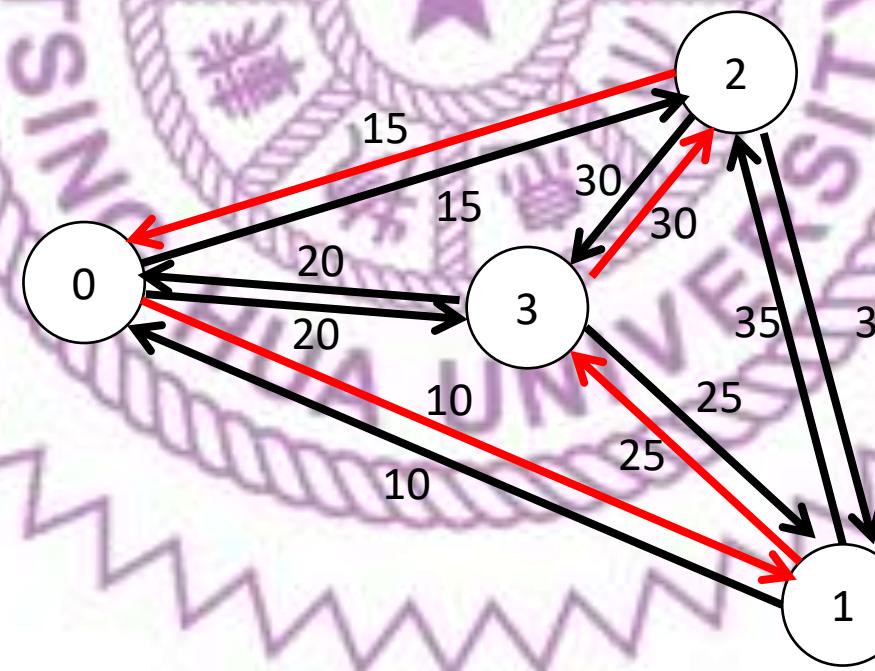
Input

4
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0

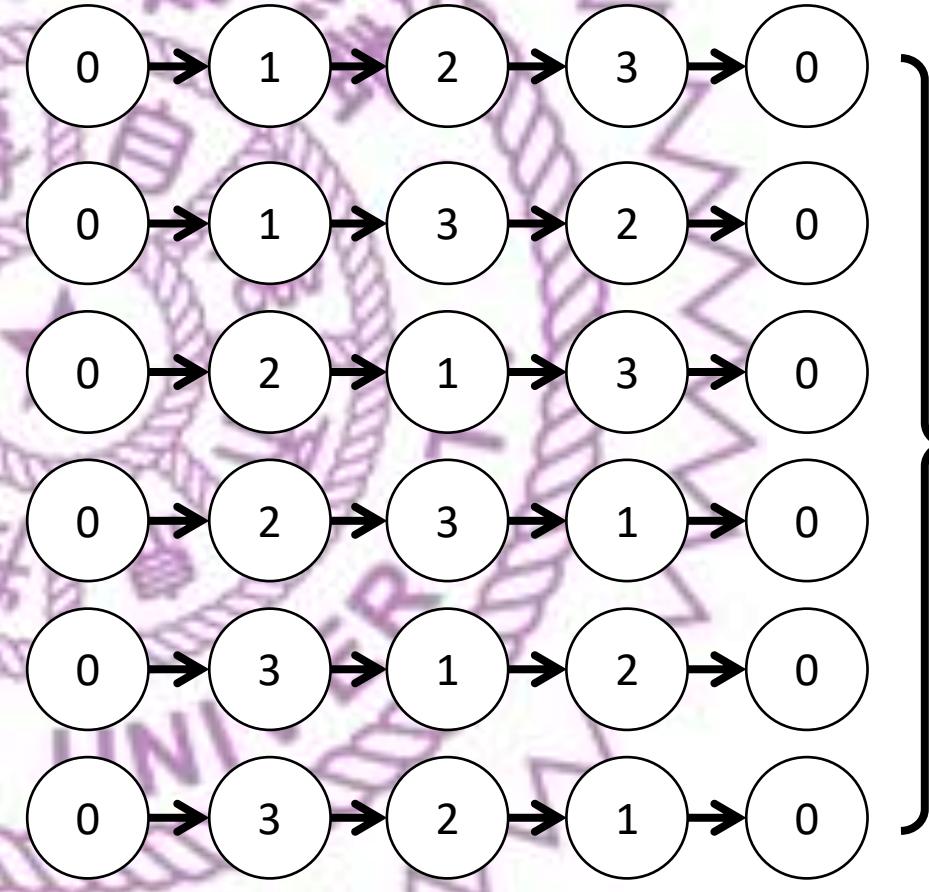
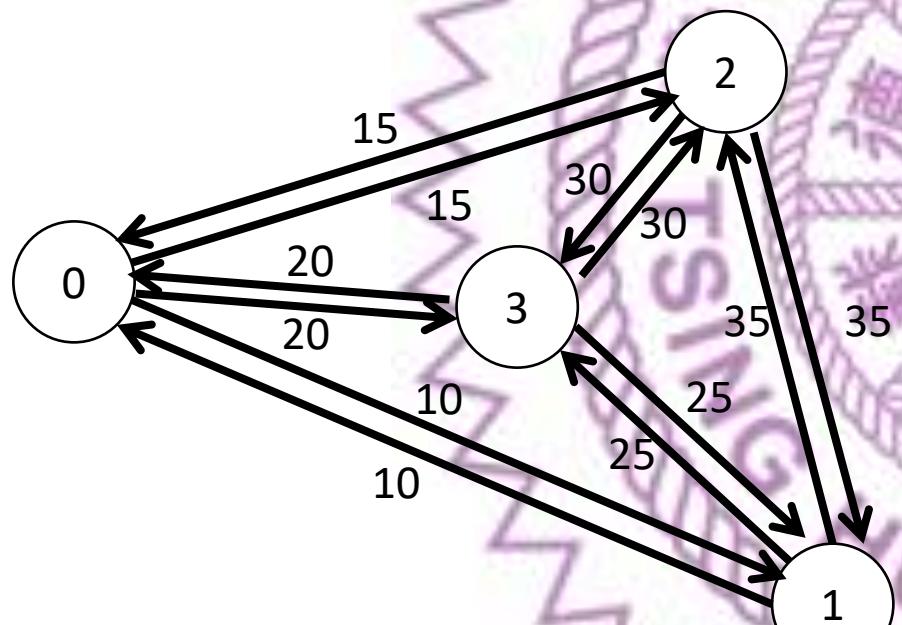
Output

