武陵資訊讀書會講義

李杰穎

May 3, 2021

Contents

Chapter 1

前言

1.1 課程介紹

先來介紹一下資訊讀書會吧！

資訊讀書會是去年 (2019) 才新成立的讀書會。我們上學期會教 C++ 的語法/演算法，而下學期主要是 C++ 算法的加強及一些如何做專題的技巧。下學期也有可能會用 Python 和 PyTorch 來上一點人工智慧的部份。課表部份未來應該會公佈在課程網站上。

前面兩堂課主要是上最基礎的語法，所以我們會採用發講義的形式讓大家可以看著講義練習寫寫看題目。當然，講師會先把講義內容清楚的帶過一次在讓你們寫練習題。如果在寫練習題的過程中有遇到困難也可以詢問各個講師。

1.2 編輯器/IDE(整合式開發環境) 介紹

- Dev-C++：應該算 IDE。安裝好後就可以直接使用，不需額外安裝編譯器或擴充功能。缺點是不能自動補程式碼 (autocomplete)，但主要還是推薦使用這個編輯器啦。
- Code::Blocks：功能比 Dev-C++ 完整的 IDE。Autocomplete 是好的，可以用。整體而言有點太肥了，所以我不太常用。不過還是一個不錯的選擇拉。
- Visual Studio Code：微軟出的編輯器，本身的功能可以透過許多擴充功能 (extensions) 進行擴充。如果想要在 VS Code 上寫 C++ 就要裝 C++ 的擴充功能才能比較方便的使用。經過一些設定後，autocomplete 的功能十分強大，而且內建的 terminal 對於執行程式很方便。

- NotePad++：超級威力加強板的 windows 內建筆記本，支援程式碼 highlighting，啟動很快。如果專注在打 code 或驗測資之類的也可以考慮用這款。
- **其他 (Sublime Text, Eclipse IDE, vim, nano, emacs, far file manager...)**：一些在打 competitive programming 在用的編輯器 or IDE，有興趣可以去網路上找找看相關的資料。

大家可以慢慢嘗試，找出自己最喜歡的編輯器喔。

1.3 課程網站及線上批改系統 (Online Judge)

- **課程網站** (**https://wulinginfor.github.io/**)：接下來上課的內容都會放在課程網站上，講義、投影片那些的都會放在上面，練習題也會公佈在上面。如果回家想要練習的話，就可以去網站上看看喔。
- **Online Judge**(**http://wloj.wlsh.tyc.edu.tw/**(WLOJ))：如果有在刷題的人應該對這個名字不太陌生。Online Judge(或簡稱 OJ) 就是一個會有許多程式題目的網站，在每一題中，都有題目敘述、輸入、輸出，我們要做的就是寫一個程式滿足題目敘述，使當輸入輸進來時程式都可以得到正確的輸出。當你認為你已經把程式寫完了，就可以把原始碼上傳到網站上，網站經過批改後，就會告訴你的程式是不是好的 (Accepted(AC)) 或者是錯的 (Wrong Answer(WA)) 或者是編譯錯誤 (Compile Error(CE)) 又或者程式是否超過了題目所限制的資源 (時間、記憶體等)(Time Limit Exceeded(TLE), Memory Limit Exceeded(MLE))

1.4 教學資源

- 2020 資訊之芽語法班：資訊之芽語法班是台大和清大一起合開的課，在清大和台大都有開課。他們會把上課的內容全部丟到網路上讓有需要的人可以學習。
- 2020 資訊之芽算法班：資訊之芽算法班是更為進階的課程，只有在台大開課。對算法班課程有興趣的人可以關注一下資訊之芽的粉絲專頁，他們每年大約寒假的時候就會有課程報名的消息了。
- GeeksForGeeks：國外的網站，有很多語法和算法的教學，如果對某個語法或算法不熟可以上去找一下。

1.5 Online Judge

- Sprout OJ：資訊之芽的 OJ，上面有各種難度的題目，有興趣的可以刷刷看。
- ZeroJudge：應該是台灣最大的 OJ, 題目超級多，幾乎不可能刷完。水題偏多 (至少前 30 題都滿水的)，但還是有不錯的題目。
- UVa Online Judge：全世界第一個 OJ，想當然爾題目非常的多。缺點是網站的界面有點不太直觀，但似乎準備要更新了。
- CodeForces：全世界最大的線上程式競賽網站，題目相當有水準，如果上面的題目都解不出來也不要太挫折喔。
- CS Academy：CS Academy 上面有很多不錯的題目，而且每題都會分成 Easy, Medium 及 Hard。大家可以看自己的解題情況決定要怎麼刷題喔。

Chapter 2

Hello World!

程式碼 2.1: 一個標準的 C++ 程式碼

```cpp
#include <iostream>
using namespace std;
int main(){
    cout << "Hello World!" << endl;
    return 0;
}
```

大家可以把這段 code 複製到 Dev C++ 上。

想必大家在學習各種程式語言的第一個程式應該都是 Hello World 吧。資訊讀書會也不例外，我們就來看看這段程式碼在寫什麼吧。

第一行是引入 C++ 的輸入輸出標頭檔 (header file)，如果沒有引入這個標頭檔，C++ 程式就沒有辦法在程式執行的畫面 (小黑窗) 上輸出或輸入東西了，也就看不到程式的執行過程了。

第二行對於目前來講不太重要啦，下面是如果不加這行 code，一個能成功編譯的 code 會長怎樣。

程式碼 2.2: 一個標準的 C++ 程式碼但不加 `using namespace std`

```cpp
#include <iostream>
int main(){
```

```
3     std::cout << "Hello World!" << std::endl;
4     return 0;
5 }
```

可以發現如果不加這行，`cout` 跟 `endl` 前面就要再加上 `std::`，實際寫程式的時候變得相當的麻煩，所以一般來說在寫 C++ 的時候多會在 `#include` 下面加上這行。

第三行是一個特別的函數 (function)，我們稱之為 main function，它是整段 C++ 程式開始執行的進入點，C++ 的執行都會從這個函數開始。至於 `int main() { ... }` 是什麼意思我們以後會提到。

第四行就是整段程式輸出 Hello World! 的地方，主要由 `cout`, `<<`, `endl` 組成，大家可以把 `cout` 想成用來處理輸出的關鍵字 (keywords)，我們用 `<<` 來把我們要輸出的文字 (字串) 括起來，而 `endl` (endline) 則是用來換行的，大家可以試試看如果把 `endl` 刪掉會發生什麼事。

第五行是整個 main function 的終止點。一般來說，如果 main function 有成功執行的話，其回傳值就是 0。至於回傳值是什麼我們也等到以後再提吧。

2.1 練習題

- WLOJ 1001: Hello World!

Chapter 3

表達式與變數

程式碼 3.1: 解一元二次方程式

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main(){
    // ax^2 + bx + c = 0
    double a, b, c;
    cin >> a >> b >> c;
    cout << (-b+sqrt(b*b-4*a*c))/(2*a) << " ,";
    cout << (-b-sqrt(b*b-4*a*c))/(2*a) << endl;
    return 0;
}
```

在這段 code 中，我們多引入了一個標頭檔 cmath，引入這個標頭檔可以讓我們使用 sqrt() 這個函數，sqrt 是 square root 的縮寫，也就是開根號。有了根號我們就可以利用公式解，解出一元二次方程式的兩個根。

3.1 合法的程式

- 標頭檔 (header file)：
  標頭檔像是指令的說明書。特定功能要在特定的說明書中才會描述。所以當使用不同函數時，可能要引入不同的標頭檔。
- 主函式 (main function)：
  顧名思義，主程式。程式由此開始執行。

3.2 輸入輸出

在 C++ 中，我們通常使用 cin、cout 來處理輸入、輸出。使用這兩個功能需要引入標頭檔 iostream，也要記得打 using namespace std;

- `cout` (念 see-out) 代表輸出
- `cin` (念 see-in) 代表輸入
- 注意 `cout`、`cin` 的箭頭方向是相反的 (`<< (cout)`、`>> (cin)`)，可以想像成資料的流向
- `<<`、`>>` 裡面的東西會被解讀成字或特殊符號
- 最後記得加分號，代表敘述句結束

3.3 運算子

運算子是 C++ 用來進行數學運算的符號，C++ 除了能做到基本的加減乘除外，還可以做到許多特殊的運算。

下面是一些常見的運算子 (按運算優先級排列)：

- `++`：遞增, `i++`
- `--`：遞減, `i--`
- `[]`：陣列存取, `a[1]`
- `!`：邏輯非 (not), `!a`

- `*`：就是乘號
- `/`：就是除號
- `%`：取餘數
- `+`：就是加號
- `-`：就是減號
- `>>`：位元右移, `a >> 1`
- `<<`：位元左移, `a << 1`
- `>`：大於關係, `3 > 2` (運算結果 `true`)
- `≥`：大於等於關係, `3 ≥ 3` (運算結果 `true`)
- `<`：小於關係, `3 < 2` (運算結果 `false`)
- `≤`：小於等於關係, `3 ≤ 3` (運算結果 `false`)
- `==`：等於關係, `3 == 3` (運算結果 `true`)
- `!=`：不等於關係, `3 != 3` (運算結果 `false`)
- `&&`：邏輯 AND, `true && false` (運算結果 `false`)
- `||`：邏輯 OR, `true || false` (運算結果 `true`)
- `=`：直接賦值
- `+=`：以和賦值, `a = 1, a += 2`, (`a` 此時為 3) (等價於 `a = a + 2`)
- `-=`：以差賦值
- `*=`：以積賦值
- `/=`：以商賦值
- `%=`：以取餘數賦值

3.4 表達式

表達式是一種廣義的算式，代表一個**值**。每個表達式包含運算子和運算元，運算規則就是先乘除後加減，每個運算的優先度不同。

程式碼 3.2: 各種表達式

```
(-b-sqrt(b*b-4*a*c)/(2*a) // 值由 a, b, c 的值決定
7122 % 20 // 取餘數運算，算出的值為 2
0 < 3 // 小於運算, 算出的值為 True，也就是 1
6/2(1+2) // 少了乘號，編譯器看不懂
a*x*x + bx + c = 0 // 這是方程式，本身沒有值，編譯器看不懂
```

3.5 變數

變數是程式中很重要的元素。程式中如果沒有變數就沒辦法達成很多我們想要達成的功能了。

在 C++ 中，變數在使用前需要先宣告 (declare)。與 Python 不同，C++ 的變數在宣告時就要決定這個變數的資料型態 (等等會提到)。變數的名稱只能是英文或底線開頭，後面可以有數字，但不能是保留字 (keywords, 後面應該會提到)。另外，大小寫對於變數的名稱是不同的。

變數可以讀，也可以改 (const 除外)，變數的數值儲存在電腦的記題體中。讀值的時候直接當作數值使用。而寫值的時候通常使用等號來寫，這個等號是賦值 (把一個數值給一個變數) 的意思，與數學中的等號不同。通常我們把等號左邊稱作 lvalue、右邊稱為 rvalue。

程式碼 3.3: 一堆變數

```
#include <iostream>
using namespace std;
int main(){
    int price = 8740;
    int my_money_Monday = 314159;
    int my_money_Tuesday = my_money_Monday - price;
    my_money_Monday = my_money_Monday - price*6; // 寫入
```

