AP325－APCS 實作題檢測從三級到五級

釋出版本紀錄：2020/6/24, v:0.1, chapter 0~1。

目錄

例題與習題目錄

註：例題習題中標示(APCS)者表示該題為從出現於過去考試的類似題，如有標註 subtask 表示這一個例(習)題的解法為當初考試的某一子題而非 100 分的解。

0. 教材說明與預備知識

0.1. 教材說明

這是一份針對程式上機考試與競賽的教材，特別是 APCS 實作考試，325 是 3-to-5 的意思，這份教材主要目標是協助已經具有 APCS 實作三級分程度的人能夠進步到 5 級分。

APCS 實作考試每次出 4 題，每題 100 分，三級分是 150~249 分，四級分 250~349 分，而 350 分以上是五級分。由於題目的難度排列幾乎都是從簡單到難，因此，三級分程度大概是會做前兩題，而要達到五級分大概是最多只能錯半題，所以可以簡單的說，要達到四五級就是要答對第三第四題。根據過去的考題以及官網公告的成績說明，前兩題只需要基本指令的運用，包括：輸入輸出、運算、判斷式、迴圈、以及簡單的陣列運用，而第三與第四題則涉及常見的資料結構與演算法。以往程式設計的課程大多只在大學，以一般大學程式設計教科書以及大學資訊科系的課程來看，第三四題所需要的技術大概包含程式設計課的後半段以及資料結構與演算法的一部份，這就造成學習者在尋找教材時的困難，因為要把資料結構與演算法的教科書內容都學完要花很多時間。另外一方面，APCS 實作考試的形式與題型與程式競賽相似，程式競賽雖然網路上可以找到很多教材與資源，但是範圍太廣而且難度太深，而且多半是以英文撰寫的，對三級分的程式初學者來說，很難自己裁剪。

這份教材就是以具有 APCS 實作題三級分的人為對象，來講解說明四五級分所需要的解題技術與技巧，涵蓋基礎的資料結構與演算法，它適合想要在 APCS 實作考試拿到好成績的人，也適合競賽程式的初學者，以及對程式解題有興趣的人。如果你是剛開始學習程式，這份教材應該不適合，建議先找一本基礎程式設計的教科書或是教材，從頭開始一步一步紮實的學習與做練習。

這份教材的內容與特色如下：

- 在這一章裡面，除了教材介紹外，也說明一些預備知識。下一章開始則依主題區分為：遞迴、排序與二分搜、佇列與堆疊、貪心演算法、分治、動態規劃、樹狀圖、以及圖的走訪。
- 對於每一個主題，除了介紹常用的技巧之外，著重在以例題與習題展現解題技巧。例題與習題除了包括該主題的一些經典題目，也涵蓋過去考題出現過的以及未來可能出現技巧。所有的例題與習題除了題目敘述的範例之外，都提供足夠強度的測試資料以方便學習者測試自己的程式。

- 所有的例題都提供範例程式，有些例題會給多個不同的寫法，習題則是給予適當的提示。由於最適合程式檢測與競賽的語言是 C++，所以教材中的範例程式都採用 C++。檢測與競賽並不必須使用物件導向的寫法，但是 C++的 STL 中有許多好用的函式庫，本教材中也適度的介紹這些常用的資料結構與庫存函式。以 APCS 的難度，雖然幾乎每一題都是不需要使用特殊的資料結構就可以寫得出來，但是使用適當的資料結構常常可以有更簡單的寫法，何況在一般程式競賽中，STL 往往是非要不可的。

- 本教材是針對考試與競賽，所以範例程式的寫法是適合考場的寫法，而非發展軟體的好程式風格。因為時間壓力，考場的程式寫法有幾個不好的特性包括：比較簡短、變數名稱不具意義、濫用全域變數。但因為本教材是要寫給學習者看的，所以也不會刻意簡短而且會加上適當的註解，也不會採用巨集(#define)方式來縮短程式碼，但是會教一些在考場可以偷懶的技巧。

本教材例題習題所附測資檔名的命名方式以下面的例子說明，例如檔名為

P_1_2_3.in

表示是例題 P_1_2 的第 3 筆輸入測資，而

P_1_2_3.out

則是它對應的正確輸出結果。測資都是純文字檔，採 unix 格式，Unix 格式與 Windows 文字檔的換行符號不同，但目前 Win 10 下的筆記本也可以開啟 unix 格式的文字檔。

0.2. 預備知識

這份教材假設讀者已經有了一點撰寫 C 或 C++程式的基礎，所以撰寫與編譯程式的工具(IDE 或者命令列編譯)就不做介紹。在這一節中要提出一些預備知識，這些知識對於程式考試與比賽時撰寫程式有幫助。在本教材中的範例程式都採用與 APCS 相同的編譯環境，簡單說 C++版本為 C++ 11，而編譯器優化為 O2，也就是編譯命令類似是：

```
g++ -O2 -std=c++11 my_prog.cpp
```

如果是使用 IDE 的人，記得在 compiler 設定的地方設定這兩個參數，例如 Code::Block 是在 settings 下面選 compiler 然後勾選相關的設定。

接下來我們將說明以下主題。

- 基本 C++模板與輸入輸出
- 程式測試與測試資料
- 複雜度估算

- 需要留意的事

0.2.1. 基本 C++模板與輸入輸出

C++與 C 一樣，在使用庫存函數前都必須引入需要的標頭檔，剛開始寫簡單程式的時候，我們通常只需要使用

#include <cstdio> // 或者 C 的<stdio.h>

但是隨著使用庫存函數的增加，往往要引入多個標頭檔，更麻煩的是記不住該引入那些標頭檔，這裡介紹一個萬用的標頭檔，我們建議以下列範例的形式來寫程式。注意前兩行，其中第一行讓我們不需要再引入任何其它的標頭檔，而第二行讓我們使用一些資料結構與庫存函數時不需要寫出全名。如果你目前不能了解它們的意義，建議先不要追根究柢，記起來就好了，將來再去了解。

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    // code here
    return 0;
}
```

輸入與輸出是程式中最基本的指令，C 最常用的輸入輸出是 scanf/printf，而 C++最常用的是 cin/cout。在 C++裡面當然也可以用 scanf/printf，相較之下，cin/cout 用起來比較簡單，不過它卻有個缺點，就是效率不佳，在資料量小的時候無所謂，但是當資料量很大時，往往光用 cin 讀入資料就已經 TLE(time limit exceed, 執行超過時限)。這不是太不合理了嗎？同樣的程式如果把 cin 換成 scanf 可能就不會 TLE 了。事實上如果上網查 cin TLE 會找到解方，這解方就是在程式的開頭中加上兩行，所以程式就像以下這樣：

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    // code here
    return 0;
}
```

至於原因呢？也是有點複雜，所以一開始只好硬記起來，如果有興趣了解，可以上網查一下。這書一開始就叫人硬記一些東西，實在不是好的學習方式，這實在也是不得已，如果一開始就解釋一大堆可能把初學者搞得頭昏眼花。如果不想記這兩行，還有個方式就是不要用 cin/cout，只用 scanf/printf 就沒有這個問題了，本書的範例大部分的時候都是採用 scanf/printf。

這裡要特別提醒一件事，假設你加了上面那兩行之後，就千萬不可以把 cin/cout 與 scanf/printf 混用，例如你在程式中同時有使用 cout 與 printf，那麼可能會發生一些不可預期的事：在程式中輸出的東西的先後順序可能會亂掉。簡言之，下面兩個方法選一個使用，但不要混用：

- 使用 scanf/printf，或者
- 使用 cin/cout 加上那兩行。

0.2.2. 程式測試與測試資料

一個程式寫出來之後必須經過測試，除了語法沒有問題之外，更重要的是它可以計算出正確的答案而且可以在需求的執行時間之內完成計算。要測試程式就是拿資料餵給程式，看看程式的執行結果與時間是否符合要求，我們在考場寫出來的程式，繳交後也是以這個方式來進行測試。要測試程式第一步就必須有測資(測試資料)，產生適合的測資不是一件簡單的事，對某些題目來說根本就是比解答還要困難。本教材的例題與習題都有提供測資，有些人可能未必了解測資的使用方法，所以以下做一些簡介。