80

順序 : 0 -> 1 -> 3 -> 2 -> 0
 $10+25+30+15 = 80$



暴力法 $O(n!)$



每種排列計算總距離 $O(n)$

$(n - 1)!$ 種排列

暴力法程式碼

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

vector<bool> used;
int ans;

void dfs(int x, int cost);

int solve() {
    used.resize(n, false);
    ans = 0x3f3f3f3f;
    dfs(0, 0);
    return ans;
}
```

```
void dfs(int x, int cost) {
    bool isAllTrue = true;
    for (auto y : used) isAllTrue &= y;
    if (isAllTrue && x == 0) {
        ans = min(ans, cost);
        return;
    }
    for (int y = 0; y < n; ++y) {
        if (y == x || used[y]) continue;
        used[y] = true;
        dfs(y, cost + dist[x][y]);
        used[y] = false;
    }
}
```

無法成為 DP 的原因

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

vector<bool> used;
int ans;

void dfs(int x, int cost);

int solve() {
    used.resize(n, false);
    ans = 0x3f3f3f3f;
    dfs(0, 0);
    return ans;
}
```

```
void dfs(int x, int cost) {
    bool isAllTrue = true;
    for (auto y : used) isAllTrue &= y;
    if (isAllTrue && x == 0) {
        ans = min(ans, cost);
        return;
    }
    for (int y = 0; y < n; ++y) {
        if (y == x || used[y]) continue;
        used[y] = true;
        dfs(y, cost + dist[x][y]);
        used[y] = false;
    }
}
```

