自主學習成果報告

深究動態規劃(DP)演算法

新竹女中 **212** 班

黃惟 **Yui Huang**

個人網站：https://YuiHuang.com/

學習成果：

日期	檢定 / 競賽	成績
10/17/2020	2020 年國際英文組青年程式設計競賽 (ISSC 2020)	佳作 (團隊) Solved: 9 out of 10
11/4/2020	109 學年度高中資訊學科能力競賽台灣省複賽(北二區)	佳作 (個人)
11/21/2020	NPSC2020 第二十二屆網際網路程式設計全國大賽(初賽)	晉級決賽 (自己一隊)
12/5/2020	NPSC2020 第二十二屆網際網路程式設計全國大賽(決賽)	第 11 名 (自己一隊)
12/12/2020	109 學年度全國資訊學科能力競賽模擬賽	第 27 名 (個人)
12/16/2020	2020 NCTU CPTC	第 16 名 (團隊)

學習綱要：

章節	自學主題	頁數
1	動態規劃演算法(Dynamic Programming, DP)介紹	3
2	動態規劃演算法與遞迴(Recursion)關係	4
3	動態規劃演算法與貪心演算法(Greedy)	6
4	動態規劃演算法的基本結構	10
5	動態規劃演算法的實作方式(1) – Bottom-up 填表實作	11
6	動態規劃演算法的實作方式(2) – Top-down遞迴實作	14
7	動態規劃演算法的應用(1) – 計算排列組合數量	17
8	動態規劃演算法的應用(2) – 子陣列和的最大值	20
9	動態規劃演算法的應用(3) – 最大子矩陣和	21
10	動態規劃演算法的應用(4) – 找最佳解(最大值、最小值)	23
11	動態規劃演算法的應用(5) – 背包問題	26
12	動態規劃演算法的應用(6) – 最長共同子序列	29
13	動態規劃演算法的應用(7) – 最長遞增子序列	32
14	動態規劃演算法的應用(8) – 編輯距離	35
15	動態規劃演算法的應用(9) – 最短路徑	37
16	動態規劃演算法的應用(10) – 位元 DP (狀態壓縮)	40
17	動態規劃演算法的應用(11) – 數位 DP (計數用)	42
18	動態規劃演算法的應用(12) – 區間 DP	44

前備知識：

1 C++ 語法、STL

2 基礎資料結構：

- 2.1 佇列 (queues)【筆記】
- 2.2 堆疊 (stacks)【筆記】
- 2.3 樹狀圖 (tree)，圖形 (graph)【筆記】
  - 2.3.1 BFS (Breadth First Search，廣度優先搜尋)【筆記】
  - 2.3.2 DFS (Depth First Search，深度優先搜索)【筆記】

3 基礎演算法：

- 3.1 排序 (sorting)【筆記】
- 3.2 搜尋 (searching) 【筆記】
- 3.3 貪心法則 (greedy method)【筆記】

§1. 動態規劃演算法(Dynamic Programming, DP)介紹

解題時評估每一種情況，得到的必為正解，但耗時驚人(例 3-1)。若一個問題可分成很多子問題，且子問題的解答能決定原問題的解答，把每次計算過的結果儲存起來，下次遇到重疊子問題就直接查表，便能加快速度(例 2-2)。

動態規劃演算法(Dynamic Programming, DP)便是由 Richard Bellman 提出來解決上述問題的演算法，據說原來的命名是 multistage decision process，但因為他的老闆很討厭數學理論，於是便取了一個和數學無關的名稱。

DP 是分治法(Divide and Conquer)的延伸，再加上以記憶法(memorization)的技巧，以記憶體空間為代價，縮短計算時間。

參考資料：

1. 網站：[演算法筆記- Dynamic Programming]()
2. Chapter 7 動態規劃 Dynamic Programming，培養與鍛鍊程式設計的邏輯腦(汪任捷)
3. Chapter 9 動態規劃演算法，寫程式前就該懂得演算法. (Aditya Y. Bhargava)
4. Chapter 3.5 Dynamic Programming, Competitive. Programming 3 (Steven Halim &. Felix. Halim)
5. Chapter 15 Dynamic Programming, Introduction to Algorithms, Third Edition (Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein)

§2. 動態規劃演算法與遞迴(Recursion)關係

有些問題可以透過遞迴(recursion)分割成許多更小的問題，當子問題與原問題完全相同，只有數值範圍不同，即為具有覆現性(recurrence)。

例 2-2：費氏數列(如下圖)

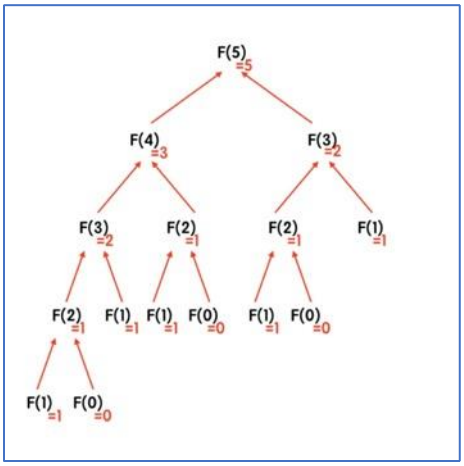
F(0) = 0

F(1) = 1

F(n) = F(n-1) + F(n-2), n > 1

解法 1：遞迴(recursion)，時間複雜度為 O(2^n)
過程中會產生很多重複的計算，當 n > 12 時，多數的電腦都得算上老半天。

解法 2：迭代(iterative)，時間複雜度為 O(n)
F(n)只跟 F(n-1)和 F(n-2)兩個值有關，把計算過的 F(i)都記憶(memorization)下來，就可以避免重複的計算，降低時間複雜度。



【例題】ZeroJudge d212: 東東爬階梯
【網址】https://yuihuang.com/zj-d212/

解法	程式碼	執行結果
遞迴	#include <iostream> using namespace std;	TLE

	long long f(int x) { if (x == 0 || x == 1) return 1; else return f(x-1) + f(x-2); } int main() { int n; while (cin >> n) { cout << f(n) << endl; } }	
迭代	#include <iostream> using namespace std; int main() { int n; long long f[100] = {0}; f[0] = 1;	AC

```
    f[1] = 1;
    for (int i=2; i<100; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    while (cin >> n) {
        cout << f[n] << endl;
    }
}
```

§3. 動態規劃演算法與貪心演算法(Greedy)

有些問題，使用貪心演算法得不到最佳解(WA, wrong answer)，使用暴力搜索(Complete Search)又會超出時間限制(TLE, time limit exceeded)，這時，DP 可能就成了最佳解方。

例 3-1：有一名小偷進入商店，看到三樣商品，重量和價值如下表。如果小偷的背包只能負重 4 磅，請問他能偷走的商品價值最高為何？

商品	重量	價值
電腦	4 磅	$30,000
筆電	3 磅	$20,000
吉他	1 磅	$15,000

解法 1：貪心 (Greedy)