所謂的測資是指一組輸入以及它對應的正確輸出答案。一支程式的標準行為就是讀入資料，進行計算，然後輸出結果。通常我們寫好一支程式在電腦上執行時，它就會等待輸入，此時我們可以鍵入測試用的輸入，等待它輸出結果，然後再比對答案。如果是在視窗環境下，我們也可以用複製貼上的方式將輸入資料拷貝過去，但這方法在資料量太大的時候就無法使用，另外有個缺點是這樣做無法量測程式的執行時間。

在 C/C++裡面有三個定義的系統輸入輸出裝置，分別是：

- stdin：標準輸入裝置(standard in)，預設是鍵盤。
- stdout：標準輸出裝置(standard out)，預設是螢幕。
- stderr：標準錯誤記錄裝置(standard error)，預設是螢幕。

我們在程式中執行 scanf/cin 這一類輸入指令時，就是到 stdin 去抓資料，而 printf/cout 這些指令時是將資料輸出至 stdout。而這些裝置其實都可以改變的，這就是輸入輸出的重新導向(I/O redirection)，我們介紹兩個方法來做。

如果是用 unix 環境下以命令列執行程式，則以下列方式可以將輸入輸出重導：

```
./a.out <test.in >test.out
```

其中 a.out 是執行檔的名稱，在後面加上 <test.in 的意思就是將 test.in 當作輸入裝置，也就是原本所有從鍵盤讀取的動作都會變成從 test.in 檔案中去讀取。而 >test.out 則是將輸出重新導向至 test.out 檔案，原本會輸出至 stdout 的都會變成寫入檔案 test.out 中。輸入與輸出的重導可以只作其中任何一個，也可以兩者都做。

另外一個方法是在程式裡面下指令。我們可以在程式中以下列兩個指令來做 IO 重導：

```
freopen("test.in", "r", stdin);

freopen("test.out","w", stdout);
```

freopen 的意思是 file reopen，上述第一個指令的意思是將檔案 test.in 當作 stdin 來重新開啟，其中"r"的意思是 read 模式。第二個指令就是要求將檔案 test.out 當作 stdout 重新開啟，"w"的意思則是 write 模式。這兩個指令可以只執行其中一個，也可以兩個都做，也可以在一個程式裡面重導多次。

在很多場合，輸出的內容很少，當我們要測試程式的時候，我們只要做輸入重導，然後看輸出的結果是否與測資的輸出內容一樣就可以了。在某些場合，輸出也很大，或者想輸出訊息除錯，那就可以利用輸出重導把輸出的內容寫到檔案。如果要將程式的輸出與測資的輸出檔做比較，在 unix 下可以用 diff 來比較檔案內容。如果是 Windows 呢？可能要自己寫一支簡單的程式來比較兩個檔案。好在這份教材大部分的輸出都很小，用眼睛看就可以比較。另外要提醒一點，如果我們將輸出重新導向了，但還是想要在螢幕上輸出訊息的話，那可以利用 stderr 來輸出至螢幕，以下的例子展示如何使用。

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    int a, b;
    freopen("test.in","r",stdin);
    freopen("test.out","w",stdout);
    scanf("%d%d", &a, &b); // read from test.in
    fprintf(stderr,"a=%d, b=%d\n", a, b); // output to screen
    printf("%d", a+b); // output to test.out
    return 0;
}
```

測試程式另一個問題是程式的執行時間是否符合題目要求。檢測考試與競賽的題目需要講求效率，每一題都有執行時限(time limit)，這是指對於每一筆測資，繳交的程式必須在此時限內完成計算。在 unix 環境下我們可以用以下的指令來量測執行時間：

```
time a.out <test.in
```

其中 a.out 是執行檔。如果是在 windows 環境下，有些 IDE(如 Code::Blocks)在執行後會顯示執行時間，如果只要簡單的估計是可以，但要注意時間包含等待輸入的時間，所以通常要將輸入重導後顯示的時間才比較真實。下面介紹一個在程式中計算執行時間的方法。我們可以在程式中加上指令來計算時間，請看以下的範例：

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    clock_t t1, t2;
    t1=clock();
    // code here
    t2=clock();
    fprintf(stderr,"time from t1 to t2 = %f sec.\n", \
        (float)(t2-t1)/CLOCKS_PER_SEC);

    return 0;
}
```

這裡提醒一下，在考試時候時間寶貴，大部分的考試與比賽只有小的範例並未提供大測資，多數人也未必能多餘的時間去產生測資。只有在平常練習的時候，或者是考試時間非常充裕的時候，才適合去產生測資與精確量測時間。大部分的題目不需要去仔細的量測時間，只要估算複雜度就足夠了，複雜度就是下一節要談的主題。

0.2.3. 複雜度估算

要評估程式的執行效率，最準確的做法是量測時間，但是程式的執行時間牽涉的因素太多，而且不同資料量的時候會有不同表現，所以通常我們用複雜度來評估一支程式或演算法的效率。複雜度涉及一些定義與數學的計算，所以並不容易，這一節我們來講如何做簡易的複雜度估算，雖然只是講簡易的部分，但這些知識幾乎足以面對大部分常見的狀況。

複雜度估算方式

複雜度以 O(f(n)) 來表示，念作 big-O，也確實一定要大寫，小寫的 o 有不同的意義，其中習慣上以 n 來表示資料量，而 f(n) 是 n 的函數，複雜度代表的意義是「當資料量為 n 時，這個程式 (演算法) 的執行時間 (執行指令數) 不超過 f(n) 的某個常數倍」。對於常見的問題，資料量大小就是輸入的資料量，例如輸入整數的個數，或者輸入字串的長度，常見的複雜度有：O(n)、O(nlog(n))、$O(n^2)$、$O(2^n)$ 等等。Big-O 的計算與表示有下面幾個原則：

- 根據定義，不計算常數倍數，所以我們只說 O(n) 而不會說 O(3n) 或 O(5n)。因為常數倍數不計，所以 log(n) 不必指明底是 2 或 10。
- 兩個函數相加的 Big-O 等於比較大的函數。也就是說，若當 n 足夠大的時候 f(n)≥g(n)，則 O(f(n)+g(n)) = O(f(n))。如果一個程式分成兩段，一段的複雜度是 O(f(n))，另外一段是 O(g(n))，則整個程式的複雜度等於複雜度比較大的那一段。
- 如果某一段程式的複雜度是 O(f(n)) 而這段程式執行了 g(n) 次，則複雜度為 O(f(n)×g(n))。
- 程式的複雜度通常會因為輸入資料不同而改變，即使是相同的資料量，對某些資料需要的計算量少，對某些資料的計算量大。一般的複雜度指的是 worst-case 複雜度，也就是最難計算的資料所需的時間。

來看一些常見的例子。下面這段程式是一個迴圈跑 n 次，每一次作一個乘法與一個加法，所以複雜度是 O(2×n)=O(n)。(事實上每次迴圈還需要做 i++與 i<n 兩個指令，所以應該是 4n，反正都是 O(n))

```
for (int i=0; i<n; i++) {
    total += a[i]*i;
}
```

下面這段程式在計算一個陣列中有多少的反序對，也就是有多少 (i,j) 滿足

i<j 而 a[i]>a[j]。

程式有兩層迴圈，所以內層的 if 指令一共被執行 C(n,2)=n(n-1)/2 次，而 if 指令做一個比較與一個可能的加法，所以是 O(1)，整個來看複雜度是 $O(n(n-1)/2)=O(n^2)$，因為只需要看最高項次。

```
for (int i=0; i<n; i++) {
    for (int j=i+1; j<n; j++)
        if (a[j]<a[i])
            inversion++;
```

```
}
```