```
    my_money_Monday -= price * 7; // 偷懶的寫法
    cout << my_money_Monday * 1.0106 << endl;
    return 0;
}
```

程式碼 3.4: 不好的示範

```
#include <iostream>
using namespace std;
int main(){
    int my_money=59457, price=879457 // 沒有分號
    bool have_money, 1_thing_to buy=0; // 數字開頭、有空隔
    my_money <- (my_money - buy*price); // <- 不能給值，請用 =
    have_Money = my_Money <> 0; // 沒有宣告 (大寫)、不等於請用!=
    cin << price; // 方向錯
    cout << " 我" < have_Money << " 定不是沒錢\n"; // 小於?
    cout << yee~ << my_Money > 0; // 沒引號，> 運算要加括號
    return 0;
}
```

3.6 變數的資料型態

在 C++ 中，變數被分為了很多不同的資料型態。資料型態有非常多種，甚至可以自己創一個資料型態，以下只列出最基礎的幾種。

類型	資料型態	名稱	佔有記憶體空間 (bytes)	儲存值範圍
整數	bool	布林	1	0 ∼255
	int	整數	4	$-2^{31} \sim 2^{31}-1$
	unsigned int	無號整數 (無負數)	4	$0 \sim 2^{32}-1$
	long long	長整數	8	$-2^{63} \sim 2^{63}-1$
	unsigned long long	無號長整數 (無負數)	8	$0 \sim 2^{64}-1$
浮點數 (小數)	float	單精度浮點數	4	$\pm 3.4 \times 10^{-38} \sim \pm 3.4 \times 10^{38}$
	double	倍精度浮點數	8	$\pm 1.7 \times 10^{-308} \sim \pm 1.7 \times 10^{308}$
	long double	長倍精度浮點數	12∼16	$\pm 1.7 \times 10^{-308} \sim \pm 1.7 \times 10^{308}$
字元	char	字元	1	0 ∼255

有了這些資料型態，我們就可以在適當的時機使用特定資料型態的變數，來達成目標。

3.7 練習題

- WLOJ 1002: A+B Problem
- WLOJ 1003: 球球的表面積
- WLOJ 1004: 我要成為化學大師
- WLOJ 1006: A+B Problem (Hard Version)

Chapter 4

流程控制 (if...else)

4.1 if

if 就是英文中如果的意思，在 C++ 中，我們可以透過 if 來進行流程控制。具體直接看 code 吧。

程式碼 4.1: if

```
#include <iostream>
using namespace std;
int main(){
    if (1 * 2 >= 2){
        cout << "1 * 2 >= 2" << endl;
    }
    cout << "statements after if" << endl;
}
```

簡單來說，當 if 後面跟的那個括號那個敘述是 true 的話，程式就會執行大括號內的程式碼。

小提醒，如果你想要判斷一個數字 x 是否介於 30 跟 60 之間的話，判斷式不能寫成 `30≤x≤60`，要寫成 `(30 ≤ x) && (x ≤ 60)`。原因是因為 `30≤x≤60` 可以寫成 `(30≤x)≤60`，但 `(30≤x)` 可以是 0 或 1，而 0<=60，1<=60，所以不管 x 的值是多少這個敘述都是對的。

4.2 else & else if

else 就是英文中的不然的意思，else 配合 if 就可以達到更完整的流程控制。else 只能接在 if 後面，當 if 內的敘述不成立時，程式就會繼續往下看是否有 else 或 else if。else 與 else if 的差別在於 else if 後面也是跟著一個敘述，而 else 沒有敘述，當程式碰到 else 的時候，就會直接執行大括號裡面的程式碼。

程式碼 4.2: if, else if, else

```
#include <iostream>
using namespace std;
int main(){
    if (condition1) {
        cout << "I'm in condition1" << endl;
    } else if (condition2) {
        cout << "I'm in condition2" << endl;
    } else {
        cout << "I'm in else" << endl;
    }
}
```

小提醒，在一段 if, else if, else 區段中，只有一個大括號會被程式執行。也就是說，當區段中已經有大括號內程式碼被執行的話，其他大括號內的程式碼是絕對不會被執行的到，就是其敘述是正確的。例如下面這段 code。

程式碼 4.3: Never be executed

```
#include <iostream>
using namespace std;
int main(){
    if (true) {
        cout << "I will be executed." << endl;
    } else if (true) {
```

```
7           cout << "I will never be executed QQ" << endl;
8       }
9   }
```

4.3 練習題

- WLOJ 1005: 季節判斷
- WLOJ 1007: 3n+1 Problem (Easy Version)
- WLOJ 1008: 天將降大任於斯人也

Chapter 5

while 迴圈

如果你現在想要把 1~10 的數字分別印出來的話，你會怎麼做呢？以下是其中一種方法：

程式碼 5.1: Naive 的作法

```cpp
#include <iostream>
using namespace std;
int main(){
    cout << 1 << endl;
    cout << 2 << endl;
    cout << 3 << endl;
    cout << 4 << endl;
    cout << 5 << endl;
    cout << 6 << endl;
    cout << 7 << endl;
    cout << 8 << endl;
    cout << 9 << endl;
    cout << 10 << endl;
}
```

看起來是不是有點繁瑣呢？但如果這時你又想要印出可能 1~1000、1~10000 的數字時，以上的作法就會變得非常的不好，程式沒辦法簡潔的寫出來。

此時，while 迴圈就派上用場了，while 迴圈具體會長這樣：

程式碼 5.2: while 迴圈

```
#include <iostream>
using namespace std;
int main(){
    while(condition){
        // statements
    }
}
```

while 迴圈具體會做的事就是當 conditon 為 true 時，while 迴圈就會執行大括號裡面的程式碼。但與 if 不同的是，當大括號裡面的程式碼執行完後，while 迴圈會再一次的檢查 condition 是否仍然是 true，如果是的話就繼續執行大括號內的程式碼，直到 condition 為 false 為止。

所以上面的 code 就可以改寫成這樣：

程式碼 5.3: while 迴圈的作法

```
#include <iostream>
using namespace std;
int main(){
    int n = 1;
    while (n <= 10){
        cout << n << endl;
        n ++;  // 與 n = n + 1 和 n += 1 相同
    }
}
```

5.1 do...while 迴圈

除了一般的 while 迴圈以外，還有一種特殊的 while 迴圈，稱之為 do...while 迴圈，具體長這樣：

程式碼 5.4: do...while 迴圈

```cpp
#include <iostream>
using namespace std;
int main(){
    do{
        // statements
    } while(condition);
}
```

可以發現在 do...while 迴圈中，出現了 do 這個關鍵字，且 while 的 condition 也放在了大括號的後面。do...while 迴圈與 while 迴圈不同的地方在於 do...while 迴圈將先會執行一次大括號內的程式碼，再去判斷 condition 是否為 true。所以無論 condition 是否為 true，do-while 迴圈大括號內的程式碼至少會執行一次。因此，do-while 循環屬於後測迴圈 (post-test loop)。相對的，while 迴圈就被稱為前測迴圈 (pre-test loop)。

一般來說，do...while 迴圈很少被使用，除了以後會講到的 `next_permutation()` 函數才會與 do...while 迴圈配合使用。

5.2 練習題

- WLOJ 1009: 3n+1 Problem (Hard Version)
- WLOJ 1010: 階乘問題
- WLOJ 1011: 武陵專用拖鞋

5.3 break & continue

5.3.1 break

當在迴圈中，如果你突然不想讓迴圈繼續執行接下來的程式碼的話，我們就可以加上一行 `break;`，使迴圈直接跳出，不繼續執行接下來的程式碼，具體而言在 while 迴圈中的使用如下：

程式碼 5.5: break

```cpp
#include <iostream>
using namespace std;
int main(){
    while(condition){
        // some statements
        break;
        // this part of statements will never be executed
    }
}
```

可以發現由於 break 的特性，break 通常與 if 一起配合使用。也就是說讓迴圈控制在一定條件才會跳出。

5.3.2 continue

另一個與 while 迴圈配合使用的是 continue，它的功能和 break 有點類似，都是使程式不會執行到在它下面的程式碼。但與 break 不同的是，continue 並不會跳出迴圈，而是繼續執行迴圈，只不過使它以下的程式碼不會被執行而已。

程式碼 5.6: continue

```cpp
#include <iostream>
using namespace std;
int main(){
    while(condition){
        // some statements
        continue;
        // this part of statements will never be executed,
        // but the rest of statements will be executed continuously.
    }
}
```

同樣的，continue 也常跟 if 一起搭配使用。

break 和 continue 都是我們所謂的關鍵字 (keywords)，這些關鍵字在程式中佔有重要的位置。

Chapter 6

for 迴圈

有了 while 迴圈，所以需要迴圈才能做到的事似乎就迎刃而解了。但事情可沒想像的那麼簡單，我們來看看下面這個例子，假設你現在想要重複做一件事 (執行一段程式碼) 10 次，在 while 迴圈大概就要這樣寫：

程式碼 6.1: 用 while 迴圈來重複做事

```
#include <iostream>
using namespace std;
int main(){
    int i = 0;
    while(i < 10){
        // some statements
        i ++;
    }
}
```

有沒有覺得很麻煩呢？變數的初始化、判斷式、變數的累加被分到了三行，感覺就一整個不對了，所以程式設計師很聰明的設計出了 for 迴圈，上面的程式碼就可以被縮減成這樣：

程式碼 6.2: 用 for 迴圈來重複做事

```cpp
#include <iostream>
using namespace std;
int main(){
    for(int i=0;i<10;i++){
        // some statements
    }
}
```

有沒有覺得看起來簡短很多呢？變數 (計數器) 的初始化、判斷式、變數的累加全部被縮到了一行，而且很好的是，在 for 迴圈中宣告的變數生命週期 (lifetime) 只有在 for 迴圈中 (生命週期以後會提到)，這當然包括了這裡的 `int i` 變數，所以我們就可以在這個 for 迴圈後，重複使用 `int i` 變數，而不會出現重複宣告的問題。下面是 for 迴圈的具體使用方法。

程式碼 6.3: for 迴圈

```cpp
#include <iostream>
using namespace std;
int main(){
    for(initialization;condition;afterthought){
        // some statements
    }
}
```

for 迴圈的括號內主要包含三個部分：初始化 (initialization)、條件式 (condition) 和遞增 (afterthought)，**值得注意的是這三個部分是由分號; 來做分割的**，並不像函數是用逗號, 來做分割的，這點是需要注意的，如果不小心寫錯就會無法編譯成功了。

初始化是宣告（或者賦值）任何需要的變數的動作。如果你要使用多個變數，則變數的種類要一致。條件的部分則是檢查是否需要跳出 for 迴圈。如果條件為 true，則繼續執行迴圈。如果條件判斷為 false，則離開迴圈。遞增在每跑一次迴圈都會重複執行一次。

for 迴圈的三個部分是可有可無的，也就是說，我們其實可以將 for 迴圈當作無窮迴圈使用：

程式碼 6.4: for 無窮迴圈

```cpp
#include <iostream>
using namespace std;
int main(){
    for(;;){
        // some statements
    }
}
```

6.1 巢狀 for 迴圈

我們可以在 for 迴圈中，再加上一個 for 迴圈，具體會長這樣：

程式碼 6.5: for 巢狀迴圈

