

AP325 — 從 APCS 實作題檢測三級到五級

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，.....。

第 1.0 版自序

寫書最難的是：不知道在跟誰說話。而教學最重要的是：見人說人話，見鬼說鬼話，意思是要用對方可以聽得懂的方式來解釋新的觀念與技術，演算法更是如此，因為演算法每個步驟要詳細到什麼程度沒有固定的表達方式。

這一份教材設定的讀者對象是已經對基礎的程式語法與架構有所了解，而教材的目的是透過解題的方式來介紹一些資料結構的使用與演算法的思考方式。以 APCS 實作題檢測的程度來說，教材的對象就是從三級分要進步到五級分的人，或者說是學會基礎程式後想要進入程式競賽的人。教材的主要內容是題目與解題的說明，每一個單元的前面也會有一些基礎知識的說明。雖然是解題為主的教材，但並不是去網路上蒐集題目與題解，而是判斷需要什麼才放什麼。因為也考慮對競賽程式有興趣的入門者，在選題目還是遭遇了一些取舍的困擾，後來有一些超過 APCS 範圍的題材也選了進來。除了一部份標註來源的題目之外，其餘的題目大部分都是常用來教學的題目與經典題，所以都沒有標註來源的問題，程式碼都是我自己寫的，也沒有標註來源的問題。

這份教材的一個好處是所有的題目都附了測資，讀者可以測試自己的程式。因為不想占用太大的空間，幾乎每個題目都只有放五筆測資，其實例題的測資產生程式都附在資料夾中，需要的人可以自己產生測資，部份題目我也放了效率較差的程式。理論上測資不可能防所有的假解，依我的判斷，在合理的狀況下這些測資都夠強了，當然，難免有意外發生的可能。對大部分的題目，時間限制都有很大的寬容度，效率好壞的算法差距應該都在幾十倍以上，但有少部份題目的差距沒這麼大，所以讀者必須自己判斷一下，尤其是要把題目放上裁判系統的人。

倉促之間，錯誤難免，有問題歡迎反映給我。

作者：吳邦一，2020/8/7

有關版權：本教材歡迎分享使用，但未經同意不得做為商業出版與使用。

目錄

0.	教材說明與預備知識	1
0.1.	教材說明	1
0.2.	預備知識	2
0.2.1.	基本 C++ 模板與輸入輸出	3
0.2.2.	程式測試與測試資料	4
0.2.3.	複雜度估算	6
	複雜度估算方式	6
	由資料量大小推估所需複雜度	8
	複雜度需要留意的幾件事	9
0.2.4.	需要留意的事	10
	整數 overflow	10
	浮點數的 rounding error	12
	判斷式的 short-circuit evaluation	12
	編譯器優化	13
1.	遞迴	16
1.1.	基本觀念與用法	16
1.2.	實作遞迴定義	18
1.3.	以遞迴窮舉暴搜	25
2.	排序與二分搜	34
2.1.	排序	34
2.2.	搜尋	39
	基本二分搜的寫法	40
	C++ 的二分搜函數以及相關資料結構	41
	C++ 可供搜尋的容器 (set/map) (*)	46

2.3.	其他相關技巧介紹.....	51
	Bitonic sequence 的搜尋	51
	快速冪	51
	快速計算費式數列.....	53
2.4.	其他例題與習題.....	54
3.	佇列與堆疊.....	72
3.1.	基本原理與實作.....	72
	佇列 (queue)	72
	堆疊 (Stack)	74
	雙向佇列 (deque)	76
3.2.	應用例題與習題.....	76
	串列鏈結 (Linked list)	91
	滑動視窗 (Sliding window)	94
4.	貪心演算法與掃描線演算法.....	108
4.1.	基本原理.....	108
	優先佇列 (priority_queue)	110
4.2.	例題與習題.....	112
4.2.1.	單欄位資料排序.....	113
4.2.2.	結構資料排序.....	116
4.2.3.	以 PQ 處理動態資料(*).....	122
4.2.4.	外掛二分搜.....	126
4.2.5.	掃描線演算法 sweep-line	129
4.3.	其他習題.....	140
5.	分治演算法.....	144
5.1.	基本原理.....	144
	分治的複雜度	146

	架勢起得好，螳螂鬥勝老母雞	147
5.2.	例題與習題	148
	口渴的烏鴉	148
5.3.	補充說明	160
6.	動態規劃	161
6.1.	基本原理	161
6.1.1.	基本思維與步驟	161
6.1.2.	狀態轉移	164
6.1.3.	分類與複雜度	165
6.1.4.	Top-down memoization	166
6.2.	例題與習題	168
6.2.1.	1D0D	168
6.2.2.	2D0D	175
6.2.3.	1D1D	183
6.2.4.	2D1D 與其他	197
6.3.	進階題(*)	210
7.	基本圖論演算法	219
7.1.	基本觀念與名詞	219
7.2.	資料結構與基本演算法	222
7.2.1.	資料結構	222
	不使用 vector 的鄰接串列做法 (*)	225
7.2.2.	BFS	225
7.2.3.	DFS	230
7.2.4.	DAG 與 topological sort	235
7.3.	例題與習題	237
	DAG 上的路徑問題	242
7.4.	補充教材(*)	247

7.4.1.	Dijkstra 演算法.....	248
7.4.2.	併查集(Union and Find).....	253
7.4.3.	最小生成樹.....	260
8.	樹上演算法.....	267
8.1.	基本觀念與名詞.....	267
8.2.	樹的儲存與基本演算法.....	269
8.2.1.	儲存樹的資料結構.....	269
8.2.2.	DFS.....	271
8.2.3.	BFS.....	274
8.2.4.	由下而上的走訪.....	279
8.3.	例題與習題.....	284

例題與習題目錄

P-1-1. 合成函數 (1)	18
Q-1-2. 合成函數 (2) (APCS201902)	20
P-1-3. 棍子中點切割	20
Q-1-4. 支點切割 (APCS201802) (@@)	23
Q-1-5. 二維黑白影像編碼 (APCS201810)	24
P-1-6. 最接近的區間和	25
P-1-7. 子集合乘積	27
Q-1-8. 子集合的和 (APCS201810, subtask)	28
P-1-9. N-Queen 解的個數	29
Q-1-10. 最多得分的皇后	31
Q-1-11. 刪除矩形邊界 - 遞迴 (APCS201910, subtask)	32
P-2-1. 不同的數-排序	38
P-2-2. 離散化 - sort	44
P-2-2C. 離散化 - set/map (*)	50
P-2-3. 快速幂	52
Q-2-4. 快速幂--200 位整數	53
Q-2-5. 快速計算費式數列第 n 項	54
P-2-6. Two-Number problem	55
Q-2-7. 互補團隊 (APCS201906)	56
Q-2-8. 模逆元 (*)	58
P-2-9. 子集合乘積 (折半枚舉) (@@)	59
Q-2-10. 子集合的和 (折半枚舉)	63
P-2-11. 最接近的區間和 (*)	63

Q-2-12. 最接近的子矩陣和 (108 高中全國賽) (*)	65
Q-2-13. 無理數的快速幕 (108 高中全國賽, simplified)	65
Q-2-14. 水槽 (108 高中全國賽) (@@)	66
P-2-15. 圓環出口 (APCS202007)	68
P-3-1. 樹的高度與根 (bottom-up) (APCS201710)	76
P-3-2. 括弧配對	80
Q-3-3. 加減乘除	83
P-3-4. 最接近的高人 (APCS201902, subtask)	84
Q-3-5. 帶著板凳排雞排的高人 (APCS201902)	87
P-3-6. 砍樹 (APCS202001)	88
P-3-7. 正整數序列之最接近的區間和	95
P-3-8. 固定長度區間的最大區段差	96
P-3-9. 最多色彩帶	98
P-3-10. 全色彩帶 (需離散化或字典) (@@)	100
Q-3-11. 最長的相異色彩帶	105
Q-3-12. 完美彩帶 (APCS201906)	105
Q-3-13. X 差值範圍內的最大 Y 差值	106
Q-3-14. 線性函數 (@@)	107
P-4-1. 少林寺的代幣	108
P-4-2. 笑傲江湖之三戰	113
P-4-3. 十年磨一劍 (最少完成時間)	114
P-4-4. 幾場華山論劍 (activity selection)	116
P-4-5. 嵩山磨劍坊的問題 (加權最小完成時間)	119
Q-4-6. 少林寺的自動寄物櫃 (APCS201710)	121
P-4-7. 岳不群的併派問題 (Two-way merge) (*)	123

Q-4-8. 先到先服務 (*)	125
P-4-9. 基地台 (APCS201703)	126
Q-4-10. 恢復能量的白雲熊膽丸	128
P-4-11. 線段聯集 (APCS 201603)	129
P-4-12. 一次買賣	132
P-4-13. 最大連續子陣列	133
P-4-14. 控制點 (2D-max)	135
P-4-15. 最靠近的一對 (closest pair) (@@)	137
Q-4-16. 賺錢與罰款	140
Q-4-17. 死線高手	140
Q-4-18. 少林寺的櫃姐 (@@) (*)	141
Q-4-19. 五嶽盟主的會議場所	142
Q-4-20. 監看華山練功場	142
P-5-1. 最大值與最小值	145
P-5-2. 最大連續子陣列 (分治) (同 P-4-13)	147
P-5-3. 合併排列法	149
P-5-4. 反序數量 (APCS201806)	150
Q-5-5. Closest pair (同 P-4-15, 分治版) (@@)	153
P-5-6. 線性函數 (同 Q-3-14, 分治版)	154
P-5-7. 大樓外牆廣告 (分治版)	156
Q-5-8. 完美彩帶 (同 Q-3-12, 分治版) (APCS201906)	159
P-6-1. 小朋友上樓梯最小成本	168
P-6-2. 不連續的表演酬勞	170
P-6-3. 最小監控鄰居的成本	171
Q-6-4. 闖關二選一	173

Q-6-5. 二維最大子矩陣	174
P-6-6. 方格棋盤路線	175
P-6-7. LCS	177
Q-6-8. Local alignment	180
P-6-9. 大賣場免費大搬家	180
Q-6-10. 置物櫃出租 (APCS201810)	183
P-6-11. Catalan number	184
Q-6-12. 楊鐵心做 1 休 K	185
P-6-13. 周伯通的基地台 (@@)	186
Q-6-14. K 次買賣 (106 高中全國賽 subtask)	189
P-6-15. 一覽衆山小	190
P-6-16. 山寨華山論劍	194
P-6-17. 切棍子	197
Q-6-18. 矩陣乘法鏈	200
P-6-19. 階梯數字 (APCS201802) (@@)	201
P-6-20. Hyper-cube path	205
P-6-21. 刪除邊界 (APCS201910)	207
P-6-22. 文無第一武無第二 (@@) (*)	210
Q-6-23. 直升機抓寶 (@@) (*) (105 高中全國賽)	213
Q-6-24. 隔離採礦 (@@@) (108 全國高中賽)	215
Q-6-25. 貨郎問題 (@@)	216
P-7-1. 探索距離	227
P-7-2. 開車蒐集寶物	232
P-7-3. 機器人走棋盤 (APCS 201906)	237
P-7-4. 方格棋盤的最少轉彎數路線	239

Q-7-5. 闖關路線 (APCS201910)	241
P-7-6. DAG 的最長與最短路徑	243
Q-7-7. AOV 最早完工時間	246
Q-7-8. 小寶的著色問題	247
P-7-9. 最短路徑 (*)	249
P-7-10. 挖坑跳 (@@) (TOI 入營考)	255
Q-7-11. 紅白彩帶 (APCS201902)	259
P-7-12. 最小生成樹 (*)	263
P-8-1. 樹上的推銷員	271
P-8-2. 物流派送系統	275
P-8-3. 購物中心選位置	279
P-8-4. 樹的高度與根 (同 P-3-1) (APCS201710)	284
P-8-5. 自動分裝 (APCS202002)	288
Q-8-6. 樹狀圖的距離總和	291
P-8-7. 寶石的顏色 (108 全國高中賽)	292
P-8-8. 樹的最大獨立集	295
Q-8-9. 服務中心選位置	298
Q-8-10. 竊聽間諜網	299
P-8-11. 公司派對 (NCPC)	300
Q-8-12. 佔領連續的城鎮	303
P-8-13. 不同成本的購物中心 (@@)	304
P-8-14. 血緣關係 (APCS201603)	307
Q-8-15. 樹上一位不回家的推銷員	311
Q-8-16. 病毒演化 (APCS202007)	312

註：P 開頭為例題，Q 開頭為習題。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 或其他出處者) 表示該題出現於過去考試或比賽的類似題，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。

Bangye.Wu

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、圖的走訪、以及樹狀圖。
- 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： $O(n)$ 、 $O(n\log(n))$ 、 $O(n^2)$ 、 $O(2^n)$ 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 $f(n) \geq g(n)$ ，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 $O(f(n) \times g(n))$ 。
- 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 $O(2 \times n) = O(n)$ 。(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 $10^6 \sim 10^7$ 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 $O(n \log(n))$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n!)$

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 $O(n \log(n))$ ；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 $O(n \log(n))$ ，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/qsrt` 的複雜度可以用 $O(n \log(n))$ 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 $1+2+\dots+n$ 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 <stdio>
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 $a[i] \neq x$ 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 <stdio>
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 $(i < 5 \ \&\& \ a[i] \neq x)$ ，會先計算 $i < 5$ ，如發現不成立就不會去計算 $a[i] \neq x$ 。但如果像程式裡面反過來寫，就會先計算 $a[i] \neq x$ ，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)。編譯器优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， $\text{strlen}(s)$ 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 $\text{strlen}(s)$ 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，編譯器的優化功能會發現字串長度在迴圈中沒有被改變的可能，因此 $\text{strlen}(s)$ 的動作被移到迴圈之外，所以執行檔中 $\text{strlen}(s)$ 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 $\text{strlen}(s)$ ，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

「遞迴是小事化小，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 ， $\text{fac}(n)$ 定義為所有不大於 n 的正整數的乘積，也就是

$$\text{fac}(n) = 1 \times 2 \times 3 \times \dots \times n。$$

如果採遞迴的定義，則是：

$$\begin{aligned} \text{fac}(1) &= 1; \text{ and} \\ \text{fac}(n) &= n \times \text{fac}(n-1) \text{ for } n > 1。 \end{aligned}$$

若以 $n=3$ 為例，直接的定義告訴你如何計算 $\text{fac}(3)$ ，而遞迴的定義並不直接告訴你 $\text{fac}(3)$ 是多少，而是

$$\text{fac}(3) = 3 \times \text{fac}(2)，\text{而}$$

$$\text{fac}(2) = 2 \times \text{fac}(1)，\text{最後才知道}$$

$$\text{fac}(1) = 1。$$

所以要得到 $\text{fac}(3)$ 的值，我們再逐步迭代回去，

$$\begin{aligned} \text{fac}(2) &= 2 \times \text{fac}(1) = 2 \times 1 = 2， \\ \text{fac}(3) &= 3 \times \text{fac}(2) = 3 \times 2 = 6。 \end{aligned}$$

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：

$$F(1) = F(2) = 1;$$

$$F(n) = F(n-1) + F(n-2) \text{ for } n > 2.$$

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

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。

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。

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 $p[1] \sim p[N]$ ，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 。 $N \leq 5e4$ ， $L < 1e9$ 。

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

$$x = (p[\text{right}] + p[\text{left}]) / 2$$

所以我們要找某個 m ，滿足 $p[m-1] < x \leq p[m]$ ，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，但二分搜尋是更快的方法，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 $p[m] \geq x$ ；第二是內迴圈要用 `while` 而不能只用 `if`，因為事實上內迴圈最多會做兩次。

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 $\geq x$ 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。

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，但不可以是兩端點，也就是說，若區段的範圍是 $[s, t]$ ，則要找出切點 $m \in [s+1, t-1]$ ，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。所謂切割層級的限制，第一次切以第一層計算。

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 $p[1] \sim p[N]$ ，數字間以空白隔開，總和不超過 10^9 。 $N \leq 50000$ ，切割層級限制 $K < 30$ 。

輸出：所有切點的 $p[]$ 值總和。

範例一輸入： 7 3 2 4 1 3 7 6 9	範例一輸出： 11
範例二輸入： 5 1 1 2 3 4 100	範例二輸出： 4

範例一說明：第一層切在 7，切成第二層的 $[2, 4, 1, 3]$ 與 $[6, 9]$ 。左邊 $[2, 4, 1, 3]$ 切在 4 與 1 都是最佳，選擇 4 來切，切成 $[2]$ 與 $[1, 3]$ ，右邊 $[6, 9]$ 不到 3 個就不切了。第三層都不到 3 個，所以終止。總計切兩個位置 $7+4=11$ 。

範例二說明：第一層切在 4 (注意切點不可在端點)，切出 $[1, 2, 3]$ 與 $[100]$ ，因為 $K=1$ ，所以不再切。

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。

Q-1-5. 二維黑白影像編碼 (APCS201810)

假設 n 是 2 的幕次，也就是存在某個非負整數 k 使得 $n = 2^k$ 。將一個 $n \times n$ 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 ， $1 \leq n \leq 1024$ ，中 n 必為 2 的幕次。

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難，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。

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 $1 \leq i \leq j \leq n$ 。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 $O(n^2)$ 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

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

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。

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 $O(n \cdot 2^n)$ 。

```
// 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;
}
```

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。 $P \leq 1000000009$, $0 < n < 26$ 。

Time limit: 1 秒

輸入格式：第一行是 n 與 P ，第二行是 n 個可挑選的正整數，大小不會超過 P ，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 $N \times N$ 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。

P-1-9. N-Queen 解的個數

計算 N-Queen 有幾組不同的解，對所有 $0 < N < 12$ 。

(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 $0 \sim N-1$ 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<12;i++)
        printf("%d ",nq(i));
    return 0;
}

```

接著看遞迴的寫法，每呼叫一次遞迴，我們檢查這一系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<12;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 $O(n \times n!)$ ，而且實際上比 $O(n \times n!)$ 快很多，因為你可以看得到，在遞迴過程中，被已放置皇后攻擊的位置都被略去了，所以實際上並未嘗試所有排列。這種情況下我們知道他的複雜度上限，但真正準確的複雜度往往難以分析，如果以實際程式來量測， $n=11$ 時， $n!=39,916,800$ ，但遞迴被呼叫的次數只有 16,6926 次，差了非常多。

```
// 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。

Q-1-10. 最多得分的皇后

在一個 $n \times n$ 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。 $0 < n < 11$ 。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，本題不限恰好 N 個皇后，所以遞迴搜尋過程要考慮某些列不放皇后的可能)

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。

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最右行。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。

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。 $m + n \leq 13$ 。

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）

Bangye.Wu

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，包括快速幕與折半枚舉。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，如果第一個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("(%d, %d) ",a[i].first, a[i].second);
07     printf("(%d, %d)\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。

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，第二行由小到大輸出這些相異整數，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
-1 0 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。

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, 3, 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，它其實是一個稱為平衡的二元搜尋樹 (BST, Binary Search Tree) 的動態的資料結構，所謂動態資料結構指的是可以 (有效率的) 做插入與刪除。在這裡我們著重在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，但是它其實是不同的，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一點可以提醒：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。

P-2-2C. 離散化 - set/map (*)

題目敘述請見 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 $x^y \pmod{P}$ 。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。

P-2-3. 快速冪

輸入正整數 x, y , 與 p , 計算 $x^y \pmod{p}$ 。 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 $x^{19} = x^{16} * x^2 * x^1$ 。

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 $((((1*10\%p+2)\%p)*10)\%p+3)\%p$ 。

Q-2-4. 快速幂--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：

123456789012345678901234567890 5 11

範例輸出：

10

快速計算費式數列

利用快速幂我們可以計算費式數列的第 n 項。

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

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

將上式迭代展開，因為矩陣滿足結合律，所以可以得到

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

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

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

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。

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 $a \in A, b \in B$ 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 A 中的每一個 a ，在 B 中以滑動的方式找 $K-a$ 。
- 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。

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 ， $2 \leq m \leq 26$ ， $1 \leq n \leq 50000$ 。第二行開始有 n 行，每行一個字串代表一個團隊，每個字串的长度不超過 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 CEF GH。
- 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
    team |= 1<<(s[j] - 'A'); // set bit to 1
complement = ff - team; // 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 $0 \sim P-1$ 。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 ， $(a^{P-2} \% P)$ 是 a 在 $[1, P-1]$ 區間的唯一乘法反元素。」根據此性質搭配快速幂就可以用 $O(\log(P))$ 個運算計算出模逆元。另一種求模逆元的方法是用 Extended Euclidean algorithm，這裡就不介紹了。

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 ， $P \leq 1000000009$ ， $0 < n < 10$ 。

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
```

註：費馬小定理屬於數論中的定理，應該不在 APCS 考試的背景知識內，但如果題目中給予提示說明，則模逆元並非屬於一定不可以考的範圍。

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。

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。 $P \leq 1000000009$ ， $0 < n < 37$ ，且假設 A 中元素皆小於 P 。

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] = (M2[t] + e.second)%p;
        }
        M2[a[i]] = (M2[a[i]] + 1)%p; // 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 $O(n \cdot 2^{n/2})$ ，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 $O(n \cdot 2^{n/2})$ ，所以總時間複雜度是 $O(n \cdot 2^{n/2})$ ，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幂方法。

```

// 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]%p : 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])%p; // 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。

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 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 $O(n^2)$ 的方法，以下的例題我們來展示如何有效率的解這個問題。

P-2-11. 最接近的區間和 (*)

輸入一個整數序列 ($A[1], A[2], \dots, A[n]$)，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 $ps[i-1] \geq ps[r]-K$ 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 $O(n \log(n))$ ，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。

```
#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-2-11 的方法去做，這樣的時間複雜度會是 $O(M^2N \log(N))$ 。

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 $1 \leq i \leq j \leq M$ and $1 \leq p \leq q \leq N$ ， $\sum_{s=i}^j \sum_{t=p}^q A[s][t]$ 最接近 K 而不超過 K 。
 $M \leq 50$ 且 $M \times N \leq 300,000$ ，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，是快速幂的一個應用。

Q-2-13. 無理數的快速幂 (108 高中全國賽, simplified)

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。 $p=10^9+9$ 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。

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 $i + 1$ 擋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。

無窮延伸的透明玻璃板

於水槽 2 注水
17 立方公分

寬度1公分

擋板編號	0	1	2	3	4	5	6
擋板高度	999	3	2	5	1	6	998

(單位:公分)

Time limit: 1 秒

輸入格式：第一行有三個正整數 n ($3 \leq n \leq 10^5$)、 i ($0 \leq i \leq n - 2$) 和 w ($1 \leq w \leq 10^{12}$)，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：輸出為一行，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：

- $\text{water} \geq (\text{right} - \text{left}) * H$ ，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。

P-2-15. 圓環出口 (APCS202007)

有 n 個房間排列成一個圓環，以順時針方向由 0 到 $n - 1$ 編號。玩家只能順時針方向依序通過這些房間。每當離開第 i 號房間進入下一個房間時，即可獲得 $p(i)$ 點。玩家必須依序取得 m 把鑰匙，鑰匙編號由 0 至 $m-1$ ，兌換編號 i 的鑰匙所需的點數為 $Q(i)$ 。一旦玩家手中的點數達到 $Q(i)$ 就會自動獲得編號 i 的鑰匙，而且手中所有的點數就會被「全數收回」，接著要再從當下所在的房間出發，重新收集點數兌換下一把鑰匙。遊戲開始時，玩家位於 0 號房。請計算玩家拿到最後一把鑰匙時所在的房間編號。

以下是一個例子。有 7 個房間， $p(i)$ 依序是 (2, 1, 5, 4, 3, 5, 3)，其中 0 號房間的點數是 2。假設所需要的鑰匙為 3 把， $Q(i)$ 依序是 (8, 9, 12)。從 0 號房出發，在「離開 2 號房，進入 3 號房」時，獲得恰好 $2+1+5 = 8$ 點，因此在進入 3 號房時玩家兌換到 0 號鑰匙；接著從 3 號房開始繼續累積點數，直到「離開 5 號房，進入 6 號房」時，手中的點數為 12，於是在進入 6 號房時獲得 1 號鑰匙，手中點數再次被清空。最後，從 6 號房出發，直到「離開 3 號房，進入 4 號房」時，方可獲得至少 12 點的點數，來兌換最後一把鑰匙。因此，拿到最後一把鑰匙時所在的房間編號為 4。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 n 與 m ，第二行有 n 個正整數，依序是各房間的點數 $p(i)$ ，第三行有 m 個正整數依序是各鑰匙需要的點數 $Q(i)$ ，。同一行連續二數

字間以空白隔開。n 不超過 $2e5$ ，m 不超過 $2e4$ ， $p(i)$ 總和不超過 $1e9$ ， $Q(i)$ 不超過所有 $p(i)$ 總和。

輸出格式：輸出拿到最後一把鑰匙時所在的房間編號。

範例輸入：

```
7 3
2 1 5 4 3 5 3
8 9 12
```

範例輸出：

```
4
```

簡單說，這個題目就是每次要在一個正整數陣列中，對於某個開始位置，找到一個最小的區間，滿足區間和大於等於某輸入 $Q(i)$ 。關於圓環，有兩個簡單的處理方式，一個是判斷超過結尾時，從頭開始；另外一種方式是將陣列重複兩次，每次找到後除以 n 取餘數。最簡單的搜尋方法就是一個一個往下找，程式也很好寫，但是效率不夠好，以下是範例程式。

```
#include <stdio>
#define N 200010
int p[N];

int main() {
    int i, n, m;
    scanf("%d%d", &n, &m);
    for (i=0; i<n; i++)
        scanf("%d", &p[i]);
    int room=0; // current room
    for (i=0; i<m; i++) {
        int q;
        scanf("%d", &q);
        while (q>0) {
            q -= p[room++];
            if (room==n) room=0; //the next of n-1
        }
    }
    printf("%d\n", room);
    return 0;
}
```

要有效率的搜尋，腦海中浮現的自然是二分搜，但是二分搜必須是在單調序列上才能使用，而且我們要找的是區間和。在 P-1-6 中我們介紹過前綴和 (Prefix-sum)，如果我們把所有房間的點數改成計算出前綴和，那麼正數的前綴和必然是遞增的，而我們要

找的區間和也自然的轉換成找一個前綴和。關於二分搜，我們推薦前面介紹的一路往前跳的寫法，以下是範例程式。第 10 行的迴圈計算前綴和，然後第 13 行的迴圈處理每一次需求的點數，在第 15 行，如果目前房間不是 0，就將所需的點數 q 加上前一個房間的前綴和，這就是我們要搜尋的前綴和；因為本題是圓環，在第 17 行我們判斷是否需求會超過陣列尾端從 0 開始。接著進行二分搜，因為我們要始終保持 $p[\text{room}] < q$ 這個特性，跳躍結束時所在位置是最後一個小於 q 的，所以根據題意，要加 2 才是下一個開始位置。要小心在二分搜開始往前跳之前，需要先檢查目前位置是否已經滿足需求。

```

00 // P-2-15, repeatedly search range sum in circular array
01 #include <cstdio>
02 #define N 500010
03 int p[N];
04
05 int main() {
06     int i, n, m;
07     scanf("%d%d", &n, &m);
08     for (i=0; i<n; i++)
09         scanf("%d", &p[i]);
10     for (i=1; i<n; i++)
11         p[i] += p[i-1];
12     int room=0, q, total=p[n-1];
13     for (i=0; i<m; i++) {
14         scanf("%d", &q);
15         if (room != 0) // desired prefix-sum
16             q += p[room-1];
17         if (q > total) // over the end
18             room = 0, q -= total;
19         // find the last one < q
20         if (p[room] >= q) { // current room is enough
21             room = (room+1)%n;
22             continue;
23         }
24         // binary search
25         for (int jump=(n-room)/2; jump>0; jump/=2) {
26             while (room+jump<n && p[room+jump]<q)
27                 room += jump;
28         }
29         room = (room+2)%n; // room+1 is the first >=q
30     }
31     printf("%d\n", room);
32     return 0;
33 }
34

```

我們當然也可以用 STL 的 `lower_bound` 來做二分搜，以下是範例程式，這裡我們以將原來陣列複製一份在後面的方式來處理圓環轉過尾端從開始的問題。其他的寫法都類似。

```

// P-2-15, using extended array and lower_bound
#include <bits/stdc++.h>
using namespace std;
#define N 500010
int p[N];

int main() {
    int i, n, m;
    scanf("%d%d",&n, &m);
    for (i=0; i<n; i++)
        scanf("%d", &p[i]);
    for (i=0; i<n; i++) // double array
        p[n+i] = p[i];
    // prefix sum
    for (i=1; i<2*n; i++)
        p[i] += p[i-1];
    int room=0, q;
    for (i=0; i<m; i++) {
        scanf("%d", &q);
        if (room != 0) // desired prefix-sum
            q += p[room-1];
        // binary search the first >= q
        room = lower_bound(p+room, p+2*n, q) - p;
        room = (room+1)%n; // adjust
    }
    printf("%d\n",room);
    return 0;
}

```

3. 佇列與堆疊

想到堆疊，想到深度；想到深度就想到歐陽修這首蝶戀花：

「庭院深深深幾許，楊柳堆煙，簾幕無重數。玉勒雕鞍遊冶處，樓高不見章臺路。
兩橫風狂三月暮，門掩黃昏，無計留春住。淚眼問花花不語，亂紅飛過鞦韆去。」

關於窗戶：

「Where there's will, there's a way;
where there's way, there's a wall;
where there's wall, there's a window.」

這一章介紹三個基本的資料結構：佇列 (queue)、堆疊 (stack) 與雙向佇列 (deque，念做 "deck"，正式名稱是 double-ended queue)，除了說明基本原理與應用外，也介紹一種與雙向佇列類似的演算法：滑動視窗 (sliding window)。另外我們也介紹鏈結串列 (linked list)，這是一種在實際用途很廣，但是在解題上比較少用的資料結構。這幾種資料結構都是課本上的基本資料結構，有很多應用。滑動視窗這個名字倒是有很多不同的稱呼方式，也有稱為爬行法，或認為是雙指針 (two-pointers) 方法的一種。反正名稱不是那麼重要，懂得原理與會運用才是最重要的事。我們先介紹基本原理與實作方式，然後再來看應用的題目。

3.1. 基本原理與實作

佇列、堆疊與雙向佇列都是簡單的資料結構，簡單到可以自己實作，但也有 STL 的容器可以使用，自己實作的在碰到空間問題時會比較麻煩，所以使用 STL 或許比較好一點。不過在解題的場合，很少會碰到空間問題，因此兩種方式都可以。這三種資料結構都是 "線性" 的，可以在一維陣列實作。鏈結串列最大的優點是將線性的資料放在不連續的位置，這特性在實際系統與軟體中有時非常重要，也因如此，串列需要以指標與記憶體配置來搭配。但解題的場合很少有空間的問題，也幾乎都避免使用 pointer，所以串列部分我們只說明解題中常見的方法。

佇列 (queue)

佇列就好像是一群排隊排成一列，有一個出口與一個入口，通常出口稱為前端 (front) 而入口稱謂後端 (back)。成員進入時只能從入口進入，也就是加入尾端；離

開時只能依序從出口離開，佇列的特性是先進先出 (FIFO, first-in first-out)。與陣列一樣，在標準的佇列中，所有的成員必須是相同的資料型態。佇列通常需要四個基本操作，新增、刪除、查看出口的成員、以及檢查是否為空，有時也會查詢目前成員數。

我們可以很簡單的以陣列來實作佇列，在解題的場合，空間通常不是問題，所以我們宣告一個足夠大的陣列，初始時後端在陣列開頭的位置，每次加入時，後端往後移，每次刪除時，從陣列前端離開，在一連串操作時，佇列相當於在此陣列中往後滑動。我們需要維護好兩個變數，front 記著佇列前端出口位置，back 記著佇列後端入口位置。請看下面的範例程式展示著自製佇列的基本用法以及支援的四個操作。我們沿用 STL 的名詞，加入稱為 push，刪除稱為 pop，這名字其實來自堆疊。

```
// self-made queue
#include <bits/stdc++.h>
#define N 1000
int main() {
    int Q[N], front=0, back=0;
    int n, t;
    char cmd[10];
    scanf("%d", &n);
    for (int i=0;i<n;i++) {
        scanf("%s", cmd);
        if (strcmp(cmd,"push")==0) {
            scanf("%d", &t);
            Q[back++]=t; // push(t)
        }
        else if (strcmp(cmd,"pop")==0) {
            if (front==back) printf("empty\n");
            else {
                t=Q[front++]; // pop to t
                // save Q[front] to t, and then front++
                printf("%d\n",t);
            }
        }
    }
    return 0;
}
```

這個程式在檔案中有輸入資料可以測試，它有若干個命令，每個命令是 push 一個數字或是 pop，程式讀取每一個命令，如果是 push 就將資料加入 queue，如果是 pop 要先檢查佇列是否是空的，否則刪除資料並輸出。下面是以 STL 提供的 queue 來做一樣的事，目的只是展示如何操作。宣告時 queue<int> Q，意義就是宣告一個 int 型態的 queue 名字叫做 Q。

```
// ch3_2 STL queue
```

```

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

int main() {
    queue<int> Q;
    int n, t;
    char cmd[10];
    scanf("%d", &n);
    for (int i=0;i<n;i++) {
        scanf("%s", cmd);
        if (strcmp(cmd,"push")==0) {
            scanf("%d", &t);
            Q.push(t);
        }
        else if (strcmp(cmd,"pop")==0) {
            if (Q.empty()) printf("empty\n");
            else {
                t=Q.front();
                Q.pop();
                printf("%d\n",t);
            }
        }
    }
    return 0;
}

```

堆疊 (Stack)

堆疊像是把子彈放入子彈夾(可能沒看過)，或者想像把硬幣堆疊在桌上。堆疊的出口與入口相同，就是最上面那一個，通常稱為 top。成員進入時只能從上方壓入(往上疊)，稱為 push；離開時只有頂端 top 的成員可以離開。堆疊的特性是後進先出(LIFO, last-in first-out)。在標準的堆疊中，所有的成員也必須是相同的資料型態。堆疊通常需要四個基本操作，push、pop、查看 top 的成員、以及檢查是否為空。

以陣列來實作堆疊比佇列更簡單，因為只有一端開口，我們將堆疊的底部設在陣列開頭之處，每次 push 時 top 往後移，每次 pop 時，top 往前移，在一連串操作時，堆疊只是 top 位置前後移動，像是一端黏在牆壁的彈簧，我們只要維護好一個變數 top 就可以了。請看下面的範例程式，它的測試資料與佇列相同。不熟悉的讀者可以用範例程式來了解用法。

```

// ch3_3 self-made stack, test data = ch3_1.in
#include <bits/stdc++.h>
#define N 1000
int main() {
    int S[N], top=-1;

```

```

int n, t;
char cmd[10];
scanf("%d", &n);
for (int i=0;i<n;i++) {
    scanf("%s", cmd);
    if (strcmp(cmd,"push")==0) {
        scanf("%d", &t);
        S[++top]=t; // push(t)
    }
    else if (strcmp(cmd,"pop")==0) {
        if (top<0) printf("empty\n");
        else {
            t=S[top--];
            printf("%d\n",t);
        }
    }
}
return 0;
}

```

下面的範例是以 STL 提供的 stack 來做一樣的事。宣告時 `stack<int> S`，意義就是宣告一個 `int` 型態的 stack 名字叫做 `S`。

```

// ch3_4 STL stack, test data at ch3_1.in
#include <bits/stdc++.h>
using namespace std;
#define N 1000
int main() {
    stack<int> S;
    int n, t;
    char cmd[10];
    scanf("%d", &n);
    for (int i=0;i<n;i++) {
        scanf("%s", cmd);
        if (strcmp(cmd,"push")==0) {
            scanf("%d", &t);
            S.push(t); // push(t)
        }
        else if (strcmp(cmd,"pop")==0) {
            if (S.empty()) printf("empty\n");
            else {
                t=S.top();
                S.pop();
                printf("%d\n",t);
            }
        }
    }
    return 0;
}

```

雙向佇列 (deque)

如果了解了 queue 的操作之後，deque 的操作就容易了解了，它與 queue 很類似，差異在它的前後都可以進出，所以加入與刪除的操作有 `push_front`，`push_back`，`pop_front`，`pop_back` 四種，這些操作的意義都可以從名字上了解。雙向佇列建議直接使用 STL，因為自己做的話，會需要處理空間的問題，比較麻煩，除非使用時已知它只會從一端加入，那就跟 queue 的製作很類似了。因為與 queue 很類似，這裡我們就不再提供範例，在後面的題目中會碰到。

3.2. 應用例題與習題

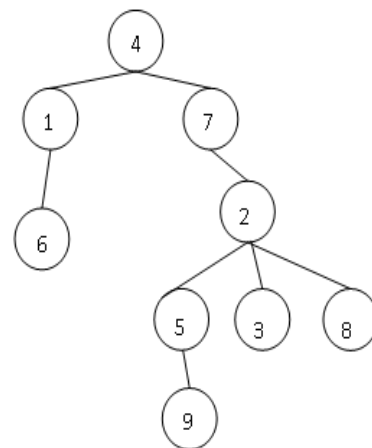
佇列堆疊與雙向佇列常常是用其他演算法裡面做為一個基本步驟，單獨出現的不算太多。尤其單純 queue 的題目並不多，除非是簡單的操作模擬，但是在樹與圖的走訪卻是非常重要的，這些要等到後續單元介紹了圖與樹的基本概念後再介紹比較合適。這裡提出一個例題，事實上是 tree 的題目，因為不需要太多專業術語，可以先放在此處當作 queue 的例題，等到後面的章節會再提出其他解法。

P-3-1. 樹的高度與根 (bottom-up) (APCS201710)

有一個大家族有 N 個成員，編號 $1 \sim N$ ，我們請每個成員寫下此家族中哪幾位是他的孩子。我們要計算每個人有幾代的子孫，這個值稱為他在這個族譜中的高度，也就是說，編號 i 的人如果高度記做 $h(i)$ ，那麼 $h(i)$ 就表示在所有 i 的子孫中，最遠的子孫與他隔了幾代。本題假設除了最高的一位祖先（稱為 root）之外，其他成員的 parent 都在此家族中（且只有一個 parent）。本題要計算所有成員高度的總和以及根的編號。

右圖是一個例子，每個成員是一個圈起來的號碼，劃線相聯的表示上面的點是下面點的 parent。編號 4 是根，有兩個孩子 1 與 7。編號 6, 9, 3, 8 都沒有孩子， $h(6)=h(9)=h(3)=h(8)=0$ ，此外 $h(2)=2$ 因為 9 與他隔兩代。你可以看出來 $h(5)=h(1)=1$ ， $h(7)=3$ ， $h(4)=4$ ，所以高度總和是 11。

Time limit: 1 秒



輸入格式：第一行有一個正整數 N 。接下來有 N 行，第 i 行的第一個數字 k 代表 i 有 k 個孩子，第 i 行接下來的 k 個數字就是孩子的編號。每一行的相鄰數字間以空白隔開。 $N \leq 1e5$ 。

輸出：第一行輸出根的編號，第二行輸出高度總和。

範例輸入：

```
9
1 6
3 5 3 8
0
2 1 7
1 9
0
1 2
0
0
```

範例結果：

```
4
11
```

沒有處理過樹狀圖的人可能會覺得這題複雜，但其實不難，這裡我們盡量不使用樹的術語來講。首先我們用個陣列把每個人的 `parent` 讀進來，第一個問題是如何找根。沒有 `parent` 的就是根，所以很簡單，只要看看哪個點沒有 `parent` 就行了。第二個問題要找高度，因為我們有每個點的 `parent`，一個直覺的做法就是：

從每一個點 i 出發，依據 `parent` 一直往上走直到無路可走（到達 `root`），並且用一個記步器 `cnt` 記錄走了幾步。每到一個點 j ，根據 `cnt` 就可以知道 i 是 j 的第幾代子孫。因為高度是要取所有子孫離他最遠的，所以到達 j 點時，就取 $h[j] = \max(h[j], cnt)$ ；在所有的點都走完之後，各點的高度就已經算好了。

夠簡單吧！以下是這個解法的範例程式。

```
// tree height slow method O(nL), L=tree height
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int main() {
    int parent[N]={0}; // parent of node i
    int h[N]={0}; // height of node i
```



```

int deg[N]; // num of children
int n, i, j, ch;
scanf("%d", &n);
for (i=1; i<=n; i++) {
    scanf("%d", &deg[i]); // num of children
    for (j=0; j<deg[i]; j++) {
        scanf("%d", &ch);
        parent[ch] = i; // set parent
    }
}
int root;
for (root=1; root<=n; root++)
    if (parent[root]==0) break; // no parent
printf("%d\n", root);
long long step=0;
for (i=1; i<=n; i++) { // for each
    int cnt=0; // num of step from i
    // go up until root
    for (j=i; j!=0; j=parent[j]) {
        h[j]=max(h[j],cnt);
        cnt++; step++;
    }
}
long long total=0;
for (i=1; i<=n; i++)
    total += h[i];
printf("%lld\n", total);
fprintf(stderr, " %lld steps, ans=%lld\n", step, total);
return 0;
}

```

這程式雖然容易理解，答案跑出來也是對的，但是有個問題，跑太慢，他的複雜度是多少呢？以一個極端的例子來看，如果這個家族 10 萬個成員全部都是單傳，最高祖先的高度是 99999，那麼這個程式的複雜度是多少呢？你或許會抗議哪裡有這麼多代的家族呀！唉呀，競賽程式就是這樣會給刁鑽的測資，worst case 就真的很 worst，況且題目根本沒說是人的家族，細菌 10 萬代就很有可能。好了，廢話少說，因為上面這個程式裡，每個點都要往上爬到根，所以在這個極端的例子裏，最下面的要跑 99999，倒數第二的要跑 99998，...，所以總共的複雜度約是 1 加到 99999，複雜度是 $O(n^2)$ ，一秒是跑不完的。

找到問題就設法改善吧。程式跑太慢通常是做了太多重複的事，在前面這個 10 萬代單傳的例子裏，這個方法之所以慢，就是走了太多一樣的路：最下一層的往上爬了 N 層，他的 parent (倒數第二層) 爬了 $N-1$ 層，兩位爬的路幾乎重複，假設倒數第二層確定知道自己的高度是 1 的話，那這兩位重疊的路只要爬一次不需要爬兩次。也就是說，對於 i 點，如果 $h[i]$ 已經確定，那麼 i 的子孫不必每位都爬 i 到根那段路，只要從 i 的高度往上計算爬一次就好了。那麼，要如何確定 i 的高度呢？根據定義，只要 i 的子孫都已經爬到 i 點， i 的高度就確知了。因為子孫關係有遞移性，我們得到一個結論：

如果我們可以調整計算的順序，使得「每個點都在他的所有孩子之後計算，那麼，每個點只需要向他的 parent 回報自己的高度就可以了」。

這樣一個順序我們稱為由下而上的順序 (bottom-up sequence)。假設 seq[] 存放著一個 bottom-up 順序，那麼，計算高度只需要以下簡單的程式碼：

```
for (i=0; i<n; i++){
    int v = seq[i];
    h[parent[v]] = max(h[parent[v]], h[v]+1);
}
```

所以重點在於如何找出一個 bottom-up 順序，這就是 queue 出場的時候了，請看以下的算法。我們說一個點可以離開是指他的孩子都已經離開，顯然這樣的離開順序是一個 bottom-up sequence：

對所有 i ，以 $deg[i]$ 紀錄 i 目前還在的孩子數量，以一個 queue Q 放置所有可以離開但尚未離開的點，一開始把所有 $deg[]$ 為 0 的點放入 Q 中。每次從 Q 中拿出一個點 v ，讓 v 離開，更新他 parent 的 $deg[]$ ，如果他 parent 的 $deg[]$ 降為 0，就將他 parent 放入 Q 。

結合這樣的順序，我們可以將上面的程式改成下面的範例程式，這裡我們使用 STL 的 queue，另外我們也不需要特別去找 root，因為 bottom-up 順序的最後一個必然是 root。

```
// tree height bottom-up dp, using STL queue
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int main() {
    int parent[N]={0}; // parent of node i
    int h[N]={0}; // height of node i
    int deg[N]; // num of children
    queue<int> Q;
    int n, i, j, ch;
    scanf("%d", &n);
    for (i=1; i<=n; i++) {
        scanf("%d", &deg[i]);
        if (deg[i] == 0) Q.push(i);
        for (j=0; j<deg[i]; j++) {
            scanf("%d", &ch);
            parent[ch] = i;
        }
    }
    int root, total=0; // total height
    // a bottom-up traversal
    while (1) { // queue is not empty
        int v = Q.front(); // pop Q to v
```

```

    Q.pop();
    total += h[v]; // add to total
    int p = parent[v];
    if (p == 0) { // root
        root = v;
        break;
    }
    h[p] = max(h[p], h[v]+1);
    if (--deg[p] == 0) // push parent to queue
        Q.push(p);
}
// root must be the last one in the queue
printf("%d\n%d\n", root, total);
return 0;
}

```

計算一下時間複雜度，輸入把 $\text{deg}[]$ 為 0 的放入 Q 中，花了 $O(n)$ ；while 迴圈裡面只有 $O(1)$ 的運算，問題是 while 迴圈做了幾次，因為每次有一點離開點 Q ，而一個點不可能進入 Q 兩次，所以整個時間複雜度是 $O(n)$ ，比改善前好太多了。這個題目在 APCS 五級分的程度已經算難的，如果第一次看 tree 就看得懂，恭喜你，如果不能太了解，沒關係，後面還有專門的章節來講樹狀圖。

接下來要看個比較簡單的例子，這是堆疊運用的經典題目。

P-3-2. 括弧配對

在本題中我們假設有三種括弧： $\{\}$ ， $()$ ， $[\]$ 。輸入一個由括弧組成的字串，請判斷是否是**平衡的**，括弧可以巢狀但是不可以交叉，例如「 $() [\] \{ \} \}$ 」、「 $[(()) \{ \}]$ 」、「 $[() \{ \} \{ \}] \{ \} ()$ 」都是平衡的，但「 $([]$ 」不是平衡的，此外「 $() []$ 」也不是，因為[沒有配對。

Time limit: 1 秒

輸入格式：輸入包括若干行，最多 20 行，每行是一個表示式，由六個括弧字元組成的字串，沒有其他字元，字串長度不超過 150。

輸出：依序輸出每行是否是平衡的括弧，是則輸出 yes，否則輸出 no。

範例輸入：

```

() [] { [] }
[ ( ( ) ) { } ]
[ ( ) { } { } ] { } ( )
( [ ]

```

```
() [
```

範例結果：

```
yes
yes
yes
no
no
```

括弧配對其實是計算表示式 (expression evaluation) 的其中一個環節，是資料結構課程中的一個重要課題，因為這是程式 compiler 裡面一定要做的工作。括弧的結構事實上也是遞迴定義的，所以可以用遞迴的方式做，但用堆疊更簡單。用堆疊檢查括弧是否平衡的算法也很直覺，基本上我們需要找到每一個右括弧對應的左括弧，算法如下：

啟用一個 stack S 。掃描輸入字串的每一個字元，

如果是左括弧的一種：將他壓入 S ；

否則 (是右括弧)，取出堆疊最上方的左括弧與其配對，如果不配，則錯誤。

如果字串結束後堆疊中還有東西，也是錯誤。如果都沒錯誤則是平衡的。

本題有三種括弧，在實作時要避免很繁瑣的窮舉三種括弧，我們使用了一個小技巧，請看以下的範例程式。將 6 種括弧字元對應到整數 0~5，先排左括弧再排右括弧，同類的一對的讓他們剛好差 3，這樣可以簡單的檢查是否配對。為了做這個對應，我們先宣告了一個字串 `ch[7]="([{}])"`；對每一個輸入字元 `in[i]`，用一個字串函數 `strchr()` 來找出 `in[i]` 在 `ch[]` 中的位置，這個函數如果不熟悉，也可以自己寫一個簡單的迴圈 (如同程式中的註解)。

```
// p3-2 parenthesis matching, no STL
#include <bits/stdc++.h>

int main() {
    char in[210], ch[7]="([{}])";
    int S[210], top; // stack
    while (scanf("%s",in)!=EOF) {
        top=-1; // clear stack
        int len=strlen(in);
        assert(len<=150);
        bool error=false;
        for (int i=0;i<len;i++) {
            int k=strchr(ch,in[i])-ch;
            //or using: for (k=0; k<6 && ch[k]!=in[i]; k++);
            assert(k<6); // no invalid char
            if (k<3) // left
                S[++top]=k; // push
            else {
```

```

        if (top<0 || k!=S[top]+3) { //mismatch
            error=true;
            break;
        }
        top--; // pop
    }
}
if (top>=0) error=true;
printf((error)?"no\n":"yes\n");
}
return 0;
}

```

上面的程式是自製堆疊，如果使用 STL，則如下面的範例，程式雷同，就不多做解釋了。但是有件事情要提醒一下，因為這題有多筆測資要處理，你可以注意到我們把堆疊宣告在一個副程式中，每一筆測資呼叫一次副程式，這樣做的好處是每一筆測資進來的時候不必把 stack 清空，如果是寫在主程式中以迴圈處理每一筆測資(像上面那支範例的寫法)，那要留意你的寫法是否 stack 可能留有前一筆測資留下來的東西，而要将 stack 清空必須把他 pop() 到空為止。

```

// p3-2 parenthesis matching, using STL
#include <bits/stdc++.h>
using namespace std;

int sol() {
    stack<int> S;
    char in[210], ch[7]="([{}])";
    if (scanf("%s",in)==EOF) return -1;
    // fprintf(stderr,"input length=%d\n",strlen(in));
    bool error=false;
    for (int i=0,len=strlen(in);i<len;i++) {
        int sym=strchr(ch,in[i])-ch;
        if (sym>=6) {
            fprintf(stderr,"i=%d, %d, %c",i, sym, in[i]);
            return -1;
        }
        assert(sym>=0 && sym<6);
        if (sym<3) { // left
            S.push(sym);
        }
        else {
            if (S.empty() || sym!=S.top()+3) { //mismatch
                error=true;
                break;
            }
            //match
            S.pop();
        }
    }
}

```

```

    }
}
if (!S.empty()) error=true;
printf((error?"no\n":"yes\n");
return 0;
}
int main() {
while (sol()==0) ;
return 0;
}

```

下一個習題，我們是一個簡化版的計算表示式，輸入一個數學運算式，請計算出他的值，計算式中的運算只有加減乘除，其中除法是整數除法，運算的數字都只有一位正整數。

Q-3-3. 加減乘除

輸入一個只有加減乘除的計算式，請計算並輸出其結果。其中的數字皆為一位正整數，計算結果的絕對值不會超過 10 億。

Time limit: 1 秒

輸入格式：輸入一行是一個計算式，長度不超過 100。假設是一個正確的計算式。

輸出：輸出計算結果。

範例輸入：

5+2*3-6-7*2

範例結果：

-9

提示：因為加減乘除都是左結合的二元運算，左結合是指連續兩個運算時先算左邊，例如 $2-3-5=(2-3)-5$ 。現在需要考慮的只有先乘除後加減。

一開始先將第一個數字放起來，接著從前往後掃描計算式，每次考慮兩個字元，第一個一定是個運算符號 (op) 第二個一定是個數字 (num)。如果 op 是個乘法或除法，我們可以立刻將前面的運算結果與 num 做運算，因為沒有別的運算優先權高與乘除。如果是加法或減法，我們必須考慮兩種情形：(1) 如果之前有被保留的運算要先執行，目前的運算要保留，因為後面可能是乘除法；(2) 如果之前沒有被保留的運算，這個運算要被保留。在考慮一連串的運算過程中，最多只有一個運算會被保留，而且這個被保留的運算

必定是加法或減法。最後要提醒一點：全部的運算都考慮完畢後要檢查是否有被保留的運算。

完整的 expression evaluation 可以在資料結構的課本上或是網路上查到，要考慮的因素很多，所以做法比較複雜。

下面一題是堆疊運用的經典題。

P-3-4. 最接近的高人 (APCS201902, subtask)

對於一個人來說，只要比他高的人就是高人。有 N 個人排成一列，對於每一個人，要找出排在他前方離他最近的高人，排在之後的高人不算，假設任兩個相鄰的人的距離都是 1，本題要計算每個人和他前方最近高人的距離之總和。如果這個人前方沒有高人(前方沒人比他高)，距離以他的位置計算(等價於假設最前方有個無窮高的高人)。

Time limit: 0.5 秒

輸入格式：第一行有一個正整數 N ，第二行有 N 個正整數依序代表每個人的身高，相鄰數字間以空白隔開。 $N \leq 2e5$ ，身高不超過 $1e7$ 。

輸出：輸出每個人與前方最近高人的距離總和。

範例輸入：

```
8
8 6 3 3 1 5 8 1
```

範例結果：

```
18
```

為了方便起見，我們可以假設在最前方位置 0 的地方有個無限大的數字，這樣可以簡化減少邊界的檢查。最天真無邪又直接的方法就是：對每一個 i ，從 $i-1$ 開始往前一一尋找，直到碰到比他大的數字或者前方已走投無路。但是這個方法太慢，算一下複雜度，在最壞的情形下，每個人都要看過前方的所有人，所以複雜度是 $O(n^2)$ 。這一題 n 是 $2e5$ ，一定是 $O(n)$ 或 $O(n \log(n))$ 的方法。

要改善複雜度，要想想是否有沒有用的計算或資料。假設身高資料放在陣列 $a[]$ ，如果 $a[i-1] \leq a[i]$ ，那麼 $a[i-1]$ 不可能是 i 之後的人的高人，因為由後往前找的時候會先碰到 $a[i]$ ，如果 $a[i]$ 不夠高， $a[i-1]$ 也一定不夠高。同樣的道理，當我們計算到 i 的時候，任何 $j < i$ 且 $a[j] \leq a[i]$ 的 $a[j]$ 都是沒有用的。如果我們丟掉那些沒有用的，會剩下甚麼呢？一個遞減的序列，只要維護好這個遞減序列，就可以提高計算效率。單調序列要用二分搜嗎？其實連二分搜都不需要，想想看，當處理 $a[i]$ 時，我

們要先把單調序列後方不大於 $a[i]$ 的成員全部去除，然後最後一個就是 $a[i]$ 要找的高人，因為既然要做去除的動作，就一個一個比較就好了。那麼，要如何維護這個單調序列呢？第一個想法可能是將那些沒用的刪除，那可不行，陣列禁不起的折騰就是插入與刪除，因為必須搬動其後的所有資料。想想我們要做的動作：每次會從序列後方刪除一些東西，在加入一個東西。答案呼之欲出了，因為只從後方加入與刪除，堆疊是最適合的。請看以下的範例程式，這個程式非常簡單。

身高資料放在 $a[i]$ ，讀入之後就不再變動。準備一個堆疊 S 放置那個單調序列在 $a[]$ 中的索引位置。對每一個 $a[i]$ ，從堆疊中 pop 掉所有不大於它的東西，然後堆疊的最上方就是 i 的高人，接著把 $a[i]$ push 進堆疊。

```
// p_3_4a, stack for index
#include <bits/stdc++.h>
using namespace std;
#define N 300010
#define oo 10000001
int a[N];
stack<int> S; // for index
int main() {
    int i,n;
    long long total=0; // total distance
    scanf("%d",&n);
    S.push(0);
    a[0]=oo;
    for (i=1; i<=n; i++) {
        scanf("%d",&a[i]);
    }
    for (i=1; i<=n; i++) {
        while (a[S.top()] <= a[i])
            S.pop();
        total += i - S.top();
        S.push(i);
    }
    printf("%lld\n",total);
    return 0;
}
```

時間複雜度？看到雙迴圈以為是 $O(n^2)$ 嗎？跑跑看就知道這麼快一定不是 $O(n^2)$ ，難不成測資出太弱了！台語說：事情不像憨人想的那麼簡單。這個程式雖然有個雙迴圈，內層 while 迴圈也的確有可能某幾次跑到 $O(n)$ ，但是整體的複雜度其實只有 $O(n)$ ，原因是，雖然單一次 while 迴圈可能跑很多個 pop ，但是全體算起來不會 pop 超過 n 次，因為每個傢伙只會進入堆疊一次，進去一次不可能出來兩次，所以整體的複雜度是 $O(n)$ 。這種算總帳的方式來計算複雜度稱為均攤分析 (amortized analysis)，這個題目是均攤分析的經典教學題目。

上面的範例程式把資料放在 `a[]`，用 `stack` 存 `index`，事實上我們也可以只存那個單調序列，這樣在 `stack` 中要存身高與位置，以下是這樣的寫法，這只是給有興趣的人參考，上面那個寫法就夠好了。