從價值最高又不超出負重限制的商品開始拿，小偷帶走電腦，價值$30,000。

解法 2：暴力搜索 (Complete Search)

三樣商品都可以選擇拿或不拿，總共會有 2^3 = 8 種情況要考慮。最佳解為小偷帶走筆電和吉他，價值$35,000。

但是，當店裡有 32 種商品時，便有 2^32 = 40 億種組合要考慮，時間花費驚人。

電腦	筆電	吉他	總重	價值
X	X	X	0	0
X	X	V	1	15000
X	V	X	3	20000
X	**V**	**V**	**4**	**35000**
V	X	X	4	30000

V	X	V	5	45000
V	V	X	7	50000
V	V	V	8	65000

解法 3：動態規劃 (DP)

將在後續章節"動態規劃演算法的應用(5) – 背包問題"中探討。這裏先用 一個找零錢問題來比較 Greedy 和 DP 的不同 。

【網址】https://yuihuang.com/change-coins/

【目的】用最少數量的硬幣完成找零的動作。依照該組硬幣面額的設計，有些題目可以利用 Greedy 方式直接求解，有些則不行。

- 比較「每個硬幣的面額」與「其次小硬幣的面額」
- 【情況 1】硬幣有：$50, $10, $5, $1 四種面額，
    - $50 >= 2 * $10，可以貪心。
- 【情況 2】硬幣有：$10, $8, $1 三種面額，
    - $10 < 2 * $8，不能貪心。
    - 改用 DP 考慮全部條件，因為有些情況下，每次採取當下的最佳解，不一定是全面的最佳解。
    - 【例子】$16 = $8 + $8 = $10 + $1 + $1 + $1 + $1 + $1 + $1

【例題 1】LeetCode 860. Lemonade Change

- 硬幣有：$20, $10, $5 三種面額，且 $20 >= 2 * $10，$10 >= 2 * $5，可以貪心。
- 盡可能從面額較大的硬幣開始使用。

程式碼	結果
class Solution { public: bool lemonadeChange(vector<int>& bills) {	AC

```
int coins[] = {0, 0, 0}; //$20, $10, $5
for (int i=0; i<bills.size(); i++){
    if (bills[i] == 20){
        coins[0]++;
        if (coins[1] > 0 && coins[2] > 0){
            coins[1]--;
            coins[2]--;
        } else if (coins[2] >= 3){
            coins[2] -= 3;
        } else return false;
    } else if (bills[i] == 10){
        coins[1]++;
        if (coins[2] > 0){
            coins[2]--;
        } else return false;
    } else {
        coins[2]++;
    }
}
```

`return true;` `}` `};`	

【例題 2】LeetCode 322. Coin Change

- 題目不保證每一組硬幣的面額可以滿足貪心成立的條件，改用 DP 處理。
- dp[x]：紀錄目前要湊成金額 x 的最少硬幣數目。
- 測試每個硬幣面額 coins[i]
- 【狀態轉移方程】當採用一個新的硬幣面額可以使用更少的硬幣數目來湊成金額 j 時，更新 dp[j] = min(dp[j], dp[j - coins[i]] + 1);

程式碼	結果
`class Solution {` `public:` `int coinChange(vector<int>& coins, int amount) {` `int dp[amount+1];` `memset(dp, 0x3F, sizeof(dp));` `dp[0] = 0;` `for (int i=0; i<coins.size(); i++){` `for (int j=coins[i]; j<=amount; j++){` `dp[j] = min(dp[j], dp[j-coins[i]]+1);` `}` `}`	AC

```
        if (dp[amount] < 0x3F3F3F3F) return dp[amount];

        else return -1;

    }

};
```

§4. 動態規劃演算法的基本結構

Dynamic Programming = Divide and Conquer + Memorization

DP 是分治法(Divide and Conquer)的延伸，再加上以記憶法(memorization)的技巧，以記憶體空間為代價，縮短計算時間。

使用 DP 需要符合的條件以及運作過程如下(註 4-1)：

1. 把原問題遞迴分割成許多更小的問題。(recurrence)
   - 1-1. 子問題與原問題的求解方式皆類似。(optimal sub-structure)
   - 1-2. 子問題會一而再、再而三的出現。(overlapping sub-problems)
2. 設計計算過程：
   - 2-1. 確認每個問題需要哪些子問題來計算答案。(recurrence)
   - 2-2. 確認總共有哪些問題。(state space)
   - 2-3. 把問題一一對應到表格。(lookup table)
   - 2-4. 決定問題的計算順序。(computational sequence)
   - 2-5. 確認初始值、計算範圍。(initial states / boundary)
3. 實作，主要有兩種方式：
   - 3-1. Bottom-up
   - 3-2. Top-down

(註 4-1) 資料出處：http://web.ntnu.edu.tw/~algo/DynamicProgramming.html

§5. 動態規劃演算法的實作方式(1) – Bottom-up 填表實作

Bottom-up 填表實作的方式需先建立表格，然後由最小的問題開始計算，反覆地讀取數據、計算數據、儲存數據。需仔細考慮設計狀態轉移方程，逐步更新整個表格的資料，與 Top-down 作法相比，通常速度較慢。

【筆記】DP：Top-down vs. Bottom-up
【網址】https://yuihuang.com/dp-top-down-vs-bottom-up/

【例題】UVA 11450 Wedding shopping
【網址】https://yuihuang.com/uva-11450/

解法	程式碼
Bottom-up	#include <iostream> #include <cstring> using namespace std; int main() { ios_base::sync_with_stdio(0); cin.tie(0); int T, M, C; cin >> T; while (T--){ cin >> M >> C; int price[25][25] = {0};

```
for (int i = 0; i < C; i++){
    cin >> price[i][0];
    for (int j = 1; j <= price[i][0]; j++){
        cin >> price[i][j];
    }
}
bool dp[2][M]; //check whether a state is reachable
memset(dp, false, sizeof(dp));
for (int i = 1; i <= price[0][0]; i++){
    if (M - price[0][i] >= 0){
        dp[0][M - price[0][i]] = true;
    }
}
int cur = 1, pre = 0;
for (int g = 1; g < C; g++){
    for (int money = 0; money < M; money++){
        dp[cur][money] = false;
    }
    for (int money = 0; money < M; money++){
```

```
            if (dp[pre][money]){
                for (int i = 1; i <= price[g][0]; i++){
                    if (money - price[g][i] >= 0){
                        dp[cur][money - price[g][i]] = true;
                    }
                }
            }
        }
        swap(pre, cur);
    }
    int ans = -1;
    for (int i = 0; i < M; i++){
        if (dp[pre][i]){
            ans = i;
            break;
        }
    }
    if (ans >= 0) cout << M - ans << "\n";
    else cout << "no solution\n";
```

