AP325－APCS 實作題檢測從三級到五級

程設資結迷惘，
算法遞迴如夢，
冷月豈可葬編程，
柳絮靜待東風。

鄭愁予：「我打江南走過，那等在季節裏的容顏如蓮花的開落，東風不來，三月的柳絮不飛，你的心如小小寂寞的城，...」直到遇上了程式，.....。

釋出版本紀錄：2020/6/24, v:0.1, chapter 0~1。
2020/6/30, v0.2, chapter 2.

作者：吳邦一

目錄

例題與習題目錄

註：例題習題中標示(@@)符號者題目稍難，標示(*)者可能超過 APCS 考試範圍，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。

```
#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

「遞迴是大事化小，要記得小事化無。」

「暴力或許可以解決一切，但是可能必須等到地球毀滅。」

「遞迴搜尋如同末日世界，心中有樹而程式(城市)中無樹。」

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：

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

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 $\left|\sum_{i=s}^{t} p[i]\times(i-m)\right|$ 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(n³)。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(n²)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 $\mathrm{ps}[i] = \sum_{j=1}^{i} A[j]$，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

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

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)

2. 排序與二分搜

「完全無序，沒有效率；排序完整，增刪折騰；完美和完整不是同一回事。」
(sorted array vs. balanced tree)

「上窮碧落下黃泉，姐在何處尋不見，人間測得上或下，不入地獄，就是上天；
天上地下千里遠，每次遞迴皆減半，歷經十世終不悔，除非無姐，終能相見。」
(binary search)

本章介紹排序與二分搜的基本用法與應用，同時也介紹其推廣運用，包括快速冪與折半枚舉。從這一章開始，我們會介紹到一些 C++的 container，這些容器都是一些別人已經寫好的資料結構，對於 APCS 這樣程度的考試來說，不使用這些容器也都可以考到五級分，但是使用這些容器往往可以簡化我們的程式，所以我們還是介紹常用的一些使用方式。如果你對程式還不是很熟練覺得學太多新的東西會負擔太大，那麼可以暫時略過你覺得比較複雜的部份。此外，這些容器的完整使用說明其實還蠻複雜的，這份教材中只會介紹簡單與常用的功能，需要更完整的說明，網路上都有技術文件可以參考，只要搜尋 C++就可找到。

2.1. 排序

排序是將一群資料根據某種順序由小到大排列，最常見的順序就是數字的由小到大或是由大到小，當然也可能是字元字串或是其他使用者自訂的順序。排序是非常重要的程序，計算機科學在發展初期就研發了很多種的排序演算法，這些演算法目前依然常常用在教學上當作例子與學習教材。在這裡我們主要介紹它的應用技巧而非排序演算法，對排序演算法有興趣的人可以查網路。

排序通常有幾個運用的時機：

- 需要把相同資料排在一起
- 便於快速搜尋
- 做為其他演算法的執行順序，例如 Sweeping-line, Greedy, DP。

在考試與競賽的場合，最重要的是會使用庫存的排序函數，在 C 是 qsort()，在 C++是 sort()，這些庫存函數都非常的有效率，如果要自己寫出這麼有效率的排序需要花

費很多的時間還不一定能寫好。以下我們先介紹它的使用方式與考試競賽的使用技巧，qsort 的使用比較麻煩，我們建議使用 C++的 sort()。

最簡單使用 sort()的方式就是針對一群數字的排序，例如陣列中的整數，傳入的第一個參數代表排序的起始位置，第二個參數是排序範圍結束的**下一個位置**。其使用方式如下面的範例程式所示範，其中我們用了一個 random_shuffle()的函數，它是用來將某個範圍的順序弄亂的。請注意，C++中函數定義區間時，幾乎都是左閉右開區間，例如以上的 sort()與 random_shuffle()：

```
#include <bits/stdc++.h>
using namespace std;
#define N 10
void show(int a[], int n) {
    for (int i=0;i<n-1;i++)
       printf("%3d ",a[i]);
    printf("%3d\n",a[n-1]);
    return;
}

int main() {
    int a[N], n=10;
    for (int i=0;i<n;i++) a[i]=rand()%100;
    show(a,n);
    printf("Sort entire array\n");
    sort(a, a+n);
    show(a,n);

    printf("Random shuffle\n");
    random_shuffle(a, a+n);
    show(a,n);

    printf("Sort a[3..7]\n");
    sort(a+3, a+8);
    show(a,n);
    return 0;
}
```


如果想要由大排到小，通常有兩個方法：一種是將原來的數字變號(x 變-x)，另外一種方法是自行指定比較函數。事實上 sort()除了前兩個參數指定範圍之外，還可以傳入第三個參數，第三個參數是一個比較函數，這個函數負責告訴 sort()排序的順序是什麼：比較函數必須傳入兩個參數，如果第一個參數要排在第二個的前面就回傳 true，否則回傳 false。這樣聽起來很複雜，看例子比較簡單，我們也一起說明多欄位資料的排序。

先看 cmp1(int s, int t)，它傳入兩個整數，回傳是否 s>t，意思是前面的要比較大。第 36 行 sort(a, a+n, cmp1);會讓資料從大排到小。我們也可以呼叫庫存的比較函數 greater，如第 40 行的寫法，但這個寫法可能不容易記得。為了展示多欄位資

料的排序，我們定義了一個結構 struct point 用來存放平面座標，並且展示了三個不同的比較函數讓資料依照不同的要求來排序。

```
00 #include <bits/stdc++.h>
01 using namespace std;
02 #define N 10
03 void show(int a[], int n) {
04     for (int i=0;i<n-1;i++)
05         printf("%3d ",a[i]);
06     printf("%3d\n",a[n-1]);
07     return;
08 }
09
10 bool cmp1(int s, int t) {
11     return s>t;
12 }
13 struct point {
14     int x,y;
15 };
16 bool cmp2(point &s, point &t) {
17     return s.x < t.x;
18 }
19 bool cmp3(point &s, point &t) {
20     return s.y > t.y;
21 }
22 bool cmp4(point &s, point &t) {
23     return s.x+s.y < t.x+t.y;
24 }
25 void showp(point a[], int n) {
26     for (int i=0;i<n-1;i++)
27         printf("(%2d, %2d) ",a[i].x, a[i].y);
28     printf("(%2d, %2d)\n",a[n-1].x, a[n-1].y);
29     return;
30 }
31 int main() {
32     int a[N], n=10;
33     for (int i=0;i<n;i++) a[i]=rand()%100;
34     show(a,n);
35     printf("from large to small\n");
36     sort(a, a+n, cmp1);
37     show(a,n);
38
39     printf("Using greater\n");
40     sort(a, a+n, greater<int>());
41     show(a,n);
42
43     point p[N];
44     for (int i=0;i<n;i++)
45         p[i].x=rand()%100, p[i].y=rand()%100;
46     printf("%d points on the plane\n");
47     showp(p,n);
48
49     printf("sorted for x from small to large\n");
50     sort(p, p+n, cmp2);
```

```
51      showp(p,n);
52      printf("sorted for y from large to small\n");
53      sort(p, p+n, cmp3);
54      showp(p,n);
55      printf("sorted for x+y from small to large\n");
56      sort(p, p+n, cmp4);
57      showp(p,n);
58      return 0;
59  }
```

多欄位資料的排序在很多地方都用得到，初學者應該要花時間了解 struct 與其排序的寫法。我們以下再說明一個偷懶的方法，考試與競賽時常碰到多欄位資料的排序，尤其是兩個欄位，例如平面 XY 座標或是線段左右端點，如果不想寫 struct，有沒有辦法偷懶呢？答案是有的。

C++ STL 中有定義好的 pair 可以用來存兩個欄位的資料，而 pair 的比較運算(<)就是兩個欄位的字典順序(lexicographic order)，也就是先比第一個欄位，如果第一欄位相同再比第二欄位。在應用時，我們只要把要排序的欄位放在第一欄位就可以了。要提醒幾件事：

- pair 的第一欄位叫做 first 而第二欄位叫做 second；
- 建構一個 pair 可很簡單的以用大括號{}，例如 p[i]={5, 3};但如果是比較舊版的 C++可能並不支援這個寫法，如果不支援大括號直接建構的場合，就要用 make_pair()的函數，在網路上看比較舊的程式碼可能會很常看到類似 make_pair(5,3)這樣的寫法。

此外，如果超過兩個欄位呢？事實上也有做法，但這裡並不鼓勵初學者學太多庫存函數來偷懶，而且 struct 與比較函數還是應該要學的。如果有興趣的人可以自己去查詢 C++ tuple。以下看簡單的例子來了解用法。