Step 1: 反向思考讓 ans 變成 local 變數

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

vector<bool> used;

void dfs(int x);

int solve() {
    used.resize(n, true);
    return dfs(0);
}
```

```
int dfs(int x) {
    bool isAllFalse = true;
    for (auto y : used) isAllFalse &= !y;
    if (isAllFalse) {
        if (x == 0) return 0;
        return 0x3f3f3f3f;
    }
    int ans = 0x3f3f3f3f;
    for (int y = 0; y < n; ++y) {
        if (y == x || !used[y]) continue;
        used[y] = false;
        ans = min(ans, dfs(y) + dist[y][x]);
        used[y] = true;
    }
    return ans;
}
```

Step 2: used 可以直接當成參數

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

void dfs(int x);

int solve() {
    vector<bool> used(n, true);
    return dfs(0, used);
}
```

```
int dfs(int x, vector<bool> used) {
    bool isAllFalse = true;
    for (auto y : used) isAllFalse &= !y;
    if (isAllFalse) {
        if (x == 0) return 0;
        return 0x3f3f3f3f;
    }
    int ans = 0x3f3f3f3f;
    for (int y = 0; y < n; ++y) {
        if (y == x || !used[y]) continue;
        used[y] = false;
        ans = min(ans, dfs(y, used) + dist[y][x]);
        used[y] = true;
    }
    return ans;
}
```

Step 3: 記憶算過的答案

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

map<tuple<int, vector<bool>>, int> DP;
void dfs(int x);

int solve() {
    vector<bool> used(n, true);
    return dfs(0, used);
}
```

```
int dfs(int x, vector<bool> used) {
    bool isAllFalse = true;
    for (auto y : used) isAllFalse &= !y;
    if (isAllFalse) {
        if (x == 0) return 0;
        return 0x3f3f3f3f;
    }
    if (DP.count({x, used})) return DP[{x, used}];
    int ans = 0x3f3f3f3f;
    for (int y = 0; y < n; ++y) {
        if (y == x || !used[y]) continue;
        used[y] = false;
        ans = min(ans, dfs(y, used) + dist[y][x]);
        used[y] = true;
    }
    return DP[{x, used}] = ans;
}
```

used 的範圍

- $n \leq 15$
- 把true當成1，false當成0
- 整個 used 可以編碼成 $0 \sim 2^n - 1$ 的正整數
- $2^n \leq 2^{15} = 32768$ 不需要用map存！