```cpp
#include <iostream>
using namespace std;
int main(){
    int x, y;
    for(int i=0;i<x;i++){
        for(int j=0;j<y;j++){
            // some statements
        }
    }
}
```

可以想一下這個巢狀迴圈內的 statements 總共會執行幾次？

同樣的，我們也可以在 for 迴圈中使用 break/continue，其效果與 while 迴圈中相同。

6.2 for 迴圈與 while 迴圈的比較

- for 迴圈和 while 迴圈可以互相轉換，主要視情況使用。

- 一般來說，for 迴圈主要是用於當迴圈要跑的次數是固定時或每次跑完迴圈都要有固定的更新。
- 反之，若迴圈執行次數較不明確，或條件牽扯到使用者輸入，則適合使用 while 迴圈。

條件牽扯到使用者輸入，例如當輸入是由 EOF(End of File) 做結尾時，就適合使用 while 迴圈進行輸入的處理：

程式碼 6.6: `while(cin >> n)`

```cpp
#include <iostream>
using namespace std;
int main(){
    int n;
    while(cin >> n){
        // some statements to deal with inputs
    }
}
```

這個程式碼為甚麼可以偵測到 EOF 呢？原因是因為當 `cin` 偵測到現在讀入到 EOF 時，`cin >> n` 就會因為沒辦法正確的讀取到一個整數而回傳 false(反之正常情況下，`cin >> n` 都會回傳 true)，所以當 `cin` 讀到一個整數的時候，就會直接跳出 while 迴圈，輸入部分也跟著處理完成，就可以開始進行題目需要做的運算。

Chapter 7

Coding/Programming Style

我覺得這份 2020 資訊之芽語法班的簡報寫的很好，可以去看看：https://www.csie.ntu.edu.tw/~b06902029/reveal.js/Sprout/2020/CodingStyle，以下是一些簡報裡面提到的重點：

- **大準則：**
  - 可讀性 (Readability Counts)
  - 前後一致 (Be Consistent)
  - **想想自己五年後還看不看得懂這份 code**
- **變數命名：**
  - 一定要有它的意義，避免無意義的變數：
    - O: `numOfStudents, num_of_students`
    - X: `a, b, ..., tmp`
  - 可以使用一些縮寫、命名慣例：
    - 縮寫：`number` → `num`, `length` → `len`
    - 命名慣例：迴圈中的 `i, j, k`
  - 變數的命名若是由多個詞組合而成，主要可以使用兩種方式進行串接：
    - **camelCase:**
      - 直接把多個詞串在一起。第一個字字首不大寫，其他字字首皆大寫。
      - 如 `numOfStudents`

    - **under_score:**
        - 全部小寫，兩字中間用底線 `_` 串連。
        - 如 `num_of_students`

- **程式長相：**
    - 縮排：
        - 縮排很重要，沒有縮排沒人會想幫你 debug。
        - 通常進入一個大括號縮排一次。
        - 注意縮排的格數要一致，沒有一致跟沒縮一樣。
    - 空格：
        - 適時加上空格可以增加可讀性。
        - `int k = a * b + c;` > `int k=a*b+c;`
        - `cout << some << thing << endl;`
    - 註解：
        - 適時加上註解
            - `//`：單行註解
            - `/* ... */`：區塊註解
        - 註解盡量使用 ASCII

Chapter 8

陣列

8.1 1000 根棍子

假設你是某個工廠的老闆，今天工廠生產了 1000 根棍子，編號分別是 1∼ 1000，而且每根棍子都有不同的高度。

這時有一位員工的工作是把編號 1 的棍子和編號 1000 的棍子接起來、編號 2 的棍子和編號 999 的裉子接起來、...、編號 n 和編號 $1001 - n$ 的棍子接起來。但是因為要趕著出貨的關係，所以你想要在這位員工還沒把棍子接起來前就知道這 500 根棍子的長度了。

如果這時你不會陣列的話，把上面的任務寫成程式大概會長這樣：https://ideone.com/MZnVgd

非常的慘，所以這時候應該要怎麼做呢？

8.2 一維陣列

甚麼是陣列 (array) 呢？簡單來說就是把有相同性質的資料連接起來，且這些連接起來的資料可以使用索引 (index) 來存取，具體可以看 code：

程式碼 8.1: 陣列 (array)

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

```
int main(){
    int s[4];
    cin >> s[0] >> s[1] >> s[2] >> s[3];
    cout << s[0] + s[3] << endl;
    cout << s[1] + s[2] << endl;
}
```

8.2.1 宣告

首先看到 main function 的第一行，這行就是宣告陣列的地方，宣告的方式則是“資料型態 陣列名稱 [陣列大小]”，所以在上述的程式碼中，我們在第一行就宣告了一個 “長度為 4(代表陣列中有 4 個變數) 的整數 s 陣列”。

8.2.2 存取

再來是第二行的部分，這邊主要就是存取陣列內的變數，我們可以透過在方括號中填入索引值 (index) 來存取變數，要注意的是在 C++ 中，陣列的第一個變數 (元素) 的索引值為 0，這點非常重要，很多初學者都會不小心忘記這點。所以在上述變數中，如果試著存取 `s[4]` 就會使程式無法順利的執行下去，可能會出現 Runtime Error 或甚至在編譯期間就不給過了。

8.2.3 初始化

C++ 中，陣列的初始化有幾種作法，具體如下：

程式碼 8.2: 陣列的初始化

```
#include <iostream>
using namespace std;
int main(){
    int Sa[5] = {0, 1, 2, 3, 4};
    int Sb[5] = {0, 1}; // 自動補零
    int Sc[] = {0, 1, 2, 3, 4, 5, 6, 7}; //自動計算長度
```

```
    int Sd[1000000] = {}; // 全部補零, 因為會自動補零
}
```

大家可以試著編譯看看，然後陣列裡面的某個變數，看看裡面放了甚麼。

所以上面的 1000 根棍子問題就可以很簡單的解決了，具體的 code 如下：

程式碼 8.3: 1000 根棍子

```
#include <iostream>
using namespace std;
int main(){
    int s[1000];
    for(int i=0;i<1000;i++){
        cin >> s[i];
    }
    for(int i=0;i<500;i++){
        cout << s[i] + s[999 - i] << endl;
    }
}
```

8.2.4 不標準寫法

程式碼 8.4: 不標準寫法

```
#include <iostream>
using namespace std;
int main(){
    int n;
    cin >> n;
    int arr[n] = {};
    return 0;
}
```

ㄟ感覺滿對的喔，然後丟到 Dev-C++ 編譯也是對的，這有甚麼錯呢？其實在 C++ 的標準中，陣列長度內放變數是不正確的，有些編譯器不會接受這種寫法，所以盡量不要這樣寫喔 (雖然我有時候也這樣寫 www，不過不要學我)。

那有沒有其他方式可以讓陣列長度放變數呢？答案是有的，不過是使用 C++ STL 中的 **vector** 的寫法：

程式碼 8.5: 高階寫法

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main(){
    int n;
    cin >> n;
    vector<int> arr(n);
    return 0;
}
```

這個東西叫 vector，直翻雖然叫向量，不過它跟向量沒有太大的關係，可以把 vector 想成是一種 C++ 的動態陣列，可以一直往後面加東西。這個東西是屬於 C++ 的 STL(Standard Template Library)，STL 裡面還有很多實用的東西，以後的課程應該會提到。

8.3 二維陣列

在 C++ 中，陣列除了有一維的形式，還可以二維的形式存在，這就是二維陣列。

8.3.1 宣告

二維陣列宣告的方式如下：“資料型態 變數名稱 [長度一][長度二];”，像如果想要宣告一個資料型態為 **int** 且長度一為 4、長度二則為 3 的 A 陣列，其程式碼就會像下面那樣。

程式碼 8.6: 二維陣列的宣告

```cpp
#include <iostream>
using namespace std;
int main(){
    int A[4][3];
}
```

圖 8.1: 二維陣列圖解

那長度一、長度二具體是甚麼意思呢？我們可以看一下圖 8.1，可以發現長度一的 4 可以想成二維陣列裡面放的一維陣列數量，也就是圖中的橫的那部分，而長度二則代表每個一維陣列的長度。

所以簡單來說，二維陣列可以想成在一個一維陣列再塞一維陣列。

8.3.2 存取

如圖 8.1 二維陣列的每個值就像是表格中所填的那樣，第一個方括號是用來選取是哪個一維陣列，第二個方括號是用來選取是一維陣列中的哪個值。

8.3.3 初始化

初始化主要有下面幾種寫法：

程式碼 8.7: 二維陣列的初始化

```cpp
#include <iostream>
using namespace std;
int main(){
    int A[4][3]; // 自動補 0
    int B[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    // 按照 [0][0], [0][1] 的順序依序填入
    int C[4][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12},
    }; // 寫成許多一維陣列的形式
    int D[][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9},
        {10, 11, 12},
    }; // 可以自動算長度，不過最後一個長度一定要是給定的
}
```

8.4 多維陣列

除了二維陣列以外，我們可以在宣告時加上更多的方括號，成為多維陣列。

其實更高維度的陣列也可以用前面的思考方式，想成在陣列裡面塞陣列，像三維陣列就是把一個陣列塞一個陣列，那一個陣列裡面再塞一個陣列。

多維陣列的具體使用方法和二維陣列差不多，我就不再多講一次。

小問題：`int A[i][j][k][l]` 這個陣列裡面有幾個元素呢？

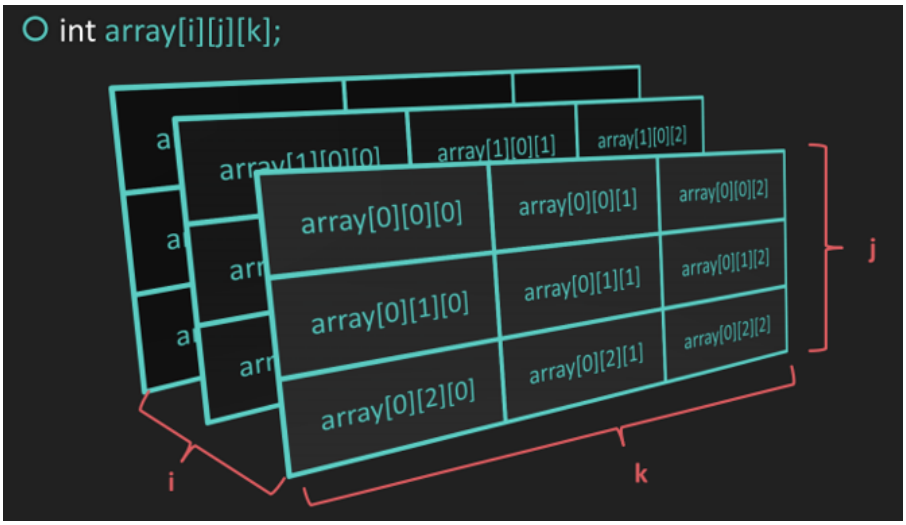
int array[i][j][k];
array[1][0][1]
array[1][0][2]
array[0][0][0]
array[0][0][1]
array[0][0][2]
array[0][1][0]
array[0][1][1]
array[0][1][2]
array[0][2][0]
array[0][2][1]
array[0][2][2]
i
j
k