```
00 #include <bits/stdc++.h>
01 using namespace std;
02 #define N 10
03
04 void showp(pair<int,int> a[], int n) {
05     for (int i=0;i<n-1;i++)
06         printf("(%1d, %1d) ",a[i].first, a[i].second);
07     printf("(%1d, %1d)\n",a[n-1].first, a[n-1].second);
08     return;
09 }
10 int main() {
11     pair<int,int> p[N];
12     int n=10;
13     for (int i=0;i<n;i++)
14         p[i]={rand()%10, rand()%10};
15     showp(p,n);
```

```
16
17      printf("sorted by lexicographic order\n");
18      sort(p, p+n);
19      showp(p,n);
20
21      vector<pair<int,int>> q(p, p+n); // copy p to q
22      for (auto s:q) {
23          printf("(%d,%d) ", s.first, s.second);
24      }
25      printf("\nsorted by greater order\n");
26      sort(q.begin(), q.end(), greater<pair<int,int>>());
27      for (auto s:q) {
28          printf("(%d,%d) ", s.first, s.second);
29      }
30
31      return 0;
32  }
```

第 21 行開始的最後一段我們是展示了如何對 vector 使用 sort()，並且使用 greater 將它從大到小排序。vector 基本上可以看成陣列，只是長度可變，除非在處理樹狀圖與圖形問題上，通常不一定要使用，如果暫時對你太複雜，你都可以用陣列來替換它。

下面的例題是排序的一個應用，常用在某些題目的前置處理，為了方便練習，我們把它寫成題目的樣子。

例題 P-2-1. 不同的數–排序

假設有 N 個整數要被讀到一個陣列中，我們想要將這些數字排序並去除重複的數字，例如輸入的整數序列是 (5, 3, 9, 3, 15, 9, 8, 9)，這些數如從小到大排是 (3, 3, 5, 8, 9, 9, 9, 15)，去除重複者後為(3, 5, 8, 9, 15)。寫一個函數，傳回有多少不同的數字並且將結果放在陣列中回傳。

Time limit: 1 秒

輸入格式：輸入兩行，第一行是正整數 N，N 不超過 10 萬，第二行是 N 個整數，大小不超過 10^9，以空白間隔。

輸出： 第一行輸出有多少相異整數，第二行輸出這些相異整數，相鄰數字之間以一個空白間隔。

範例輸入：

```
7
0 3 9 3 3 -1 0
```

範例結果：

```
4
0 -1 3 9
```

請看以下的範例程式。函數 distinct()傳入兩整數陣列：from[]是要傳進來的數字，to[]是擺放結果的陣列，另外 n 是傳入數字的個數。我們不希望破壞原陣列的內容，先將資料複製到另外一個 vector v，並且將 from[]的內容複製到 v。接著將 v 內容排序，排序後相同的數字會在一起，因此除了 v[0]之外，只要是 v[i]!=v[i-1]的 v[i]都是一個相異的數字。請留意我們以變數 num 儲存目前找到相異數的個數，並且 to[num]就是儲存位置，將相異數字複製到 to[num]之後必須將 num 的值增加 1，這兩個動作可以寫成一個指令來完成：

```
to[num++] = v[i];
```

```
// P_2_1 distinct number -- sort
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int distinct(int from[], int to[], int n) {
    if (n<1) return 0;
    vector<int> v(from, from+n); // copy from[] to v
    sort(v.begin(), v.end());
    to[0]=v[0];
    int num=1; // number of distinct number
    for (int i=1; i<n; i++)
        if (v[i]!=v[i-1]) // distinct
            to[num++] = v[i];
    return num;
}
int main() {
    int a[N], b[N],n, k;
    scanf("%d", &n);
    for (int i=0;i<n;i++)
        scanf("%d", a+i);
    k=distinct(a,b,n);
    printf("%d\n",k);
    for (int i=0;i<k-1;i++)
        printf("%d ",b[i]);
    printf("%d\n",b[k-1]);
    return 0;
}
```

2.2. 搜尋

在一群資料中搜尋某筆資料，通常有線性搜尋與二分搜尋，線性搜尋就是一個一個找過去，在沒有順序的資料中，也只好如此。如果資料已經依照順序排好，我們可以採用二分搜尋，原因是效率好太多了。以一個 100 萬筆的資料來說，線性搜尋可能需要比對 100 萬次，而二分搜尋的 worst case 複雜度是 O(log(n))，最多只要比較 20 次就可以找到了，也就是有 5 萬倍的效率差異，所以對很多問題是非用不可。二分搜除了自己寫之外，也有庫存函數可以呼叫，此外，C++也有一些好用的資料結構可以達到搜尋的目的，我們在這一節中會介紹以下技巧：

- 基本二分搜的寫法
- C++的二分搜函數以及相關資料結構

此外，二分搜尋不單是在一群資料中找資料，它往往也搭配其他演算法策略使用，這一部份會出現在後續的章節中。

基本二分搜的寫法

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。

```
#include <bits/stdc++.h>
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++) printf("%d ",p[i]);
    printf("\n");
```

```
    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;
}
```

在很多時候，我們需要知道不只是某元素在不在，當它不在時，希望找到小於它的最大值的位置(或是大於它的最小值)，這看似只要把上述程式簡單修改一下就好了，但事實上很容易寫錯。假設我們需要知道第一個大於等於 x 的位置，我們將前述程式直接修改如下。

```
// find the first >= 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 ???; // maybe wrong
}
```

你檢查後可能發現應該要回傳 left 是對的，事實上很多人會弄錯，另外也會不小心寫成無窮迴圈。這個寫法並非一定不好，這裡只是希望提醒讀者這個寫法容易犯錯。比較推薦的寫法在前一章曾經提出過，不是用左右搜的方式，而是用一路往前跳的方式，你想想看，正常人如果反覆的向左向右是不是很容易昏頭，一路往前應該比較不容易搞錯。

請看以下的範例程式，傳入的是搜尋區間的大小 n，在一開始檢查一下 a[0]是否就是所要答案，如果不是，程式中就會一直保持 a[po]<x 這個不變性，在此特性下，只要還能往前跳就往跳，跳的距離每次把它折半。

```
// binary search the first >=x between a[0..n-1]
int jump_search(int a[], int n, int x) {
    if (a[0]>=x) return 0; // check the first
    int po=0; // the current position, always < x
    for (int jump=n/2; jump>0; jump/=2) { //jump distance
        while (po+jump<n && a[po+jump]<x)
            po += jump;
    }
    return po+1;
}
```

時間複雜度也是 O(log(n))，因為跳的距離每次折半，而且內層的 while 迴圈至多執行兩次(不會很難證明)。

C++的二分搜函數以及相關資料結構

C++中提供了許多跟搜尋有關的函數以及資料結構，在考試或比賽中，如果是適合的題目，使用庫存函數可以節省時間也避免錯誤，這裡介紹一些基本的用法。常用的二分搜相關函數有下列三個：

- binary_search()
- lower_bound()
- upper_bound()

其實只要學會使用 lower_bound()，這個函數是用來找到第一個**大於等於**某個值的位置，另外兩個都很像，upper_bound()找到第一個**大於**某值的位置，而 binary_search()是只傳回是否找到。

與 sort()一樣，函數 lower_bound()允許自訂比較函數，當用在基本資料型態且以內定的比較函數時，它的用法最簡單，例如資料是整數且以小於來比較。下面的範例我們示範用於陣列與 vector。基本上，呼叫時以