	`` } return 0; } ```
Top-down	``` #include <iostream> #include <cstring> using namespace std; int T, M, C; int dp[205][25]; int price[25][25]; int shop(int money, int g){ if (money < 0) return -1e9; if (g == C) return M - money; int &ans = dp[money][g]; if (ans != -1) return ans; for (int i = 1; i <= price[g][0]; i++){ ans = max(ans, shop(money - price[g][i], g+1)); } return ans; ```

```
}


int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cin >> T;
    while (T--){
        cin >> M >> C;
        for (int i = 0; i < C; i++){
            cin >> price[i][0]; // K
            for (int j = 1; j <= price[i][0]; j++){
                cin >> price[i][j];
            }
        }
        memset(dp, -1, sizeof(dp));
        int ans = shop(M, 0);
        if (ans < 0) cout << "no solution\n";
        else cout << ans << "\n";
    }
```

	`` return 0; } ```

§6. 動態規劃演算法的實作方式(2) – Top-down 遞迴實作

Top-down 的方式採遞迴實作，通常程式碼較為簡潔，可讀性也比較好。有時遇到 Nim Game 題目類型，如果一下子想不出好的解法，直接使用 DP 又會 MLE，也可利用 DP 找出規律後，再用公式解題。

【例題】Codeforces 1194D. 1-2-K Game
【網址】https://yuihuang.com/cf-1194d/

本題的數據範圍很大，$0 \le n \le 1e9, 3 \le k \le 1e9$，用 top-down DP 會 MLE。可以先利用小數據 DP 找出規律，再用更快的方法實作。

目的	程式碼
小數據 DP 找出規律	#include <iostream> #include <cstring> using namespace std; int T, n, k; int dp[25]; int solve(int x){ if (x < 0) return 0; if (dp[x] != 0) return dp[x]; int mn = 0; mn = min(mn, solve(x-1)); mn = min(mn, solve(x-2));

```
    mn = min(mn, solve(x-k));
    if (mn == -1) return dp[x] = 1;
    else return dp[x] = -1;
}


int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);


    cin >> T;
    while (T--){
        cin >> n >> k;
        memset(dp, 0, sizeof(dp));
        solve(n);
        for (int i=0; i<=n; i++){
            cout << dp[i] << ' ';
        }
        cout << '\n';
    }
```

	return 0; }

- 觀察小數據執行的結果（下圖），
- 當 k 非 3 的倍數時，只要 n 是 3 的倍數，先手輸。
- 當 k 是 3 的倍數時，
  - (I) 如果 n 是 (k + 1) 的倍數（n % (k + 1) = 0），先手輸。
  - (II) n % (k + 1) 不等於 k，但是 3 的倍數，先手輸。
  - 結合 (I) (II)， (n % (k + 1)) % 3 == 0 && n % (k + 1) != k，先手輸。

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K=3	DP	-1	1	1	1	-1	1	1	1	-1	1	1	1	-1	1	1	1	-1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K=4	DP	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K=5	DP	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K=6	DP	-1	1	1	-1	1	1	1	-1	1	1	-1	1	1	1	-1	1	1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K=7	DP	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
K=8	DP	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1	1	1	-1

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K=9	DP	-1	1	1	-1	1	1	-1	1	1	1	-1	1	1	-1	1	1	-1	1	1	1	-1

目的	程式碼
公式解題	#include <iostream>

```
#include <bitset>

using namespace std;



int t, n, k;



int main() {

    ios_base::sync_with_stdio(0);

    cin.tie(0);

    cin >> t;

    while (t--){

        cin >> n >> k;

        if (k % 3 == 0){

            n %= (k+1);

            if (n % 3 == 0 && n != k) cout << "Bob\n";

            else cout << "Alice\n";

        }

        else {

            if (n % 3 == 0) cout << "Bob\n";

            else cout << "Alice\n";
```

	`` } } } ```

§7. 動態規劃演算法的應用(1) – 計算排列組合數量

DP 可以用來計算組合數量，比如可能的路徑組合或可能的硬幣兌換方式。此類問題如果用陽春的遞迴作法，通常會超時(TLE, Time Limit Exceeded)，DP 可以避免重複的計算，達到加快速度的目的。有些題目還可以更聰明的作法減少記憶體用量。

【例題】ZeroJudge d212: 東東爬階梯
【網址】https://yuihuang.com/zj-d212/
【說明】題目限制爬階梯有三種走法：（1）第一步走一階，第二步走二階。（2）第一步走二階，第二步走一階。（3）全程都走一階。
【解題想法】假設爬 x 段階梯有 f(x) 種走法，我們可以先確定邊界條件：

f(0) = 1

f(1) = 1

當 x > 1，f(x) = f(x-1) + f(x-2)，因為不管前一步怎麼走，這一步都只能走一階或二階。

作法	程式碼	結果
遞迴	long long f(int x) { if (x == 0 || x == 1) return 1; else return f(x-1) + f(x-2); } int main() { int n; while (cin >> n) { cout << f(n) << endl;	TLE

	`    }` `    return 0;` `}`	
迭代	`int main() {` `    int n;` `    long long f[100] = {0};` `    f[0] = 1;` `    f[1] = 1;` `    for (int i=2; i<100; i++) {` `        f[i] = f[i-1] + f[i-2];` `    }` `    while (cin >> n) {` `        cout << f[n] << endl;` `    }` `    return 0;` `}`	AC
更節省記憶體	`int main() {` `    int n;`	AC

	`` while (cin >> n){ long long a = 1, b = 1; for (int i = 0; i < n; i++){ b = a + b; a = b - a; } cout << a << "\n"; } } ```	

【例題】ZeroJudge d253: 00674 – Coin Change
【網址】https://yuihuang.com/zj-d253/
【說明】把一個金額兌換成硬幣，硬幣面額有 1、5、10、25、50 五種，計算共有多少種硬幣組合？
【解題想法】枚舉每一種硬幣面額，計算能排列出該金額的硬幣組合數量。以下分別嚐試 top-down 及 bottom-up DP 來求解。

作法	程式碼	結果
Top-down	int coins[] = {1, 5, 10, 25, 50}; int dp[5][7500]; int ways(int idx, int money){ if (idx == 5 || money < 0) return 0;	AC

```
    if (money == 0) return 1;
    if (dp[idx][money] != -1) return dp[idx][money];
    int ret = 0;
    for (int i = idx; i < 5; i++){
        if (money >= coins[i]){
            ret += ways(i, money - coins[i]);
        }
    }
    return dp[idx][money] = ret;
}


int main() {
    int n;
    memset(dp, -1, sizeof(dp));
    while (cin >> n){
        cout << ways(0, n) << "\n";
    }
    return 0;
}
```

Bottom-up

```
const int maxn = 7500;

int N = 5; // types of coins

int coin[] = {1, 5, 10, 25, 50};

long long dp[maxn];



int main() {

    int n;

    memset(dp, 0, sizeof(dp));

    dp[0] = 1;

    for (int i=0; i<N; i++) {

        for (int j=coin[i]; j<maxn; j++) {

            dp[j] = dp[j] + dp[j-coin[i]];

        }

    }



    while (cin >> n) {

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

    }

    return 0;
```