```
// p_3_4, stack
#include <bits/stdc++.h>
using namespace std;
#define N 300010
#define oo 10000001
stack<pair<int,int>> S; // (b[i],i)
int main() {
    int i,bi,n;
    long long total=0; // total distance
    scanf("%d",&n);
    S.push({oo<<1,0});
    for (i=1; i<=n; i++) {
        scanf("%d",&bi);
        // <=bi is useless
        while (S.top().first <= bi)
            S.pop();
        total += i - S.top().second;
        S.push({bi,i});
    }
    printf("%lld\n",total);
    return 0;
}
```

前面的做法是由前往後掃，做到 `i` 的時候往前找 `i` 的對象(高人)。射鵰英雄傳的歐陽鋒都可以逆練九陰真經了，這個題目是不是可以倒著來解：由後往前，做到 `i` 時看看 `i` 是誰的對象(高人)。確實可以，基本想法是這樣：

由後往前掃，使用一個資料結構 `S` 存放目前尚未找到高人的人。掃到 `i` 時，我們檢查 `i` 是那些人的高人，也就是 `S` 中那些人的身高小於 `i` 的身高，然後將這些人從 `S` 中移除，最後把 `i` 放入 `S` 中，結束這一回合。

為了讓有興趣的人練習資料結構，我們介紹使用 `map` 來做的方法，因為身高可能相同，我們要用 `multimap`。(APCS 的難度應該不會考到 `map`，但有興趣的人或學習競賽的人可以練習。)

```
// p_3_4c, using multimap
#include <bits/stdc++.h>
using namespace std;
#define N 300010
#define oo 10000001
int a[N];
```

```

int main() {
    int i,n;
    long long total=0; // total distance
    scanf("%d",&n);
    a[0]=0;
    for (i=1; i<=n; i++) {
        scanf("%d",&a[i]);
    }
    multimap<int,int> M; //(a[i],i) of unsolved
    for (i=n; i>=0; i--) {
        auto it=M.begin();
        // find the whose solution is i
        while (it!=M.end()&& it->first<a[i]) {
            total += it->second - i; // add distance
            it=M.erase(it); // erase and point to next
        }
        M.insert({a[i],i});
    }
    printf("%lld\n",total);
    return 0;
}

```

下面這一題，是前一題的一種變化，我們當作習題。

Q-3-5. 帶著板凳排雞排的高人 (APCS201902)

身高高不一定比較厲害，個子矮的如果帶板凳可能會變高。有 n 個人排隊買雞排，由前往後從 1 到 n 編號，編號 i 的身高為 $h(i)$ 而他帶的板凳高度為 $p(i)$ 。若 j 在 i 之前且 j 的身高大於 i 站在自己板凳上的高度，也就是說 $h(j) > h(i) + p(i)$ ，則 j 是 i 的「無法超越的位置」，若 j 是 i 最後的無法超越位置，則兩人之間的人數 $(i-j-1)$ 稱為 i 的「最大可挑戰人數」 $S(i)$ ；如果 i 沒有無法超越位置，則最大可挑戰人數 $S(i) = i-1$ ，也就是排在他之前全部的人數。

舉例來說，假設 $n=5$ ，身高 h 依序為 $(5, 4, 1, 1, 3)$ 而板凳高度 p 依序是 $(0, 0, 4, 0, 1)$ 。計算可得 $h(i) + p(i) = (5, 4, 5, 1, 4)$ ，編號 1 的人前面沒有人，所以 1 的最大可挑戰人數 $S(1) = 0$ 。對編號 2 的人而言， $h(2) + p(2) = 4$ ，而往前看到第一個大於 4 的是 $h(1) = 5$ ，所以 $S(2) = 2-1-1 = 0$ 。而 $h(3) + p(3) = 5$ ，前面沒有人的身高大於 5，所以 $S(3) = 2$ 。而 $h(4) + p(4) = 1$ ， $h(2) = 4 > 1$ ，所以位置 2 是 4 的無法超越位置， $S(4) = 4-2-1 = 1$ 。同理可求出 $S(5) = 3$ ，他的不可超越位置是 1 的身高 5。

輸入 $h()$ 與 $p()$ ，請計算所有人 $S(i)$ 的總和。

Time limit: 1 秒

輸入格式：第一行為 n ， $n \leq 2e5$ ；第二行有 n 個正整數，依序是 $h(1) \sim h(n)$ ；第三行有 n 個非負整數，依序代表是 $p(1) \sim p(n)$ ；數值都不超過 $1e7$ ，同一行數字之間都是以空白間隔。

輸出：輸出 $S(i)$ 的總和。答案可能超過 2^{32} ，但不會超過 2^{60} 。

範例輸入：

```
5
5 4 1 1 3
0 0 4 0 1
```

範例結果：

```
6
```

Q-3-5 提示：本題與 P-3-4 的主要差異是多了板凳高度，另外有個小差異是計算的值略有不同。例題的題解中最重要重點是我們只要維護一個單調遞減序列，既然是單調序列，就可以用二分搜找到第 i 個人站在板凳上的高度。此外，例題中我們介紹了另外一種使用 `multimap` 的解法，一樣可以稍作修改後解此題。

雖然保護樹木很重要，尤其學資料結構與演算法的人，對樹總是抱持著尊敬的心，但是有些樹種的目的就是要砍下來當木材的，請環保人士不要對下一題的名字抗議。來看下一個例題，砍樹。

P-3-6. 砍樹 (APCS202001)

N 棵樹種在一排，現階段砍樹必須符合以下的條件：「讓它向左或向右倒下，倒下時不會超過林場的左右範圍之外，也不會壓到其它尚未砍除的樹木。」。你的工作就是計算能砍除的樹木。若 $c[i]$ 代表第 i 棵樹的位置座標， $h[i]$ 代表高度。向左倒下壓到的範圍為 $[c[i]-h[i], c[i]]$ ，而向右倒下壓到的範圍為 $[c[i], c[i]+h[i]]$ 。如果倒下的範圍內有其它尚未砍除的樹就稱為壓到，剛好在端點不算壓到。

我們可以不斷找到滿足砍除條件的樹木，將它砍倒後移除，然後再去找下一棵可以砍除的樹木，直到沒有樹木可以砍為止。無論砍樹的順序為何，最後能砍除的樹木是相同的。

Time limit: 1 秒

輸入格式：第一行為正整數 N 以及一個正整數 L ，代表樹的數量與右邊界的座標；第二行有 N 個正整數代表這 N 棵樹的座標，座標是從小到大排序的；第三行有 N 個正整數代表樹的高度。同一行數字之間以空白間隔， $N \leq 1e5$ ， L 與樹高都不超過 $1e9$ 。

輸出：第一行輸出能被砍除之樹木數量，第二行輸出能被砍除之樹木中最高的高度。

範例輸入：

```
6 140
10 30 50 70 100 125
30 15 55 10 55 25
```

範例結果：

```
4
30
```

題目中有句話：無論砍樹的順序為何，最後能砍除的樹木是相同的。所以不斷找到滿足砍除條件的樹木，將它砍倒後移除，直到沒有樹木可以砍為止，就可以得到正確的答案，問題只是效率好不好。我們先看看天真的作法。

寫程式最重要的是知道每次要做甚麼事情，資料中存放的是甚麼，有何特性。如果我們用陣列來存這些樹，那麼立刻碰到一個問題：當樹被砍掉了之後怎麼辦？我們當然必須維護好陣列中是沒有被砍掉的樹，否則判斷就會出問題。第一個想法就是想要把砍倒的樹的資料從陣列中移除，那麼陣列中如何移除資料呢？只能把後面的往前搬，陣列最經不起插入與刪除的折騰，好吧，反正電腦累，程式碼倒是簡單。

下面的範例程式是這樣的想法寫出來的。先寫一個副程式從頭到尾檢查找到第一根可以砍的樹。這裡我們加了一個小技巧，在兩個邊界種上無限高的樹，這是常用的技巧，目的是省卻檢查邊界條件的麻煩。主程式中每次呼叫副程式，找到可以砍的樹之後就藉著搬移後面的資料來移除這筆資料。

```
// O(n^2), remove cut tree from array
#include <stdio.h>
#define N 100010
#define oo 1000000000
// return a removable tree; return -1 if none
int removable(int c[], int h[], int k) {
    for (int i=1; i<k-1; i++)
        if ((c[i]-h[i]>=c[i-1]) || (c[i]+h[i]<=c[i+1]))
            return i;
    return -1;
}

int main() {
    int n, l, c[N], h[N]; // center and height
    int total=0, high=0; // total removed and max height
    scanf("%d%d", &n, &l);
    // set left and right boundaries
```

```

c[0]=0, h[0]=oo;
c[n+1]=1, h[n+1]=oo;
for (int i=1;i<=n;i++)
    scanf("%d",&c[i]);
for (int i=1;i<=n;i++)
    scanf("%d",&h[i]);
n += 2; // two boundary
while (1) {
    int cut = removable(c, h, n); // the previous alive
    if (cut < 0) break;
    total++;
    if (h[cut] > high) high=h[cut];
    // remove cut from array
    for (int i=cut; i<n-1; i++)
        h[i] = h[i+1]; // overwrite by next
    for (int i=cut; i<n-1; i++)
        c[i] = c[i+1];
    n--;
}
printf("%d\n%d\n", total, high);
return 0;
}

```

複雜度如何？ $O(n^2)$ ，因為砍樹的順序剛好是從前往後，那麼總共搬移的資料量就會是從 1 加到 $n-1$ 。處理陣列資料被移除的方式除了真的把他移除之外，還有一種常用的方法是將他標記，這一題我們可以把砍除的樹標記為高度 0，這麼一來刪除就只花 $O(1)$ ，但是有一好沒兩好，我們在檢查左右存活的樹的時候，就必須往左往右尋找，因為旁邊的樹可能已經被砍除。以下是這樣寫出來的程式，很不幸，複雜度還是 $O(n^2)$ 。

```

// P_3_6, n^2 method, mark removed tree
#include <stdio.h>
#define N 100010
#define oo 10000000000

int main() {
    int n, l, c[N], h[N]; // center and height;
    int total=0, high=0; // total removed and max height
    scanf("%d%d",&n,&l);
    // set left and right boundaries
    c[0]=0, c[n+1]=1;
    h[0]=h[n+1]=oo;
    for (int i=1; i<=n; i++)
        scanf("%d", &c[i]);
    for (int i=1; i<=n; i++)
        scanf("%d", &h[i]);
    // repeat until no tree can be removed
    while (1) {
        int i;
        for (i=1; i<=n; i++) { // check each tree

```

```

        if (h[i]==0) continue; // already removed
        int prev=i-1, next=i+1;
        while (h[prev]==0) prev--; // previous remaining tree
        while (h[next]==0) next++; // next remaining tree
        if (c[i]-h[i]>=c[prev] || c[i]+h[i]<=c[next])
            break; // removable
    }
    if (i>n) break; // no tree can be removed
    total++;
    if (h[i] > high) high=h[i];
    h[i]=0;
}
printf("%d\n%d\n", total, high);
return 0;
}

```

這一題有人以為應該從矮的樹開始砍，但這顯然沒有甚麼幫助，因為高的樹可能比矮的樹先砍除。其實我們只要想想：目前不能被砍除的樹，什麼狀況會變成可以砍除呢？一定與他相鄰的(還存活的)樹被砍掉，才有可能空出多餘的空間。加油！解法就快要浮現了。

假設我們以一個佇列 Q 存放那些砍除的樹，一開始先檢查一次那些樹可以被砍除，把他們放入 Q ，然後我們一次將 Q 中的樹拿出來，檢查他左右相鄰的樹是否可以變成可以砍除，如果可以，就砍除並放入 Q ，這樣做直到 Q 中沒有樹為止。因為每一棵被砍除的樹都有考慮他的左右，所以不會漏砍。這是個正確的算法，但是這個過程中牽涉一個問題，就是必須很快的決定某棵樹的左右鄰居。趁這個機會我們簡單介紹一下串列鏈結 (linked list)。

串列鏈結 (Linked list)

串列是一個很重要的資料結構，顧名思義，他是將資料以鏈結的方式串成一列。他最重要的運用場合是當資料無法依序儲存在連續空間的時候，否則，用陣列會比串列更簡單有效率。但是陣列的缺點就是必須連續擺放，在某些應用時，這個限制變得不可能，例如檔案存在硬碟中，因為檔案會新增與刪除，所以將一個檔案放在連續的磁碟位置變得不可能。鏈結串列的概念是不必把資料連續擺放，只要對每一筆資料記載他的下一筆所在位置，有了這些資料，我們就可以一筆接一筆的把全部的資料追蹤一遍了。鏈結串列有單向與雙向，看應用而定，雙向的串列我們會儲存每一個資料的前一筆與下一筆鏈結，單向的只存下一筆。

通常鏈結串列在運用時多半搭配記憶體的配置 (memory allocation) 與指標 (pointer) 變數，但是在解題程式中，我們通常會避免這兩個東西，原因之一是指標很難除錯，另外一個原因是解題的程式通常都有辦法避掉指標，另外用偷懶的方式來做。

這份教材是針對考試與競賽的解題，所以我們就不對這個部份多做說明了，有興趣的讀者可以很容易在網路上或者資料結構的教科書中找到教學文件。在解題的場合，我們可以在陣列中構造所需的鏈結串列，用陣列的索引 (index) 來取代記憶體位置 (指標)。我們接著就看看串列如何用在 P-3-6。

在下面的範例程式中，我們定義了一個 struct，裡面存放一棵樹的資料，包括：位置、高度、前一棵的位置、下一棵的位置、以及是否還存活。這裡需要使用雙向鏈結，其中前一棵與下一棵的位置都是指在陣列中的索引值。如前所述，我們以一個佇列 Q 存放那些砍除的樹。函數 removable() 用來檢查第 i 棵樹是否可以被砍掉，如果可以，我們會將他從串列中移除，並且放入 Q 中。把一筆資料從串列中移除不需要像陣列一樣搬一大堆資料，只需要將左邊的 next 改成 i 的 next，把右邊的 pre 改成 i 的 pre，這樣在串列中沿著鏈結追蹤時就再也找不到 i 了，也就相當於從串列中移除了。

主程式一開始先做一次地毯式搜索，對每一個檢查每一個 i 呼叫一次 removable(i)，接著進入一個 while 迴圈，迴圈會一直做到 Q 必成空的為止。迴圈中每次從 Q 中拿出一棵樹，對他的左右會叫一次 removable()。這個程式的複雜度是多少？很顯然 removable() 是 $O(1)$ ，因為其中沒有迴圈也沒有遞迴。因為每棵樹最多只會被放入 Q 中一次，所以整體的複雜度是 $O(n)$ 。

```
// P_3_6b, topological sort and linked list
#include <queue>
#include <cstdio>
using namespace std;
#define N 100010
#define oo 1000000001
struct {
    int c, h; // center and height
    int pre, next; // linked list
    int alive;
} tree[N];
queue<int> Q; // queue for removed tree
// check if i is removable, if yes, remove it
void removable(int i) {
    if (!tree[i].alive) return;
    int s=tree[i].pre, t=tree[i].next;
    if (tree[i].c - tree[i].h >= tree[s].c \
        || tree[i].c+tree[i].h<=tree[t].c) {
        tree[i].alive = 0;
        Q.push(i); // insert into queue
        tree[s].next = t; //remove from linked list
        tree[t].pre =s;
    }
    return;
}

int main() {
```

```

int n,l,i;
int total=0, high=0;
scanf("%d%d",&n,&l);
// initial data
for (i=1; i<=n; i++)
    scanf("%d", &tree[i].c);
for (i=1; i<=n; i++)
    scanf("%d", &tree[i].h);
for (i=1; i<=n; i++) {
    tree[i].pre = i-1;
    tree[i].next = i+1;
    tree[i].alive = 1;
}
// two boundaries
tree[0].c = 0, tree[0].h = oo;
tree[n+1].c = l, tree[n+1].h = oo;
// initial check
for (i=1; i<=n; i++)
    removable(i);
while (!Q.empty()) {
    int v=Q.front();
    Q.pop();
    total++;
    high = max(high, tree[v].h);
    // check its previous and next
    removable(tree[v].pre);
    removable(tree[v].next);
}
printf("%d\n%d\n", total,high);
return 0;
}

```

上面這個程式的方法，其實類似於圖論演算法中計算 dag 的 topological sort 所用的方式，我們在後續介紹圖的章節中會再介紹。事實上，這一題還有更簡單的作法。

回到剛才講到的關鍵點：一棵目前不能被砍除的樹，只有在相鄰的樹被砍掉後才有可能變成可以砍除。假設我們從前往後掃描所有的樹，可以砍的就砍了，不能砍的暫且保留起來，那麼被保留的樹和時會變成可以被砍呢？因為他的左方(前方)不會變動(我們從左往右做)，所以一定是右邊的樹被砍了之後才有可能。此外，那些被保留的樹，一定只有最後一棵需要考慮！因為其他的被保留樹的右邊都沒有動。好了，最後的問題是要怎麼存那些被保留的樹呢？我們一樣可以用鏈結串列，但是沒有必要，因為我們只要知道每一棵樹還存活的的前一棵樹就可以，而且(重點是)我們只需要檢查最後一棵保留樹，也只需要從後面往前刪除。答案出來了，用堆疊就夠了。

以下是用堆疊寫的範例程式。需要留意的是，當一棵樹被砍除時，我們要用一個 while 迴圈對堆疊出口的樹做檢查，因為可能砍到一棵可能引發連鎖效應一路往前砍掉很多棵。複雜度 $O(n)$ ，因為每個樹最多只進入堆疊一次。請注意我們在堆疊中只放樹的 index，當然也可以寫成在堆疊中存放樹的位置與高度。


```

// P_3_6, O(n) using stack
#include <bits/stdc++.h>
using namespace std;
#define N 100010
#define oo 1000000000

int main() {
    int n,l,i, c[N],h[N];
    int total=0, high=0; // num of removable and max height
    stack<int> S; // the index of remaining trees so far
    scanf("%d%d",&n,&l);
    // oo trees at two boundaries
    c[0]=0;
    h[0]=oo;
    c[n+1]=l;
    h[n+1]=oo;
    for (i=1;i<=n;i++)
        scanf("%d", &c[i]);
    for (i=1;i<=n;i++)
        scanf("%d", &h[i]);
    S.push(0);
    for (i=1;i<=n;i++) { // scan from left to right
        // if i is removable
        if (c[i]-h[i]>=c[S.top()] || c[i]+h[i]<=c[i+1]) {
            total++;
            high = max(high, h[i]);
            // backward check remaining tree in stack
            while (c[S.top()+h[S.top()]<=c[i+1]) {
                total++; high = max(high, h[S.top()]);
                S.pop();
            }
        } else { // i is not removable
            S.push(i);
        }
    }
    printf("%d\n%d\n", total,high);
    return 0;
}

```

滑動視窗 (Sliding window)

接下來要介紹滑動視窗的技巧，這個名字可能不是很統一，基本上是以兩個指標維護一段區間，然後逐步將這區間從前往後（從左往右）移動。維護兩個指標這一類的方法也有人稱為雙指標法，但也包含兩個指標從兩端往中間移動，就像我們曾在介紹過，在排好序的序列中尋找兩數相加之和的問題時，可以用兩個指標逐步移動來做，會比連續的二分搜還好。也有人稱為爬行法，維護一個逐步右移的區段很像是毛毛蟲的爬行，前方先往前，後方再跟上，不斷的重複這個亦步亦趨的方式。反正名稱不重要，我們來看看一些例子。

下一個例題與 P-2-11 幾乎相同，但此處限制輸入的數字為正整數，原題正負不限，所以這一題比較簡單。原題必須以 set 來解超過 APCS 考試範圍，這一題則是滑動視窗的基本題型。

P-3-7. 正整數序列之最接近的區間和

輸入一個正整數序列 ($A[1], A[2], \dots, A[n]$)，另外給了一個非負整數 K ，請計算哪些連續區段的和最接近 K 而不超過 K ，以及這樣的區間有幾個。 n 不超過 20 萬，輸入數字與 K 皆不超過 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：第一行輸出最接近 K 但不超過 K 的和，第二行輸出這樣的區間有幾個。

範例輸入：

```
5 10
4 3 1 7 4
```

範例輸出：

```
8
2
```

說明：(4, 3, 1) 與 (1, 7) 兩個區間的和都是 8。

假設一開始我們先固定左端在 0 的位置，從 0 開始一直移動右端以尋找最好的右端，也就是區段的和不超過 K 的最大，因為數字都是正的，右端越往右移，區段和就越大，所以當移動到不能再移時 (否則就會超過 K)，我們就找到了「以 0 為左端的最好區間」。接著我們將這個區段 (視窗) 逐步右移，每次先將右端往右移動一格，這時總和可能會超過 K ，所以我們要移動左端直到滿足區段和不超過 K 為止。每次右端移動一格，就會找到以此為右端的最佳解，所以當視窗向右滑到底時，我們就找出了最佳解。範例程式如下，以可以用 P_2_11 的程式來解做為對照，不過要稍微修改一下。

```
// P3-7, sum of subarray, positive
#include <bits/stdc++.h>
using namespace std;
#define N 1000010
int a[N];

int main() {
    int n, k;
    long long total=0;
```

```

scanf("%d%d",&n,&k);
for (int i=0;i<n;i++) {
    scanf("%d", &a[i]);
    total+=a[i];
}
int left, right,ans=0, sum=0, num=0;
// sum is for range [j,i]
// move right one by one
for (left=0,right=0;right<n;right++) {
    sum+=a[right];
    // move left until sum<=k
    while (sum>k) {
        sum-=a[left];
        left++;
    }
    // check answer
    if (sum>ans) {
        ans=sum;
        num=1;
    }
    else if (sum==ans) num++;
}
printf("%d\n%d\n",ans,num);
fprintf(stderr,"total=%lld, k=%d; %d %d\n",total,k,ans,num);
return 0;
}

```

上面一題的視窗大小是會變動的，其實比較像毛毛蟲，不像窗戶。下面一題我們來個固定長度的窗戶。

P-3-8. 固定長度區間的最大區段差

對於序列的一個連續區段來說，區段差是指區段內的最大值減去區段內的最小值。有 N 個非負整數組成的序列 seq ，請計算在所有長度為 L 的連續區段中，最大的區段差為何。

Time limit: 1 秒

輸入格式：第一行是 N 與 L ，第二行是序列內容，相鄰數字間以空白隔開。 $L \leq N \leq 2e5$ ，數字不超過 $1e9$ 。

輸出：輸出所求的最大區間差。

範例輸入：

```

9 4
1 4 3 6 9 8 5 7 1

```

範例結果：

7

說明：(8,5,7,1)的長度是 4，區段差是 7。

連續的 L 個格子是一個窗戶，長度 N 的序列有 $N-L+1$ 個可能的窗戶位置，對每一個位置，把窗戶內的最大值減去最小值，可以得到這個區段的區段差，所有 $N-L+1$ 個區段差中要找到最大的。直接的天真做法當然不行，每個區段跑一遍找最大最小需要 $O(L)$ ， $N-L+1$ 個區段就是 $O(L(N-L+1))$ ， $L=N/2$ 時，就是 $O(N^2)$ 。

這個題目解法的主要想法是紀錄區段內的最大值與最小值，當區段往右移動時，更新最大與最小值。乍看之下，更新最大最小值很容易就可以做到，因為只移進來一個值，移出去一個值。可是仔細想想，如果移出去的是最大值，那最大值會變成多少呢？可能會變也可能不變，這麼想下去覺得問題沒有那麼簡單。

前一章學過的 `multiset` 是否有用呢？因為 `set` 可以簡單的找到最大與最小，移進移出也都可以操作，所以確實以 `multiset` 可以來幫忙，時間複雜度是 $O(N \log(N))$ 。不過這裡要介紹的是 $O(N)$ 的解法，用的資料結構是 `deque`，也就是雙向佇列。

我們要不斷的找出窗戶內的最大值與最小值，我們先看最大值的找法，最小值的做法完全類似。其實這一題跟例題「P-3-4 最接近的高人」有點像，有個相同的單調性觀察：「當窗戶的右端推到 i 時，在 i 之前所有不大於 `seq[i]` 的元素，對未來找最大值都是沒有用的。」因此，我們可以刪除那些沒用的資料，只要維護好窗戶內的一個遞減子序列，與 P-3-4 不同處在於，因為窗戶會往右移動，資料有可能從窗戶左端離開，因此我們不能用堆疊來做，要用 `deque` 來做，以下是範例程式。

找最小值的做法相同，只要改變大小關係就可以了，所以我們用兩個 `deque`：`max_q` 與 `min_q`。函數 `put_max(i)` 處理將 `seq[i]` 從窗戶右端放入 `max_q` 時的動作：先從 `max_q` 的後端移除所有不大於它的成員（它是個遞減序列），然後把自己推進 `max_q` 內。在主程式中，跑一個迴圈讓窗戶的右端 i 逐步往右推，每次執行 `put_max()` 與 `put_min()`，另外檢查兩個 `deque` 的前端元素是否已經離開窗戶左端。處理完畢後 `max_q.front()` 就是窗戶內的最大值，`min_q.front()` 就是最小值。留意因為比較短的區段差不會比較大，所以初始時我們窗戶右端從 0 開始跑，此時窗戶寬度小於 L 但並不影響答案。

```
// P3-8, O(N) using deque
#include <bits/stdc++.h>
using namespace std;
#define MAXN 200010
```

```

int seq[MAXN];
deque<int> max_q, min_q; // index of max and min queue
// put seq[i] from back of max_q
void put_max(int i) {
    // remove useless
    while (!max_q.empty() && seq[max_q.back()]<=seq[i])
        max_q.pop_back();
    max_q.push_back(i);
}
// put seq[i] from back of min_q
void put_min(int i) {
    // remove useless
    while (!min_q.empty() && seq[min_q.back()]>=seq[i])
        min_q.pop_back();
    min_q.push_back(i);
}

int main() {
    int n, L, i, j, max_diff=0;
    scanf("%d%d\n", &n, &L);
    for (i=0; i<n; i++) {
        scanf("%d", &seq[i]);
    }
    put_max(0);
    put_min(0);
    for (i=1; i<n; i++) {
        // put the ith element into max_queue
        if (max_q.front()<=i-L) // out-of-range
            max_q.pop_front(); // its index
        put_max(i);
        // put the ith element into min_queue
        if (min_q.front()<=i-L) // out of range
            min_q.pop_front();
        put_min(i);
        // the diff of this subarray
        int diff=seq[max_q.front()]-seq[min_q.front()];
        max_diff=max(max_diff, diff);
    }
    printf("%d\n", max_diff);
    return 0;
}

```

前面的題目都是跟序列中數字的大小有關係，接下來看有數字但沒有大小的題目，一共有四個類似的題目，我們示範兩題，另外兩題當做習題。

P-3-9. 最多色彩帶

有一條細長的彩帶，彩帶區分成 n 格，每一格的長度都是 1，每一格都有一個顏色，相鄰可能同色。給定長度限制 L ，請找出此彩帶長度 L 的連續區段最多可能有多少種顏色。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 和 L ，滿足 $2 \leq L \leq n \leq 2 \times 10^5$ ；第二行有 n 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號為小於 n 的非負整數。

輸出：長度 L 的彩帶區段最多有多少種顏色。

範例輸入：

```
10 5
4 6 1 4 0 6 8 0 5 6
```

範例結果：

```
5
```

說明：區間 $[3, 7]$ 有 5 色。

這是一個簡單的題目，把一個固定長度 L 的視窗，從左到右逐步滑動過去，找出視窗內的顏色總數就行了。所以重點只有一個：如何找出是窗內的顏色數。因為顏色是從 0 開始連續編號，我們只要準備一個表格 cnt ，以 $cnt[i]$ 紀錄顏色 i 的出現次數。每次進入一個顏色，就將該顏色計數器加一；移出一個顏色就將顏色計數器減一。那怎麼知道窗內的顏色數呢？不能每次都掃描表格，因為太浪費時間，我們可以只關注顏色數的變動。我們可以用一個變數記錄目前的顏色數，若 $cnt[i]++$ 之後變成 1，那就知道顏色數多了一種；相同的，如果 $cnt[i]$ 減 1 之後變成 0 就是少了一色，其他狀況的顏色數量沒變化。

```
// P_3_9, sliding window, max color of fixed length
#include<bits/stdc++.h>
using namespace std;
#define N 200010
int seq[N], cnt[N]={0}; // counter of color i

int main() {
    int n,L,i;
    scanf("%d%d",&n,&L);
    int n_color=0;
    for (i=0;i<n;i++)
        scanf("%d",&seq[i]);
    // initial window
    for (i=0;i<L;i++) {
        int color=seq[i];
        cnt[color]++;
        if (cnt[color]==1) n_color++;
    }
    int ans=n_color, left;
```

```

// sliding window, seq[i] in, seq[left] out
for (left=0,i=L; i<n; i++,left++) {
    int color=seq[i];
    cnt[color]++;
    if (cnt[color]==1) n_color++;
    color=seq[left];
    cnt[color]--;
    if (cnt[color]==0) n_color--;
    ans=max(ans,n_color);
}
printf("%d\n",ans);
return 0;
}

```

下面一題很類似，但多了一些要處理的問題。

P-3-10. 全彩彩帶 (需離散化或字典) (@@)

有一條細長的彩帶，彩帶區分成 n 格，每一格的長度都是 1，每一格都有一個顏色，相鄰可能同色。如果一段彩帶中的顏色包含整段彩帶的所有顏色，則稱為「全彩彩帶」。請計算出最短的全彩彩帶長度。

Time limit: 1 秒

輸入格式：第一行為整數 n ，第二行有 n 個以空白間隔的非負整數，依序代表彩帶從左到右每一格的顏色編號， $n \leq 2e5$ ，顏色編號不超過 $1e9$ 。

輸出：最短的全彩彩帶長度。

範例輸入：

```

10
6 4 1 6 0 4 5 0 7 4

```

範例結果：

```

7

```

說明：彩帶共有 $\{0, 1, 4, 5, 6, 7\}$ 五色區，區間 $[3, 9]$ 是最短的包含五色區段。

基本上我們打算維持住一個視窗，視窗裡面包含所有的色彩，每次當視窗往右堆一格的時候，檢查左端是否可以往右移以便縮減寬度，當視窗推到底之後，我們就已經檢查過所有可能的右端點，在過程中找到最窄的視窗寬度，就是答案。

何時左端可以縮減呢？答案很明顯，如果左端點顏色在是窗內出現超過一次，就可以丟棄，否則，不能右移。另外一個問題是顏色的編號範圍，在前一題 P-3-9 時，顏色的編號不大於 n ，所以我們可以很容易的開一個表格當每個顏色的計數器，但是在這一題，顏色的編號可能達到 10 億，想要開一個 10 億的表格是會碰上麻煩的。雖然編號可能高達 10 億，但是資料最多就只有 20 萬個，顏色當然最多也只有 20 萬種，因此，如果我們可以将顏色的編號先修改為 0 開始的連續編號，就可以用上一題的表格方式來處理計數器了。在第二章的 P-2-2 與 P-2-2C 中我們示範了如何用 sort 或 map 做離散化，在這裡就可以派上用場了。下面的範例程式是以 sort 以及自己寫二分搜來做離散化的寫法，副程式 `c_id()` 就是自己寫的二分搜來查詢 color 在 `dic[]` 中的位置。在 35~36 行我們把所有的 `seq[]` 都換成 `rank` 之後就完成了離散化的工作。接著進行滑動視窗來找答案。以兩個變數 `left` 與 `right` 來維持至目前的視窗範圍，`right` 每次加 1，`left` 則只要顏色還有重複就盡可能縮減。要注意的是，為了簡化程式碼，我們並不保證一開始的視窗是包含所有的色彩的，但是在第 49 行加了一個 `if` 來確保當全部的色彩都在時，才會記錄答案。如果一開始要先找到出於全彩視窗也可以，但可能要多寫幾行。這個範例的程式碼雖然有一點長（其實也不算太長），但是方法的正確性不難理解，重點在於左端的顏色如果在是窗內僅出現一次，那就絕對不能右移。

```

00 // P_3_10 shortest all-color range, only basic instruction
01 #include<bits/stdc++.h>
02 using namespace std;
03 #define N 1000010
04 int seq[N], cnt[N]={0}; // counter of color
05 int dic[N]; // dictionary, map color to 0~m-1
06 // binary search to find id of color
07 int c_id(int color, int nc) {
08     if (color<=dic[0]) return 0;
09     int p=0;
10     for (int jump=nc/2; jump>0; jump>>=1) {
11         while (p+jump<nc && dic[p+jump]<color)
12             p+=jump;
13     }
14     return p+1;
15 }
16
17 int main() {
18     int n;
19     scanf("%d",&n);
20     for (int i=0;i<n;i++)
21         scanf("%d",&seq[i]);
22     // discretization
23     for (int i=0;i<n;i++)
24         dic[i]=seq[i];
25     sort(dic, dic+n);
26     // remove duplicate color
27     int n_color=1; // num of color
28     for (int i=1;i<n;i++) { // not checking dic[0]

```



```

29     if (dic[i]!=dic[i-1]) {
30         dic[n_color]=dic[i];
31         n_color++;
32     }
33 }
34 // replace color with its rank
35 for (int i=0;i<n;i++)
36     seq[i]=c_id(seq[i],n_color);
37 // end of discretization, start sliding window
38 // w_color = num of color in window [left, right-1]
39 int left=0, right=0, w_color=0, shortest=n;
40 while (right<n) {
41     cnt[seq[right]]++;
42     if (cnt[seq[right]]==1) w_color++;
43     right++;
44     // shrink left until left color appear only once
45     while (cnt[seq[left]]>1) {
46         cnt[seq[left]]--;
47         left++;
48     }
49     if (w_color==n_color)
50         shortest=min(shortest, right-left);
51 }
52 printf("%d\n",shortest);
53 return 0;
54 }
55

```

在第二章我們曾經揭露出以 STL 中的 map 來做離散化的寫法，底下是這一題用 map 的範例程式。這裡再強調一次，以 APCS 來說，應該不至於會考出非用特殊資料結構(如 map) 才能解的題目，但是很多題目如果使用這些資料結構可以簡化我們的程式，在考試與比賽時，程式碼的長短對犯錯機率與 debug 影響很大，所以要不要學，要看個人狀況而定，程度已經夠了，就多學一點好用的工具，如果對基本的程式還很生疏，就練習熟練後再說。

```

// P_3_10 shortest all-color range, using map
#include<bits/stdc++.h>
using namespace std;
#define N 1000010
int seq[N], cnt[N]={0}; // counter of color
map<int,int> dic; // dictionary

int main() {
    int n;
    scanf("%d",&n);
    // input and discretization
    for (int i=0;i<n;i++) {
        scanf("%d",&seq[i]);
        dic[seq[i]]=0; // insert if not existing
    }
}

```

```

}
int n_color=0; // num of color
for (auto &p:dic) p.second=n_color++;
// find initial window of all color
int left=0, right=0, w_color=0, shortest=n;
// w_color = num of color in window [left, right-1]
// sliding window
while (right<n) {
    int color=dic[seq[right]];
    cnt[color]++;
    if (cnt[color]==1) w_color++;
    right++;
    // shrink left until left color appear only once
    while (1) {
        color=dic[seq[left]];
        if (cnt[color]==1) break;
        cnt[color]--;
        left++;
    }
    if (w_color==n_color)
        shortest=min(shortest, right-left);
}
printf("%d\n",shortest);
return 0;
}

```

這一題因為顏色數字並沒有大小關係，所以其實順序不重要，離散化的過程可以用 `unordered_map` 取代 `map`，不過這份教材並沒有打算介紹那個資料結構，所以就不多說了，但範例程式中有附，有興趣的人可以自己參考。

上面的寫法，如果不計離散化，時間複雜度都只有 $O(n)$ ，這一題應該已經完美了，但以下還要提出一個比較慢的方法！慢且(台語)！學完快的為什麼還要學慢的呢？解題不是重點，技術才是重點，因為題目千變萬化，只有學到了技術才有用。下面這個方法饒富趣味，而且後面的章節我們也會碰到。

對於一個固定的輸入序列來說，令 $f(L)$ 是長度 L 的連續區段可以擁有的最多顏色數。那麼，P-3-9 就是對輸入的 L 計算 $f(L)$ 的值，而 P-3-10 則是要計算滿足 $f(L) \geq nc$ 的最小 L 值，其中 nc 是總共的顏色數。函數 $f()$ 明顯有單調性，因為長度更長的情況下，最多擁有的最多顏色數只會增加或不變，不會變少。因此，如果把 P-3-9 的程式寫成函數，然後不斷的呼叫 $f()$ 去計算 $f(1)$, $f(2)$, ...，直到第一個回傳值不小於 nc 時，我們就找到了 P-3-10 的解。且慢，既然 $f()$ 具單調性，線性搜尋就太遜了，我們當然可以用二分搜。以下是這樣的想法寫出的程式，時間複雜度是多少？如果離散化不計，P-3-9 是 $O(n)$ ，長度範圍是 $1 \sim n$ ，二分搜會呼叫 $O(\log(n))$ 次，所以是 $O(n \log(n))$ 。

```

// P_3_10 shortest all-color range, repeatedly call P-3-9
// using binary search to find shortest
#include<bits/stdc++.h>
using namespace std;
#define N 1000010
int seq[N];
int dic[N]; // dictionary, map color to 0~m-1
// max num of color of window size w
int w_col(int w, int n) {
    vector<int> cnt(n, 0); // initial counter
    int i, j, nc=0;
    for (i=0;i<w;i++) {
        if (++cnt[seq[i]] ==1)
            nc++;
    }
    int maxc=nc;
    // insert i, delete j
    for (j=0, i=w;i<n;i++,j++) {
        if (++cnt[seq[i]] ==1)
            nc++;
        if (--cnt[seq[j]] ==0)
            nc--;
        maxc=max(maxc,nc);
    }
    return maxc;
}

int main() {
    int n;
    scanf("%d",&n);
    for (int i=0;i<n;i++)
        scanf("%d",&seq[i]);
    // discretization
    for (int i=0;i<n;i++)
        dic[i]=seq[i];
    sort(dic, dic+n);
    // remove duplicate color
    int n_color=1; // num of color
    for (int i=1;i<n;i++) { // not checking dic[0]
        if (dic[i]!=dic[i-1]) {
            dic[n_color]=dic[i];
            n_color++;
        }
    }
    // replace color with its rank
    for (int i=0;i<n;i++)
        seq[i]=lower_bound(dic,dic+n_color,seq[i])-dic;
    // end of discretization
    if (n_color==1) {printf("1\n"); return 0;}
    // using binary search to find the longest invalid length
    int length=1;
    for (int jump=n/2; jump>0; jump>>=1) {
        while (w_col(length+jump,n)<n_color)
            length+=jump;
    }
}

```

```

}
printf("%d\n", length+1);
return 0;
}

```

接下來兩題與前兩題類似，我們當做習題。

Q-3-11. 最長的相異色彩帶

有一條細長的彩帶，彩帶區分成 n 格，每一格的長度都是 1，每一格都有一個顏色，相鄰可能同色。如果一段彩帶其中的每一格顏色皆相異，則稱為「相異色彩帶」。請計算最長的相異色彩帶的長度。

Time limit: 1 秒

輸入格式：第一行為整數 n ，滿足 $n \leq 2 \times 10^5$ ；第二行有 n 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號是不超過 n 的非負整數。

輸出：最長的相異色彩帶的長度。

範例輸入：

```

10
6 4 1 6 0 4 5 0 7 4

```

範例結果：

```

5

```

說明：區間 $[3, 7]$ 的顏色 $(1, 6, 0, 4, 5)$ 皆不相同。

Q-3-12. 完美彩帶 (APCS201906)

有一條細長的彩帶，總共有 m 種不同的顏色，彩帶區分成 n 格，每一格的長度都是 1，每一格都有一個顏色，相鄰可能同色。長度為 m 的連續區段且各種顏色都各出現一次，則稱為「完美彩帶」。請找出總共有多少段可能的完美彩帶。請注意，兩段完美彩帶之間可能重疊。

Time limit: 1 秒

輸入格式：第一行為整數 m 和 n ，滿足 $2 \leq m \leq n \leq 2 \times 10^5$ ；第二行有 n 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號是不超過 10^9 的非負整數，每一筆測試資料的顏色數量必定恰好為 m 。

輸出：有多少段完美彩帶。

範例輸入：

```
4 10
1 4 1 7 6 4 4 6 1 7
```

範例結果：

3

說明：區間 $[2, 5]$ 是一段完美彩帶，因為顏色 4、1、7、6 剛好各出現一次，此外，區間 $[3, 6]$ 與 $[7, 10]$ 也都是完美彩帶，所以總共有三段可能的完美彩帶。

下面的習題與 P-3-8 類似。

Q-3-13. X 差值範圍內的最大 Y 差值

輸入平面上 N 個點的座標 $(x[i], y[i])$ 以及一個正整數 L ，計算並輸出

$$\max_{1 \leq i < j \leq N} \{ |y[i] - y[j]| : |x[i] - x[j]| \leq L \}。$$

Time limit: 1 秒

輸入格式：第一行是 N 與 L ，第二行各點的 x 座標，第三行依序是對應點的 y 座標，相鄰數字間以空白隔開。 $N \leq 2e5$ ，座標絕對值不超過 $1e9$ 。

輸出：輸出所求的最大差值。

範例輸入：

```
10 3
4 1 2 -10 3 5 6 9 7 8
6 1 4 10 3 9 8 1 5 7
```

範例結果：

7

說明： x 距離 3 以內的最大 Y 差值是 $(6, 8)$ 與 $(9, 1)$ ， Y 值差 7。

下面這個習題需要一點幾何知識，別擔心，國中就學過了。

Q-3-14. 線性函數 (00)

有 N 線性函數 $f_i(x) = a_i x + b_i$, $1 \leq i \leq N$ 。定義 $F(x) = \max_i f_i(x)$ 。輸入 $c[i]$, $1 \leq i \leq m$, 請計算 $\sum_{i=1}^m F(c[i])$ 。

Time limit: 1 秒

輸入格式：第一行是 N 與 m 。接下來有 N 行，依序每行兩個整數 a_i 與 b_i ，最後一行有 m 個整數 $c[1], c[2], \dots, c[m]$ 。每一行的相鄰數字間以空白隔開。 $N \leq 1e5$, $m \leq 5e4$, 輸入整數絕對值不超過 $1e7$, 答案不超過 $1e15$ 。

輸出：計算結果。

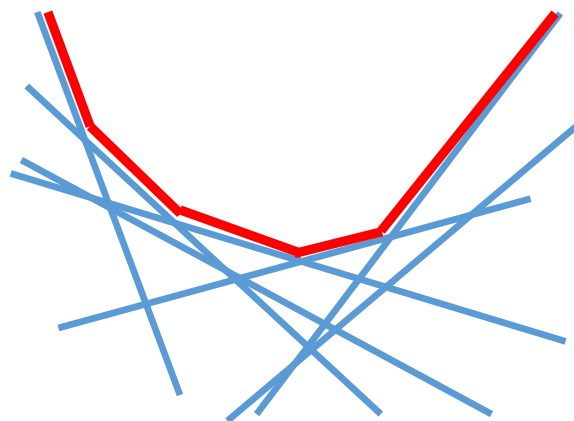
範例輸入：

```
4 5
-1 0
1 0
-2 -3
2 -3
4 -5 -1 0 2
```

範例結果：

```
15
```

提示：給 N 個線性函數（一次函數），另外定義 F 是這些函數的最大值，現在給 m 個 x 值，要計算 F 在這些點的函數值總和，也就是對每一個 x 值，要計算這些線性函數的最大值。直接的做法全部計算取最大，總共需要 $O(mn)$ ，效率不佳。請看右圖。藍線是這些線性函數，紅線就是 F ，他必定會形成一個（向下）凸的形狀，而且是一段一段的，每一段都是原來的某個藍線（函數），如果我們可以



找出這個紅線，計算這些函數值就沒問題了。我們需要一個觀察：若 f 與 g 是兩個線性函數， f 的斜率小於 g ，那麼，在兩線的交點以前是 f 大，以後是 g 大。我們可以將這些直線根據斜率排序，然後逐一將直線加入 F 。這一題另外也有分治的演算法可以做，在後續章節中會再出現。

4. 貪心演算法與掃描線演算法

「貪心通常比較簡單，但多半不能解決問題，在演算法與人生中都是這樣。」

孔子說：「及其老也，血氣既衰，戒之在得。」日常生活中最常看到貪的狀況就是政治人物，而想到政治人物就想到金庸的笑傲江湖。這一章的例題習題部分就以笑傲江湖中的人物來串場，不過不必擔心，沒看過故事不影響解題。

這一章介紹貪心演算法 (Greedy algorithm) 以及掃描線演算法 (sweep-line algorithm)，這兩種演算法有點類似，所以放在同一章。這裡的貪心沒有負面的意思，單純指以當前最有利的方式做決定。貪心演算法通常很直覺，所以寫程式往往簡單，但不是對所有問題都能夠找到最佳解。Greedy 通常需要證明，但上機考試與競賽的場合，不需要證明，所以很多人不知道怎麼做的時候就用 Greedy 寫下去猜猜看，考試本來就是該猜就要猜，不過貪心也要貪的正確。掃描線演算法的名字出自計算幾何，是指解題過程想像用一條掃描線沿著某個方向掃描過去，他是一個很好的思考模式，很多經典的分治演算法都有掃描線的版本。有一些數列的問題往往可以看成數線上的問題，所以也可以用掃描線的方式來思考。

貪心演算法與掃描線演算法通常需要搭配排序或優先佇列 (priority_queue, 常簡稱 PQ)，我們在第二章介紹過排序了，所以這裡不再重複。優先佇列是一種重要的資料結構，在大部分資料結構的課本中都有介紹自製的方法 (heap) 以及應用，它在 STL 中也有內建的，但它應該不能算“常用簡單”的資料結構，所以跟 set/map 的狀況類似，在 APCS 考試中應該不會出現非用它不可的題目，我們會對它 PQ 做簡單的介紹。

4.1. 基本原理

所謂貪心演算法指的是：一個算法由一連串的決定組成，每次決定時都以當時最有利的方式挑選，而且一旦挑選後就不再更改。我們先以舉一個很簡單的例子來說明。

P-4-1. 少林寺的代幣

令狐沖去少林寺觀光，少林寺有四種代幣，面額分別是 [1, 5, 10, 50]，令狐沖拿了 n 元去換代幣，請問以這四種代幣湊成 n 元，最少要多少枚代幣？

Time limit: 1 秒

輸入格式：第一行是一個正整數 m ，代表有幾筆測資，以下有 m 行，每行是一筆測資，每筆測資是一個正整數 n ， $n < 1000$ 。

輸出：依序每一行輸出每一筆測資的最少的代幣數量。

範例輸入：

```
3
21
50
123
```

範例結果：

```
3
1
7
```

說明： $21 = 10 * 2 + 1$ ， $50 = 50 * 1$ ， $123 = 50 * 2 + 10 * 2 + 1 * 3$ 。

因為代幣的數量希望要最少，直覺貪心的想法就是要盡量使用面額大的硬幣，代幣由大面額的往小的考慮，能換盡量換，剩下不足的再考慮比較小的面額。

```
#include <stdio>

int main() {
    int coin[4]={1,5,10,50};
    int m,n;
    scanf("%d",&m);
    for (int i=0;i<m;i++) {
        scanf("%d", &n); // input money
        int num=0; // num of coins
        for (int j=3; j>=0 && n>0; j--) { // large to small
            num += n/coin[j]; // num of coin[j]
            n %= coin[j]; // remaining
        }
        printf("%d\n",num);
    }
    return 0;
}
```

程式寫起來很簡單，簡單到如果考試碰到這種題目，會懷疑是不是有陷阱。這個方法很直覺，但怎麼知道它是對的呢？有些人可能會說：就很直覺啊，路邊隨便找個少林寺掃地的也會這樣做。那麼請看看下面的狀況：假設代幣的面額是 [1, 4, 5] 三種，上面的貪心方法會找到正確答案嗎？假設 $n=8$ ，貪心的方法會先換一個 5 元，然後剩下 3 元

就換 3 個 1 元代幣，所以 8 元換了 4 個代幣。但是如果拿兩個 4 元代幣就可以湊 8 元了，所以正確答案應該是 2 而不是 4。請不要懷疑人生，P-4-1 的做法是對的，只是並不是能用在任何場合，我們必須證明。如何證明呢？

在代幣是 $[1, 5, 10, 50]$ 的情形下，如果 1 元的代幣總額不小於 5，一定可以拿出 5 個換成一個 5 元，所以 1 元代幣的個數不會超過 4 個；同樣的你可以證明 5 元代幣不會超過 1 個，10 元代幣不會超過 4 個。也就是說，對任意第 j 種代幣 $\text{coin}[j]$ ，小於 $\text{coin}[j]$ 的代幣總額不會超過 $\text{coin}[j]$ ，所以 Greedy 是對的。在 $[1, 4, 5]$ 的狀況，就沒有這個性質了。順便一提，不要小看少林寺的掃地僧，掃地僧往往是最厲害的。

這個例子中我們看到了貪心法的一些特質。

- 算法由一連串決定組成 (由大到小逐一考慮各種代幣的數量)，每次考慮當前最好決定 (能換盡量換)，選擇後就不再更改。
- 算法的正確性必須經過證明，內容就是要證明所下的決定是對的，也就是一定有一個最佳解包含了貪心法所做的決定，而證明的方法通常都是用換的：假設有一個最佳解跟貪心法做的決定不一樣，那麼，我們可以換成一個與貪心法一樣的決定，而解會更好，或至少不會變差。

在下一節中我們會看到更多的例子。貪心算法往往都是要不斷的找最大值或最小值，如果資料不會變動，可以在開始的時候做一次排序，如果資料是會變動的呢？就必須用一個動態的資料結構來處理，以下我們對優先佇列做一點介紹。

優先佇列 (priority_queue)

第三章介紹過佇列，佇列跟現實生活中排隊的狀況一樣，是先進者先出，但更貼近現實生活中的狀況是有些事情後發生的卻必須先處理，因為緊急或是更重要。優先佇列就是具有優先權的排列，不管進入的順序，優先權高的會排在前面。STL 中有提供內建的 PQ，資料結構的課本中也都有教以 heap 來實作，heap 是一個完全二元樹 (complete binary tree)，但實作在陣列中，也就是心中有樹但程式中沒有樹。heap 的實作並不算困難，但考試與比賽時間寶貴，通常能使用內建的就不要自己寫，因此本書中並不介紹 heap 的實作，有興趣的可以自己查詢網路或書籍。優先佇列 (PQ) 常用的操作與時間複雜度如下，其中 n 表示成員的佇列的大小：

- 加入新元素 (push)， $O(\log(n))$ 。
- 查詢最大值 (top)， $O(1)$ 。
- 刪除最大值 (pop)， $O(\log(n))$ 。

- 檢查是否為空 (empty) , $O(1)$ 。

請注意刪除與查詢只能針對最大值。以下介紹 PQ 的用法，PQ 內定是最大值在 top，如果是基本型態 (如 int) 而且以內定大小的比較函數，使用起來簡單。它也跟排序一樣可以使用自定的結構與自定的比較方式，但比較複雜，因為涉及一些物件導向的寫法。我們只介紹一些簡單的用法。以下的範例程式介紹基本用法，包含以 pair 來做兩個欄位的資料的寫法，若要改變大小順序，最簡單的方法就是改變 key 的正負號。

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

int main() {
    int a[10]={1,4,1,3,8,5,4,9,8,7}, n=10;
    for (int i=0;i<n;i++) printf("%d ",a[i]);
    printf("\nIteratively insert and report max\n");
    priority_queue<int> pq;
    for (int i=0;i<n;i++) {
        pq.push(a[i]);
        printf("%d ",pq.top());
    }
    printf("\n");
    int sum=0;
    while (!pq.empty()) {
        sum+=pq.top();
        pq.pop();
    }
    printf("Delete all and sum=%d\n",sum);
    printf("PQ of struct, top is smallest, insert backward\n");
    priority_queue<pair<int,int>> pq2; // (value, index)
    for (int i=n-1;i>=0;i--) {
        pq2.push({-a[i],i}); // push minus value
        printf("a[%d]=%d, ",pq2.top().second,-pq2.top().first);
    }
    printf("\n");
    return 0;
}
```

在第二章我們介紹過 set/map，set 也是動態的 (可以支援加入與刪除)，而且也有大小順序，所以是不是可以用來找 PQ 呢？set 的寫法比較簡單用途也比較廣，這樣可以偷懶少學一種。如果我們用 multiset (因為可能重複)，我們可以在 $O(1)$ 找到最大值，新增與刪除也都是 $O(\log(n))$ 。所以基本上是可以，畢竟會使用 set 已經很不錯。下面的範例程式，我們展示以 set 來代替並且測試比較兩者的速度。我們把 set 的大小關係改成大於，這樣 begin() 就是最大值。測試後發現 set 的時間超過 PQ 的兩倍以上。所以如果是要參加競賽的人，最好還是多學一種比較好。

```

// compare pq and multiset
#include <bits/stdc++.h>
using namespace std;
#define N 1000000
multiset<int,greater<int>> S;
priority_queue<int> pq;

int main() {
    int a[N], n=N;
    for (int i=0;i<n;i++) a[i]=rand();

    clock_t t1, t2;
    long long sum=0;
    t1=clock();
    for (int i=0;i<n;i++) {
        S.insert(a[i]);
        sum+=*S.begin();
    }
    while (!S.empty()) {
        sum+=*S.begin();
        S.erase(S.begin());
    }
    printf("%lld\n",sum);
    t2=clock();
    printf("set takes %f sec\n", (float)(t2-t1)/CLOCKS_PER_SEC);
    sum=0;
    t1=clock();
    for (int i=0;i<n;i++) {
        pq.push(a[i]);
        sum+=pq.top();
    }
    while (!pq.empty()) {
        sum+=pq.top();
        pq.pop();
    }
    printf("%lld\n",sum);
    t2=clock();
    printf("PQ takes %f sec\n", (float)(t2-t1)/CLOCKS_PER_SEC);
    return 0;
}

```

4.2. 例題與習題

這一節我們來看貪心與掃描線演算法的題目，我們將題型區分為以下幾類，每一類挑選一些經典題來說明，另外在下一節有一些其他的補充習題：

- 單欄位資料排序
- 結構資料排序

- 以 PQ 處理動態資料
- 貪心搭配二分搜
- 掃描線演算法

很多貪心算法的重點在證明，但本書以程式實作為主，不以證明為重點，所以有些題目的證明只簡略說明重點，或者請讀者上網路查詢。

4.2.1. 單欄位資料排序

P-4-2. 笑傲江湖之三戰

笑傲江湖的遊戲中，你扮演令狐沖的角色，因為「長恨此身非我有，何時忘卻盈盈」，所以你率領任我行等魔教高手前往少林寺搭救任盈盈，欲知故事原委可參見笑傲江湖第 27 回「三戰」。套句丸尾班長的口頭禪，總而言之，少林寺與五嶽劍派等正教派出 n 位高手，而你方也有 n 位高手，每個人都有一個戰力值。雙方要進行一對一的對戰，每個人不管輸或贏都只能比一場，假設兩人對戰時，戰力值高的獲勝。對於對方的每一位高手，你可以指定派哪一位與其對戰，為了解救盈盈，你希望贏越多場越好，平手則沒有幫助。請計算出最多能贏多少場。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，第二行有 n 個非負整數是對方的戰力，第三行有 n 個非負整數是我方的戰力，同行數字間以空白間格。 n 與戰力值不超過 $1e5$ 。

輸出：輸出最多可能的勝場數。

範例輸入：

```
5
3 1 7 0 2
8 3 5 0 0
```

範例結果：

```
3
```

我們只要關心可以戰勝的那些場次需要誰去出戰，其他場次贏不了，就在剩下的人裡面隨便挑選去應戰。假設我方目前最弱的戰力是 a 而對方最弱的是 b ，如果 $a \leq b$ ，則 a

對誰都不會贏，是沒用的，可以忽略。否則 $a > b$ ，我們宣稱「一定有一個最佳解是派 a 去出戰 b 」，證明如下：

假設在一個最佳解中， a 對戰 x 而 y 對戰 b ，我們將其交換改成「 a 對 b 而 y 對 x 」。根據我們的假設 $y \geq a$ ，交換後 a 可以贏 b 而 y 對 x 的戰績不會比 a 對 x 差，所以交換後勝場數不會變少。

既然如此，我們可以確定派 a 對戰 b 一定可以得到最佳解，並將 a 與 b 移除後，繼續依照上述原則挑選。範例程式如下，因為要不斷的挑選最小值，而戰力不會改變，所以是個靜態的資料。我們可以將雙方戰力先進行一次排序，然後依序由小往大考慮即可。這一題也可以由大到小考慮，基本原則是能贏的贏一點就好，當然可以用優先佇列 (PQ)，在範例程式中有以 PQ 寫的方式，做法簡單，我們就不在此說明，有興趣練習 PQ 的可以自行參考。

```
// P_4_2, one-on-one O(nlogn), sorting+greedy
#include <bits/stdc++.h>
using namespace std;
#define N 100010
int main() {
    int n, enemy[N], ours[N];
    scanf("%d",&n);
    for (int i=0;i<n;i++)
        scanf("%d",&enemy[i]);
    for (int i=0;i<n;i++)
        scanf("%d",&ours[i]);
    sort(enemy, enemy+n); // sort enemy power
    sort(ours, ours+n); // sort our power
    int win=0; // num of win
    for (int i=0,j=0;i<n && j<n;i++) { // for each of our power
        // j is currently weakest enemy
        if (ours[i]> enemy[j]) { // can win some enemy
            win++;
            j++; // next enemy
        }
        // otherwise, ours[i] is useless
    }
    printf("%d\n",win);
    return 0;
}
```

P-4-3. 十年磨一劍 (最少完成時間)