接下來看一個線性搜尋的例子，在一個陣列中找到某個數字。在下面的程式中，迴圈執行的次數並不一定，如果運氣好可能第一個 a[0]就是 x，而運氣不好可能需要跑到 i=n 時才確定找不到。因為複雜度要以 worst-case 來看，所以複雜度是 O(n)。

```
for (i=0; i<n; i++) {
    if (a[i] == x)
        break;
}
if (i<n)
    printf("found\n");
else printf("not found\n");
```

程式的結構大致是迴圈或遞迴，迴圈的複雜度通常依照上面所說複雜度乘法的原則就可以計算，但遞迴複雜度也有一套分析的方法，但牽涉一些數學也比較困難。另外有的時候也需要使用均攤分析(amortized analysis)，在後面的章節中，我們會碰到一些例子，屆時會視需要說明。

由資料量大小推估所需複雜度

在解題時，題目會說明資料量的大小與執行時限(time limit)，我們可以根據這兩個數字來推估這一題需要的時間複雜度是多少。同樣的題目，複雜度的需求不同，就是不同的難度。在 APCS 與大部分高中的程式比賽中，通常採取 IOI 模式，也就是一個題目中會區分子題(或稱子任務)，最常見的子題區分方式就是資料量的大小不同。

對於簡單題(例如 APCS 的第一第二題)，通常是沒有效率的問題，所以複雜度並不重要，但是，對於比較要求計算效率的第三第四題，會估算題目所需的複雜度，對思考解答就非常的重要。在一般的考試與競賽中，執行時限的訂定方式有兩種：多筆測資的執行時間限制或者是單筆測資的執行時間限制。APCS 採取 IOI 模式，所以大部分的時候都是指單一筆測資的執行時限。

程式的執行時間很難精確估算，因為每一秒可以執行的指令數牽涉的因素很多，例如：硬體速度、作業系統與編譯器、是否開啟編譯器優化、以及指令類型等等。當要估算複雜度時，建議可以一秒 10^6~10^7 的指令數量來估計，所以，假設執行一筆測資的時限為 1 秒，我們可以得到下面常見的複雜度推估：

n	超過 1 萬	數千	數百	20~25	10

複雜度	O(n) or O(nlog(n))	$O(n^2)$	$O(n^3)$	$O(2^n)$	O(n!)

舉例來說，如果有一題的資料量上限為 5 萬，那麼這一題需要的複雜度就是 O(n)或 O(nlog(n))；如果資料上限是 200，那麼需求的複雜度可能是 $O(n^3)$。

那麼，可不可能發生題目敘述中說資料量上限 100000，但是測資中的資料量只有 1000 呢？如果出題者想要的複雜度是 O(n)或 O(nlog(n))，那是測資出弱了，這是在高水準的考試與比賽中不應該出現的情形，因為它會不利於有能力者而造成不公平。如果出題者想要的複雜度只是 $O(n^2)$，那更糟糕了，出題者自己的程式都過不了宣稱的資料。實際考試與比賽中，照理說是不會出現這些狀況，但是也不能排除出題失誤的可能。

複雜度需要留意的幾件事

在本節最後，我們要提出幾個複雜度容易誤解與需要被注意的地方。

- 常用到的庫存函式 sort/qsort 的複雜度可以用 O(nlog(n))來看，雖然理論上 worst-case 也許不是。
- 理論上 Big-O 是指上限，而未必是緊實的上限(tight bound)。一個程式的複雜度如果是 O(n)，你說它是 $O(n^2)$理論上也沒有錯，但一般來說，我們會講盡可能緊實的上限，但並非每一個程式的緊實上限都能被準確計算。
- 複雜度有無限多種，但如果簡單來看，指數函數上升的速度遠遠高過多項式函數，因此一個演算法的複雜度如果是指數函數，通常能解的資料量大小就很小。在探討一個問題是否容易解時，指的是它是否有多項式複雜度的解，這裡的多項式函數只要指數是常數就可以，不一定要整數常數，例如 $O(n^{2.5})$也視作多項式複雜度。
- 複雜度是輸入資料量的函數，而且評估的是當資料量趨近於無限大時的行為。輸入資料量通常是資料的個數(數字個數，字串長度)，但是如果涉及大數運算時，必須以輸入資料的二進位編碼長度當做資料量大小。例如，對於一個輸入的正整數，我們可以用下列簡單的試除法來檢驗它是否是質數：

```
for (i=2; i*i<=n; i++)
    if (n%i == 0) break;
if (i*i > n) printf("prime\n");
else printf("not a prime\n");
```

這個程式的複雜度是多少？$O(\sqrt{n})$是沒錯，但它是多項式複雜度嗎？非也，這裡的資料量是多少？輸入一個數，所以是 1？那就算不出複雜度了。輸入一個數字必須

輸入它的二進位編碼，所以資料量的大小是 log(n)，因此，上面的程式其實是指數函數複雜度，令 L=log(n)，它的複雜度是 O($2^{L/2}$)。其實道理不難懂，所謂在一般情形下，指的是數字的運算可以在常數個指令完成的狀況，但電腦 CPU 的 bit 數為有限長度，當數字過大時，一個加法或乘法就必須軟體的方式來完成，也就不是常數個指令可以完成的。再舉一個例子，考慮以下問題：

Problem Q：輸入一個正整數 n，請計算並輸出 1+2+…+n 的值。

用最直接的方法跑一個迴圈去做連加的總和，這樣的程式複雜度是 O(n)，大家都知道等差級數的和可以用梯形公式計算，因此，我們可以寫出下面的程式：

```
scanf("%d", &n);
printf("%d\n", n*(n+1)/2);
```

這個程式的複雜度是多少？O(1)，沒錯吧？是。那麼，我們可以說：「Problem Q 存在 O(1)複雜度的解」嗎？答案卻是否，當 n 趨近無限大時，計算一個乘法或加法就不能在 O(1)時間內完成。也就是說這個程式的複雜度是 O(1)但它並不能完全解 Q。

現實中並不存在無窮大，所有的題目都有範圍，有人也可以說，在理論上，只要界定了有限範圍，任何題目都可以在 O(1)解決，因為即使複雜度高達 O(2^{10000})也是常數，只是要算到地老天荒海枯石爛而已。這麼講好像複雜度理論沒有用？其實不是的，在大部分的場合，n 在合理的範圍時就已經達到理論上的趨近無窮大，Big-O 也都可以讓我們正確的估算程式的效率，理論不是沒用，只是不能亂用。我們唯一要注意的是，Big-O 忽略常數，但是實際使用時，不可完全忽視常數帶來的影響，特別如果有可能是很大的常數。

0.2.4. 需要留意的事

這一節中提出一些需要注意的事，程式解題講求題目與解答的完整完善，或者可以說測資往往都很刁鑽，只要符合題目敘述的各種狀況都需要考慮，此外，為了測試效率，常常需要大的測資，對於程式解題比較沒有經驗的人，需要留意一些容易犯的錯誤。

整數 overflow

C++可以用來表示整數的資料型態目前有 char, short, int 與 long long int(也可以只寫 long long)，每種型態有它可以表示的範圍，實際的範圍可以查技術文件，每個型態也還可以指定為 unsigned 來表示非負的整數並且讓範圍再多一個位元。

解題程式通常不太需要過度節省記憶體，所以整數通常只會用 int 與 long long，目前在大多數系統上，前者是 32-bits 而後者是 64-bits。變數如果超過可以表示的範圍(不管太大或太小)，就會發生 overflow(溢位)的錯誤，這種錯誤不是語法上的，所以編譯器無法幫你得知，有些語法上可能的溢位，編譯器會給予警告，但是很多人寫程式的時候就是習慣不看警告的，因此溢位帶來的錯誤往往很難除錯。