```
lower_bound(first, last, t);
```

要求在[first, last)的範圍內以二分搜找到第一個大於或等於 t 的值，找到時回傳所在位置，找不到時(t 比最後一個大)，回傳 last。以下有幾點請注意：

- 找到時回傳的是位置，要得到陣列索引值就將它減去陣列起始位置。
- 呼叫 lower_bound()必須確定搜尋範圍是排好序的，如果你針對亂七八糟的資料以 lower_bound()去進行二分搜，只會得到亂七八糟的結果，編譯器是不會告訴你沒排序的。
- 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。

以下這個範例程式很簡單，可以把它拿來執行一下就可以了解 lower_bound 的基本運用。

```
// demo lower_bound for int array
// binary search the first >=x
#include <bits/stdc++.h>
using namespace std;
#define N 5

int main() {
```

```
    int p[N]={5, 1, 8, 3, 9};
    int n=5;
    sort(p, p+n);
    for (int i=0;i<n;i++) printf("%d ",p[i]);
    printf("\n");
    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;
}
```

與 sort()一樣，當我們要使用非內定的比較函數或是資料為多欄位結構時，我們就需要自己寫比較函數，同樣地，一個偷懶的方法是使用庫存的結構 pair。以下的範例程式展示結構資料的二分搜以及利用 pair 來存放資料而逃避寫比較函數的方式

```
// demo lower_bound for struct and pair
#include <bits/stdc++.h>
using namespace std;
#define N 5
struct point {
    int x,y;
};
// compare by only x
bool pcmp(point s, point t) {
    return s.x < t.x;
}
int main() {
    point p[N];
    int n=5;
    for (int i=0;i<n;i++) p[i].x=rand()%10, p[i].y=rand()%10;
    sort(p, p+n, pcmp);
    for (int i=0;i<n;i++) printf("(%d,%d) ",p[i].x, p[i].y);
    printf("\n");
    for (int i=0;i<5;i++) {
        point t={i*3,rand()%10};
        int ndx=lower_bound(p, p+n, t, pcmp) - p;
        if (ndx<n)
            printf("Find >=(%d,%d) at [%d]=(%d,%d)\n", \
                  t.x, t.y, ndx,p[ndx].x, p[ndx].y);
```

```
        else
            printf("No one >=(%d,%d)\n",t.x, t.y);
    }
    // for vector
    vector<point> v(p, p+n);
    for (int i=0;i<5;i++) {
        point t={i*3,rand()%10};
        auto q=lower_bound(v.begin(), v.end(), t, pcmp);
        if (q!=v.end())
            printf("Find >=(%d,%d) at [%d]=(%d,%d)\n", \
                  t.x, t.y, q-v.begin(), q->x, q->y);
        else
            printf("No x>=(%d,%d)\n",t.x, t.y);
    }
    // using pair, default compare function
    printf("Compare by x and then y\n");
    vector<pair<int,int>> a;
    for (point e:v) a.push_back({e.x,e.y});
    for (int i=0;i<5;i++) {
        pair<int,int> t={i*3,rand()%10};
        auto q=lower_bound(a.begin(), a.end(), t);
        if (q!=a.end())
            printf("Find >=(%d,%d) at [%d]=(%d,%d)\n", \
                t.first,t.second, q-a.begin(), q->first, q->second);
        else
            printf("No x>=(%d,%d)\n",t.first, t.second);
    }
    printf("Compare by only x\n");
    for (int i=0;i<5;i++) {
        pair<int,int> t={i*3,rand()%10};
        // set t.second to minimum to ignore comparison of y-value
        auto q=lower_bound(a.begin(), a.end(), make_pair(t.first,-1));
        if (q!=a.end())
            printf("Find >=(%d,%d) at [%d]=(%d,%d)\n", \
                  t.first,t.second, q-a.begin(), q->first, q->second);
        else
            printf("No x>=(%d,%d)\n",t.first, t.second);
    }

    return 0;
}
```

下面一個例題是一個有趣的排序與搜尋練習，也是實際解題時經常需要用的步驟，它有個名字是「離散化」，雖然這個名字似乎不是那麼恰當，但是從以前就這麼多人這樣稱呼它。它需要用到前一個例題 P_2_1 (不同的數) 當作它的步驟。

例題 P-2-2. 離散化 - sort

假設有 N 個整數要被讀到一個陣列中，我們想要將這些整數置換成從 0 開始依序排列的整數並且維持它們原來的大小關係，例如輸入的整數序列是 (5, 3, 9, 3, 15, 9, 8, 9)，這些數如從小到大排是 (3, 3, 5, 8, 9, 9, 9, 15)，去除重複者後為(3, 5, 8, 9, 15)，所以我們要替換的是：

3 ➔ 0
5 ➔ 1
8 ➔ 2
9 ➔ 3
15 ➔ 4
所以原先的序列就會變成(1, 0, 9, 0, 4, 3, 2, 3)。

Time limit: 1 秒

輸入格式：輸入兩行，第一行是正整數 N，N 不超過 10 萬，第二行是 N 個整數，大小不超過 10^9，以空白間隔。

輸出：輸出置換後的序列，兩數之間以一個空白間隔。

範例輸入：

```
7
0 3 9 3 3 -1 0
```

範例輸出：

```
1 2 3 2 2 0 1
```

如果已經了解如何處理不同的數字，加上二分搜尋就可以簡單地處理這個問題，請看以下範例。我們先呼叫剛才寫好的函數將陣列中的相異數字排序在 b[]陣列中，然後，對於原陣列 a[]中的每一個數字，將其置換成它在 b[]中的 index 就可以了，而要快速的在 b[]中找到 a[i]，我們用二分搜，自己寫或是呼叫 lower_bound()都可以，因為 a[i]一定在 b[]中，所以我們不需要擔心找不到的情形。

```
// P_2_2 discretization -- sort
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int distinct(int from[], int to[], int n) {
    if (n<1) return 0;
    vector<int> v(from, from+n); // copy from[] to v
    sort(v.begin(), v.end());
    to[0]=v[0];
    int num=1; // number of distinct number
    for (int i=1; i<n; i++)
        if (v[i]!=v[i-1]) // distinct
            to[num++] = v[i];
    return num;
}
int main() {
    int a[N], b[N],n, k;
```

```
    // input data
    scanf("%d", &n);
    for (int i=0;i<n;i++)
        scanf("%d", a+i);
    // sort distinct number to b
    k=distinct(a,b,n);
    // replace number with its rank
    for (int i=0;i<n;i++) {
        a[i] = lower_bound(b, b+k,a[i]) - b; // always found
    }
    // output
    for (int i=0;i<n-1;i++)
        printf("%d ",a[i]);
    printf("%d\n",a[n-1]);
    return 0;
}
```

檔案中範例程式 P_2_2b.cpp 則是以自己寫二分搜的方式來做，這裡就不把它列出來了。

C++可供搜尋的容器(set/map)

C++的 set 可以看成一個容器，它其實是一個平衡的二元搜尋樹，是一種動態的資料結構，所謂動態資料結構指的是可以做插入刪除。在這裡我們著重在如何使用它而非它的原理，我們只要知道在 set 中插入、刪除與搜尋一個元素都可以在 O(log(n))的時間完成，此外它是有順序的，如果我們從頭到尾做一個歷遍(Traversal)，它經過的資料的順序就是由小到大排列的，它是一個非常有效能而且容易使用的技術，值得花一些時間加以練習。

在以下的範例中，我們展示了 set 的基本用法，包含如何宣告、歷遍、插入資料、刪除資料、以及搜尋資料。有件事情要特別注意，set 和一般中學數學中的一般集合一樣，不允許多重元素，如果 set 中插入已經有的元素，事實上該插入的動作是不成功的，set 的 insert()有多種形式，其中有的會回傳是否成功，需要了解的人可以參考網路上的技術文件。

```
// demo set
#include <bits/stdc++.h>
using namespace std;
#define N 10
#define P 10

int main() {
    set<int> S; // a set S for storing int
    printf("Insert: ");
    for (int i=0;i<N;i++) {
        int t=rand()%P;
        S.insert(t); // insert an element into S
        printf("%d ",t);
```