人們常用說「十年磨一劍」來比喻下的功夫深，但是這句話對於華山磨劍坊就不適用了，因為要磨劍的客人非常的多，一把劍如果磨太久，客人等待的時間過長是會被客

訴的。華山磨劍坊目前有 n 筆訂單，每筆訂單要磨一把劍，每把劍所需的時間不盡相同。磨劍師傅每次只能磨一把劍，現在希望每一筆訂單的完成時間之總和能夠最小，希望能找出最好的磨劍順序。舉例來說，如果有四把劍，磨劍需要的時間分別是 $(3, 1, 3, 4)$ ，如果以 $(3, 1, 3, 4)$ 的順序來磨，第一把的完成時間是 3，第二把完成時間是 $3+1=4$ ，第三把是 $3+1+3=7$ ，第四把是 $3+1+3+4=11$ ，總和是 $3+4+7+11=25$ 。如果把順序改成 $(1, 3, 3, 4)$ ，那麼完成時間分別是 $(1, 4, 7, 11)$ ，總和是 23，這是這一題最好的解。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，第二行有 n 個正整數是每一把劍需要的時間，同行數字間以空白間格。 n 與每把劍的時間都不超過 $1e5$ 。

輸出：輸出最小的完成時間總和。

範例輸入：

```
4
3 1 3 4
```

範例結果：

```
23
```

有 n 個工作要安排一個順序後依序完成，希望每個工作的完成時間總和最小，這是個經典的 minimum completion time 問題。直覺上短的工作要先做，放在越前面的工作，等它的人越多。如果把計算時間的方式仔細看一下，你會發現如果 $job[i]$ 是第 i 個工作所需的時間，那麼完成總時間是 $\sum_{i=0}^{n-1} (n-i) \times job[i]$ ，所以 shortest-job-first 是對的。另外一個證明方法是假設 $job[i]$ 的順序是最佳解而存在某個工作 $job[i] > job[i+1]$ ，那麼，我們交換這兩個相鄰工作，其他的工作時間不會改變，但這兩個工作的完成時間總和會減少，如此可知，最佳解中不可能大的排在小的前面。這種考慮相鄰交換的情形是個證明的技巧，因為其它的工作不會改變，可以得到很簡單的證明。

程式非常簡單，排序後計算就好了，但是不要把計算寫成 $O(n^2)$ 的笨方法了，運用 prefix-sum 的手法就很容易在排序後以 $O(n)$ 完成計算，另外一個方法是用上面的公式。

```
// P_4_3, min waiting time, sorting+greedy
#include <bits/stdc++.h>
using namespace std;
```

```

#define N 100010
int main() {
    int n, job[N];
    scanf("%d",&n);
    for (int i=0;i<n;i++)
        scanf("%d",&job[i]);
    sort(job, job+n); // sort from small to large
    long long int comp=0, total=0;
    for (int i=0; i<n; i++) {
        comp += job[i]; // completion time of i
        total += comp; // total completion time
    }
    printf("%lld\n",total);
    return 0;
}

```

4.2.2. 結構資料排序

在上面兩個例題中，每個資料都只是一個數字，接下來我們要看需要使用 struct 來存資料的狀況。在第二章我們介紹過結構資料的排序，自訂結構排序時，必須寫一個比較函數，當然也可以利用 STL 中 pair 來偷懶，如果不熟悉的人，請再次參考第二章的方法。下面這一題是很有名的經典題(activity selection problem)。

P-4-4. 幾場華山論劍(activity selection)

自從華山論劍聲名大噪之後，想參加的人絡繹不絕，因此華山派決定每年都舉辦非常多場的華山論劍，每一場都有自己的開始時間與結束時間，參加者必須全程在場，所以不能在同一時間參加兩場。令狐冲拿到了今年所有場次的資料，希望能參加越多場越好，以便盡速累積經驗值，請你幫忙計算最多可以參加幾場。請注意，所挑選活動必須完全不重疊，兩場活動只在端點重疊也是不可以同時參加的，也就是說前一場的結束時間必須小於下一場的開始時間。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，代表一共舉辦了 n 場華山論劍，接下來 n 行，每行兩個非負整數 s 與 t ，代表這場活動的時間區間是 $[s, t]$ ，兩數字間以空白間格。 n 不超過 $1e5$ ， $s < t < 1e9$ 。

輸出：最多參加幾場活動。

範例輸入：

4

```

1 4
0 3
3 4
4 6

```

範例結果：

2

說明：可以挑選 $[0, 3]$ 與 $[4, 6]$ 兩場活動。

這個題目簡單說就是：數線上有若干線段，要挑選出最多的不重疊的線段。第一個想法可能是盡量挑短的，但是範例中有一個反例，挑短的不是正確答案。我們宣稱下面這個的性質：

「若 $[x, y]$ 是所有現存線段中右端點最小的，那麼一定有個最佳解是挑選了 $[x, y]$ 。」

證明：假設最佳解沒有挑 $[x, y]$ ，令 $[a, b]$ 是最佳解中右端點最小的。根據我們的假設， $y \leq b$ ，因此將 $[x, y]$ 取代 $[a, b]$ 不會重疊到任何最佳解中的其他線段，我們可以得到另外一個最佳解包含 $[x, y]$ 。

根據此性質，我們可以不斷地將最小右端的線段挑選進解答，然後略過任何與它重疊的線段。因為線段不會改變，一開始依照右端由小排到大，然後從頭到尾掃描一次就可以得到答案。以下是範例程式，在此程式中我們定義了一個 struct 包含兩個整數欄位，用來存放一個線段的左右端點。為了排序，定義了一個比較函數 cmp，用來比較線段的右端點，對結構排序不熟悉的請複習第二章。在排序後，以一個迴圈從頭到尾掃描一次，以變數 endtime 保持著前一個已挑選活動的結束時間（初值給 -1 因為輸入的時間皆為非負），迴圈內以 if 來判斷 ac[i] 開始時間是否大於 endtime，若是，挑選起來，修改答案與 endtime；若否，可以直接丟棄它（不做任何事）。

```

// P_4_4, activity selection, sorting+greedy
#include <bits/stdc++.h>
using namespace std;
#define N 100010
struct ACT{
    int s,f; // start and finish time
};
// compare finish time
bool cmp(ACT p, ACT q) {
    return p.f<q.f;
}
int main() {
    int n;
    ACT ac[N];
    scanf("%d",&n);
    for (int i=0;i<n;i++)

```



```

scanf("%d%d",&ac[i].s, &ac[i].f);
sort(ac, ac+n, cmp); // sort from small to large
int endtime=-1, total=0;
for (int i=0; i<n; i++) {
    if (ac[i].s>endtime) { // compatible
        total++;
        endtime=ac[i].f;
    }
}
printf("%d\n",total);
return 0;
}

```

我們當然可以利用 STL 的 `pair` 來逃避寫比較函數，以下是這樣的寫法，有興趣的參考。我們用 `vector` 來寫，你當然可以用陣列來寫，重點是把右端點放在第一個欄位，因為 `pair` 的大小關係是依欄位順序的字典順序。

```

// P_4_4b,activity selection, sorting+greedy
#include <bits/stdc++.h>
using namespace std;
#define N 100010

int main() {
    int n;
    vector<pair<int,int>> act;
    scanf("%d",&n);
    for (int i=0;i<n;i++) {
        int s,t;
        scanf("%d%d",&s, &t);
        act.push_back({t,s}); // key = finish time
    }
    sort(act.begin(), act.end()); // sort by finish
    int endtime=-1, total=0; // end of previous activity
    for (auto p: act) {
        if (p.second>endtime) { // compatible
            total++;
            endtime=p.first; // set end-time
        }
    }
    printf("%d\n",total);
    return 0;
}

```

在 P-4-3 十年磨一劍的例題中，每一件磨劍工作，無論需要時間的長短，重要性都是一樣的，我們要求的只是完成時間的總和最小化，下面是一個變化題，每一個工作延後的代價不同。

P-4-5. 嵩山磨劍坊的問題 (加權最小完成時間)

嵩山磨劍坊接了 n 筆磨劍工作，磨劍師傅每次只能磨一把劍。除了每把劍各有所需的時間之外，每件工作的重要性也不同。假設第 i 筆訂單需要的時間是 $t[i]$ ，而重要性是 $w[i]$ 。磨劍坊的計價方式是：每件工作都已經先收了一筆款項，假設第 i 筆訂單在時間 f 時完成，則需要扣款 $f*w[i]$ ，現在希望將 n 筆磨劍工作安排最好的順序，使得扣款總額越小越好。嵩山派掌門左冷禪是非常嚴厲的老闆，希望你能幫磨劍師傅找出最好的順序以免他遭到處罰。

舉例來說，如果有四把劍，磨劍需要的時間分別是 $t=(1, 4, 5, 6)$ ，而重要性依序是 $w=(1, 3, 4, 7)$ 。如果依訂單編號順序 $(1, 2, 3, 4)$ 來磨，也剛好是工作時間由短到長的順序，每件工作的完成時間依序是 $(1, 5, 10, 16)$ ，扣款總額是 $1*1 + 5*3 + 10*4 + 16*7 = 168$ 。如果依照訂單編號順序 $(4, 1, 3, 2)$ 來磨，則 t 與 w 重新排列後分別是 $t=(6, 1, 5, 4)$ ， $w=(7, 1, 4, 3)$ 。完工時間是 $(6, 7, 12, 16)$ ，扣款總額是 $6*7 + 7*1 + 12*4 + 16*3 = 145$ 。這是這一題 24 種排列中最好的解。

Time limit: 1 秒

輸入格式：輸入的第一行工作數 N ， $N < 1e5$ 。第二行有 N 個正整數，依序是各訂單所需時間 $t[1]$ 、 $t[2]$ 、...、 $t[N]$ 。第三行有 N 個非負整數，依序是各訂單的重要性 $w[1]$ 、 $w[2]$ 、...、 $w[N]$ ，時間與重要性皆不超過 1000，相鄰以空白間隔。

輸出：輸出最小的扣款總額。答案不超過 $1e18$ 。

範例輸入：

```
4
1 4 5 6
1 3 4 7
```

範例結果：

```
145
```

如果每個工作(訂單)的重要性(w)都是一樣的話，這一題跟 P-4-3 就是一樣的，答案就是依照工作時間以 Shortest-job first 的方式排列，但是在有權重的狀況下，題目範例中就告訴你時間短的不一定在前面。我們再想想看，只看時間的話，時間短的應該排前面；那麼如果只看權重的話呢？權重大的應該排前面，因為延遲的話罰得重。但是兩個都要考慮的話，就必須兩個因子都列入考慮。我們宣稱以下特性：最佳解中，排在前面的， t/w 值一定比較小。證明？跟 P-4-3 類似，考慮排在相鄰的兩個工作，假設 $t[i]/w[i] > t[i+1]/w[i+1]$ 。我們交換兩工作，其他的工作完成時間不會改

變，而 i 延後了 $t[i+1]$ 的時間， $i+1$ 則提早了 $t[i]$ 的時間，因此變化量為 $w[i]*t[i+1] - w[i+1]*t[i] < 0$ ，也就是解會變好。程式的寫法不難，但是需要兩個欄位 struct 的排序，而大小的比較函數是以交叉相乘的方式。這裡要提醒，在類似的情形下要注意資料範圍與 overflow 的問題。以這一題的範圍，1~1000，相乘之後不會有問題，如果數字比較大，相乘之前可能需要轉 long long，另外分數的比較也要注意負號，如果有分母有可能是負整數，交叉相乘要小心。

```
// P_4_5 greedy, using c++ sort+struct
#include <bits/stdc++.h>
using namespace std;
struct Item {
    int t,w;
};
bool comp(Item p, Item q) {
    return p.t * q.w < p.w * q.t;
}

int main() {
    int n;
    scanf("%d", &n); assert(n<100000);
    Item Q[N];
    for (int i=0; i<n; i++)
        scanf("%d", &Q[i].t);
    for (int i=0; i<n; i++)
        scanf("%d", &Q[i].w);
    for (int i=0; i<n; i++)
        assert(Q[i].t>0 && Q[i].t<=1000 \
            && Q[i].w>=0 && Q[i].w<=1000);
    sort(Q, Q+n, comp);
    long long ans=0, comp_t=0; // completion time
    for (int i=0; i<n; i++) {
        comp_t += Q[i].t;
        ans += comp_t * Q[i].w;
    }
    printf("%lld\n", ans);
    return 0;
}
```

或許有人想知道，既然是依照 t/w 排序，何不直接相除當做 key？是可以，但必須考慮兩個問題，第一是精準度夠不夠，是否會發生捨位誤差造成順序錯誤。以這一題來說是不會。第二個問題，分母是否有可能是 0？這一題是會。所以需要特殊處理一下，下面的範例程式是以這個方式寫的，另外用了 STL 的 tuple，因為有三個欄位。這個寫法其實沒有比較簡單，其實並不推薦，有興趣的參考就好。

```
// P_4_5b greedy, using c++ sort+tuple
```

```

// key of type double t/w, oo for weight 0,
#include <bits/stdc++.h>
using namespace std;
#define N 100010
#define oo 10000.0
int main() {
    int n;
    scanf("%d", &n);
    vector<tuple<double,int,int>> Q;
    for (int i=0; i<n; i++) {
        int t;
        scanf("%d", &t);
        Q.push_back({0,t,0});
    }
    long long total_w=0; // total weight
    for (int i=0; i<n; i++) {
        int w;
        scanf("%d", &w);
        total_w+=w;
        get<2>(Q[i])=w;
        get<0>(Q[i])=(w==0)? oo: (double)get<1>(Q[i])/w;
    }
    sort(Q.begin(), Q.end());
    long long ans=0; // another way to compute
    for (int i=0; i<n; i++) {
        ans += total_w * get<1>(Q[i]);
        total_w -= get<2>(Q[i]);
    }
    printf("%lld\n", ans);
    return 0;
}

```

下面一題很類似，因此留做習題，不多解釋。

Q-4-6. 少林寺的自動寄物櫃 (APCS201710)

少林寺的自動寄物櫃系統存放物品時，是將 N 個物品堆在一個垂直的貨架上，每個物品各佔一層。系統運作的方式如下：每次只會取用一個物品，取用時必須先將在其上方的物品貨架升高，取用後必須將該物品放回，然後將剛才升起的貨架降回原始位置，之後才會進行下一個物品的取用。

每一次升高貨架所需要消耗的能量是以這些物品的總重來計算。現在有 N 個物品，第 i 個物品的重量是 $w(i)$ 而需要取用的次數為 $f(i)$ ，我們需要決定如何擺放這些物品的順序來讓消耗的能量越小越好。

舉例來說，有兩個物品 $w(1)=1$ 、 $w(2)=2$ 、 $f(1)=3$ 、 $f(2)=4$ ，也就是說物品 1 的重量是 1 需取用 3 次，物品 2 的重量是 2 需取用 4 次。我們有兩個可能的擺放順序（由上而下）：

- $(1, 2)$ ，也就是物品 1 放在上方，2 在下方。那麼，取用 1 的時候不需要能量，而每次取用 2 的能量消耗是 $w(1)=1$ ，因為 2 需取用 $f(2)=4$ 次，所以消耗能量數為 $w(1) * f(2)=4$ 。
- $(2, 1)$ 。取用 2 的時候不需要能量，而每次取用 1 的能量消耗是 $w(2)=2$ ，因為 1 需取用 $f(1)=3$ 次，所以消耗能量數= $w(2) * f(1)=6$ 。

在所有可能的兩種擺放順序中，最少的能量是 4，所以答案是 4。再舉一例，若有三物品而 $w(1)=3$ 、 $w(2)=4$ 、 $w(3)=5$ 、 $f(1)=1$ 、 $f(2)=2$ 、 $f(3)=3$ 。假設由上而下以 $(3, 2, 1)$ 的順序，此時能量計算方式如下：取用物品 3 不需要能量，取用物品 2 消耗 $w(3) * f(2)=10$ ，取用物品 1 消耗 $(w(3)+w(2)) * f(1)=9$ ，總計能量為 19。如果以 $(1, 2, 3)$ 的順序，則消耗能量為 $3*2 + (3+4) * 3=27$ 。事實上，我們一共有 $3!=6$ 種可能的擺放順序，其中順序 $(3, 2, 1)$ 可以得到最小消耗能量 19。

Time limit: 1 秒

輸入格式：輸入的第一行是物品件數 N ， $N < 1e5$ 。第二行有 N 個正整數，依序是各物品的重量 $w(1)$ 、 $w(2)$ 、...、 $w(N)$ 。第三行有 N 個正整數，依序是各物品的取用次數 $f(1)$ 、 $f(2)$ 、...、 $f(N)$ ，重量與次數皆不超過 1000，相鄰以空白間隔。

輸出：輸出最小能量消耗值。答案不超過 $1e18$ 。

範例輸入：

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

範例結果：

```
19
```

4.2.3. 以 PQ 處理動態資料(*)

這一小節中我們提出需使用優先佇列(priority queue, PQ)的題目。由於 APCS 的範圍應該不包含 PQ，所以照理說不太會出現在 APCS 考試中。

P-4-7. 岳不群的併派問題 (Two-way merge) (*)

江湖上門派林立，華山派掌門岳不群認為要消除門派的衝突就是要合併各門派，但不能操之過急，必須每次挑兩個門派合併，如此逐步合併，最後將所有門派合併成一派。合併門派需要一些成本，每個門派都有他的成員數，第 i 個門派的成員數是 $m(i)$ ，合併兩派的成本就是雙方成員數之和，而兩派合併後，雙方成員就會歸到新合併的門派。岳不群不希望耗費太多成本，輸入各派人數，請幫他計算最少的合併總成本。

舉例來說，一開始如果有三派， $m=(3, 5, 1)$ ，若 3 人與 5 人的兩派先合併，成本是 $3+5=8$ ，合併後剩下兩派，人數是 $(8, 1)$ ，再合併這兩派，成本是 $8+1=9$ ，此時已經只剩下一派，也就是完成所有合併統一江湖了，總成本是 $8+9=17$ 。如果我們更換合併的順序，先合併 3 人與 1 人的兩派 (成本 4)，再合併剩下的 5 人 (成本 $4+5=9$)，則總成本只有 $4+9=13$ 。這是這個例子的最低成本。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，代表初始的門派數量，第二行有 n 個正整數是每派的成員數，同行數字間以空白間格。 n 不超過 $2e5$ ，每個門派初始人數都不超過 $1e5$ 。

輸出：第一行輸出總人數，第二行輸出最小的合併總成本。

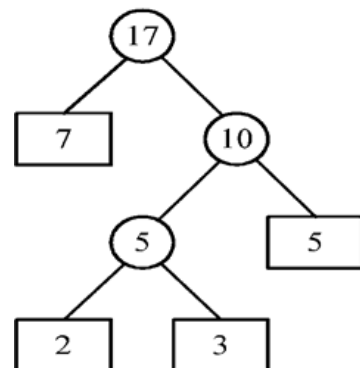
範例輸入：

```
3
3 5 1
```

範例結果：

```
9
13
```

這一題事實上是 Two-way merge tree 的問題，與一個有名的 Huffman code 是等價的。正確的合併順序的計算方法是：每次挑選人數最少的兩派合併，重複此步驟直到剩下一派為止。任何一種合併的過程可以用一個二元樹表示，右圖是一個例子，圖中每個方塊表示一個初始的門派，圓圈表示一個合併。初始人數 $m(i)$ 是 $(7, 2, 3, 5)$ ，先合併 $(2, 3)$ 變成 5，再合併 $(5, 5)$ 變成 10，最後合併 $(7, 10)$ 變成 17。合併的成本圓圈中的數字相加，這個例子 $5+10+17=32$ 就是最低成本。這一題的證明稍嫌麻煩，我們在此不給完整的證明，只做一些說明。總成本也可用另外一個



方式計算，每一個方塊往上走到頂會經過幾次圓圈 (稱為深度)，就代表他經過幾次合併，所以每個 $m(i)$ 乘上他的深度加總後就是總成本，所以直覺上看得出來，數字越大的要放在深度小的地方，也就是說數字越小的要越早合併，完整的證明留給讀者上網搜尋 Huffman code 或 two-way merge tree。

雖然證明有點麻煩，但是程式非常好寫，我們只要每次找出最小的兩個數，將他們刪除，再將他們的和放入。重複這個步驟直到剩下一個數字。但是要有效率的做到必須借助資料結構，如果使用排序，每次刪除最小值或許不是大問題 (記住有效位置，每次後移兩格就好)，但是插入兩數相加的和就會很浪費時間了。以下是用陣列的方法寫的範例程式，時間複雜度是 $O(n^2)$ 。雖然他效率不夠好，但是我們也使用了一些技巧來減少資料搬移的動作，所以也算個好的練習。

```
// P_4_7 greedy, slow method using array
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
#define N 400010
int main() {
    int n;
    LL m[N];
    scanf("%d", &n);
    for (int i=0; i<n; i++) {
        scanf("%lld", &m[i]);
    }
    sort(m,m+n);
    LL left=0, right=n-1, cost=0; // [left, right]=current data
    // merge, n reduced by 1 each time
    for ( ; n>1; n--) {
        // merge the smallest two sets
        LL x=m[left]+m[left+1];
        cost += x;
        // delete 0 and 1, insert x
        int j=right;
        left += 2; // delete two smallest
        while (j>=left && m[j]>x) { // find position
            m[j+1]=m[j];
            j--;
        }
        m[j+1]=x; // insert x
        right++;
    }
    printf("%lld\n%lld\n", m[left], cost);
    return 0;
}
```

要能夠有效率的做到增刪資料，我們必須借助資料結構，優先佇列每次可以找到最大值 (或最小)，是這一題最適合的資料結構，只要每次 pop 出兩個資料，相加後 push 回

去就可以了。注意數值的大小以及我們放在 PQ 中的要變號，以便每次找到最小值。檔案中還有一支程式是以 multiset 取代 PQ，留給有興趣的人參考，這裡就不多說明了。

```
// P_4_7 greedy, using PQ
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;

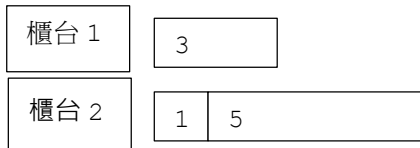
int main() {
    int n;
    scanf("%d", &n);
    priority_queue<LL> PQ;
    for (int i=0; i<n; i++) {
        int m;
        scanf("%d", &m);
        PQ.push(-(LL)m); // change sign (to find minimum)
    }
    LL cost=0; // total cost
    for (int i=0; i<n-1; i++) { // merge n-1 times
        // merge the smallest two sets
        LL m=PQ.top();
        PQ.pop();
        m += PQ.top();
        PQ.pop();
        PQ.push(m);
        cost += m;
    }
    printf("%lld\n%lld\n", -PQ.top(), -cost);
    return 0;
}
```

下面一個題目是在工作排程時的問題，做法簡單，但證明需要想一下，留做習題，如果資料量不大，可以用陣列做，但資料量大時，必須使用資料結構。

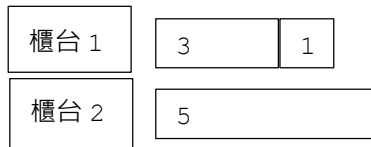
Q-4-8. 先到先服務 (*)

有 n 個客人要分配到 m 個櫃台來服務，客人依編號順序到達，第 i 個客人需要 $t(i)$ 的時間來完成（無論分配到哪一個櫃台），而且每個客人的需求必須在同一個櫃台完成。因為公平的因素，先到客人不可以比晚到客人更晚開始服務。現在希望安排每個客人的服務櫃檯，以便可以在最短的時間內完成所有客人的需求。

舉例來說，如果有三位客人與兩個櫃台，需求的時間依序是 (3, 1, 5)，我們可以如下圖分配，最後的完成時間是 6。



請注意，我們不可以如下分配，雖然這樣的完成時間會減少到 5，但是第 3 個客人比第 2 個客人早開始服務違反了先到先服務的原則。



Time limit: 1 秒

輸入格式：輸入兩行，第一行是正整數 n 與 m ，第二行是 n 個正整數 $t(1), t(2), \dots, t(n)$ ，同行數字間以空白間格。 n 與 m 不超過 $2e5$ ， $t(i)$ 不超過 $1e4$ 。

輸出：最早可能服務完所有客人的時間。

範例輸入：

```
3 2
3 1 5
```

範例結果：

```
6
```

4.2.4. 外掛二分搜

貪心演算法經常可以搭配二分搜，以下舉兩個例子。

P-4-9. 基地台 (APCS201703)

直線上有 N 個要服務的點，每架設一座基地台可以涵蓋直徑 R 範圍以內的服務點。輸入服務點的座標位置以及一個正整數 K ，請問：在架設 K 座基地台以及每個基地台的直徑皆相同的條件下，基地台最小直徑 R 為多少？

Time limit: 1 秒

輸入格式：輸入有兩行。第一行是兩個正整數 N 與 K ，以一個空白間格。第二行 N 個非負整數 $P[0], P[1], \dots, P[N-1]$ 表示服務點的點座標，相鄰數字以空白間隔。座標範圍不超過 $1e9$ ， $1 \leq K < N \leq 5e4$ 。

輸出：最小的基地台直徑。

範例輸入：

```
6 2
5 2 1 7 5 8
```

範例結果：

```
3
```

服務點可以看成數線上的點，基地台可以看成數線上的線段，要計算以 K 根長度相同的線段蓋住所有的點，最小的線段長度。這一題是給數量 K 要求長度 R 。要解決這個問題，先看以下問題：輸入給數線上 N 個點以及 R ，請問最少要用幾根長度 R 的線段才能蓋住所有輸入點。

我們以貪心法則來思考。考慮座標值最小的點 P ，如果有一個解的最左端點小於 P ，我們可以將此線段右移到左端點對齊 P ，這樣的移動不會影響解的合法性，因此，可以得到以下結論：「一定有一個最佳解（最少的 K ）是將一根線段的左端放在最小座標點上。」因此，我們可以放上一個線段在 $[P, P+R]$ ，然後略過所有以被蓋住的點，對剩下的點繼續此步驟。重複執行到所有的點被蓋住，就可以找到最小的 K 值。

回到我們的原來問題。假設 $f(R)$ 是給定長度 R 的最少線段數，那麼上述的演算法可以求得 $f(R)$ 。很直覺地，當 R 增加時， $f(R)$ 必然只會相同或減少，因為用更長的線段去蓋相同的點，不會需要更多線段。所以我們可以二分搜來找出最小的 R ，滿足 $f(R) \leq K$ 。

以下是範例程式。先寫一個函數 `enough(x)` 計算長度 x 的線段是否足夠，我們不需要算出最少的線段數量，只要知道 K 根是否足夠即可。主程式中將點座標排序後，就以二分搜來找出最長的“不足夠”長度，最後加一就是最短的足夠長度。這裡的二分搜採取第二章介紹的一路往前跳的方式，範例程式中也有傳統維護左右範圍的二分搜寫法，這裡就不多做說明。

```
// P_4_9 min length R to cover points by k segment of same length
// O(nlogL), L is distance
#include <bits/stdc++.h>
using namespace std;
#define N 50010
int p[N], n, k;
```

```

// check if k segment of length r is enough
bool enough(int r) {
    int nseg=k, endline = -1; // current covered range
    for (int i=0; i<n; i++) {
        if (p[i] <= endline) continue;
        if (nseg == 0) return false;
        // use a segment to cover
        nseg--; // remaining segments
        endline = p[i] + r;
    }
    return true;
}

int main() {
    scanf("%d%d", &n, &k);
    for (int i=0; i<n; i++)
        scanf("%d", p+i);
    sort(p, p+n);
    // binary search, jump to max not-enough length
    int len = 0, L = p[n-1] - p[0];
    for (int jump=L/2; jump>0; jump>>=1) {
        while (len+jump<L && !enough(len+jump))
            len += jump;
    }
    printf("%d\n", len+1);
    return 0;
}

```

下一題是類似的習題。

Q-4-10. 恢復能量的白雲熊膽丸

令狐沖闖黑木崖要依序通過 n 個關卡，第 i 個關卡要消耗 $p[i]$ 的能量，如果能量不足就會闖關失敗。不管當時的能量剩下多少，吃下一顆恆山派的「白雲熊膽丸」就會讓令狐沖的能量值恢復到滿額的 F 。假設在開始時能量是 F ，闖關過程中最多只能服用 m 次，輸入 $p[]$ 與 m ，請問令狐沖的能量額 F 至少必須是多少才能闖關成功。請注意，連吃兩顆是沒用的，能量還是 F 。

Time limit: 1 秒

輸入格式：第一行是正整數 n 與非負整數 m ，第二行，有 n 個正整數 $p[1], p[2], \dots, p[n]$ 。 $p[i]$ 不超過 $1e5$ ， $0 \leq m < n \leq 1e5$ 。

輸出：輸出 F 的最小值。

範例輸入：

```

7 2
3 4 1 7 4 1 2

```

範例結果：

8

說明：闖過三關(3, 4, 1)，吃藥，闖過(7)，再吃藥，闖過最後 3 關(4, 1, 2)。F=7 的話至少要吃三次藥才能通過。

4.2.5. 掃描線演算法 sweep-line

掃描線演算法的名字來自在解某些幾何問題時，想像用一條掃描線沿著某個方向掃描，過程中記錄某些資料或維護某些結構。因此與貪心演算法類似，往往第一步就是排序。常見的題目有數線上的點或線段，以及平面上的點或直線。限於難度，這裡僅提出一些經典而不太複雜的題目。

P-4-11. 線段聯集 (APCS 201603)

輸入數線上的 N 個線段，計算線段聯集的總長度。

Time limit: 1 秒

輸入格式：第一行是一個正整數 N ，接著的 N 行每一行兩個非負整數，代表一根線段的左端點與右端點，左端點座標值小於等於結束端點座標值，兩者之間以一個空格區隔。 N 不超過 $1e5$ ，座標絕對值皆不超過 $1e8$

輸出：線段聯集總長度。

範例輸入：

```
5
10 20
20 20
30 75
5 15
40 80
```

範例結果：

```
65
```

這一題有個比較簡單的常見版本是座標範圍很小而且線段數量也不多，例如 N 是 1000 而座標範圍在 $0 \sim 1000$ 。在此情形下把 $[0, 1000]$ 這個區間看成 1000 個長度為 1 的區段，以一個陣列 `seg[]` 來記錄每一個區段是否有被線段覆蓋。具體來說：

$seg[i] = 1$ if 區間 $[i, i+1]$ 有被覆蓋，否則 $seg[i] = 0$ 。

初始時，所有 $seg[i] = 0$ ，表示沒有線段。對於每一個輸入的線段 $[left, right]$ ，我們將陣列 $seg[left] \sim seg[right-1]$ 的區段都設為 1，模擬將線覆蓋在區間上。等到所有線段都處理完畢後，數一數有多少區段被設為 1 就知道覆蓋的總長度了。

```
// P-4-11, subtask, table method
#include <stdio.h>

int main() {
    int i, left, right, n;
    int seg[1001]={0}, total=0;
    scanf("%d", &n);
    for (i=0; i<n; i++) {
        scanf("%d%d", &left, &right);
        for (j=left; j<right; j++)
            seg[j] = 1;
    }
    for (i=0; i<1000; i++)
        total += seg[i];
    printf("%d\n", total);
    return 0;
}
```

這個方法的複雜度是線段長度的總和，在大座標範圍時的效率很差。若要與座標範圍無關，基本想法是將線段逐一加入，維護好目前的聯集 S ， S 顯然是一群不相連的線段。每次加入一根線段時，如果他與 S 中的線段沒有交集，那太好了，把他放進去就行了；如果有交集呢？就必須將有交集的部分找出來做成聯集。這樣做不是不行，但是非常的麻煩。我們試試看掃描線的想法。

想像有根掃描線，從左往右掃，過程中我們一樣保持住目前已經看到的線段的聯集 S 。掃描過程可能碰某個線段的左端或是右端，其他的位置對線段聯集的變化並無影響可以忽略。碰到某線段左端時，代表有一個線段要加入 S 。從左往右掃的好處是，當一個線段 $[x, y]$ 加入時，最多只會跟 S 中的一根線段(最後一根)有交集！因為他的左端 x 不會在 S 中任何一根線段的左方(記得吧，我們從左往右)。什麼時候沒交集呢？如果 x 大於 S 中的最大右端。而且，如果這個線段與 S 沒交集，後面的線段也都不會再跟 S 有交集，因為後面的線段的左端都大於等於 x 。這麼一來，程式就很好寫了，我們只要依照線段左端從小到大的順序逐一將線段加入 S ，維護好 S 中的最後一根線段就好了。以下是範例程式，我們定義了一個 `struct` 結構來放線段，並且定義一個比較函數 `cmp()` 以便依照線段左端排序，時間複雜度 $O(n \log(n))$ ，因為排序。

```
//P4-11 union of segments, using struct
```

```

#include <bits/stdc++.h>
using namespace std;
#define N 100010
struct Seg {
    int left, right;
};
bool cmp(Seg p, Seg q) {
    return p.left < q.left;
}
int main() {
    int n, total=0; // total length
    Seg s[N];
    scanf("%d",&n);
    for (int i=0; i<n; i++)
        scanf("%d%d", &s[i].left, &s[i].right);
    sort(s, s+n, cmp); // sort by left
    Seg last = s[0]; // last segment in the union
    for (int i=1; i<n; i++) { // insert each segment
        if (s[i].left > last.right) { // disjoint
            total += last.right - last.left;
            last = s[i];
            continue;
        }
        // else part, merge last and s[i]
        last.right = max(last.right, s[i].right);
    }
    total += last.right - last.left; // don't forget
    printf("%d\n",total);
    return 0;
}

```

偷懶的方法就是利用 `pair` 來放線段，把排序的依據放在 `first`，可以省卻寫比較函數。其實 C++ 的 `struct` 與比較函數都很好寫（比起 C），也不一定要學偷懶的方法。下面還是列出這個寫法給有興趣的人參考。

```

//P4-11 union of segments, using pair
#include <bits/stdc++.h>
using namespace std;
#define Left first
#define Right second

int main() {
    int i,n,total;
    vector<pair<int,int>> s;
    scanf("%d",&n);
    for (i=0; i<n; i++) {
        int p, q;
        scanf("%d%d", &p, &q);
        s.push_back({p,q});
    }
}

```

```

sort(s.begin(), s.end());
total = 0;
pair<int,int> last = s[0];
for (auto seg: s) {
    if (seg.Left > last.Right) { // disjoint
        total += last.Right - last.Left;
        last = seg;
        continue;
    }
    // else part, merge last and s[i]
    last.Right = max(last.Right, seg.Right);
}
total += last.Right - last.Left;
printf("%d\n",total);
return 0;
}

```

下面一個例題非常簡單又直覺，但是很有趣的是，我們稍後會看到他的另外一種形式，當他以另外一個形式出現時，卻似乎比較難理解。

P-4-12. 一次買賣

某商品在某個時期每一天的價格是 $p(1), p(2), \dots, p(n)$ 。假設只能先買後賣，請計算買賣一次的最大獲利價差，允許當天買賣，也就是一次都不買 (獲利 0)。

Time limit: 1 秒

輸入格式：第一行是正整數 n ，第二行有 n 個正整數 $p[1], p[2], \dots, p[n]$ 。 n 不超過 $1e5$ ，價格皆不超過 $1e9$ 。

輸出：買賣一次的最大價差。

範例輸入：

```

5
3 5 1 4 0

```

範例結果：

```

3

```

範例中暗示你，挑選最大值與最小值並非正確的答案。我們只需要考慮在第 i 天賣出的最大獲利，然後對各個 i 取最大值就是答案了。第 i 天賣出的最大獲利當然就是買在 i 以前出現的最小值，也就是 $i-1$ 的 prefix-minimum。所以我們可以一路掃描過去，維護好 prefix-min，再取較大的價差就好了，以下是範例程式。

```

//P4-12 max price difference, O(n)
#include <bits/stdc++.h>
using namespace std;
#define oo 1000001

int main() {
    int n, max_diff=0; // allow zero
    int p_min=oo; // prefix minimum
    scanf("%d",&n);
    for (int i=0; i<n; i++) {
        int p;
        scanf("%d", &p);
        max_diff=max(max_diff, p - p_min);
        p_min = min(p_min, p);
    }
    printf("%d\n",max_diff);
    return 0;
}

```

下面這一題是教學的經典名題，有人把他在貪心算法，也有人可能歸類到動態規劃。網路上找得到很多解法的介紹。

P-4-13. 最大連續子陣列

輸入一個整數陣列 $A[1:n]$ ，請計算 A 的連續子陣列的最大可能總和，空陣列的和以 0 計算。

Time limit: 1 秒

輸入格式：第一行是正整數 n ，第二行有 n 整數 $A[1]$, $A[2]$, ..., $A[n]$ 。 n 不超過 $1e5$ ，陣列內容絕對值不超過 $1e9$ 。

輸出：最大可能總和。

範例一輸入： 8 4 12 -17 5 8 -2 7 -3	範例一輸出： 18
範例二輸入： 3 -1 -1 -1	範例二輸出： 0

P-4-13 與 P-4-12 有何關係呢？P-4-12 的輸入是每日價格，如果把每日價格與前一日的價差放在 $A[]$ 中， $A[i]=p(i)-p(i-1)$ ，那麼就變成 P-4-13 了，因為連續一段時間的累積價差就等於最後一日與最前一日的價差

$$A[i]+A[i+1]+\dots+A[j]=(p(i)-p(i-1))+(p(i+1)-p(i))+\dots+(p(j)-p(j-1)) \\ =p(j)-p(i-1)$$

P-4-12 的 $O(n)$ 解法非常直覺明顯，而 P-4-13 的 $O(n)$ 解法則似乎比較不容易。當然你可以把輸入轉換成前一題的輸入，但我們也可以直接看這個題目。最簡單的解法是窮舉 $O(n^2)$ 個所有區間（子陣列），每一個區間再用一個迴圈求和，這樣導致了一個 $O(n^3)$ 的解。改善複雜度降到 $O(n^2)$ 有兩個方法，一種是對所有左端 i ，用一個迴圈求出所有 $[i, j]$ 區間的最大和，這個很容易做到因為

$$\text{sum}(A[i:j+1]) = \text{sum}(A[i:j]) + A[j+1]。$$

另外一種方法是利用前綴和（prefix sum）。先算好前綴和之後，每一個區間和可以用一個減法完成。

但是這兩種都不夠好，從前一題就知道這題可以在 $O(n)$ 求解。令 $f(i)$ 是以 i 為右端（各種可能的左端）的最大區間和。觀察 $f(i+1)$ 與 $f(i)$ 的關係， $f(i+1)$ 所選的左端如果在 i 之前，那麼一定與 $f(i)$ 的左端一樣。所以 $f(i+1)$ 只有以下兩種可能：

- $f(i+1)=f(i)+A[i+1]$ ，左端與 $f(i)$ 相同；或者
- $f(i+1)=A[i+1]$ ，左端在 $i+1$ 。

兩者挑大的就好了，事實上，這就像一代一代累積財產，若 $f(i)>0$ ，爸媽留給你的是正資產，則繼承過來；否則就拋棄繼承。這樣一比喻，這題的解其實也是非常直覺自然的。以下是範例程式。

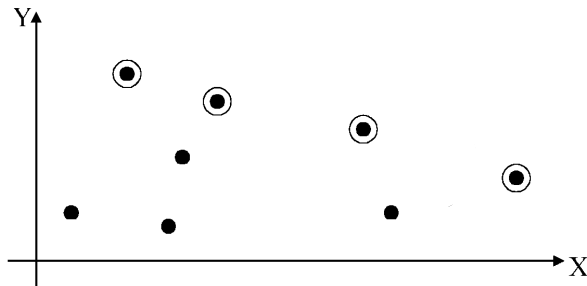
```
//P4-13 max subarray, O(n)
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
int main() {
    LL n, max_sum=0; // allow zero
    LL p_max=0; // prefix minimum
    scanf("%lld",&n);
    for (int i=0; i<n; i++) {
        LL p;
        scanf("%lld", &p);
        p_max=(p_max>0)? p_max+p : p;
        max_sum = max(max_sum, p_max);
    }
    printf("%lld\n",max_sum);
    return 0;
}
```

}

下兩題是平面幾何的題目，兩題都常被拿來當作分治演算法(下一章的主題)的教材，但事實上都有掃描線的解法。

P-4-14. 控制點 (2D-max)

平面上有 N 個點，第 i 點的座標為 $(x[i], y[i])$ 。若 $x[i] \leq x[j]$ 且 $y[i] \leq y[j]$ ，則我們說第 j 點可以控制第 i 點。我們希望能在輸入的點中，選取最少的點做為控制點，使得每一個輸入的點都至少被一個控制點所控制。輸入 N 個點的座標，請計算出最少要選取多少個控制點。以下圖為例，輸入 8 個點中，右上方圈起的四個點即是最少選取的控制點。



Time limit: 1 秒

輸入格式：每筆測試資料的第一行為一個正整數 N ， $N \leq 1e5$ 。第二行有 N 個非負整數，依序是這些點的 x 座標，第三行有 N 個非負整數，依序是這些點的 y 座標，座標值不超過 $1e9$ 。

輸出：輸出最少的控制點數量。

範例一：輸入 4 1 2 3 3 1 2 3 3	範例一：正確輸出 1
範例二：輸入 8 3 1 5 3 4 7 8 9 8 2 5 1 2 4 2 3	範例二：正確輸出 4

這個題目稱為 2D Maximal point，2D 平面上的 maximal point 是指沒有人的 xy 皆大於等於此點，是否會有相同座標點輸入則看題目定義，本題是允許，但本題中如果有兩個相同座標的極大點，只需要選一個。直接的做法可以兩兩比對，複雜度 $O(n^2)$ ，無疑是不夠好。假設我們從左到右掃描這些點，維護好目前找到的解 (maximal points)，假設做到第 $i-1$ 點時的解是 $S(i-1)$ ，考慮當第 i 點 $(x[i], y[i])$ 進入時的狀況。 $S(i-1)$ 中 Y 值不大於 $y[i]$ 的點都應該從解中移除，因為前面點的 x 值都不大於 $x[i]$ ；而 $(x[i], y[i])$ 必然是 $S(i)$ 中的點，因為它具有目前最大的 x 值。要如何刪除 $S(i-1)$ 中 Y 值不大於 $y[i]$ 的點呢？乍看之下以為要使用什麼特別的資料結構，確實，使用 set/map 一定可以做，但其實並不需要。因為 maximal 的點是以 x 值遞增的方式加入，他們的 y 值一定是遞減，只要從後往前刪除就可以了。從後往前刪除，然後自己加入到最後面，所以 Stack 是好的選擇，如果有印象的話，這個方法在前一章找前方的高人那個題目中使用過。請看以下範例程式，為了依照 x 排序，所以程式中定義了一個結構與一個比較函數。因為每個點最多只進入堆疊一次，除了排序外的時間複雜度是 $O(n)$ 。

```
//P-4-14 2d maximal points (forward sweep + stack) O(nlogn)
#include<bits/stdc++.h>
using namespace std;
#define N 100010
#define oo 1000000001
struct Point {
    int x, y;
};
bool less_x(Point a, Point b) {
    return a.x < b.x;
}
int main() {
    int i, j, n, x;
    scanf("%d", &n);
    Point p[N];
    for (int i=0; i<n; i++)
        scanf("%d", &p[i].x);
    for (int i=0; i<n; i++)
        scanf("%d", &p[i].y);
    sort(p, p+n, less_x); // sort from left to right
    stack<int> S; // keep y-value of current maximal
    S.push(oo); // a dummy maximal (-oo, oo)
    for (i=0; i<n; ++i) { // x from small to large
        while (p[i].y >= S.top())
            S.pop(); // delete smaller y
        S.push(p[i].y);
    }
    printf("%d\n", (int)S.size()-1);
    return 0;
}
```

事實上這個題目有更簡單的寫法。當我們從左往右掃描時，前面找到的 maximal point 有可能需要刪除，如果從右往左掃描呢？因為看到的點的 x 值是遞減，先找到的 maximal point 就一定是解！我們只要記住目前右方的最大 y 值就可以決定下一點是否需要加入解中了（如果他有更大的 y 值）。以下是範例程式，我們用 STL 的 vector 與 pair 來寫，為了讓 x 從大到小，我們存的時候存 $-x$ 。除了排序外，時間複雜度顯然是 $O(n)$ ，與前面那支一樣。

```
//P-4-14 2d maximal points backward sweep O(nlogn)
#include<bits/stdc++.h>
using namespace std;
int main() {
    int n, total=0; // num of maximal points
    scanf("%d",&n);
    vector<pair<int,int>> V;
    for (int i=0; i<n; i++) {
        int x;
        scanf("%d", &x);
        V.push_back({-x,0}); // change sign for large to small
    }
    for (int i=0; i<n; i++) {
        scanf("%d", &V[i].second);
    }
    sort(V.begin(), V.end());
    int max_y = -1; // max y value so far
    for (auto p: V) {
        int y = p.second;
        if (y > max_y) { // maximal point
            total++;
            max_y = y;
        }
    }
    printf("%d\n",total);
    return 0;
}
```

下一題也是平面上的點的問題，稍難，需要一點幾何特性。

P-4-15. 最靠近的一對(closest pair) (@@)

平面兩點的 L_1 距離為兩點的 x 差值與 y 差值的和，也就說，如果兩點座標是 (a,b) 與 (c,d) ，則 L_1 距離是 $|a-c|+|b-d|$ 。輸入 n 個點的座標，請計算出 L_1 距離最近兩點的 L_1 距離。

Time limit: 1 秒

輸入格式：第一行為一個正整數 n ，接下來 n 行，每行兩個整數 x 與 y 代表一點的座標。 n 不超過 $1e5$ ，座標值絕對值不超過 $1e8$ 。

輸出：最近兩點的 L_1 距離。

範例輸入：

```
4
-1 5
4 0
3 1
-2 -3
```

範例結果：

```
2
```

這個題目也是演算法的經典題，我們在下一章分治演算法也會再談到它，網路上可以找到很多相關資料與教學，大部分是直線距離的版本，也就是兩點之間的距離以直線距離計算，在線性代數中，直線距離稱為 L_2 ，與這一題的 L_1 距離雖然不一樣，但是類似， L_1 距離也有很多重要的應用，例如在 IC 設計上，網路上有很多說明資料。這一題用 L_1 只是為了計算更簡單而且不需要浮點數，演算法其實是一樣的。

想像一根垂直掃描線由右往左掃過去，計算出目前的解，也就是目前已經納入點的最小距離，每次碰到一個新的點，就找出新點與前面點的最近距離，並更新最小距離。如果有多點 x 值若相同，以任意順序並無影響。假設每個新進點都要跟前面的點計算距離，那這個方法跟窮舉沒兩樣，所以我們要想辦法減少計算的點數。

假設目前碰到的點是 $p[i] = (x[i], y[i])$ 而目前已求出來的最小距離是 d ，令 S 是 $p[0] \sim p[i-1]$ 這些點的集合 (如果沒有相同 x 可以簡單說 $p[i]$ 左方的點)，也就是說 S 中任兩點的距離至少是 d 。首先， S 中只有 Y 座標值在 $[y[i]-d, y[i]+d]$ 範圍內的點才需要計算，因為其他的點到 $p[i]$ 的距離必然大於 d ；此外，根據相同的理由， S 中 x 座標值小於 $x[i]-d$ 的點都不需要計算，而且不止這回合，以後碰到新點時，這些太左邊的點也都不需要計算了，因為越後面的點 x 座標越大。根據這兩個觀察，我們可以來寫程式了，為了有效地找到 Y 座標在 $[y[i]-d, y[i]+d]$ 範圍內的點，我們需要一個動態資料結構，因為需要增加與刪除資料。以下是範例程式，為了排序在第 6 行我們定義了一個 `struct Point` 以及比較函數 `less_x()`。第 19 行將點依照 x 座標由小到大排序，第 20 行宣告一個 `map`，用來存放已經掃過的點，`map` 中以 Y 座標當作 `key`，所以會將 Y 座標放在第一欄位。第 23 行開始的迴圈是進行逐點掃描的動作，每次碰到一個點，第 25 行以 `lower_bound()` 找到 Y 值在 $[y[i]-d, y[i]+d]$ 範圍內的第一點，然後 `while` 迴圈會跑範圍內的所有點，其中如果碰到 x 座標太小的點，就把它移除。第 31 行就更新可能的最小距離。

```

00 // P_4_15 closest pair, sweep line, L1 distance
01 #include<bits/stdc++.h>
02 using namespace std;
03 typedef long long LL;
04 #define N 100010
05 #define oo 200000001
06 struct Point {
07     int x,y;
08 };
09 bool less_x(Point &s, Point &t) {
10     return s.x < t.x;
11 }
12
13 int main() {
14     int n;
15     scanf("%d",&n);
16     Point p[N];
17     for (int i=0;i<n;i++)
18         scanf("%d%d",&p[i].x,&p[i].y);
19     sort(p, p+n, less_x); // sort by x
20     multimap<int,int> mm; // (y,x), sort by y
21     int min_d = oo; //min distance
22     // for each point from left to right
23     for (int i=0;i<n; i++) {
24         // check [p.y-d, p.y+d]
25         auto it=mm.lower_bound(p[i].y - min_d);
26         while (it!=mm.end() && it->first <= p[i].y+min_d) {
27             if (it->second < p[i].x - min_d) {
28                 it=mm.erase(it); // x too small, out of date
29                 continue;
30             }
31             min_d = min(min_d, \
32                 abs(p[i].x-it->second)+abs(p[i].y-it->first));
33             it++;
34         }
35         mm.insert({p[i].y,p[i].x});
36     }
37     printf("%d\n",min_d);
38     return 0;
39 }
40

```

這個程式因為用了動態資料結構 map，所以有點超過 APCS 的範圍，如果用分治演算法就可以避免 map，我們在下一章會介紹。

且慢，這個題目是不是還沒說明完畢？複雜度呢？複雜度其實是 $O(n \log(n))$ ，雖然不容易從程式碼中看出來，原因是 while 迴圈中，如果不計因為 x 座標太小被刪除的點，每個點計算距離的不會超過 8 個點！而那些因為 x 值被刪掉的反正全部的過程中只會被刪一次。為什麼是 8 點呢？直覺上來看，因為左邊的點之間距離都不小於 d，而我們會納入計算的點都落在寬度為 d 高度 2d 的矩形內，這樣的點數不可能太多，如果是直

線距離，最多 6 點，如果是 L_1 距離，最多是 8 點，這並不特別困難，我們留給讀者自己想一想，網路上也可以找到說明。

4.3. 其他習題

Q-4-16. 賺錢與罰款

泰山派的磨劍坊接了 n 筆訂單，每一筆訂單有需要的工時 $t[i]$ 以及完工要求時間 $d[i]$ ，如果在時間 x 的時後交貨就可以賺 $d[i]-x$ 的錢，也就是說越早完成賺越多，超過時間完成的話，越晚賠越多。磨劍坊每次只能做一件工作，所以要把這 n 筆訂單做一個排程，希望利潤最大，也就是賺最多錢，如果不可能賺錢就要賠最少。泰山派掌門天門道長聽到之後，深怕會賠太多的錢而破產，請你幫忙找出最好的排程。

Time limit: 1 秒

輸入格式：輸入的第一行是工作數 N 。第二行有 N 個正整數，依序是各訂單所需時間 $t[1]$ 、 $t[2]$ 、...、 $t[N]$ 。第三行有 N 個非負整數，依序是各訂單的完工要求 $d[1]$ 、 $d[2]$ 、...、 $d[N]$ ，相鄰以空白間隔。 $N < 1e5$ ，時間不超過 1000，完工要求不超過 $1e8$ 。

輸出：最大利益。

範例輸入：

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

範例結果：

```
-16
```

Q-4-17. 死線高手

華山派每個地子都有很多作業，每個作業都有死線 (dead-line)，必須在死線之前完成否則會受到處罰。令狐沖現在有 n 個作業，每個作業需要花的時間是 $t[i]$ 而死線是 $d[i]$ ，此外，每次只能進行一個作業，不可能一次做兩個作業。如果有任何一個作業超過死線，就會被罰到華山之巔面壁一年，請問他是否可能安排作業的順序，讓每個作業的完成時間都不會超過死線，否則小師妹就可能會移情別戀了。

Time limit: 2 秒

輸入格式：輸入包括多筆測資，第一行是測資筆數 T ， $T < 20$ ，以下是 T 筆測資的資料。每筆測資的第一行是作業數 n 。第二行有 n 個正整數，依序是各作業所需時間 $t[1]$ 、 $t[2]$ 、...、 $t[N]$ 。第三行有 n 個正整數，依序是各作業的死線 $d[1]$ 、 $d[2]$ 、...、 $d[N]$ ，相鄰以空白間隔。 $n < 1e5$ ，時間不超過 1000，死線不超過 $1e8$ 。

輸出：依序輸出每筆測資是否所有作業都可以在死線前完成，是則輸出 `yes`，否則輸出 `no`。

範例輸入：

```
2
5
2 1 3 1 2
6 6 3 8 9
4
2 1 2 1
3 5 2 6
```

範例結果：

```
yes
no
```

下一題跟 Q-4-8 有關係，是他的變化題。

Q-4-18. 少林寺的櫃姐 (CC) (*)

少林寺是觀光勝地，每天要接待很多客人，根據數據分析，方丈決定即使在尖峰時期也必須在時間 D 內完成 n 個客人的服務，現在的問題是不知道要開設多少個服務櫃台。假設客人依編號順序到達，第 i 個客人需要 $t(i)$ 的時間來完成他的需求（無論分配到哪一個櫃台），而且每個客人的需求必須在同一個櫃台完成。因為公平的因素，先到客人的服務開始時間不可以大於晚到客人的服務開始時間。因為每開設一個櫃台就要請一位櫃姐，為了節省人事成本，請計算出最少要開設多少櫃台才能讓這 n 個客人的服務都在時間 D 內完成。

Time limit: 1 秒

輸入格式：輸入兩行，第一行是正整數 n 與 D ，第二行是 n 個正整數 $t(1)$ ， $t(2)$ ，...， $t(n)$ ，同行數字間以空白間格。 n 不超過 $1e5$ ， $t(i)$ 不超過 $1e4$ ， D 不超過 $1e9$ 也不小於 $1e4$ ，所以一定有解。

輸出：最少的櫃檯數。

範例輸入：

```
5 9
3 2 5 7 3
```

範例結果：

```
3
```

Q-4-19. 五嶽盟主的會議場所

武林中一共有 n 個門派，每個門派都要上嵩山去見五嶽劍派盟主左冷禪，每個門派的人數以及到達與停留的時間不盡相同，第 i 個門派有 $m(i)$ 個人要去嵩山，到達時間是 $s(i)$ ，而到達後會一直停留到時間 $t(i)$ ，也就是在嵩山的時間是閉區間 $[s(i), t(i)]$ 。左冷禪需要知道最多會有多少人同時在嵩山，以便準備夠大的會議場所，請計算最多在嵩山的人數。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，接著的 n 行每一行有三個整數，依序是 $m(i)$ 、 $s(i)$ 與 $t(i)$ ，代表一個門派的人數以及到達與最後停留時間，兩者之間以一個空格區隔。 n 不超過 $1e5$ ，各派人數不超過 $1e4$ ， $0 \leq s(i) < t(i) \leq 1e9$

輸出：依序輸出最多同時在嵩山的人數。

範例輸入：

```
3
5 1 5
2 5 7
4 8 9
```

範例結果：

```
7
```

Q-4-20. 監看華山練功場

華山派有 n 個弟子，每個弟子的練功時間都不盡相同，第 i 個弟子到練功場所練功的時間是區間 $[s(i), t(i)]$ 。最近華山頗不平靜，掌門岳不群要求令狐沖找一些弟子練功時順便監看練功場，對於想要監看的時間區間 $[x, y]$ ，請問他最少只要找幾位弟子，這些弟子的練功時間就可以涵蓋整個 $[x, y]$ 。

Time limit: 1 秒

輸入格式：第一行是個正整數 n ，第二行是兩個整數 x 與 y ，接著的 n 行每一行有兩個整數 $s(i)$ 與 $t(i)$ ，同行相鄰兩數之間空白區隔。 n 不超過 $1e5$ ， $0 \leq x < y \leq 1e9$ ，且對所有 i ， $0 \leq s(i) < t(i) \leq 1e9$ 。

輸出：練功時間可以涵蓋 $[x, y]$ 的最少的弟子數。如果無解輸出 -1 。

<p>範例一：輸入</p> <pre>5 1 10 0 3 1 5 5 7 8 9 6 10</pre>	<p>範例一：正確輸出</p> <pre>3</pre>
<p>範例二：輸入</p> <pre>5 1 10 0 3 1 5 5 7 8 9 8 10</pre>	<p>範例二：正確輸出</p> <pre>-1</pre>

5. 分治演算法

分治：「一刀均分左右，兩邊各自遞迴，返回合併之日，解答浮現之時。」

「架勢起得好，螳螂鬥勝老母雞。在分治的架構下，簡單的資料結構與算法往往也有好的複雜度。」

這一章介紹分治演算法 (Divide-and-Conquer algorithm, DaC)，又稱分而治之演算法，也有人稱為各個擊破法。分治是一種非常重要的演算法思維模式與策略，有很多重要的演算法都是根據分治的思維模式，例如快速排序法、合併排序法、快速傅立葉轉換 (FFT)、矩陣乘法、整數乘法以及在一些在計算幾何的知名演算法都是分治的策略。相較於它的重要性，分治出現在題目中的比例卻不是那麼高，主要原因有二：第一，很多分治算法的問題都有別的解法的版本，樹狀圖的分治也往往歸類到動態規劃；第二，一些重要的分治算法已經被納入在庫存函數中，例如排序，還有一些則因為太難而不適合出題。即使如此，分治還是很重要必須學習的思維模式，當我們碰到一個不曾見過的問題，分治往往是第一個思考的重點，因為分治的架構讓我們很容易找到比暴力或天真做法要好的方法。