圖 8.2: 多維陣列？

Chapter 9

字元與字串

9.1 字元

字元 (char) 代表的可以是一個數字、字母或是符號。在程式中，我們會以 ASCII 碼 (https://zh.wikipedia.org/wiki/ASCII) 來表示字元。除了 0123456789、abcdABCD、# *@!() 外，還有一些控制字元 (不會顯示在畫面上)，像是:

控制字元	說明
'\t'	定位字元 (TAB)
'\n'	換行 (ENTER)
'\r'	回車 (RETURN)
'\b'	倒退 (Backspace)
'\0'	字串結尾

程式碼 9.1: 字元宣告與初始化方式

```
#include <iostream>
using namespace std;
int main(){
    char A;//宣告一個字元
    char B = 97;//宣告字元並設初始值為 97
    char C = 'a';
```

```
}
```

在程式碼中，會以 char 型態儲存一個字元。若要表示一個字元，會以一對單引號 (') 將字元夾起來。在先前有提過，char 型態存的是 ASCII 數值，所以'a' 會對應到 97，並將值賦給 char。

如果想要表示單引號 (') 要怎麼辦呢? 這時，就要在前面加上反斜線 (\) 來表示，反斜線 (\) 又稱為「跳脫字元 (escape character)」，可以用來表示特殊字元，例如:

特殊字元	表示方法
單引號 (')	\'
雙引號 (")	\"
\	\\
控制字元	如先前所提

由於字元本身就是一個整數，所以它可以做運算。以下的等號都是成立的:

程式碼 9.2: 字元運算

```
'b' == 'a' + 1
'A' == 'a' ^ ' '
2 == 'c' - 'a'
'y' == ('c' + 'h' + 'a' + 'r')%26 + 'a'
```

9.2 字串

字串 (string)，顧名思義就是把字元串在一起。

程式碼 9.3: 字串 (string)

```
#include <iostream>
#include <cstring> //for string
using namespace std;
int main(){
```

```
5     char A[9];//宣告長度為 9 個字元的陣列
6     string B;
7     cin >> A >> B;
8     cout << A[0] << B[0];
9     cout << A << B;
10 }
```

9.2.1 宣告

首先看到 main function 的第一行，字元陣列宣告方式為”char 字元陣列名稱 [字串長度]”。此外，也可以引入 <cstring> 標頭檔，宣告方式為 “string 字串名稱”。

9.2.2 存取

再來是第 3 ∼ 5 行，先前兩種宣告方式都可以

1. 讀入整行字串 (第 4 行)。若用 cin» 讀入，會讀到’ ’,’\t’,’\n’ 為止。
2. 存取字串中某個字元 (第 5 行)。
3. 存取整個字串 (第 6 行)。在每個字串的最後，皆會有’\0’ 作結。故輸出到’\0’ 時，會停止輸出。

9.2.3 初始化

程式碼 9.4: 字串初始化

```
1 #include <iostream>
2 #include <cstring> //for string
3 using namespace std;
4 int main(){
5     char A[] = {'s','t','r','i','n','g'};
6     char B[] = "string";
7     string C = "string";
8     char D[6] = "string";//這是錯誤的初始化方式，可能會無法 compile
```

```
9 }
```

在初始化時，可以 char 型態的陣列一樣以 char 分開儲存。但這樣太麻煩了，所以我們可以用雙引號 (") 將字串夾起來。在 main function 中的第 4 行是錯誤的，為什麼呢? 這是因為字串會以'0'結尾，所以若用字元陣列宣告的字串長度必須加 1。

9.2.4 文字處理函式

https://en.cppreference.com/w/cpp/string/byte有詳細整理各種函式，以下整理常用函式。

獲取文字系列：

- cin.get()：讀取下一個字元 (包括空白字元' '、換行字元'\n')
- cin.get(陣列 A, 讀取數量 n)：讀取 n 個字元至陣列 A
- cin.getline()：讀取整行 (包括空白字元' ')
- getchar()：讀取一個字元

str 系列，以下函式必須加入 #include<cstring> 才能使用：

- strlen(字串)：得到字串長度
- strcmp(字串 A, 字串 B)：回傳 0 表示 AB 一樣，回傳 1 表示 AB 不一樣
- strcpy(字串 A, 字串 B)：把 B 複製到 A

以下函式必須加入 #include<cctype> 才能使用：

is** 系列，如果字元屬於 ** 則回傳 1，否則回傳 0:

- isalpha(字元)：判斷字母
- isdigit(字元)：判斷數字
- isspace(字元)：判斷空白

to** 系列，將字元轉成 **：

- `tolower(字元)`：轉成小寫
- `toupper(字元)`：轉成大寫

Chapter 10

資料結構 Data Structure

相信大家都是 C 或者 C++ 的使用者。

如果你 C 的使用者，強烈建議你現在開始使用 C++。語法相容，但又有強力的 library，是你在競賽時間壓力下不可或缺的強力工具。

如果你是 C++ 的使用者，強烈建議你把 C++ 內建又好用的東西摸熟。這樣就可以省去許多功夫，開心使用前人為你寫好的東西了。

10.1 什麼是資料結構

就是字面上的意思：儲存資料的結構。

像前面有提到的陣列就是一種資料結構，它直接把一堆同樣型別的資料排成一排好好管理。但除了陣列之外，還有許多資料結構（有些甚至不是內建的）是需要瞭解的。以下就講些簡單常用的內建資料結構吧。

這些資料結構在「標準模板庫」(Standard Template Library，簡稱 STL）之中，是一個 C++ 的程式庫。注意，這個程式庫的所有東西都在 `namespace std` 底下。

10.2 型別模板 (Template)

不過在繼續之前，有個重要的 C++ 觀念要講。

試想有一個作為容器用的型別 `C`，可以容納一些型別是給定的型別 `T` 的東西。那你在宣告這樣的一個變數時，理論上你應該告訴編譯器這個變數容納的型別是哪個型別。然而 C 語言中並沒有這樣

的語法，所以在這樣的需求下模板（template）就在 C++ 誕生了。

型別模板的功用就是生出一堆新的型別。比方說剛剛的 `C` 好了，存放 `int` 型別的 `C` 和存放 `double` 型別的 `C`，型別應該要是不同的。所以在宣告一個變數時，不是寫「`C (變數名)`」，而是寫「`C<(int 或 double)> (變數名)`」以指定這個變數的型別要是存放 int 的那種型別，還是存放 double 的那種型別。

有些型別模板的參數不只一個，寫好寫滿就對了（例：`priority_queue<A, B, C>` 就需要三個型別）。如果沒有寫好寫滿，會自動採用預設的型別（前提是要有預設的）。

想瞭解更多的話，關鍵字搜「模板」跟「C++」吧。

註：C++11 開始，模板的參數數是可變的。

10.3 迭代器 (Iterator)

設想有個容器 `C`，裡面已經裝一些東西了。我該如何遍歷 `C` 中的所有元素呢？要知道 C 可能長得不像陣列，沒有「下標」這種東西。為了解決這個問題，C++ STL 為每個容器提供一個成員型別，叫做「迭代器」。

你可以把迭代器想像成是指標（事實上，指標也算一種迭代器）。如果你今天有一個迭代器 `i`，存取 `i` 指向的內容的方法，跟指標一樣，是在前面加個星號（`*i`）。而迭代器分成三種，取決於迭代器能進行的運算，由功能強到弱排序如下：

1. 隨機存取 (Random Access) 迭代器：這這類迭代器能夠和整數做加減法，加 s 代表從這項開始往後數 s 項，減 s 代表往前數 s 項。當然，遞增、遞減運算也沒有問題。你可以把指標當作這種迭代器。
2. 雙向 (Bidirectional) 迭代器：這類的迭代器只能做遞增（`++`）和遞減（`--`）運算，分別代表後一項和前一項。
3. 單向 (Forward) 迭代器：這類的迭代器只能做遞增（`++`），代表後一項。

而按照迭代器的使用方法，分成兩種：

1. 輸入 (Input) 迭代器：當你只有要讀取迭代器指向的內容時，這時迭代器當作輸入迭代器使用。所有的迭代器都可以當作輸入迭代器。
2. 輸出（Output）迭代器：當你要直接更改迭代器指向的內容時，這時迭代器當作輸出迭代器使

用。除了常數（const）迭代器（也就是規定不能更動迭代器指向的內容）以外，所有的迭代器都可以當作輸出迭代器。

C++ 內建的迭代器都很和藹，只要是可以做的運算，複雜度都是 O(1)。

為了滿足人們的需求，C++ 內建的容器通常有兩種迭代器：正常的迭代器以及逆向迭代器。假如容器的型別是 `C`，因為迭代器是原本型別的成員型別，宣告時名稱分別是 `C::iterator` 和 `C::reverse_iterator`。前者會從前迭代到後，後者會從後迭代到前。另外，每個容器 `c` 的兩種迭代器各有兩個迭代器代表頭尾，分別是 `c.begin()`、`c.end()` 和 `c.rbegin`、`c.rend()`。`c.begin()` 指向 `c` 的第一項，而 `c.end()` 指向 `c` 的最後一項的**後一項**。也就是說，`*c.end()`、`*c.rend()` 是不存在的，如果你這樣寫會造成不可預期的後果。

10.4 vector

vector 位於標頭檔 `<vector>` 中，vector 可以視為動態陣列的實現。一般的陣列在宣告時就確定了，然而 vector 的長度可以任意伸縮。

以下 vector 的常用語法 (假設變數名為 `v`)：

1. (建構式) `vector<T> v(size_type a, const T& b)`：一開始這個 `v` 會被 `b` 填滿，總共填 `a` 個。如果只有指定 `a`，那麼 `b` 會是 `T` 的預設值。如果 `a, b` 都沒有指定，那 `v` 會是一個空的 vector。建構複雜度 $O(a)$。
2. `v[i]`：`v` 中第 `i` 項，就把這個語法想成陣列就好。複雜度 $O(1)$。
3. `v.size()`：這個函式會回傳 `v` 目前的長度。複雜度 $O(1)$。
4. `v.push_back(T a)`：在 `v` 的尾端加一個 `a`。均攤複雜度 $O(1)$。
5. `v.pop_back()`：刪除 `v` 的最末項。如果 `v` 是空的，會發生不可預期的結果。複雜度 $O(1)$。
6. `v.clear()`：清空 `v`。複雜度 $O(size)$。原本 `v` 的空間會被保留，不會釋放掉。
7. `v.resize(size_type a, const T& b)`：強制將 `v` 的長度變為 `a`。如果比原本短，則將 `v` 原本的末段捨去，複雜度 $O(D(size-a))$，D 是解構 `T` 的時間。如果比原本長，在 `v` 的後面加 `b` 直到足夠為止（如果只有指定 `a`，那麼 `b` 是 `T` 的預設值），通常複雜度 $O(C(a-size))$，但如果需重新配置記憶體（reallocate），複雜度 $O(Ca)$，其中 C 是建構（複製）`T` 的時間。

8. `v.reserve(size_type n)`：預留放至少 n 個 `T` 的空間。如果需重新配置記憶體，複雜度 $O(size)$。如果 $n < size$，這個函數不造成任何影響。

vector 的迭代器屬於隨機存取 (Random Access) 迭代器。

比較需要講的是 vector 重新配置記憶體的耗時較長，所以如果能預先知道記憶體最多需要多少，就在一開始建構的時候開滿或者先 `reserve` 吧！(儘管對時間只有常數的影響)