```
    }
    printf("\nTraversal after insertion: ");
    for (auto it=S.begin(); it!=S.end(); it++) {
        printf("%d ", *it); // iterator as a pointer
    }
    printf("\nAnother traversal: for (int e:S)");
    for (int e: S) { // for each element e in S, do ...
        printf("%d ", e);
    }
    printf("\nClear and re-insert data:\n");
    S.clear(); // clear set t empty
    for (int i=0;i<N;i++)
        S.insert(i*5);
    for (int e: S)
        printf("%d ", e);
    printf("\n");
    // to find if an element in the set
    auto it=S.find(15);
    if (it!=S.end()) // return end() when not found
        printf("find 15 in S\n");
    else printf("15 is not in S\n");
    int x=15;
    printf("After S.erase(15)\n");
    S.erase(x); // erase element of value x
    it=S.find(x);
    if (it!=S.end())
        printf("find %d in S\n",x);
    else printf("%d is not in S\n",x);
    // find lower_bound, the first one >=x
    it=S.lower_bound(x);
    if (it!=S.end())
        printf("lower_bound of %d is %d\n", x, *it);
    else printf("no lower_bound of %d\n",x);
    // find upper_bound, the first one >x
    x=(N-1)*5;
    it=S.upper_bound(x);
    if (it!=S.end())
        printf("upper_bound of %d is %d\n", x, *it);
    else printf("no upper_bound of %d\n",x);

    return 0;
}
```

除了 set 之外，C++中有幾個與它很類似資料結構，包括 multiset 以及 map 與 multimap。顧名思義，multiset 與 set 的差別在於允許多重元素，也就是相同的元素可以有多個。而 map 與 set 其實是相同的資料結構，只是它比 set 多提供了一個欄位，也就是說它的每一個元素是一個兩個欄位的 pair 資料類型，第一個欄位(名字是 first)是 key，第二個欄位(second)可以用來存放其它想要的資料，所以它的用途更廣。map 與 set 一樣不可以有相同 key 的元素，如果需要重元素則可以使用 multimap。map 還有一個特殊的地方是它可以使用類似陣列的[]表示式，在下面的範例中我們也示範了這樣的用法。

```
// demo multi-set, map
#include <bits/stdc++.h>
using namespace std;
#define N 10
#define P 10

int main() {
    multiset<int> S;
    printf("Insert: ");
    for (int i=0;i<N;i++) {
        int t=rand()%P;
        S.insert(t);
        printf("%d ",t);
    }
    printf("\nTraversal after insertion: ");
    for (int e: S) {
        printf("%d ", e);
    }
    printf("\nClear and re-insert data:");
    S.clear(); // clear set
    vector<int> v({5,5,2,3,7,7,8});
    S.insert(v.begin(), v.end());
    for (int e: S)
        printf("%d ", e);
    printf("\nAfter S.erase(5): ");
    S.erase(5);
    for (int e: S)
        printf("%d ", e);
    printf("\nAfter erase one of 7: ");
    auto it=S.find(7);
    if (it!=S.end())
        S.erase(it);
    for (int e: S)
        printf("%d ", e);
    printf("\n");
    // demo map
    map<char, int> M;
    char str[100]="a demo of c++ map", ch;
    int len=strlen(str);
    for (int i=0; i<len; i++) M[str[i]]+=1;
    printf("\nAfter insert %s: ",str);
    for (auto e: M)
        printf("(%c:%d)", e.first, e.second);
    printf("\n");
    ch='x';
    auto mit=M.find(ch);
    if (mit==M.end())
        printf("No %c in M\n",ch);
    else printf("count(%c)=%d\n",ch,mit->second);
    ch='m';
    printf("M[%c]=%d\n",ch,M[ch]);
    ch='y';
    printf("M[%c]=%d\n",ch,M[ch]);
    M['y'] = 5;
    printf("After M['y']=5, M[%c]=%d\n",ch,M[ch]);
    printf("After erase('y'), ");
    M.erase(ch);
    mit=M.find(ch);
    if (mit==M.end())
```

```
        printf("No %c in M\n",ch);
    M[ch];
    printf("After M['y'];, M[%c]=%d\n",ch,M[ch]);

    return 0;
}
```

上面的範例中我們特別示範了 multiset 中刪除元素的兩種不同方法，刪除相同鍵值的所有元素或是刪除某一個元素。C++是物件導向的程式語言，往往一個相同的函數名字可以有多種不同的呼叫方式，這與傳統 C 語言有很大的差異，好處是使用起來更方便，但壞處是用錯了會不知道，因為 compiler 往往抓不到這一類錯誤，它算是語意的錯誤而非語法錯誤。以下再舉一個容易被誤用的例子，請看下面的程式。

這支程式其實沒有要做甚麼事情，只是建構一個很大的 set，然後在這個 set 中做 100 次的 lower_bound()搜尋。set 的搜尋應該是程式中的第一個寫法，在程式中我們寫了第二個寫法，那其實是用在陣列或 vector 上的 lower_bound 的寫法，兩邊名字一樣讓我們好記，但是它其實是不同，壞就壞在第二種寫法對 compiler 來說並不算錯誤，只是，它跑起來很慢。你可以試試看，執行時間相差非常的多。

```
// misusing lower_bound
#include <bits/stdc++.h>
using namespace std;
#define N 500000
#define P 1000000009
int ran30(int n) {
#if RAND_MAX < 40000
    return ((rand()<<15)|rand())%n;
#else
    return rand()%n;
#endif
}
int main() {
    set<int> S;
    for (int i=0;i<N;i++)
        S.insert(ran30(P));
    vector<int> v;
    int n=100;
    for (int i=0;i<n;i++) v.push_back(ran30(P));
    clock_t t1,t2;
    t1=clock();
    int total=0;
    for (int x:v) {
        auto f=S.lower_bound(x);
        if (f!=S.end())
            total=(total+*f)%P;
    }
    t2=clock();
    printf("total=%d, time=%f\n",total,(float)(t2-t1)/CLOCKS_PER_SEC);
    // misusing lower_bound
    t1=clock();
    total=0;
    for (int x:v) {
```

```
        auto f=lower_bound(S.begin(), S.end(), x);
        if (f!=S.end())
            total=(total+*f)%P;
    }
    t2=clock();
    printf("bad using lower_bound, total=%d, time=%f\n", \
        total,(float)(t2-t1)/CLOCKS_PER_SEC);
    return 0;
}
```

上面這支程式中還有兩點可以提醒：第一點是程式中為什麼寫了一個奇怪的副程式 ran30()來產生亂數，而不直接用 rand()%n 就好呢？其實是因為目前某些電腦上還有很多 C++編譯器的版本的 rand()函數是 15-bit 的隨機亂數。這樣寫可以讓在不同編譯器下都可以達到足夠大的亂數範圍。

接下來我們以前面看過的例題 P_2_2 來做 set/map 的練習，就是做離散化。前面用的是排序以及二分搜，事實上用 set/map 也是很好寫。

例題 P-2-2C. 離散化 - set/map (*)

題目敘述請見例題 P-2-2。

我們利用 set/map 不可以重複 key 而且也是從小排到大的特性，只要將所有的資料裝進去，就自然可以達到第一個步驟找出相異數字的目的，在第二步驟的替換工作方面，因為不像陣列可以直接以 index 來替換，我們在開始查詢前，先從小到大走一遍，設定每一個值要替換的值，為了這個目的，我們需要用 map 來做，因為要把替換的值(相異數的排序位置)放在 map 的第二欄位。以下是範例程式。

```
// P_2_2 discretization -- map
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int main() {
    int a[N],n, k;
    // input data
    scanf("%d", &n);
    for (int i=0;i<n;i++)
        scanf("%d", a+i);
    map<int,int> S;
    for (int i=0;i<n;i++)
        S[a[i]] = 0; // insert a[i] and set rank=0
    int r=0;
    // traversal and set rank in second
    for (auto it=S.begin(); it!=S.end(); ++it) {
        it->second = r++;
    }
```