5.1. 基本原理

從名稱 `divide-and-conquer` 就可以看出來，分治算法的精神就是將問題切割後再克服，事實上，分治的算法都可以分成三個主要步驟：

1. 切割。將問題切割成若干個子問題，通常是均勻地切成兩個。
2. 分別對子問題求解。
3. 合併。將子問題的解合併成原問題的解。

這裡所謂的子問題切割，並不是將問題分成不同的問題，而是「相同問題比較小的輸入資料」，也因為如此，第二步驟對子問題分別求解其實是透過遞迴呼叫來解子問題，也就是說，除了終端條件外，第二步驟什麼都不必做。這裡就看出分治算法的迷人之處，在思考一個問題的解的時候，你不需要去想解的步驟，只要去想「如何將子問題的解合併成整個問題的解」就可以了。如果我們要計算的問題是一個函數 f ，而輸入資料是 S 。以下是一個典型的分治算法：

演算法 $f(S)$ ：

如果 S 已經很小，達到遞迴終端條件，則直接計算後回傳 $f(S)$ ，結束；

將 S 切割成 S_1 與 S_2 ；

遞迴呼叫計算 $f(S_1)$ 與 $f(S_2)$ ；

將 $f(S_1)$ 與 $f(S_2)$ 合併成 $f(S)$ 後回傳；

結束；

我們用簡單的例子來做具體的說明。

P-5-1. 最大值與最小值

在一個長度為 n 的整數序列中找出最大值與最小值。

(本題只為了說明，太簡單所以沒附測資)

這一題太簡單了，我們只是要用他來說明分治算法的精神。直接的做法：從頭到尾掃描一遍，記住目前看到的最大值，每走一步就嘗試更新最大值，這樣可以找出最大值；相同的方法再做一次找最小值。時間複雜度 $O(n)$ ，也不可能更好了，因為每個資料一定要被檢查過才可能確保答案的正確性。

我們來試試看分治的想法。假設我們要計算陣列在區間 $[left, right-1]$ 中的最大與最小值，我們從中點把區間分成左右兩個區間 $[left, m-1]$ 與 $[m, right-1]$ ，對兩邊分別遞迴求解；左邊的最小與右邊的最小取比較小的就是全部的最小；相同的，左邊的最大與右邊的最大取比較大的就是全部的最大。遞迴的終結條件呢？如果區間裡面只剩一個數字，最小最大都是它。以下是這樣的想法實作出來的程式，為了讓大家多習慣結構，我們故意在這題用了 `struct`。副程式 `f()` 是一個遞迴函式，其作用在找出 `A` 陣列在 $[s, t)$ 區間的最大與最小值，請注意我們用了 C++ 習慣的左閉右開的區間，程式中先檢查遞迴終端條件，這裡設的是區間剩下一個元素，接著計算中點 `m`，然後對左右兩邊分別遞迴呼叫求解，最後合併解回傳。

```
#include <bits/stdc++.h>
using namespace std;
struct P {
    int pmin, pmax;
};
// range [s, t)
P f(int A[], int s, int t) {
    if (s+1==t) return {A[s],A[s]};
    int m=(s+t)/2;
```

```

    P le=f(A,s,m), ri=f(A,m,t);
    return {min(le.pmin, ri.pmin), max(le.pmax, ri.pmax)};
}

int main() {
    int a[10]={3, 0, 7, -3, -10, 5, 1, 7, 6, -2};
    P x=f(a,0,10);
    printf("%d\n%d\n",x.pmin, x.pmax);
    return 0;
}

```

再次強調分治的重點只在如何合併解答，其它步驟都是非常簡單的，也因為只需要思考合併，分治的正確性往往是比較容易證明的。還有一個好處是複雜度往往很容易計算（當然是指一般情形下），在進行下個例子前，我們先來說明分治演算法的複雜度如何計算。

分治的複雜度

分治是一個遞迴的演算法，不像迴圈的複雜度可以用加總的方法或是乘法計算，遞迴的複雜度是由遞迴關係式 (recurrence relation) 所表達。計算複雜度必須解遞迴關係式。遞迴關係又稱為差分方程式 (difference equation)，解遞迴關係是個複雜的事情，超過本教材要講的範圍（也超過中學生的國際資訊奧林匹亞競賽的範圍），所以在這裡我們只說明一些常見的狀況。

分治演算法的常見形式是將一個大小為 n 的問題切成 a 個大小為 b 的子問題（相同問題較小的輸入資料），此外必須做一些分割與合併的工作。假設大小為 n 的問題的複雜度是 $T(n)$ ，而分割合併需要的時間是 $f(n)$ ，我們可以得到以下遞迴關係：

$T(n) = a \times T(n/b) + f(n)$ ，這裡我們省略不整除的處理，因為只計算 big-O，通常不會造成影響。這個遞迴式子幾乎是絕大部分分治時間複雜度的共同形式，所以也有人稱為 divide-and-conquer recurrences，對於絕大部分會碰到的狀況 ($a, b, f(n)$)，這個遞迴式都有公式解，有興趣的請上網查詢 Master theorem。以下我們只說明最常見的幾個狀況的結果，這些遞迴關係的終端條件都是 $T(c) = O(1)$ ，對某常數 c ，意思是當問題的輸入小到某個程度時可以常數時間求解。

遞迴式	時間複雜度	說明
$T(n) = T(n/b) + O(1)$	$O(\log(n))$	切兩塊其中一塊不需要，如二分搜
$T(n) = T(n/b) + O(n)$	$O(n)$	每次資料減少一定比例
$T(n) = 2T(n/2) + O(1)$	$O(n)$	例如找最大值
$T(n) = 2T(n/2) + O(n)$	$O(n \log(n))$	如 merge sort

$T(n) = 2T(n/2) + O(n \log(n))$	$O(n \log^2(n))$	分割合併花 $O(n \log(n))$
$T(n) = 2T(n/2) + O(n^2)$	$O(n^2)$	$f(n)$ 大於 n 的一次方以上，結果皆為 $f(n)$

上面描述的狀況都是子問題的大小減少一定比例，它的複雜度通常都是不錯的結果，某些分治的子問題的只減少常數的大小，這時候的複雜度通常都是差的。例如

$T(n) = T(n-1) + O(n)$ ，結果是 $T(n) = O(n^2)$ ，例如插入排序法。

架勢起得好，螳螂鬥勝老母雞

我們再來看一個例子，分治或許不會讓我們一下子找到效率最好的解，但是如果使用分治，即使合併的方法很天真，往往也很容易找到突破天真算法複雜度的解法。

P-5-2. 最大連續子陣列 (分治) (同 P-4-13)

有一個整數陣列 $A[0:n-1]$ ，請計算 A 的連續子陣列的最大可能總和，空陣列的和以 0 計算。

(檔案中的測資與 P-4-13 相同)

這一題我們在第四章的 P-4-13 看過了，當時我們先說明了天真的 $O(n^2)$ 算法以及超天真的 $O(n^3)$ 算法，最後給了一個 $O(n)$ 的算法，而且這個問題跟 P-4-12 (一次買賣) 是等價的問題。現在假設我們不知道 $O(n)$ 解法，重新以分治的思維來尋找突破 $O(n^2)$ 的方法。

如果我們要計算陣列在 $[L, R-1]$ 區間的最大連續和，首先將區間平均切成兩段 $[L, M-1]$ 與 $[M, R-1]$ ，其中 $M = (L+R) / 2$ 。要找的解 (子陣列) 可能在左邊、右邊、或者跨過兩邊，所以只要三者取最大就是最後的答案。左右可以分別遞迴求解，唯一要做的是就是找出跨過兩邊的最大和，對於左邊，我們就笨笨的從中點往左一直累加，計算每一個可能左端的區間和 (也就是左邊的 `suffix-sum`)，然後取最大；同樣的，對於右邊計算所有 `prefix-sum` 的最大，然後兩者相加就是跨過中點的最大區間和。以下是範例程式，除了分治的概念外，非常的直覺。

```
//P-5-2 max subarray, DaC, O(nlogn)
#include <bits/stdc++.h>
#define N 200020
using namespace std;
typedef long long LL;

// max subarray in [le, ri)
LL subarr(LL a[], int le, int ri) {
```

```

if (le >= ri) return 0;
if (le+1 == ri) return max(a[le], (LL)0);
int mid=(le+ri)/2;
// recursively solve left and right parts
LL largest=max(subarr(a, le, mid), subarr(a, mid, ri));
// find largest sum cross middle
LL lmax=0, rmax=0;
// max suffix sum of the left
for (LL sum=0, i=mid-1; i>=le; i--) {
    sum += a[i];
    lmax=max(lmax, sum);
}
// max prefix sum of the right
for (LL sum=0, i=mid; i<ri; i++) {
    sum += a[i];
    rmax=max(rmax, sum);
}
return max(largest, lmax+rmax);
}

int main() {
    LL n, a[N];
    scanf("%lld", &n);
    for (int i=0; i<n; i++)
        scanf("%lld", &a[i]);
    printf("%lld\n", subarr(a, 0, n));
    return 0;
}

```

事實上除了分治的概念外，找前綴和與後綴和的做法跟原來的 $O(n^2)$ 做法類似，只是原來的做法外面套了一個迴圈（嘗試所有左端），這裡則是外面套了一個分治的遞迴，這麼直覺的做法看起來複雜度要糟糕了，讓我們來算一下複雜度。若 $T(n)$ 是此算法在資料量為 n 時的複雜度，因為分割與合併的處理時間是 $O(n)$ ，所以 $T(n) = 2T(n/2) + O(n)$ ，答案是 $O(n \log(n))$ ！俗話說：架勢起得好，螳螂鬥勝老母雞。分治用的好，天真的合併做法也可以得到不錯的解。如果你去仔細思考其中的原因，你會發現，雖然每一次合併都是笨笨的做，但是任一個資料參與合併的次數只有 $\log(n)$ 次，所以所有資料參與到的總次數只有 $n \log(n)$ 。分治很迷人，雖然這一題有更好的算法。

在結束這一題之前，或許有人想問：這一題用分治是否也可以做到 $O(n)$ 呢？事實上不難，提高算法效率的原則，就是去檢查看看算法中有那些重複的計算，我們留給有興趣的人自己去思考，所附檔案中有一支 $O(n)$ 的範例程式。

5.2. 例題與習題

口渴的烏鴉

看下一個例題前先講一個笑話：有一隻口渴的烏鴉，找到了一個裝有一半水的水瓶，但是瓶口太小喝不到水，於是他飛呀飛的，去到潺潺流水的河邊，找到一些小石頭叼回來放進瓶子裡，烏鴉飛了幾遍，隨著小石頭放入瓶中，瓶中的水位終於慢慢升起，烏鴉很開心的喝水，他欣賞自己聰明的頭腦可以解決這麼複雜的問題。忽然間，他想到他剛才不是去了河邊好幾次嗎？

看下一個例題。

P-5-3. 合併排列法

有一個整數陣列 $A[0:n-1]$ ，請將 A 中的數字從小到大排列。

(本題為示範方法，無測資)

合併排序法是一個利用分治的經典排序法，放這個例題不是要解題，只是讓大家了解他是如何運作，下面是範例程式。函數 `merge_sort()` 負責將陣列的 $[le, ri-1]$ 區間從小排到大，第一步先檢查遞迴終端條件：剩下不到兩個就不必排了。否則計算中間點 m ，遞迴呼叫兩邊；然後重頭戲就是將兩邊已經各自排好序的序列合併成一個，我們啟用一個臨時的空間 `temp[]` 放置合併後的序列，用 j 指著右邊已經處理到的位置，用 k 指著要放入 `temp[]` 的位置。對於每個左邊的元素 $a[i]$ ：先把右邊比 $a[i]$ 小的 $a[j]$ 複製到 `temp[]`，然後再把 $a[i]$ 複製到 `temp[]`。 i 迴圈跑完後，右邊可能有剩下的，但是它們都已經在正確的位置上了，所以不需要處理。最後我們將 `temp[]` 中的內容拷貝回陣列原來位置。

```
// p-5-3 merge_sort
#include <bits/stdc++.h>
using namespace std;

// [le, ri)
void merge_sort(int a[], int le, int ri) {
    if (le+1 >= ri) return; // terminal case
    int m = (le+ri)/2;
    merge_sort(a, le, m); // recursively sort left part
    merge_sort(a, m, ri); // recursively sort right part
    // merge the two sorted lists
    int temp[ri-le], j=m, k=0;
    for (int i=le; i<m; i++) {
        while (j<ri && a[j]<a[i])
            temp[k++] = a[j++];
        temp[k++] = a[i];
    }
    // the remaining of right part are already on position
    // copy back to a
    for (k=le; k<j; k++)
        a[k] = temp[k-le];
}
```



```

    return;
}

int main() {
    int a[10]={3, 0, 7, -3, -10, 5, 1, 7, 6, -2};
    merge_sort(a, 0, 10);
    for (int i=0; i<10; i++)
        printf("%d ",a[i]);
    printf("\n");
    return 0;
}

```

合併排序法的時間複雜度是 $O(n \log(n))$ ，因為合併的部分需要的是 $O(n)$ ，所以複雜度遞迴式為： $T(n) = 2T(n/2) + O(n)$ 。

下一題是分治的經典題。

P-5-4. 反序數量 (APCS201806)

考慮一個數列 $A[1:n]$ 。如果 A 中兩個數字 $A[i]$ 和 $A[j]$ 滿足 $i < j$ 且 $A[i] > A[j]$ ，也就是在前面的比較大，則我們說 $(a[i], a[j])$ 是一個反序對 (inversion)。定義 $W(A)$ 為數列 A 中反序對數量。例如，在數列 $A = (3, 1, 9, 8, 9, 2)$ 中，一共有 $(3, 1)$ 、 $(3, 2)$ 、 $(9, 8)$ 、 $(9, 2)$ 、 $(8, 2)$ 、 $(9, 2)$ 一共 6 個反序對，所以 $W(A) = 6$ 。請注意到序列中有兩個 9 都在 2 之前，因此有兩個 $(9, 2)$ 反序對，也就是說，不同位置的反序對都要計算，不管兩對的內容是否一樣。請撰寫一個程式，計算一個數列 A 的反序數量 $W(A)$ 。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，代表數列長度，第二行有 n 個非負整數，是依序數列內容，數字間以空白隔開。 n 不超過 $1e5$ 數列內容不超過 $1e6$ 。

輸出：輸出反序對數量。

範例輸入：

```

6
3 1 9 8 9 2

```

範例結果：

```

6

```

給定一個數列 A ，計算 $w(A)$ 最簡單的方法是對所有 $1 \leq i < j \leq n$ ，檢查數對 $(A[i], A[j])$ ，但是時間複雜度 $O(n^2)$ 顯然不夠好。以分治思維來考慮。要計算一個區間的反序數量，將區間平分為兩段，一個反序對的兩個數可能都在左邊或都在右邊，否則就是跨在左右兩邊。都在同一邊的可以遞迴解，我們只要會計算跨左右兩邊的就行了，也就是對每一個左邊的元素 x ，要算出右邊比 x 小的有幾個。假設對左邊的每一個，去檢查右邊的每一個，那會花 $O(n^2)$ ，不行，我們聚焦的目標是在不到 $O(n^2)$ 的時間內完成跨左右的反序數量計算。記得第二章學過的排序應用，假設我們將右邊排序，花 $O(n \log(n))$ ，然後，每個左邊的 x 可以用二分搜就可算出右邊有多少個小於 x ，這樣只要花 $O(n \log(n))$ 的時間就可以完成合併，那麼，根據分治複雜度 $T(n) = 2T(n/2) + O(n \log(n))$ ， $T(n)$ 的結果是 $O(n \log^2(n))$ ，成功了！

以下是這樣寫出來的程式，其中第 4~13 行是自己寫二分搜計算一個陣列中比某數小的元素個數，如果喜歡使用 STL 的可以用第 26 行的函數呼叫就可以了。計算反序數的遞迴函數在第 16~29 行，第 18 行檢查終端條件(剩下不超過一個元素的反序數自然是 0)，第 20 行遞迴呼叫左右之後，在第 21 行將右邊排序，然後用一個迴圈對每一個左邊的元素計算右邊有幾個比他小。

```

00 // p_5_4_a nlog^2(n) method
01 #include <bits/stdc++.h>
02 using namespace std;
03 typedef long long LL;
04 // binary search the num <x from a[0] to a[len-1]
05 int bsearch(int a[], int len, int x) {
06     if (a[0] >= x) return 0;
07     int pos=0;
08     for (int jump=len/2; jump>0; jump>>=1) {
09         while (pos+jump<len && a[pos+jump]<x)
10             pos += jump;
11     }
12     return pos+1;
13 }
14
15 // interval= [le,ri), return #inversion
16 LL inv(int ar[], int le, int ri) {
17     int i,j,mid = (ri+le)/2; // middle point
18     if (le+1 >= ri) return 0; // terminal case
19     // recursively solve left and right parts
20     LL w = inv(ar, le, mid) + inv(ar, mid, ri), cross=0;
21     sort(ar+mid, ar+ri); // sort the right part
22     // for each in left, binary search to find #inversion
23     for (i=le; i<mid; i++) {
24         cross += bsearch(ar+mid, ri-mid, ar[i]);
25         // following is using STL binary search
26         // cross += lower_bound(ar+mid, ar+ri, ar[i]) - (ar+mid);
27     }
28     return w + cross;
29 }
30 int main() {

```

```

31     int i, n, ar[100010];
32     scanf("%d", &n);
33     for (i=0; i<n; i++) scanf("%d", ar+i);
34     printf("%lld\n", inv(ar, 0, n));
35     return 0;
36 }
37

```

程式並不難，複雜度也算不錯了，但可不可以精益求精做到 $O(n \log(n))$ 呢？剛才只是單邊排序，所以很多次的二分搜花了 $O(n \log(n))$ 。假設兩邊遞迴呼叫時，回傳都是排好序的，我們就可以不必寫二分搜了，因為連續二分搜不如一路爬過去。所以副程式可以改成下面這樣。

```

// return the inversion in [le, ri) ans sort them
LL inv(int a[], int le, int ri) {
    if (le+1 >= ri) return 0; // terminal case
    int m = (le+ri)/2;
    // recursively solve the left and the right parts
    LL w=inv(a, le, m) + inv(a, m, ri), cross=0;
    // count a[j]<a[i] for i<m<=j
    int j=m;
    for (int i=le; i<m; i++) {
        while (j<ri && a[j]<a[i]) // until >=a[i]
            j++;
        cross += j - m; // num <a[i]
    }
    sort(a+le, a+ri); // sort before return
    return w + cross;
}

```

雖然避掉了二分搜，但是要排序，所以合併階段還是花了 $O(n \log(n))$ ，整個程式的複雜度還是 $O(n \log^2(n))$ 並沒有降下來。

再看看這個程式是不是有點熟悉呢？計算右邊有幾個比 $a[i]$ 小的部分不就幾乎是在合併左右兩個排好序的陣列嗎！確實，這支程式跟 P-5-3 的合併排序法少有 87% 以上相同，有沒有突然想起前面講的那隻口渴的烏鴉？是的，我們剛才在一個很像合併排序法的程式裡面呼叫 `sort()`！就像是烏鴉只想著要找石頭而忘了最終目的是喝水。所以，其實呼叫排序根本是不必要的，我們在合併兩個排好序的序列時，本來就幾乎已經算了反序數了。以下是這樣的寫法，我們只要把合併排序法稍加修改就可以了，時間複雜度也跟 merge sort 一樣是 $O(n \log(n))$ 。

```

// p-5-4 inversion number DaC O(nlogn)

```

```

#include <bits/stdc++.h>
using namespace std;
typedef long long LL;

// return the inversion in [le, ri) and sort the array
LL inv(int a[], int le, int ri) {
    if (le+1 >= ri) return 0; // terminal case
    int m = (le+ri)/2;
    // recursively solve the left and the right parts
    LL w=inv(a, le, m) + inv(a, m, ri), cross=0;
    // merge the two sorted lists
    int temp[ri-le], j=m, k=0;
    for (int i=le; i<m; i++) {
        while (j<ri && a[j]<a[i])
            temp[k++] = a[j++];
        temp[k++] = a[i];
        cross += j - m; // num <a[i]
    }
    // the remaining of right part are already on position
    // copy back to a
    for (k=le; k<j; k++)
        a[k] = temp[k-le];
    return w + cross;
}

int main() {
    int i, n, ar[100010];
    scanf("%d", &n);
    for (i=0; i<n; i++) scanf("%d", ar+i);
    printf("%lld\n", inv(ar, 0, n));
    return 0;
}

```

下一題與第四章的 P-4-15 是一樣的，放在這裡當作習題是希望有興趣的人可以練習分治的寫法，後面會給予提示。

Q-5-5. Closest pair (同 P-4-15, 分治版) (@@)

平面兩點的 L_1 距離為兩點的 x 差值與 y 差值的和，也就說，如果兩點座標是 (a, b) 與 (c, d) ，則 L_1 距離是 $|a-c|+|b-d|$ 。輸入 n 個點的座標，請計算出 L_1 距離最近兩點的 L_1 距離。

Time limit: 1 秒

輸入格式：第一行為一個正整數 n ，接下來 n 行，每行兩個整數 x 與 y 代表一點的座標。 n 不超過 $1e5$ ，座標值絕對值不超過 $1e8$ 。

輸出：最近兩點的 L_1 距離。

範例輸入：

```
4
-1 5
4 0
3 1
-2 -3
```

範例結果：

```
2
```

Q-5-5 提示：將點依照 x 座標排序後，將所有點均分為左右兩部分，以遞迴分別求出左右的最近距離在取最小值，在合併階段的工作是找出來左右各一點的最小距離。假設目前已找到的最小距離是 d ， x 為中值的點座標是 (p, q) ，那麼兩邊的點只有 x 座標在 $[p-d, p+d]$ 區間的點需要計算。如果兩邊的點已經依照 y 座標排序，對左邊的任一點 (x, y) ，右邊的點中只有 y 座標落在 $[y-d, y+d]$ 區間的點需要計算與 (x, y) 的距離。在 P-4-15 掃描線算法時，我們必須使用一個動態的資料結構在點逐步加入時來維持 y 座標排序，但是使用分治時，利用 merge sort 類似的方法，只需要用簡單的陣列就可以做到了，時間複雜度一樣是 $O(n \log(n))$ 。

下一題也是見過的，之前出現在第三章的習題，這裡當作例題說明分治的寫法。

P-5-6. 線性函數 (同 Q-3-14, 分治版)

有 N 線性函數 $f_i(x) = a_i x + b_i, 1 \leq i \leq N$ 。定義 $F(x) = \max_i f_i(x)$ 。輸入 $c[i], 1 \leq i \leq m$ ，請計算 $\sum_{i=1}^m F(c[i])$ 。

Time limit: 1 秒

輸入格式：第一行是 N 與 m 。接下來有 N 行，依序每行兩個整數 a_i 與 b_i ，最後一行有 m 個整數 $c[1], c[2], \dots, c[m]$ 。每一行的相鄰數字間以空白隔開。 $N \leq 1e5$ ， $m \leq 5e4$ ，輸入整數絕對值不超過 $1e7$ ，答案不超過 $1e15$ 。

輸出：計算結果。

範例輸入：

```
4 5
-1 0
1 0
-2 -3
2 -3
4 -5 -1 0 2
```

範例結果：

每一個線性函數對應到平面上一條直線，這一題需要一點幾何特性，簡單的說，越往右邊，斜率大的函數值 (Y 值) 會越大。因此，假設我們先將直線函數依照斜率由小到大排序，對於某一點 $x = c$ 時，先找出在此點的最大函數值是某個 f_i 。那麼，在 $x > c$ 時，我們不需要計算斜率比 f_i 小的函數值；同時，在 $x < c$ ，不需要斜率比 f_i 大的函數值。為了分治的效率，我們每次抓 x 的中值，計算出哪一條直線在此的函數值最大 (笨笨的算就可以)，然後遞迴左右，以下是範例程式。

為了排序，我們定義了一個結構來存每一條直線，也定義了一個比較函數：先比斜率，斜率相同比截距，當然也可以用 STL 的 `pair` 來寫而略過比較函數。程式重頭戲就是遞迴函數 `dc(l1, r1, l2, r2)`，這個遞迴函數計算點 `point[l2]~point[r2-1]` 在直線 `line[l1]~line[r1-1]` 的最大值總和，做法依照上一段的說明，其實很簡單。

```
// P-5-6. Let F be the maximum of a set of linear function.
// Find F(c[1]) + F(c[2]) + ... F(c[m]), divide and conquer
#include <bits/stdc++.h>
using namespace std;
#define N 100010
using namespace std;
typedef long long LL;
struct Line {
    int a, b; // y=ax+b
} line[N];
// compare by slope first
bool cmp(Line p, Line q) {
    if (p.a == q.a)
        return p.b < q.b;
    return p.a < q.a;
}
int point[N];

// for line [l1,r1), point [l2,r2)
LL dc(int l1,int r1,int l2,int r2) {
    if (l2 >= r2) return 0; // no points
    int mid = (l2+r2)>>1;
    // find max y of the middle point
    LL max_y = LLONG_MIN, who;
    for (int i=l1; i<r1; i++) {
        LL y=(LL)point[mid]*line[i].a+(LL)line[i].b;
        if (max_y < y) {
            max_y = y;
            who = i;
        }
    }
    // left side need only smaller slope, right side larger slope
    return max_y + dc(l1,who+1,l2,mid) + dc(who,r1,mid+1,r2);
}
```

```

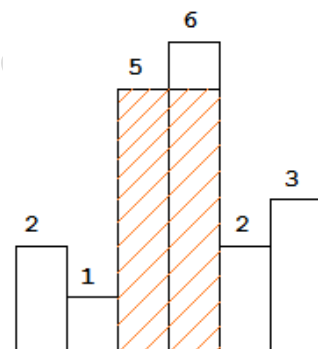
int main() {
    int i,m,n;
    scanf("%d%d",&n,&m);
    for (i=0;i<n;i++) {
        scanf("%d%d",&line[i].a,&line[i].b);
    }
    for (i=0;i<m;i++) scanf("%d",point+i);
    sort(line, line+n, cmp);
    sort(point, point+m);
    printf("%lld\n",dc(0,n,0,m));
    return 0;
}

```

下一題是網路上有很多解法的例題。

P-5-7. 大樓外牆廣告 (分治版)

有一條長街林立著許多大樓，每棟樓高高低低不一定，但寬度都是相同。現在想在牆面上找一塊面積最大的矩形來作外牆廣告，此矩形的一對邊必須平行地面，假設每一棟樓的寬度都是 1 單位。以右圖為例，有六棟樓，高度依序為 (2, 1, 5, 6, 2, 3)，最大矩形如圖中標示的部分，面積為 10。



Time limit: 1 秒

輸入格式：第一行 n ，代表有 n 棟樓，第二行有 n 個非負整數，依序代表從左到右每棟樓的高度。 n 不超過 $1e5$ ，樓高不超過 $1e8$ 。

輸出：最大矩形的面積。

範例輸入：

```

6
2 1 5 6 2 3

```

範例結果：

```

10

```

天真的解法是嘗試所有左右端區間 $[i, j]$ ，對於每個區間最大可能的高度是在此區間的最小樓高，如果對每一個區間都跑一個迴圈找最小高度，需要花 $O(n^3)$ ，那也天真的太過分了。類似 maximum subarray，我們可以對任一個固定左端 i ，一個迴圈跑完所

有可能的右端，過程中不斷的更新 prefix-minimum， $O(n^2)$ 就行了。這個解法太簡單了，所以也不需要放上版面來。我們來想想看分治的思維。

分治的想法當然是每次要算某個區間的解，問題是如何分割以及如何合併。假設第 i 棟大樓的高度 $h[i]$ 是全部高度中最低的，那麼，能跨 i 左右的最大矩形，他的水平位置是整個區間，高度就是 $h[i]$ 。既然其他的矩形都不可能跨過 i 的左右，我們就可以將區間切割成左右兩塊，分別遞迴求解，在所有找到的解中最大的就是最後的解。以下這樣想法之下的範例程式。

```
// p_5_7 divide and conquer at smallest, O(n^2)
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
#define N 100010
LL h[N];
// largest rectangle in range [s, t)
LL rect(int s, int t) {
    if (s>=t) return 0;
    if (s+1==t) return h[s];
    // find min height
    int m=min_element(h+s, h+t) - h;
    LL max_rect=(t-s)*h[m];
    // recursively solve the left and right
    max_rect=max(max_rect, rect(s,m));
    max_rect=max(max_rect, rect(m+1,t));
    return max_rect;
}

int main() {
    int n;
    scanf("%d",&n);
    for (int i=0;i<n;i++)
        scanf("%lld", &h[i]);
    printf("%lld\n",rect(0,n));
    return 0;
}
```

上面這個分治的時間複雜度是多少？必須看切割的狀況，複雜度遞迴式是：

$$T(n) = T(i) + T(n-i-1) + O(n),$$

如果運氣很好，每次切割左右都不小於一定比例，總時間複雜度會是 $O(n \log(n))$ ，但很不幸的，如果每次的最低點都在邊緣，例如說原輸入高度是遞增或遞減的，時間複雜度會高到 $O(n^2)$ 。能改善嗎？如果每次找最小值都可以很快的查詢出來，那時間複雜度是可以改善的，但必須動用到比較複雜的資料結構來做區間最小值查詢 (RMQ, range minimum query)。

假使我們從一定要降低複雜度的需求來思考。前面見過：架勢起的好，螳螂鬥勝老母雞。分治要有好的效率，分割時最好是平均分割，切成兩塊一樣大的，如此一來，只要合併花的時間小於 $O(n^2)$ ，整體的複雜度就會降下來了。均勻切割成兩塊不是問題，中點一切就行，問題在合併，也就是找出跨中點的最大矩形，我們來試試看。

假設目前要處理的區間範圍是 $[s, t-1]$ ，中點是 m ，那麼跨過中點矩形的高度顯然不能超過 $h[m]$ 。在高度不小於 $h[m]$ 的條件下盡量延伸左右範圍直到不能延伸為止，也就是會找到一個區間 $[i+1, j-1]$ ， $h[i] < h[m]$ 或者已達左邊界，而且 $h[j] < h[m]$ 或者已達右邊界。這樣我們可以求得以高度為 $h[m]$ 的最大矩形。每一次我們嘗試逐步下降高度，然後再往外擴增範圍，因為要逐步下降，所以下一次的高度基本上是 $\max(h[i], h[j])$ ，除非其中之一以超越邊界，那就取另一端。重複以上的步驟，我們會嘗試過區間內所有可能的最高高度，找到最大的矩形。程式碼其實不難，但要小心邊界。

在擴增範圍的過程中， i 只會由中點往左邊跑，而 j 只會從中點往右邊跑，所以合併的步驟只會花線性時間，因此整體的複雜度是 $O(n \log(n))$ 。因此，利用分治的架構，我們寫了一個不複雜的合併步驟，就達到不錯的時間複雜度。

```
// p_5_7 divide and conquer at middle, O(nlogn)
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
#define N 100010
LL h[N];
// largest rectangle in range [s, t)
LL rect(int s, int t) {
    if (s>=t) return 0;
    if (s+1==t) return h[s];
    int m=(s+t)/2;
    // recursively solve the left and right
    LL max_rect=max(rect(s,m),rect(m+1,t));
    // merge step, find rectangle cross middle
    LL i=m, j=m, height=h[m], largest=0;
    // each time, find maximal [i+1,j-1] of given height
    while (i>=s || j<t) {
        // choosing smaller as the next height
        if (i<s) height=h[j];
        else if (j>=t) height=h[i];
        else height=max(h[i],h[j]);
        // extend the boundary
        while (i>=s && h[i]>=height) // left boundary
            i--;
        while (j<t && h[j]>=height) // right boundary
            j++;
        largest =max(largest, (j-i-1)*height);
    }
    return max(max_rect, largest);
}
```

```

int main() {
    int n;
    scanf("%d",&n);
    for (int i=0;i<n;i++)
        scanf("%lld", &h[i]);
    printf("%lld\n",rect(0,n));
    return 0;
}

```

P-5-7 存在更好的解法。分治程式的合併階段考慮的是左右邊各一個區段，如果你把他想成左邊的一個區段與右邊的一點如何合併解答，就會是掃描線演算法，範例程式中有這樣的解法，但我們這裡只要是說明分治，所以就留給有興趣的人自己去思考一下。

下一個習題是在第三章 Q-3-12 出現過的完美彩帶，當時使用的技巧是滑動視窗，這一題也可以使用分治法，合併階段要找跨過中點的解，只要能在 $O(n)$ 完成合併階段，整體複雜度就可以做到 $O(n \log(n))$ 。

Q-5-8. 完美彩帶 (同 Q-3-12, 分治版) (APCS201906)

有一條細長的彩帶，總共有 m 種不同的顏色，彩帶區分成 n 格，每一格的長度都是 1，每一格都有一個顏色，相鄰可能同色。長度為 m 的連續區段且各種顏色都各出現一次，則稱為「完美彩帶」。請找出總共有多少段可能的完美彩帶。請注意，兩段完美彩帶之間可能重疊。

Time limit: 1 秒

輸入格式：第一行為整數 m 和 n ，滿足 $2 \leq m \leq n \leq 2 \times 10^5$ ；第二行有 n 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號是不超過 10^9 的非負整數，每一筆測試資料的顏色數量必定恰好為 m 。

輸出：有多少段完美彩帶。

範例輸入：

```

4 10
1 4 1 7 6 4 4 6 1 7

```

範例結果：

```

3

```

說明：區間 $[2, 5]$ 是一段完美彩帶，因為顏色 4、1、7、6 剛好各出現一次，此外，區間 $[3, 6]$ 與 $[7, 10]$ 也都是完美彩帶，所以總共有三段可能的完美彩帶。

5.3. 補充說明

正如我們在本章開頭所說的，很多分治演算法所解的問題都有其他版本的解法，本章提出的例題與習題其實都存在別的解法，甚至別的解法的複雜度還更好。其實，從基本的道理上看，很多分治的做法可以轉換成掃描線演算法並不意外。分治法的重點在合併，也就是跨過左邊與右邊的解；而掃描線演算法是一點一點的逐步加入右邊的資料。

如果仔細去看合併排序法與插入排序法，我們可以這麼說：其實合併排序法就是一次處理一整批的插入排序法。但分治的好處也在這裡，因為是整批的合併，所以每個成員參與合併的次數不會太多(典型是 $\log(n)$ 次)，所以合併時即使不使用什麼高超的技巧或複雜的資料結構，最後還是可以得到不錯的整體複雜度。反觀掃描線演算法，因為一次加入一個資料，所以前面處理的某些結構或部分結果不可以丟棄，因此通常需要有個資料結構來幫忙維護，尤有甚者，這個資料結構往往必須是動態的，因為結構會不斷的變動。也就是說，對於一些題目，如果用分治法，搭配簡單的陣列可能就可以做，但是如果用掃描線演算法就必須配備 STL 中的資料結構，甚至是 STL 中沒有的資料結構。

具體舉例來說，對於 Q-5-5(closest pair)，如果以掃描線算法來作，必須保持已經掃過的點以 Y 值排序，我們必須使用一個動態資料結構(如 STL 中的 `multimap`)，但是分治法就不需要了。以考試或比賽範圍來說，`multimap` 可能不在考試範圍，但是使用分治法並不需要，所以還是可以考。另外一個例子更“有趣”，那是 P-5-4 的反序數量，我們在這一章中已經看到了，將分治法的合併演算法稍加修改就可以解，如果用掃描線演算法，每次碰到 `a[i]`，必須找到前面有幾個數字大於 `a[i]`，很不幸的，STL 中並沒有一個資料結構可以很簡單做到這事情，所以很多程式競賽的選手幾乎都是用線段樹(segment tree)來做這件事。也就是說，如果用分治計算反序數只需要簡單的程式與陣列，但是用掃描線算法則必須裝備一個“頗為複雜”的資料結構(線段樹)。結論是：分治是一個很重要的演算法概念與思維模式。

最後藉著線段樹這個話題，附帶提一下課本上念資料結構與程式競賽的資料結構不一樣的地方。STL 中的 `set` 基本上是個平衡的 BST(binary search tree)，如果這棵 BST 是自己種的，想要在 BST 上查詢比某個數字小的有多少，那只要簡單的修改一下 BST 的結構就可以簡單的做到，但是 STL 雖然有 `BST(set)`，但是沒辦法做這件事，所以練競賽的選手需要學線段樹，當然線段樹的用途很廣同。其實線段樹也就是一種可以偷懶的 BST，因為在競賽的場合，不太可能有時間去自己寫 BST，所以可以用 STL 中的就用，不能用的就會找出替代手法(或是說偷懶手法)，這樣的狀況在競賽程式中屢見不鮮，另外一個很有名的手法就是在優先佇列中降 key 的處理(decreasing key in a priority queue)，不過這些都超過本教材的範圍，僅此做一些額外的說明給有興趣的人參考。

6. 動態規劃

「每一個 DP 背後都有一個 dag, DP 需要 dag 指引方向, 如同視障者需要 dog。」
(Dynamic programming, Bottom-up)

「設計 DP 以尋找遞迴式開始, 以避免遞迴來完成, 除非, 除非你準備小抄。」
(Dynamic programming, Top-down memoization)

這一章介紹動態規劃(Dynamic Programming, DP)。DP 名字中 Programming 的意思是指「以表格紀錄」, 而非現在常用的「寫程式」, 所以 DP 的意思是以變動的表格來求解的方法。DP 的應用很廣, 變化很多, 在學術界也發展的很早, 在程式競賽也用的非常多, 並且有很多困難而精妙的(優化)方法。這本教材只限於一些常用的典型的題型, 此外樹與圖的 DP 將放在後續討論樹的專章中, 而不在本章中。

6.1. 基本原理

6.1.1. 基本思維與步驟

DP 與分治有個相同之處, 都是將問題劃分成子問題, 再由子問題的解合併成最後的解, 其中子問題是指相同問題比較小的輸入資料。所以, 設計 DP 的方法從找出遞迴式開始, 設計 DP 的演算法通常包含下面幾個步驟:

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

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

- 問題一: 小朋友上樓梯, 每步走一階或兩階, 如果走到第 n 階有 $f(n)$ 種走法, 輸入 n , 計算 $f(n)$ 。例如 $n=3$ 時, 走法有 $1+1+1=3$ 或 $1+2=3$ 或 $2+1=3$, 一共 3 種, $f(n)=3$ 。

我們的問題是計算 $f(n)$ 。定義子問題: 對 $i < n$, $f(i)$ 表示走到第 i 階的走法數。接著要找出 $f(n)$ 與子問題的遞迴關係, 我們雖然沒辦法立刻寫出 $f(n)$ 是多少, 但是我們只

要去想，走到第 n 階的最後一步一定是走了一階或兩階（沒有其他的可能），而最後一步是一階與兩階的走法必然不同（沒有重複多算）。最後走一階的走法數就是抵達第 $n-1$ 的走法數 $f(n-1)$ ，而最後一步走兩階的走法數就是抵達第 $n-2$ 階的走法數 $f(n-2)$ ，因此，我們得到遞迴關係式 $f(n) = f(n-1) + f(n-2)$ ，但是不要忘了以上推論是對於 $n > 2$ 時的狀況，遞迴的初始條件 $f(1)$ 與 $f(2)$ 要另外找，不難想一下就可以知道，所以完整的遞迴關係是：

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

如果直接依照遞迴寫程式，以下是這樣簡單的程式。

```
//ch6 stair, recursion
#include <stdio>
long long stair(int n) {
    if (n<3) return n;
    return stair(n-1)+stair(n-2);
}
int main() {
    int n;
    scanf("%d", &n);
    printf("%lld\n", stair(n));
    return 0;
}
```

這個程式可以跑，只要你不要輸入太大的數字。如果輸入 40 大概會看到螢幕停頓一下就跑出來，輸入 60 的話肯定可以去喝杯咖啡再看看跑出來沒；如果輸入 100，跑到你死了也跑不出來；如果輸入 200，那到地球毀滅也跑不出來。原因何在？這是一個純遞迴的程式，一個遞迴呼叫兩個，兩個就會呼叫四個，雖然 $f(n-1)$ 與 $f(n-2)$ 並不一樣，但它會接近 2 的 n 次方，確切的時間複雜度大概是 $O(1.6^n)$ 。

我們來把它改成 DP，所以要進行第三步驟：找出子問題的計算順序來避免以遞迴的方式進行計算。什麼樣的計算順序會讓我們可以不需要遞迴呢？原則上「**只要讓遞迴式等號右邊出現的在左邊的之前先算，並且用表格把算過的記下來，就可以不需要遞迴了。**」以這一題為例，遞迴式 $f(n) = f(n-1) + f(n-2)$ ，如果 n 從小往大算，算到 $f(n)$ 的時候， $f(n-1)$ 與 $f(n-2)$ 都已經算過了，所以只要當初算完的時候有記下來，那就可以直接從表格中取出 $f(n-1)$ 與 $f(n-2)$ 來相加得到 $f(n)$ ，完全不需要用到遞迴。那麼如何用表格紀錄呢？這一題我們可以開一個陣列 $F[]$ ，以 $F[i]$ 記錄 $f(i)$ 的值，程式就變成一個簡單的迴圈，沿著我們的計算順序 (n 從小到大)，逐步從表格中取出值，計算後存入表格，程式如下：

```

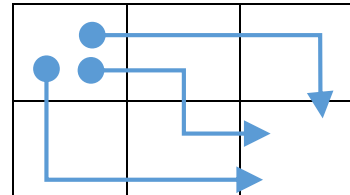
//ch6 stair, dp
#include <cstdio>
int main() {
    long long F[10000];
    int n;
    scanf("%d", &n);
    F[1]=1, F[2]=2;
    for (int i=3;i<=n;i++)
        F[i]=F[i-1]+F[i-2];
    printf("%lld\n", F[n]);
    return 0;
}

```

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

我們再來看一個例子。

- 問題二：有一個 $m \times n$ 的矩形方格，從左上角的格子出發，要到達右下角的格子，每一步只能向右或向下移動一格，若不同的路徑有 $g(m, n)$ 條，輸入 m 與 n ，計算 $g(m, n)$ 。例如 $g(2, 3)=3$ ，如右圖。



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

$g(i, j) = g(i, j-1) + g(i-1, j)$ for $i > 1$ and $j > 1$ ；初始條件為

$g(i, j) = 1$ for $i = 1$ or $j = 1$ 。得到遞迴式之後直接寫遞迴程式就是純遞迴，程式如下，再一次，遞迴的程式非常沒有效率：

```

//ch6 grid, recursion
#include <cstdio>
long long grid(int m, int n) {
    if (m==1 || n==1) return 1;
}

```

```

    return grid(m, n-1)+grid(m-1, n);
}
int main() {
    int m,n;
    scanf("%d%d", &m, &n);
    printf("%lld\n", grid(m,n));
    return 0;
}

```

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

```

//ch6 grid, DP
#include <cstdio>
int main() {
    long long G[50][50];
    int m,n;
    scanf("%d%d", &m, &n);
    for (int i=1; i<=m; i++)
        G[i][1]=1;
    for (int j=1; j<=n; j++)
        G[1][j]=1;
    for (int i=2; i<=m; i++)
        for (int j=2; j<=n; j++)
            G[i][j] = G[i-1][j] + G[i][j-1];
    printf("%lld\n", G[m][n]);
    return 0;
}

```

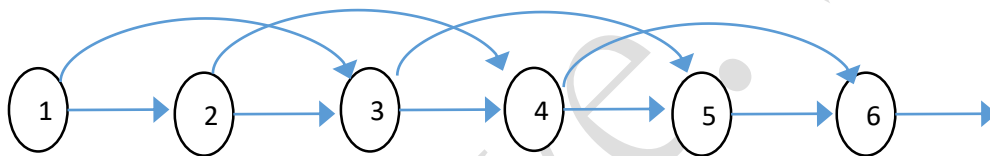
6.1.2. 狀態轉移

DP 子問題的遞迴關係式也有人稱為「狀態轉移」，這個名詞大概是從有限狀態機 (數位電路設計) 來的。子問題可以看成一個狀態，目前的狀態是根據之前的那些狀態，這樣

的轉換關係就稱作狀態轉移，設計 DP 時，最重要的就是找出狀態轉移。之前我們將子問題看成「部分解」，所以是在找部分解與更小的解之間的關係，這兩個講法只是描述的方式不一樣，其實是同一回事，有的時候這樣想比較方便，有的時候那樣想比較直覺。

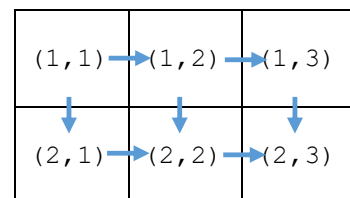
雖然我們要在下一章會討論圖 (graph)，但提到狀態轉移，用圖的術語比較好說明。我們說狀態 A 是狀態 B 的前置狀態 (predecessor)，如果狀態 B 是由 A 轉移過來的，一個狀態可能有多個前置狀態，通常一個狀態的值是由它的前置狀態的值做某些計算得到。如果我們把每個狀態想像每一個點，若 A 是 B 的前置狀態，則由 A 劃一個箭頭指向 B，我們會得到一個所謂的圖 (graph)，這個圖就是這個 DP 的狀態轉換圖，在圖論上來說，它必然是個有方向但沒有環路的圖 (directed acyclic graph, dag)，所謂環路是指從某點出發，沿著箭頭前進，最後又走到出發點。遞迴關係不會無限遞迴，所以必然沒有環路。我們來看看前面兩個簡單的問題的狀態轉移。

小朋友上樓梯問題的子問題是走到第 i 階，關係式 $f(n) = f(n-1) + f(n-2)$ ，所以狀態圖如下，也就是每個狀態由它的前兩個狀態轉移而來。



第二個問題是方格圖的問題，遞迴式是：

$g(m, n) = g(m-1, n) + g(m, n-1)$ ，每一個格子是一個點 (狀態)，前置狀態是他的左方與上方。



對於一個 dag，我們一定可以將它的所有點排成某個順序，在此順序中，每個箭頭都由前往後指，也就是說，每個點的前置點都在它之前，這樣的順序稱為拓樸順序 (topological sort)，我們的 DP 就是要沿著一個拓樸順序來進行。有件事情要提醒，本節說明的狀態轉移以及 dag 與拓樸順序，是方便思考用的，在 DP 解題時通常並不需要把它建出來，也就是建在心中就可以了，不必建在程式中。我們在第三章的 P-3-1 樹的高度的問題時，曾經講過利用 queue 找樹的 bottom-up 順序，那就是個拓樸順序，而那一題確實就是樹的 DP。

6.1.3. 分類與複雜度

題目的分類往往有很多種，主要看目的何在而分類，這裡我們以便於理解與學習來分類。一個 DP 如果狀態有 $O(n^x)$ 而轉移式涉及 $O(n^y)$ 個狀態，一般可稱為 $xDyD$ 的 DP，例如小朋友上樓梯是 $1D0D$ 的，因為有 n 個要算，每次算 $f(i)$ 只涉及 2 個 $f(i-1)$ 與 $f(i-$

2)。方格路徑的問題則是 2D0D，因為有 n^2 (假設 $m=n$) 個要算，每次只需要 2 個 (左方與上方)。當然也有一些 DP 不在這個分類中。

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

6.1.4. Top-down memoization

前面講到設計 DP 算法的步驟，基本上要先找出遞迴式，然後找出計算順序來避免遞迴。這算是標準的 DP，也稱為 Bottom-up 的 DP。另外有一種實現 DP 的方式是不找計算順序，直接依照遞迴式寫遞迴，但是因為一定要避免遞迴的重複計算，所以採取了一些步驟，這種實現的方式稱為 top-down memoization 的 DP，其中 memoization 字典上應該沒有這個字，大概是為了這個技術自己創造的，這個字是由 memo (備忘錄，便籤小紙條) 變來的，這個方法基本上應該是打比賽或重實作的人上發展出來的 (偷懶) 方法，因為從理論的角度，除了程式比較好寫之外，這麼做沒有好處，但考試與比賽還真的就希望比較好寫。

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

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

我們來看看前面介紹的兩個問題的遞迴版本如何改成 top-down DP。下面是第一題，首先注意到它就是拿前面的遞迴版本來改的，修改的部分：第 2 行宣告了一個表格 $F[]$ ；在遞迴函數中第 5 行做了一個檢查是否 $F[n] > 0$ 的動作，如果是則直接回傳，這一題的數字都是正數，所以我們可以用 0 來表示沒算過；如果沒算過，第 6 行就真的遞迴呼叫去計算，然後存在表格中在回傳。

```
00 //ch6 stair, top-down DP
01 #include <cstdio>
```

```

02 long long F[100]={0}; // memo, 0 if not computed
03 long long stair(int n) {
04     // if (n<3) return n; not necessary
05     if (F[n]>0) return F[n]; // check memo
06     F[n] = stair(n-1)+stair(n-2); // record to memo
07     return F[n];
08 }
09 int main() {
10     int n;
11     scanf("%d", &n);
12     F[1]=1, F[2]=2;
13     printf("%lld\n", stair(n));
14     return 0;
15 }

```

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

```

//ch6 grid, top-down DP
#include <cstdio>
long long G[100][100]; // as memo
long long grid(int m, int n) {
    if (m==1 || n==1) return 1;
    if (G[m][n]>=0) return G[m][n]; // check memo
    // record before return
    return G[m][n]=grid(m, n-1)+grid(m-1, n);
}
int main() {
    int m,n;
    scanf("%d%d", &m, &n);
    for (int i=0; i<=m; i++)
        for (int j=0; j<=n; j++)
            G[i][j] = -1; // initial unknown
    printf("%lld\n", grid(m,n));
    return 0;
}

```

Top-down 的 DP 基本上是個偷懶的方法，我們的建議如下：如果計算順序不難找，還是寫迴圈版，不要過分偷懶，因為遞迴還是有成本與限制，尤其某些系統之下（尤其架在視窗系統下），遞迴的深度有一定的限制，超過限制會導致執行階段的錯誤。在計算順序比較不好找或者比較麻煩時，而且系統是允許的狀況下，就採用 top-down 吧！最

常見的情形就是 Tree 的 DP，幾乎參加比賽的選手都寫遞迴的 DFS 版本，因為迴圈版
的要找 bottom-up 順序(如 P-3-1)，比較麻煩。不過這裡還是要建議：兩種都要會，
可以偷懶的時候才偷懶。

6.2. 例題與習題

在本節中我們提出一些動態規劃的經典例題與習題，來讓大家更熟悉以 DP 的思維來設
計演算法的方式，我們依照 1D0D、2D0D、1D1D、以及 2D1D 與其他共分成四個小節，
在下一節中我們在放了少數比較進階的例子。

6.2.1. 1D0D

這一節我們先看一些 1D0D 的 DP。事實上，在上一節的小朋友上樓梯的走法數就是之
前出現過的費氏數列，也就是 1D0D 的一種最簡單形式。用陣列搭配迴圈或甚至使用
Top-down 的方法都可以很快的在 $O(n)$ 求解。我們在第二章 Q-2-5 中曾經提出以矩
陣快速幂來做到 $O(\log(n))$ 計算費氏數列的第 n 項，事實上就是這一類 1D0D 的優化
方式，如果遞迴式改變時，相同的方法也都可以運用。

不過費氏數列那一類基本上沒有輸入資料的遞迴，其實根本存在數學解，而並非訓練演
算法的好題目，大部分的題目是會搭配輸入資料的。我們之前用到過的前綴和
(prefix-sum) 其實也可以算是 1D0D 的 DP，因為遞迴式是 $ps[n]=ps[n-1]+a[n]$ ，
其中資料放在 $a[]$ 而 $ps[n]$ 是前 n 項的和，另外像類似的 prefix-minimum 也都是
類似的題目，這些都太簡單就不列在例題習題中了。第一個我們選的例題是前面用來說
明的小朋友上樓梯，但是是有輸入資料的版本。

P-6-1. 小朋友上樓梯最小成本

小朋友玩上樓梯遊戲，每一步可以往上走一階或兩階，開始位置在第 0 階，從第一階
開始每階都有一個數字，踩在第 i 階，分數就要扣第 i 階的數字，請問走到第 n 階
的最少的扣分是多少。

Time limit: 1 秒

輸入格式：第一行是正整數 n 。第二行有 n 個正整數，依序代表第 1 階開始的數字，
數字間以空白隔開。 $n \leq 1e5$ ，每階的數字不超過 $1e4$ 。

輸出：走到第 n 階的最小總扣分。

範例一輸入： 8 2 1 1 7 3 2 9 2	範例一輸出： 9
範例二輸入： 5 1 2 3 1 5	範例二輸出： 8

範例一說明：最小扣分是 $1+1+3+2+2=9$ 。

範例二說明：最小扣分是 $2+1+5=8$ 。

假設踩在第 i 階的扣分是 $cost[i]$ ，基本的 DP 問題只要寫出遞迴關係就差不多完成了，通常子問題就是從小到大。根據這個原則定義 $c[i]$ 是走到第 i 階的最小扣分，因為最後一步一定是一階或兩階，我們很快可以得到：

$c[i] = cost[i] + \min(c[i-1], c[i-2])$ for $c > 2$ ；初始條件呢？因為扣分沒有負的，我們很簡單可以知道 $c[1]=cost[1]$ 且 $c[2]=cost[2]$ 。

根據找到的遞迴關係，把 $c[i]$ 存放在陣列中，計算的順序就是 i 由小到大計算，簡單的估算知道用 `int` 就夠大，程式如下很簡單，我們不另外開一個陣列 `c[]`，直接將計算結果放在原來輸入的陣列。

```
// P-6-1
#include <bits/stdc++.h>
using namespace std;
#define N 100005
int main() {
  int n;
  int cost[N];
  scanf("%d", &n);
  for (int i=1; i<=n; i++)
    scanf("%d", &cost[i]);
  for (int i=3; i<=n; i++)
    cost[i] += min(cost[i-1], cost[i-2]);
  printf("%d\n", cost[n]);
  return 0;
}
```

這一題跟費氏數列一樣，可以只用 $O(1)$ 的空間，因為我們算 $c[i]$ 只需要 $c[i-1]$ 與 $c[i-2]$ ，所以可以只用兩個變數記住前兩個狀態，以下是這樣的範例程式。我們用了一個小技巧讓前兩個初始設為 0，這樣可以不需要單獨寫程式處理 $n < 3$ 的情形。

```
// P-6-1, O(1) space
#include <bits/stdc++.h>
using namespace std;
#define N 100005
int main() {
    int n, cost, p=0, pp=0; // previous and pre-previous
    scanf("%d", &n);
    for (int i=1; i<=n; i++) {
        scanf("%d", &cost);
        cost += min(p, pp);
        pp = p;
        p = cost;
    }
    printf("%d\n", cost);
    return 0;
}
```

P-6-2. 不連續的表演酬勞

楊鐵心帶著義女穆念慈當街頭的武術表演者，他接到許多的邀約，每天均有一場。每一場表演都可以得到某些金額的報酬，但是武術表演很辛苦，無法連續兩天都進行表演，請你寫一支程式協助他決定應該接受那些表演以得到最大的報酬。

Time limit: 1 秒

輸入格式：第一行是正整數 n 。第二行有 n 個非負整數，依序代表第 1 天開始每天邀約報酬，數字間以空白隔開。 $n \leq 1e5$ ，每天酬勞不超過 10000。

輸出：最大可能獲得的總酬勞。

範例一輸入： 5 1 2 3 1 5	範例一輸出： 9
範例二輸入： 8 2 1 1 7 3 2 9 2	範例二輸出： 18

範例一說明：挑選 $1+3+5=9$ 。

範例二說明：挑選 $2+7+9=18$ 。

把每天可以獲得的酬勞放在一個一維陣列 $p[]$ 中，再一次我們將子問題定義為：
 $dp[i]$ 是前 i 天可以獲得的最大報酬。初始條件很容易想， $dp[0]=p[0]$ ，因為沒有選擇， $dp[1]$ 會是多少呢？因為兩天只能選擇其中之一，所以一定選大的， $dp[1]=\max(p[0], p[1])$ 。要如何建構出遞迴關係式呢？我們可以這麼想：把第 i 天選或不選分開考慮，如果第 i 天不選，前 i 天的獲利就是前 $i-1$ 天的獲利，也就是 $dp[i]=dp[i-1]$ ；如果第 i 天要選呢？我們可以獲得 $p[i]$ ，但是第 $i-1$ 就不可選了，因此最大的獲利是 $p[i]+dp[i-2]$ 。在第 i 天選或不選之間挑最大的，我們得到：
 $dp[i] = \max(dp[i-1], p[i]+dp[i-2])$ 。
 程式就很容易了，讓 i 從小算到大就不需要遞迴了。

```
// max independent on array
#include <bits/stdc++.h>
using namespace std;
#define N 100005
int main() {
    int n, p[N];
    scanf("%d", &n);
    for (int i=0; i<n; i++)
        scanf("%d", &p[i]);
    int dp[N]; // dp[i] = the max until i
    dp[0]=p[0], dp[1]=max(p[0], p[1]);
    for (int i=2; i<n; i++)
        dp[i] = max(dp[i-1], dp[i-2]+p[i]);
    printf("%d\n", dp[n-1]);
    return 0;
}
```