另外，`vector<bool>` 有神奇優化成一個 bool 佔的空間只有 1 bit，是 `bool[]` 的 1/8。

10.5 string

string 位於標頭檔 `<string>` 裡，等價於 `basic_string<char>`（如果不知道這個，可能會看不懂編譯訊息）。string 的用法很像 `vector<char>`，但因字串太常使用了，所以有經過一些優化。除此之外，還有一些好用的東西（假設變數名為 `s`）：

1. `s = t`：如果 `t` 是一個 string 或是 C 式字串，`s` 會變得跟 `t` 一樣。複雜度不明，但通常是 $O(size_s + size_t)$。
2. `s += t`：如果 `t` 是一個 string 或是 C 式字串，在 `s` 的尾端加上 `t`。複雜度通常是 $O(size_s + size_t)$。
3. `s.c_str()`：這個函式會回傳跟 `s` 一樣的 C 式字串。在 C++11 中保證複雜度為 $O(1)$。
4. `s`（比較大小或相等的符號）`t`：回傳比較 `s` 跟 `t` 字典序的結果。通常複雜度是 $O(max(size_s, size_t))$。
5. `cin >> s`：輸入字串至 `s`，直到讀到空白字元。
6. `cout << s`：輸出字串 `s`。
7. `getline(cin, s, char c)`：輸入字串至 `s`，直到讀到字元 `c`。未指定時，`c` 是換行符號 (`'\n'`)。

string 的迭代器屬於隨機存取迭代器。

除了列舉的之外，vector 有的 string 都有。除此之外，s.size() 有個同義的函式 `s.length()`，可能是怕人打錯才多加了這個函式吧。

順帶一提，關於 string 是不是「容器」其實也有些爭議，但它有大部分容器的性質，所以在此仍然將其歸類為容器。

10.6 deque

deque 位於標頭檔 `<deque>` 裡。deque 可以視為可以在最前面加東西、刪東西的 vector，除此之外它就是 vector 了。

假設變數名是 `d`，想要移除第一項，就用 `d.pop_front()`。想要在前面加一個東西 `a`，就用 `d.push_front(a)`。這兩個函式不會使迭代器失效，但會改變 deque 的下標。

雖說功能比 vector 強，但代價是時間和空間幾乎翻倍，所以沒事別用 deque。

10.7 list

list 位於標頭檔 `<list>` 裡。list 是個「雙向鏈結（doubly linked）結構」，也就是說對於 list 中的每一項，都可以 $O(1)$ 知道它的前一項和後一項。如此做的好處是，如果我要一次性加入一堆東西，只需要 $O(1)$ 的代價，比 vector 優。然而代價是，存取第 i 項的複雜度是 $O(i)$，因此沒有內建的下標運算。同樣列舉一下常用語法（假設變數名為 `s`）：

1. （建構式）`list<T> s(size_type a, const T& b)`：同 vector。
2. `s.push_front(T a)`、`s.push_back(T a)`、`s.pop_front()`、`s.pop_back()`：同 deque。
3. `s.size()`：回傳 `s` 中有幾項。相當需要注意的是 C++98 中這個函式的複雜度只有保證 $O(size)$，C++11 則保證 $O(1)$。
4. `s.empty()`：回傳一個 bool，代表 `v` 是否是空的。複雜度 $O(1)$。
5. `s.insert(iterator p, T a)`：在 `p` 指的那一項前面插入一個 `a` 並回傳一個指向 `a` 的迭代器。複雜度 $O(1)$。
6. `s.insert(iterator p, size_type n, T a)`：在 `p` 指的那一項前面插入 `n` 個 `a`。複雜度 $O(n)$。
7. `s.erase(iterator p)`：把 `p` 指的那項刪掉並回傳指向之後那項的迭代器。注意刪完之後 `p` 就失效了。複雜度 $O(1)$。
8. `s.erase(iterator first, iterator last)`：把 [first,last) 指到的東西全砍光光，回傳 `last`。複雜度和砍掉的東西個數呈線性關係。

9. `s.splice(iterator p, list& x, iterator first, iterator last)`：`first` 和 `last` 是 `x` 的迭代器。這個函式會把 [`first`,`last`) 指到的東西從 x 中移除並加到 p 指的那項前面。注意到 `x` 會因為這個函式而改變。如果沒有指定 `last`，那只將 `first` 從 `x` 刪去並加入 `s`。如果 `first` 和 `last` 都沒指定，那會將 `x` 中所有東西移到 `s` 中使 `x` 變為空的。複雜度是轉移元素個數的線性。

list 的迭代器屬於雙向迭代器。

可以看出 list 最大的功能是可以用 $O(1)$ 的代價進行一些別的容器做不到的事(`insert`、`erase`)。然而 list 最大的問題是所佔空間過於龐大，而且實用性低。C++11 中多了一個 forward_list 以改善空間過大的問題，代價是迭代器變成單向迭代器（也因此沒有 `reverse_iterator`)。

10.8 stack

stack 位於標頭檔 `<stack>` 裡。可以把它想像成一疊書，每次可以放一本書在最上面，也可以從最上面拿一本書走。簡單來說就是秉持著「後進先出」（LIFO）的精神。

stack 這個模板需要的型別參數有兩個：`T` 和 `C`，其中 `T` 是內容物的型別，而 `C` 是採用的容器。為了方便改造，stack 對 C 有些要求：要有 `empty`、`size`、`back`、`push_back`、`pop_back` 這些函式，而在內建的容器中能夠勝任這角色的有 vector、deque 和 list。

stack 常用的語法列舉如下（假設變數名為 `s`)：

1.（建構式）`stack<T, C> s(C& a)`：`s` 一開始會有一份 `a` 的複製品。如果沒有指定 `C` 的話，`C` 是 `deque<T>`。如果沒有指定 `a` 的話，`s` 一開始會是空的。複雜度 $O(size)$。2. `s.size()`、`s.empty()`：同 vector。3. `s.top()`：存取最後一個進入 `s` 的元素，即「一疊書中最上面的那一本」。複雜度 $O(1)$。4. `s.push()`：將一個元素加入 s 中。複雜度 $O(1)$。5. `s.pop()`：將最後一個進入 s 的元素移除。複雜度 $O(1)$。

至於 `C` 應該要選用什麼，個人建議是 `vector<T>`。而且事實上，stack 能做的事 vector 都能做到，所以平常用 vector 就可以了。只是用 stack 可以增加程式的可讀性。

10.9 queue

queue 位於標頭檔 `<queue>` 裡。可以把它想像成排隊等著結帳的人群，要嘛有新的人來排在隊伍的尾端，要嘛最前面有一個人結完帳要走了。簡單來說，就是秉持著「先進先出」（First In, First

Out; FIFO）的精神。

queue 這個模板同樣需要兩個型別參數 `T` 和 `C`，跟 stack 一樣。不過不同的是，queue 對 C 的要求不太一樣：要有 empty、size、front、back、push_back、pop_front 這些函式，而在內建的容器中能夠勝任這角色的只有 deque 和 list。

queue 常用的語法列舉如下（假設變數名為 `q`）：

1. （建構式）`queue<T, C> q(C& a)`：q 一開始會有一份 `a` 的複製品。如果沒有指定 `C` 的話，`C` 是 `deque<T>`。如果沒有指定 `a` 的話，`q` 一開始會是空的。複雜度 $O(size)$。
2. `q.size()`，`q.empty()`：同 vector。
3. `q.front()`：存取第一個進入 `q` 的元素，即「隊伍中最前面的人」。複雜度 $O(1)$。
4. `q.back()`：存取最後一個進入 `q` 的元素，即「隊伍最末端」。複雜度 $O(1)$。
5. `q.push()`：將一個元素加入 `q` 中。複雜度 $O(1)$。
6. `q.pop()`：將第一個進入 `q` 的元素移除。複雜度 $O(1)$。

建議 `C` 就依照預設的即可。不過和 stack 一樣，要用 queue 不如用 deque。

10.10 priority_queue

`<queue>` 中其實還藏有一威力極大的適配器：priority_queue。priority_queue 利用幾個內建函式實現「二叉堆」（binary heap）結構，一個在任何時候維持最頂的元素永遠都是最大的資料結構。priority_queue 雖然實作容易，但應用廣泛，每次都手刻一次會很浪費時間。以後會見到更多 priority_queue 的應用。

priority_queue 這個模板需要三個型別參數 `T`、`Con` 和 `Cmp`。`T` 代表內容物的型別（需可以比較大小），`Con` 代表使用的容器，而 `Cmp` 代表使用的比大小的依據（之後會再詳細說明）。Con 的要求是擁有隨機存取迭代器以及 `empty`、`size`、`front`、`push_back`、`pop_back` 這些函式，而滿足這些條件的內建函式有 vector（預設值）和 deque。在詳細地認識 `Cmp` 之前，只需要知道 `Cmp` 是 `less<T>`（預設值）時 `priority_queue` 是最大堆，而是 `greater<T>` 的時候 `priority_queue` 是最小堆。

priority_queue 常用的語法列舉如下（假設變數名為 `pq`）：

1. (建構式) `priority_queue<T, Con, Cmp> pq`：建構一個空的 `pq`。複雜度 $O(1)$。
2. (建構式) `priority_queue<T, Con, Cmp> pq(iterator first, iterator last)`：建構一個 pq，內含 `[first,last)` 指到的東西，這裡 iterator 可以是任何迭代器。複雜度 $O(size)$。
3. `pq.size()`、`pq.empty()`：同 vector。
4. `pq.top()`：回傳 `pq` 中最大（最小）的元素（無法修改）。複雜度 $O(1)$。
5. `pq.push(T a)`：將 `a` 加入 `pq` 中。複雜度 $O(\log size)$。
6. `pq.pop()`：將 `pq` 中最大（最小）的元素移除。複雜度 $O(\log size)$。

比較需要注意的是在建構的時候直接餵內容物的時間複雜度是 $O(size)$，而一個一個 push 進去的時間複雜度是 $O(size \log size)$。雖然一般情況下沒什麼差，不過有必要的時候請記得 priority_queue 的建構式可以減少複雜度。

10.11 pair

pair 位於標頭檔 `<utility>` 裡面（注意沒有 <pair> 這個標頭檔）。pair 其實很單純，就是把兩個（可能不同型別的）變數綁在一起，變成一個變數。為此，pair 需要接收兩個型別，分別代表一對的第一項和第二項的型別。

pair 常用的語法列舉如下（假設變數名為 `p`）：

1. (建構式) `pair<A, B> p(A a, B b)`：建構一個把型別 `A` 和型別 `B` 綁在一起的 `p`，其中第一項是 `a`，第二項是 `b`。
2. `p = s`：如果 `s` 也是同型別的 pair，把 `p` 變得跟 `s` 一樣。
3. `p` (比較大小或相等的符號) `s`：如果 `s` 也是同型別的 pair，先比第一項，如果一樣再比第二項。
4. `p.first`、`p.second`：存取第一項、第二項。