```
    // replace number with its rank
    for (int i=0;i<n;i++) {
        a[i] = S.find(a[i]) -> second; // always found
        // find() return the iterator, then take the rank
        // or S.lower_bound(a[i]) -> second;
    }
    // output
    for (int i=0;i<n-1;i++)
        printf("%d ",a[i]);
    printf("%d\n",a[n-1]);
    return 0;
}
```

2.3. 其他相關技巧介紹

在這一節中，我們將介紹以下主題：

- Bitonic sequence 的搜尋
- 快速冪

Bitonic sequence 的搜尋

Bitonic sequence 是指先遞增後遞減(或者先遞減再遞增)的序列，也有人叫它一山峰(一山谷)序列。在單調遞增序列(函數)中可以用二分搜來找第一個超越 0 值或某值的位置，那麼對於 Bitonic 序列是否可以類似二分搜有效率地找到極值呢？例如在一山谷的序列中找最小值。答案是有的，常用的方法有兩種：三分搜以及差分二分搜。

與二分搜一樣，我們始終維護一個搜尋區間，二分搜是每次將區間二等分，一個比較之後刪除一半。三分搜是將區間三等分，如果 1/3 的位置大於 2/3 的位置，可以丟掉左邊的 1/3；反之，如果 2/3 的位置較大，則可以丟棄右邊 1/3 的區間。如此每次可以將區間長度減少 1/3，在 O(log(n))次比較可以找到最低點。

另外一個方法更簡單，若序列 f 為先遞減再遞增(一山谷)，則差分
g[i]=f[i]-f[i-1]
在前半段是負的，在後半段是正的，只要以二分搜找到差分序列第一個大於等於 0 的位置就是 f 的最小值。

快速冪

所謂快速冪是如何快速計算 x^y 的方法，這裡要探討的不是浮點數的運算，通常是針對整數，而這個數字通常都大到超過整數變數的範圍，所以通常都是求模(mod) P 的運算，

也就是給定 x, y, P，要計算 x^y (mod P)。庫存函數中計算指數的函數都是浮點數的運算，會有運算誤差，所以不能用。

例題 P-2-3. 快速冪

輸入正整數 x, y, 與 p，計算 x^y (mod p)。x, y, p 皆不超過 1e9+9。例如 x=2, y=5, p=11，則答案是 10。

Time limit: 1 秒

輸入格式：輸入 x, y, 與 p 在同一行以空白間隔。

輸出格式：輸出計算結果。

範例輸入：

```
2 5 11
```

範例輸出：

```
10
```

計算指數最直接的方法就是按照定義一個一個的乘，如下列的程式碼，提醒一下，每次運算後都要取餘數，否則有 overflow 的可能。

```
t = 1;
for (int i =0; i<y; i++)
    t =(t*x) % p;
```

但是這個方法效率太差，如果 y 是 10 億就要執行 10 億次乘法運算。前面介紹二分搜逐步縮減跳躍距離的方法，以類似的概念，我們可以設計出只要 log(y) 次乘法的方法，這個方法以遞迴的形式最簡單。在以下範例中我們同時展示遞迴與迴圈兩種寫法。

```
// find x^y mod P, 32-bit positive int x, y, p
#include <cstdio>
typedef long long LL;
LL exp(LL x, LL y, LL p) {
    if (y==0) return 1;
    if (y & 1) return (exp(x, y-1,p)*x)%p;
    // otherwise y is even
    LL t=exp(x, y/2, p);
    return (t*t)%p;
}

LL exp2(LL x,LL y,LL p) {
```

```
    LL t=1, xi=x, i=1; // t is result, xi = x^ (2^i)
    while (y>0) {
        if (y & 1) // odd, (i-1)-bit of y = 1
            t=(t*xi)%p;
        y>>=1;
        xi=(xi*xi)%p;
        i=i*2; // i is useless, for explanation
    }
    return t;
}

int main() {
    long long x, y, p, res;
    scanf("%lld%lld%lld", &x, &y, &p);
    printf("%lld\n",res=exp(x, y,p));
    if (res!=exp2(x,y,p))
        fprintf(stderr,"different result");
    return 0;
}
```

遞迴的寫法很好懂，若 y 是奇數則先遞迴求出 y-1 次方後再乘一次 x；若 y 是偶數則求出 y/2 次方後自乘，終止條件為 y=0 時結果是 1。遞迴的版本中，xi 是每次自乘的結果，所以它的內容是 x 的 1 次方、2 次方、4 次方、8 次方、…這樣的方式變化，將 y 以二進制來看，對每一個 bit 為 1 的位置，把對應項的 x 次方乘到結果中就是答案。舉例來說，若 y=19，二進制是 10011，就是要計算

$x^{17} = x^{16} * x^{2} * x^{1}$。

兩種寫法的效率差不多，都是很快的方法，遞迴版本尤其容易記。以下是一個稍加變化的習題，給一個提示：當 x 大到超過 long long 的時候，我們需要先求 x 除以 p 的餘數，而除以 p 的餘數可以一位一位的累加計算。例如 x=123, p=5，x 可以看成 (1*10+2)*10+3，所以 x 除以 p 的餘數也就等於 ((((1*10%p+2)%p)*10)%p+3)%p。

習題 Q-2-4. 快速冪--200 位整數

題目與輸入皆同於 P-2-3，但 x 的範圍是不超過 200 位的正整數

範例輸入：

```
12345678901234567890123456789O 5 11
```

範例輸出：

```
10
```

快速計算費式數列

利用快速冪我們可以計算費式數列的第 n 項。

習題 Q-2-5. 快速計算費式數列第 n 項

令 f[0]=0, f[1]=1, 以及 f[n]=f[n-1]+f[n-2] for n>1。輸入非負整數 n，請輸出 f[n]除以 p 的餘數, p=1000000007。$n<2^{31}$。

Time limit: 1 秒

輸入格式：輸入可能有多行，每一行有一個整數是一筆測資，最後一行以-1 代表結束，不需要處理該筆測資。

輸出格式：每一行依序輸出計算結果。

範例輸入：

```
6
123456789
100
-1
```

範例輸出：

```
8
62791945
687995182
```

Q_2_5 說明：我們可以將費式序列的定義寫成以下矩陣的形式

- $\begin{bmatrix} f[n] \\ f[n-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} f[n-1] \\ f[n-2] \end{bmatrix}$, $\begin{bmatrix} f[1] \\ f[0] \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

將上式迭代展開，因為矩陣滿足結合律，所以可以得到

- $\begin{bmatrix} f[n] \\ f[n-1] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \times \begin{bmatrix} f[1] \\ f[0] \end{bmatrix}$

如果我們可以很快的算出那個矩陣 A 的 n-1 次方，就可以得出

$f[n]=A^{n-1}[0][0]$。

2.4. 其他例題與習題

排序與搜尋大多都會與其他的演算法結合，成為某個解題方法中的一個步驟，單獨考排序的並不多。以下舉一些例子，先看一個很常見的基本問題。

例題 P-2-6. Two-Number problem

假設 A 為 m 個相異整數的集合，B 為 n 個相異整數的集合，而 K 是一個整數。請計算有多少對(a, b)的組合滿足 a ∈ A, b ∈ B 且 a+b = K。

Time limit: 1 秒

輸入格式：輸入可能有多行，第一行有三個整數 m, n 與 K，第二行有 m 個整數是 A 中的元素，第三行有 n 個整數 B 中的元素一筆測資。同一行相鄰數字間以空白間隔。兩集合元素個數均不超過 10 萬，整數的絕對值不超過 10 億。

輸出格式：輸出組合個數。

範例輸入：

```
3 4 2
1 6 -3
5 1 -1 -3
```

範例輸出：

```
2
```

這一題有多個解法，基本的想法是排序後搜尋，有下列幾種作法：

- 將 A 排序後，對 B 中的每一個 b，以二分搜在 A 中尋找 K-b。
- 將 A 與 B 分別排序後，對 B 中的每一個 b，以滑動的方式找 K-b。
- 把A放進 set 來做
- 把A放進 unordered_set 來做

由於作法類似，我們省略過第一與第四種。以下是第二種的範例程式，在 a 與 b 分別排序後，由前往後對於每一個 a[i]，以 while 迴圈往其找到第一個小於等於 k - a[i] 的位置，請注意，由於 a[i]是由小到大，因此 j 每次不必從最尾端重新開始，只要從前一次停下的位置開始就可以了，所以如果不計排序，這個程式的複雜度是 O(m+n)。

這個在陣列中維護兩個位置的方法，也有人稱為 Two-pointers method(雙指針方法)，它雖然用了雙迴圈但複雜度並不是平方，類似的情形我們在下一章會看到更多例子。