下一題跟上一題看起來似乎有點類似。

P-6-3. 最小監控鄰居的成本

有一條長長的大街被劃分成 n 個區段，編號依序為 $1\sim n$ 。在第 i 個區段設置監控設備的話，需要成本 $c[i]$ ，而可以監控第 $i-1$ 到第 $i+1$ 的區段（超出範圍可忽略）。現在要選一些區段裝設監控設備，以便每一個區段都可以被監控到，請計算最少的總成本。

Time limit: 1 秒

輸入格式：第一行是正整數 n 。第二行有 n 個非負整數，依序代表第 i 個區段裝設監控設備的成本，數字間以空白隔開。 $2 < n \leq 1e5$ ，每處成本不超過 $1e4$ 。

輸出：最小監控總成本。

範例一輸入： 5 1 2 3 1 5	範例一輸出： 2
範例二輸入： 8 2 1 1 7 3 2 9 2	範例二輸出： 6

範例一說明：挑選 $1+1=2$ 。

範例二說明：挑選 $1+1+2+2=6$ 。

在一維陣列 $c[]$ 中有一群非負整數，我們要挑選總和盡量小，但是每個沒有被挑選的點必須左右至少有一點被挑到。看起來與前一題類似，假設我們以類似的方式定義 $dp[i]$ 為前 i 個的最小成本，寫遞迴式的時候會碰到一點困難：如果 $dp[i-1]$ 的值是有挑選到 $i-1$ 的，那麼第 i 個點已經被監控到， $dp[i]=dp[i-1]$ ；但是如果 $dp[i-1]$ 的值來自沒有挑選到 $i-1$ ，那麼第 i 點變成非挑不可，但我們又不知道 $i-1$ 是否已經有挑選。

改變一下子問題的定義，我們讓 $dp[i]$ 是前 i 個點的最小監控成本而且最後一點選在 i 。那麼，成本 $c[i]$ 需要加進來，前面的點雖然不確定那些點一定要挑，但是不可能第 $i-1$ 、 $i-2$ 與 $i-3$ 都沒有挑選，否則 $i-2$ 就沒被監控了，因此得到：

$$dp[i] = c[i] + \min(dp[i-1], dp[i-2], dp[i-3]) \text{ for } i > 2;$$

初始條件是 $dp[0]=c[0]$ ， $dp[1]=c[1]$ ， $dp[2]=c[2]+\min(c[0],c[1])$ 。而最後的解呢？不一定出現在 $dp[n-1]$ ，因為也可能是 $dp[n-2]$ 。程式如下，很簡單就不多做解釋。

```
// min dominating cost on array
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define N 100005
int main() {
    int n, c[N];
    scanf("%d", &n);
```

```

for (int i=0; i<n; i++)
    scanf("%d", &c[i]);
int dp[N]; // dp[i] = the min c until i and include i
dp[0]=c[0], dp[1]=c[1];
dp[2]=c[2]+min(dp[0],dp[1]);
// the last one must be at i-3, i-2, or i-1
for (int i=3; i<n; i++)
    dp[i] = c[i] + min(min(dp[i-3],dp[i-2]),dp[i-1]);
printf("%d\n", min(dp[n-2],dp[n-1]));
return 0;
}

```

如果是剛開始接觸學習 DP 的人要留意，別人跟你說遞迴式很容易懂，但如果你自己碰到題目要列遞迴式的時候，往往列不出來或是列錯，所以在學習的過程不能只看別人怎麼解，而要了解思考的方式。

除了多看題目吸取經驗之外，碰到類似的狀況，通常的解決原則就是把子問題加條件，或是將子問題依照不同的條件再進一步分類。這一題你也可以思考將前 i 個解分成兩類： $dp_0[i]$ 代表第 i 個沒選， $dp_1[i]$ 代表第 i 個有選，分類後會比較容易列式。但是這一題很多人會列錯遞迴式，原因是這題的成本並沒有單調性，並非範圍越大成本越高， $dp_1[i]$ 可能會小於 $dp_1[i-1]$ 。如何列式我們留給有興趣的人自己想一想，範例程式中有另外一支以這樣的方式設計的，如果你認真看一看，你會發現兩個程式做的事情是一樣的。我們要強調一下，達到解的思考方式每個人未必相同，有人覺得這樣想比較簡單，有人認為那樣做比較直覺，其實是要找到適合自己的思考模式。

下一題是個習題。

Q-6-4. 闖關二選一

某個遊戲要依序闖過 n 個關卡，初始的時候有一個幸運數字，每個關卡有兩個關卡數字，你必須把自己的幸運數字調整為兩個關卡數字之一，才能通過此關卡，調整的成本是你的幸運數字與關卡數字的差值（絕對值）。請計算出最低闖關總成本。

舉例來說，一開始的幸運數字為 1， $n=2$ ，第一關的過關數字為 (5, -5)，第二關的過關數字為 (-3, -2)。要依序通過兩關，假設第一關把幸運數字調整成 5，第二關調整為 -2，則需要成本為 $|1-5| + |5-(-2)| = 11$ 。如果，第一關把幸運數字 -5，第二關調整為 -3，則需要成本為 $|1-(-5)| + |(-5)-(-3)| = 8$ 。你可以看得出來，總共有四種方式過關，其中最小成本是 8。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 n 與 t ，代表關卡數以及初始幸運號碼。接下來有 n 行，每行兩個整數，依序代表每一關的過關數字。 $n \leq 1e5$ ，過關數字的絕對值皆不超過 $1e4$ 。

輸出：最小過關成本。

<p>範例一輸入：</p> <pre>2 1 5 -5 -3 -2</pre>	<p>範例一輸出：</p> <pre>8</pre>
<p>範例二輸入：</p> <pre>3 0 -5 1 -8 3 -7 -7</pre>	<p>範例二輸出：</p> <pre>9</pre>

Q-6-4 提示：我們以狀態轉移的方式來思考，有 n 個關卡要通過，每個關卡通過的成本跟上一關通過的方式有關，但是跟更早的無關。所以可以想成通過每一關有兩種狀態，達到第 i 關的第 j 種狀態的最低成本是 $cost[i][j]$ 其中 $i \leq n$ 而 $j=0$ 或 1 。列出相鄰兩關的成本計算方式即可。

Q-6-5. 二維最大子矩陣

輸入一個 $m \times n$ 的二維整數矩陣 $A[1:m][1:n]$ ，要找一塊總和最大的連續子矩陣，輸出其總和。以下圖為例，挑選 $A[1:3][2:3]$ 可以獲得最大總和 13。

2	-2	3	3
-6	5	2	-8
3	7	-2	4

Time limit: 1 秒

輸入格式：第一行有兩個正整數 m 與 n 。接下來 m 行每行 n 個整數，代表矩陣由上而下由左而右的內容。 m 與 n 皆不超過 200，矩陣內的數字絕對值皆不超過 $1e4$ 。

輸出：子矩陣的最大可能總和。

<p>範例一輸入：</p> <pre>3 4 2 -2 3 3 -6 5 2 -8 3 7 -2 4</pre>	<p>範例一輸出：</p> <pre>13</pre>
--	-----------------------------

範例二輸入： 1 6 -2 1 3 -1 4 -5	範例二輸出： 7

這一個習題其實是之前的題目變化成二維的形式，給一個提示：想想看 P-4-13。

6.2.2. 2D0D

1D 的 DP，他們的子問題往往都是以一個整數由小到大編號，2D 的 DP 的子問題通常是由一個二維陣列來編號，第一個要介紹的問題非常簡單，他其實是方格棋盤路徑數問題的有輸入資料版本。

P-6-6. 方格棋盤路線

在一個 $m \times n$ 方格棋盤上，每個格子都有一個分數(可正可負)，現在要從左上角的格子走到右下角，每次只能從當時位置移動到右方或下方的格子，請計算出經過路線上的數字的最大可能總和。以下圖為例，最大的總和路線是經過 (2, -2, 5, 7, -5, 4)，總和為 11。

2	-2	3	3
-6	5	2	-8
4	7	-5	4

Time limit: 1 秒

輸入格式：第一行有兩個正整數 m 與 n 。接下來 m 行，每行 n 個整數，代表方格由上而下，由左而右的內容。 m 與 n 皆不超過 200，矩陣內的數字絕對值皆不超過 $1e4$ 。

輸出：最大總和。

範例一輸入： 3 4 2 -2 3 3 -6 5 2 -8 3 7 -5 4	範例一輸出： 11
範例二輸入： 2 5	範例二輸出： 10

1 2 3 1 -5	
-2 5 -8 2 1	

範例一說明：如題目中說明

範例二說明：先向右走三步，向下一步，再向右一步，得分 $1+2+3+1+2+1=10$ 。

從第一節介紹的狀態轉移就知道，我們可以將到達每一個格子當作一種狀態，因為只能往右或往下移動，所以每一個格子的前置狀態就是他的左方與上方(如果有的話)。因此我們設計子問題的方式就是：以 $f[i][j]$ 當作是走到位置 (i, j) 所能獲得的最大得分，遞迴式為 $f[i][j] = \max(f[i-1][j], f[i][j-1]) + a[i][j]$ ；其中 $a[i][j]$ 是該格子的分數。初始條件呢？因為最上面一行只能從左方走過來，而最左邊一行只能從上方走過來，所以我們可以對這兩個邊界單獨計算，另外一種方式是將超界的最左方與最上方訂為負的無窮大。計算的順序呢？根據原則，遞迴式等號右邊出現的必須在等號左邊出現的先計算，所以每一個格子的左方與上方只要先出現就是可以的順序，最直接的就是由上而下，由左至右。下面是範例程式，我們沒有另外宣告一個陣列放 $f[][]$ ，而是直接把它存在 $a[][]$ 中。時間複雜度很簡單 $O(mn)$ ，因為有這麼多格子要算，每一格只要 $O(1)$ 。

```
// Grid max weight monotonic path
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define N 205
int main() {
    int m,n;
    int a[N][N]={0};
    scanf("%d%d", &m, &n);
    for (int i=0; i<m; i++) for (int j=0; j<n; j++)
        scanf("%d", &a[i][j]);
    // first row and first column
    for (int j=1; j<n; j++)
        a[0][j] += a[0][j-1];
    for (int i=1; i<m; i++)
        a[i][0] += a[i-1][0];
    // max of left and up
    for (int i=1; i<m; i++) for (int j=1; j<n; j++)
        a[i][j] += max(a[i-1][j], a[i][j-1]);
    printf("%d\n", a[m-1][n-1]);
    return 0;
}
```

Longest Common Subsequence (LCS) 是序列分析的重要問題，一個序列的子序列是指將其中某些元素刪除後所得到的序列，字串可以看成字母組成的序列，以“algorithm”為例，“algtm”與“lgh”都是它的子序列，但是“agl”則不是，因為你不可以調整位置重新排列。輸入兩序列，LCS 要找一個最長的序列，它是兩輸入序列的共同子序列。LCS 可以是一種作為字串相似度的定義，有很多重要的應用。

P-6-7. LCS

輸入兩字串，計算其 LCS 的長度。

Time limit: 1 秒

輸入格式：第一行與第二行個有一個字串，字串均只含小寫字母，長度不超過 500。

輸出：LCS 長度。

範例一輸入： algorithm alignment	範例一輸出： 4
範例二輸入： fakefamefade acefaaaace	範例二輸出： 6

範例一說明：algm 或 algt 都是 LCS。

範例二說明：LCS 是 ae faae。

如果要暴力搜尋的話可慘了，一個長度 n 的序列有 2^n 個子序列，因為每一個位置都可以刪除或不刪除。假設第一個字串是 s_1 ，第二個是 s_2 ，以 DP 的思考我們來設法定義子問題，比對兩個字串最普通的方式是從前往後比對，我們定義 $lcs[i][j]$ 為： s_1 的前 i 個字元與 s_2 的前 j 個字元的 LCS 長度，也就是 s_1 與 s_2 的長度分別為 i 與 j 的 prefix 的 LCS 長度。我們分兩個情形：

- 如果 $s_1[i] \neq s_2[j]$ ：既然兩者不相等，兩者中至少有一個對找 LCS 是沒有幫助的，也就是說，我可以刪掉 s_1 的最後一個字母 $s_1[i]$ 或者刪掉 $s_2[j]$ ，因為要找越長越好的，所以兩者取其大，
 $lcs[i][j] = \max(lcs[i-1][j], lcs[i][j-1])$ 。

當然也可能兩者都刪除，因為是一直遞迴上去，兩者都刪的狀態是被包含在刪除其中一個的狀態之中。

- 如果 $s1[i]=s2[j]$ ：因為我們只計算 LCS 的長度，跟在哪個位置對中無關，所以既然對中，不對白不對，把其中一個刪除不會更好，因此可以得到

$$lcs[i][j]=lcs[i-1][j-1]+1。$$

我們已經完成遞迴式的設計了，邊界條件(初始條件)呢？依照我們定義的 $lcs[i][0]$ 與 $lcs[0][j]$ 代表其中一個字串的長度為 0 的狀況，長度為 0 字串與他人的 LCS 當然是空字串，所以 $lcs[i][0]=lcs[0][j]=0$ 。整理一下結果：

$$lcs[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ lcs[i-1][j-1] + 1 & \text{if } s1[i] = s2[j] \\ \max(lcs[i-1][j], lcs[i][j-1]) & \text{otherwise} \end{cases}$$

是不是有點眼熟呢？把 $lcs[][]$ 看成一個表格，每一個格子跟左方、上方與左上方的格子有關，當然還要看 $s1[i]$ 與 $s2[j]$ 是否相等。跟前一題一樣的計算順序就可以滿足 DP 避免遞迴的要求，所以由上而下，由左至右就是一個好的順序。以下是範例程式，時間複雜度是 $O(mn)$ ，因為每格只需要 $O(1)$ 。

```
// LCS, O(mn) space
#include <bits/stdc++.h>
using namespace std;
#define N 505
int main() {
    char s1[N], s2[N];
    int lcs[N][N];
    scanf("%s%s", s1, s2);
    int n=strlen(s1), m=strlen(s2), i, j;
    for (j=0; j<=n; j++) lcs[0][j]=0;
    for (int i=1; i<=m; i++) {
        lcs[i][0]=0;
        for (int j=1; j<=n; j++)
            if (s2[i-1]==s1[j-1])
                lcs[i][j] = lcs[i-1][j-1]+1;
            else
                lcs[i][j]=max(lcs[i-1][j], lcs[i][j-1]);
    }
    printf("%d\n", lcs[m][n]);
    return 0;
}
```

其實很多 DP 的程式都很簡單，遞迴式找出來之後，迴圈與表格從小到大拉過去就結束了。這一題我們多講一個空間上的問題，二維的問題有的時候會有空間用量太大的麻煩。如果兩個字串長度到 1 萬，時間或許還可以接受，但是空間會出現不足的問題。如果真

的需要這麼大的表格，那也是無可奈何，但如果只是要算 LCS 長度，根據上面的遞迴式，我們事實上並不需要一直留著整張表格。計算 $lcs[i][j]$ 時，我們只需要第 i 列與前一列就足夠了，所以有一個直接節省記憶體的方法：宣告一個 $2 \times N$ 的表格，永遠把第 0 列當作前一列，第 1 列當作第 i 列，也就是從第 0 列算到第 1 列，整列算完後將第 1 列複製到第 0 列，然後重複這個步驟。但是這麼做有個缺點就是多了複製的時間，雖然複雜度沒變。改善的方式是用輪流的：奇數回合從第 0 列算到第 1 列，偶數回合從第 1 列算回第 0 列。這個方式有人稱為滾動陣列，寫的方式不只一種，以下是其中一種寫法，交換變數 $from$ 與 to 我們用了 $swap()$ 函數，否則也可以用大家熟悉的交換方式。

```
// LCS, linear space
#include <bits/stdc++.h>
using namespace std;
#define N 505
int main() {
    char s1[N], s2[N];
    scanf("%s%s", s1, s2);
    int n=strlen(s1), m=strlen(s2);
    int lcs[2][N]={0}, from=0, to=1;
    for (int i=0; i<m; i++) {
        for (int j=1; j<=n; j++)
            lcs[to][j] = (s2[i]==s1[j-1])? lcs[from][j-1]+1: \
                max(lcs[to][j-1], lcs[from][j]);
        swap(from, to); // int temp=from; from=to; to=temp;
    }
    printf("%d\n", lcs[from][n]); // final result at [from]
    return 0;
}
```

Alignment 是生物序列 (如 DNA 與蛋白質) 分析的重要問題，輸入兩個字元序列，我們要將兩序列的字母依照原順序做一個對應，過程中可以將任一序列的某些位置加入空白，以輸入 "CTTAAC" 與 "CGGATCAT" 為例，以下是一個可能的 alignment (以 - 表示空白)：

```
CTTAAC-T
CGGATCAT
```

以下也是一個 alignment：

```
C---TTAACT
CGGATCA--T
```

alignment 的目的是要評估兩序列有多相似，我們會有一個評分機制，以下是一個例子：兩字母相同得 8 分，相異 -5 分，字母與空白對應 -3 分。給定評分機制，輸入兩序列，要計算最大的 alignment 分數。Alignment 可分為 global alignment 與

local alignment，前者是輸入的序列整個要納入 alignment，而後者是兩序列各選取連續的一段來做 alignment。

Q-6-8. Local alignment

輸入兩字串，計算 local alignment 的最大分數。評分機制為：兩字母相同得 8 分，相異-5 分，字母與空白對應-3 分。

Time limit: 1 秒

輸入格式：第一行與第二行各有一個字串，字串均只由 ATCG 四個字母組成，長度不超過 500。

輸出：Local alignment 的最大分數。

範例一輸入： ATATCTTAACTGG CGCGGATCATAA	範例一輸出： 43
範例二輸入： AAAACTGAGGG GGCTATT	範例二輸出： 21

範例一說明：第一個字串取 ATCTTAA，第二個取 ATCATAA。

範例二說明：第一個字串取 CTGA，第二個取 CTA。

提示：計算 global alignment 與 LCS 很類似，計算 local alignment 與 global 的差別在於：可以“昨日種種譬如昨日死”，DP 過程中，如果前面的分數不好，可以放棄繼承，此外，最大分數未必出現在最後的位置。

下面一個例題是經典名題 0/1-knapsack，經典的原因是他很多地方都是一個具有代表性的問題。

P-6-9. 大賣場免費大搬家

你抽中了大賣場的周年慶的抽獎活動，在不超過總重量 w 的限制下，你可任意挑選商品免費帶走。現場一共 n 項商品，每項商品有它的重量與價值，每項商品只可以選或不選，不可以拆開只拿一部份。請計算可以獲得的最大價值總和。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 n 與 w 。第二行有 n 個正整數，依序代表商品的重量，第三行有 n 個正整數，依序代表對應 n 項商品的價值。同一行數字間以空白隔開。 $n \leq 100$ ， w 與各商品重量及價值皆不超過 $1e5$ 。

輸出：最大價值總和。

範例一輸入： 7 10 3 4 2 3 3 6 5 5 5 2 4 4 5 6	範例一輸出： 14
範例二輸入： 5 14 1 2 3 4 5 3 2 4 4 4	範例二輸出： 15

範例一說明：選擇第 1, 2, 4 三項，重量為 $3+4+3=10$ ，價值為 $5+5+4=14$ 。

範例二說明：選擇第 1, 3, 4, 5 四項，重量為 $1+3+4+5=13$ ，價值為 $3+4+4+4=15$ 。

假設每個物品的價值與重量恰好成正比，或是更簡單的說，價值等於重量，這個問題就變成在總重量 w 限制下要裝越重越好，這個簡化版的問題稱為 Subset-sum 就是我們下一個習題。其實他更早就出現在我們第一章的 Q-1-8，當時是用來介紹 $O(2^n)$ 的暴搜法，物品數量小於 26。這個題目在沒有其他條件下，並沒有人發明出真正好的演算法，這裡說好的意思是時間複雜度為多項式而不會跑到指數函數。這一題我們把 n 的範圍放到 100，但增加了一個限制條件： w 的範圍不超過 $1e5$ ，也就是說在 w 不大的狀況下，有一個好的算法，這是這一題的經典之一，也是放在此處的主要目的。

以 DP 的思維模式，假設我們將子問題定義成 $dp[i]$ 是考慮前 i 個物品的最佳解，我們在設法列出遞迴式時會遭遇到困難，主要因素是加入一個新的物品時，前面找出來最佳解可能全部改變，例如考慮前 5 項時，最佳解可能是挑第 1 與第 2 件，但是第六項納入考慮時，可能最佳解變成第 6 與第 3, 4 項。這樣找不到子問題最佳解之間的關係，所以是個失敗的定義。前面我們說過一個原則，找不出的時候常用的手法就是加條件，也可以說把解進一步分類。

我們考慮加入第二個條件：重量。令 $d[i][j]$ 是考慮前 i 項物品且重量不超過 j 的最佳解(最大可以獲得價值)，那麼遞迴式就好列了。假設第 i 項物品的重量是 $w[i]$ ，而價值是 $p[i]$ ，我們分成以下情形：

- $w[i] > j$ ：也就是根本不可能挑選，所以第 i 項物品有沒有都一樣， $d[i][j] = d[i-1][j]$ 。
- 第 i 項物品的重量不超過 j ，但是選擇不放：跟前一個情形一樣。
- 挑選第 i 項物品：既然已經挑選了第 i 項，那麼前 $i-1$ 項中挑選的重量不超過 $(j - w[i])$ ，因此， $d[i][j] = p[i] + d[i-1][j-w[i]]$ 。

最後的答案出現在 $d[n][W]$ ，而邊界條件為：沒有物品或重量限制 0 的時候的最大價值都是 0。整理一下結果：

$$d[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ d[i-1][j] & \text{if } w[i] > j \\ \max(d[i-1][j], p[i] + d[i-1][j-w[i]]) & \text{otherwise} \end{cases}$$

還是有點眼熟，把 $d[][]$ 看成一個表格，每一個格子跟上方與左上方的某個格子有關。所以我們依然可以用 i 由小到大， j 由小到大的順序來計算，以下是範例程式，時間複雜度是 $O(nw)$ ，其中 w 是總重限制，它跟之前看到的複雜度只有 n 不一樣，當 w 很大的時候 (例如 $1e20$)，這個程式很糟；但如果 w 不大 (例如本題 $1e5$) 這個程式跑起來要比暴搜好太多了，這個時間複雜度被稱為“偽多項式時間複雜度” (pseudo-polynomial time complexity)。

```

// 01-knapsack, O(Wn) space
#include <bits/stdc++.h>
using namespace std;
#define N 101
#define W 100005
int d[N][W]={0}; // max price of first i items with <= w weight
int main() {
    int n, total;
    scanf("%d%d", &n, &total);
    int w[N], p[N]; // weight and price
    for (int i=1; i<=n; i++)
        scanf("%d", &w[i]);
    for (int i=1; i<=n; i++)
        scanf("%d", &p[i]);
    for (int i=1; i<=n; i++) {
        for (int j=0; j<w[i]; j++)
            d[i][j]=d[i-1][j]; // i-th item cannot be included
        for (int j=w[i]; j<=total; j++) { // choose or not choose
            d[i][j]=max(d[i-1][j-w[i]]+p[i], d[i-1][j]);
        }
    }
    printf("%d\n", d[n][total]);
    return 0;
}

```

以下的習題是背包問題的退化形式，解法類似，留做習題，這題當然可以用前面介紹過的滾動陣列來節省記憶體，其實它甚至可以只用一維陣列，有興趣的人可以試試看。

Q-6-10. 置物櫃出租 (APCS201810)

王老先生有一個置物櫃，共有 M 個相同大小的格子，置物櫃目前已經租給 n 個客戶，第 i 位客戶所租的大小為 $f(i)$ 個格子 ($1 \leq i \leq n$)。目前的承租量總和不超過 M ，但是現在王老先生自己需要使用 s 個格子的置物櫃，如果剩下的容量不夠，就必須退掉某些客戶的租約。假設每個客戶所租容量只能全退或全不退，而退租第 i 個客戶損失的利益與其所租容量 $f(i)$ 相同，請寫一支程式計算王老先生最小的損失利益，如果不須退租，則損失為 0。

Time limit: 1 秒

輸入格式：測試資料有兩行，第一行有三個正整數，依序是 n 、 M 與 s ，其中 $s \leq M$ ，第二行是 n 個正整數 $f(1)$ ， $f(2)$ ， $f(3)$ ， \dots ， $f(n)$ ，同一行的數字間以空白隔開。 $1 \leq n \leq 100$ ， $M \leq 2e5$

輸出：輸出王老先生最小的損失利益。

範例一輸入： 3 10 6 4 4 1	範例一輸出： 5
範例二輸入： 5 20 14 8 2 7 2 1	範例二輸出： 15

範例一說明：總共容量 10，已出租 9，剩下 1，需要 6，最少退租 $4+1=5$ 。

範例二說明：總共容量 20，已出租 20，剩下 0，需要 14，最少退租 $8+7=15$ 。

6.2.3. 1D1D

1D1D 的 DP 是指有 $O(n)$ 個子問題 (狀態)，但每一個子問題的計算需要 $O(n)$ 個子問題的結果，也就是說一個狀態會牽涉 $O(n)$ 個前置狀態。如果直接的計算，時間複雜度就是 $O(n^2)$ ，不過很多 1D1D 問題的計算都會利用資料結構或問題的特性來降低複雜度，

也就是因為如此，1D1D 的 DP 出現在競賽或考試場合的題目通常比前面看到的 1D0D 或 2D0D 的題目來得難，有些題目甚至可以說非常難。以下我們挑選一些適合這份教材難度的典型題目。

我們先從簡單的開始，這一個例題我們在第一章說明遞迴時曾經拿出來舉例，這是一個非常有名的數，興趣的可以去網路搜尋一下，這裡我們只關心如何計算。

P-6-11. Catalan number

Catalan number 的遞迴式定義：

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

計算 C_n 除以 P 的餘數， $P=1e9+9$ 。

Time limit: 1 秒

輸入格式：輸入一個非負整數 n ， $n < 100$ 。

輸出： C_n 除以 P 的餘數。

範例一輸入： 3	範例一輸出： 5
範例二輸入： 20	範例二輸出： 564120366

題目中直接給了 Catalan number 的遞迴定義，要計算它的第 n 項，因為數字很大，所以要求輸出除以 P 的餘數。遞迴式中涉及的運算只有加法與乘法，因此只要每次運算完都取一次餘數就可以了，而 P 的平方可能超過 `int` 的範圍但不會超過 `long long` 的範圍，所以計算過程要使用 `long long`，但要記得每一次運算完都要取餘數，不可以幾個運算寫在一起算完了才取餘數，那就有可能在過程中溢位。雖然這一題正常的 DP 寫法就很容易，以下我們用它來示範 top-down 的寫法。

以下是範例程式。Top-down DP 的重點是準備一個 `memo[]` 來記錄算過的答案，主程式第 19 行我們將 `memo` 初設為 -1 來表示沒算過，接著依照定義 `memo[0]=1`，然後就呼叫遞迴函數 `cat(n)` 計算與輸出答案。函數 `cat(n)` 一進入時先檢查 `memo[n]` 是否大於等於 0，若是，表示曾經算過，直接回傳答案；否則，依照定義在第 8~12 行利用遞

迴計算，最後在第 13 行回傳，且回傳前先存入 memo[n] 中。如果不使用 memo 的技術，純遞迴會非常耗時間，但如果這樣寫，時間複雜度跟 bottom-up 的一樣都是 $O(n^2)$ 。

```

00 // P-6-11 Catalan number // top-down
01 #include <bits/stdc++.h>
02 using namespace std;
03 typedef long long ll;
04 #define P 1000000009
05 ll memo[101]={0};
06 ll cat(int n) {
07     if (memo[n]>=0) return memo[n];
08     ll sum=0;
09     for (int i=0; i<n;i++) {
10         ll t=cat(i)*cat(n-1-i)%P;
11         sum = (sum+t)%P;
12     }
13     return memo[n]=sum;
14 }
15
16 int main() {
17     int n;
18     scanf("%d",&n);
19     for (int i=0;i<=n;i++)
20         memo[i]=-1;
21     memo[0]=1;
22     printf("%lld\n",cat(n));
23     return 0;
24 }

```

所附檔案中還有兩支範例程式，一支是正常 bottom-up 的寫法，另外一支是運用數學解，Catalan number 在數學上可以解出來，第 n 項的值是 $C(2n, n) / (n+1)$ ，其中 $C(2n, n)$ 是 $2n$ 個相異物取 n 個的組合數，所以我們可以直接用公式計算它的值，但是因為中間涉及除法，而模 P 的運算不可以直接使用除法，需要轉換成乘以除數的模反元素。計算 Catalan number 是解遞迴關係(差分方程)一個經典例子，但這些都超過中學生的數學範圍，在離散數學的課程中可以學到。上面這段補充說明只是給有興趣的人參考，如果覺得困難，你可以略過。

下面這一題是 P-6-2 的推廣，留做習題。

Q-6-12. 楊鐵心做 1 休 K

楊鐵心帶著義女穆念慈做武術表演已經很久了，因為有些表演(例如胸口碎大石)實在很傷身體，所以現在每次表演完都要休息養傷。他接到許多的邀約，每天均有一場。每一場表演都可以得到某些金額的報酬，但是每次表演完至少要休息 K 天，請你協助他決定應該接受那些表演以得到最大的報酬。

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，依序代表第 1 天開始每天邀約報酬，數字間以空白隔開。 $K < n \leq 1e5$ ，每天酬勞不超過 10000。

輸出：最大可能獲得的總酬勞。

範例一輸入： 5 1 1 2 3 1 5	範例一輸出： 9
範例二輸入： 8 3 2 1 1 7 3 2 9 2	範例二輸出： 11

範例一說明：挑選 $1+3+5=9$ 。

範例二說明：挑選 $2+9=11$ 。請注意，如果挑了 9，就不可以挑 7，兩者只隔 2 天。

下一題是 P-6-3 的推廣型。

P-6-13. 周伯通的基地台 (CC)

周伯通開了一家通訊大廠並以他的名字命名，就叫做伯通公司，也有不少人弄錯了稱為博通。周伯通標下了一個政府標案，要在一條長長的大街架設基地台，長街被劃分成 n 個區段，編號依序為 $1 \sim n$ 。在第 i 個區段架設基地台的話，需要成本 $c[i]$ ，而可以服務第 $i-k$ 到第 $i+k$ 的區段 (超出範圍可忽略)。現在輸入 k ，要選一些區段架設基地台，以便每一個區段都可以被服務到，請計算最少的總成本。

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，依序代表 $c[1]$, $c[2]$, ..., $c[n]$ ，數字間以空白隔開。 $k < n \leq 2e5$ ，每處成本不超過 1000。

輸出：最小總成本。

範例一輸入： 5 1 1 2 3 1 5	範例一輸出： 2
----------------------------	-------------

範例二輸入： 8 2 2 1 1 7 3 2 9 2	範例二輸出： 3
範例一說明：挑選 1+1=2。	
範例二說明：挑選 1+2=3。	

若 $k=1$ ，這一題就是 P-6-3。令 $p(i)$ 表示可以服務 i 之前的所有位置而最後一個站架在 i 最小成本，因為與前一站的距離不可超過 $2k+1$ ，所以前一站必然在 $[i-2k-1, i-1]$ ，我們可以得到

$$p(i) = c[i] + \min\{p(j) : i-2k-2 < j < i\} \text{ for } i > k+1; \text{ and}$$

$$p(i) = c[i] \text{ for } i \leq k+1$$

這裡假設超出範圍的 $p(i)$ 為無窮大。最後的解在 $\min\{p(i) : n-k \leq i \leq n\}$ 。如果直接做，需要 $O(kn)$ 的時間，所以我們要設法很快的求得區間最小值，有一些比較高階的資料結構可以每次求的區間最小值 (RMQ, range minimum query)，但那超出了這份講義目前要講的範圍。在 P-3-8 例題中，我們曾經處理過這樣的問題：在一個滑動的視窗中每次可以找出最小值，所以這一題要綜合之前學過的這個技術，算是比較困難的題目。

回顧 P-3-8 的做法，基本上我們啟用一個 deque (雙向佇列)，這一題只會往前滑動，所以自製也容易。在 deque 中我們只須要維護一個遞增序列，因為後面如果有碰到比較小的，前面的就沒用了。所以 deque 的前端就是區間最小值，每回合先檢查前端是否已出界，然後把新的從後端放進 deque 並且移除任何大於它的資料。以下是盡量不使用 STL 的寫法，如果還沒有裝備 STL 的人可以使用這樣的寫法。

```
// P-6-13 min cost of k-domination O(n), no STL
// p[i]= min cost to cover [1..i] and choosing i
// p[i] = c[i]+min{p[j]: j>i-2k-2}; and p[i]=c[i] for i<=k+1
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define N 200005
int main() {
    int n,k,c[N],p[N];
    scanf("%d%d", &n, &k);
    for (int i=1; i<=n; i++)
        scanf("%d", &c[i]);
    // deque, index of p[], increasing p[]
    int minq[N], head=0, tail=0;
```

```

p[1]=c[1];
minq[tail++]=1; // push_back
for (int i=2; i<=n; i++) {
    if (i<=k+1)
        p[i]=c[i];
    else {
        if (minq[head]<i-2*k-1)
            head++; // out of range, pop_front
        p[i] = p[minq[head]] + c[i];
    }
    // pop larger and push itself
    while (head<tail && p[minq[tail-1]]>=p[i])
        tail--;
    minq[tail++]=i; // push i
}
// find answer: min p[n-k:n]
int ans=p[n-k];
for (int i=n-k+1; i<=n; i++)
    if (p[i]<ans) ans=p[i];
printf("%d\n", ans);
return 0;
}

```

以下是使用 STL 的 deque 的寫法。

```

// P-6-13 min cost of k-domination
// p[i]= min cost to cover [1..i] and choosing i
// p[i] = c[i]+min{p[j]: j>i-2k-2}; and p[i]=c[i] for i<=k+1
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define N 200005
int main() {
    int n,k;
    int c[N];
    scanf("%d%d", &n, &k);
    for (int i=1; i<=n; i++)
        scanf("%d", &c[i]);
    int p[N];
    deque<int> minq; // index of p[], increasing
    p[1]=c[1];
    minq.push_back(1);
    for (int i=2; i<=n; i++) {
        if (i<=k+1)
            p[i]=c[i];
        else {
            if (minq.front()<i-2*k-1)
                minq.pop_front();
            p[i] = p[minq.front()] + c[i];
        }
    }
    // pop larger and push itself

```

```

    while (!minq.empty() && p[minq.back()]>=p[i])
        minq.pop_back();
    minq.push_back(i);
}
printf("%d\n", *min_element(p+n-k, p+n+1));
return 0;
}

```

因為我們之前也教了 PQ (priority_queue)，這一題因為區間一直往右滑動，所以區間的最小值也可以用 PQ 來做，不過前兩個寫法的複雜度只有 $O(n)$ ，而使用 PQ 是 $O(n \log(n))$ ，而且這個寫法基本上是一種偷懶的方法，如果不是走競賽程式的人，不推薦學這種方式。

```

// P-6-13 min cost of k-domination, using PQ
// p[i]= min cost to cover [1..i] and choosing i
// p[i] = c[i]+min{p[j]: j>i-2k-2}; and p[i]=c[i] for i<=k+1
#include <bits/stdc++.h>
using namespace std;
#define N 200005
int main() {
    int n,k, c[N], p[N];
    scanf("%d%d", &n, &k);
    for (int i=1; i<=n; i++)
        scanf("%d", &c[i]);
    priority_queue<pair<int,int>> pq; // (-p[i], i)
    p[1]=c[1];
    pq.push({-p[1],1});
    for (int i=2; i<=n; i++) {
        if (i<=k+1)
            p[i]=c[i];
        else {
            // remove out-of-range minimum
            while (pq.top().second<i-2*k-1)
                pq.pop();
            p[i] = -pq.top().first + c[i];
        }
        pq.push({-p[i],i});
    }
    printf("%d\n", *min_element(p+n-k, p+n+1));
    return 0;
}

```

下一題是 P-4-12. (一次買賣)的推廣型，

Q-6-14. K 次買賣 (106 高中全國賽 subtask)

某商品在某個時期每一天的價格是 $p[1], p[2], \dots, p[n]$ 。假設只能先買後賣，且賣了之後才能再買，請計算買賣 k 次的最大獲利總價差，允許當天買賣，也就是不超過 k 次的買賣。

Time limit: 1 秒

輸入格式：第一行是正整數 n 與 k ，第二行有 n 個正整數 $p[1], p[2], \dots, p[n]$ 。
 n 不超過 $1e5$ ， $k < 100$ ，商品價格皆不超過 $1e7$ 。

輸出：買賣不超過 k 次的最大總價差。

範例一輸入： 5 1 3 5 1 4 0	範例一輸出： 3
範例二輸入： 7 2 1 3 7 5 1 4 0	範例二輸出： 9

範例一說明：買 1 賣 4。

範例二說明：買 1 賣 7，再買 1 賣 4。

(本題有效率更高的解法，但難度太高，這裡只要求 $O(kn)$ 的解)

Q-6-14 提示：令 $f(i, j) =$ 在第 j 天之內完成不超過 i 次買賣的最大獲利，則
 $g(i, j) = \max\{p[j] - p[t] + f(i-1, t-1) : \text{for all } t < j\}$ 是在第 j 天最後賣出的最大獲利，顯然算出 g 之後再取一次的 prefix maximum 就可以得到 f 。運用一個 i 迴圈跑 k 次，在 i -迴圈內，對每一個 j 點找出對應的 t 就可以算出 $g()$ 與 $f()$ 了。

P-6-15. 一覽衆山小

所謂「會當凌絕頂，一覽衆山小。」很多人都想爬到高處，為的是將群山看小，但是登高必自卑，行遠必自邇，最好是一步一步逐步往上，妄想一步登天的人往往摔的慘重。小說與遊戲中出現的人物往往都是越晚出現越厲害，現在已知所有人物的戰力與出場順序，想要找出依照出場順序而且戰力逐步上升的人物序列，請你計算滿足這樣要求的序列的最大可能長度。

Time limit: 1 秒

輸入格式：第一行有一個正整數 n 代表出場的人物數。第二行有 n 個非負整數，是依出場順序的每一位人物的戰力，數字間以空白隔開。 $n \leq 2e5$ ，戰力值不超過 $1e8$ 。

輸出：出場順序與戰力皆遞增的序列的最大可能長度。

範例一輸入： 8 2 4 1 3 6 4 6 9	範例一輸出： 5
範例二輸入： 5 5 4 3 2 1	範例二輸出： 1

範例一說明：可以挑選子序列 (1, 3, 4, 6, 9)，長度為 5。

範例二說明：輸入是個遞減序列，找不到長度大於 1 的遞增子序列。

一個序列的子序列是指可以任意挑選某些成員，但是不可以交換他們的前後順序，也可以說是刪除某些元素。本題是要在一個數列中找到一個最長的遞增子序列，所以這個問題被稱為 LIS (longest Increasing subsequence)。這算是一個經典的 1D1D 的 DP，因為不會太困難但是又需要一些技巧。

假設序列是 $S[1:n]$ ，從 DP 的思維開始想，我們定義 $lis(i)$ 是以第 i 點結束的最長 LIS 長度，因為前一點一定要比 $S[i]$ 小，所以很容易可以寫出遞迴式，為了方便起見，我們在序列前方加一個 -1，這樣便不需要初始條件了：

$$lis(i) = \max\{lis(j) + 1 : j < i \text{ and } S[j] < S[i]\}。$$

如果直接實作這個遞迴式，顯然需要 $O(n^2)$ ，因為每個 i 要往前搜尋所有它前面的點。考慮有那些沒有用的計算，假設我們已經做到第 i 個點。對於 $j < i$ ，若 $S[i] \leq S[j]$ 而 $lis(i) \geq lis(j)$ ，也就是 $S[i]$ 是比較小的結尾但是有比較長的 LIS，那麼 j 顯然是沒有用的，因為可以接在 $S[j]$ 後面的一定也可以接在 $S[i]$ 後面，而接在 $S[i]$ 後面可以得比較好的結果。如果我們刪除那些沒有用的子問題結果，會剩下甚麼呢？每一種可能的長度只會有一個點被留下來， lis 長度為 L 而被留下來的就是「長度 L 的最小可能結尾」。如果我們將這個資料存放在一個陣列 $last[]$ ，這個陣列的元素是單調遞增的，在第 i 個回合計算 $lis(i)$ 時，我們搜尋找到 $L = \max(j : last[j] < S[i])$ ，於是我們得到 $lis(i) = L + 1$ ，最後我們要將 $lis(i)$ 的資料納入，也就是說 $L + 1$ 的長度

多一個可能結尾是 $S[i]$ ，因此 $last[L+1]=\min(last[L+1], S[i])$ 。請稍等，還記得 L 是怎麼來的嗎？ L 是小於 $S[i]$ 的最大的那一個 $last[]$ ，也就是說除非 L 是 $last[]$ 的最後一個元素，否則 $last[L+1] \geq S[i]$ ，因此根本不必多做一次比較。

以下的範例程式是直接根據這樣的思考寫的，作法很直接，序列也不必先讀進來存，讀一個做一個就可以。雖然比 $O(n^2)$ 好，但是還是不太好，因為時間複雜度其實是 $O(nL)$ ，其中 L 是 LIS 的長度，也就是 $last[]$ 的長度，在最壞的狀況還是 $O(n^2)$ 。

```
// P-6-15 LIS O(nL), L is the length, linear search
#include <bits/stdc++.h>
using namespace std;
#define N 200010

int main() {
    int n;
    scanf("%d", &n);
    int last[N], len=0; // last[i]=the min end of length i+1
    // last will be increasing
    for (int i=0; i<n; i++) {
        int si;
        scanf("%d", &si);
        int r = 0;
        // find the max last[r] < t
        while (r<len && last[r]<si)
            r++;
        last[r] = si;
        if (r==len) len++;
    }
    printf("%d\n", len);
    return 0;
}
```

記得 $last[]$ 是個單調遞增的陣列嗎？長度越長的結尾越大，既然是單調陣列當然就可以二分搜了，二分搜可以用 STL 的 `lower_bound()`，如果還不熟也可以自己寫，建議用我們第二章介紹的一路往前跳的方式而避免使用左右轉頭搜的方式。以下的範例程式中，我們把兩種方法寫在同一個程式中以供參考，時間複雜度都是 $O(n \log(L))$ 。

```
// P-6-15 LIS O(nlog(L)), L is the length
#include<bits/stdc++.h>
using namespace std;
#define N 200010
// binary search to find the first >=x
int bsearch(int s[], int n, int x) {
    if (n==0 || s[0]>=x) return 0;
    int po=0;
    for (int jump=n/2; jump>0; jump>>=1) {
```

```

        while (po+jump<n && s[po+jump]<x)
            po += jump;
    }
    return po+1;
}
int main() {
    int n;
    scanf("%d",&n);
    int last[N], len=0; // last[i]=the min end of length i+1
    // last will be increasing
    for (int i=0;i<n;i++) {
        int si;
        scanf("%d",&si);
        // one of the next two binary search
        int r = bsearch(last, len, si);
        // only for checking
        int it = lower_bound(last, last+len, si) - last;
        assert(r==it);
        last[r] = si;
        if (r==len) len++;
    }
    printf("%d\n",len);
    return 0;
}

```

下面在附上用 vector 的寫法，背後有不簡單的思維，但最後結果非常簡潔，無怪乎成為經典。

```

// P-6-15 LIS, using STL
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n, si;
    scanf("%d",&n);
    vector<int> last;
    for (int i=0; i<n; i++) {
        scanf("%d", &si);
        auto it = lower_bound(last.begin(), last.end(), si);
        if (it==last.end()) last.push_back(si);
        else *it=si;
    }
    printf("%d\n", (int)last.size());
    return 0;
}

```

以下也是個經典例題，它是 P-4-4 的加權版本。

P-6-16. 山寨華山論劍

自從華山論劍聲名大噪之後，很多人都想參加，但 20 年辦一次實在太久了，五輪奧運都跑 5 趟了，於是各地都紛紛舉辦華山論劍，往往同一天就有很多場。每一場都有自己的開始時間與結束時間，參加者必須全程在場，所以兩場的時間如果有重疊就不能同時參加。令狐冲本來想參加最多場來盡速累積經驗值，但後來他發現有很多場次能獲得的經驗值很少，參加了反而浪費時間，現在要請你幫忙計算最多可以得到的經驗值總和。請注意，所挑選活動必須完全不重疊，兩場活動只在端點重疊也是不可以同時參加的，也就是說前一場的結束時間必須小於下一場的開始時間。

Time limit: 1 秒

輸入格式：第一行是一個正整數 n ，代表一共舉辦了 n 場華山論劍，接下來 n 行，每行有三個非負整數 s 、 t 與 e ，代表這場活動時間區間是 $[s, t]$ ，而可以得到的經驗值是 e ，兩數字間以空白間格。 n 不超過 $1e5$ ， $s < t < 1e9$ ，答案不超過 $1e9$ 。

輸出：最大可能經驗值總和。

範例一輸入： 3 1 2 1 2 3 3 3 4 1	範例一輸出： 3
範例二輸入： 3 1 2 2 2 3 3 3 4 2	範例二輸出： 4

範例一說明：挑選 $[2, 3]$ ，得到經驗值 3。

範例二說明：挑選 $[1, 2]$ 與 $[3, 4]$ ，得到經驗值 $2+2=4$ 。

每一場華山論劍(活動)看成一個線段，每個線段除了有起點(左端)與終點(右端)之外，還有一個權重。問題是要挑選權重總和最大且互相不重疊的線段，這個問題也稱為 weighted activity selection problem，如果權重都一樣，就是要挑選最多活動，那就是第四章的 P-4-4，沒權重的時候是用貪心法則來求解，挑選最早結束的活動

就對了，但是在加權版本時，範例一告訴你，挑最早結束的是錯的，範例二告訴你，挑權重最大的也是錯的。

放棄貪心來試試 DP。假設線段放在 $s[]$ 中，第 i 個線段的區間是 $[s[i].L, s[i].R]$ ，權重是 $s[i].w$ 。先讓線段依照右端排序，定義 $dp[i]$ 是前 i 個線段的最佳解 (最大權重和)。第 i 個線段可能挑或不挑，不挑的話，很簡單 $dp[i]=dp[i-1]$ ；如果挑呢？跟它重疊的都不能挑，也就是說右端落在 $s[i]$ 範圍內的線段都要剔除，根據 $dp[i]$ 的定義，我們暫且不需要管 $i+1$ 以後的線段。剔除了右端落在 $s[i]$ 範圍的線段就是右端在 $s[i].L$ 之前的線段，也就是 $\max\{dp[j]: s[j].R < s[i].L\}$ 。因為 $dp[]$ 顯然是非下降的，所以我們要找的 $dp[j]$ 就是滿足右端小於 $s[i].L$ 的最大編號 j 。整理一下我們的遞迴式，選或不選 i ，兩者挑最大：

$$dp[i] = \max\{dp[i-1], s[i].w + dp[j]\}, \quad j = \max\{x: s[x].R < s[i].L\}$$

從狀態轉移的角度來看， i 的前置狀態有兩個，一個是 $i-1$ (不選 i)，另外一個就是在它開始之前最後一個結束的線段 (選 i)，這樣看其實不難理解，當然，要找到 j 需要一點技術。先提出根據上面的分析後的天真作法，排序後一一尋找，雖然說天真但也不太單純，我們需要排序一個三個欄位的資料，因此定義了一個 `struct` 來放置線段，定義了一個比較函數以便依照右端排序，不熟悉的話可能需要複習第二章。排序後以一個 i 迴圈每次計算 $dp[i]$ ，其中以一個 j 迴圈來找到 i 開始前的最後一個線段，如果沒有，之前的最大權重相當於 0，程式很直接，複雜度 $O(n^2)$ 。

```
// Weighted activity selection problem, O(n^2) method
// overlap at endpoint is not allowed
// dp[i] is the max for interval until right(i), sorted by right
// dp[i]=max{dp[i-1], dp[j]+w[i]}, j is the max right < left(i)
#include<bits/stdc++.h>
using namespace std;
#define N 200010
typedef struct {
    int L,R,W;
} SEG;
SEG seg[N];
// sort by right, same right points OK since dp[] is increasing
bool comp(const SEG p1, const SEG p2) {
    return p1.R < p2.R;
}

int main() {
    int n, dp[N];
    scanf("%d", &n);
    for (int i=0; i<n; i++)
        scanf("%d%d%d", &seg[i].L, &seg[i].R, &seg[i].W);
    sort(seg, seg+n, comp); //sort by right
    dp[0]=seg[0].W;
    for (int i=1; i<n; i++) {
```

```

    // linear search largest j < left[i]
    int j=i-1;
    while (j>=0 && seg[j].R>=seg[i].L)
        j--;
    dp[i] = seg[i].W + ((j<0)? 0 : dp[j]);
    if (dp[i]<dp[i-1]) dp[i]=dp[i-1];
}
printf("%d\n", dp[n-1]);
return 0;
}

```

因為右端是經過排序的，當然應該用二分搜比較有效率。為了簡化邊界條件，我們可以塞一個右端為-1 而權重為 0 的線段，如此一來每一個每一個線段都找得到它的前置線段。以下是自己寫二分搜的範例程式。

```

// Weighted activity selection problem, using binary search.
// overlap at endpoint is not allowed
// dp[i] is the max for interval until right(i), sorted by right
// dp[i]=max{dp[i-1], dp[j]+w[i]}, j is the max right <left(i)
#include <bits/stdc++.h>
using namespace std;
#define N 200010
typedef struct {
    int L,R,W;
} SEG;
SEG s[N];
// sort by right, same right points OK since dp[] is increasing
bool comp(const SEG p1, const SEG p2) {
    return p1.R < p2.R;
}

int main() {
    int n, dp[N];
    scanf("%d", &n);
    for (int i=1; i<=n; i++) {
        scanf("%d%d%d", &s[i].L, &s[i].R, &s[i].W);
        assert(s[i].L<s[i].R);
    }
    s[0].R = -1; // dummy for boundary
    sort(s, s+n+1, comp); //sort by right
    dp[0] = 0;
    for (int i=1; i<=n; i++) {
        // binary search to find largest j < left[i]
        int j=0;
        for (int jump=i/2; jump>0; jump>>=1) {
            while (j+jump<i && s[j+jump].R<s[i].L)
                j += jump;
        }
        dp[i] = dp[j] + s[i].W;
        if (dp[i]<dp[i-1]) dp[i] = dp[i-1];
    }
}

```

```

}
printf("%d\n", dp[n]);
return 0;
}

```

當然二分搜也可以呼叫 `lower_bound()`，但是要注意比較函數的寫法，另外也可以用 `map` 來做，範例程式中有這兩種寫法，提供有興趣的人參考，這裡就不多說明了。

6.2.4. 2D1D 與其他

P-6-17. 切棍子

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，我們可以選擇棍子上的切割位置，機器會將此棍子在指定位置切斷，而切割成本是該段棍子的長度。現在有一根長度 L 的棍子，上面標有 n 個需要切割的位置座標，因為不同的切割順序會影響切割總成本，請計算出最小的切割總成本。

例如 $L=10$ ，三個切割點的座標是 $(2, 4, 7)$ 。如果切割順序是 $(2, 4, 7)$ ，則第一次切的成本是 10，第二次的成本是 8，第三次成本 6，總成本 $10+8+6=24$ 。如果切割順序改成 $(4, 2, 7)$ ，第一次切的成本是 10，切成長度 4 與 6 的兩段，第二次的成本是 4，第三次成本 6，總成本 $10+4+6=20$ 。

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 $p[1] \sim p[N]$ ，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 。 $N \leq 200$ ， $L \leq 1e6$ 。

輸出：最小的切割總成本。

範例一輸入： 3 10 2 4 7	範例一輸出： 20
範例二輸入： 5 10 1 6 7 8 9	範例二輸出： 24

從題目給的範例，很可能讓我們猜想每次應該取中點或 median 來切割，但事實上都是錯的。直接思考 DP 的做法，定義一段棍子的切割成本最直接的方式就是以左右端點，因為是有給座標的，我們就以所給的座標當作第 $1, 2, \dots, n$ 點，定義 $\text{cost}(i, j)$ 是第 i 點到第 j 點之間這段的最低切割成本，為了方便，以座標 0 與 L 當作第 0 與第 $n+1$ 點，所以我們要計算的就是 $\text{cost}(0, n+1)$ 。

對任意 $i < j$ ， $\text{cost}(i, j)$ 就是一個子問題，所以我們有 $O(n^2)$ 個子問題，現在要找出遞迴式。對於 $j=i+1$ ， $\text{cost}(i, j)=0$ ，因為中間沒有要切割的點；否則 $j>i+1$ ，假設第一刀切在 k ，我們雖然不知道 k 在哪裡，但是一旦第一刀切在 k 之後，剩下兩段的成本就是 $\text{cost}(i, k)$ 與 $\text{cost}(k, j)$ 。DP 的思維：既然不知道 k 是多少，那就全部都算吧！所以我們可以得到

$$\text{cost}(i, j) = \begin{cases} 0 & \text{if } j = i + 1 \\ \min_{i < k < j} \{ \text{cost}(i, k) + \text{cost}(k, j) \} + p[j] - p[i] & \text{otherwise} \end{cases}$$

其中 $p[j]$ 與 $p[i]$ 是兩點的座標。這是個遞迴式嗎？是的，雖然跟之前看到的有點不一樣，遞迴都是大往小，我們注意等號右邊的座標間格 $(k-i)$ 與 $(j-k)$ 都會小於左邊的 $(j-i)$ ，所以它是區間由大到小的遞迴，類似二分搜尋與排序的狀況。所以這一題的計算順序稍嫌麻煩，是一個試試 top-down 好時機。我們可以看到寫 top-down DP 有多麼簡單（在你找出遞迴式之後）。

以下是範例程式，基本上直接寫遞迴函式差不多。一開始把輸入座標讀進來，然後準備小抄 memo，因為我們的子問題是所有 (i, j) ，就宣告一個 `int memo[N][N]`，有一半用不到不需要太吝嗇，第 23~25 行將它們初設為 -1，用來代表沒有計算過，而將所有 `memo[i][i+1]` 設為 0，然後就直接呼叫 `cost(0, n+1)` 輸出。在函數 `cost(i, j)` 中，一進入時先檢查對應的 `memo[i][j]` 是否不是負的（第 9 行），若是，表示曾經算過，便直接回傳表格內容。否則我們要進行遞迴呼叫來計算，第 10~12 行就是根據遞迴式寫的計算各種第一刀 k 的可能，在其中取最小值，計算完畢後在第 14 行要回傳前將他寫入表格 `memo[i][j]` 中。是不是很簡單？這個程式的複雜度是多少呢？直接看遞迴不容易，看表格就好，有 $O(n^2)$ 個格子，每個格子一個不超過 $O(n)$ 的迴圈，所以就是 $O(n^3)$ 。這一題是存在優化的方法，不過這些優化需要依靠一些數學上的性質，不是我們這份教材的範圍。

```
00 // P-6-17 rod-cutting, top-down DP O(n^3)
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 600
04 #define oo 99999999
05 int p[N]; // points
06 int memo[N][N];
07 // min cost of [i,j]
```

```

08 int cost(int i, int j) {
09     if (memo[i][j]>=0) return memo[i][j];
10     int mincost = oo;
11     for (int k=i+1; k<j; k++)
12         mincost = min( mincost, cost(i,k) + cost(k,j));
13     mincost += p[j] - p[i];
14     return memo[i][j]=mincost;
15 }
16
17 int main() {
18     int n,L;
19     scanf("%d%d", &n, &L);
20     for (int i=1; i<=n; i++)
21         scanf("%d",&p[i]);
22     p[0]=0, p[n+1]=L;
23     for (int i=0;i<n+2;i++)
24         for (int j=i+1;j<n+2;j++)
25             memo[i][j]=-1;
26     for (int i=0; i<n+1; i++)
27         memo[i][i+1] = 0;
28     printf("%d\n", cost(0,n+1));
29     return 0;
30 }

```

如果要寫非遞迴非 top-down 的寫法要如何寫呢？其實也不是多難，我們剛才提到子問題的大小看 $j-i$ ，所以非遞迴的順序要依照 $j-i$ 由小到大計算，如果看成一個矩陣，計算的順序是由對角線逐漸往右上角算。底下是非遞迴的範例程式，要小心迴圈的邊界，這個程式裡面是用一個變數代換 $len=j-i$ ，這樣比較好思考也比較不容易寫錯。

```

// P-6-17 rod-cutting, bottom-up DP  $O(n^3)$ 
#include <bits/stdc++.h>
using namespace std;
#define N 600
#define oo 999999999

int main() {
    int n, L, p[N], cost[N][N];
    scanf("%d%d", &n, &L);
    for (int i=1; i<=n; i++)
        scanf("%d",&p[i]);
    p[0]=0, p[n+1]=L;
    for (int i=0; i<n+1; i++)
        cost[i][i+1] = 0;
    // len=j-i
    for (int len=2; len<n+2; len++) {
        for (int i=0; i+len<n+2; i++) { // cost(i, i+len)
            int j=i+len, mcost = oo;
            for (int k=i+1; k<j; k++)
                mcost = min(mcost, cost[i][k]+cost[k][j]);
            cost[i][j] = mcost + p[j] - p[i];
        }
    }
}