在使用 pair 時，常常會使用到一個非成員函式 `make_pair`。`make_pair` 的好處在於，你不用特別指明第一項和第二項的型別，編譯器會自行幫你解析。用法是 `make_pair(A a, B b)`，函式會回傳一個 `pair<A, B>`，其中第一項是 `a`，第二項是 `b`。

另外，C++11 開始允許可變長度的模板，所以也有 pair 的推廣版 tuple（在標頭檔 `<tuple>` 中）。有興趣的可以自己看看。

10.12 set

set 位於標頭檔 `<set>` 裡。set 實現了自平衡二元查找樹，用白話文來講，可以 $O(\log n)$ 插入、刪除或查詢一個值有沒有在其中。特別的是，裡面的元素不會重複，因此我們會把元素的值稱為鍵值 (key)。

set 的常用語法列舉如下（假設變數名為 `s`）：

1. （建構式）`set<K> s`：建構一個空的 `s`。複雜度 $O(1)$。
2. `s.size()`、`s.empty()`：同 vector。
3. `s.insert(K k)`：在 `s` 中放入一個鍵值為 `k` 的元素。如果本來就有了，什麼事都不會做。複雜度 $O(\log size)$。
4. `s.erase(iterator first, iterator last)`：刪除 `[first,last)`。如果沒指定 `last`，只刪除 `first`。只刪除一個時，均攤複雜度 $O(1)$，刪除多個時複雜度是刪除個數的線性。
5. `s.erase(K k)`：刪除所有鍵值為 `k` 的元素並回傳刪除的項數（在 set 中只會是 0 或 1）。複雜度 $O(\log size)$。
6. `s.find(K k)`：回傳指向鍵值為 `k` 的元素的迭代器。如果沒有這種東西，回傳 `m.end()`。複雜度 $O(\log size)$。
7. `s.count(K k)`：回傳有幾個鍵值為 `k` 的元素（在 set 中只會是 0 或 1）。複雜度 $O(\log size)$。
8. `s.lower_bound(K k)`：回傳迭代器指向第一個鍵值大於等於 `k` 的項。複雜度 $O(\log size)$。
9. `s.upper_bound(K k)`：回傳迭代器指向第一個鍵值大於 `k` 的項。複雜度 $O(\log size)$。

set 的迭代器是雙向迭代器。`set::iterator` 會由小迭代到大，`set::reverse_iterator` 則會由大迭代到小。比較容易被忽視的是 set 的迭代器在遞增和遞減的時候，理論上不只是均攤複雜度，連複雜度也是 $O(1)$。但有些實作只保證均攤複雜度。

要注意的是 `lower_bound` 和 `upper_bound` 的微妙差別：一個是大於等於、一個是大於。用途通常是找鍵值在 `[l,u)` 的那些項，找法是 `[s.lower_bound(l), s.upper_bound(u))`。記住，C++ 通常是左閉右開區間。

10.13 map

map 位於標頭檔 `<map>` 裡。map 可以當成 set 的每一個元素都對應到另一個值，也就是可以用 $O(\log n)$ 插入、刪除或尋找一個鍵值對應的值。因此，map 這個模板需要兩個型別參數 `K` 和 `T`，其中 `K` 是鍵值的型別（需要可以比大小），而 `T` 代表對應到的值的型別。

另外，map 中的每一個元素其實是 `pair<K, T>`，所以迭代器指向的東西是一個 pair，第一項是鍵值，第二項是對應的值。

map 常用的語法列舉如下（假設變數名為 `m`）：

1. （建構式）`map<K, T> m`：建構一個空的 `m`。複雜度 $O(1)$。
2. `m.size()`、`m.empty()`、`m.erase(iterator first, iterator last)`、`m.erase(K k)`、`m.find(K k)`、`m.count(K k)`、`m.lower_bound(K k)`、`m.upper_bound(K k)`：同 set。
3. `m[k]`：存取鍵值 `k` 對應的值。如果 `k` 沒有對應的值，會插入一個元素，使 `k` 對應到預設值並回傳之。複雜度 $O(\log size)$。
4. `m.insert(pair<K, T> k)`：如果沒有鍵值為 `k.first` 的值，插入一個鍵值為 `k.first` 的值對應到 `k.second`，並回傳一個 pair，`first` 是指向剛插入的元素的迭代器、`second` 是 true；如果已經有了，回傳一個 pair，`first` 是指向鍵值為 `k.first` 的元素的迭代器，`second` 是 false。複雜度 $O(\log size)$。

和 set 的迭代器一樣，map 的迭代器是雙向迭代器。

10.14 multiset, multimap

在 `<set>`、`<map>` 中分別還有 multiset 和 multimap。和前面大致相同，唯一的差別在於 multiset 和 multimap 中鍵值可以重複出現，不像 set 和 map 鍵值不能一樣。如此一來，`count` 和 `erase` 回傳的值便不一定是 0 或 1。此外，由於 multimap 中一個鍵值可能對應到許多不同的值，因此也不支援下標操作。

在 multiset 和 multimap 中有一個特別好用的函式 `equal_range(K k)`，會回傳一個 iterator 的 pair，第一項代表 `lower_bound(k)`，第二項代表 `upper_bound(k)`。這兩項迭代器之間的項就是那些鍵值是 `k` 的項。雖然這個函式 set 跟 map 也有，但在 set 和 map 中就顯得有點雞肋。

10.15 (C++11)unordered_(multi)set, unordered_(multi)map

在 C++11 以後，unordered 系列常常擔任優化掉 map 和 set $O(\log n)$ 複雜度的角色。

unordered_(multi)set 和 unordered_(multi)map 分別在標頭檔 **<unordered_set>** 和 **<unordered_map>** 裡。這四個模板需要的前一（二）個型別參數和 set（map）一樣，而接著是一個型別 Hash，代表要使用的雜湊函數的函數型別（之後會提）或指標。不過 C++11 有預設的內建型別（含任意型別的指標）的雜湊函數，所以除非情況特殊，你可以不用理會這一項。

你可以把 unordered 系列當成 (multi)map 和 (multi)set 來用。不過有幾點不同：

1. unordered 系列的迭代器為單向迭代器。
2. unordered 系列沒有將所有項依鍵值排序（這也是它名字的由來），因此迭代器在遍歷容器時不會依鍵值的大小順序遍歷。
3. 因為沒有排序，所以理所當然的沒有 lower_bound、upper_bound。
4. 比起 (multi)map 和 (multi)set，unordered 系列的期望複雜度少一個 log。

在宣告變數（建構式）時，你可以指定 bucket 至少有幾個。如果鍵值是內建型別而且你沒有指定 bucket 至少有幾個，那麼它會很耐斯地幫你搞定。

10.16 bitset

bitset 位於標頭檔 **<bitset>** 裡。你可以把它想像成有一堆 0 跟 1 的一個陣列，也就是說 bitset 的大小是固定的。但是它每一項只會用到 1 bit 的空間，而 bitset 的位元運算是被優化過的（雖然只限於同大小的 bitset），運作起來非常快速，對常數有極巨大的影響，同時也可以達到壓縮空間的效用，是常數優化的好幫手。

bitset 這個模板需要一個整數 **n** 作為模板的參數，代表 bitset 的長度。

bitset 的一些常用語法列舉如下（假設變數名為 **b**）：

1. （建構式）**bitset<N> b(a)**：用 **a** 初始化一個長度為 **N** 的 bitset。這裡 **a** 可以是 unsigned long、string 或 C 式字串。如果沒有指定 **a**，或者如果 **b** 有一些地方沒被 **a** 初始化，那些地方預設為 0。
2. **b.count()**：回傳 **b** 有幾個位元是 1。複雜度 $O(N)$。

3. `b.size()`：回傳 `b` 有幾個位元。複雜度 $O(1)$。
4. `b` （位元運算）：不管是一元還是二元的位元運算都可以。如果是兩個 bitset 的二元位元運算，兩個 bitset 的長度需一致。複雜度 $O(N)$。
5. `b[a]`：存取第 `a` 位。複雜度 $O(1)$。
6. `b.set()`：將所有位元設為 1。複雜度 $O(N)$。
7. `b.reset()`：將所有位元設為 0。複雜度 $O(N)$。
8. `b.flip()`：將所有位元的 0、1 互換（反白）。複雜度 $O(N)$。
9. `b.to_string()`：回傳一個字串和 `b` 的內容一樣。複雜度 $O(N)$。
10. `b.to_ulong()`：回傳一個 unsigned long 和 `b` 的內容一樣（在沒有溢位的範圍內）。複雜度 $O(N)$。

bitset 不是容器，而且它也沒有迭代器。

通常而言，如果要估計常數的話，相較於直接使用陣列，空間是 1/8、count 約是 1/6、位元運算約是 1/30。當然這些都不是絕對的。

要注意的是，上述的複雜度沒有明文規定，不過通常是如此。

Chapter 11

亂數 Random Number

在寫程式時候，我們有時候會需要生成一定範圍內的亂數來完成某些任務。例如：寫一個抽籤程式、模擬擲骰子的過程、寫一個換位子的程式等等。亂數甚至還可以用來估算圓周率和解線性規劃問題等等。

在本章中，我們會講到要如何在程式中引入亂數。

11.1 亂數的用法

程式碼 11.1: 亂數的用法

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    for(int i=0; i<10; i++)
      cout << rand() % 1000 << endl;
}
```

大家可以在電腦上執行看看程式碼 11.1 。可能會發現每次的執行結果都相同。這顯然不是我們想要的，因為這樣就不是亂數了嗎。

所以要怎麼辦呢？我們可以看看程式碼 11.2

程式碼 11.2: srand()

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    srand(7122); // 設定亂數種子為 7122
    for(int i=0; i<10; i++)
        cout << rand() % 1000 << endl;
}
```

Hmmm, 雖然輸出的數字不一樣了，但是每次執行所輸出的數字都一樣呀。所以到底要怎麼辦呢？

程式碼 11.3: srand(time(NULL))

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main(){
    srand(time(NULL)); // 設定亂數種子為程式執行當下的時間
    for(int i=0; i<10; i++)
        cout << rand() % 1000 << endl;
}
```

可以發現在程式碼 11.3 中, 我們將亂數的種子設成當下的時間，這樣每次執行時，亂數的種子都不會一樣，隨機出來的數字也都不一樣。我們的目的也就暫時完成了。

但其實以上的用法是最簡單的作法，隨機出來的數字不會是完美的隨機。如果想要了解更好的隨機方法，可以看看C++11 random。

11.2 原理？

程式碼 11.4: rand(), srand() 的原始碼