AC

	}	

§8. 動態規劃演算法的應用(2) – 子陣列和的最大值

用 DP 的觀念來找出一個一維陣列中，子陣列（在原陣列中一串連續的元素）的數值和的最大值，通常在書上被歸類成 Max 1D Range Sum 求解問題。實作時無須建表。

【例題】ZeroJudge a540: 10684 – The jackpot
【網址】https://yuihuang.com/zj-a540/
【說明】找出子陣列和的最大值(Max 1D Range Sum)
【解題想法】讀入每一筆測資時順道加計總和（子陣列和），如果目前總和（sum）大於 mx（目前為止的子陣列和的最大值），更新 mx。如果目前總和小於零，則表示納入延續這個子陣列已經徒勞無功，於是將 sum 清零，重新開始一個新的子陣列。

程式碼	結果
(see code below)	AC

```
#include <iostream>
#include <cstring>
using namespace std;
int n, tmp;
int main() {
    while (cin >> n) {
        if (n == 0) break;
        int sum = 0;
        int mx = 0;
        for (int i=0; i<n; i++) {
            cin >> tmp;
            sum += tmp;
            if (sum > mx) mx = sum;
            if (sum < 0) sum = 0;
        }
        if (mx <= 0) cout << "Losing streak.\n";
        else cout << "The maximum winning streak is " << mx << ".\n";
    }
    return 0;
}
```

§9. 動態規劃演算法的應用(3) – 最大子矩陣和

用 DP 來找出一個平面（二維陣列）中，任意矩形區塊（子矩陣）的數值和的最大值，通常在書上被歸類成 Max 2D Range Sum 求解問題。實作時搭配排容原理，枚舉所有子區域 (sub-rectangle)，避免枚舉四個角落座標，進而降低時間複雜度。

【例題】ZeroJudge d206: 00108 – Maximum Sum
【網址】https://yuihuang.com/zj-d206/
【說明】給一個 N*N 的陣列，找出有最大和的子區域 (sub-rectangle)，其和為多少？
【方法 1】枚舉所有可能的子區域 (sub-rectangle)，左上角座標 (i, j)，右下角座標 (k, l)。
【方法 2】利用前綴作法進一步優化

作法	程式碼	結果
枚舉	<pre>int a[101][101]; int main() { int N; while (cin >> N) { for (int i=0; i<N; i++) { for (int j=0; j<N; j++) { cin >> a[i][j]; if (i > 0) a[i][j] += a[i-1][j]; if (j > 0) a[i][j] += a[i][j-1]; if (i > 0 && j > 0) a[i][j] -= a[i-1][j-1]; }</pre>	AC

<table>
<tr><td></td><td><pre>
        }
        int maxSum = -127*100*100;
        int subRect = 0;
        for (int i=0; i<N; i++)
            for (int j=0; j<N; j++)
                for (int k=i; k<N; k++)
                    for (int l=j; l<N; l++) {
                        subRect = a[k][l];
                        if (i > 0) subRect -= a[i-1][l];
                        if (j > 0) subRect -= a[k][j-1];
                        if (i > 0 && j > 0) subRect += a[i-1][j-1];
                        maxSum = max(maxSum, subRect);
                    }
        cout << maxSum << endl;
    }
    return 0;
}
</pre></td><td></td></tr>
<tr><td>前綴</td><td><pre>
long long n, a[105][105], ans, total;
int main() {
</pre></td><td>AC</td></tr>
</table>

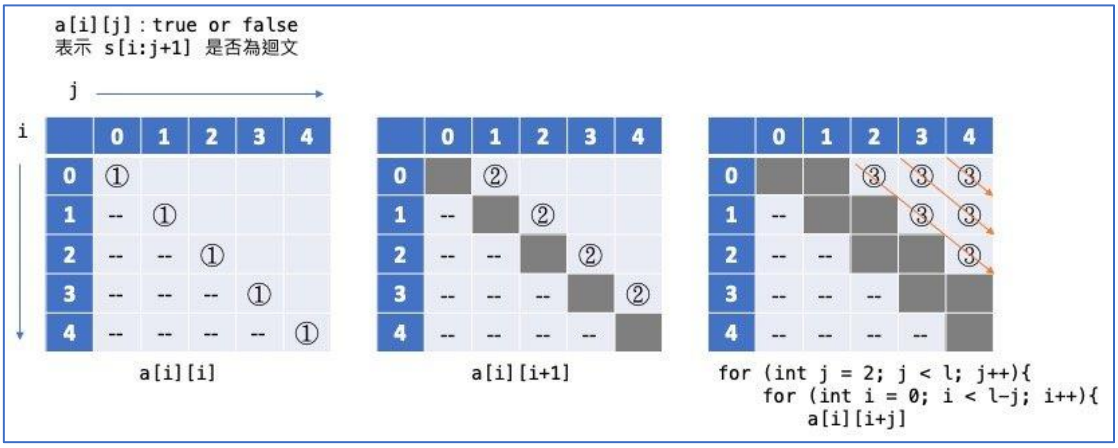
```
while (cin >> n){
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= n; j++){
            cin >> a[i][j];
            a[i][j] += a[i][j-1];
        }
    }
    ans = 0;
    for (int i = 0; i < n; i++){
        for (int j = i+1; j <= n; j++){
            total = 0;
            for (int k = 1; k <= n; k++){
                total += a[k][j]-a[k][i];
                ans = max(ans, total);
                if (total < 0) total = 0;
            }
        }
    }
    cout << ans << "\n";
```

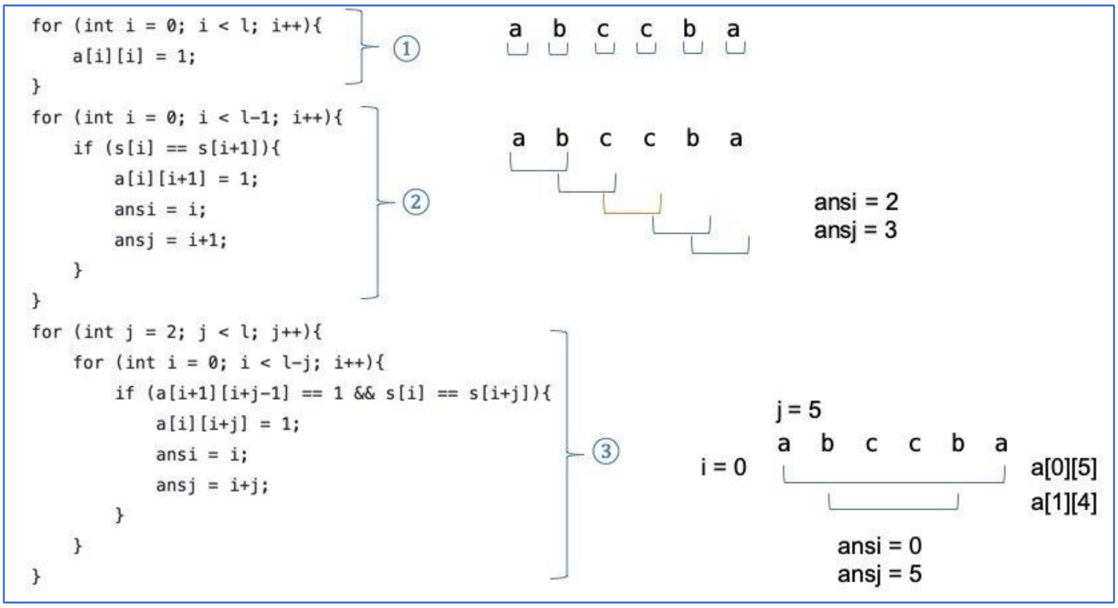
```
    }
}
```

§10. 動態規劃演算法的應用(4) – 找最佳解(最大值、最小值)