請看以下的程式，因為 p 的值在 2^{31} 以內，(a*b)%p 的結果必然是整數的範圍之內，但是 b = (a*b)%p; 這個指令的結果卻是錯的，發生了溢位的錯誤。我們要了解，這樣一行 C 的指令實際的運算過程是：先計算(a*b)，再除以 p 取餘數，最後才把結果存入 b。但是當計算 a*b 時，因為兩個運算子都是 int，所以會以 int 的型態來做計算，也就發生了 overflow，後面就沒救了。如同範例程式中顯示的，解決的方法有兩個，一個改成 b = ((LL)a*b)%p，其中(LL)是要求做型態轉換成 LL，這樣就不會 overflow 了，另外一種偷懶的方式是乾脆就使用 long long 的資料型態來處理，缺點是多佔一點記憶體，在一般的解題場合不太需要省記憶體，所以是很多人偷懶所採取的方法。

```
#include <bits/stdc++.h>
using namespace std;
#define N 100010
typedef long long LL;
int main() {
    int a, b, p=1000000009;
    a = p-1, b = p-2;
    b = (a*b)%p; // overflow before %
    printf("%d\n",b);
    // correct
    a = p-1, b = p-2;
    b = ((LL)a*b)%p; // type casting to LL
    printf("%d\n",b);
    // or using LL
    LL c = p-2;
    c = (a*c)%p; // auto type-casting
    printf("%lld\n",c);

    return 0;
}
```

另外一個類似的情形經常發生在移位運算，如果我們寫

```
long long x = 1<<40;
```

預期得到 2^{40}，但事實上在存到 x 之前就溢位了，因為 1 和 40 都會被當 int 看待，正確的寫法是

```
long long x = (long long)1<<40;
```

類似的錯誤也發生在整數轉到浮點數的時候，例如

```
int a=2, b=3;
float f = b/a;
```

有些人這樣寫期待 f 會得到 1.5 的結果，事實不然，因為在將值存到 f 之前，b/a 是兩個整數相除，所以答案是 1，存到 f 時轉成浮點數，但捨去的小數部分就像是到了殯儀館才叫醫生，來不及了。

再提醒一次，Overflow 是很多人不小心容易犯的錯，而且很難除錯。這裡也提供一些除錯時的技巧，找出問題與印出某些資訊是除錯最主要的步驟，下面兩個指令有時可以派上用場

```
assert(判斷式); // 若判斷式不成立，則中斷程式，並且告知停在此處
fprintf(stderr,"…", xxx);//用法跟 printf 一樣，輸出至 stderr
```

有些 overflow 的錯誤可以藉著偵測是否變成負值找出來，我們在重要的地方下 assert(b>=0);可以偵測 b 是否因為溢位變成負值。

浮點數的 rounding error

跟整數一樣，浮點數在電腦中也有它能表示的範圍以及精準度。如果把下面這段程式打進電腦執行一下，結果或許會讓你驚奇。

```
#include <cstdio>
int main() {
    printf("%.20f\n", 0.3);
    double a=0.3, b=0.1+0.2;
    printf("%.20f %.20f\n", a, b);
    return 0;
}
```

浮點數儲存時會產生捨位誤差(rounding error)，其原因除了儲存空間有限之外，二進制的世界與十進制一樣也存在著無限循環小數。事實上，0.3 在二進制之下就是個無限循環小數，因此不可能準確的儲存。

因為浮點數存在捨位誤差，不同的計算順序也可能導致不同的誤差，數學上的恆等式並不完全適用在浮點數的程式裡面。所以，一般解題程式的考試與競賽常常避免出浮點數的題目，因為在裁判系統上會比較麻煩，但有時還是可以看到浮點數的程式題目。無論如何，不管是否是競賽程式或是撰寫一般的軟體，在處理浮點數時，都必須對捨位誤差有所認識。一個必須知道的知識就是，浮點數基本上不應該使用==來判斷兩數是否相等，而應該以相減的絕對值是否小於某個允許誤差來判斷是否相等。

判斷式的 short-circuit evaluation

下面是一個很簡單常見的程式片段，這段程式想要在一個陣列中找尋是否有個元素等於 x，我們在判斷式中除了 a[i]!=x 之外，也寫了陣列範圍的終止條件 i<5，但事實上這個程式是錯的，有的時候沒事，在某些環境下有的時候會導致執行錯誤，原因是條件式內的兩個條件順序寫反了！

```
#include <cstdio>
int main() {
    int a[5]={1,2,3,4,5}, i=0, x=6;
    while (a[i]!=x && i<5)
        i++;
    printf("%d\n", i);
    return 0;
}
```

正確的寫法是

```
while (i<5 && a[i]!=x)
```

這是唬弄人吧？誰不知道在邏輯上(A && B)與(B && A)是等價的呢？在程式的世界裡真的不一樣。當程式執行到一個判斷式的時候，它必須計算此判斷式的真偽，對於(A && B)這樣由兩個條件以「&&」結合的判斷式，會先計算第一個 A，如果 A 為真，再計算 B 是否為真；如果 A 為假，已知整個判斷式必然是假，所以並不會去計算 B。以上面的例子來說，當 i 是 5 的時候，對於(i<5 && a[i]!=x)，會先計算 i<5，如發現不成立就不會去計算 a[i]!=x。但如果像程式裡面反過來寫，就會先計算 a[i]!=x，因為 i=5，就發生陣列超出範圍的問題。

對於布林運算式，只有在第一個條件不足以判斷整個的真偽時，才會去計算第二個。這樣的設計稱為 short-circuit evaluation。同樣的情形發生在(A || B)的狀況，若 A 為 true，就不會再計算 B 了。

除了對陣列超過範圍的保護，這樣的設計也用來避免發生計算錯誤，例如我們要檢查兩數 a/b 是否大於 x，若無法確保 b 是否可能為 0，我們就應該寫成

```
if (b!=0 && a/b>x)
```

這樣才能保護不會發生除 0 的錯誤。

編譯器優化

所謂編譯器優化是指，當編譯器編譯我們的程式成為執行檔的時候，會幫助將程式變得速度更快或是檔案變得更小。C/C++的優化有不同層級，目前多數考試與比賽規定的環境是第二級優化(-O2)。編譯器優化的原則是不會變動原來程式的行為，只會提升效率。如果我們的程式本來就寫得很好，那優化不會做太多事，但是如果本來的程式寫得很爛，它可能幫你做了很多事。利用編譯器優化偷懶也不是不可以，但是至少寫程式的人自己應該知道，否則一直有壞的習慣而不自知，碰到沒有優化或者優化不能幫忙的情況，就慘了。來看一個例子：

```
for (int i=1;i<strlen(s);i++) {
    if (s[i]=='t') cnt++;
}
printf("%d\n",cnt);
```

這個程式片段計算字串 s 中有多少't'，是很常見的片段也是經常見到的寫法。事實上這是很不好的寫法，因為依照 C 的 for 迴圈的行為，第一次進入迴圈以及每次迴圈到底要再次進入前，都會去檢查迴圈進入條件是否成立。也就是說，如果這麼寫的話，strlen(s)這個呼叫函數計算字串長度的動作會被執行 O(n)次，其中 n 是字串長度，因為計算一次 strlen(s)就是 O(n)，所以這個程式片段的時間是 $O(n^2)$！那麼為什麼很多人都這麼寫呢？如果字串很短的時候，浪費一點點時間是不被察覺的，另外一個原因是開了編譯器優化，在此狀況下，編譯器的優話功能會發現字串長度在迴圈中沒有被改變的可能，因此 strlen(s)的動作被移到迴圈之外，所以執行檔中 strlen(s)只有被執行一次，效率因此大幅提高。以下是正確的寫法：

```
for (int i=1, len=strlen(s); i<len; i++) {
    if (s[i]=='t') cnt++;
}
printf("%d\n",cnt);
```

如果有人覺得反正編譯器會優化，不懂也沒關係，有一天寫出下面這樣的程式可能就慘了。因為迴圈內有動到字串中的值，可能影響 strlen(s)，因此編譯器不會幫你優化。

```
for (int i=1; i<strlen(s); i++) {
    if (s[i]=='t') {
        cnt++;
        s[i]=ch;
    }
}
printf("%d\n",cnt);
```