```c
static unsigned long int next = 1;

int rand(void) // RAND_MAX assumed to be 32767
{
  next = next * 1103515245 + 12345;
  return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
  next = seed;
}
```

在程式碼 11.4 中，我們可以發現 `rand()` 是透過固定的運算產生下一個數字，而且亂數會有一個最大值 `RAND_MAX`。以這種方式產生的亂數稱為偽隨機亂數，但是一般來說，`rand()` 函數對於一般的任務已經足夠了。

Chapter 12

二分搜尋 Binary Search

在一群資料中搜尋某個資料，通常有**線性搜尋**和**二分搜尋**兩種。線性搜尋就是利用迴圈，從頭到尾的搜尋某個資料。對於沒有排序過的資料，線性搜尋是唯一的方法。但如果資料已經按照大小順序排列好了，我們就可以使用二分搜尋。使用二分搜尋 $(O(\log N))$ 的效率相較於線性搜尋 $(O(N))$ 好太多了。

在本章中，我們會提到要如何要怎麼寫基本的二分搜，也會介紹 C++ 內建的二分搜函數及相關的資料結構。

12.1 基本的二分搜

最簡單的二分搜是在一個**排好序的陣列**中尋找某個元素，找到了回傳 index，否則回傳找不到(例如-1)，這樣的二分搜好寫也不容易寫錯，以下是個範例。二分搜的重點在於 `left` 與 `right` 兩變數紀錄著搜尋範圍的左右邊界，每次取出中間位置來比較，結果是找到了或捨棄左半邊或捨棄右半邊。當 `left>right` 表示搜尋區間已經沒有了，這時離開迴圈，因為要找的元素不存在。若初始的區間範圍是 N，時間複雜度是 $O(\log N)$，原因是區間長度每次都減半。

程式碼 12.1: 基本的二分搜

```
#include <iostream>
using namespace std;
#define N 10
// binary search x between a[left..right]
```

```
int bsearch(int a[], int left, int right, int x) {
  while (left <= right) {
    int mid = (left + right) / 2;  // middle element
    if (a[mid] == x) return mid;
    if (a[mid] < x)
      left = mid + 1;  // search right part
    else
      right = mid - 1;  // search left part
  }
  return -1;
}
int main() {
  int p[N];
  int n = 10;
  for (int i = 0; i < n; i++) p[i] = rand() % 100;
  sort(p, p + n);
  for (int i = 0; i < n; i++) cout << p[i] << " ";
  cout << endl;
  printf("search %d ⇒ return %d\n", p[7], bsearch(p, 0, n - 1,
  ↪  p[7]));
  int t = rand() % 100;
  printf("search %d ⇒ return %d\n", t, bsearch(p, 0, n - 1, t));
  return 0;
}
```

12.2 C++ 的二分搜函數以及相關資料結構

其實在 C++ 中，提供了很多跟搜尋有關的函數和資料結構。這裡會介紹一些基本的用法。常用的二分搜函數有以下三種：

- `binary_search()`

- `lower_bound()`
- `upper_bound()`

`lower_bound()` 函數是用來找出第一個大於等於某個值的位置，而 `upper_bound()` 是用來找出第一個大於某個值的位置，`binary_search()` 只回傳是否有這個值。

和 `sort()` 函數相同，`lower_bound()` 允許傳入自定義的比較函數。基本上呼叫的時候以 `lower_bound(first, last, t);`，就可以搜尋在 `[first, last)` 範圍中，以二分搜搜尋第一個大於等於 t 的元素的位置。如果找不到時，就回傳 `last`。

在使用 `lower_bound()` 函數時，有幾點需要注意：

- 找到元素時，回傳的是那個元素的**位置**，如果想要得到索引值，還要再減去陣列的起始位置。
- 呼叫 `lower_bound()` 函數時，必須確定搜尋範圍是排好序的，如果你針對一個未排序的資料以 `lower_bound()` 進行二分搜，只會得到錯誤的結果。編譯器是不會告訴你資料沒有排序的。
- 搜尋範圍是傳統的左閉右開區間，也就是不含 `last`。
- 找不到時回傳 `last`，通常這個位置是超過陣列的範圍，所以沒確定找到時不可以直接去引用該位置的值。

程式碼 12.2 是 `lower_bound()` 函數的範例程式，執行一下就可以了 `lower_bound()` 函數是在幹嘛的。

程式碼 12.2: `lower_bound()` 的使用方法

```
1 // demo lower_bound for int array
2 // binary search the first >=x
3 #include <bits/stdc++.h>
4 using namespace std;
5 #define N 5
6 int main() {
7   int p[N] = {5, 1, 8, 3, 9};
8   int n = 5;
9   sort(p, p + n);
```

```
  for (int i = 0; i < n; i++) cout << p[i] << " ";
  cout << endl;
  for (int i = 0; i < 5; i++) {
    int t = i * 3;
    // search [first=p, last=p+n) to find the first >=t
    int ndx = lower_bound(p, p + n, t) - p;
    if (ndx < n)
      printf("The first >=%d is at [%d]\n", t, ndx);
    else  // return the last address if not found
      printf("No one >=%d\n", t);
  }
  // for vector
  vector<int> v(p, p + n);  // copy p to vector v
  for (int i = 0; i < 5; i++) {
    int t = i * 3;
    int ndx = lower_bound(v.begin(), v.end(), t) - v.begin();
    if (ndx < n)
      printf("The first >=%d is at [%d]\n", t, ndx);
    else  // return v.end() if not found
      printf("No one >=%d\n", t);
  }
  return 0;
}
```

Chapter 13

動態規劃 Dynamic Programming[1]

這一章介紹動態規劃 (Dynamic Programming, DP)。DP 名字中 Programming 的意思是指「以表格紀錄」，而非現在常用的「寫程式」，所以 DP 的意思是以變動的表格來求解的方法。DP 的應用很廣，變化很多，在學術界也發展的很早，在程式競賽也用的非常多，並且有很多困難而精妙的 (優化) 方法。

13.1 基本原理

13.1.1 基本思維與步驟

DP 與我們以後會提到的分治 (Divide and Conquer) 都有個相同之處：將問題劃分成許多的子問題，再由子問題的解合併成最後的解，其中子問題是指相同問題比較小的輸入資料。所以，設計 DP 的方法從找出遞迴式開始，設計 DP 的演算法通常包含下面幾個步驟：

1. 定義子問題
2. 找出問題與子問題之間的 (遞迴) 關係。
3. 找出子問題的計算順序來避免以遞迴的方式進行計算。

如果沒有進行第三個步驟，而只是以遞迴的方式寫程式，就會變成與第一章相同的純遞迴方式，純遞迴往往會遭遇到非常大的效率問題，所以也可以說 DP 就是要改善純遞迴的效率問題的技術。我們先介紹兩個簡單的問題，以問題來舉例說明會比較清楚。

[1]本章內容多參考吳邦一教授的《從 APCS 實作題檢測三級到五級》一書

熊熊上樓梯

熊熊上樓梯[2]，每步走一階或兩階，如果走到第 n 階有 $f(n)$ 種走法，輸入 n，計算 $f(n)$。例如 $n = 3$ 時，走法有 $1+1+1=3$ 或 $1+2=3$ 或 $2+1=3$，一共 3 種，$f(n) = 3$。

我們的問題是計算 $f(n)$。定義子問題：對 $i < n, f(i)$ 表示走到第 i 階的走法數。接著要找出 $f(n)$ 與子問題的遞迴關係，我們雖然沒辦法立刻寫出 $f(n)$ 是多少，但是我們只要去想，走到第 n 階的最後一步一定是走了一階或兩階 (沒有其他的可能)，而最後一步是一階與兩階的走法必然不同 (沒有重複多算)。最後走一階的走法數就是抵達第 $n-1$ 的走法數 $f(n-1)$，而最後一步走兩階的走法數就是抵達第 $n-2$ 階的走法數 $f(n-2)$，因此，我們得到遞迴關係式 $f(n) = f(n-1) + f(n-2)$，但是不要忘了以上推論是對於 $n > 2$ 時的狀況，遞迴的初始條件 $f(1)$ 與 $f(2)$ 要另外找，不難想一下就可以知道，所以完整的遞迴關係是：

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n > 2 \\ n & \text{otherwise} \end{cases} \tag{13.1}$$

如果把Equation 13.1寫成簡單的遞迴程式的話，會像是程式碼 13.1。

程式碼 13.1: 熊熊上階梯的簡單遞迴式

```cpp
#include <iostream>

using namespace std;

long long stair(int n) {
    if (n < 3) return n;
    return stair(n - 1) + stair(n - 2);
}

int main() {
    int n;
    cin >> n;
    cout << stair(n) << endl;
```

[2]本題跟 WLOJ 上面那題不太一樣，請不要直接複製講義上的程式碼

```
14
15      return 0;
16  }
```

這個程式可以跑，只要你不要輸入太大的數字。如果輸入 40 大概會看到螢幕停頓一下就跑出來，輸入 60 的話肯定可以去喝杯咖啡再看看跑出來沒；如果輸入 100，跑到你死了也跑不出來；如果輸入 200，那到地球毀滅也跑不出來。原因何在？這是一個純遞迴的程式，一個遞迴呼叫兩個，兩個就會呼叫四個，雖然 $f(n-1)$ 與 $f(n-2)$ 並不一樣，但它會接近 2^n，確切的時間複雜度大概是 $O(1.6n)$[3]。我們來把它改成 DP，所以要進行第三步驟：找出子問題的計算順序來避免以遞迴的方式進行計算。什麼樣的計算順序會讓我們可以不需要遞迴呢？原則上「只要讓遞迴式等號右邊出現的在左邊的之前先算，並且用表格把算過的記下來，就可以不需要遞迴了」。

以這一題為例，遞迴式 $f(n) = f(n-1) + f(n-2)$，如果 n 從小往大算，算到 $f(n)$ 的時候，$f(n-1)$ 與 $f(n-2)$ 都已經算過了，所以只要當初算完的時候有記下來，那就可以直接從表格中取出 $f(n-1)$ 與 $f(n-2)$ 來相加得到 $f(n)$，完全不需要用到遞迴。那麼如何用表格紀錄呢？這一題我們可以開一個陣列 `F[]`，以 `F[i]` 記錄 $f(i)$ 的值，程式就變成一個簡單的迴圈，沿著我們的計算順序 (n 從小到大)，逐步從表格中取出值，計算後存入表格，程式如程式碼 13.2。

程式碼 13.2: 熊熊上階梯的 DP 寫法

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      long long F[10000];
7      int n;
8      cin >> n;
9      F[1] = 1, F[2] = 2;
10     for (int i = 3; i <= n; i++){
11         F[i] = F[i - 1] + F[i - 2];
```

[3]如果想知道這個時間複雜度是怎麼算出來的話可以查查看主定理 (master theorem)

```
    }
    cout << F[n] << endl;

    return 0;
}
```

這個程式跑得很快，不過數字太大會溢位就是了，一般在考這一類題目的時候都要求輸出模某數的餘數。我們完成了一個 DP 算法的設計，看起來並不是太難。請再回想一下剛才走過的思維步驟：定義子問題、找遞迴關係、最後找計算順序與定義表格來避免遞迴。所以 DP 從遞迴開始，但以去除遞迴來完成，看來似乎有點難，但以這題來說又很簡單。其實，對於大部分的題目，尋找計算順序與表格都很簡單，最常出現的計算順序就是像這一題：由小到大，而表格就是用遞迴的參數來直接定義。真正難的是遞迴關係式的尋找。

路徑問題

有一個 $m \times n$ 的矩形方格，從左上角的格子出發，要到達右下角的格子，每一步只能向右或向下移動一格，若不同的路徑有 $g(m,n)$ 條，輸入 m 與 n，計算 $g(m,n)$。例如 $g(2,3) = 3$，如Figure 13.1。