用 DP 的方法來找出一個題目的最佳解，通常是用來求最大值或最小值，或一個最佳路徑。比之暴力遍歷所有可能性，速度會快上許多。

【例題】LeetCode 5. Longest Palindromic Substring
【網址】https://yuihuang.com/lc-5/
【說明】找出一字串中最長的迴文子字串
【方法 1】DP，複雜度 $O(N^2)$





【方法 2】除了 DP 之外，這一道題目還可以使用更進階的 Manacher's Algorithm，把複雜度降為 O(N)。因超出本文主題，說明在此略過。

<table>
<tr><th>解法</th><th>程式碼</th><th>時間</th></tr>
<tr><td>DP</td><td><pre><code>class Solution {
public:
    string longestPalindrome(string s) {
        int ansi = 0, ansj = 0;
        int l = s.length();
        if (l == 0) return "";
        int a[l][l];
        memset(a, 0, sizeof(a));
        for (int i = 0; i < l; i++){
            a[i][i] = 1;
        }
        for (int i = 0; i < l-1; i++){
            if (s[i] == s[i+1]){
                a[i][i+1] = 1;
                ansi = i;
                ansj = i+1;
            }
        }</code></pre></td><td>88ms</td></tr>
</table>

	for (int j = 2; j < l; j++){ for (int i = 0; i < l-j; i++){ if (a[i+1][i+j-1] == 1 && s[i] == s[i+j]){ a[i][i+j] = 1; ansi = i; ansj = i+j; } } } return s.substr(ansi, ansj-ansi+1); } };	
Manacher's Algorithm	class Solution { public: string longestPalindrome(string s) { int l = s.length(), dp[l*2+5], mxr, mxp, idx = 0; string ans = ""; memset(dp, 0, sizeof(dp)); char ss[l*2+5];	4ms

```
ss[0] = '#';
ss[1] = '#';
for (int i = 0; i < l; i++){
    ss[i*2+2] = s[i];
    ss[i*2+3] = '#';
}
l *= 2;
mxr = 0;
mxp = 0;
for (int i = 1; i <= l; i++){
    if (i <= mxr){
        dp[i] = min(dp[mxp*2-i], mxr-i);
    }
    if (i + dp[i] >= mxr){
        while (ss[i+dp[i]+1] == ss[i-dp[i]-1]) dp[i]++;
    }
    if (i+dp[i] > mxr){
        mxr = i+dp[i];
        mxp = i;
```

```
            }
            if (dp[i] > dp[idx]) idx = i;
        }
        for (int i = idx-dp[idx]; i <= idx+dp[idx]; i++){
            if (ss[i] != '#'){
                ans += ss[i];
            }
        }
        return ans;
    }
};
```

§11. 動態規劃演算法的應用(5) – 背包問題

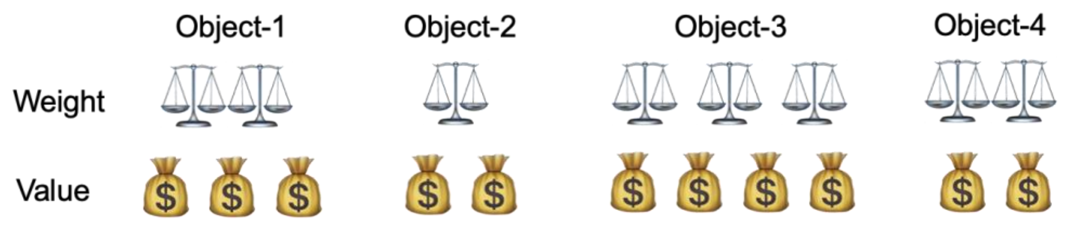
0-1 背包問題是最經典的 DP 應用。在物品無法分割的情況下，一個承重有限的背包，最多能裝入多大價值的物品？暴力法一定會超時；貪心法在多數條件下無法求得正解。

【筆記】DP: 0-1 Knapsack (0-1 背包問題)
【網址】https://yuihuang.com/0-1-knapsack/

【範例】承重有限的背包，最多能裝入多大價值的物品？

- 每種物品只有一個且不可分割，只能選擇拿或不拿。每種物品的價值為 v，重量為 w。
- 在背包負重有限的情況下，求背包能夠容納的物品的最大價值。

- **暴力枚舉法**：有 N 種物品，每一種都可以選擇拿或不拿，總共有 2^N 種可能性要考慮。N = 20 時，就有超過一百萬種組合要考慮。



	Object-1	Object-2	Object-3	Object-4	Total W	Total V
選擇-1	0	0	0	0	0	0
選擇-2	0	0	0	1	2	2
選擇-3	0	0	1	0	3	4
選擇-4	0	0	1	1	5	6
選擇-5	0	1	0	0	1	2
	...	...	...	...	...	...
選擇-16	1	1	1	1	8	11

20個物品 → 2^{20} ~ 1,000,000 種選擇

- **DP (Dynamic Programming)**：建表紀錄目前位置最好的結果，一步一步地考慮狀態轉移。

  【方法 1】二維的 DP 表格

- 建立二維的 DP 表格，dp[m+1][W+1] (m 種物品，背包最大負重 W)，初始值為 0。
- dp[i+1][j]：考慮到第 i 種物品時，最大負重為 j 的背包，能夠拿取的最大價值。
- 狀態轉移方程：dp[i+1][j] = max(dp[i][j], dp[i][j – w[i]] + v[i]);

【方法 2】一維的 DP 表格

- 建立一維的 DP 表格，dp[W+1] (背包最大負重 W)，初始值為 0。
- 狀態轉移方程：dp[j] = max(dp[j], dp[j-w[i]] + v[i]);
- 注意：範例程式碼的第 15 行，為了重複利用記憶體，迴圈需逆向執行。

【方法 3】交互使用兩個一維陣列。

	Weight-0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0
Object-1	0	0	3	3	3	3	3	3	3
Object-2	0	2	3	5	5	5	5	5	5
Object-3	0	2	3	5	6	7	9	9	9
Object-4	0	2	3	5	6	7	9	9	11