最後我們在看一個優化的例子，請看下面的程式，猜猜看這個程式要跑多久。這個程式有個 10 萬的整數陣列，有個 i-迴圈計算有幾個 6，迴圈內還有個 jk 雙迴圈計算某個

叫 foo 的東西，依照複雜度計算這是 $O(n^3)$，所以要指令數量是 10^{15}，大概要跑很久吧。

```
#include <bits/stdc++.h>
#define N 100010
int main() {
    int a[N], i, j, k, n=100000;
    for (i=0; i<n; i++) a[i]=rand()%100;
    int cnt=0, foo=0;
    for (i=0; i<n; i++) {
        if (a[i]==6) cnt++;
        for (j=0;j<n;j++)
            for (k=0;k<n;k++)
                if (a[j]+a[k]<a[i])
                    foo++;
    }
    printf("%d\n", cnt);
    return 0;
}
```

如果編譯器開了優化，這個程式瞬間就跑完了；如果不開優化，那可真是跑很久。原因何在？如果你稍微修改程式，在結束前輸出 foo，那麼即使加了優化，也要跑很久。真相逐漸浮現，因為編譯器優化時發現，那個 jk 雙迴圈中計算的 foo 在之後並沒有被用到(jk 的值變化也沒有影像到後面)，所以編譯器把那個雙迴圈整個當作垃圾給丟掉了，在執行檔中根本沒有這段程式碼。

再次強調，利用優化來偷懶不是不行，但要知其所以然，知道何時不可偷懶，也知道正確的寫法。

1. 遞迴

本章介紹遞迴函數的基本方法與運用，也介紹使用窮舉暴搜的方法。

1.1. 基本觀念與用法

遞迴在數學上是函數直接或間接以自己定義自己，在程式上則是函數直接或間接呼叫自己。遞迴是一個非常重要的程式結構，在演算法上扮演重要的角色，許多的算法策略都是以遞迴為基礎出發的，例如分治與動態規劃。學習遞迴是一件重要的事，不過遞迴的思考方式與一般的不同，它不是直接的給予答案，而是間接的說明答案與答案的關係，不少程式的學習者對遞迴感到困難。以簡單的階乘為例，如果採直接的定義：

> 對正整數 n，fac(n)定義為所有不大於 n 的正整數的乘積，也就是
> fac(n) = 1 × 2 × 3 × … × n。

如果採遞迴的定義，則是：

> fac(1) = 1; and
> fac(n) = n × fac(n-1) for n>1。

若以 n=3 為例，直接的定義告訴你如何計算 fac(3)，而遞迴的定義並不直接告訴你 fac(3)是多少，而是
fac(3) = 3 × fac(2)，而
fac(2) = 2 × fac(1)，最後才知道
fac(1) = 1。

所以要得到 fac(3)的值，我們再逐步迭代回去，
fac(2) = 2 × fac(1) = 2 × 1 = 2，
fac(3) = 3 × fac(2) = 3 × 2 = 6。

現在大多數的程式語言都支援遞迴函數的寫法，而函數呼叫與轉移的過程，正如上面逐步推導的過程一樣，雖然過程有點複雜，但不是寫程式的人的事，以程式來撰寫遞迴函數幾乎就跟定義一模一樣。上面的階乘函數以程式來實作可以寫成：

```
int fac(int n) {
    if (n == 1) return 1;
    return n * fac(n-1);
}
```

再舉一個很有名的費式數列(Fibonacci)來作為例子。費式數列的定義：

```
F(1) = F(2) = 1;
F(n) = F(n-1) + F(n-2) for n>2.
```

以程式來實作可以寫成：

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

遞迴函數一定會有終端條件(也稱邊界條件)，例如上面費式數列的 F(1) = F(2) = 1 以及階乘的 fac(1) = 1，通常的寫法都是先寫終端條件。遞迴程式與其數學定義非常相似，通常只要把數學符號換成適當的指令就可以了。上面的兩個例子中，程式裡面就只有一個簡單的計算，有時候也會帶有迴圈，例如 Catalan number 的遞迴式定義：

$$C_0 = 1 \text{ and } C_n = \sum_{i=0}^{n-1} C_i \times C_{n-1-i} \text{ for } n \geq 1.$$

程式如下：

```
int cat(int n) {
    if (n == 0) return 1;
    int sum = 0;
    for (int i = 0; i<n; i++)
        sum += cat(i) * cat(n-1-i);
    return sum;
}
```

遞迴程式雖然好寫，但往往有效率不佳的問題，通常遞迴函數裡面只呼叫一次自己的效率比較不會有問題，但如果像 Fibonacci 與 Catalan number，一個呼叫兩個或是更多個，其複雜度通常都是指數成長，演算法裡面有些策略是來改善純遞迴的效率，例如動態規劃，這在以後的章節中再說明。

遞迴通常使用的時機有兩類：

- 根據定義來實作。
- 為了計算答案，以遞迴來進行窮舉暴搜。

以下我們分別來舉一些例題與習題。

1.2. 實作遞迴定義

例題 P-1-1. 合成函數(1)

令 f(x)=2x-1, g(x,y)=x+2y-3。本題要計算一個合成函數的值，例如 f(g(f(1),3))=f(g(1,3))=f(4)=7。

Time limit: 1 秒

輸入格式：輸入一行，長度不超過 1000，它是一個 f 與 g 的合成函數，但所有的括弧與逗號都換成空白。輸入的整數絕對值皆不超過 1000。

輸出：輸出函數值。最後答案與運算過程不會超過正負 10 億的區間。

範例輸入：

```
f g f 1 3
```

範例輸出：

```
7
```

題解：合成函數的意思是它的傳入參數可能是個數字也可能是另外一個函數值。以遞迴的觀念來思考，我們可以將一個合成函數的表示式定義為一個函式 eval()，這個函式從輸入讀取字串，回傳函數值。其流程只有兩個主要步驟，第一步是讀取一個字串，根據題目定義，這個字串只有 3 種情形：f, g 或是一個數字。第二步是根據這個字串分別去進行 f 或 g 函數值的計算或是直接回傳字串代表的數字。至於如何計算 f 與 g 的函數值呢？如果是 f，因為它有一個傳入參數，這個參數也是個合成函數，所以我們遞迴呼叫 eval()來取得此參數值，再根據定義計算。如果是 g，就要呼叫 eval()兩次取得兩個參數。 以下是演算法流程：

```
int eval() // 一個遞迴函式，回傳表示式的值
    讀入一個空白間隔的字串 token;
    if token 是 f then
        x = eval();
        return 2*x - 1;
    else if token 是 g then
        x = eval();
        y = eval();
        return x + 2*y - 3;
    else // token 是一個數字字串
        return token 代表的數字
end of eval()
```

程式實作時，每次我們用字串的輸入來取得下一個字串，而字串可能需要轉成數字，這可以用庫存函數 atoi()來做。

```
// p 1.1a
#include <bits/stdc++.h>
int eval(){
    int val, x, y, z;
    char token[7];
    scanf("%s", token);
    if (token[0] == 'f') {
        x = eval();
        return 2*x - 1;
    } else if (token[0] == 'g') {
        x = eval();
        y = eval();
        return x + 2*y -3;
    } else {
        return atoi(token);
    }
}

int main() {
    printf("%d\n", eval());
    return 0;
}
```

atoi()是一個常用的函數，可以把字串轉成對應的整數，名字的由來是 ascii-to-int。當然也有其它的方式來轉換，這一題甚至可以只用 scanf()就可以，這要利用 scanf()的回傳值。我們可以將 eval()改寫如下，請看程式中的註解。

```
// p_1_1b
int eval(){
    int val, x, y, z;
    char c;
    // first try to read an int, if successful, return the int
    if (scanf("%d",&val) == 1) {
        return val;
    }
    // otherwise, it is a function name: f or g
    scanf("%c", &c);
    if (c == 'f') {
        x = eval(); // f has one variable
        return 2*x-1;
    } else if (c == 'g') {
        x = eval(); // g has two variables
        y = eval();
        return x + 2*y -3;
    }
}
```