```
vector<bool> used(n, true);
```

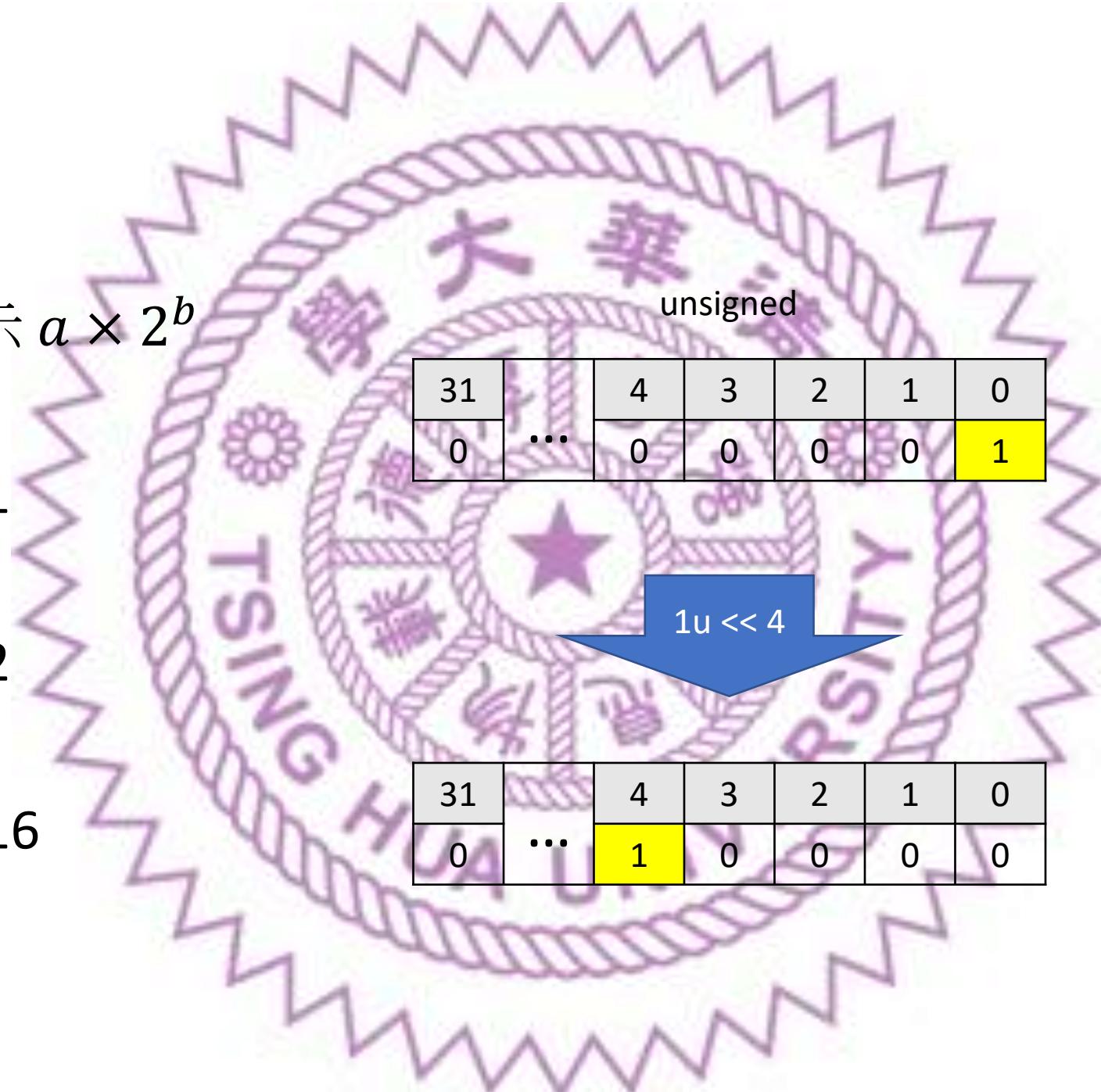
Shift

- $a \ll b$ 表示 $a \times 2^b$

- $1u \ll 0 = 1$

- $1u \ll 1 = 2$

- $1u \ll 4 = 16$



The logo of Tsinghua University is faintly visible in the background of the slide, featuring a central five-pointed star surrounded by the university's name in English and Chinese.

31	...	4	3	2	1	0
0	...	0	0	0	0	1

$1u \ll 4$

31	...	4	3	2	1	0
0	...	1	0	0	0	0

陣列操作 → 位元操作

- `vector<bool> used(n, true);` → `unsigned used = (1u << n) - 1;`
- `if(used[y] == true)` → `if (used & (1u << y) != 0)`
- `if(used[y] == false)` → `if (used & (1u << y) == 0)`
- `used[y] = !used[y]` → `used ^= (1u << y)`

狀態壓縮 DP (位元 DP)

```
const int MAXN = 15;
int n; // 點的編號為 0 ~ n-1
int dist[MAXN][MAXN];

int DP[MAXN][1u << MAXN];
void dfs(int x);

int solve() {
    return dfs(0, (1u << n) - 1);
}
```

```
int dfs(int x, unsigned used) {
    if (used == 0) {
        if (x == 0) return 0;
        return 0x3f3f3f3f;
    }
    if (DP[x][used]) return DP[x][used];
    int ans = 0x3f3f3f3f;
    for (int y = 0; y < n; ++y) {
        if (y == x || (used & (1u << y)) == 0)
            continue;
        used ^= (1u << y);
        ans = min(ans, dfs(y, used) + dist[y][x]);
        used ^= (1u << y);
    }
    return DP[x][used] = ans;
}
```

時間複雜度

- 狀態數量 $n \times 2^n$

```
int DP[MAXN][1u << MAXN];
```

- 計算每個狀態的時間 $O(n)$

```
for (int y = 0; y < n; ++y)
```

- $O(n^2 2^n)$



迴圈版本

```
int solve() {
    for (unsigned U = 0; U < (1u << n); ++U) {
        for (int x = 0; x < n; ++x) {
            if (U == 0) {
                if (x == 0) DP[x][U] = 0;
                else DP[x][U] = 0x3f3f3f3f;
                continue;
            }
            int ans = 0x3f3f3f3f;
            for (int y = 0; y < n; ++y) {
                if (y == x || (U & (1u << y)) == 0)
                    continue;
                U ^= (1u << y);
                ans = min(ans, DP[y][U] + dist[y][x]);
                U ^= (1u << y);
            }
            DP[x][U] = ans;
        }
    }
    return DP[0][(1u << n) - 1];
}
```