【例題】ZeroJudge b131: NOIP2006 2.开心的金明

解法	程式碼	結果
方法 1	#include <iostream> #include <cstring> using namespace std;	AC

```
int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    int N, m; //N（<30000）表示总钱数，m（<25）为希望购买物品的个数
    cin >> N >> m;
    int v[m], w[m]; //v 表示该物品的价格(v<=10000)，p 表示该物品的重要度
(1~5)
    int dp[m+1][N+1];

    for (int i=0; i<m; i++){
        cin >> v[i] >> w[i];
    }
    memset(dp, 0, sizeof(dp));
    for (int i=0; i<m; i++){
        for (int j=0; j<=N; j++){
            if (j < v[i]) {
                dp[i+1][j] = dp[i][j];
```

```
            } else {
                dp[i+1][j] = max(dp[i][j], dp[i][j - v[i]] + v[i] * w[i]);
            }
        }
    }
    cout << dp[m][N] << '\n';
    return 0;
}
```

方法2

```
#include <iostream>
#include <cstring>
using namespace std;


int main(){
    int N, m;
    cin >> N >> m;
    int v[m], w[m];
    for (int i=0; i<m; i++){
        cin >> v[i] >> w[i];
    }
```

AC

	<pre> int dp[N+1]; memset(dp, 0, sizeof(dp)); for (int i=0; i<m; i++){ for (int j=N; j>=v[i]; j--){ dp[j] = max(dp[j], dp[j-v[i]] + v[i]*w[i]); } } cout << dp[N] << '\n'; return 0; }</pre>	
方法3	<pre>#include <iostream> #include <cstring> using namespace std; int main(){ int N, m; cin >> N >> m; int v[m], w[m]; for (int i=0; i<m; i++){ cin >> v[i] >> w[i]; } int dp[2][N+1]; memset(dp, 0, sizeof(dp)); int idx = 0; for (int i=0; i<m; i++){ for (int j=0; j<=N; j++){ if (j < v[i]) { dp[idx^1][j] = dp[idx][j];</pre>	AC

```
            } else {
                dp[idx^1][j] = max(dp[idx][j], dp[idx][j
- v[i]] + v[i] * w[i]);
            }
        }
        idx ^= 1;
    }
    cout << dp[idx][N] << '\n';
    return 0;
}
```

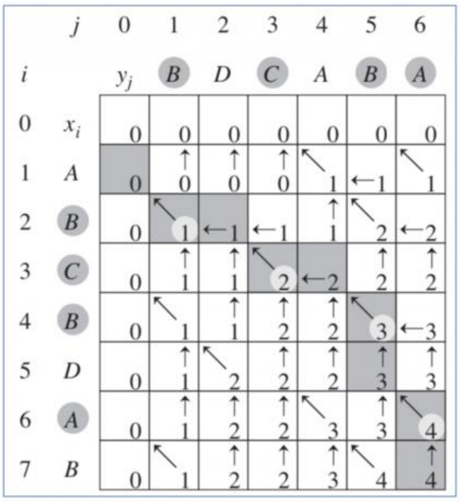
§12. 動態規劃演算法的應用(6) – 最長共同子序列

比對兩個字串，找出它們的最長共同字序列（不需是連續字元，也就是說，非子字串），是一個 DP 的基礎應用。建表 dp[i][j]紀錄 字串 s1 的前 i 個字元，與 s2 的前 j 個字元，共同子序列的長度。

【筆記】DP：LCS 最長共同子序列
【網址】https://yuihuang.com/dp-lcs/

下圖取自"Introduction to Algorithms"的 Chapter 15.4，說明利用 DP 求解時，狀態轉移的寫法。

```
LCS-LENGTH(X, Y)
    m = X.length
    n = Y.length
    let b[1..m, 1..n] and c[0..m, 0..n] be new tables
    for i = 1 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x_i == y_j
                c[i, j] = c[i − 1, j − 1] + 1
                b[i, j] = "↖"
            elseif c[i − 1, j] ≥ c[i, j − 1]
                c[i, j] = c[i − 1, j]
                b[i, j] = "↑"
            else c[i, j] = c[i, j − 1]
                b[i, j] = "←"
    return c and b
```



Source: Introduction to Algorithms (Chapter 15.4)

- 兩個字串為 s1, s2

- dp[i][j]：s1 的前 i 個字元（s1[0] ~ s1[i-1]），與 s2 的前 j 個字元（s2[0] ~ s2[j-1]），共同子序列的長度。
- 【例】s1 = “abcdgh”，s2 = “aedfhr”，dp[4][3] = 2 (“abcd” 與 “aed” 的共同子序列為 “ad”)
- 更新 dp[i][j] 時，先比較 s1[i-1] 與 s2[j-1]，
    - 如果兩個字元相同，則 dp[i][j] = dp[i-1][j-1] **+ 1**;
    - 如果兩個字元不同，則 dp[i][j] = **max**(dp[i][j-1], dp[i-1][j]);

【例題】ZeroJudge c001: 10405 – Longest Common Subsequence
【網址】https://yuihuang.com/zj-c001/

程式碼

```
const int maxn=1005;

string s1, s2;

int dp[maxn][maxn];

int main() {

    int l1, l2;

    while (cin >> s1 >> s2) {

        l1 = (int)s1.length();

        l2 = (int)s2.length();

        memset(dp, 0, sizeof(dp));

        for (int i=1; i<=l1; i++) {

            for (int j=1; j<=l2; j++) {

                if (s1[i-1] == s2[j-1]) {

                    dp[i][j] = dp[i-1][j-1] + 1;

                } else {

                    dp[i][j] = max(dp[i][j-1], dp[i-1][j]);

                }

            }
```

```
        }

        cout << dp[l1][l2] << endl;

    }

}
```

§13. 動態規劃演算法的應用(7) – 最長遞增子序列

DP 的另一個經典應用為，從一連串的整數序列中選出最長的嚴格遞增子序列。關鍵技巧是用一個 vector v[i]紀錄可位居第 i 順位的最小值 ，讓後續數字有機會發展出更長的遞增子序列。

【筆記】DP：LIS 最長遞增子序列
【網址】https://yuihuang.com/dp-lis/

【例題】ZeroJudge d242: 00481 – What Goes Up
【網址】https://yuihuang.com/zj-d242/
【說明】

- a[ ]：讀入測資
- dp[idx]：掃描測資至 index-i 時的最長遞增子序列的長度
- v[i]：紀錄可位居第 i 順位的最小值 (讓後續數字有機會發展出更長的遞增子序列)。
- 【例子】a[ ] = {2, 1, 4, 3, 6, 7, 5}
    - i = 0, a[0] = 2, dp[0] = 1, v = {2}
    - i = 1, a[1] = 1 比 v.back( ) 小。找出 lower_bound(v.begin(), v.end(), a[1]) 的位置，將其置換為 a[1]。dp[1] = 1, v = {1}。
    - i = 2, a[2] = 4 比 v.back( ) 大，直接加進 v 的尾部。v = {1, 4}，dp[2] = 2 (此時最長遞增子序列的長度為 2)。
    - i = 3, a[3] = 3 比 v.back( ) 小。找出 lower_bound(v.begin(), v.end(), a[3]) 的位置，將其置換為 a[3]。v = {1, 3}, dp[3] = 2。
    - i = 4, a[4] = 6 比 v.back( ) 大，直接加進 v 的尾部。v = {1, 3, 6}，dp[4] = 3 (此時最長遞增子序列的長度為 3)。
    - i = 5, a[5] = 7 v.back( ) 大，直接加進 v 的尾部。v = {1, 3, 6, 7}，dp[5] = 4 (此時最長遞增子序列的長度為 4)。
    - i = 6, a[6] = 5 比 v.back( ) 小。找出 lower_bound(v.begin(), v.end(), a[6]) 的位置，將其置換為 a[6]。v = {1, 3, 5, 7}, dp[6] = 3。
    - 最後 dp[ ] = {1, **1**, 2, **2**, **3**, **4**, 3}，最長遞增子序列的長度 = max(dp) = 4。