下面是個類似的習題。

習題 Q-1-2. 合成函數(2) (APCS201902)

令 f(x)=2x-3；g(x,y)=2x+y-7；h(x,y,z)=3x-2y+z。本題要計算一個合成函數的值，例如 h(f(5),g(3,4),3)=h(7,3,3)=18。

Time limit: 1 秒

輸入格式：輸入一行，長度不超過 1000，它是一個 f, g, 與 h 的合成函數，但所有的括弧與逗號都換成空白。輸入的整數絕對值皆不超過 1000。

輸出：輸出函數值。最後答案與運算過程不會超過正負 10 億的區間。

範例輸入：

```
h f 5 g 3 4 3
```

範例輸出：

```
18
```

每個例題與習題都若干筆測資在檔案內，請自行練習。再看下一個例題。

例題 P-1-3. 棍子中點切割

有一台切割棍子的機器，每次將一段棍子會送入此台機器時，機器會偵測棍子上標示的可切割點，然後計算出最接近中點的切割點，並於此切割點將棍子切割成兩段，切割後的每一段棍子都會被繼續送入機器進行切割，直到每一段棍子都沒有切割點為止。請注意，如果最接近中點的切割點有二，則會選擇座標較小的切割點。每一段棍子的切割成本是該段棍子的長度，輸入一根長度 L 的棍子上面 N 個切割點位置的座標，請計算出切割總成本。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 N 與 L。第二行有 N 個正整數，依序代表由小到大的切割點座標 p[1]~p[N]，數字間以空白隔開，座標的標示的方式是以棍子左端為 0，而右端為 L。N ≤ 5e4，L< 1e9。

輸出：切割總成本點。

範例輸入：

```
4 10
1 2 4 6
```

範例輸出：

```
22
```

範例說明：第一次會切割在座標 4 的位置，切成兩段 [0, 4], [4, 10]，成本 10；

[0, 4] 切成 [0, 2] 與 [2, 4]，成本 4；

[4, 10] 切成 [4, 6] 與 [6, 10]，成本 6；

[0, 2] 切成 [0, 1] 與 [1, 2]；成本 2；

總成本 10+4+6+2 = 22

P-1-3 題解：棍子切斷後，要針對切開的兩段重複做切割的動作，所以是遞迴的題型。因為切點的座標值並不會改變，我們可以將座標的陣列放在全域變數中，遞迴函數需要傳入的是本次切割的左右端點。因為總成本可能大過一個 int 可以存的範圍，我們以 long long 型態來宣告變數，函數的架構很簡單：

```
// 座標存於 p[]
// 遞迴函式，回傳此段的總成本
long long cut(int left, int right) {
    找出離中點最近的切割點 m;
    return p[right]-p[left] + cut(left,m) + cut(m,right);
}
```

至於如何找到某一段的中點呢？離兩端等距的點座標應該是

```
x = (p[right] + p[left])/2
```

所以我們要找某個 m，滿足 p[m-1] < x <= p[m]，然後要找的切割點就是 m-1 或 m，看這兩點哪一點離中點較近，如相等就取 m-1。這一題因為數值的範圍所限，採取最簡單的線性搜尋即可，但二分搜是更快的方法，因為座標是遞增的。以下看實作，先看以線性搜尋的範例程式。

```
 // p_1_3a, linear search middle-point
#include <cstdio>
#define N 50010
typedef long long LL;
LL p[N];

// find the cut in (left,right), and then recursively
LL cut(int left, int right) {
    if (right-left<=1) return 0;
```

```
    LL len=p[right]-p[left], k=(p[right]+p[left])/2;
    int m=left;
    while (p[m]<k) m++; // linear search the first >=k
    if (p[m-1]-p[left] >= p[right]-p[m]) // check if m-1 is better
        m--;
    return len + cut(left, m) + cut(m, right);
}

int main() {
    int i, n, l;
    scanf("%d%d", &n, &l);
    p[0]=0; p[n+1]=l; // left and right ends
    for (i=1; i<=n; i++) scanf("%lld", &p[i]);
    printf("%lld\n",cut(0, n+1));
    return 0;
}
```

主程式中我們只需做輸入的動作，我們把頭尾加上左右端的座標，然後直接呼叫遞迴函數取得答案。如果採用二分搜來找中點，我們可以自己寫，也可以呼叫 C++ STL 的庫存函數。二分搜的寫法通常有兩種：一種 (比較常見的) 是維護搜尋範圍的左右端，每次以中點來進行比較，縮減一半的範圍。在陣列中以二分搜搜尋某個元素的話，這種方法是不會有甚麼問題，但是二分搜應用的範圍其實很廣，在某些場合這個方法很容易不小心寫錯。這裡推薦另外一種二分搜的寫法，它的寫法很直覺也不容易寫錯。

```
// 跳躍法二分搜
// 假設遞增值存於 p[]，在 p[s]~p[t]找到最後一個 < x 的位置
k = s; // k 存目前位置, jump 是每次要往前跳的距離，逐步所減
for (jump = (t - s)/2; jump>0; jump /= 2) {
    while (k+jump<t && p[k+jump]<x) // 還能往前跳就跳
        k += jump;
}
```

唯一要提醒的有兩點：第一是要先確認 p[s]<x，否則最後的結果 m=s 而 p[m]>=x；第二是內迴圈要用 while 而不能只用 if，因為事實上內迴圈做多會做兩次。

我們來看看應用在這題時的寫法，以下只顯示副程式，其他部分沒變就省略了。

```
LL cut(int left, int right) {
    if (right-left<=1) return 0;
    int m=left;
    LL k=(p[right]+p[left])/2;
    for (int jump=(right-left)/2;jump>0; jump>>=1) {
        while (m+jump<right && p[m+jump]<k)
            m+=jump;
    }
    if (p[m]-p[left] < p[right]-p[m+1])
        m++;
    return p[right]-p[left] + cut(left, m) + cut(m, right);
```

```
}
```

我們也可以呼叫 C++ STL 中的函數來做二分搜。以下程式是一個範例。
呼叫 lower_bound(s,t,x)會在[s,t)的區間內找到第一個>= x 的位置，回傳的是位置，所以把它減去起始位置就是得到索引值。

```
// 偷懶的方法，加以下這兩行就可以使用 STL 中的資料結構與演算法函數
#include <bits/stdc++.h>
using namespace std;
#define N 50010
typedef long long LL;
LL p[N];

// find the cut in (left,right), and then recursively
LL cut(int left, int right) {
    if (right-left<=1) return 0;
    LL k=(p[right]+p[left])/2;

    int m=lower_bound(p+left, p+right,k)-p;
    if (p[m-1]-p[left] >= p[right]-p[m])
        m--;
    return p[right]-p[left] + cut(left, m) + cut(m, right);
}
```

有關二分搜在下一章裡面會有更多的說明與應用。

習題 Q-1-4. 支點切割 (APCS201802)

輸入一個大小為 N 的一維整數陣列 p[]，要找其中一個所謂的最佳切點將陣列切成左右兩塊，然後針對左右兩個子陣列繼續切割，切割的終止條件有兩個：子陣列範圍小於 3 或切到給定的層級 K 就不再切割。而所謂最佳切點的要求是讓左右各點數字與到切點距離的乘積總和差異盡可能的小，也就是說，若區段的範圍是[s,t]，則要找出切點 m，使得 $|\sum_{i=s}^{t} p[i]\times(i-m)|$ 越小越好，如果有兩個最佳切點，則選擇編號較小的。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 N 與 K。第二行有 N 個正整數，代表陣列內容 p[1] ~ p[N]，數字間以空白隔開，總和不超過 10^9。N ≤ 50000，切割層級限制 K<30。