```
// p_2_6a find a+b=k
#include <bits/stdc++.h>
using namespace std;
#define N 100010
int a[N], b[N];

int main() {
    int m, n, k,i;
    scanf("%d %d %d", &m,&n,&k);
    for (i=0;i<m;i++) scanf("%d", a+i);
    for (i=0;i<n;i++) scanf("%d", b+i);
    sort(a, a+m); // sort a from small to large
    sort(b, b+n); // sort b from small to large
    int j=n-1; // index of b, from n-1 to 0
    int ans=0;
    for (i=0; i<m; i++) { // each a[i]
        while (j>0 && b[j]>k-a[i]) // backward linear search
            j--;
        if (a[i]+b[j]==k) ans++;
    }
    printf("%d\n",ans);
    return 0;
}
```

接下來看用利用 set 的做法，請看以下範例程式，我們可以注意到，在這個例子上我們甚至不需要用到任何陣列。

```
// p_2_6b find a+b=k, using set
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int main() {
    int m, n, k,i,t;
    scanf("%d %d %d", &m,&n,&k);
    set<int> S;
    // read A into set
    for (i=0;i<m;i++) {
        scanf("%d", &t);
        S.insert(t);
    }
    int ans=0;
    for (i=0;i<n;i++) { // for each t in B
        scanf("%d", &t);
        if (S.find(k-t)!=S.end()) // search k-t in A
            ans++;
    }
    printf("%d\n",ans);
    return 0;
}
```

接下來是一個與前者類似但較複雜一點的習題。

習題 Q-2-7. 互補團隊 (APCS201906)

前 *m* 個英文大寫字母每個代表一個人物，以一個字串表示一個團隊，字串由前 *m* 個英文大寫字母組成，不計順序也不管是否重複出現，有出現的字母表示該人物出現在團隊中。兩個團隊**沒有相同的成員而且聯集起來是所有 *m* 個人物**，則這兩個團隊稱為「互補團隊」。輸入 *m* 以及 *n* 個團隊，請計算有幾對是互補團隊。我們假設沒有兩個相同的團隊。

Time limit: 1 秒

輸入格式：第一行是兩個整數 m 與 n，2 ≤ m ≤ 26，1 ≤ n ≤ 50000。第二行開始有 n 行，每行一個字串代表一個團隊，每個字串的長度不超過 100。

輸出格式：輸出有多少對互補團隊。

範例輸入：

```
10 5
AJBA
HCEFGGC
BIJDAIJ
EFCDHGI
HCEFGA
```

範例輸出：

```
2
```

解題提示：與前面的例題在結構上是相似的，前面是找數字，這裡是找集合。因為字串中有重複的字母而且沒有照順序，我們需要將每一個集合表示成唯一的表式方式才能夠有效的搜尋。有兩個方法可以做：

- 將每個字串去除重複字母且裡面的字母是由小到大排列的。例如 AJBA 就改成 ABJ，HCEFGGC 就改成 CEFGH。
- 以一個整數表示一個集合，第 i 個 bit 設為 1 代表第 i 個字母在集合中，否則為 0。此法在在第一章窮舉子集合時也介紹過。

第二種方法要比第一種方法更快，因為數字的搜尋要比字串來得快。以下的程式片段可以將一個字串轉換成對應以及互補集合的整數：

```
//transforming a string to corresponding int
    ff = (1<<m) - 1; // bits 0~(m-1) are 1
    scanf("%s",s);
    int len=strlen(s), team=0;
    for (int j=0;j<len;j++) // 1 for existing
        teams |= 1<<(s[j] - 'A'); // set bit to 1
    complement = ff - teams[i]; // int of the complement
```

以下的習題我們用來展示如何求 Modular multiplicative inverse，就是在模運算下的乘法反元素，有些人稱為「模反元素」或「模逆元」。

對於任何一個正整數 a，它的模 P 乘法反元素就是滿足 (a * b) % P = 1 的整數 b，這裡的 % 就是取餘數運算。計算 a 的模逆元是一個很重要的運算也有許多運用，最簡單的方法是如以下的窮舉測試：

```
for (b=1; b<P; b++)
    if ((a * b) % P == 1) {
        // b is an inverse
    }
```

窮舉法的問題在於效率太差。在許多運用場合 P 是一個質數，而且探討的整數範圍都只在 0 ~ P-1。在這些假設下有一個重要的數學性質可以幫助我們快速的計算模逆元 (費馬小定理)：「若 P 為質數，對任意正整數 a，a^{P-2} 是 a 在 [1, P-1] 區間的唯一乘法反元素。」根據這個性質搭配快速冪就可以用 O(log(P)) 的運算計算出模逆元。另外一種求模逆元的方法是使用 Extended Euclidean algorithm，這裡就不介紹了。

習題 Q-2-8. 模逆元 (*)

輸入 n 個正整數，以及一個質數 P，請計算每一個輸入數的模逆元。輸入的正整數的大小不超過 P，P<=1000000009，0 < n < 10。

Time limit: 1 秒

輸入格式：第一行是 n 與 P，第二行 n 個整數，同行數字以空白間隔。

輸出格式：依照輸入順序輸出每一個數的模逆元，相鄰數字間間隔一個空白。

範例輸入：

```
3 7
3 4 1
```

範例輸出：

```
5 2 1
```

透過下面的例題，我們要來介紹折半枚舉的技巧，這一題算是有難度的題目。

例題 P-2-9. 子集合乘積(折半枚舉) (@@)

輸入 n 個正整數 A[1..n]，以及一個質數 P，請計算 A 中元素各種組合中，有多少種組合其相乘積除以 P 的餘數等於 1。每個元素可以選取或不選取但不可重複選，A 中的數字可能重複。P<=1000000009，0 < n < 37。

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：滿足條件的組合數，因為數字可能太大，請輸出該組合數除以 P 的餘數。

範例輸入：

```
5 11
1 1 2 6 10
```

範例輸出：

```
7
```

說明乘積等於 1 的組合有：(1), (1), (1,1), (2,6), (1,2,6), (1,2,6), (1,1,2,6)共 7 種

我們可以透過枚舉所有的子集合來測試哪些的組合乘積為 1。在第一章我們說明過以遞迴來枚舉所有子集的方法，他的效率是 $O(2^n)$，在這裡 n 可能達到 36，顯然無法在一秒內完成。以下的範例程式是一個修改的方法，在相同的乘積多的場合他的效率會有所改善，但不足以通過這一題所有測資。這個程式的方法也很直接，我們逐一考慮每一個元素，以一個 map M1 來記錄著目前所有可能的子集合乘積，相同的乘積只記錄一筆，但記錄能產生此乘積的子集合個數。對於下一個元素 a[i]，把原有的所有可能在乘上此新元素，就是新的可能乘積。在計算過程，新的乘積不能直接加到原來的裡面，否則無法分辨新舊。因此要利用一個 map M2 來暫存，當一個元素處理完畢後交換 M1 與 M2，以便下一個元素時再從 M1 計算到 M2。