```

```

    }
}
printf("%d\n", cost[0][n+1]);
return 0;
}

```

下面一題是類似的習題，也是 2D1D 的 DP，這裡同樣只要求 $O(n^3)$ 的複雜度，這一題同樣存在考數學性質的優化。

Q-6-18. 矩陣乘法鏈

兩個矩陣 A 與 B 要相乘，必須滿足 A 的行數等於 B 的列數，若 A 是 $p \times q$ 的矩陣，B 是 $q \times r$ 的矩陣，則 $A \times B$ 的結果是一個 $p \times r$ 的矩陣，而需要的純量乘法數假設是 $p \times q \times r$ 。矩陣乘法滿足結合律，也就是說 $A \times B \times C = (A \times B) \times C = A \times (B \times C)$ ，可是不同順序所需要的純量乘法數不同。輸入 $p[0], p[1], \dots, p[n]$ ，代表有 n 個矩陣相乘的乘法鏈，而它們的矩陣的大小依序分別是 $p[0] \times p[1]$ 、 $p[1] \times p[2]$ 、 \dots 、 $p[n-1] \times p[n]$ ，請找出最好的相乘順序使得使用的純量乘法數的總和最少。

舉例來說， $n=3$ ，矩陣大小為， $A_1: 3 \times 5$ ， $A_2: 5 \times 4$ ， $A_3: 4 \times 2$

乘法順序為 $(A_1 \times A_2) \times A_3$ 時，乘法數 $3 * 5 * 4 + 3 * 4 * 2 = 84$ ；若乘法順序為 $A_1 \times (A_2 \times A_3)$ 時，乘法數 $3 * 5 * 2 + 5 * 4 * 2 = 70$ 。最少的純量乘法數為 70。

Time limit: 1 秒

輸入格式：第一行是正整數 n 。第二行有 $n+1$ 個正整數， $p[0] \sim p[n]$ ，數字間以空白隔開。 $n \leq 200$ ， $p[i] \leq 200$ 。

輸出：最小的純量乘法數量。

範例一輸入： 3 3 5 4 2	範例一輸出： 70
範例二輸入： 4 5 1 3 4 2	範例二輸出： 30

下一題出現在 APCS 的考試中，算是歷屆比較難的題目，因為不在常見的套路中。

P-6-19. 階梯數字 (APCS201802) (@@)

一個正整數如果從高位數往低位數看過去，每一位數字只會相等或變大，則我們稱它為階梯數字，例如：9、234、777、11222233。給定一正整數 N ，請計算不大於 N 的階梯數字總共有幾個，請注意本題只算正整數，所以 0 不算階梯數字，而且階梯數字不會以 0 開始。

提示：對於 N 不大的情形，例如 100000，我們可以枚舉每一個不超過 N 的數字來判斷。對於 N 很大的時候，就需要比較快速的方法了。我們知道所有的一位數 1~9 都是階梯數字，如果要計算 D 位數的階梯數字總數，我們可以依據最高位數字分成 9 類，然後根據 $(D - 1)$ 位的 9 類階梯數字總數來計算。

Time limit: 1 秒

輸入格式：輸入一個正數字 N 。 $N \leq 1e18$ 。

輸出：不大於 N 的階梯數字個數。

範例一輸入： 25	範例一輸出： 22
範例二輸入： 101	範例二輸出： 54

範例一說明：1~9, 11~19, 22~25, 共 22 個。

範例二說明：1~9, 11~19, 22~29, 33~39, 44~49, 55~59, 66~69, 77~79, 88~89, 99, 一共有 54 個。

要判斷一個數字是否是階梯數字並不難，只要把每一位數字拆解出來再檢查是否單調上升就可以了，题目的提示中也提到，數字範圍小的時候可以枚舉，但 N 很大時就必須想快速的方法了。這一題比較需要思考，我們先來看看如何以枚舉來解簡單的子題。以下是範例程式，我們設計一個副程式 `step(n)` 用來判斷 n 是否是階梯數字，將 n 從低位往高位拆解，如果有上升的狀況即回傳 0，如果都沒有，最後回傳 1。主程式中就把 1 開始的所有數字都拿來檢查看看。

```
// P_6-19 slow method
#include <cstdio>
typedef long long LL;
```

```

int step(LL n) {
    int d=9;
    while (n) {
        if (n%10>d) return 0;
        d = n%10;
        n /= 10;
    }
    return 1;
}

int main() {
    LL n,i,ans=0;
    scanf("%lld",&n);
    for (ans=0,i=1; i<=n; i++)
        ans += step(i);
    printf("%lld\n",ans);
    return 0;
}

```

上述方法當然只能得到子題的分數，我們來看看题目的提示：「如果要計算 D 位數的階梯數字總數，我們可以依據最高位數字分成 9 類，然後根據 $(D - 1)$ 位的 9 類階梯數字總數來計算。」舉例來說明，假設 $N = 358$ 是一個三位數，要計算不大於 358 有多少階梯數字：

- 我們可以用最高位數分成 4 類：以 0, 1, 2, 或 3 開頭的三位階梯數，其中 0~2 開頭的所有階梯數都要包含，但 3 開頭的只要不大於 358 的階梯數
- 3 開頭不大於 358 的階梯數，進一步可分成 33 開頭、34 開頭、以及 35 開頭，其中 35 開頭的只要不大於 358 開頭的。
 - 33 開頭的三位階梯數其實就是 3 開頭的二位數，同樣的，34 開頭的三位數就是 4 開頭的二位數。
- 35 開頭不超過 358 的就是不大於 8 的一位數，有 4 個，355~358。

從提示與上面的例子可以看出所需要的方法分兩大步驟：

- 計算一個表格 $step[i][j]$ ， $step[i][j]$ 為 j 開頭的 i 位階梯數字的個數。
- 依據輸入的數字 N ，在表格中挑選所需要累加的表格位子。

以下的表格是 4 位數以內的 $step$ 表格。

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9
--	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

i=1	0	1	1	1	1	1	1	1	1	1
i=2	9	9	8	7	6	5	4	3	2	1
i=3	54	45	36	28	21	15	10	6	3	1
i=4	219	165	120	84	56	35	20	10	4	1

首先看表格如何計算。表格的第一列對應到 $i=1$ 位數的情形，此時根據定義除了 $j=0$ 是 0 (因為 0 不是階梯數字) 之外，其它都是 1。再看表格最右方的欄， $j=9$ 代表以 9 開頭的階梯數字，很容易知道不管幾位數，9 開頭的只有一個 99...9。對於 8 開頭的 2 位數 $\text{step}[2][8]$ ，它的第 1 位是 8，第 2 位就必須不小於 8，所以 $\text{step}[2][8] = \text{step}[1][8] + \text{step}[1][9]$ ，同理， $\text{step}[2][7] = \text{step}[1][7] + \text{step}[1][8] + \text{step}[1][9]$ ，所以我們可以推論出

$$\text{step}[i][j] = \sum_{k=j}^9 \text{step}[i-1][k]$$

如果我們進一步把 $\text{step}[i][j+1]$ 帶入計算式右方，可得

$$\text{step}[i][j] = \text{step}[i][j+1] + \text{step}[i-1][j]$$

也就是說，除了第 1 列與最後一欄之外，每一個格子都是它的右方與上方格子相加的和。所以我們可以 i 從小到大， j 從 8 到 0，很容易地計算整個需要的表格。

第二個步驟要根據輸入的數字挑選出表格中的數字來相加。以上面舉例的 $N=358$ 來看，上方表格中有標記的是前面所討論應該要加起來的格子，假設輸入 N 的每一位數儲存在 $n[]$ 陣列中而 len 是 N 的位數，在此例中也就是 $n[0]=3, n[1]=5, n[2]=8$ 而 $\text{len}=3$ ，計算過程如下：

- 從 $n[0]$ 開始看，因為 $n[0]=3, \text{len}=3$ ，所以要加 0, 1, 2 開頭的三位數，也就是 $\text{step}[3][0] + \text{step}[3][1] + \text{step}[3][2] = 135$ 。
- 現在考慮的是 3 開頭的 3 位數， $n[1]=5$ ，所以要加入的是 3 與 4 開頭的 2 位數，也就是 $\text{step}[2][3] + \text{step}[2][4] = 13$ 。
- 最後考慮的是 35 開頭的 3 位數， $n[2]=8$ ，所以要加的是 5~8 開頭的 1 位數，也就是 $\text{step}[1][5 \sim 8] = 4$ 。最後的答案就是 $135 + 13 + 4 = 152$ 。

以上大致找出了規則，但還有一個問題要處理，上面的例子 N 本身是個階梯數字，這樣處理沒錯，我們來看看 N 不是階梯數字時候的情形。例如 $N=353$ ，前兩位數的推論方式

是一樣的，但是看到第 3 位時， $n[2] = 3 < n[1]$ ，所以最後的 1 位數是不要加的。又例如 $N=317$ ，3 位數的推論依舊成立，但看第 2 位時發現 $n[1] = 1 < n[0]$ ，所以後面都不要計算了，事實上我們發現第一個往下降的位數時，後面就都不需要算了，就相當於後面都是 0 是一樣的，例如 $N=3357499$ 的答案跟 $N=3357000$ 是一樣的。

根據上面的規則可以寫成以下的範例程式。第 8~11 行以字串方式處理輸入，並將字元換成數字以便計算。第 13~22 行根據上面推出的規則計算表格，第 24~31 行根據輸入數字加總表格，請注意最後的那個 `if`，因為之前的內迴圈 `j` 只計算到 $n[i]-1$ ，所以如果 N 是階梯數字，則迴圈結束後要補一個進去。

```

00 // build step table first and then count the number
01 #include <bits/stdc++.h>
02 #define ND 20
03
04 int main() {
05     int i, j, len;
06     char n[100];
07     long long step[ND][10], total=0;
08     scanf("%s", n);
09     len = strlen(n);
10     for (i=0; i<len; i++)
11         n[i] -= '0';
12     // build table, step[i][j] for i-digits and start with j
13     for (j=1; j<10; j++)
14         step[1][j] = 1;
15     step[1][0] = 0; // 0 is not stepping number
16     // step[i][j] =sum(step[i-1][k] for k>=j)
17     // or = step[i][j+1] + step[i-1][j], suffix sum
18     for (i=2; i<=len; i++) {
19         step[i][9] = step[i-1][9];
20         for (j=8; j>=0; j--)
21             step[i][j] = step[i][j+1] + step[i-1][j];
22     }
23     // counting
24     for (j=0; j<n[0]; j++)
25         total += step[len][j];
26     for (i=1; i<len; i++) {
27         if (n[i] < n[i-1]) break;
28         for (j=n[i-1]; j<n[i]; j++)
29             total += step[len-i][j];
30     }
31     if (i == len) total++; // input is a stepping number
32     printf("%lld\n", total);
33     return 0;
34 }
35

```

以下這一節最後兩個例題維度已經不是 2D，以下我們看一個很有名的交換網路。

P-6-20. Hyper-cube path

一個維度為 n 的 hypercube 是由 2^n 個節點組成的網路，每個節點被賦予唯一一個 $0 \sim 2^n - 1$ 的編號，對於任兩個節點 i 與 j ，他們之間會有邊相連若且惟若 i 與 j 的二進位編碼恰好相差一個位元，我們對於每個節點 i 給予一個正整數的權重 $w(i)$ ，請找出編號 0 到編號 $2^n - 1$ 的一條最短路徑，使得該路徑所經過的點權重總合 (包含起點與終點) 為最大。

Time limit: 1 秒

輸入格式：第一行是正整數 n ，第二行則是這 2^n 個節點的正整數權重 $w(0), w(1), \dots, w(2^n - 1)$ ，數字之間皆以一個空白間隔，其中 $n < 20$ 而每個權重值為非負整數不超過 100。

輸出：最大權重總和。

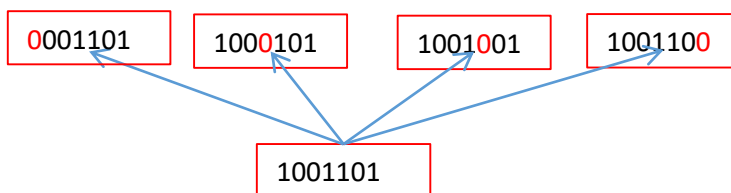
<p>範例一輸入：</p> <pre>2 1 2 3 4</pre>	<p>範例一輸出：</p> <pre>8</pre>
<p>範例二輸入：</p> <pre>3 1 2 3 4 5 6 7 8</pre>	<p>範例二輸出：</p> <pre>21</pre>

範例一說明：1 (00) -> 3 (10) -> 4 (11)，總和 8，括弧內為節點邊號的二進位。

範例二說明：1 (000) -> 5 (100) -> 7 (110) -> 8 (111)，總和 21。

一個點代表一個 n -bit 的正整數，兩點可以直接相通若且惟若兩個數字只差一個 bit，我們知道起點就是 $00\dots 0$ ，終點是 $11\dots 1$ ，每走一步就是將一個 0-bit 換成 1，所謂最短路徑就是不會 0-bit 換 1-bit 後又換回 0，所以路徑的走法就是有 $n!$ 種，也就是更換 bit 的排列順序。不過不是要求路徑數，而是哪一條路徑經過的權重總和最大。其實這一題跟方格單調路徑是類似的，只是圖形比方格棋盤複雜而已。

在方格棋盤時，每一個格子的前置狀態是它的左方與上方格子，在這一題 hyper-cube (超立方體) 時，一個數字點的前置狀態是他所有為 1 的 bit 換成 0，以 1001101



為例，它的每一個 1-bit 換成 0 就是可能走到它的前置狀態，所以一共有四個 (如下圖)。

有了這個概念後，我們必須找 DP 的計算順序，這一題是 top-down 的好範例，因為計算順序比較麻煩。每一步把一個 0-bit 改成 1，所以我們應該沿著 1-bit 個數少的往多的順序來計算，也就是先算一個 1-bit 的，然後兩個 1-bit 的，...，這實在非常麻煩。我們來看看 top-down 有多簡單，以下是範例程式。遞迴函數在第 6~18 行，相同的，進來先檢查表格 `c[goal]` 是否已經算過 (≥ 0)，如果算過直接回傳。接著我們跑一個迴圈找所有前置狀態的最大值，對於每一個 bit，`(goal & (1<<i))` 檢查 goal 的第 `i` 個 bit 是否為 1，如果是，以 `(goal ^ (1<<i))` 把第 `i` 個 bit 換成 0 就是它的一個前置狀態。最大值計算完畢後記得加上該點的權重並且寫入表格 `c[]`。

```

00 // P-6-20, hypercube path, top-down
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 1000000
04 int c[N],w[N];
05
06 int dp(int goal,int n) {
07     if (c[goal]>=0) return c[goal];
08     int imax=0;
09     // check each bit, find max previous state
10     for (int i=0; i<n; i++) {
11         if (goal & (1<<i)) { // bit-i=1
12             int k = dp(goal ^ (1<<i), n); // set bit-i to 0
13             if (k>imax) imax=k;
14         }
15     }
16     c[goal]=imax+w[goal];
17     return c[goal];
18 }
19
20 int main() {
21     int i,n,m;
22     scanf("%d", &n);
23     m = 1<<n;
24     for (i=0;i<m;i++)
25         scanf("%d", &w[i]);
26     for (i=0;i<m;i++) c[i] = -1; // not computed
27     c[0] = w[0]; // starting point
28     printf("%d\n", dp(m-1,n));
29     return 0;
30 }

```

這一題沒辦法寫非遞迴的形式嗎？當然是可以，但是以 graph 裡面的 BFS 比較容易，我們在範例程式中有放一支這樣的寫法，請有興趣的人自己去看。

下一題也是 top-down 的好例子。

P-6-21. 刪除邊界 (APCS201910)

一個矩陣的第一列與最後一列以及第一行與最後一行稱為該矩陣的四條邊界線，如果某一條邊界線的內容都是相同元素，則可以刪除該邊界線。如果一條邊界線的內容不完全相同，你可以修改某些格子的內容讓邊界線的內容變成相同後，再刪除該邊界線。矩陣在刪除一條邊界線後，還是一個矩陣，但列數或行數會減少一，本題的目標是重複執行修改與刪除邊界的動作，最後將整個矩陣刪除。輸入一個 0/1 矩陣，請計算最少要修改多少個元素才能將整個矩陣刪除。請注意：根據定義，只有一列或是只有一行的矩陣可以不需要任何修改就可以被全部刪除。

Time limit: 1 秒

輸入格式：第一行為兩個不超過 25 的正整數 m 和 n ，以下 m 列 n 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。

輸出：最少修改次數。

<p>範例一輸入：</p> <pre>4 5 0 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0</pre>	<p>範例一輸出：</p> <pre>2</pre>
<p>範例二輸入：</p> <pre>3 5 0 0 0 1 0 1 0 1 1 1 0 0 0 1 0</pre>	<p>範例二輸出：</p> <pre>1</pre>

範例一說明：刪除最後一列 (修改 1 成 0，成本 1)，刪除最後一列 (成本 0)，刪除最後一列 (修改一個 1，成本 1)，成本總和 2。

範例二說明：刪除最右行 (修改 1 個 1，成本 1)，刪除最右行 (成本 0)，刪除第一列 (成本 0)，修改最後一列 (成本 0)，總和 1。

給一個矩陣，每次刪除一個邊界，刪除邊界的成本是該邊界線上 0 與 1 比較少的個數，要計算最小的刪除總成本，也就是刪除邊界的最好的順序。我們在第一章的 Q-1-11 曾經看到這一題，當時的要求是以遞迴的方式來解，所以矩陣的大小比較小，遞迴的時間複雜度是 $O(4^{m+n-2})$ ，因為每次有 4 個選擇，每選擇一次， $m+n$ 會減少 1。

從 DP 的想法來看，因為刪除的是邊界，需要計算的不外乎所有可能的子矩陣，也就是某一塊矩形，那麼有多少矩形呢？有 $O(mn)$ 個可能的左上角與 $O(mn)$ 個可能的右下角，所以有 $O(m^2n^2)$ 個，以 top-down 的寫法，反正稀哩糊塗的拉個四層迴圈把表格建好，然後照遞迴的方式稍微改一下就可以了，以下是範例程式，雖然程式不很短，但只是四個邊界每邊都要算一次，其實程式碼很簡單。

```
// P-6-21, top-down DP,  $O(m^2 * n^2)$ 
#include <cstdio>
#define N 60
int a[N][N];
int memo[N][N][N][N]; // record the best for each rectangle
// rectangle= [left, top, right, bottom]
int cost(int l,int t,int r,int b) {
    if (memo[l][t][r][b] >= 0) // already computed
        return memo[l][t][r][b];
    int i, mcost, n0, n1, c;
    // top row
    n0 = n1 = 0; // numbers of 0 and 1
    for (i=l; i<=r; i++)
        a[t][i]? n1++ : n0++;
    mcost = cost(l, t+1, r, b) + (n1<n0 ? n1 : n0);
    // bottom-row
    n0 = n1 = 0;
    for (i=l; i<=r; i++)
        a[b][i]? n1++ : n0++; // can also in this way
    c = cost(l, t, r, b-1) + (n1<n0 ? n1 : n0);
    if (c<mcost) mcost = c;
    // left-column
    n0 = n1 = 0;
    for (i=t; i<=b; i++)
        a[i][l]? n1++ : n0++;
    c = cost(l+1, t, r, b) + (n1<n0 ? n1 : n0);
    if (c<mcost) mcost = c;
    // right column
    n0 = n1 = 0;
    for (i=t; i<=b; i++)
        a[i][r] ? n1++ : n0++;
    c = cost(l, t, r-1, b) + (n1<n0 ? n1 : n0);
    if (c<mcost) mcost = c;
    memo[l][t][r][b] = mcost;
    return mcost;
}

int main() {
    int m,n,i,j;
```

```

scanf("%d%d", &m, &n);
for (i=0;i<m;i++) for (j=0;j<n;j++)
    scanf("%d", &a[i][j]);
for (int l=0;l<n;l++) for (int r=l;r<n;r++)
    for (int t=0;t<m;t++) for (int b=t;b<m;b++)
        memo[l][t][r][b] = (r==l || b==t)? 0: -1;
printf("%d\n", cost(0,0,n-1,m-1));
return 0;
}

```

那麼非遞迴的寫法呢？要計算的矩形顯然需要從小算到大，精確一點的說，寬度由小到大且高度由小到大，對於每一個寬度與高度，再枚舉所有可能的左上角位置，以下是範例程式，我們將計算一列與一行的成本寫成副程式，這樣程式比較清爽。

```

// non-recursive dp O(m^2 * n^2)
#include <cstdio>
#define N 60
#define oo 4000
int a[N][N];
int dp[N][N][N][N]; // record the best for each rectangle
// the cost of boundary line (row,le) to (row,right)
int r_cost(int row, int le, int right) {
    int sum = 0, i;
    for (i=le; i<=right; i++)
        sum += a[row][i];
    i = right - le + 1;
    if (sum<=i/2) return sum;
    return i - sum;
}
// the cost of boundary line (top,col) to (bot,col)
int c_cost(int col, int top, int bot) {
    int sum = 0, i;
    for (i=top; i<=bot; i++)
        sum += a[i][col];
    i = bot - top + 1;
    if (sum<=i/2) return sum;
    return i - sum;
}
int main() {
    int m,n,i,j;
    scanf("%d%d", &m, &n);
    for (i=0;i<m;i++) for (j=0;j<n;j++)
        scanf("%d", &a[i][j]);
    int w, h, top, le; // width, height, top, le
    for (h=1; h<=m; h++) for (w=1; w<=n; w++) {
        // each possible top-le corner
        for (top=0; top+h<=m; top++) for (le=0; le+w<=n; le++) {
            if (w==1 || h==1) {
                dp[h][w][top][le]=0;
                continue;
            }

```

```

    }
    int mcost, cost;
    mcost = dp[h-1][w][top+1][le] + r_cost(top, le, le+w-1);
    cost = dp[h-1][w][top][le] + r_cost(top+h-1, le, le+w-1);
    if (cost < mcost) mcost = cost;
    cost = dp[h][w-1][top][le+1] + c_cost(le, top, top+h-1);
    if (cost < mcost) mcost = cost;
    cost = dp[h][w-1][top][le] + c_cost(le+w-1, top, top+h-1);
    if (cost < mcost) mcost = cost;
    dp[h][w][top][le] = mcost;
  }
}
printf("%d\n", dp[m][n][0][0]);
return 0;
}

```

6.3. 進階題 (*)

DP 的題目很多，有些在競賽中出現的往往需要某些特殊資料結構與特別的優化技巧，這些題目形式對 APCS 來說是超出範圍的，以這份教材的原出發點，或許並不適合將他們放進來。不過看這份教材的也許有些是對競賽程式有興趣的，所以選了幾題不太複雜的放在這一節以供參考，如果覺得有些困難，可以跳過它們。

下一題是 LIS 的變形加權版。

P-6-22. 文無第一武無第二 (@@) (*)

在一個群體中有 n 個成員，編號為 $1 \sim n$ ，第 i 個人的戰力為 $p[i]$ ，現在想要挑選一些人組成一個團隊擔任特別的任務。目前除了戰力外，還考量兩項能力值：程式力 $c[i]$ 與數學力 $m[i]$ ，俗話說「文無第一，武無第二」，如果挑選的人員中某一位的程式力與數學力都不如另外一位的話，他就會非常的沮喪而導致團隊總戰力受到傷害。你的目標是要挑出一個戰力總和最大的團隊，而且如果 i 與 j 都是被挑選的人的話，不允許 $c[i] < c[j]$ 且 $m[i] < m[j]$ 。

Time limit: 1 秒

輸入格式：第一行是正整數 n 。第二行有 n 個正整數，代表戰力 $p[1] \sim p[n]$ ，第三行有 n 個非負整數，代表程式力 $c[1] \sim c[n]$ ，第四行有 n 個非負整數，代表數學力 $m[1] \sim m[n]$ ，同行數字間以空白隔開。 n 不超過 $1e5$ ，戰力總和不超過 $1e9$ ，個程式力與數學力均不超過 $1e8$ 。

輸出：最大戰力總和。

範例一輸入： 5 2 4 1 3 7 1 2 3 3 4 6 7 4 5 1	範例一輸出： 15
範例二輸入： 5 1 2 3 4 9 4 5 4 5 8 3 6 4 5 7	範例二輸出： 9
範例一說明：挑選 $i=2\sim 5$ ，戰力總和 $4+1+3+7=15$ 。 範例二說明：挑選 $i=5$ ，戰力總和 9。	

如果我們將每個人的 $c[i]$ 看成 X 座標， $m[i]$ 看成 Y 座標， $p[i]$ 看成點的權重，這題就是平面上給 n 個點座標與權重，要找一些點的權重總和最大，條件是不允許某點的 XY 座標皆大於另外一點。如果我們從左往右看，就是一條左上往右下逐步遞減（非遞增）的序列，如果從右往左看，就是找一個權重最大的 LIS (longest increasing subsequence)，只是依照題意，等號是允許的（前面 LIS 那一題等號是不允許的）。

這一題比基本 LIS 難的原因主要不在排序，而是權重。在只需要考慮長度時，因為長度必然是 $1, 2, 3, \dots$ ，所以我們很容易的將長度 L 的最小可能結尾存在一個陣列 $last[L]$ 中，但在加權版本，我們要存的是以某數 t 結尾的最大權重，而且必須保持資料的單調性（結尾越大，權重越大），因此我們需要一個動態的資料結構。以下的範例程式是以 `map` 來存放這些資料，複雜度是 $O(n \log(n))$ 。

```
// max weight anti-chain, equality is allowed
// O(nlogn) DP
#include <bits/stdc++.h>
using namespace std;
#define N 100010
struct Member {
    int p, c, m;
} seq[N];
// large c first, breaking tie by small m
bool cmp(Member &x, Member &y) {
    if (x.c != y.c) return x.c > y.c;
    return x.m < y.m;
};
int main() {
    int n;
```

```

scanf("%d",&n);
for (int i=0;i<n;i++)
    scanf("%d",&seq[i].p);
for (int i=0;i<n;i++)
    scanf("%d",&seq[i].c);
for (int i=0;i<n;i++)
    scanf("%d",&seq[i].m);
sort(seq, seq+n, cmp);
int ans=0;
// find max weight nondecreasing sequence
map<int,int> M; // (m, P) largest P with ending m
M.insert({-1,0}); // as a boundary
for (int i=0; i<n; i++) {
    auto it = M.upper_bound(seq[i].m);
    it--; // seq[i] can append
    int w=it->second + seq[i].p;
    ans = max(ans,w);
    it = M.insert(it,{seq[i].m,w});
    if (it->second < w) { // when insert fail
        it->second = w;
    }
    // removing larger end but smaller weight
    it++;
    while (it!=M.end() && it->second<= w) {
        it=M.erase(it);
    }
}
printf("%d\n",ans);
auto it=M.end(); it--;
fprintf(stderr,"%d\n", it->second);
return 0;
}

```

下面是沒有利用單調性優化的版本，所以也不需要 map，複雜度是 $O(n^2)$ 。

```

// max weight anti-chain, equality is allowed
// O(nlogn) DP
#include <bits/stdc++.h>
using namespace std;
#define N 100010
struct Member {
    int p, c, m;
} seq[N];
// large c first, breaking tie by small m
bool cmp(Member &x, Member &y) {
    if (x.c != y.c) return x.c > y.c;
    return x.m < y.m;
};
int main() {
    int n;
    scanf("%d",&n);

```

```

for (int i=0;i<n;i++)
    scanf("%d",&seq[i].p);
for (int i=0;i<n;i++)
    scanf("%d",&seq[i].c);
for (int i=0;i<n;i++)
    scanf("%d",&seq[i].m);
sort(seq, seq+n, cmp);
int ans=0;
// find max weight nondecreasing sequence
map<int,int> M; // (m, P) largest P with ending m
M.insert({-1,0}); // as a boundary
for (int i=0; i<n; i++) {
    auto it = M.upper_bound(seq[i].m);
    it--; // seq[i] can append
    int w=it->second + seq[i].p;
    ans = max(ans,w);
    it = M.insert(it,{seq[i].m,w});
    if (it->second < w) { // when insert fail
        it->second = w;
    }
    // removing larger end but smaller weight
    it++;
    while (it!=M.end() && it->second<= w) {
        it=M.erase(it);
    }
}
printf("%d\n",ans);
auto it=M.end(); it--;
fprintf(stderr,"%d\n", it->second);
return 0;
}

```

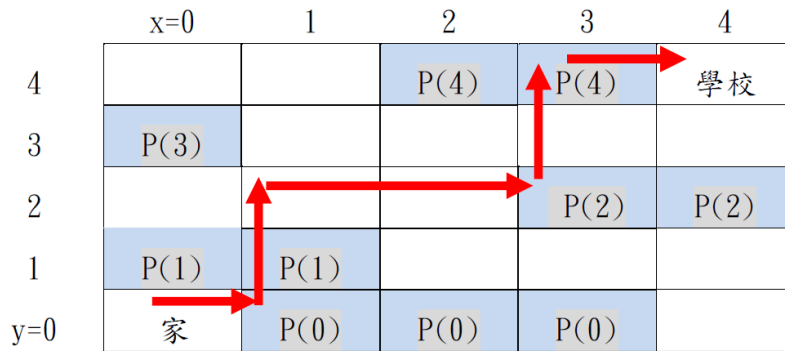
下面一題的程式其實非常簡單，只是需要一點思考，也需要使用動態的資料結構，我們把它列作習題，但給予提示。

Q-6-23. 直升機抓寶 (@@) (*) (105 高中全國賽)

小智的學校在天空之城，他每天開直升機上學，現在他想要規劃一條路徑以便在從家裡到學校的路上可以抓到最多的寶貝，請你寫個程式幫助他。

我們將學校與他家之間所有的位置劃分成 $N \times N$ 的格子，每個格子以坐標 (x, y) 表示，其中 x 代表水平距離， y 代表高度，而小智家的坐標在 $(0, 0)$ 的位置，學校在 $(N-1, N-1)$ 的位置，由於他的父母不准他上學途中貪玩繞路，直升機被設定成每次只能向前或向上一格，也就是說，如果從 (x, y) 到 $(x+1, y)$ 或 $(x, y+1)$ ，當然，他也不可以超過 $x < N$ 且 $y < N$ 範圍，否則會無法到達學校。

小智到學校的路途上一共有 N 隻寶貝，每隻寶貝可以補獲的範圍是某特定高度而水平座標在某連續區間的格子。明確的說，對於寶貝 $P(i)$ ， $0 \leq i < N$ ，要捕抓到 $P(i)$ ，小智必須經過下列座標之一： $\{(x, y) \mid S(i) \leq x \leq T(i) \text{ and } y=i\}$ ，其中所有的 $S(i)$ 與 $T(i)$ 都已經透過抓寶雷達得到資料了。



上圖是一個 $N=5$ 的例子，藍色區塊顯示可以捕抓到寶貝的地方，請注意，每一個寶貝都是在一個水平連續區間。紅線所顯示的路徑是一條合乎規定的路徑，因為他每一步都只有向右或向上，沿這一條路徑可以捕抓到四隻寶貝，是所有可能路徑中可以捕抓到寶貝數最多的。

Time limit: 1 秒

輸入格式：第一行是座標範圍的 N ，接下來 N 行，每一行有兩個整數 $S(i)$ 與 $T(i)$ ，依序是 $i=0, 1, \dots, N-1$ ，其中 $0 \leq S(i) \leq T(i) < N \leq 2.5e5$ 。

輸出：最多可能抓到的寶貝數。

範例一輸入： 5 1 3 0 1 3 4 0 0 2 3	範例一輸出： 4
範例二輸入： 2 1 1 0 0	範例二輸出： 1

如果我們從下往上一步一步看，令 $p[x]$ 是當時走到 x 座標 x 能抓到的最多寶貝數量，顯然 $p[]$ 是一個非遞減陣列。那麼其實我們要做的很簡單：在高度 i 時，讀取一個區間

$[s, t]$ ，對於 $p[s] \sim p[t]$ ，把數字增加 1；對於 $x > t$ ，如果 $p[x] < p[x-1]$ ，則 $p[x] = p[x-1]$ ，也就是做 prefix maximum 的動作。以下是這樣直接寫的程式，時間複雜度是 $O(n^2)$ ，這一題的重點就是如何優化，作法就列入(困難的)習題了，給一個提示： $p[]$ 是一個非遞減陣列，我們不一定要存 $p[]$ ，可以只存它上升的位置。

```
// touch as many intervals as possible by an increasing path
// naive method, O(N^2) method
#include <bits/stdc++.h>
#define N 250000
using namespace std;
int main() {
    int n, p[N]={0};
    scanf("%d", &n);
    for (int i=0; i<n; i++) {
        int s, t, j;
        scanf("%d%d", &s, &t);
        for (j=s; j<=t; j++)
            p[j]++;
        while (j<n && p[j]<p[j-1]) {
            p[j]=p[j-1];
            j++;
        }
    }
    printf("%d\n", p[n-1]);
    return 0;
}
```

下面這一題不需要特殊的資料結構，但是思考比較困難。

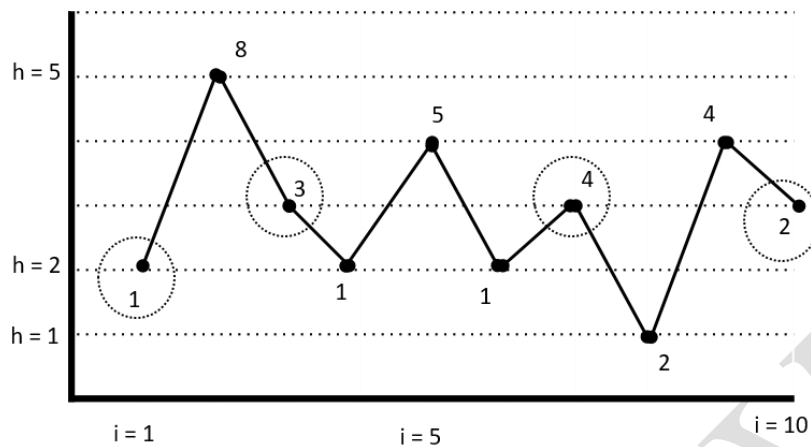
Q-6-24. 隔離採礦 (@@@) (108 全國高中賽)

外星採礦隊在烏邦圖星球已經探勘到某種礦物的礦脈，這條礦脈是一個筆直的直線，且一共有 n 個可以挖礦井的地點，這些地點就稱為可開採點。所有的可開採點由 1 到 n 依序編號，第 i 個可開採點的高度為 $h(i)$ 而開採價值為 $v(i)$ 。

現在希望能找出某些可開採點來實際進行鑽井挖礦，因為安全因素，任兩個礦井之間必須有另外一個開採點，其高度超過這兩個礦井。身為外星採礦隊的程式設計師，你的目標是計算出最大的開採價值總和。

下圖是一個 $n = 10$ 的例子，圖中各點的高度為 y 軸座標，而開採價值則標註於各點旁邊。如圖所示，各點的高度依序是 $h = (2, 5, 3, 2, 4, 2, 3, 1, 4, 3)$ ，而開採價值則依序是 $v = (1, 8, 3, 1, 5, 1, 4, 2, 4, 2)$ ，也就是說 $h(1)=2$ ， $h(2)=5$ ， \dots ， $h(10)=3$ ，而 $v(1)=1$ ， $v(2)=8$ ， \dots ， $v(10)=2$ 。在此

例中，最佳的開採方式是圖中被圓圈所圈起來的四個點，所能獲得的最大開採價值總和為 $1+3+4+2=10$ 。你可以看到，任兩個實際開採點之間都有一個高度更高的點。



Time limit: 1 秒

輸入格式：每筆測資的第一行是一個正整數 n ， $n \leq 1e6$ ，代表可開採點的數量。第二行是 n 個非負整數，依序代表每一點的高度 $h(1), h(2), \dots, h(n)$ ；第三行是 n 個正整數，依序代表每一點的開採價值 $v(1), v(2), \dots, v(n)$ 。每一行相鄰數字間以一空白間隔。每一點的高度不會超過 $1e9$ ，價值不會超過 $1e5$ 。

輸出：輸出為一整數，代表最大的開採價值總和。每筆測資的答案不超過 $2e9$ 。

範例一輸入： 10 2 5 3 2 4 2 3 1 4 3 1 8 3 1 5 1 4 2 4 2	範例一輸出： 10
範例二輸入： 5 3 3 3 3 3 5 3 2 3 4	範例二輸出： 5

這題有很多解法，最好的解法是 $O(n)$ 的 DP，但是轉移式比較不容易想，另外也有幾種 $O(n \log(n))$ 的解法。

下面這一題其實是赫赫有名的經典題 TSP (travelling salesperson problem)。

Q-6-25. 貨郎問題 (QQ)

有一個賣貨郎要旅行 n 個城市做生意並且回到他的家，由於路途遙遠，他希望走的路程總和越短越好，希望你幫他規劃最短路線。這 n 個城市由 $0 \sim n-1$ 編號，他的家在編號 m 的城市。對任意兩個城市 i 與 j ，已知兩者之間的距離是 $d[i][j]$ ，而且繞經第三地的路程絕對不會更短。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 n 與 m 。接下來 n 行每行 n 個非負整數是矩陣 $d[i][j]$ 的內容，順序由上而下，由左而右， i 從 $0 \sim n-1$ ， j 從 $0 \sim n-1$ 。 $n \leq 16$ ，假設 $d[i][j]=d[j][i]$ ， $d[i][i]=0$ ，且 $d[i][j] \leq 100$ 。

輸出：最短總路程。

<p>範例一輸入：</p> <pre>4 0 0 1 1 2 1 0 1 2 1 1 0 1 2 2 1 0</pre>	<p>範例一輸出：</p> <pre>5</pre>
<p>範例二輸入：</p> <pre>4 2 0 2 1 1 2 0 1 1 1 1 0 2 1 1 2 0</pre>	<p>範例二輸出：</p> <pre>4</pre>

範例一說明：走的順序與距離 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$ ，距離總和 $1+1+1+2=5$ 。

範例二說明：走的順序與距離 $0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0$ ，距離總和 $1+1+1+1=4$ 。

一個人要把 n 個城市都走一遍再回到起點，題目中說，對任兩個城市來說，繞經第三地並不會比較近，因此單純的可以看作是一個找排列的方式，這一題的 n 到 20， $20!$ 實在太大了，所以要找更快的方法。

這一題並不需要特殊的資料結構或演算法，所以未必是 APCS 不能考的範圍，我們給予提示，作為習題。因為要繞一圈，起點根本不重要，輸入資料中給起點只是唬人的把戲，我們可以假設 0 是起點。以狀態轉移來想想看，有哪些狀態呢？對於兩個相同起點不同的路徑（尚未走完），如果它們走過的點集合一樣，最後停留的點也一樣，那麼這兩條路徑就只需要一條比較短的就可以了，例如 $P=(0, 1, 3, 7, 4)$ 與 $Q=(0, 7, 3,$

1, 4)，他們經過的點集合都是 $\{0, 1, 3, 4, 7\}$ 而最後停留點都是 4，所以並不需要都留下來，也就是說，我們需要考慮的狀態是所有的 (s, p) ，其中 s 是所有的子集合，而 p 是所有可能的點，這樣的 DP 需要的複雜度還是指數，因為有 2^n 個子集合與 n 個可能的點，所以狀態數是 $O(n2^n)$ ，每一個轉移會牽涉 $O(n)$ 前置狀態，所以時間複雜度是 $O(n^2 \cdot 2^n)$ ，雖然是指數形式，但比 $O(n!)$ 要好很多。

Bangye.Wu

7. 基本圖論演算法

「BFS、DFS、與 Dijkstra 演算法之間有何差異？」

「BFS 像是小寶寶自己找父母，出生第一眼就到的就認做 parent；DFS 像是幫孩子找爹娘，最後結婚生下的才是你孩子的 parent；Dijkstra 則是長大後自己找乾爹乾媽，最有利的才是乾爹乾媽。」

「那麼，Prim 與 Dijkstra 演算法的差異呢？」

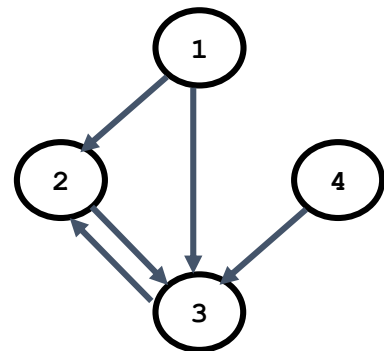
「認乾爹乾媽的依據稍有不同。Prim 只看乾爹乾媽對你有多好，Dijkstra 則除了對你多好之外，還有他從祖先繼承了多少會留給你的財產。」

這一章介紹圖 (graph)，圖是用來表達二元關係的一種方式，有很多重要的應用，也發展出很多特殊的演算法，這些演算法都是過去科學家研究發展出來的，也就是說，我們要了解它的話，需要靠學習與理解，很難只靠基本的演算法原則自己去想出來。圖在競賽程式中也扮演了很重要的腳色，但基於高中資訊領域的範圍與 APCS 的考試範圍，這份教材只會包含一些圖的基本演算法。本章內容包含：圖的基本觀念與名詞、儲存圖的資料結構、廣度優先搜尋、深度優先搜尋、以及 DAG 與拓撲順序，在本章最後我們則介紹了三個比較進階的技術，包括 Dijkstra 演算法、併查集、以及最小生成樹。至於在樹狀圖上做計算的相關問題則留在下一章。

7.1. 基本觀念與名詞

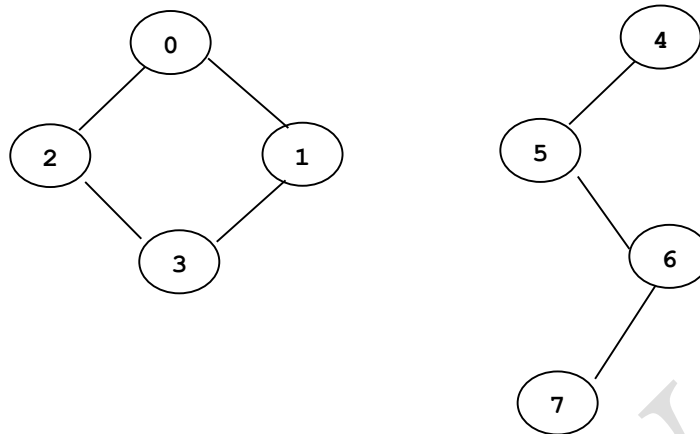
資料結構與演算法中稱的圖 (graph) 與我們日常生活中的圖 (地圖，圖形，藍圖) 不太一樣，這裡指的是圖論 (graph theory) 中的圖，而圖論算是離散數學中的一支。Graph 基本上是二元關係的一種表達方式，目的在研究一群事務中兩兩之間的關係。一個圖寫作 $G=(V, E)$ ，表示一個圖名字是 G ，由點的集合 V 與邊的集合 E 組成，其中 V 是點集合，也就是要研究的對象；而 E 是邊的集合，也就是誰與誰之間存在這種關係。

右圖是一個例子，圖中四個圓圈代表我們要研究的四個人，研究的關係是誰喜歡誰，因此，有一條線從 1 指到 2 表示 1 喜歡 2，記做 $(1, 2)$ 。同樣的，圖中還有另外 4 個邊 $(1, 3)$ 、 $(2, 3)$ 、 $(3, 2)$ 與 $(4, 3)$ ，分別代表 1 喜歡 3、2 喜歡 3、3 喜歡 2 以及 4 喜歡 3。圖中沒出現的邊，就表示該二者之間沒有這種關係，例如 1 沒有喜歡 4，2 沒有喜歡 1。



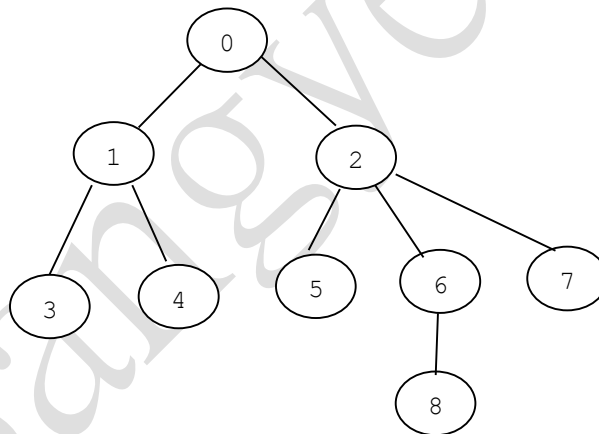
圖的術語很多，以下我們介紹一些基本術語，另外有一些在我們碰到的時候再介紹。

- 有向圖與無向圖(directed and undirected graph)：邊有方向性的圖是有向圖，通常以箭頭表示邊。有向圖表示的是非對稱的二元關係，如上圖的例子，1 喜歡 2 但 2 沒有喜歡 1，圖上有 $(1, 2)$ 但沒有 $(2, 1)$ ，有向圖上如果雙向關係都存在則需要用兩個邊表示，例如上圖中的 $(2, 3)$ 與 $(3, 2)$ 。無向圖的邊是沒有指明方向的，其實無向圖就是雙向圖，每一個邊都是雙向關係，無向圖表達的是一種對稱的關係，也就是對任意 a 與 b ， (a, b) 存在若且惟若 (b, a) 存在，或者說 (a, b) 與 (b, a) 是一樣的。
- 無加權圖與加權圖(unweighted and weighted graph)：圖上的點與邊都可以有權重，例如以點表示城市，邊代表道路的交通網圖，點的權重可能表示城市人口數，邊的權重可能表示路的長度。如果沒有說明，通常以 $G=(V, E, w)$ 表示一個邊上有權重的圖，其中 w 是一個邊的函數， $w(a, b)$ 代表邊 (a, b) 的權重。無加權圖就可以看成每個點(邊)的權重都是 1 的加權圖。
- 路徑與環路：從一個點 v_0 出發，經過某個邊 (v_0, v_1) 到達 v_1 ，再經由某個邊 (v_1, v_2) 到達 v_2 ，...，最後到達 v_k ，這樣的一個序列稱為路徑(path)，若任兩點之間最多只有一條邊(稱為 simple graph)，則我們可以用點的序列 $(v_0, v_1, v_2, \dots, v_k)$ 來代表這條路徑。一個路徑上如果沒有經過重複的點，就稱為簡單路徑(simple path)，若沒有特殊指明，通常路徑指的是簡單路徑。頭尾是相同的點的路徑則稱為環路(cycle/circuit)，也就是從某點出發經過若干點之後回到出發點。在邊無加權的圖上，路徑的長度指的是經過的邊的數量，以上例來說，路徑長度是 k ；在邊有加權的狀況下，路徑的長度是經過邊的權重總和，這時候邊的權重往往也稱為長度。兩點之間可能有多條(或沒有)路徑，長度最短的路徑稱為最短路徑(shortest path)，最短路徑的長度通常稱為距離(distance)或強調稱為最短距離。在無加權圖中，每走一條邊可以比喻為走一步，最短距離就是最少走幾步可以到達目的點。
- 無向連通圖與連通區塊：一個無向圖如果任兩點之間都存在路徑相通，則稱為連通圖(connected graph)，否則稱為非連通圖(disconnected graph)。一個圖中一群彼此互相有路徑相通的點所形成的區塊稱為一個連通區塊(connected component)，但不被包含在其他連通區塊的才能稱為連通區塊。下圖是一個例子， $\{0, 1, 2, 3\}$ 形成一個連通區塊， $\{4, 5, 6, 7\}$ 是另外一個連通區塊。但 $\{0, 1, 2\}$ 不算。從圖形上看，連在一起的就是屬於一個連通區塊，以道路交通圖為例，則離島之間是不連通的，每個離島可能屬於單獨的連通區塊。



A disconnected graph with two **connected components**.

- 一個連通而無環路的圖稱為樹 (或樹狀圖)，如果是簡單圖，則樹的邊數必然等於點數減一。樹中如有一點被指定為根 (root)，則稱為有根樹，否則是無根樹。有根樹通常把根畫在最上方像一個家族圖，如下圖是一個有根樹，其中 0 是根。除了根以外，每個點的上方 (通往根的路徑上的第一個點) 一定恰有一點，這個點稱為他的



A rooted tree with 9 vertices and 8

parent，例如 4 與 3 的 parent 是 1；5, 6 與 7 的 parent 是 2；而 1 與 2 的 parent 是 0 (根)。若 a 是 b 的 parent，則稱 b 是 a 的孩子 (child)。只要指出每一個點的 parent 就可以惟一決定一棵樹。樹的相關問題很多，我們留在下一章介紹，但是這一章裡面也會碰到一些與樹有關的術語。

- 有向圖的強連通：在無向圖上，一個圖只要連在一起就是連通圖，任兩點之間皆有路徑相連。但在有向圖上比較複雜，因為有向圖的路徑必須依照箭頭方向前進，所以連在一塊的不一定有路徑可以到達。若對任意兩點 a 與 b ，圖上皆存在由 a 到 b 的路徑 (以及由 b 到 a 的路徑)，則稱為強連通圖 (strongly connected graph)。

- 點的分支度 (degree) 與鄰居：兩個點如有邊相鄰則稱為鄰居，無向圖上是互為鄰居，有向圖則要區分 in-neighbor 與 out-neighbor。鄰居的個數則為分支度 (degree)，也就是該點所接邊的個數 (假設是簡單圖)。同樣的，在有向圖上 degree 要分為 in-degree 與 out-degree。

一個圖 $G=(V,E)$ ，習慣上用 n 代表點數， m 代表邊數。圖的問題的輸入通常包含點與邊，所以輸入資料的大小是 $O(n+m)$ ，在表達演算法複雜度時，往往也需要表示成 n 與 m 的函數。我們知道邊數與點數是有一定的關係，在大部分狀況下 (simple graph)， m 不會超過 $O(n^2)$ ，所以有的時候也只表示成 n 的函數，但是表示成 n 與 m 的函數會更精準，特別是很多問題所要處理的圖的邊數 m 只在 $O(n)$ 的大小，這時把 m 列為複雜度參數就很重要了。

在分析圖的演算法時，常常涉及點與邊的關係，以下是一個簡單而常用的性質，也被稱為「握手定理」：

在無向圖中，各點的 degree 總和等於 $2m$ ；在有向圖中，各點的 in-degree 總和與 out-degree 總和，都等於 m 。

握手定理名稱的由來是這定理的無向圖版本可以描述成：在一場聚會中，每個人握手次數的總和是兩倍的握手發生次數。這定理很容易理解，因為一根邊會貢獻兩個 degree，只是不一定貢獻給哪個點。而在有向圖時，一根邊一定貢獻出一個 in-degree 與一個 out-degree。

7.2. 資料結構與基本演算法

在這一節中我們介紹處理圖的基本資料結構與幾個基本演算法。

7.2.1. 資料結構

要進行圖的資料計算，第一件事就是必須要能夠存取圖，也就是要有儲存圖的資料結構。常用的資料結構有兩種：鄰接矩陣 (adjacency matrix) 與鄰接串列 (adjacency list)。圖在處理過程中，最基本上操作就是找出某個點的鄰居，使用不同的資料結構，程式就會不同的寫法。

一個圖 $G=(V, E)$ ，習慣上用 n 代表點數， m 代表邊數，而將點以 $0\sim n-1$ 編號 (或是 $1\sim n$)。所謂鄰接矩陣就是一個 $n\times n$ 的二維陣列 $A[i][j]$ ，其中 $A[i][j]=1$ 代表有一個邊 (i, j) ，若 $A[i][j]=0$ 則表示該邊不存在。在無向圖時，此矩陣必然是對稱矩陣，也就是對所有 i 與 j ， $A[i][j]=A[j][i]$ ；如果是有向圖，則不一定。至於對角線上 $A[i][i]$ 則要看應用而定，在某些問題中， $A[i][i]$ 代表自迴圈 (self loop) 的邊，也就是由 i 到 i 的邊。

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

對於加權圖 $G=(V, E, w)$ ，我們可以將 $A[i][j]$ 存 (i, j) 邊上的權重 $w(i, j)$ ，若邊的權重是非正的，可以用 0 來表示邊不存在，但更常見的時候是以無窮大來表示，至於要以多大數字表示無窮大則看題目需要。

用鄰接矩陣來存圖最大的好處是簡單方便，要找出點 v 的 out-neighbor 就直接掃描列，找出那些 $A[v][i]=1$ 的 i ；若是要找 in-neighbor 就掃描行，找出那些 $A[i][v]=1$ 的 i 。

鄰接矩陣最大的麻煩是空間與時間效率。一個 n 個點的簡單圖，鄰接矩陣所需要的空間是 $O(n^2)$ 。另一方面，邊的個數 m 上限是 $O(n^2)$ ，如果邊數達到 $O(n^2)$ 我們通常稱為 dense graph，邊數是 $O(n)$ 的稱為 sparse graph。如果要處理的是 dense graph，那使用鄰接矩陣也是剛好。但是許多應用要處理的是 sparse graph，也就是說，如果用鄰接矩陣來存，矩陣裡面大部分都是 0，這顯然很沒有效率，因為資訊的量只有 $O(n+m)=O(n)$ 。這就是另外一種資料結構「鄰接串列」的優勢所在。

鄰接串列的概念很簡單：每個點把它的鄰居當成一個串列來存起來。這樣在有向圖中，鄰居的總數就是邊數 m ，在無向圖中，鄰居的總數就是 $2m$ (因為每個邊的兩端互為鄰居)。實作上呢？很多資料結構的書都是教用指標來做鏈結串列，將鄰居放進此串列中，但是在程式競賽與考試中，我們不需要這麼麻煩，我們可以使用 C++ STL 中的 vector，因為 vector 是長度可變的陣列，所以每一個點用一個 vector 來存它的鄰居就行了。

以下是範例的程式，讀入點數 n 與邊數 m ，然後讀取 m 個邊。這個範例程式只是要示範如何儲存圖，我們只把每個點的鄰居輸出並且計算一下 degree。第一支程式是示範鄰接矩陣，這裡假設是有向圖：

```
//ch7 demo adjacency matrix
#include <bits/stdc++.h>
using namespace std;
#define N 20
int main() {
    int adj[N][N]={0};
```

```

int n, m;
scanf("%d%d", &n, &m);
for (int i=0; i<m; i++) {
    int from, to;
    scanf("%d%d", &from, &to);
    adj[from][to]=1;
}
printf("adjacency matrix:\n");
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++)
        printf("%d ", adj[i][j]);
    printf("\n");
}
printf("deg = ");
int deg[N]={0};
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++)
        deg[i] += adj[i][j];
    printf("%d ", deg[i]);
}
printf("\n");
return 0;
}

```

第二支是示範鄰接串列：

```

//ch7 demo adjacency list
#include <bits/stdc++.h>
using namespace std;
#define N 20
int main() {
    vector<int> adj[N]; // adj[i]: neighbors of vertex i
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i=0; i<m; i++) {
        int from, to;
        scanf("%d%d", &from, &to);
        adj[from].push_back(to);
    }
    printf("adjacency list:\n");
    for (int i=0; i<n; i++) {
        printf("%d: ", i);
        for (int v: adj[i])
            printf("%d ", v);
        printf("\n");
    }
    printf("deg = ");
    for (int i=0; i<n; i++)
        printf("%d ", (int)adj[i].size());
    printf("\n");
    return 0;
}

```

```
}

```

不使用 `vector` 的鄰接串列做法 (*)

如果不使用 STL 的 `vector` 可以存鄰接串列嗎？當然是可以，事實上也不至於困難到考試或比賽時寫不出來，但還是使用 `vector` 最簡單，所以以下的說明只提供參考，並不建議用這個方法。

不使用 STL 物件時，一個做法就是自己用 `malloc()` 分配記憶體。給每一個點宣告一個整數指標，算一算每個點有多少鄰居就給他多少記憶體，或者先給一部分空間，當資料輸入多到不夠存時，在重新 `malloc()` 記憶體並將資料複製到新的位置，要留意的是不要每次只多給一個記憶體，通常的做法是不夠的時候就給兩倍，這樣根據均攤分析，每一個資料被複製搬移的時間只需要 $O(1)$ ，事實上這麼做差不多就是寫一個 `vector` 的 `push_back()`。

另外一個做法是宣告一個結構 (`struct`) 的陣列，結構中含兩個欄位放兩端點，如果邊上有加權就在加一個欄位。把所有的邊全放到這個陣列之後，呼叫一個排序讓這些邊依照第一端點排序，目的是讓相同端點的邊放在一起。接著我們對每一個點 `v`，計算以 `v` 為端點的邊在此陣列中的開始位置 `start[v]`，這就完成前置的準備工作了。當我們要找 `v` 的鄰居時，就可以用

```
for (i=start[v]; i<start[v+1]; i++)
```