輸出：所有切點的 p[ ]值總和。

範例輸入：

```
7 3
2 4 1 3 7 6 9
```

範例輸出：

```
11
```

提示：與 P_1_3 類似，只是找切割點的定義不同，終端條件多了一個切割層級。

習題 Q-1-5. 二維黑白影像編碼 (APCS201810)

假設 n 是 2 的冪次，也就是存在某個非負整數 k 使得 $n = 2^k$ 。將一個 n*n 的黑白影像以下列遞迴方式編碼：
如果每一格像素都是白色，我們用 0 來表示；
如果每一格像素都是黑色，我們用 1 來表示；
否則，並非每一格像素都同色，先將影像**均等劃分**為四個邊長為 n/2 的小正方形後，然後表示如下：先寫下 2，之後依續接上左上、右上、左下、右下四塊的編碼。

輸入編碼字串 S 以及影像尺寸 n，請計算原始影像中有多少個像素是 1。

Time limit: 1 秒

輸入格式：第一行是影像的編碼 S，字串長度小於 1,100,000。第二行為正整數 n，$1 \le n \le 1024$，中 n 必為 2 的冪次。

輸出格式：輸出有多少個像素是 1。

範例輸入：

```
2020020100010
8
```

範例輸出：

```
17
```


1.3. 以遞迴窮舉暴搜

窮舉(Enumeration)與暴搜(Brute Force)是一種透過嘗試所有可能來搜尋答案的演算法策略。通常它的效率很差，但是在有些場合也是沒有辦法中的辦法。暴搜通常是窮舉某種組合結構，例如：n 取 2 的組合，所有子集合，或是所有排列等等。因為暴搜

也有效率的差異，所以還是有值得學習之處。通常以迴圈窮舉的效率不如以遞迴方式來做，遞迴的暴搜方式如同以樹狀圖來展開所有組合，所以也稱為分枝演算法或 Tree searching algorithm，這類演算法可以另外加上一些技巧來減少執行時間，不過這個部份比較困能，這裡不談。

以下舉一些例子，先看一個迴圈窮舉的例題，本題用來說明，未附測資。

例題 P-1-6. 最接近的區間和

假設陣列 A[1..n]中存放著某些整數，另外給了一個整數 K，請計算哪一個連續區段的和最接近 K 而不超過 K。

(這個問題有更有效率的解，在此我們先說明窮舉的解法。)

要尋找的是一個連續區段，一個連續區段可以用一對駐標[i,j]來定義，因此我們可以窮舉所有的 1<=i<=j<=n。剩下的問題是對於任一區段[i,j]，如何計算 A[i..j]區間的和。最直接的做法是另外用一個迴圈來計算，這導致以下的程式：

```
// O(n^3) for range-sum
int best = K; // solution for empty range
for (int i=1; i<=n; i++) {
    for (int j=i; j<=n; j++) {
        int sum=0;
        for (int r=i; r<=j; r++)
            sum += A[r];
        if (sum<=K && K-sum<best)
            best = K-sum;
    }
}
printf("%d\n", best);
```

上述程式的複雜度是 $O(n^3)$。如果我們認真想一下，會發現事實上這個程式可以改進的。對於固定的左端 i，若我們已經算出[i,j]區間的和，那麼要計算[i,j+1]的區間和只需要再加上 A[j+1]就可以了，而不需要整段重算。於是我們可以得到以下 $O(n^2)$ 的程式：

```
// O(n^2) for range-sum
int best = K; // solution for empty range
for (int i=1; i<=n; i++) {
    int sum=0;
    for (int j=i; j<=n; j++) {
        sum += A[j]; // sum of A[i] ~ A[j]
        if (sum<=K && K-sum<best)
            best = K-sum;
    }
```

```
}
printf("%d\n", best);
```

另外一種 O(n²)的程式解法是利用前綴和(prefix sum)，前綴和是指：對每一項 i，從最前面一直加到第 i 項的和，也就是定義 $ps[i] = \sum_{j=1}^{i} A[j]$，前綴和有許多應用，基本上，我們可以把它看成一個前處理，例如，如果已經算好所有的前綴和，那麼，對任意區間[i,j]，我們只需要一個減法就可以計算出此區間的和，因為

$\sum_{r=i}^{j} A[j] = ps[j] - ps[i-1]$。

此外，我們只需要 O(n)的運算就可以計算出所有的前綴和，因為 ps[i]=ps[i-1]+A[i]。以下是利用 prefix-sum 的寫法，為了方便，我們設 ps[0] = 0。

```
// O(n^2) for range-sum, using prefix sum
ps[0]=0;
for (int i=1; i<=n; i++)
    ps[i]=ps[i-1]+A[i];
int best = K; // solution for empty range
for (int i=1; i<=n; i++) {
    for (int j=i; j<=n; j++) {
        int sum = ps[j] - ps[i-1];
        if (sum<=K && K-sum<best)
            best = K-sum;
    }
}
printf("%d\n", best);
```

接下來看一個暴搜子集合的例題。

例題 P-1-7. 子集合乘積

輸入 n 個正整數，請計算其中有多少組合的相乘積除以 P 的餘數為 1，每個數字可以選取或不選取但不可重複選，輸入的數字可能重複。P=10009，0<n<26。

輸入第一行是 n，第二行是 n 個以空白間隔的正整數。

輸出有多少種組合。若輸入為{1, 1, 2}，則有三種組合，選第一個 1，選第 2 個 1，以及選兩個 1。

time limit = 1 sec。

我們以窮舉所有的子集合的方式來找答案，這裡的集合是指 multi-set，也就是允許相同元素，這只是題目描述上的問題，對解法沒有影響。要窮舉子集合有兩個方法：迴

圈窮舉以及遞迴，遞迴會比較好寫也較有效率。先介紹迴圈窮舉的方法。

因為通常元素個數很有限，我們可以用一個整數來表達一個集合的子集合：第 i 個 bit 是 1 或 0 代表第 i 個元素在或不在此子集合中。看以下的範例程式：

```
// subset product = 1 mod P, using loop
#include<bits/stdc++.h>
using namespace std;

int main() {
    int n, ans=0;
    long long P=10009, A[26];
    scanf("%d", &n);
    for (int i=0;i<n;i++) scanf("%lld", &A[i]);
    for (int s=1; s< (1<<n); s++) { // for each subset s
        long long prod=1;
        for (int j=0;j<n;j++) { // check j-th bit
            if (s & (1<<j)) // if j-th bit is 1
                prod = (prod*A[j])%P; // remember %
        }
        if (prod==1) ans++;
    }
    printf("%d\n", ans);
}
```

以下是遞迴的寫法。為了簡短方便，我們把變數都放在全域變數。遞迴副程式 rec(i, prod)之參數的意義是指目前考慮第 i 個元素是否選取納入子集合，而 prod 是目前已納入子集合元素的乘積。

遞迴的寫法時間複雜度是 $O(2^n)$ 而迴圈的寫法時間複雜度是 $O(n*2^n)$。

```
// subset product = 1 mod P, using recursion
#include<bits/stdc++.h>
using namespace std;
int n, ans=0;
long long P=10009, A[26];
// for i-th element, current product=prod
void rec(int i, int prod) {
    if (i>=n) { // terminal condition
        if (prod==1) ans++;
        return;
    }
    rec(i+1, (prod*A[i])%P); // select A[i]
    rec(i+1, prod); // discard A[i]
    return;
}

int main() {
    scanf("%d", &n);
    for (int i=0;i<n;i++) scanf("%lld", &A[i]);
    ans=0;
    rec(0,1);
```

```
    printf("%d\n", ans-1); // -1 for empty subset
    return 0;
}
```

習題 Q-1-8. 子集合的和 (APCS201810, subtask)

輸入 n 個正整數，請計算各種組合中，其和最接近 P 但不超過 P 的和是多少。每個元素可以選取或不選取但不可重複選，輸入的數字可能重複。P<=1000000009，0 < n < 26。