```
// subset product = 1 mod P, slow method
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;

int main() {
```

```
    int i, n;
    LL p, a[50];
    scanf("%d%lld", &n, &p);
    for (i=0;i<n;i++)
        scanf("%lld", &a[i]);
    map<LL,LL> M1; // (product, number)
    M1[a[0]]=1; // The first element appear once
    for (i=1;i<n;i++) { // for each element
        // compute from M1 to M2
        map<LL,LL> M2(M1); // copy M1 to M2
        for (auto e:M1) {
            LL t=(e.first*a[i])%p;
            M2[t]+=e.second;
        }
        M2[a[i]] +=1; // for {a[i]}
        M1.swap(M2);
    }
    printf("%lld\n", M1[1]);
    return 0;
}
```

在這種場合我們可以用折半枚舉來做的更有效率一點。

對於 n 個數字，我們先將他任意均分為兩半 A 與 B，我們要找的解(子集合乘積等於 1)有三種可能：在 A 中、在 B 中、以及跨 AB 兩端(包含兩邊的元素)。我們對 A 與 B 分別去窮舉它們的子集合乘積，可以找到在 A 中與在 B 中的解。對於跨兩邊的解，我們以下列方式計算：將其中一邊(例如 B)的所有子集合乘積予以排序後，我們對每一個 A 的子集合乘積 x，在 B 的子集合乘積中去搜尋 x 的模逆元(使得 xy = 1 的 y)。

有一點必須留意，我們要把 B 的子集合乘積中，將相同的乘積予以合併，原因是這一題我們需要找出組合數，若相同元素太多，一個 x 需要搜尋很多的模逆元就會太花時間。至於 A 那一邊，合併也可以提升效率，但 worst case 複雜度沒有影響。我們可以算一下整個時間複雜度：對 n/2 個元素窮舉子集合需要 $O(2^{n/2})$，兩邊都做所以乘以 2，排序需要 $O(n*2^{n/2})$，接著對 $2^{n/2}$ 個數字在 $2^{n/2}$ 個數字中做二分搜，需要的時間是 $O(n*2^{n/2})$，所以總時間複雜度是 $O(n*2^{n/2})$，這比起單純窮舉 $O(2^n)$ 要快得多了。以下是範例程式，其中求模逆元是前面介紹過的快速冪方法。

```
// subset product = 1 mod P, O(n*2^(n/2)), sort
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;
LL sa[1<<19], sb[1<<19]; // subset product of a and b

// generate all products of subsets of v[]
// save result in prod[], return length of prod[]
int subset(LL v[], int len, LL prod[], LL p) {
    int k=0; // size of prod[]
    for (int i=0;i<len;i++) {
```

```
        for (int j=0;j<k;j++) { // (each subset)*v[i]
            prod[k+j]=(prod[j]*v[i])%p;
        }
        prod[k+k]=v[i]; // for subset {v[i]}
        k += k+1;
    }
    return k;
}
// find x^y mod P
LL exp(LL x, LL y, LL p) {
    if (y==0) return 1;
    if (y & 1) return (exp(x, y-1,p)*x)%p;
    // otherwise y is even
    LL t=exp(x, y/2, p);
    return (t*t)%p;
}

int main() {
    int i, n;
    LL a[30], b[30]; // input data
    LL p;
    scanf("%d%lld", &n, &p);
    int len_a=n/2;
    int len_b=n-len_a;
    for (i=0;i<len_a;i++)  // half in a
        scanf("%lld", &a[i]);
    for (i=0;i<len_b;i++)  // half in b
        scanf("%lld", &b[i]);
    int len_sa=subset(a,len_a,sa,p); // all subsets of a
    int len_sb=subset(b,len_b,sb,p); // all subsets of a
    sort(sb, sb+len_sb);
    // merge same element of sb, assume not empty
    LL num[1<<19], len_sb2=1;
    num[0]=1; //its multiplicity
    for (i=1;i<len_sb;i++) {
        if (sb[i]!=sb[i-1]) { // new element
            sb[len_sb2]=sb[i];
            num[len_sb2]=1;
            len_sb2++;
        }
        else {
            num[len_sb2-1]++;
        }
    }
    LL ans = (sb[0]==1) ? num[0]: 0; // the number of 1 in sb2
    // compute 1 in sa and cross the two sides
    // for each x in sa, find its inverse in sb2
    for (i=0; i<len_sa; i++) {
        if (sa[i]==1) ans=(ans+1)%p;
        LL y = exp(sa[i], p-2, p); // inverse
        int it = lower_bound(sb, sb+len_sb2, y) - sb;
        if (it<len_sb2 && sb[it]==y)  // found
            ans = (ans + num[it])%p;
    }
    printf("%lld\n", ans);
    return 0;
}
```

這一題當然也可以用 set/map 來做，因為需要合併，我們用 map 來做會比前一支程式方便，時間複雜度則是相同的。這裡的輸入元素放在 vector 中，而子集合窮舉是以遞迴方式寫的，在遞迴到最後時將所產生的子集合乘積放入 map 中。

```
// subset product = 1 mod P, O(n*2^(n/2)), using map
#include<bits/stdc++.h>
using namespace std;
typedef long long LL;
// recursive generate product of subsets of v[0..i]
// current product=prod, result stored in M
void rec(vector<LL> &v, int i, LL prod, map<LL,LL> &M, LL p) {
    if (i>=v.size()) { // terminal condition
        M[prod] += 1; // insert into map
        return;
    }
    rec(v, i+1, (prod*v[i])%p, M, p); // select v[i]
    rec(v, i+1, prod, M, p); // discard v[i]
    return;
}
// find x^y mod P
LL exp(LL x, LL y, LL p) {
    if (y==0) return 1;
    if (y & 1) return (exp(x, y-1,p)*x)%p;
    // otherwise y is even
    LL t=exp(x, y/2, p);
    return (t*t)%p;
}

int main() {
    int i, n;
    vector<LL> a, b; // input data
    LL p;
    scanf("%d%lld", &n, &p);
    for (i=0;i<n/2;i++) { // half in a
        LL t;
        scanf("%lld", &t);
        a.push_back(t);
    }
    for ( ;i<n;i++) { // half in b
        LL t;
        scanf("%lld", &t);
        b.push_back(t);
    }
    map<LL,LL> M1, M2;
    rec(a,0,1,M1,p); // all subsets of a
    rec(b,0,1,M2,p); // all subsets of b
    M1[1] -= 1; // empty set was counted as product 1
    M2[1] -= 1; // empty set
    LL ans=M1[1]+M2[1]; // the number of 1 in both sides
    // compute 1 cross the two sides
    // for each x in M1, find its inverse in M2
    for (auto e: M1) {
        LL x=e.first, num=e.second;
        LL y = exp(x, p-2, p); // inverse of x
        //printf("%lld, %lld; ",x,y);
        auto it=M2.find(y);
```

```
        if (it!=M2.end()) { // found
            ans = (ans + num*it->second)%p;
        }
    }
    printf("%lld\n", ans);
    return 0;
}
```

最後我們給一個類似的習題，這個題目在上一章也出現過，當時的 n<26，現在把他放寬到 38，解法跟前面的例題很類似，注意這裡的 P 不一定也不需要是質數。因為沒有要求模逆元也不必做合併，這題比前面的例題簡單一點，請注意數字的範圍需要使用 long long。

例題 Q-2-10. 子集合的和(折半枚舉)

輸入 n 個正整數 A[1..n]，另外給了一個整數 P，請計算 A 中元素各種組合中，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，A 中的數字可能重複。A[i]與 P 均不超過 2^{60}，$0 < n \leq 38$。

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
```

在例題 P-1-6，我們提出了一個尋找最接近區間和的問題，當時用來展示一些窮舉的技巧，只有提出 O(n²)的方法，以下的例題我們來展示如何有效率的解這個問題。

例題 P-2-11. 最接近的區間和 (*)

輸入一個整數序列(A[1], A[2], …, A[n])，另外給了一個非負整數 K，請計算哪一個連續區段的和最接近 K 而不超過 K。n 不超過 10 萬，數字總和不超過 10 億。

Time limit: 1 秒

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

輸出格式：在所有區間和中，最接近 K 但不超過 K 的和。

範例輸入：

```
5 10
5 -5 8 -3 4
```

範例輸出：

```
9
```

題目並沒說輸入的是正數，這裡的輸入數字當然有可能是負的，否則有另外一種解法(下一章會討論)。一個區段可以用兩端點[l,r]表示，對於可能的右端 r，我們要設法找到最好的左端 l，最好的意思就是讓 A[l:r]的和最接近 K 而不超過 K，如此一來，然後對所有的 r 再找出最好的解就是最佳解。所以問題的核心再如何對任一右端計算出最好的解。