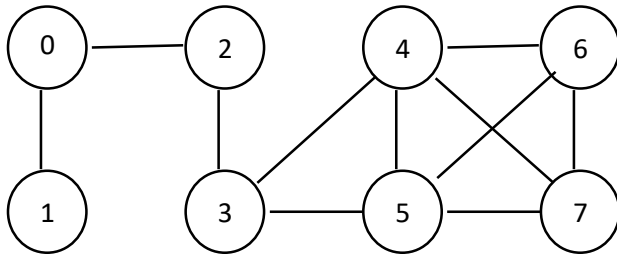
去找到 `v` 的鄰居。

這兩個方法只有在只能使用 C 而不能使用 C++ 的時機 (環境) 下，才建議這麼做，畢竟還是比較花費時間。當然，在某些狀況下，可能需要照著教科書用指標做鏈結串列，那就是另外一回事了。

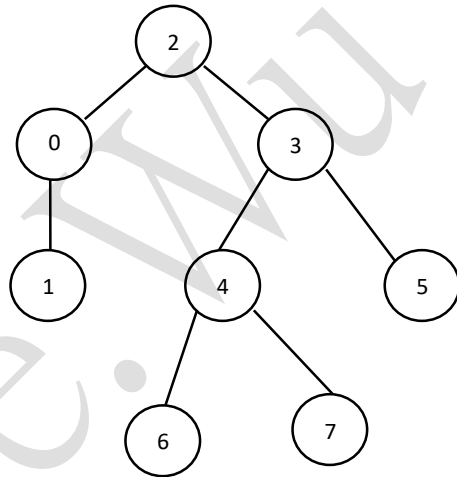
7.2.2. BFS

圖的搜尋是圖的基本問題：「輸入一圖 $G=(V,E)$ 以及其中一點 `s`，要從 `s` 作為起點探索此圖，或者簡單的說，請問從 `s` 出發的話，可以到達哪些點。」基本有兩種常用的演算法，這兩種方法各有它們的特殊應用，在這一小節先介紹廣度優先搜尋 (Breadth-First Search, BFS)。(對了，英文要念得清楚些，不然很容易聽成麵包優先搜尋。)

在看 BFS 的演算法前，我們先以一個例子來說明算法要做的事情，以下是一個無向圖。



假設我們要以 $s=2$ 為起點以 BFS 探索這個圖。我們會先找出距離 s 一步的點，然後找兩步的點，...，一直到所有 s 能走到的點都找完為止。過程中我們將 s 比喻成一個家族的第 0 代祖先，與 s 距離 1 步的點就是第 1 代，距離 2 步的就是第 2 代，...。最後建出一個像類似家族世系圖的結果，如右圖。這個圖是一個樹狀圖，而且由 BFS 走出來的，我們稱它是一個 BFS tree。BFS tree 是一層一層的，每往下一層與起點 s 的距離就增加 1 步。每一個點會與上一層的某個點相連，這個點稱為它的 parent，在 BFS tree 中，每個點的 parent 就是第一個發現它的點。



在探訪第 i 代的某個點 u 時，例如 $u=4$ ，我們檢查 u 的所有鄰居 $\{3, 5, 6, 7\}$ ，這些鄰居有可能是第 $i-1$ 代 (如 3)、第 i 代 (如 5)、或是第 $i+1$ 代 (6 與 7)，所以我們必須加以區分，區分的方法是記錄每個點目前是否被發現過，當我們探訪完所有第 $i-1$ 代的點時，所有第 i 代與更早的點都一定已經被發現過，所以 u 的鄰居中未曾被發現的就是所要的第 $i+1$ 代的點。有了以上的認知，我們可以來說明 BFS 的演算法了，以下是 BFS 的演算法。對每一個點 v ，我們建立下列資料：

- $visit[v]$ 用來表示 v 點是否以被發現。
- $v.d$ 是距離起點 s 的距離。
- $v.p$ 是紀錄 v 的 parent，也就是發現它的點。

因為一個點可能會發現多個點，但我們必須依照發現的順序逐一探訪，所以我們準備一個 FIFO (先進先出) 的佇列 Q ，初始的時候只有起點 s 放入 Q 中，程式執行過程中，被發現的點就會被放入 Q 中，程式會執行到 Q 中沒有點為止。每一回合從 Q 中取出一點 u 來加以探訪，探訪就是檢查它的所有鄰居，其中尚未被發現的，就設定相關資料並放入 Q 中；如果是已經被發現的，就不予理會。

```

初始化對所有點  $v$ ,  $visit[v]=false$ ;
//  $v.d$  是  $s$  到  $v$  的距離,  $v.p$  是  $v$  在 BFS tree 上的 parent
BFS( $G, s$ ) {
    初始化一個 FIFO 的 queue  $Q$  只含  $s$ ;
     $s.d = 0$ ;  $visit[s] = true$ ;  $s.p = -1$ ;
    while ( $Q$  not empty) do
         $Q$  中移出一點  $u$ ;
        for each  $v \in adj[u]$  do
            if ( $visit[v] == false$ ) then
                 $visit[v] = true$ ;
                 $v.d = u.d + 1$ ;
                 $v.p = u$ ; // 第一個發現  $v$  的就是  $v$  的 parent
                將  $v$  加入  $Q$  中;
            end if
        end for
    end while
}

```

演算法其實並不困難，實作程式要搭配儲存圖的資料結構，因為每一個點最多只會進入 Q 中一次，時間複雜度不會超過 $O(n^2)$ 。如果以鄰接串列來儲存，複雜度是 $O(n+m)$ ，因為每個點從 Q 中出來時，做的事情是檢查它的所有鄰居，所以花的時間是它的鄰居數，也就是它的 degree，而根據握手定理，所有點的 degree 總和就是 $O(m)$ 。

BFS 通常有下列用途：

- 計算某個點出發可以到達的點。
- 在無加權圖中，計算一個點到其他點的距離，或是找兩點的最短路徑。
- 計算無向圖的 connected component。從一個點出發可以拜訪到該點所在的連通區塊的所有點，若對每一個點當起點進行 BFS，則可以找到所有連通區塊，但要注意應略過前面的點已經被拜訪過的點。

以下先看一個簡單例題，其他應用放在後面的例題與習題中。

P-7-1. 探索距離

輸入一個有向圖 G 與一個起點 s ，請計算由 s 出發可以到達的點數 (不包含 s)，並且計算這些可以到達的點與 s 的距離和，假設每個邊的長度均為 1。兩點之間可能有多個邊，邊的起點與終點未必不同。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表圖的點數與邊數，圖的點是以 $0 \sim n-1$ 編號，第二行是 s 的編號，接下來有 m 行，每一行兩個整數 a 與 b 代表一個邊 (a, b) 。
 n 不超過 100， m 不超過 4000。

輸出：第一行輸出可以到達的點數，第二行輸出與 s 的距離總和。

範例一輸入： 7 6 1 5 1 1 3 1 4 2 3 4 6 6 0	範例一輸出： 4 7
範例二輸入： 3 2 0 1 2 2 1	範例二輸出： 0 0

這一題就是要在有向圖中從一點 s 做 BFS 就可以計算可以到達的點，以及 s 到這些點的距離。因為點數不多，我們同時示範鄰接矩陣與鄰接串列的做法，以下先看矩陣。這個程式就直接照 BFS 的演算法寫，沒有太多需要進一步說明的，因為適用鄰接矩陣儲存的，所以在找鄰居時，必須跑一個迴圈檢查所有的點。

```
// P-7-1, using matrix
#include <bits/stdc++.h>
using namespace std;
#define N 102
int n, m, s;
int adj[N][N]={0};
int d[N]; // distance

void bfs(int s) {
    bool visit[n] = {false};
    queue<int> Q;
    Q.push(s);
    visit[s]=true, d[s]=0;
    int cnt=0, totald=0;
    while (!Q.empty()) {
        int v=Q.front();
        Q.pop();
        for (int i=0;i<n;i++) {
```

```

        if (adj[v][i] && !visit[i]) {
            d[i]=d[v]+1;
            visit[i]=true;
            Q.push(i);
            cnt++; // num of reachable
            totald+=d[i]; // total distance
        }
    }
}
printf("%d\n%d\n", cnt, totald);
return;
}

int main() {
    scanf("%d%d%d", &n, &m, &s);
    for (int i=0; i<m; i++) {
        int from, to;
        scanf("%d%d", &from, &to);
        adj[from][to]=1;
    }
    bfs(s);
    return 0;
}

```

再來看鄰接串列的寫法，惟一的差別只在圖的儲存方式，因為用 vector 來存，所以在找鄰居時可以很簡單的使用 range-based 的寫法

```
for (int u: adj[v]) { }
```

請注意這一題的輸入雖然兩點之間可能有多個邊而且可能有自己走到自己的邊（自迴圈），但對 BFS 完全沒有影響，所以程式中沒有去判別這些狀況。

```

// P-7-1, using adjacency list
#include <bits/stdc++.h>
using namespace std;
#define N 102
int n, m, s;
vector<int> adj[N];
int d[N]; // distance

void bfs(int s) {
    bool visit[n] = {false};
    queue<int> Q;
    Q.push(s);
    visit[s]=true, d[s]=0;
    int cnt=0, totald=0;
    while (!Q.empty()) {
        int v=Q.front();
        Q.pop();
        for (int u: adj[v]) {

```



```

        if (!visit[u]) {
            d[u] = d[v]+1;
            visit[u] = true;
            Q.push(u);
            cnt++; // num of reachable
            totald += d[u]; // total distance
        }
    }
}
printf("%d\n%d\n", cnt, totald);
return;
}

int main() {
    scanf("%d%d%d", &n, &m, &s);
    for (int i=0; i<m; i++) {
        int from, to;
        scanf("%d%d", &from, &to);
        adj[from].push_back(to);
    }
    bfs(s);
    return 0;
}

```

BFS 容易寫錯的地方是 `visit[]` 沒有設定好或是忘了檢查，那有可能會造成一個點被重複的放進 `queue` 中，而在執行時發生錯誤 (例如無窮迴圈)。

7.2.3. DFS

除了 BFS 之外，另外一個常用的圖的搜尋演算法是深度優先搜尋 (DFS, Depth-First Search)。BFS 像水面掀起的水波，從中心點往外一圈一圈的擴散，DFS 通常是以遞迴的概念來說明：「先從一個鄰居出發探索所有可以探索的點，如果還有剩下，則在從下一個鄰居出發，重複此步驟，直到所有鄰居嘗試完畢。」遞迴的演算法通常比較簡潔但不易了解，DFS 也不例外。

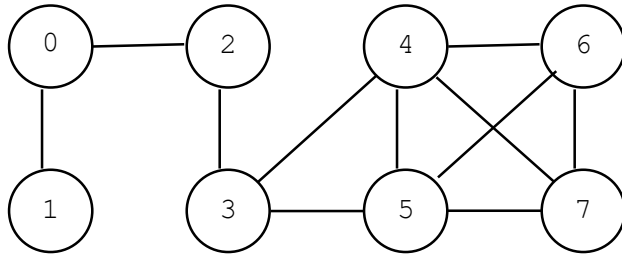
```

初始化對所有點 v，visit[v]=false;
DFS(s) {
    visit[s] = true;
    for each v ∈ adj[s] do
        if (visit[v] == false) then
            v.p = s; // 呼叫 v 的點就是 v 的 parent
            DFS(v);
        end if
    end for
    return;
}

```



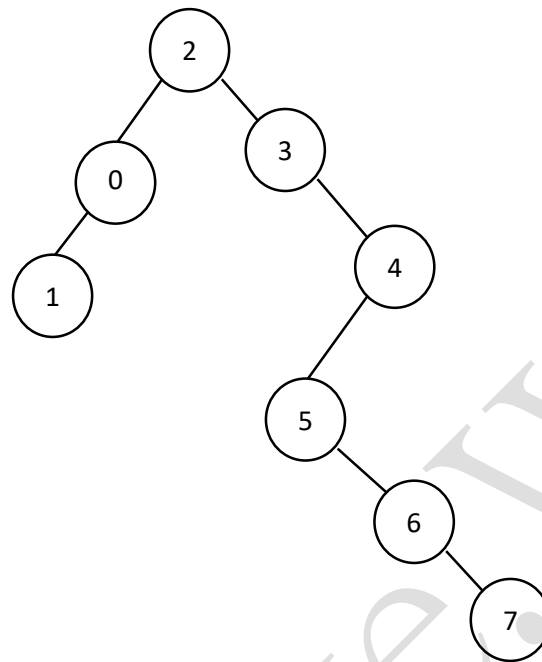
我們實際看一下 DFS 在下面這個無向圖上的運作狀況，會比較了解此演算法。假設一樣從 2 出發。



演算法運作過程如下 (假設鄰居的檢查順序依照編號)：

- A. DFS (2)：2 有兩個鄰居 {0, 3}，而且都未曾被拜訪，先去 0，留著 3 未檢查；
- B. DFS (0)：0 有兩個鄰居 {1, 2}，先去 1；
- C. DFS (1)：1 只有個鄰居 {0}，但是 0 已經被拜訪過，所以 return 回到 B；
B 是從 0 探訪它的鄰居，本來先去 1，現在從 1 回來了，所以檢查下一個鄰居 2，但 2 是已經被拜訪過的，所以也結束，return 回到 A；
A 是從 2 探訪它的鄰居，本來先去 0，現在從 0 回來了，所以檢查下一個鄰居 3，3 尚未被拜訪，所以去 3；
- D. DFS (3)：3 有 3 個鄰居 {2, 4, 5}，先檢查 2，被拜訪過；再檢查 4，沒拜訪過，所以去 4 (留著 5 還沒檢查)；
- E. DFS (4)：4 有 4 個鄰居 {3, 5, 6, 7}，先檢查 3，被拜訪過；再檢查 5，沒拜訪過，所以去 5 (留著 6, 7 還沒檢查)；
- F. DFS (5)：5 有 4 個鄰居 {3, 4, 6, 7}，先檢查 3 與 4，都被拜訪過；再檢查 6，沒拜訪過，所以去 6 (留著 7 還沒檢查)；
- G. DFS (6)：沒被拜訪過的鄰居只有 7；
- H. DFS (7)：沒有沒被拜訪過的鄰居，所以 return 回到 G (6 號點)；
G 沒有剩下沒拜訪過的鄰居，所以 return 回到 F (5 號點)；
F 本來留著 7 尚未檢查先去 6，所以 6 回來時檢查 7，但此時 7 已經被拜訪過了！！所以也沒有剩下沒拜訪過的鄰居，也 return 回到 E (4 號點)；
E 是 4 號點先去 5，留著 6 與 7 尚未檢查，所以從 5 號點回來會去檢查 6 與 7，但此時兩點都已經被拜訪過了，所以就 return 回到 D (3 號點)；
類似地，3 號回到 2 號，然後就結束了整個探訪的過程。

如果我們把每個點被誰呼叫當作它的 parent，然後把每點與它的 parent 連起來，一樣可以得到一棵樹，這棵樹是由 DFS 走出來的，所以就稱為 DFS tree (如下圖)，它可以讓我們了解 DFS 運作的過程。



回顧一下 DFS 的過程：從 2 出發，拜訪 0，拜訪 1，回到 0，回到 2，拜訪 3，拜訪 4，拜訪 5，拜訪 6，拜訪 7，回到 6，回到 5，回到 4，回到 3，回到 2，結束。所以它的行為是如果還能 (更深入) 往前走，就深入往前，直到沒有沒拜訪過的，就回頭。所以深度優先搜尋的名字是這麼來的。

- DFS 通常寫成遞迴形式，所以比較好寫，但有遞迴的缺點，在某些系統環境上，遞迴的深度會受到限制。
- DFS 與 BFS 一樣可以計算到達的點與無向圖的連通區塊，但是不能算起點到各點的距離。
- DFS 在進階的圖論演算法中有很多重要的應用，但大多超過了這份教材的難度，所以沒有在這教材中說明。

以下是一個簡單的例題來示範 DFS 的寫法。

P-7-2. 開車蒐集寶物

參加一個蒐集寶物的遊戲，你拿到一個地圖，地圖上有 n 個藏寶點，每個藏寶點有若干價值的寶物，由於你的團隊是最頂尖的，只要能到達藏寶點一定可以取得該藏寶點的寶藏。從地圖上看得到一共有 m 條道路，每條道路連接兩個藏寶點，而且每條道路

都是雙向可以通行的。在遊戲的一開始，你可以要求直升機將你的團隊運送到某個藏寶點，而且你可以獲得一部車與充足的油料，但是直升機的載送只有一次，所以你必须決定要從哪裡開始才可以獲得最多的寶藏總價值。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表藏寶地點數與道路數，地點是以 $0 \sim n-1$ 編號，第二行 n 個非負整數，依序是每一個地點的寶藏價值，每個地點的寶藏價值不超過 100。接下來有 m 行，每一行兩個整數 a 與 b 代表一個道路連接的兩個地點編號。 n 不超過 $5e4$ ， m 不超過 $5e5$ 。兩點之間可能有多條道路，有些道路的兩端點可能是同一地點。

輸出：最大可以獲得的寶藏總價值。

範例一輸入： 7 6 5 2 4 2 1 1 8 5 1 1 3 1 4 2 0 2 0 3 3	範例一輸出： 9
範例二輸入： 3 2 2 1 5 1 0 0 1	範例二輸出： 5

範例一說明：降落在 0 可以獲得 0 與 2 的寶藏，價值為 $5+4=9$ 。

範例二說明：降落在 2 可獲得價值 5。

輸入的是一個無向圖，每個點有個權重，要找出最大權重的連通區塊，所以用 DFS 或 BFS 都可以，這一題我們用來示範 DFS，所以下面是 DFS 的寫法。這一題的點數很大，所以必須使用連接串列的方式來存圖，主程式主要在讀取圖的資料，因為是無向圖，所以讀到一個邊 (u, v) 時，必須兩點互相加為鄰居。DFS 函式的部分可以看得出遞迴的寫法真的要比 BFS 簡潔，這裡我們要找的是連通區塊的權重總和，所以，對於每一次 DFS，我們讓他回傳可以到達點的權重和就可以了，因為有判斷 `visit`，所以一個點不會被走訪兩次，權重也就不會被重複計算了。

```

// P-7-2, max weight component, adjacency list
#include <bits/stdc++.h>
using namespace std;
#define N 50100
int n, m, s;
vector<int> adj[N];
int w[N]; // weight of vertices
bool visit[N] = {false};

// return the total weight can be reached from s
int dfs(int s) {
    int weight=w[s];
    visit[s]=true;
    for (int u: adj[s]) {
        if (!visit[u])
            weight += dfs(u); // total weight
    }
    return weight;
}

int main() {
    scanf("%d%d", &n, &m);
    int total=0;
    for (int i=0;i<n;i++) {
        scanf("%d", &w[i]);
        total += w[i];
    }
    for (int i=0; i<m; i++) { // each edge
        int u,v;
        scanf("%d%d", &u, &v);
        adj[u].push_back(v); // v is a neighbor of u
        adj[v].push_back(u); // u also a neighbor of v
    }
    int maxw=0; // max total weight
    for (int v=0; v<n; v++) {
        // search the weight of component of v
        if (!visit[v])
            maxw=max(maxw,dfs(v));
    }
    printf("%d\n", maxw);
    // for checking
    fprintf(stderr, "ans=%d, total=%d\n",maxw, total);
    return 0;
}

```

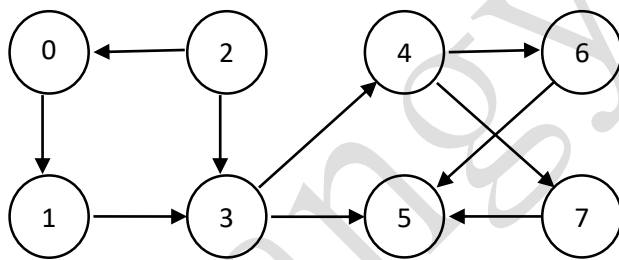
這一題當然也可以用 BFS 來作，有興趣的人可以自己練習試試，範例程式中有一支 BSF 的作法，因為只是標準的 BFS，這裡就不多說明了。

7.2.4. DAG 與 topological sort

DAG (directed acyclic graph) 是指一種沒有環路的有向圖，DAG 是應用很多的一個圖的類別，我們在上一章動態規劃中簡單說明過 DAG 可以用來描述狀態轉移，其實不只是 DP 的狀態，在很多前置關係的問題上都有類似的情形，例如一件計畫分成很多工作，某些工作必須在某些工作完成之後才能進行。我們可以將每個工作看成一個點，若 u 是 v 的前置工作，就在圖上加入一個 (u, v) 有向邊，這樣建立出來的圖必須是個 DAG，否則就會產生雞生蛋蛋生雞的矛盾而有某些工作無法完成。

對於一個 DAG，我們最常須要找的就是一個拓樸順序 (topological sort)，所謂拓樸順序是將所有的點排成一個序列，使得所有的有向邊都是從前往後，而沒有從後往前的邊。拓樸順序顯然不惟一，但通常只要找出其中一個就可以了。

以下圖為例來找一個拓樸順序，因為前置關係是有遞移性的，前置的前置也必須是前置，我們可以看得出來，2 一定要放在第一個，相同的 5 一定是最後一個。因為 3 要放在右邊所有點前，所以最前面的四個點一定是 $(2, 0, 1, 3)$ 。置於右邊的四個點，因為 4 必須在 6 與 7 之前，所以 3 後面只能接 4，但 6 與 7 不一定誰先誰後，所以我們知道 $(2, 0, 1, 3, 4, 6, 7, 5)$ 是一個拓樸順序，而 $(2, 0, 1, 3, 4, 7, 6, 5)$ 也是一個。



我們的重點要放在如何計算一個拓樸順序，常用的方法有兩個。其中一個是利用 DFS，假如在 DFS 時，對每一個點在完成探訪時 (return 之前)，將該點的編號輸出，那麼將此輸出順序反轉就可以得到一個拓樸順序。證明不是很難，簡單的說，在執行 DFS (v) 時，所有 v 可以到達的點都是應該排在 v 之後的，而 DFS 會把這些點都拜訪完畢後才結束 v 的探訪。

DFS 的複雜度線性時間，所以用 DFS 來做拓樸順序應該很完美，但是它是遞迴的，很多時候我們不希望使用遞迴，例如 DP 的過程，因為不想 (或是環境不允許) 使用遞迴，所以才要找拓樸順序，所以我們需要一個非遞迴的演算法。這個演算法我們在 P-3-1 (樹的高度) 時介紹過了，當時是用在找樹的一個 bottom-up sequence，事實上樹在指定方向 (例如由下而上) 後就是一個簡化的 DAG。以下我們看對任意 DAG 都可以找出拓樸順序的演算法。

這個方法很簡單，它的原理就是：只要是 $\text{in-degree}=0$ 的點，也就是沒有箭頭指向它的點，都是可以出列的點。以下是一個使用 adjacency list 的範例程式，第 6 行宣告一個整數陣列存每個點的 in-degree ，在輸入每一個邊的時候，於第 13 行計入各點的 in-degree 。第 15 行的陣列 `topo[]` 以及兩個變數 `head` 與 `tail` 是自製的佇列，因為我們想要保留拓撲順序，所以使用 STL 的 `queue` 反而不便。第 17~19 行的迴圈將 in-degree 為 0 的點放入佇列 (`topo[]`) 中，然後跑一個 `while` 迴圈直到佇列中沒有元素為止。在迴圈中，我們每次從佇列前端拿出一點 `v`，將 `v` 指向的每個點 `u` 的 in-degree 減一，若減一後降為 0，則將其放入佇列中。如果這是一個 DAG，`while` 迴圈跑完時，所有的點一定都處理過，否則便是有迴圈。程式的時間複雜度是 $O(n+m)$ ，因為每個點最多進入佇列一次，離開佇列時所花的時間是他的 out-degree ，而所有點的 out-degree 總和就是邊數。

```

00 //ch7 topological sort O(m+n), non-recursive
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 20
04 int main() {
05     vector<int> adj[N]; // adjacency list
06     int indeg[N]={0};
07     int n, m;
08     scanf("%d%d", &n, &m);
09     for (int i=0; i<m; i++) {
10         int from, to;
11         scanf("%d%d", &from, &to);
12         adj[from].push_back(to);
13         indeg[to]++;
14     }
15     int topo[N], head=0, tail=0; //queue
16     // push all indeg==0
17     for (int i=0; i<n; i++)
18         if (indeg[i]==0)
19             topo[tail++]=i; // push
20     while (head<tail) { // until queue is empty
21         int v=topo[head++]; // pop from queue
22         for (int u: adj[v]) {
23             if (--indeg[u] ==0)
24                 topo[tail++] = u;
25         }
26     }
27     // output
28     if (tail<n) printf("No a dag\n");
29     else {
30         for (int i=0; i<n; i++)
31             printf("%d ", topo[i]);
32         printf("\n");
33     }
34     return 0;
35 }

```

7.3. 例題與習題

這一節裡面我們提出幾個例題與習題，由於圖論演算法的題目雖多，但大部分超過 APCS 的範圍，所以這裡只有一些比較簡單的題目。有一些一維與二維陣列的題目其實用圖的概念會更容易了解，我們也放了一些這樣的題目。第一題是一個很簡單的二維陣列走訪的題目，沒有學過圖的也能作答，但用一些技巧會更簡單。

P-7-3. 機器人走棋盤 (APCS 201906)

有一個方格棋盤的每一個格子裡都標示了一個不同的整數，有一個機器人在此方格棋盤上行動，每一次只會移動到目前位置的上下左右四個相鄰格子其中之一。起點是數字最小的格子，每次移動則在可以移動位置中挑選數字最小的格子，但是走過的格子就不會再走，當無路可走的時候，機器人就會停下來。輸入方格棋盤中每個格子的數字，請模擬機器人走過的路徑，並輸出機器人走過的格子的數字總和。

以下是一個例子，輸入的 4×5 的方格內的數字如圖中所標示。在本例子中，機器人的起點會是 1，所走的路徑是 $1 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 14 \rightarrow 20 \rightarrow 21 \rightarrow 29 \rightarrow 30$ 。走到 30 的時候已經無路可走，所以機器人就停止了，而經過的數字總和是 132。

24	7	14	20	30
11	6	4	21	29
2	8	1	35	40
3	9	5	12	15

Time limit: 1 秒

輸入格式：輸入的第一行是兩個不超過 100 的正整數 m 與 n ，代表是一個 $m \times n$ 的方格棋盤，接下來有 m 行，每行 n 個數字，分別是方格棋盤由上而下，由左而右的數字。方格內的數字皆為不超過 $1e5$ 的非負整數，同一行數字之間以空白間隔。

輸出：機器人走過的格子中數字的總和。

範例一輸入：

```
2 7
6 8 7 2 1 4 5
9 3 10 11 12 13 14
```

範例一輸出：

```
36
```


<p>範例二輸入：</p> <pre>4 5 24 7 14 20 30 11 6 4 21 29 2 8 1 35 40 3 9 5 12 15</pre>	<p>範例二輸出：</p> <pre>132</pre>
<p>範例一說明：機器人走的路線是(1,2,7,8,3,9,6)。</p> <p>範例二說明：此為題目敘述中的範例。</p>	

這一題可以分成兩個部分：(1)找出一個二維陣列中的最小值，以該位置為起點；(2)模擬機器人的行走，對於某一個位置，找到上下左右四個鄰居中沒走過的最小值，而且要能辨識是否已經無路可走。第一個部分很簡單，處理二維陣列(矩陣)，我們可以搭配一個雙迴圈，一方面讀取矩陣內容，同時檢查最小值。為了要模擬機器人的行走位置，我們以一對變數(s_i, s_j)代表機器人所在位置在第 s_i 列 s_j 行，這裡我們採取矩陣行列的定位方式，使用平面座標的定位方式也可以。

為了要避免走到曾經走過的格子，我們可以採用圖形走訪的技巧，利用一個陣列 `visit[][]` 來記錄哪些格子被走過與否，另外為了預防出界，我們可以在原矩陣的外圍擺一圈虛擬的格子，並將他們都設定成已走過，這樣我們的程式碼會比較簡單。此外，每次我們要檢查四個鄰居，所以要寫四個 `if`，如果是可以走鄰近八方位的題目則要寫 8 個 `if`，這實在太囉嗦，所以我們把他寫成迴圈的樣子。請看以下範例程式，我們宣告了兩個長度為 4 的陣列 `di[]` 與 `dj[]`，裡面存放的其實是右下左上四個鄰居的列與行位移量。在檢查四個鄰居時，每一個鄰居的位置就是 $(s_i+di[k], s_j+dj[k])$ ，為了讓指令容易懂，不要太過複雜，所以另外設兩個變數 `ni` 與 `nj`。

```
#include <bits/stdc++.h>
using namespace std;
#define N 102
#define oo 100001

int main() {
    int a[N][N];
    bool visit[N][N]={false};
    int di[4]={0,1,0,-1}, dj[4]={1,0,-1,0};
    int i,j,n,m;
    scanf("%d%d", &m, &n);
    // set boundary to visited
    for (i=0; i<=m+1; i++)
        visit[i][0]=visit[i][n+1]=true;
```

```

for (j=0; j<=n+1; j++)
    visit[0][j]=visit[m+1][j]=true;
// read input and find minimal
int si,sj,imax=oo; // (si,sj) is current position
for (i=1; i<=m; i++) for (j=1; j<=n; j++) {
    scanf("%d",&a[i][j]);
    if (a[i][j] < imax) {
        si=i, sj=j, imax=a[i][j];
    }
}
// start walking
int total=0;
while (1) {
    total += a[si][sj]; // goto (si, sj)
    visit[si][sj] = true; // mark visited
    imax = oo;
    for (int dir=0; dir<4; dir++) { // check 4 neighbors
        int ni=si+di[dir], nj=sj+dj[dir];
        if (!visit[ni][nj] && a[ni][nj] < imax)
            i=ni, j=nj, imax=a[ni][nj];
    }
    if (imax == oo) break; // no way to go
    si=i, sj=j;
}
printf("%d\n",total);
return 0;
}

```

很多棋盤上最少走幾步的問題，其實是圖的 BFS 的一種形式，例如象棋的車、馬、西洋棋的皇后，下面是一個這樣的問題。

P-7-4. 方格棋盤的最少轉彎數路線

有一個 $m \times n$ 的方格棋盤，每個格子可能是 0 或 1，其中 0 代表可以通過而 1 代表不能通過。現在有一個機器人從左上角編號 (1, 1) 的格子出發，要到達右下角編號 (m, n) 的格子，每次可以往上下左右四個方向移動，我們要找到轉彎次數最少的路線。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 m 與 n ，代表格子棋盤的列數與行數，接下來有 m 行，每行是一個長度為 n 僅由 0 與 1 組成的字串，代表方格棋盤由上往下由左至右的內容，出發與目的格子必然是 0。 m 與 n 不超過 500。

輸出：最少的轉彎數。如果無法到達，則輸出 -1。

範例一輸入：

3 6

範例一輸出：

5

001010 101000 100010	
範例二輸入： 3 4 0110 0110 0010	範例二輸出： -1

把每一次轉彎當作走一步，這是最少步數的問題，所以可以用 BFS 來解。為了方便我們採取上一題類似的手法，在邊界放上不能走的標記，另外定義四個方向的位移。其他就是 BFS 的作法，停止條件包括可以走的已經走完或是目的位置已經達到。

```
// P-7-4 minimum turn
#include <bits/stdc++.h>
using namespace std;
#define N 510
#define oo 1000000

int main() {
    char a[N][N]; // chessboard
    int dis[N][N]; // min distance(turn)
    int di[4]={0,1,0,-1}, dj[4]={1,0,-1,0};
    int i,j,n,m;
    scanf("%d%d", &m, &n);
    // set boundary
    for (i=0; i<=m+1; i++)
        a[i][0]=a[i][n+1]='1';
    for (j=0; j<=n+1; j++)
        a[0][j]=a[m+1][j]='1';
    for (i=1; i<=m;i++)
        scanf("%s", &a[i][1]);
    for (i=1;i<=m;i++) for (j=1;j<=n;j++)
        dis[i][j]=-1; // not visited
    queue<pair<int,int>> Q; // visited (i,j)
    Q.push({1,1}); // starting position
    dis[1][1]=0;
    // start BFS
    while (!Q.empty() && dis[m][n]<0) {
        auto p=Q.front();
        Q.pop();
        int si=p.first, sj=p.second;
        for (int dir=0; dir<4; dir++) { // 4 directions
            int ni=si+di[dir], nj=sj+dj[dir];
            while (a[ni][nj]=='0') { // until touch 1
                if (dis[ni][nj]==-1) { // not visited
                    dis[ni][nj]=dis[si][sj]+1;
                }
            }
        }
    }
}
```

```

        Q.push({ni,nj});
    }
    ni+=di[dir];
    nj+=dj[dir];
}
}
}
if (dis[m][n]>0) dis[m][n]--; // count one more
printf("%d\n",dis[m][n]);
return 0;
}

```

下一個是 BFS 在一維數線上的問題，留為習題。

Q-7-5. 闖關路線 (APCS201910)

某個闖關遊戲上有一隻神奇寶貝與兩個可控制左右移動的按鍵。神奇寶貝被安置在僅可左右移動的滑軌上。滑軌分成 n 個位置，由左到右分別以 $0 \sim n - 1$ 表示。當遊戲開始時，神奇寶貝從位置 0 開始，遊戲的資訊包含 P 、 L 與 R 三個數字，其中 P 表示所須移至的目標位置， L 與 R 則分別表示每按一次左鍵或右鍵後，會往左或往右移動的格子數。此外，每一個位置 x 都對應一個瞬間移動位置 $S(x)$ ；每一次按鍵後，神奇寶貝會先依據按鍵往左或右移動到某個位置 x ，接著瞬間移動至 $S(x)$ 。某些點的瞬間移動位置等同原地點，也就是 $S(x) = x$ ，這些點稱為停留點。開始與目標位置都一定是停留點；此外，每個點的瞬間移動位置都一定是停留點（除非超出界外），也就是不會發生連續瞬間移動的情形。

遊戲的目標是以最少的按鍵數操作神奇寶貝由開始位置到達目標位置，此外，在移動過程中不可以超過滑軌的範圍 $[0, n - 1]$ ，否則算闖關失敗；某些點的瞬間移動位置也可能會超出滑軌的範圍，移動到這些點也會導致闖關失敗。

Time limit: 1 秒

輸入格式：輸入有兩行，第一行有 4 個數字，第 1 個為 n ，第 2 個為目標位置 P ，第 3 個為 L ，第 4 個為 R ，後三個數字皆為小於 n 之正整數，且 $2 \leq n \leq 1e6$ 。第二行有 n 個整數，依序是各點的瞬間移動位置 $S(0), S(1), \dots, S(n - 1)$ ，這些數字是絕對值不超過 $1e8$ 的整數。

輸出：輸出到達目標位置所需的最少按鍵數，如果無法到達目標位置，則輸出 -1 。

範例一輸入：

```

5 3 1 2
0 3 2 3 5

```

範例一輸出：

```

2

```

<p>範例二輸入：</p> <pre>10 8 2 3 0 5 2 3 4 5 6 6 8 9</pre>	<p>範例二輸出：</p> <pre>3</pre>
<p>範例一說明：位置區間為 $[0, 4]$，目標位置為 3，左鍵往左移動 1 格，右鍵往右移動 2 格。到目標位置最少的按鍵數是 2 次：從起點 0 第一次按下右鍵會到達位置 2，因為 $S(2)=2$，因此停留在 2，第二次按下左鍵，會往左移一格到達 1，因為 $S(1)=3$，所以瞬間移動到 3，由於 3 是目標位置，所以就闖關成功了。</p> <p>範例二說明：位置區間為 $[0, 9]$，目標位置為 8，按一次左鍵往左移動 2 格，按一次右鍵往右移動 3 格。到達目標位置的最少按鍵數是 3 次，其經過的路徑是：0，按右鍵 \rightarrow 3（停留在 3），按左鍵 \rightarrow 1（瞬間移動到 5），按右鍵 \rightarrow 8（停留在 8，到達）。</p>	

在一條數線上，每次按左鍵可以目前位置移動到某處，按右鍵可以到達某處。要計算從起點到終點最少要按幾次鍵。起點固定在 0 的位置，而終點則是輸入時給定。移動的規則是：按下右鍵往右移 R 格，按下左鍵往左移 L 格，此外，每次按鍵後移動到位置 x 時，會立刻移動到 $S(x)$ 。

如果沒有瞬間移動，這一題幾乎是問 $xR-yL=P$ 的最小 $x+y$ 整數解，我們這裡不探討數學問題，在有瞬間移動的狀況下，要以計算來完成求解。他顯然是 BFS 的退化在一維數線上的狀況，基本上我們算出一步（一次按鍵）可以到達的點，然後算兩步的點，...，直到走到目標或是無路可走為止，甚至不需要了解 BFS 也可以推想出作法，但是有 BFS 的知識讓我們可以很容易的作出標準作法。

DAG 上的路徑問題

借用下一個例題，我們來學習一下 DAG 的路徑長度問題。圖的路徑長度問題是一群重要的問題，典型的題目是找最短路徑或最長路徑，根據圖的性質與條件，問題的難度不一，須要使用不同的演算法。目前為止我們會使用 BFS 計算無加權圖的最短路徑，如果邊上有權重（長度），那麼 BFS 無法求得正確的解，但是對於 DAG，我們可以用類似 BFS 的方法計算出最長與最短路徑。我們先以最短路徑為例，最長路徑的做法是相同的，這裡的最長路徑是指最長的簡單路徑。

假設要計算 DAG $G=(V, E, w)$ 上某點 s 到 t 的最短路徑。對任一點 v ，令 $d[v]$ 是 s 到 v 的最短路徑長度，回想一下在動態規劃時的狀態轉移，我們可以知道， $d[s]=0$ ；而對於 $v \neq s$ ，

$$d[v] = \min\{d[u] + w(u, v) : \text{for all } (u, v) \text{ in } E\}$$

其中這些 u 就是 v 的前置狀態。這其實只是一種很簡單的 DP，類似於 P-6-6 (方格棋盤路線) 中從左上走到右下的最大權重路線，只是只能往右往下的棋盤是一個很簡單的 DAG。有了這個遞迴式之後，計算的方向就是 DAG 的拓樸順序，所以只要找到拓樸順序後，計算 $d[]$ 值就可以了，另外，也可以一面找拓樸順序一面計算 $d[]$ 。

P-7-6. DAG 的最長與最短路徑

輸入一個 DAG 以及 s 與 t 兩點，計算 s 到 t 的最短與最長簡單路徑的長度。兩點之間可能有多個邊。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表點數與邊數，點以 $0 \sim n-1$ 編號，第二行兩個整數 s 與 t ，接下來有 m 行，每行三個整數 u, v, w 代表一條有向邊 (u, v) 的長度是 w 。 n 不超過 $1e4$ ， m 不超過 $1e5$ ， w 的絕對值不超過 $1e4$ 。輸入保證是個 DAG。

輸出：第一行輸出最短路徑長度，第二行輸出最長路徑長度，如果不存在，兩者皆輸出 "No path"。

<p>範例一輸入：</p> <pre>5 6 0 4 0 2 3 0 3 1 2 1 -2 3 4 0 1 4 2 2 4 3</pre>	<p>範例一輸出：</p> <pre>1 6</pre>
<p>範例二輸入：</p> <pre>4 3 2 0 2 1 5 1 3 0 0 1 1</pre>	<p>範例二輸出：</p> <pre>No path No path</pre>

以下的範例程式中，我們採取一回合式的做法，也就是找拓樸順序的同時，就進行最短與最長路徑的計算。程式與前面計算拓樸順序的範例程式很像，不過這裡因為邊是有加權長度的，所以在儲存圖的鄰接串列時，我們需要儲存鄰居的編號以及到達該鄰居的邊長，程式中採用的方式是以 `pair<int, int>` 的結構來存一個邊，其中 `first` 是點的

編號而 second 是邊長。因為要計算最短與最長距離，我們用 $d[v]$ 與 $p[v]$ 代表起點 s 到達 v 點的最短與最長距離，只要跟著拓樸順序來作 DP 就可以了。是

```
//P-7-6 longest and shortest path on dag O(m+n), non-recursive
#include <bits/stdc++.h>
using namespace std;
#define N 10010
#define oo 1000000000
int main() {
    vector<pair<int,int>> adj[N]; // (neighbor, length)
    int indeg[N]={0};
    int n, m, source, goal;
    scanf("%d%d", &n, &m);
    scanf("%d%d", &source, &goal);
    for (int i=0; i<m; i++) {
        int from, to, w;
        scanf("%d%d%d", &from, &to, &w);
        adj[from].push_back({to,w});
        indeg[to]++;
    }
    queue<int> Q;
    int d[N], p[N], cnt=0; // min and max distance
    for (int i=0;i<n;i++) d[i] = oo;
    for (int i=0;i<n;i++) p[i] = -oo;
    d[source] = p[source] = 0;
    // push all indeg==0
    for (int i=0;i<n;i++)
        if (indeg[i]==0)
            Q.push(i);
    while (!Q.empty()) { // until queue is empty
        int v=Q.front();
        Q.pop(); cnt++;
        for (auto e: adj[v]) {
            if (d[v] < oo) { // path exists
                d[e.first] = min(d[e.first], d[v] + e.second);
                p[e.first] = max(p[e.first], p[v] + e.second);
            }
            if (--indeg[e.first] ==0)
                Q.push(e.first);
        }
    }
    // output
    if (cnt!=n) fprintf(stderr, "Not a dag, %d < %d\n", cnt, n);
    if (d[goal]==oo) printf("No path\nNo path\n");
    else printf("%d\n%d\n", d[goal], p[goal]);
    return 0;
}
```

我們當然也可以用 top-down 的 DP，寫成遞迴的樣子，以下是範例程式。為了進行遞迴呼叫，我們儲存邊的方式是存 in-neighbor，也就是存每個點的前置點有哪些。其

他部分就只是簡單的 top-down DP 的寫法，我們將找 d[] 與 p[] 分成兩個函式，事實上當然可以寫在一個函式中。

```

//P-7-6 longest and shortest path on dag, recursive DP
#include <bits/stdc++.h>
using namespace std;
#define N 10010
#define oo 1000000000
vector<pair<int,int>> pred[N]; // (neighbor, length)
int d[N], p[N]; // min and max distance
// min distance from source

int mind(int t) {
    if (d[t] <= oo) return d[t]; // check memo
    d[t] = oo; // find minimum from each predecessor
    for (auto e: pred[t]) {
        int x=mind(e.first);
        if (x<oo) d[t] = min(d[t], x + e.second);
    }
    return d[t];
}

// max distance from source
int maxd(int t) {
    if (p[t] >= -oo) return p[t];
    p[t] = -oo;
    for (auto e: pred[t]) {
        int x=maxd(e.first);
        if (x>-oo) p[t] = max(p[t], x + e.second);
    }
    return p[t];
}

int main() {
    int n, m, source, goal;
    scanf("%d%d", &n, &m);
    scanf("%d%d", &source, &goal);
    // read predecessor of each node
    for (int i=0; i<m; i++) {
        int from, to, w;
        scanf("%d%d%d", &from, &to, &w);
        pred[to].push_back({from,w});
    }
    for (int i=0;i<n;i++)
        d[i] = oo+1; // >oo as un-computed
    for (int i=0;i<n;i++)
        p[i] = -oo-1; // <-oo as un-computed
    d[source] = p[source] = 0;
    d[goal] = mind(goal); // can be combined in one pass
    p[goal] = maxd(goal);
    if (d[goal]==oo) printf("No path\nNo path\n");
    else printf("%d\n%d\n", d[goal], p[goal]);
    return 0;
}

```


以下一個題目是 DAG 在計劃管理上的應用，AOV 是 Activity On Vertex 的縮寫，我們留做習題。

Q-7-7. AOV 最早完工時間

某個計劃有 n 項工作，以 $1 \sim n$ 編號，工作之間有前置關係，我們用一個點代表一項工作，以邊 (a, b) 表示 a 是 b 的前置工作，也就是說 a 完成之後，工作 b 才能開始。每一項工作 v 有一個需求的工作時間 $w[v]$ ，代表該工作至少需要耗時 $w[v]$ 才能完成。

本題要計算此計畫的最早完工時間，也就是計畫開始後，最早可以完成所有工作的時間。此外，對於某個工作，如果該工作的任何延誤都會讓整個計畫的最早完工時間延後，這個工作就稱為關鍵工作，否則稱為非關鍵工作。請計算那些工作是關鍵工作。

舉例來說， $n=3$ ，前置關係有 $(1, 3)$ 與 $(2, 3)$ ，代表工作 1 完成後才可以開始工作 3，相同的，工作 2 完成後才可以開始工作 3。需求工時為 $w[1]=1$ ， $w[2]=3$ ， $w[3]=4$ 。我們可以看出，工作 1 與 2 可以一起開始平行作業，在時間 1 時可以完成工作 1，時間 3 時可以完成工作 2，因此工作 3 最早可以開始的時間是 3，而最早的完工時間是 7。這三項工作中，工作 2 與 3 是關鍵工作，但工作 1 不是，因為工作 1 只要不花超過 3 的工時（延誤超過 2），並不會影響整個計畫的最早完工時間。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表工作數與前置關係數，點以 $1 \sim n$ 編號，第二行是 n 個正整數，依序代表每一個工作的需求工時 $w[v]$ ，接下來有 m 行，每行兩個整數 u 與 v ，代表 u 是 v 的前置工作。 n 不超過 $1e4$ ， m 不超過 $1e5$ ， w 不超過 $1e3$ 。輸入保證有解。

輸出：第一行輸出最早完工時間，第二行輸出那些工作是關鍵工作，輸出時依照工作的編號由小到大，相鄰兩數字之間以一個空白分格。

範例一輸入： 3 2 1 3 4 1 3 2 3	範例一輸出： 7 2 3
範例二輸入： 4 4 1 2 3 4	範例二輸出： 8 1 3 4

1 2	
1 3	
1 4	
3 4	

下一個也是習題。

Q-7-8. 小寶的著色問題

小寶有個布娃娃，娃娃會變化。娃娃身上有一些圖案，這些圖案是由一些圓圈和一些線條組成，每個線條連接著某兩個不同的圓圈。小寶很喜歡著色，他決定要把這些圓圈塗上紅色或藍色，他覺得有線條相連的圓圈最好塗上不一樣的顏色，但是因為圓圈很多，他不知道能不能夠完成這樣的想法，請你幫他計算看看是否可以有辦法完成這樣的著色。

Time limit: 1 秒

輸入格式：第一行有一個整數 T ，代表接下來有 T 張圖的資料要計算。每張圖資料的第一行是兩個整數 n 與 m ，代表有 n 個圓圈與 m 個線條，第二行有 $2m$ 個整數，每兩個一組代表一根線條所連接的兩個圓圈編號，圓圈是以 $0 \sim n-1$ 編號。 n 不超過 $1e4$ ， m 不超過 $1e5$ 。 T 不超過 20。

輸出：依序每一行輸出一張圖是否可以正確著色，如果是，則輸出 `yes`，否則輸出 `no`。

範例一輸入：	範例一輸出：
2	yes
2 0	no
4 3	
0 3 3 2 2 0	

範例說明：第一張圖 $n=2$ ， $m=0$ ，表示有兩個圓圈而沒有線條，可以正確著色。第二張圖有 4 個圓圈與 3 根線條，3 根線條是 $(0, 3)$ 、 $(3, 2)$ 與 $(2, 0)$ ，這 3 個點如果只用 2 種顏色，不管如何上色都會有同色的相連。

提示：把圓圈看成圖的點，線條看成邊。每個點與他的鄰居必須不同色，我們可以用 BFS 或 DFS 做圖的走訪與著色，過程中檢查是否有衝突發生。

7.4. 補充教材 (*)

在這一節中，我們挑選一些還算常用也不太困難的圖論演算法當作補充教材，包括：Dijkstra's 演算法、併查集(Union and Find)，以及最小生成樹(Minimum spanning tree)。

7.4.1. Dijkstra 演算法

我們前面介紹了無加權圖上面以 BFS 計算最短距離的方法，對於 DAG 我們也可以沿著拓撲順序來計算距離，但是應用問題中，更常見的是邊上有加權也可能有環路的圖。Dijkstra 演算法是用在一個加權圖上計算一點到所有點的最短路徑的演算法，圖可以有向的或無向的，但是邊上的權重必須是非負的。

假設輸入的圖是 $G=(V, E, w)$ 與一個起點 s ，我們要計算 s 到 V 中每一點的一條最短路徑。演算法運作過程中，我們以 $d[v]$ 紀錄目前已知 s 到 v 的最短距離，集合 Q 表示最短距離尚未確定的點，演算法如下：

```
// Dijkstra 演算法，輸入：G=(V, E, w) 與一個起點 s
初始化：d[s]=0; 對其他 v 點，d[v]=∞; Q=V;
while 尚有未確定距離的點(Q is not empty) do
    在 Q 中找出 d[] 最小的點 v;
    將 v 移出 Q 中; // d[v] 確定為 s 到 v 的最短距離
    for each u in adj[v] and in Q do: //與 v 相鄰尚未確定距離者
        if d[v]+w(v,u) < d[u] then // 從 v 過來路更近
            d[u] = d[v]+w(v,u);
            parent[u] = v; // 已知到達 v 之最短路徑的前一點
        end if
    end for
end while
```

演算法其實不困難，每次在尚未確定距離的點中，找出 $d[]$ 最小的 v ，因為邊長都不是負的，所以 $d[v]$ 的值不可能更短了，我們就將它確定，然後，用 $d[v]+w(v,u)$ 去嘗試更新與 v 相鄰的 u 點的距離。如果圖中有些點是從 s 走不到的，最後找出來的距離就是初設的無窮大。此外，最後 $d[v]$ 是最短距離，那麼要如何找路徑呢？我們在每次更新距離時，存了 $parent[v]$ ，它記錄著從 s 到 v 的最短路徑的最後一步，所以當計算完成時，由後往前一步一步回推就可以得到路徑經過的點的順序。名字取為 $parent$ 的原因是：如果將每個點與它的 $parent$ 連起來，我們會得到一棵以 s 為根的樹狀圖，

這個樹稱為 *shortest-path tree*，在此樹上， s 到任一點的路徑都是在原圖上的最短路徑。

我們把重點擺在如何實作，以下一個例題來當作說明，這個例題是無向圖，對有向圖的做法類似。

P-7-9. 最短路徑 (*)

輸入一個無向圖 $G=(V, E, w)$ ，其中點以 $0\sim n-1$ 編號，而邊的權重是非負整數。計算 0 號點到其他點的最短路徑長度。兩點之間可能有多個邊。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表點數與邊數，接下來有 m 行，每行三個整數 u, v, w 代表一條無向邊 (u, v) 的長度是 w 。 n 不超過 $1e4$ ， m 不超過 $1e5$ ， w 是不超過 $1e4$ 的非負整數。

輸出：第一行輸出 0 到各點之最短路徑長度中的最大值，也就是在可以到達的點中，最短距離最大的是多少，第二行輸出有多少點無法從抵達。

範例一輸入：

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

範例一輸出：

```
7
1
```

範例二輸入：

```
3 3
2 1 5
1 0 0
0 1 1
```

範例二輸出：

```
5
0
```

(範例測資中前兩筆的 n 不超過 500。)

要做出 $O(n^2)$ 複雜度的 Dijkstra 演算法一點都不困難，例如在 *dense graph* 上以鄰接矩陣儲存時，我們直接依照演算法的步驟來寫，就可以得到以下的範例程式了。

```

// O(n^2) Dijkstra algorithm, dense graph using matrix
#include<bits/stdc++.h>
using namespace std;
#define N 510
#define oo 1000000000

int main() {
    int n, m, w[N][N], parent[N];
    // input data
    scanf("%d%d", &n, &m);
    if (n>510) {fprintf(stderr,"too large n\n"); return 0;}
    for (int i=0;i<n;i++) for (int j=0;j<n;j++)
        w[i][j]=oo;
    for(int i=0;i<m;i++) {
        int u,v,t;
        scanf("%d%d%d", &u, &v, &t);
        w[u][v]=w[v][u]=min(w[u][v],t); // maybe multiple edges
    }
    // initial
    int d[N], source=0;
    bool done[N]={false}; // whether the distance is converged
    for (int i=0;i<n;i++) {
        d[i]=oo;
        parent[i]=-1;
    }
    d[source]=0;
    while (1) { // converge one vertex each iteration
        // find min d[] of not-done vertex
        int v=-1, mind=oo;
        for (int i=0;i<n;i++) {
            if (!done[i] && d[i]<mind) {
                v=i;
                mind=d[i];
            }
        }
        if (v<0) break; // no more reachable vertex
        done[v]=true;
        // update neighbor
        for (int i=0;i<n;i++) {
            if (d[v]+w[v][i]<d[i]) { // shorter path to i
                d[i] = d[v]+w[v][i];
                parent[i]=v;
            }
        }
    }
    int maxd=-1, cnt=0, far;
    for (int i=0;i<n;i++)
        if (d[i]<oo) {
            if (d[i]>maxd) maxd=d[i], far=i;
        } else cnt++;
    printf("%d\n%d\n",maxd,cnt);
    /*
    while (far>=0) {
        fprintf(stderr, "%d ",far);
    */
}

```

```

        far=parent[far];
    }*/
    return 0;
}

```

要把時間複雜度降下來，除了儲存圖要改用鄰接串列之外，最重要的是每一回合如何很快地找到 $d[]$ 最小的點，這就需要借助資料結構的幫忙。因為每次找到一點之後，我們會更新其他點的 $d[]$ ，因此 $d[]$ 是會動態變化的，也就是說無法靠著一開始用排序來處理。因為每次都要找最小值，最直接的想法是啟用一個優先佇列 (PQ, priority queue)，但是這裡的困難還不只在於此，最大的問題在於每一回合可能有某些 $d[]$ 值會更動 (降低)。基本上 PQ 是以 heap 寫的，如果是自己寫的 heap，我們可以修改這個資料結構，讓它可以處理 decreasing key 的狀況，但是一般為了節省時間，我們會採用 STL 中提供的 PQ，可惜 STL 的 PQ 並不提供此功能，多數競賽選手採取的變通方法是以下“偷懶”的策略：

我們將 $(d[u], u)$ 的結構放在 PQ 中，當我們要將 $d[u]$ 修改為 x 的時候，我們並不從 PQ 中刪除或更新該筆資料，而是直接將 (x, u) 放入 PQ 中。這樣一來，對於某些點，PQ 中可能會有多筆資料，但是無論如何，最小的會先出來，因為我們只做 decreasing key，所以最先出來的是最新的資料。需要額外處理的事情是，當我們每一回合找最小值時，需要檢查出來的最小值點是否已經處理完畢了，如果是，則直接丟掉它。

另外一個比較小的問題是 PQ 的比較函數自己寫比較麻煩 (不好記)，所以最好是方法是運用 STL 中的 pair。將我們要比較的 key ($d[u]$) 放在第一欄位，點的編號放在第二欄位，這樣可以完全不必自己動手，請看以下的範例程式。第 25 行我們宣告一個 $\text{pair}<\text{int}, \text{int}>$ 的 PQ，預計將 $(-d[u], u)$ 放入資料，使用負值的原因是讓最小值先出來 (default 是最大值出來)。重點在第 26 行的 while 迴圈，每一回合會確定一點的最短距離，但是在第 30 行我們會檢查出來的點是否已經出來過了，如果是，則直接不予理會，否則，出來的才是我們要的。第 33 行的 for 迴圈，我們檢查該點的鄰居，如果有更好的 $d[]$ 值就予以更新，如同前面說的，我們將更新後的直接壓入 PQ 中。時間複雜度是多少呢？分析的方式類似 BFS，因為每個點的鄰居數總和是 $O(m)$ ，而在 PQ 上的操作都在 $O(\log(n))$ 時間內，所以總時間是 $O(m \log(n))$ 。

如果不使用 PQ，另外一種可能的處理方法是使用 set，因為 set 也可以找最小值，每次的變動都可以藉著刪除與新增來完成，我們在範例程式中有另外一支是以 set 來做的，留給有興趣的人自行參考，這裡就不多做明說了。因為要使用到動態資料結構，Dijkstra 演算法要在 APCS 中出現的機會應該是不大，或者給你演算法讓你模擬一個 $O(n^2)$ 的方法。

```

00 // Dijkstra algorithm O(mlogn) using PQ
01 #include<bits/stdc++.h>
02 using namespace std;
03 #define N 10010
04 #define oo 1000000001
05
06 vector<pair<int,int> > adj[N]; //Adjacency list
07 int d[N], parent[N];
08 bool done[N]={false};
09
10 int main() {
11     int n, m;
12     // input data
13     scanf("%d%d", &n, &m);
14     for(int i=0;i<m;i++) {
15         int u,v,t;
16         scanf("%d%d%d", &u, &v, &t);
17         adj[u].push_back({v,t});
18         adj[v].push_back({u,t});
19     }
20     // initial
21     int source=0;
22     for (int i=0;i<n;i++) d[i]=oo;
23     for (int i=0;i<n;i++) parent[i]=-1;
24     priority_queue<pair<int,int>> PQ; // (-d[], vertex)
25     PQ.push({d[source]=0,source});
26     while (!PQ.empty()) { // converge one vertex each iteration
27         auto p=PQ.top(); // find min d[] of not-done vertex
28         PQ.pop();
29         int v=p.second;
30         if (done[v]) continue; // already done, discard
31         done[v] = true;
32         // update neighbor
33         for (auto e: adj[v]) {
34             int u=e.first, w=e.second;
35             //if (done[u]) continue; // not necessary
36             if (d[v]+w < d[u]) { // shorter via v
37                 d[u] = d[v]+w;
38                 parent[u] = v;
39                 PQ.push({-d[u], u}); // push a better d[] of u
40             }
41         }
42     }
43     int maxd=-1, cnt=0, far;
44     for (int i=0; i<n; i++)
45         if (d[i]<oo) {
46             if (d[i]>maxd) maxd=d[i], far=i;
47             } else cnt++;
48     printf("%d\n%d\n", maxd, cnt);
49     /*
50     while (far>=0) {
51         fprintf(stderr, "%d ",far);
52         far = parent[far];
53     }*/
54     return 0;