Time limit: 1 秒

輸入格式：第一行是 n 與 P，第二行 n 個整數是 A[i]，同行數字以空白間隔。

輸出格式：最接近 P 但不超過 P 的和。

範例輸入：

```
5 17
5 5 8 3 10
```

範例輸出：

```
16
```

接著舉一個窮舉排列的例子。西洋棋的棋盤是一個 8*8 的方格，其中皇后的攻擊方式是皇后所在位置的八方位不限距離，也就是只要是在同行、同列或同對角線 (包含 45 度與 135 度兩條對斜線)，都可以攻擊。一個有名的八皇后問題是問說：在西洋棋盤上有多少種擺放方式可以擺上 8 個皇后使得彼此之間都不會被攻擊到。這個問題可以延伸到不一定限於是 8*8 的棋盤，而是 N*N 的棋盤上擺放 N 個皇后。八皇后問題有兩個版本，這裡假設不可以旋轉棋盤。

例題 P-1-9. N-Queen 解的個數

計算 N-Queen 有幾組不同的解，對所有 0 < N < 15。

(本題無須測資)

要擺上 N 個皇后，那麼顯然每一列恰有一個皇后，不可以多也不可以少。所以每一組解需要表示每一列的皇后放置在哪一行，也就是說，我們可以以一個陣列 p[N] 來表示一組解。因為兩個皇后不可能在同一行，所以 p[N] 必然是一個 0~N-1 的排列。我們可以

嘗試所有可能的排列，對每一個排列來檢查是否有任兩個皇后在同一斜線上(同行同列不需要檢查了)。

要產生所有排列通常是以字典順序(lexicographic order)的方式依序產生下一個排列，這裡我們不講這個方法，而介紹使用庫存函數 next_permutation()，它的作用是找到目前排列在字典順序的下一個排列，用法是傳入目前排列所在的陣列位置[s,t)，留意 C++庫存函數中區間的表示方式幾乎都是左閉右開區間。回傳值是一個bool，當找不到下一個的時候回傳 false，否則回傳 true。下面是迴圈窮舉的範例程式，其中檢查是否在同一條對角線可以很簡單的檢查 「abs(p[i]-p[j]) == j-i」。

```
#include<bits/stdc++.h>
using namespace std;

int nq(int n) {
    int p[14], total=0;
    for (int i=0;i<n;i++) p[i]=i; //first permutation
    do {
        // check valid
        bool valid=true;
        for (int i=0; i<n; i++) for (int j=i+1;j<n;j++)
            if (abs(p[i]-p[j])==j-i) { // one the same diagonal
                valid=false;
                break;
            }
        if (valid) total++;
    } while (next_permutation(p, p+n)); // until no-next
    return total;
}

int main() {
    for (int i=1;i<15;i++)
        printf("%d ",nq(i));
    return 0;

}
```

接著看遞迴的寫法，每呼叫一次遞迴，我們檢查這一列的每個位置是否可以放置皇后而不被之前所放置的皇后攻擊。對於每一個可以放的位置，我們就嘗試放下並往下一層遞迴呼叫，如果最後放到第 n 列就表示全部 n 列都放置成功。

```
// number of n-queens, recursion
#include<bits/stdc++.h>
using namespace std;

// k is current row, p[] are column indexes of previous rows
int nqr(int n, int k, int p[]) {
    if (k>=n) return 1; // no more rows, successful
    int total=0;
    for (int i=0;i<n;i++) { // try each column
```

```
        // check valid
        bool valid=true;
        for (int j=0;j<k;j++)
            if (p[j]==i || abs(i-p[j])==k-j) {
                valid=false;
                break;
            }
        if (!valid) continue;
        p[k]=i;
        total+=nqr(n,k+1,p);
    }
    return total;
}

int main() {
    int p[15];
    for (int i=1;i<15;i++)
        printf("%d ",nqr(i,0,p));
    return 0;

}
```

副程式中我們對每一個可能的位置 i 都以一個 j-迴圈檢查該位置是否被 (j,p[j]) 攻擊，這似乎有點缺乏效率。我們可以對每一個 (j,p[j]) 標記它在此列可以攻擊的位置，這樣就更有效率了。以下是這樣修改後的程式碼，這個程式大約比前一個快了一倍，比迴圈窮舉的方法則快了百倍。以迴圈窮舉的方法複雜度是 $O(n^2 \times n!)$，而遞迴的方法不會超過 O(n!)，事實上比 O(n!)快很多，因為你可以看得到，我們並未嘗試所有排列，因為在遞迴過程中，會被已放置皇后攻擊的位置都被略去了。

```
// number of n-queens, recursion, better
#include<bits/stdc++.h>
using namespace std;

// k is current row, p[] are column indexes of previous rows
int nqr(int n, int k, int p[]) {
    if (k>=n) return 1; // no more rows, successful
    int total=0;
    bool valid[n];
    for (int i=0;i<n;i++) valid[i]=true;
    // mark positions attacked by (j,p[j])
    for (int j=0; j<k; j++) {
        valid[p[j]]=false;
        int i=k-j+p[j];
        if (i<n) valid[i]=false;
        i=p[j]-(k-j);
        if (i>=0) valid[i]=false;
    }
    for (int i=0;i<n;i++) { // try each column
        if (valid[i]) {
            p[k]=i;
            total+=nqr(n,k+1,p);
        }
    }
```

```
    return total;
}

int main() {
    int p[15];
    for (int i=1;i<12;i++)
        printf("%d ",nqr(i,0,p));
    return 0;

}
```

以下是一個類似的習題。

習題 Q-1-10. 最多得分的皇后

在一個 n*n 的方格棋盤上每一個格子都有一個正整數的得分，如果將一個皇后放在某格子上就可以得到該格子的分數，請問在放置的皇后不可以互相攻擊的條件下，最多可以得到幾分，皇后的個數不限制。0 < n < 14。每格得分數不超過 100。

Time limit: 1 秒

輸入格式：第一行是 n，接下來 n 行是格子分數，由上而下，由左而右，同行數字以空白間隔。

輸出格式：最大得分。

範例輸入：

```
3
1 4 2
5 3 2
7 8 5
```

範例輸出：

```
11
```

說明：選擇 4 與 7。

(請注意：是否限定恰好 n 個皇后答案不同，但解法類似)

上面幾個例子都可以看到遞迴窮舉通常比迴圈窮舉來得有效率，有些問題迴圈的方法甚至很不好寫，而遞迴要容易得多，以下的習題是一個例子。

習題 Q-1-11. 刪除矩形邊界 – 遞迴 (APCS201910, subtask)

一個矩形的邊界是指它的最上與最下列以及最左與作右行。對於一個元素皆為 0 與 1 的矩陣，每次可以刪除四條邊界的其中之一，要以逐步刪除邊界的方式將整個矩陣全部刪除。刪除一個邊界的成本就是「該邊界上 0 的個數與 1 的個數中較小的」。例如一個邊界如果包含 3 個 0 與 5 個 1，刪除該邊界的成本就是 min{3,5} = 3。

根據定義，只有一列或只有一行的矩陣的刪除成本是 0。不同的刪除順序會導致不同的成本，本題的目標是要找到最小成本的刪除順序。0 < n < 14。每格得分數不超過 100。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 m 和 n，以下 m 行是矩陣內容，順序是由上而下，由左至右，矩陣內容為 0 或 1，同一行數字中間以一個空白間隔。m + n <=13。

輸出格式：最小刪除成本。

範例輸入：

```
3 5
0 0 0 1 0
1 0 1 1 1
0 0 0 1 0
```

範例輸出：

```
1
```

提示：將目前的矩陣範圍當作遞迴傳入的參數，對四個邊界的每一個，遞迴計算刪除該邊界後子矩陣的成本，對四種情形取最小值，遞迴的終止條件是：如果只有一列或一行則成本為 0。(本題有更有效率的 DP 解法)