- 由 dp，從後往前找出最長遞增子序列的值。
    - dp[5] = 4, ans = {7}
    - dp[4] = 3, ans = {7, 6}
    - dp[3] = 2, ans = {7, 6, 3}
    - dp[1] = 1, ans = {7, 6, 3, 1}
- 逆序印出 ans，{1, 3, 6, 7} 即為最長遞增子序列。

程式碼

```
int a[600000], num, idx, tmp, n[600000];

vector <int> v;

vector <int> ans;



int main() {

    cin >> num;

    v.push_back(num);

    n[0] = num;

    a[0] = 1;

    idx++;

    while (cin >> num){

        n[idx] = num;

        if (num > v[v.size()-1]){

            v.push_back(num);
```

```
        a[idx] = v.size();
    }
    else {
        *lower_bound(v.begin(), v.end(), num) = num;
        a[idx] = lower_bound(v.begin(), v.end(), num) - v.begin() + 1;
    }
    idx++;
}
tmp = v.size();
cout << tmp << "\n-\n";
for (int i = idx-1; i >= 0; i--){
    if (a[i] == tmp){
        ans.push_back(n[i]);
        tmp--;
    }
}
for (int i = ans.size()-1; i >= 0; i--){
    cout << ans[i] << "\n";
}
```

```
}
```

§14. 動態規劃演算法的應用(8) – 編輯距離

DP 也可以用來求得兩個字串間的最短編輯距離(Edit Distance)，亦即給定可執行之編輯動作(如插入字元、刪除字元、編輯字元)，讓兩個字串變成相同的最少動作數。

【例題】ZeroJudge e828: 3.猴子打字遊戲 (Typing)
【網址】https://yuihuang.com/zj-e828/

程式碼

```
#include <iostream>

#include <cstring>

using namespace std;

string s1, s2;



int editDistance(){

    int len1 = (int) s1.length();

    int len2 = (int) s2.length();



    int dp[2][len1 + 1];

    memset(dp, 0, sizeof dp);



    //s2 為空字串

    for (int i = 0; i <= len1; i++)
```

```
        dp[0][i] = i * 2; //add

    for (int i = 1; i <= len2; i++) {
        for (int j = 0; j <= len1; j++) {
            if (j == 0) //s1 為空字串
                dp[i % 2][j] = i * 2; //delete
            else if (s1[j - 1] == s2[i - 1]) {
                dp[i % 2][j] = dp[(i - 1) % 2][j - 1];
            } else {
                dp[i % 2][j] = min(dp[(i - 1) % 2][j] + 2,
                                   min(dp[i % 2][j - 1] + 2,
                                       dp[(i - 1) % 2][j - 1] + 3));
            }
        }
    }
    return dp[len2 % 2][len1];
}

int main() {
```

```
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cin >> s1;
    int mn = 1e9, idx = 0;
    for (int i=1; i<=3; i++){
        cin >> s2;
        int dis = editDistance();
        if (dis <= mn){
            mn = dis;
            idx = i;
        }
    }
    cout << idx << " " << mn << "\n";
}
```

§15. 動態規劃演算法的應用(9) – 最短路徑

基礎圖論中用來計算全點對最短路徑(All-pairs Shortest Path)的 Floyd-Warshall 演算法，也是經典的 DP 應用。

【筆記】Floyd-Warshall algorithm 全點對最短路徑
【網址】https://yuihuang.com/floyd-warshall-algorithm/

【用途】用來解決「有向圖」中，任意兩點間的最短路徑。可以正確處理有「負權」的邊。

【原理】枚舉 + DP

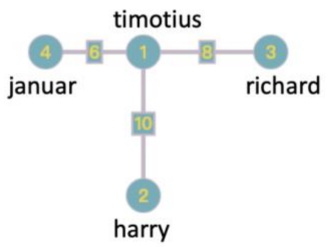
【實作】

- 枚舉所有中間點 (i)，更新所有【j –> k】的最短路徑。
- dis[j][k] = dis[j][i] + dis[i][k]

【複雜度】

- 時間複雜度：$O(N^3)$
- 空間複雜度：$O(N^2)$

【範例】[ZeroJudge d282: 11015 – 05-2 Rendezvous]



枚舉中間點 = 1

	1	2	3	4
1	0	10	8	6
2	10	0	18	16
3	8	18	0	14
4	6	16	14	0

Initial

	1	2	3	4
1	0	∞	∞	∞
2	∞	0	∞	∞
3	∞	∞	0	∞
4	∞	∞	∞	0

依序枚舉中間點 = 2, 3, 4

	1	2	3	4
1	0	10	8	6
2	10	0	18	16
3	8	18	0	14
4	6	16	14	0

Input data

	1	2	3	4
1	0	10	8	6
2	10	0	∞	∞
3	8	∞	0	∞
4	6	∞	∞	0

結果

	1	2	3	4	Sum
1	0	10	8	6	24
2	10	0	18	16	44
3	8	18	0	14	40
4	6	16	14	0	36

程式碼

```
#include <iostream>
#include <map>
#include <string>
using namespace std;


map <int, string> mp;
int n, m;
int cnt = 0;
int a[25][25];


int main() {
    while (cin >> n >> m){
        if (n == 0) break;
        cnt++;
        mp.clear();
        string str;
        for (int i = 0; i < n+1; i++){
            for (int j = 0; j < n+1; j++){
                if (i == j) a[i][j] = 0;
```

```
        else a[i][j] = 1e9;
    }
}
for (int i = 0; i < n; i++){
    cin >> str;
    mp[i+1] = str;
}
int x, y, z;
for (int i = 0; i < m; i++){
    cin >> x >> y >> z;
    a[x][y] = z;
    a[y][x] = z;
}
    // Floyd-Warshall 做法
    // i：枚舉中間點；j, k：計算 all-pairs shortest path
for (int i = 1; i < n+1; i++){
    for (int j = 1; j < n+1; j++){
        for (int k = 1; k < n+1; k++){
            if (a[j][i] + a[i][k] < a[j][k]){
```

```
                a[j][k] = a[j][i] + a[i][k];
            }
        }
    }
  }
  int mn = 1e9, idx, ans;
  for (int i = 1; i < n+1; i++){
    ans = 0;
    for (int j = 1; j < n+1; j++){
      ans += a[i][j];
    }
    if (ans < mn){
      mn = ans;
      idx = i;
    }
  }
  printf("Case #%d : %s\n", cnt, mp[idx].c_str());
 }
}
```