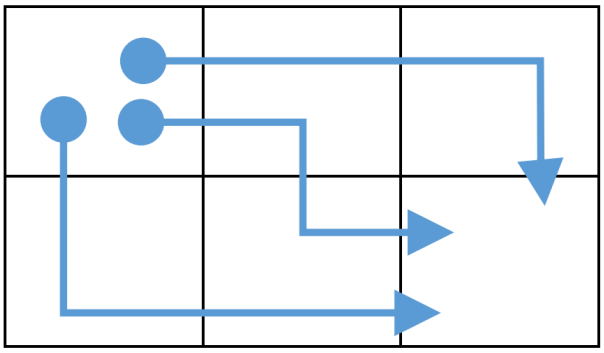

圖 13.1: 路徑問題

我們的問題是計算 $g(m,n)$。定義子問題：以 $g(i,j)$ 表示出發點走到 (i,j) 的走法數，以矩陣列與行的方式定義位置，出發點為 $(1,1)$，終點是 (m,n)。接著要找出 $g(i,j)$ 與子問題的遞迴關係，同樣的，我們不必立刻寫出 $g(i,j)$ 是多少，只要去想，最後一步一定向右或向下，也就是前一個位置一定是 $(i,j-1)$ 或 $(i-1,j)$，而且這兩類走法必然不同 (沒有重複多算)。根據定義，我們可以得到

$g(i,j) = g(i,j-1) + g(i-1,j)$ for i>1 and j>1；初始條件為 $g(i,j) = 1(i = 1 \text{ or } j = 1)$。得到遞迴式之後直接寫遞迴程式就是純遞迴，程式如程式碼 13.3，再一次，遞迴的程式非常沒有效率：

程式碼 13.3: 路徑問題的遞迴寫法

```cpp
#include <iostream>

using namespace std;

long long grid(int m, int n) {
    if (m == 1 || n == 1) return 1;

    return grid(m, n-1) + grid(m-1, n);
}
int main() {
    int m, n;
    cin >> m >> n;
    cout << grid(m, n) << endl;
    return 0;
}
```

要把純遞迴改成 DP，就要找計算順序，讓遞迴關係式右邊的出現在左邊的之前，本題的遞迴式為：$g(i,j) = g(i,j-1) + g(i-1,j)$，簡單觀察可以得知，只要 i 與 j 都從小到大就可以滿足需要，所以我們可以採取「由上而下，由左而右」的順序，當然也可以「由左而右，由上而下」，甚至可以沿著 $i+j$ 由小而大的斜線的順序，有點自找苦吃就是了。至於表格，最直覺的方式就是將 $g(i,j)$ 記錄在一個二維陣列 `G[i][j]` 中。以下是程式，跑得很快，時間複雜度是 $O(mn)$，而遞迴版本的複雜度則是組合數 $\binom{m+n-2}{m-1}$，因為這一題的答案其實有數學解，是排列組合的題目。請不需要擔心數學解這件事，通常程式考試與競賽都不出有數學解的題目，以這題來說，只要將某些格子變成禁止通行或者做一些變化，就難以用數學的方式求解了，畢竟程式考試主要考的是程式技巧。

程式碼 13.4: 路徑問題的 DP 寫法

```cpp
#include <iostream>

using namespace std;

int main() {
    long long G[50][50];
    int m, n;
    cin >> m >> n;

    for (int i=1; i<=m; i++)
        G[i][1]=1;

    for (int j=1; j<=n; j++)
        G[1][j]=1;

    for (int i=2; i<=m; i++)
        for (int j=2; j<=n; j++)
            G[i][j] = G[i-1][j] + G[i][j-1];

    cout << G[m][n] << endl;

    return 0;
}
```

13.1.2 狀態轉移

DP 子問題的遞迴關係式也有人稱為「狀態轉移」，這個名詞大概是從有限狀態機 (數位電路設計) 來的。子問題可以看成一個狀態，目前的狀態是根據之前的那些狀態，這樣的轉換關係就稱作狀態轉移，設計 DP 時，最重要的就是找出狀態轉移。之前我們將子問題看成「部分解」，所以是在找部分解與更小的解之間的關係，這兩個講法只是描述的方式不一樣，其實是同一回事，有的時候這樣想比較方便，有的時候那樣想比較直覺。雖然我們要在下一章會討論圖 (graph)，但提到狀態轉移，用圖的術語比較好說明。我們說狀態 A 是狀態 B 的前置狀態 (predecessor)，如果狀態 B 是由 A 轉移過

來的，一個狀態可能有多個前置狀態，通常一個狀態的值是由它的前置狀態的值做某些計算得到。如果我們把每個狀態想像每一個點，若 A 是 B 的前置狀態，則由 A 劃一個箭頭指到 B，我們會得到一個所謂的圖 (graph)，這個圖就是這個 DP 的狀態轉換圖，在圖論上來說，它必然是個有方向但沒有環路的圖 (directed acyclic graph, DAG)，所謂環路是指從某點出發，沿著箭頭前進，最後又走到出發點。遞迴關係不會無限遞迴，所以必然沒有環路。我們來看看前面兩個簡單的問題的狀態轉移。小朋友上樓梯問題的子問題是走到第 i 階，關係式 $f(n) = f(n-1) + f(n-2)$，也就是每個狀態由它的前兩個狀態轉移而來。

第二個問題是方格圖的問題，遞迴式是：$g(m,n) = g(m-1,n) + g(m,n-1)$，每一個格子是一個點 (狀態)，前置狀態是他的左方與上方。

對於一個 DAG，我們一定可以將它的所有點排成某個順序，在此順序中，每個箭頭都由前往後指，也就是說，每個點的前置點都在它之前，這樣的順序稱為拓樸順序 (topological sort)，我們的 DP 就是要沿著一個拓樸順序來進行。有件事情要提醒，本節說明的狀態轉移以及 DAG 與拓樸順序，是方便思考用的，在 DP 解題時通常並不需要把它建出來，也就是建在心中就可以了，不必建在程式中。

13.1.3 分類與複雜度

題目的分類往往有很多種，主要看目的何在而分類，這裡我們以便於理解與學習來分類。一個 DP 如果狀態有 $O(n^x)$ 而轉移式涉及 $O(n^y)$ 個狀態，一般可稱為 $xDyD$ 的 DP，例如小朋友上樓梯是 $1D0D$ 的，因為有 n 個要算，每次算 $f(i)$ 只涉及 2 個 $f(i-1)$ 與 $f(i-2)$。方格路徑的問題則是 $2D0D$，因為有 n^2(假設 $m = n$) 個要算，每次只需要 2 個 (左方與上方)。當然也有一些 DP 不在這個分類中。

一個 $xDyD$ 的 DP，如果沒有優化，很明顯的，時間複雜度就是 $O(n^{x+y})$，一般來說若 $y = 0$ 時比較簡單，因為遞迴式單純，而且遞迴式找出來時，程式就已經差不多寫完了，因為套個迴圈依序計算就是了。$y = 1$ 的 DP 題目通常存在某種優化或需要某些資料結構幫忙，所以會比較難。下一節開始的題目分析，我們的會先介紹 $y = 0$，然後再介紹 $y = 1$ 的狀況。

13.1.4 Top-down memoization

前面講到設計 DP 算法的步驟，基本上要先找出遞迴式，然後找出計算順序來避免遞迴。這算是標準的 DP，也稱為 Bottom-up 的 DP。另外有一種實現 DP 的方式是不找計算順序，直接依照遞迴式寫遞迴，但是因為一定要避免遞迴的重複計算，所以採取了一些步驟，這種實現的方式稱為

top-down memoization 的 DP，其中 memoization 字典上應該沒有這個字，大概是為了這個技術自己創造的，這個字是由 memo(備忘錄，便籤小紙條) 變來的，這個方法基本上應該是打比賽或重實作的人上發展出來的 (偷懶) 方法，因為從理論的角度，除了程式好比較寫之外，這麼做沒有好處，但考試與比賽還真的就希望比較好寫。

用比較俏皮的講法，這個方法就是再找到遞迴式之後，直接寫成遞迴版本的程式，但要加上三個動作：

1. 開一個表格當作小抄，用來記錄計算過的結果，表格初值設為某個不可能的值，用以辨識是否曾經算過。
2. 在遞迴之前，先偷看一下小抄是否曾經算過，如果小抄上有答案 (已經算過)，則直接回傳答案。
3. 如果小抄上沒有，就遞迴呼叫，但計算完畢回傳前，要把它記在小抄上。

我們來看看前面介紹的兩個問題的遞迴版本如何改成 top-down DP。程式碼 ??是熊熊上樓梯，首先注意到它就是拿前面的遞迴版本來改的，修改的部分：第 5 行宣告了一個表格 `F[]`；在遞迴函數中第 8 行做了一個檢查是否 `F[n]>0` 的動作，如果是則直接回傳，這一題的數字都是正數，所以我們可以用 0 來表示沒算過；如果沒算過，第 9 行就真的遞迴呼叫去計算，然後存在表格中再回傳。

程式碼 13.5: 熊熊上樓梯的 Top-down memoization 寫法

```
1  #include <iostream>
2
3  using namespace std;
4
5  long long F[100] = {0}; // memo, 0 if not computed
6
7  long long stair(int n) {
8      if (F[n] > 0) return F[n]; // check memo
9      F[n] = stair(n-1) + stair(n-2); // record to memo
10     return F[n];
11 }
12 int main() {
```

```
    int n;
    cin >> n;
    F[1]=1, F[2]=2;
    cout << stair(n) << endl;

    return 0;
}
```

這個 “遞迴” 的程式跑的慢不慢呢？基本上跟 Bottom-up 跑得一樣神速。我們再來看第二題方格路徑的改法。下面是修改版的程式，其實每一個 top-down 的 DP 幾乎都是長的這個樣子。這裡我們故意用了一些與上一題稍微不同的修改方式，雖然這題也可以用陣列初值為 0 表示沒算過，但我們自己寫陣列的初始化 (設為-1)，因為有些題目可能有此需要。上一題我們把遞迴終點條件拿掉，替代以在主程式中設定起始點的表格值，這一題我們留著遞迴的終端條件，其實這些差異都不大。這個程式也是跑的跟 Bottom-up 版一樣的神速。

程式碼 13.6: 路徑問題的 Top-down memoization 寫法

```
#include <iostream>

using namespace std;

long long G[100][100]; // as memo

long long grid(int m, int n){
    if (m == 1 || n == 1)
        return 1;
    if (G[m][n] >= 0)
        return G[m][n]; // check memo
    // record before return
    return G[m][n] = grid(m, n - 1) + grid(m - 1, n);
}

```

```
int main(){
    int m, n;
    cin >> m >> n;
    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            G[i][j] = -1; // initial unknown

    cout << grid(m, n) << endl;

    return 0;
}
```

Top-down 的 DP 基本上是個偷懶的方法，我們的建議如下：如果計算順序不難找，還是寫迴圈版，不要過分偷懶，因為遞迴還是有成本與限制，尤其某些系統之下 (尤其架在視窗系統下)，遞迴的深度有一定的限制，超過限制會導致執行階段的錯誤。在計算順序比較不好找或者比較麻煩時，而且系統是允許的狀況下，就採用 top-down 吧！最常見的情形就是 Tree 的 DP，幾乎參加比賽的選手都寫遞迴的 DFS 版本，因為迴圈版的要找 bottom-up 順序，比較麻煩。不過這裡還是要建議：兩種都要會，可以偷懶的時候才偷懶。