在 P-1-6 中我們介紹過前綴和(Prefix-sum)，令 ps[i]表示前 i 項的和，為方便起見定義 ps[0]=0。對任一右端 r，sum(A[i:r])=ps[r]-ps[i-1]，因此，要使 sum(A[i:r])不超過 K 又最接近 K，就是要找 ps[i-1]≥ ps[r]-K 的最小值。解法呼之欲出了，讓 r 從小往大跑，以資料結構維護好所有 i<r 的 ps[i]，然後找到 ps[r]-K 的 lower_bound()。那麼該用甚麼資料結構呢？因為必須不斷的將 prefix-sum 加入，所以必須以動態的資料結構來處理，set 就會符合需要。時間複雜度是 O(nlog(n))，因為在 set 中搜尋是 O(log(n))。

我們程式其實很簡單，但必須使用到 set(或是其他更複雜的資料結構)，所以這一題基本上算超過 APCS 的考試範圍。

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

int main() {
    int n, psum=0, k, v;
    scanf("%d%d", &n, &k);
    set<int> S({0}); // record the prefix_sum
    int best=0; // solution of empty range
    for (int r=0; r<n; r++) {
        scanf("%d", &v);
        psum += v; //prefix-sum(r)
        auto it=S.lower_bound(psum-k);
        if (it!=S.end()) // found
            best=max(best, psum-*it); //currently best
        S.insert(psum); // insert prefix-sum(r)
    }
```

```
    printf("%d\n",best);
    return 0;
}
```

以下的習題是二維版本的類似題。給一個提示，對所有列的範圍[i, j]，把第 i 列到第 j 列"黏"起來變成一個一維陣列，然後用 P-1-11 的方法去做，這樣的時間複雜度會是 O(M²Nlog(N))。

習題 Q-2-12. 最接近的子矩陣和 (108 高中全國賽) (*)

輸入一個整數二維矩陣 A[M][N]，另外給了一個整數 K，請計算哪一個子矩陣的和，也就是對所有 1≤i≤j≤M and 1≤p≤q≤N， $\sum_{s=i}^{j}\sum_{t=p}^{q}A[s][t]$ 最接近 K 而不超過 K。M ≤ 50 且 M × N ≤ 300,000，每一個整數的絕對值不超過 3,000。

Time limit: 1 秒

輸入格式：每筆測資的第一行有一個正整數 K；第二行有兩個正整數 M 與 N。接下來，由上而下，從左至右，有 M 行輸入，每一行有 N 個整數，每一個整數的絕對值不超過 3,000，代表 A[s][t]，同行整數間以空格隔開。

輸出格式：在所有子矩陣和中，最接近 K 但不超過 K 的和。

範例輸入：

```
6
2 3
-1 -1 1
2 2 2
```

範例輸出：

```
6
```

以下習題是 108 年高中全國賽某一題的核心要算的，是快速冪的一個應用。

習題 Q-2-13. 無理數的快速冪 (108 高中全國賽, simplifed)

若 $s+t\sqrt{2}=(x+y\sqrt{2})^n$，其中 x, y, s, t 均為正整數，輸入 x,y 與 n，請計算並輸出 s 與 t 除以 p 的餘數。p=10⁹+9 且 x,y,n < p。

Time limit: 1 秒

輸入格式：一行含三個正整數，依序為 x, y 與 n，以空格隔開。

輸出格式：s 與 t，中間空一格。

範例輸入：

```
2 3 2
```

範例輸出：

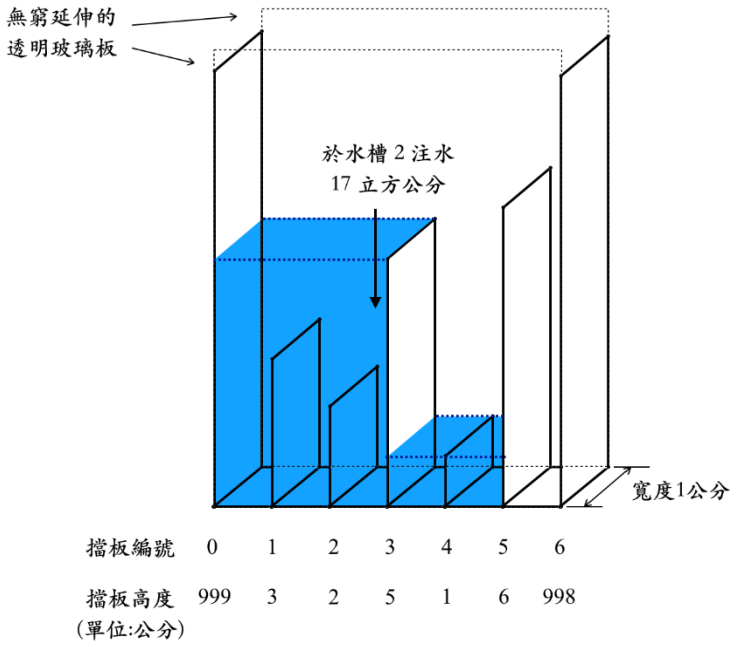
```
22 12
```

下一題是 108 高中全國賽的題目，有多種解法，題目需要一些思考，但其實有個漂亮的解法是排序與遞迴的聯合運用，不需要特別的資料結構與方法。這一題有點難，如果做得出來應該已經超過 APCS 五級分的程度了。

習題 Q-2-14. 水槽 (108 高中全國賽) (@@)

我們用一排 n 個擋板建造水槽。擋板的寬度為 1，高度為正整數且均不相同，水槽前後是兩片長寬均為無限大的玻璃板 (見下圖例)。相鄰擋板的距離都是 1，故相鄰二擋板之間會形成底面積 1 平方的水槽。

擋板由左而右依序由 0 到 $n - 1$ 編號，第 i 及 $i + 1$ 檔板中間的水槽稱為水槽 i。現在將總量為 w 立方公分的水緩緩注入水槽 i。注意水量可能溢出到別的水槽，但是由於所有擋板高度都不同，所以每當溢出時，只會先從一個方向溢出。請計算將總量為 w 立方公分的水緩緩注入水槽 i 後，所有水槽的水深。本題最左的擋板與最右的擋板是所有擋板中最高的兩個，並且保證欲注入的水不會溢出到左右邊界之外；另外，所有水槽的最後水深一定都是整數。以下圖為例，於水槽 2 注入 17 立方公分的水後，各水槽的水深依序為 5, 5, 5, 1, 1, 0。



Time limit: 1 秒

輸入格式：第一行有三個正整數 n (3 ≤ n ≤ 10^5)、i (0 ≤ i ≤ n − 2) 和 w (1 ≤ w ≤ 10^{12})，分別代表擋板數、注水水槽編號，及水量。第二行有 n 個以空白間隔的正整數，代表由左到右擋板的高度，每個擋板高度為正整數且不超過 10^9)。請注意注水量可能超過一個 32-bit 整數的範圍。。

輸出格式：輸出爲一行，共 *n* − 1 個整數，依序代表各個水槽水深，數字之間以一個空白間隔。

範例輸入：

```
8 3 27
9 7 5 3 4 6 8 10
```

範例輸出：

```
0 6 6 6 6 3 0
```

Q_2_14 解題提示：我們維護一個水槽高度尚未被決定的區間[left,right)，區間以外的水槽高度都已經確定。遞迴函數
rec(left, right, water, input)計算水量 water 從編號 input 進入後區間的各高度。

如果區間只剩一個水槽，該水槽高度=水量，結束；

否則，在區間中找出最高的隔板，假設此隔板高度為 H。區間被此隔板分成左右兩邊，考慮下面三種情形：

- water >=(right-left)*H，水量足以讓兩邊都至少 H，區間內所有水槽高度皆是相同的平均值，結束。
- 水量不足以越過輸入這一邊，那麼，另外一邊一定不會有水，縮小區間遞迴呼叫。
- 水量會越過輸入這一邊，那麼，輸入這一邊的水槽高度一定都是 H，把水量扣掉後遞迴呼叫另外一邊。

為了要找出區間內的最高隔板，我們可以一開始把所有隔板的依照高度從高到低排列，每次要找某區間最高隔板時，我們檢視目前最高的隔板，如果在我們要的區間，那就找到了；如果不在我們要的區間，這個隔板以後也不會用到(想想為何)，可以直接丟掉。因此，只要一開始把隔板排序後依序往後找就可以了，不需要特殊的資料結構，但是要把隔板的高度與位置一起存，所以需要一個兩個欄位資料的排序。