§16. 動態規劃演算法的應用(10) – 位元 DP (狀態壓縮)

有些題目，當正常開陣列(通常是高維度)，會導致記憶體過大時，可改用二進位制來表示狀態(狀態壓縮)，結合 DP 技巧進行解題，即稱位元 DP。

【例題】ZeroJudge d879: 10911 – Forming Quiz teams
【網址】https://yuihuang.com/zj-d879/

程式碼

```
#include <iostream>

#include <cmath>

using namespace std;



int Case, n, x[20], y[20];

string s;

double dp[1<<16];



double dist(int a, int b){

    double ret = sqrt((x[a]-x[b]) * (x[a]-x[b]) + (y[a]-y[b]) * (y[a]- y[b]));

    return ret;

}



double solve(int mask){
```

```
    if (dp[mask] > -1) return dp[mask];
    if (!mask) return 0;
    double mn = 1e18;
    for (int i = 0; i < n; i++){
        if (!((1<<i) & mask)) continue;
        for (int j = i+1; j < n; j++){
            if (!((1<<j) & mask)) continue;
            mn = min(mn, solve(mask ^ (1<<i) ^ (1<<j))+dist(i, j));
        }
        break;
    }
    return dp[mask] = mn;
}


int main() {
    while (cin >> n){
        if (n == 0) break;
        n *= 2;
        for (int i = 0; i < n; i++){
```

```
            cin >> s;
            cin >> x[i] >> y[i];
        }
        for (int i = 0; i < (1<<n); i++){
            dp[i] = -1e18;
        }
        Case++;
        printf("Case %d: %.2lf\n", Case, solve((1<<n)-1));
    }
}
```

§17. 動態規劃演算法的應用(11) – 數位 DP (計數用)

數位 DP 是一種計數用的 DP，一般是用來統計一個區間 [left, right] 內滿足一些條件的個數。所謂數位 DP，字面意思就是在數位(數字的位數)上進行 DP。

【筆記】數位 DP
【網址】https://yuihuang.com/digit-dp/

- 【用途】查找 **[0, x] 區間**內，符合條件的個數。用非常少的狀態（處理每個 digit）來得到需要的下一個狀態，避免疊代每個數字衍生的重複計算。
- 【範例】HDU 2089 不要 62【題解】
- dp[7][2][2]：
    - 第一個維度 pos：目前處理的位數。數字 0<n≤m<1000000，最多七位數。
    - 第二個維度 pre：紀錄前一位數是否為 6 (true/false)
    - 第三個維度 lim：是否到達上限 (true/false)。例如 n = 655，假如最高位數是 6 (達上限)，則第二位數最大只能是 5。如果最高位數小於 6 (未達上限)，則第二位數最大可以是 9。
- 函式 dfs( )：從高位數往低位數檢查。
- 函式 solve( )：計算數字的位數，然後呼叫 dfs( )，初始狀態 pre = false, lim = true。
- 題目要計算 [n, m]區間內符合條件的個數，因此答案是 solve(m) – solve(n-1)。

程式碼

```
#include <iostream>

#include <cstring>

using namespace std;
```

```
int n, m, a[7], dp[7][2][2];


int dfs(int pos, int pre, int lim){
    //遞迴終止條件
    if (pos == -1) return 1; //檢查完所有位數，得到一組解
    if (dp[pos][pre][lim] != -1) return dp[pos][pre][lim];
    //ub：這個位數的上限值
    int ub = lim ? a[pos] : 9;
    int ans = 0;
    for (int i = 0; i <= ub; i++){
        if (i == 4) continue; //不能有 4
        else if (pre && i == 2) continue; //不要 62
        ans += dfs(pos-1, i==6, lim && i==a[pos]);
    }
    dp[pos][pre][lim] = ans;
    return ans;
}
```

```
int solve(int x){

    int cnt = 0;

    memset(dp, -1, sizeof(dp));

    //cnt：數字 x 有幾位數

    while (x){

        a[cnt] = x % 10;

        x /= 10;

        cnt++;

    }

    return dfs(cnt-1, 0, 1);

}



int main() {

    while (cin >> n >> m){

        if (n == 0 && m == 0) break;

        cout << solve(m) - solve(n-1) << "\n";

    }

}
```

【更多例題】Codeforces 1245F. Daniel and Spring Cleaning

【網址】https://yuihuang.com/cf-1245f/

§18. 動態規劃演算法的應用(12) – 區間 DP

區間 DP，顧名思義，是用來求區間最值(最大或最小值)問題。通常的做法是藉由枚舉分割點，並更新小區間最佳解，以求得最終解。

【例題】ZeroJudge d686: 10003 – Cutting Sticks
【網址】https://yuihuang.com/zj-d686/
【作法】

- a[ ]：題目給定第 1～N 個切割的地方，且由小到大排列好。
- 加入第 0 個切割點及第 N+1 個切割點：
  - a[0] = 0
  - a[N+1] = L
- dp[x][y]：從第 x 個切割點到第 y 個切割點，所需最小的成本。
  - dp[x][x+1] = 0 (中間無其它切割點)
  - (Line-13) 枚舉中間的切割點，找出最小成本。
  - (Line-15) 需加上 a[y] – a[x]
- 【例子】L = 10，三個切割點：2、4、7，a[ ] = {0, 2, 4, 7, 10}|
  - 中間只剩一個切割點：抬起這根木棍的成本。
    - dp[2][4] = 6
    - dp[1][3] = 5
  - 中間還有兩個切割點：dp[1][4] = 下面兩個選項中成本最小者，再加上 8 (a[4] – a[1])，抬起這根木棍的成本。
    - 先做第 2 個切割點，dp[1][2] + dp[2][4] = 6
    - 先做第 3 個切割點，dp[1][3] + dp[3][4] = 5

程式碼

```
#include <iostream>
```

```
#include <cstring>

using namespace std;

int dp[55][55]; //最多 50 個切割的地方

int a[55]; //第 1～N 個切割的地方，由小到大排列好。



int solve(int x, int y){

    if (~dp[x][y]) return dp[x][y]; //記憶化

    if (x+1 == y) return dp[x][y] = 0;

    int cost = 0x3F3F3F3F;

    for (int i = x+1; i < y; i++){

        //枚舉中間的切割點

        cost = min(cost, solve(x, i) + solve(i, y));

    }

    return dp[x][y] = cost + a[y] - a[x];

}



int main() {

    int L, N;

    while (cin >> L){
```

```
        if (L == 0) break;

        memset(a, 0, sizeof(a));

        memset(dp, -1, sizeof(dp));

        cin >> N;

        for (int i = 1; i <= N; i++){

            cin >> a[i];

        }

        a[N+1] = L;

        cout << "The minimum cutting is " << solve(0, N+1) << ".\n";

    }

}
```