```

7.4.2. 併查集(Union and Find)

併查集是一種資料結構，用來處理一群不相交的集合(disjoint set)，說是一種資料結構，應該是一套資料儲存方式與操作的演算法。併查集處理的資料是不相交的集合，如同它的名字，提供的主要操作有兩個：合併兩集合以及查詢某個元素在哪個集合中。

併查集是一種樹狀圖的觀念，但是實作時完全以陣列來做就可以，也就是心中有樹但程式中無樹，寫起來非常容易，不容易寫錯，效率又高。為了方便說明，我們以人與社團來比喻元素與集合。假設元素的編號是 $0 \sim n-1$ ，我們只要一個陣列就可以，依循樹狀圖的概念，陣列就叫做 `parent[]`。每一個人都會屬於某個社團，但不會屬於兩個社團。除了社長，每個人 i 都有一個直接領導，就是他的衣食父母 `parent[i]`，社長就是社團的代表人，為了方便，若 i 是社長，我們令 `parent[i] = -x`，其中而 x 是該社團中的總人數。我們可以用下列方式來操作合併與查詢：

```
// 最簡單的 union and Find
find(x): // 給一個人的編號，回傳 x 所在社團的社長編號
    while (parent[x] >= 0) do
        x = parent[x]; // 一路往上找，直到社長
    end while
    return x;

union(x, y): // 合併 x 與 y 所在的集合
    r1 = find(x); // 找到 x 的社長老大
    r2 = find(y); // 找到 y 的社長老大
    if (r1 != r2) then // 不同社長才要合併
        parent[r2] = r1; // 一個社長拜另外一個社長當社長老大
    end if
```

這個演算法很簡單，查找時就一路沿 `parent[]` 往上走，直到走到社長(代表元素)為止。合併的時候，先檢查兩者是否是同一社團，如果是就不需要做任何事，如果不是，就把其中一個社團社長的衣食父母改為另外一個社長，這就為成了合併的動作。這個方法雖然簡單，但是有個缺點，在一連串合併之後，可能有些人離社長關係非常遠，想像每次把一個社團都併入一個一人社團，這就如同社團每次都來一個空降的新人當新的社

長，每併一次，最底下的那個人離社長的層級就增加一，這樣在查找的時候可能會很浪費時間。

改進的方法是這樣：每次合併時，把人數少的社團併入人數多的社團，也就是人多的社團社長當合併後的新社長。事實上，如果依照這個規定，社團的層級就不會超過 $\log(n)$ 。證明：如果合併造成你離社長的層級增加 1 的話，表示你併入一個比原來更大的社團，也就是你的社團人數至少增加一倍，如果你離社團的層級為 k 表示社團至少有 2^k 個人，在總人數只有 n 的前提下，層級不會超過 $\log(n)$ 。

其實做到這個方法，我們的併查集效率就會很不錯了，但還有另外一招來提高效率，這招數稱為路徑壓縮，做法是這樣：每當我們要做 `find()` 的時候，除了找到社長之外，我們“順便”把尋找過程中碰到的那些人的 `parent` 直接改設成社團社長。這樣做似乎看不出直接的效果，事實上它是個前人種樹後人乘涼的概念，這次多花一點時間，但後續再做查找時，因為路徑都縮短了，就會比較快。這個效率的分析很不容易，基本上有人分析出，如果有一連串的合併與查詢，在以上這樣的方法之下，每一個操作所需要的均攤時間幾乎是 $O(1)$ 。詳細的分析超過我們的範圍，這裡不多解釋了，以下是修改後的演算法，請留意社長的 `parent[]` 是存社團人數的負值，所以大小關係是顛倒的。

```
// 優化後的 union and Find
find(x): // 回傳 x 所在集合的代表元素並且做路徑壓縮 (遞迴版)
    if (parent[x]<0) then
        return x;
    end if
    parent[x] = find(parent[x]); // 遞迴, top-down DP
    return parent[x];

union(x,y): // 合併 x 與 y 所在的集合
    r1 = find(x);
    r2 = find(y);
    sum = parent[r1] + parent[r2]; // -總人數
    if (parent[r1] < parent[r2]) then
        parent[r2] = r1;
        parent[r1] = sum;
    else if (parent[r1] > parent[r2]) then
        parent[r1] = r2;
        parent[r2] = sum;
    end if
```

前面的例題 P-7-2 中，我們要找權重總和最大的連通區塊，當時是用 BFS 或 DFS 來做，事實上也可以用併查集來做，每讀到一個邊就把兩端點所在的連通區塊聯集起來，以下是這樣寫的範例程式。

```
// P-7-2 max component by find and union;
#include <bits/stdc++.h>
using namespace std;
#define N 50010
int p[N];
// find root
int rfind(int u) {
    if (p[u] < 0) return u;
    return p[u] = rfind(p[u]);
}

int main() {
    int m, n, w[N];
    scanf("%d %d", &n, &m); //printf("%d: ",n);
    for (int i=0; i<n; i++)
        scanf("%d", w+i);
    for (int i=0; i<n; i++)
        p[i] = -1; // parent<0 is a root
    for (int i=0; i<m; i++) { // read an edge and union
        int r1,r2,u,v;
        scanf("%d%d", &u, &v);
        r1 = rfind(u); // root of u
        r2 = rfind(v); // root of v
        if (r1==r2) continue; // same set
        if (p[r1] < p[r2]) {
            p[r1] += p[r2];
            p[r2] = r1;
            w[r1] += w[r2];
        } else {
            p[r2] += p[r1];
            p[r1] = r2;
            w[r2] += w[r1];
        }
    }
    int ans=0;
    for (int i=0; i<n; i++)
        if (p[i]<0 && w[i]>ans)
            ans = w[i];
    printf("%d\n", ans);
    return 0;
}
```

如同前面走訪的問題，連通區塊的概念也用在二維陣列。

P-7-10. 挖坑跳 (@@) (TOI 入營考)

小雨喜歡在土堆上挖坑，在坑裡注入水之後，然後玩跳水坑的遊戲。一個庭院被畫分成 $m \times n$ 個相同大小的矩陣方格，每一個格子可能是土堆或水坑，一個水坑格子與其上下左右四個方向的水坑格子被視為連通的同一個水坑。我們需要計算有多少個水坑以及最大的水坑佔多少個格子。除了一開始的土堆與水坑，小雨每次會指定將某個格子變成水坑，如果這個方格是新挖出來的水坑，有可能把附近的水坑連在一起。小雨一共挖了 k 次，請你計算出每一次挖了之後的水坑數與最大水坑面積。

土堆與水坑的資訊可以看成一個二維矩陣，以 0 表示土堆，而 1 表示水坑，位置的標示方式以左上角為 $(1, 1)$ ，右下角是 (m, n) 。以下是一個 $m=3, n=5$ 的例子。一開始的水坑資料如下：

1	0	0	1	1
0	1	1	0	1
1	1	0	0	1

我們可以看出來有 3 個水坑：左上角只佔 1 格的水坑，左下有一個 4 格的水坑，以及右方有一個 4 格的水坑。假設現在小雨把 $(2, 4)$ 的土堆改成水坑（紅色位置），左下與右方的水坑便會連接成一個面積為 9 的水坑。

Time limit: 1 秒

輸入格式：第一行是三個整數，依序是 m, n 與 k ，接下來 m 行是土堆與水坑資料，每一行有 n 個 0 或 1 的數字，順序為由上而下，從左至右。最後一行有 $2k$ 個數字，依序每兩個代表一個被挖成水坑的位置 (i, j) ，如果該位置本來就是水坑，就代表沒有動作。 m 與 n 不會超過 500， k 不超過 20000。

輸出：輸出兩行，第一行是每次最大水坑面積的總和（包含一開始，所以有 $k+1$ 次），第二行是每次水坑數量的總和。

範例一：輸入 3 5 1 1 0 0 1 1 0 1 1 0 1 1 1 0 0 1 2 4	範例一：正確輸出 13 5
---	-------------------------

<p>範例一說明：這是題目敘述中的例子，一開始有 3 個水坑，大小是 (1, 4, 4)，操作一次後變 2 個水坑，大小是 (1, 9)。最大水坑面積總和是 4+9=13，水坑數樣總和是 3+2=5。</p>	
<p>範例二：輸入</p> <pre>2 6 2 0 1 1 0 1 1 0 1 0 0 0 1 2 4 1 4</pre>	<p>範例二：正確輸出</p> <pre>14 6</pre>
<p>範例二說明：一開始有 2 個水坑，面積是 (3, 3)，第一次操作後有 3 個水坑，面積是 (3, 3, 1)，第二次操作後變成一個面積 8 的水坑。</p>	

給一個二維的 0/1 矩陣，本題要找出連通的 1 區塊，連通的定義是上下左右四方位。初始找出各個連通區塊可以使用 BFS 或 DFS，但是題目還有後半部：有 k 次操作，每次會將某個位置設為 1，如果該位置原來就 1，當然沒事發生，但如果是 0 變成 1，區塊就會變動。這時候如果做 DFS 或 BFS 可能會太花時間，因為可能有 2 萬次操作，每次的區塊可能達到 25 萬點。因為每個點只會屬於某個區塊，使用併查集來維護區塊是好的選擇。

題目本來是二維矩陣，每一個格子需要一對 (i, j) 來表示，在以下的範例程式中，我們把所有的點 (i, j) 以 row-major 的方式轉成一維陣列的位置 $i * (n+2) + j$ 。行數 n 加 2 的原因是我們把外圍多留了一圈當四個邊界，邊界在初值都設成 0 (土堆)。在一維轉換後，每個點的四個鄰居的位移就是 $(+1, -1, +n, -n)$ ，其中 n 已經過加 2 的修正。範例程式中一共有三個副程式，`rfind()` 與 `combine()` 是併查集使用的，另外 `dfs()` 用在出始找連通區塊，它除了將走訪到的區塊標記為代表點的編號之外，也回傳區塊的大小。程式碼雖然有點長，但是如果了解 DFS 與併查集，應該不難了解。

```
// P-7-10 components in matrix, transform to 1d
// a cell (i,j) is indexed by i*(n+2)+j
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
#define N 510
char a[N*N]={0}; // a 0/1 matrix
int p[N*N], di[4]={1,-1}; // parent and index difference
int n_area=0, max_area=0;
// find root
int rfind(int u) {
```

```

    if (p[u] < 0) return u;
    return p[u] = rfind(p[u]);
}
// find component, return size
int dfs(int v, int root) {
    p[v]=root;
    int cnt=1;
    for (int dir=0;dir<4;dir++) {
        int u=v+di[dir];
        if (a[u] && p[u]==-1)
            cnt += dfs(u, root);
    }
    return cnt;
}
// combine two area, p[] = -size
void combine(int u,int v) {
    int r1 = rfind(u); // root of u
    int r2 = rfind(v); // root of v
    if (r1==r2) return; // same set
    max_area=max(max_area, -p[r1]-p[r2]);
    n_area--;
    if (p[r1] < p[r2]) {
        p[r1] += p[r2];
        p[r2] = r1;
    } else {
        p[r2] += p[r1];
        p[r1] = r2;
    }
    return;
}
int main() {
    int m, n, k;
    scanf("%d%d%d", &m, &n, &k);
    for (int i=1; i<=m; i++) for (int j=1;j<=n;j++) {
        p[i*(n+2)+j] = -1; // parent<0 is a root
    }
    for (int i=1; i<=m; i++) for (int j=1;j<=n;j++) {
        scanf("%d", &a[i*(n+2)+j]);
    }
    n+=2;
    di[2]=n, di[3]=-n; // next row and previous row
    int mn=(m+1)*n;
    // dfs for component
    for (int v=n; v<mn; v++) {
        if (a[v] && p[v]==-1) { // -1 is unvisited
            p[v] = - dfs(v, v);
            n_area++;
            max_area=max(max_area, -p[v]);
        }
    }
    LL total_n=n_area, total_max=max_area;
    //printf("%d %d\n", total_max, total_n);
    while (k--) { // read an cell
        int i, j, v;
        scanf("%d%d", &i, &j);

```

```

assert(i<=m && j<=n);
v=i*n+j;
if (a[v]==1) continue; // already 1, do nothing
a[v]=1; // set to 1
n_area++;
max_area=max(max_area, 1);
// try to combine 4 neighbors
for (i=0; i<4; i++) {
    if (a[v+di[i]]==0) continue;
    combine(v, v+di[i]);
}
total_n += n_area;
total_max += max_area;
}
printf("%lld\n%lld\n", total_max, total_n);
return 0;
}

```

下一個一維陣列的類似題，我們留做習題，因為測資設定的範圍小，這一題不使用併查集也可以通過。

Q-7-11. 紅白彩帶 (APCS201902)

一條彩帶被分成 n 個相同大小的格子，有的格子是紅色，有些則是白色。小軒拿到彩帶後開始塗顏色，每次會將一個白色格子塗滿紅色，然後小軒會算一算目前最長與最短紅色區域的長度佔了幾格，相鄰的紅色格子就屬於同一個紅色區域。小軒一共塗了 k 次，請你計算出每一次紅色區域的最大與最小長度，並輸出個別的總和。

彩帶可以看成一維陣列，以 0 表示白色，而 1 表示紅色。彩帶的格子從 1 開始由左而右依序編號，小軒每次將某個格子塗上紅色。以下是一個例子。

格子編號	1	2	3	4	5	6	7	8	9
剛開始的彩帶	0	1	1	0	0	1	0	1	0
一開始紅色區域最大的長度是2，最小的長度是1。									
第1次塗在第5格後	0	1	1	0	1	1	0	1	0
第1次塗完後最長的紅色區域有2塊，長度都是2，最小的長度是1。									
第2次塗在第1格後	1	1	1	0	1	1	0	1	0
第2次塗完後紅色區域的最大長度是3，最小長度是1。									
第3次塗在第7格後	1	1	1	0	1	1	1	1	0
第3次塗完後紅色區域的最大長度是4，最小長度是3。									

以這個例子而言，每一次(包含剛開始)紅色區域的最大長度總和是 $2+2+3+4=11$ ；而紅色區域的最小長度總和是 $1+1+1+3=6$ 。

Time limit: 1 秒

輸入格式：輸入有三行，第一行是兩個整數，依序是 n 與 k ，代表彩帶長度以及小軒塗色的次數， $n \leq 1e5$ 、 $k \leq 2e4$ 。第二行有 n 個 0 或 1 的數字，依序代表彩帶第 1 格至第 n 格的顏色，0 代表白色，1 代表紅色。第三行有 k 個數字，依序代表每一次塗紅色的格子編號，若 $k = 0$ ，則第三行為空行。同一行數字之間都是以空白間隔。小軒每次一定都塗在白色格子，而且紅色區域的長度不會超過 10100。

輸出：輸出兩行，第一行是每次紅色區域最大長度的總和，第二行是每次紅色區域最小長度的總和。

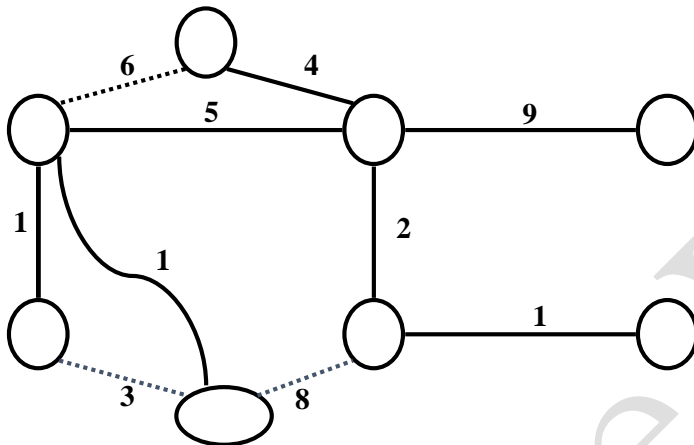
<p>範例一：輸入</p> <pre>5 1 1 0 1 0 1 2</pre>	<p>範例一：正確輸出</p> <pre>4 2</pre>
<p>範例一說明：一開始最大長度是 1 而最小長度也是 1，塗在第二格之後最大長度是 3 而最小長度還是 1。最大長度總和為 $1+3=4$；最小長度總和為 $1+1=2$。</p>	
<p>範例二：輸入</p> <pre>9 3 0 1 1 0 0 1 0 1 0 5 1 7</pre>	<p>範例二：正確輸出</p> <pre>11 6</pre>
<p>範例二說明：此為題目中所舉的例子。</p>	

7.4.3. 最小生成樹

樹是沒有環路的連通圖，對於一個圖 G ，它的一個生成樹(spanning tree)是指挑選 G 的一些邊，將所有點連成一個連通區塊，而挑選的邊形成的是一個樹。因為 n 個點至少要 $n-1$ 個邊才能形成一個連通區塊，而超過 $n-1$ 個邊必然會造成有環路，所以

spanning tree 一定恰好是 $n-1$ 個邊。G 的最小生成樹 (minimum spanning tree) 是指邊的權重總和最小的生成樹。

假設我們有 n 個城市，要建設一個交通網路將 n 個城市連起來，如果圖 G 的每一個邊代表一條可以建設的道路，而邊上有一個非負的權重代表該道路的建設成本，那麼最小生成樹就是最低的建設總成本。下圖是一個例子，其中實線代表最小生成樹的邊，虛線代表沒有挑選的邊，圖中有 8 個點，一定剛好有 7 個邊。



如果是無加權圖，也就是圖的每個邊的權重皆相同，那麼最小生成樹的成本一定剛好是 $n-1$ ，BFS 與 DFS 都可以建構出一個最小生成樹。在一般狀況下，我們要處理的是加權圖。計算最小生成樹有多個演算法，目前比較常用的有兩個：Prim 演算法與 Kruskal 演算法。

先介紹 Prim 的演算法，這個演算法與 Dijkstra 演算法幾乎一模一樣，我們從任選一個點 s 開始，逐步建構我們要的樹 T ，每一回合會從尚未連進 T 的點中挑選一個點 v 以及一個已經在 T 中的點 u ，將 v 與邊 (u, v) 加入 T 中，而挑選的點與邊是滿足上面做法中成本最小的邊，也就是一點在 T 中一點不在 T 中的所有邊中，成本最小的邊。重複這個步驟直到全部的點都加入 T 中為止。以下演算法中我們計算成本，如果要建構出樹，就把每個點 v 與 $\text{parent}[v]$ 這個邊輸出即可。

```
// Prim 演算法，輸入：G=(V,E,w) 與一個任選一個起點 s
初始化：T 為空，d[s]=0；對其他 v 點，d[v]=∞；Q=V；cost=0；
// Q 是尚未加入 T 的點，d[v] 表示將 v 加入 T 的最小成本
while (Q is not empty) do
    在 Q 中找出 d[] 最小的點 v；
    cost += d[v]； //
    將 v 移出 Q 中； // v 加入 T 中
```



```

for each u in adj[v] and in Q do: //與 v 相鄰尚未進入 T 者
  if w(v,u) < d[u] then // 從 v 連過來的成本更小
    d[u] = w(v,u);
    parent[u] = v; // u 加入 T 的最小成本是 w(v,u)
  end if
end for
end while

```

Prim 演算法與最短路徑的 Dijkstra 演算法只有比大小的 key 值不同，Dijkstra 是起點走過來的總距離，而 Prim 則只看最後與 parent 相連的邊長。實作時需要與 Dijkstra 一樣使用 PQ 的技巧，因為太相像了，我們就不多做說明，在後面的例題中，我們會給一個範例程式。

另外一個常用的是 Kruskal 演算法。這個做法也很簡單直覺，需要搭配的資料結構是併查集。演算法兩三句話就可以講完：將邊從小到大逐一考慮，如果一個邊加入不會形成環路就把它挑進來；否則將它捨棄。重複挑選邊直到沒有邊或者已經挑到 $n-1$ 個邊為止。

如果輸入的是連通圖，最小生成樹一定存在，否則全部的邊都挑完了仍無法形成生成樹。以下是演算法。

```

// Kruskal 演算法，輸入：G=(V, E, w)
初始化：設 T 為空；初始化一個併查集，每個點自成一個連通區塊；
將所有邊依照權重由小到大排序；
for each edge (u,v) do // 依權重由小到大的順序
  if u 與 v 不屬於同一個連通區塊 then // find()
    將 (u,v) 加入 T 中；
    將 u 與 v 所屬的連通區塊聯集； // union()
    // 如果 T 中已經有 n-1 個邊，可以提早結束
  end if
end for

```

下面一個簡單的例題用來介紹範例程式。

P-7-12. 最小生成樹 (*)

輸入一個無向圖 $G=(V, E, w)$ ，其中點以 $0\sim n-1$ 編號，而邊的權重是非負整數。計算 G 的最小生成樹的成本。兩點之間可能有多個邊。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 m ，代表點數與邊數，接下來有 m 行，每行三個整數 u, v, w 代表一條無向邊 (u, v) 的長度是 w 。 n 不超過 $1e4$ ， m 不超過 $1e5$ ， w 是不超過 $1e4$ 的非負整數。

輸出：輸出最小生成樹的成本，如果不存在則輸出 -1 。

範例一輸入：

```
8 10
0 1 6
0 2 4
1 2 5
2 3 9
1 4 1
1 5 1
2 6 2
4 5 3
5 6 8
7 6 1
```

範例一輸出：

```
23
```

範例二輸入：

```
4 3
2 1 5
1 0 0
0 2 1
```

範例二輸出：

```
-1
```

範例一是本小節說明所用的例子。

(範例測資中前兩筆的 n 不超過 500。)

Prim 演算法幾乎與 Dijkstra 一模一樣，就不多做說明。

```
// Prim algorithm O(mlogn) using PQ
#include<bits/stdc++.h>
using namespace std;
```

```

#define N 10010
#define oo 1000000001

vector<pair<int,int> > adj[N]; //Adjacency list
int d[N], parent[N];
bool done[N]={false};

int main() {
    int n, m;
    // input data
    scanf("%d%d", &n, &m);
    for(int i=0;i<m;i++) {
        int u,v,t;
        scanf("%d%d%d", &u, &v, &t);
        adj[u].push_back({v,t});
        adj[v].push_back({u,t});
    }
    // initial
    int source=0;
    for (int i=0;i<n;i++) d[i]=oo;
    for (int i=0;i<n;i++) parent[i]=-1;
    priority_queue<pair<int,int>> PQ; // (-d[], vertex)
    PQ.push({d[source]=0,source});
    while (!PQ.empty()) { // converge one vertex each iteration
        auto p=PQ.top(); // find min d[] of not-done vertex
        PQ.pop();
        int v=p.second;
        if (done[v]) continue; // already done, discard
        done[v] = true;
        // update neighbor
        for (auto e: adj[v]) {
            int u=e.first, w=e.second;
            if (done[u]) continue;
            if (w < d[u]) { // shorter via v
                d[u] = w;
                parent[u] = v;
                PQ.push({-d[u], u}); // push a better d[] of u
            }
        }
    }
    int cost=0, cnt=0;
    for (int i=0; i<n; i++) {
        //fprintf(stderr,"%d",d[i]); //fprintf(stderr,"\n");
        if (d[i]<oo) {
            cost+=d[i];
        } else cnt++;
    }
    if (cnt) printf("-1\n");
    else printf("%d\n", cost);
    return 0;
}

```

以下是 Kruskal 演算法的範例程式。因為要排序，所以我們定義了一個 struct 與一個比較函數用來存邊以及做排序。其他的部分就是根據演算法，對每一根邊檢查兩端是否是同一個區塊，若否，則加入該邊。

```

// Kruskal algorithm O(mlogn)
#include<bits/stdc++.h>
using namespace std;
#define N 10010
#define oo 1000000001
struct EDGE {
    int u,v,w;
};
bool cmp(EDGE &p, EDGE &q) {
    return p.w < q.w;
}
int p[N]; // parent
int rfind(int v) {
    if (p[v]<0) return v;
    return p[v]=rfind(p[v]);
}
// union two set, return true if really merge
bool joint(int u, int v) {
    int r1=rfind(u), r2=rfind(v);
    if (r1==r2) return false;
    if (p[r1]<p[r2]) {
        p[r1] += p[r2];
        p[r2] = r1;
    } else {
        p[r2] += p[r1];
        p[r1] = r2;
    }
    return true;
}
int main() {
    int n, m;
    // input data
    scanf("%d%d", &n, &m);
    vector<EDGE> edge;
    for(int i=0;i<m;i++) {
        int u,v,t;
        scanf("%d%d%d", &u, &v, &t);
        edge.push_back({u,v,t});
    }
    // initial
    for (int i=0;i<n;i++) p[i]=-1; // each vertex a set
    sort(edge.begin(), edge.end(), cmp); // sort by weight
    int cost=0, n_edge=0;
    for (EDGE e: edge) {
        if (joint(e.u, e.v)) { // insert an edge
            cost += e.w;
            n_edge++;
        }
    }
}

```

```
}  
if (n_edge < n-1) printf("-1\n");  
else printf("%d\n", cost);  
return 0;  
}
```

補充說明：不管邊是正的或是負的，最小生成樹的找法都是一樣，此外，邊的總和最大的最大生成樹以及最大的邊最小的生成樹都可以用類似的方法計算。

Bangye.Wu

8. 樹上演算法

「樹上的問題大部分都可以用 DP 解決，一個變數不夠，就用兩個。」

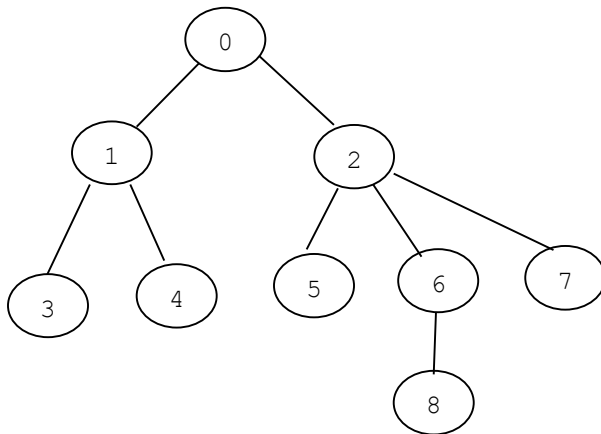
「好想抖著拖鞋在樹上打扣，從上往下看，好多問題都變簡單了。」

這一章介紹樹 (tree)，樹既然是一種圖，圖上的演算法例如 BFS 與 DFS 當然也都能用在樹上。但樹是特別稀疏的圖又有特殊的性質，所以很多問題在樹上有更好的解。另一方面，許多資料結構是使用樹的觀念建構出來的，本章探討的主要是輸入資料是一個樹的問題，而並非用樹做為資料結構的問題，這兩類問題並不相同。樹上的問題大部分都屬於貪心法則或是動態規劃，但需要搭配樹的結構來處理。

8.1. 基本觀念與名詞

回顧一下圖的基本定義，一個圖寫作 $G=(V, E)$ ，表示一個圖名字是 G ，由點的集合 V 與邊的集合 E 組成，其中 V 是點集合，也就是要研究的對象；而 E 是邊的集合，也就是誰與誰之間存在這種關係。樹 (Tree) 是一個連通而無環路的圖，通常指的是無向的簡單圖，也就是說若一棵樹 $T=(V, E)$ ，則 $m=n-1$ ，其中 n 與 m 是點數與邊數。

為了方便或需要，我們可以指定其中一點為根 (root)，稱為有根樹 (rooted tree)，一旦指定根之後，我們可以給圖上的邊訂定方向，每個邊的方向是由根往其他點，或者尤其它點往根的方向。通常把樹畫成一個家族圖，根在最上方，也就是說所有的邊都是一個上下兩點的關係，tree 所代表的二元關係就被比喻為 parent 與 child 的關係。下圖是一個有根樹，其中 0 是根。除了根以外，每個點的上方恰有一點與其相連，這個點稱為他的 parent，例如 4 與 3 的 parent 是 1；而 1 與 2 的 parent 是 0 (根)。若 a 是 b 的 parent，則稱 b 是 a 的孩子 (child)。只要指出每一個點的 parent 就可以惟一決定一棵樹。以下介紹一些基本的知識與名詞。

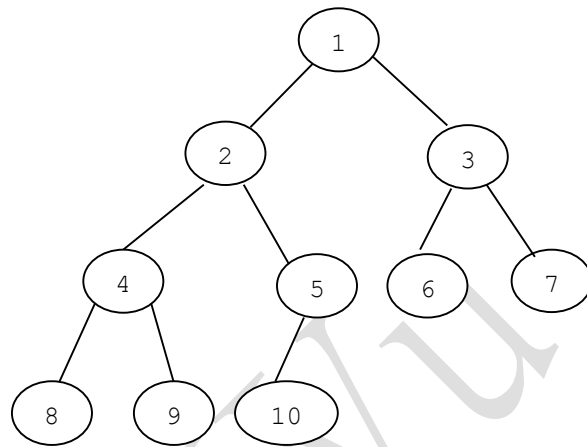


A rooted tree with 9 vertices and 8

- 樹的等價定義 (定理)：樹的定義是「連通無環路的圖」，以下的敘述也都可以證明所代表的是一棵樹。
 - 一個連通圖且 $n=m+1$ ；
 - 一個無環路的圖且 $n=m+1$ ；
 - 任兩點恰有一條路徑的圖；
 - 一個圖，點以 $1\sim n$ 編號，邊的集合是 $\{(i, p(i)) : p(i) < i, 2 \leq i \leq n\}$ 。也就是除了 1 號點之外，每個點連一條邊到編號比他小的點。這個圖其實就是個有根樹， $p(i)$ 就是 i 的 parent，每個點的編號都大於 parent 的編號。
- 每一個連通區塊都是一棵樹的圖就稱為森林 (forest)，也就是說一個森林由若干個不相交的樹組成，有 n 個點的森林，它的邊數是 $m=n-k$ ，其中 k 是連通區塊的個數。
- 在有根樹上，沒有孩子的點稱為樹葉 (leaf)，葉子也稱外節點 (external node)，不是葉子的稱為中間節點 (internal node)。每個點沿著 parent 往上一路走，都會走到 root，中間碰到的點都是它的祖先 (ancestor)。對於一個點 v ，所有以 v 為祖先的點，就是 v 的後代 (descendant)， v 與它的所有後代組成的圖就是以 v 為根的子樹 (subtree rooted at v)。在無根樹上，只有一個鄰居的點也稱為葉節點。
- 有根樹的分支度 (degree) 通常指每個點孩子數量的上限，degree 為 2 的樹 (也就是每個點最多兩個孩子) 稱為二元樹 (binary tree)，二元樹中每個節點的兩個孩子稱為左孩子與右孩子，以左 (右) 孩子為根的子樹就稱為左 (右) 子樹。二元

樹在資料結構中扮演重要的腳色，例如二元搜尋樹(Binary search tree)與堆積(heap)，但本章中用到的觀念不多，因此只做簡略的名詞介紹。

- 完整二元樹(Complete binary tree)是指一個二元樹它的葉節點只在最後兩層，而且如果葉節點在倒數第二層的話，一定在最右邊且連續出現，如右圖。完整二元樹如果用由上而下由左至右連續以1~n編號，可以用編號來找parent與child，因此可以用陣列儲存而無需用複雜的資料結構。在右圖的例子中，任一點 v 的左孩子是 $2v$ 而右孩子是 $2v+1$ ，如果這兩個孩子編號沒有超過 n ； v 的parent是 $v/2$ (整數除法)。堆積是以完整二元樹的概念建構的，常用來製作優先佇列。



- 二元搜尋樹(Binary search tree, BST)的每一個節點可以存一筆資料，每一筆資料有一個鍵值(key)作為排序的依據，BST的key滿足以下規則：對任何一點 v ，在它左子樹中的點，鍵值必定小於它；在它右子樹中的點，鍵值必定大於它。BST有不同的變化形式，是很重要到的動態資料結構，一個平衡的BST通常樹高不超過 $O(\log(n))$ ，STL中的set/map通常以平衡的BST建構。

一些其他名詞我們在後面遇到時再說明。因為樹沒有環路，如果從樹上刪掉任意一點，會造成一些獨立的子樹，既然是獨立不相交的，往往就可以獨自求解，然後在該點上合併子問題的解，這事實上就是分治的概念，如果從非遞迴的角度，它就是動態規劃(DP)。實際解題時，為了方便，通常把樹當成有根樹來做，而實作就區分為top-down(遞迴)與bottom-up(非遞迴)兩種方式。所以說樹上的問題絕大部分都可以用DP來解，以下我們會看到很多例子。

8.2. 樹的儲存與基本演算法

在這一節中我們介紹儲存樹的基本資料結構與幾個基本演算法。

8.2.1. 儲存樹的資料結構

樹是一種圖，儲存圖的資料結構當然可以用來存樹，但是樹是非常稀疏的圖，所以用鄰接矩陣是非常不合適的。如果是無根樹，用鄰接串列是合適的方式。在有根樹時，我們

還有兩種儲存方式：存 parent 以及存 child。使用上當然要看輸入給的方式，也要看程式的寫法。基本上三種方式可以互相轉換。

因為 parent 與 child 是相互的關係，如果題目給的是每個點的 parent (除了 root)，要轉成 child 的形式，就讓每一個點把自己放進 parent 的 child[]；如果題目給的是 child，就由每一個 parent 設定自己孩子的 parent。以下是簡單的範例，我們寫了兩個函數，一個從 parent 轉成 child，另外一個從 child 轉成 parent。其中要注意的是，如果 degree 沒有限定，child[] 要使用 vector 來做才比較方便 (因為陣列長度未知)。另外轉到 parent 時需要留意，因為 root 將來是沒有 parent，如果不確定 root 的編號，初設時要將 parent 設成不可能的值 (如 -1)，那麼在轉換後才能找出 root。

```

// tree, data structure demo, parent and child
#include <bits/stdc++.h>
using namespace std;
#define N 1000
int parent[N];
vector<int> child[N]; // list of children

// parent to child
void p2c(int n) {
    for (int i=0; i<n; i++)
        child[i].clear();
    for (int i=0; i<n; i++) // set i as a child of parent[i]
        child[parent[i]].push_back(i);
    return;
}

// child to parent, parent[root] = -1, return root
int c2p(int n) {
    for (int i=0; i<n; i++)
        parent[i] = -1; // -1 as no parent
    for (int i=0; i<n; i++) {
        for (int c: child[i]) // each child of i
            parent[c] = i;
    }
    for (int i=0; i<n; i++) // find root
        if (parent[i]==-1)
            return i;
}

```

child 或 parent 轉成無根樹的连接串列形式非常簡單，因為一個點的鄰居就是 parent 以及所有的 child。另一方面，要將無根樹的连接串列轉成有根樹的 child 或 parent 的形式則須要做一個 BFS 或 DFS 走訪，以下介紹 BFS 與 DFS。

8.2.2. DFS

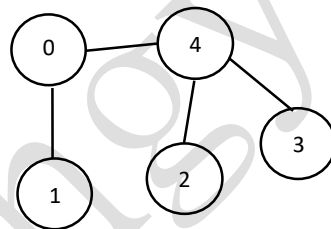
樹的 DFS 相當重要，大部分樹的題目都可以用 DFS 來解。樹既然是圖的一種，當然可以用圖的 DFS 來做樹的 DFS，但另一方面樹是一種特殊而簡單的圖，所以可以更簡單的來做。有根樹與無根樹的走訪很類似，我們先用一個例題來示範無根樹的 DFS。

P-8-1. 樹上的推銷員

有一個推銷員要走訪 n 個城市並在結束後回到出發的城市。這些城市以 $n-1$ 條道路連接，每條道路連接兩個不同的城市並且雙向可以通行，而且已知每一個城市都可以到達，不會有到達不了的狀況。輸入道路的資料，請你找出一個長度最短的程式走訪順序，因為這樣的順序很多，你必須輸出字典順序最小的那一個。字典順序是用來比較兩個序列的先後順序，其方法是由前往後找到第一個不相同的位置，該元素比較小的序列就是順序比較小，如同在英文字典中單字的順序一般。例如

$(0, 2, 3, 1) < (0, 3, 1, 2)$ ，又 $(0, 3, 2, 1) < (1, 0, 2, 3)$ 。

舉例來說，如右圖，有 5 個城市以 4 條道路連接，假設每條道路的長度都是 1。字典順序最小的最短拜訪順序是 $(0, 1, 0, 4, 2, 4, 3, 4, 0)$ 。



Time limit: 1 秒

輸入格式：第一行是正整數 n ，代表城市的數量，城市是以 $0 \sim n-1$ 編號，接下來有 $n-1$ 行是道路的資料，每行三個整數 a, b 與 w ，代表此道路連接城市 a 與 b ，道路的長度是 w 。 n 不超過 50000，每條道路長度是不超過 100 的正整數。

輸出：第一行輸出最短的旅行距離，第二行輸出拜訪順序，相鄰的數字之間要以一個空白間格。

範例一輸入：	範例一輸出：
5	8
1 0 1	0 1 0 4 2 4 3 4 0
0 4 1	
2 4 1	
4 3 1	

範例二輸入：	範例二輸出：
3	16
1 2 5	0 2 1 2 0
0 2 3	

n 個城市以 $n-1$ 條道路連接而且每個城市都可以彼此到達，顯示是一個 n 個點 $n-1$ 個邊的連通圖，所以它是個樹。旅行推銷員問題在一般圖上是很難解的問題，我們在第 6 章 DP 中有見過，但是在樹上卻相當簡單：最短的環路一定每個邊恰好走兩次。這可以用歸納法很簡單的證明，想像 v 是一個葉節點， P 是走訪其他 $n-1$ 個城市的最短環路，我們只要沿著 P 走，走到 v 的惟一鄰居時，繞過去 v 就行了。

所以任何一個 DFS 走的順序都是最短的環路，這一題的問題其實是要找字典順序的 DFS。在做圖的 DFS 時，有一個重點是對每個點需要設定一個 `visit[]` 值，來避免重複走訪到已經走過的點，但樹的 DFS 可以省略這個動作，取而代之的是遞迴呼叫 v 時，傳過去從哪個點 (p) 呼叫它，因為樹沒有環路，只要不要遞迴呼叫 p ，就不會重複走訪到同一點。請看下面的範例程式：

第 4 行的 `adj[v]` 用來存每個點 v 的鄰居，第 5 行的 `tour` 是用來放答案的環路。第 7~14 行就是 DFS 的遞迴函數。每次走到 v 我們先將 v 放入 `tour` 當做拜訪 v 的動作，接著對它的每個鄰居 u (除了來時路的 p)，遞迴呼叫 `dfs(v,u)` 以拜訪 u 點以下所有點，然後回到 v 。主程式部分，第 17~24 行是輸入，我們把邊長另外加總處理。輸入完畢後，先將每個點的鄰居依照編號排序，然後以 `dfs(-1,0)` 作為以 0 為起點的走訪。最後，第 30~34 行是輸出答案。

```

00 // lexicographic first dfs
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 50010
04 vector<int> adj[N]; // neighbor
05 vector<int> tour;
06 // dfs from v, parent=p
07 void dfs(int p, int v) {
08     tour.push_back(v);
09     for (int u: adj[v]) { // dfs each neighbor
10         if (u==p) continue; // except for parent
11         dfs(v,u);
12         tour.push_back(v);
13     }
14 }
15 int main() {
16     int n, total=0;
17     scanf("%d", &n);

```

```

18     for (int i=0;i<n-1;i++) {
19         int u,v, w;
20         scanf("%d%d%d", &u, &v, &w);
21         adj[u].push_back(v);
22         adj[v].push_back(u);
23         total += w; // total edge weight
24     }
25     for (int i=0; i<n; i++)
26         sort(adj[i].begin(), adj[i].end());
27     dfs(-1,0); // starting at 0, no parent
28     assert(tour.size()==2*n-1);
29     // output
30     printf("%d\n", total*2);
31     printf("%d", tour[0]);
32     for (int i=1; i<tour.size();i ++ )
33         printf(" %d", tour[i]);
34     printf("\n");
35     return 0;
36 }

```

這一題以無根樹的走訪來做就可以，但是為了示範有根樹的 DFS，我們也寫一個將無根樹轉換為有根樹之後再做 DFS 的範例程式，請看以下程式。

誠如上面所說，這個程式基本上是 DFS 走兩次。第 8~16 行的 `root(p,v)` 是做將樹以 `v` 點為 `root` 吊起來的動作，除了 `p` 以外的所有鄰居都變成 `v` 的 `child`。並且對每一個 `v` 的孩子 `u`，遞迴呼叫 `root(v,u)`，這樣完成了無根樹轉成有根樹的動作。

然後在第 35 行將各點的 `child` 排序後，呼叫 `tsp(0)` 來進行以 0 點開始的 DFS 走訪。因為已經轉成有根樹，遞迴走訪的函數 `tsp(v)` 不需要指名 `parent` 是誰，先拜訪 `v` 點後對每一個孩子 `ch`，進行遞迴拜訪 `tsp(ch)` 以及回到 `v` 的動作。

```

00 // lexicographic first dfs. transform to rooted tree
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 50010
04 vector<int> adj[N]; // neighbor
05 vector<int> child[N];
06 vector<int> tour;
07 // rooting tree at from v, parent=p
08 void root(int p, int v) {
09     // except for parent, all neighbors are children
10     for (int u: adj[v]) {
11         if (u!=p) {
12             child[v].push_back(u);
13             root(v,u);
14         }
15     }
16 }

```

```

17 void tsp(int v) {
18     tour.push_back(v);
19     for (int ch: child[v]) { // for each branch
20         tsp(ch);
21         tour.push_back(v);
22     }
23 }
24 int main() {
25     int n, total=0;
26     scanf("%d", &n);
27     for (int i=0;i<n-1;i++) {
28         int u,v, w;
29         scanf("%d%d%d", &u, &v, &w);
30         adj[u].push_back(v);
31         adj[v].push_back(u);
32         total+=w;
33     }
34     root(-1,0);
35     for (int i=0; i<n; i++) // sort children
36         sort(child[i].begin(), child[i].end());
37     tsp(0);
38     assert(tour.size()==2*n-1);
39     printf("%d\n",total*2);
40     printf("%d",tour[0]);
41     for (int i=1;i<tour.size();i++)
42         printf(" %d",tour[i]);
43     printf("\n");
44     return 0;
45 }

```

這是本章第一個樹的 DFS，請務必了解，因為樹的問題大部分都是利用 DFS 的架構去做的。

8.2.3. BFS

樹上的 BFS 與 DFS 一樣的都是由起點往外走，如果把起點當 root，那麼兩者走出來的順序固然不同，但都是一個 top-down 的順序，也就是任何一點都在它的孩子之前被拜訪，如果把這個順序反過來，就是一個 Bottom-up 的順序。樹的問題有很多都是 DP 或貪心法則，而都需要一個 top-down 或 bottom-up 的順序，所以通常用 DFS 的比較多，原因是 DFS 比較好寫。

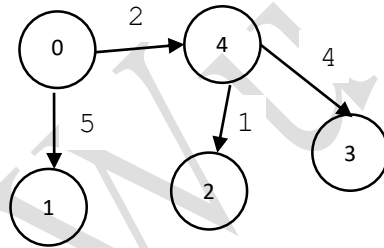
在圖的 BFS 與 DFS 我們已經看過了，BFS 需要使用一個 queue，而 DFS 只要遞迴就好了，上一節我們也看到樹的 DFS 還更簡單一點，競賽與考試的時候，程式簡潔好寫是一個極大的優點，所以寫 DFS 的比較多。在圖的時候，BFS 還有它的必要，例如在無加權圖上求起點到個點距離，這個理由在樹上也不見了，因為樹沒有環路，不管是加權圖或

是非加權圖，都可以用 DFS 來計算起點到個點的距離。結論是樹上用 BFS 的機會比較少，如果遞迴可用，通常用 DFS；只有因為某些因素遞迴不可用（例如環境限制遞迴深度），而需要 top-down 順序時才會使用。

我們以起點到各點的距離來展示樹上的 BFS。

P-8-2. 物流派送系統

有一個物流派送系統，系統有 n 個工作站與 $n-1$ 條輸送帶，工作站以 $0 \sim n-1$ 編號，其中 0 號站是進貨站，所有物品都是由 0 號站進入系統然後經過輸送帶配送到某個站。每個輸送帶都是單向的從某個站 a 把物品送到站 b ，輸送的時間是 $w(a, b)$ 。已知由進貨站可以經由輸送帶直接或間接轉送到達任何其他站，請計算下列兩者 (1). 由進貨站到達運送時間最長的站需要多少時間，以及 (2). 由進貨站需要最多次轉運的需要經過幾次輸送帶。



舉例來說，如右圖，有 5 個站以 4 條輸送帶連接，假設每條輸送帶的運送時間如圖上標示。需要最長運送的時間是 3 號站 ($2+4=6$)，最多需要經過的輸送帶是兩次 (2 號站與 3 號站)。

Time limit: 1 秒

輸入格式：第一行是正整數 n ， $n > 1$ ，接下來有 $n-1$ 行是輸送帶的資料，第 i 行 2 個整數 x 與 w ，代表此輸送帶連接 x 與 i ，輸送的時間是 w 。 n 不超過 $1e5$ ，每條輸送帶的輸送時間是不超過 1000 的正整數。

輸出：第一行輸出最長的運送總時間，第二行輸出最多經過幾次輸送帶。第一行與第二行的答案不一定是同一個站，除了輸送帶以外的時間均忽略不記。

範例一輸入： 5 0 5 4 1 4 4 0 2	範例一輸出： 6 2
範例二輸入： 6 0 9 0 1 2 1 5 3 3 2	範例二輸出： 9 4

範例一：題目中的例子

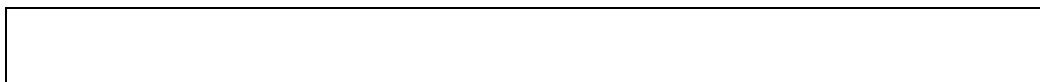
範例二：時間最久的是 0->1，時間是 9；最多次的是 0->2->3->5->4，共四次。
(測資 1 與 2 的 $n \leq 100$ 且前一站的編號必然小於該站。)

工作站是點，輸送帶是邊， n 個點 $n-1$ 個邊，而 0 可以到達所有的點，這個圖形就是一個有根樹，邊有長度，第一個要求的是根到各點的最大距離，第二個是經過最多次的邊，那也就是無加權圖的最大距離。輸入給的形式是每個點的前一站，也就是 parent。

在樹上計算根(0)到所有 v 的距離其實不過就是個簡單的 DP，遞迴式

$$d[v] = d[\text{parent}[v]] + w(\text{parent}[v], v) \text{ for all } v > 0$$

所以只要找出所有點都出現在它 parent 之後的 top-down 序列，然後根據遞迴式計算即可，以下是這樣寫的範例程式。在輸入的時候，除了 parent 之外，我們也轉存每個點的 child(第 14 行)，各點 i 到它 parent 之間的邊長則存在 $\text{leng}[i]$ 。第 17 行宣告一個 `vector<int>` 當作 queue 來使用，然後開始一個 BFS 走訪，因為是 tree，我們不需要記錄該點是否已拜訪過，所以 while 迴圈中只是簡單的從 queue 中取出一點，然後將它的孩子都放進 queue 中。接著第 28~33 行計算最遠的加權距離，只要沿著順序套遞迴式計算就可以了，最後是無加權圖的距離，我們放在 $\text{step}[v]$ ，它的算法相同，只是每個邊都算 1 罷了。



```

00 // longest distance d(0,v), BFS, two pass
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 100010
04 vector<int> child[N]; // child of i
05 int parent[N], leng[N];
06 int main() {
07     int n;
08     scanf("%d", &n);
09     for (int i=1;i<n;i++) {
10         int v, w;
11         scanf("%d%d", &v, &w);
12         parent[i]=v;
13         leng[i] = w; // length of edge (i, parent[i])
14         child[v].push_back(i);
15     }
16     // find a top-down sequence
17     vector<int> que;
18     int head=0; // a queue
19     que.push_back(0);
20     while (head < que.size()) {
21         int v=que[head++];
22         for (int ch: child[v])
23             que.push_back(ch);
24     }
25     assert(que.size()==n); // otherwise not a tree
26     // compute distance in top-down sequence
27     int d[N];
28     d[0]=0;
29     for (int i=1; i<n; i++) {
30         int v = que[i];
31         d[v] = d[parent[v]] + leng[v];
32     }
33     printf("%d\n", *max_element(d, d+n));
34     // compute max step, using range-based for
35     for (int v: que) {
36         if (v>0) // 0 has no parent
37             d[v] = d[parent[v]] + 1;
38     }
39     printf("%d\n", *max_element(d, d+n));
40     return 0;
41 }

```

其實找順序與計算距離可以在同一回合做，就像標準的 BFS 一樣。以下就是這樣的寫法。

```

// longest distance d(0,v), BFS, one pass
#include <bits/stdc++.h>
using namespace std;
#define N 100010
vector<int> child[N]; // child of i

```



```

int parent[N], leng[N];
int main() {
    int n;
    scanf("%d", &n);
    for (int i=1;i<n;i++) {
        int v, w;
        scanf("%d%d", &v, &w);
        parent[i]=v;
        leng[i] = w; // length of edge (i, parent[i])
        child[v].push_back(i);
    }
    // find a top-down sequence
    vector<int> que; // a queue
    int d[N], step[N], head=0, longest=0, max_step=0;
    d[0]=0, step[0]=0;
    que.push_back(0);
    while (head < que.size()) {
        int v=que[head++];
        for (int ch: child[v]) {
            que.push_back(ch);
            d[ch] = d[v] + leng[ch];
            longest = max(longest, d[ch]);
            step[ch] = step[v] + 1;
            max_step = max(max_step, step[ch]);
        }
    }
    assert(que.size()==n); // otherwise not a tree
    printf("%d\n%d\n", longest, max_step);
    return 0;
}

```

如同前面說的，tree 上找距離也可以用 DFS，DFS 用遞迴也不需要 queue，所以確實更簡潔。

```

// longest distance d(0,v), DFS
#include <bits/stdc++.h>
using namespace std;
#define N 100010
vector<int> child[N]; // child of i
int leng[N]; // length to its parent
int d[N], step[N]; // distance and number of step to v
int longest=0, max_step=0; // answer
void dfs(int v) {
    for (int ch: child[v]) {
        d[ch] = d[v] + leng[ch];
        longest = max(longest, d[ch]);
        step[ch] = step[v] + 1;
        max_step = max(max_step, step[ch]);
        dfs(ch);
    }
    return;
}

```

```

}

int main() {
    int n;
    scanf("%d", &n);
    for (int i=1;i<n;i++) {
        int v, w;
        scanf("%d%d", &v, &w);
        leng[i] = w; // length of edge (i, parent[i])
        child[v].push_back(i);
    }
    d[0]=0, step[0]=0;
    dfs(0); // DFS from root 0
    printf("%d\n%d\n", longest, max_step);
    return 0;
}

```

8.2.4. 由下而上的走訪

由下而上計算是很多樹的演算法所需要的順序，如果遞迴可以用，通常優先使用 DFS，如果不行，就必須自己找一個 Bottom-up 順序。非遞迴的找此順序可以採用上一節的 BFS，因為將 top-down 順序反轉過來就是 bottom-up 順序。但這樣需要先找順序再做計算，通常要兩回合。有時我們希望一個回合直接一面找順序一面做計算，我們在 P-3-1 已經介紹過這樣的做法了，其實就是把樹看成一個由下而上的 DAG。我們準備一個 queue 放置已經可以出列的點，一開始把所有葉節點都放進 queue，每一回合從 queue 中取出一點，將它的 parent 的孩子數減一，若它的 parent 變成葉子，就放進 queue 中。

我們以下一個題目來展示 bottom-up 序列的計算與運用。

P-8-3. 購物中心選位置

有 n 個小村莊以 $0 \sim n-1$ 編號，其中編號 0 的是市政府所在地，這些村莊以 $n-1$ 條道路連接，每條道路都是雙向通行且連接兩個不同的村莊，已知從任一個村莊 i 都可以經由這些道路到達市政府所在的 0 號村莊，所以也可以到達任何其他村莊，而且由村莊 i 往市政府會經過的第一站村莊是 $p(i)$ 。現在某大集團希望選其中一個村莊來蓋一個購物中心，選址的判斷方式是購物中心到所有村莊的距離總和要越小越好，請你幫忙計算哪一個村莊是最好的位置，到各村莊的距離總和最小是多少。

Time limit: 1 秒

輸入格式：第一行是正整數 n ， $n > 1$ ，村莊以 $0 \sim n-1$ 編號。接下來有 $n-1$ 行是道路的資料， i 由 1 到 $n-1$ ，第 i 行有 2 個整數 $p(i)$ 與 $w(i)$ ，代表 i 往 0 號村莊的道路第一個會經過 $p(i)$ ，而 i 與 $p(i)$ 之間的道路長度是 $w(i)$ 。 n 不超過 $1e5$ ，每條道路的長度是不超過 1000 的正整數。

輸出：第一行輸出最佳的村莊編號，如果有超過一個村莊都是最佳位置，則找離市政府最遠的那一個，第二行輸出最佳村莊到各村莊距離總和。

範例一輸入： 5 0 5 4 1 4 4 0 2	範例一輸出： 4 14
範例二輸入： 6 0 9 0 1 2 1 5 3 3 2	範例二輸出： 3 21

範例一說明：最佳位置是 4 號村莊，4 到其它村莊的距離總和是 14。

範例二說明：2 與 3 到其它的距離總和都是最小的 21，而 3 離 0 的距離比 2 與 0 的距離遠。

這些村莊與道路構成的圖顯然是個樹， $p(i)$ 就是 i 的 parent，0 號點是 root。令 $D(v)$ 是點 v 到所有其他點的距離總和，對任一點 v ，我們可以在 $O(n)$ 的時間計算出 $D(v)$ ，所以對每一點 v 都求出 $D(v)$ 來再找最小值的方法可以在 $O(n^2)$ 完成，但是還是不夠快，因為本題要求的 n 是 $1e5$ 。

假設這些點是在一條數線上，也就是 n 個數字要找其中一個與其它數字的距離總和最小，那麼答案是這 n 個數字的中位數 (median)，也就是排序後位置在中間的那個數字，如果 n 是偶數，則中間那兩個都是解 (到各點距離相同)。這個證明其實很簡單，對於任意一點，如果他的左右有一邊的點數超過 $n/2$ ，例如左邊有 x 個點， $x > n/2$ ，則將點

往左邊移動長度 y ，左方 x 個點距離減少 xy ，右邊 $(n-x)$ 個點距離增加 $(n-x)y$ ，總距離一定會減少。所以所選的點左右兩邊的點數都不會超過 $n/2$ 。

在圖上與其它點距離總和最小的點也是稱為 median。樹的 median 與數字的 median 有一樣的性質，事實上數線上的點就是樹的退化情形。若 v 是 $D(v)$ 最小的 v 點，則 v 的每一個分枝的點數都不會超過 $n/2$ 。這一題的輸入給的方式是以 0 為 root 的有根樹，因此找 median 其實是要找子樹的點數。令 $\text{num}(v)$ 是以點 v 為根的子樹的點數，也就是 v 的後代總數（含 v 自己），我們要找的是“最低”的某個點 s ，滿足 $\text{num}(s) \geq n/2$ ，最低的意思是，對所有 s 的孩子 c ， $\text{num}(c) < n/2$ 。如果 $\text{num}(s)$ 恰好是 $n/2$ ，則 s 與其 parent 都是 median，根據本題的題意，此時要輸出比較低的那一個 median。

所以本題最主要的是算各子樹的大小 $\text{num}(v)$ ，算完之後，可以依照定義找某個點，它的孩子的子樹都不到 $n/2$ 而它的子樹不小於 $n/2$ ，最後再計算到其他點距離。至於計算各子樹的大小的方法，以下是一種想法：因為每個點都是它每個祖先的後代，沿著 parent 一路往上走碰到的都是祖先，所以，我們可以對每個點都把他加到它所有的祖先中。以下是這樣寫出來的程式，主程式一開始先做輸入，第 25~32 是算 $\text{num}[]$ ；第 34~41 行是依照定義找 median；最後用 DFS 來計算距離。

```

00 // P-8-3 tree median, O(nh) method
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 100010
04 vector<pair<int,int>> adj[N]; // (neighbor, length) of i
05 int parent[N], d[N]={0};
06 // compute distance
07 void dfs(int v, int p) {
08     for (auto e: adj[v]) {
09         if (e.first != p) {
10             d[e.first] = d[v] + e.second;
11             dfs(e.first, v);
12         }
13     }
14 }
15 int main() {
16     int n;
17     scanf("%d", &n);
18     for (int i=1; i<n; i++) {
19         int w;
20         scanf("%d%d", &parent[i], &w);
21         adj[i].push_back({parent[i], w});
22         adj[parent[i]].push_back({i, w});
23     }
24     parent[0]=-1; // root has no parent
25     int num[N]={0}; // number of node in the subtree
26     for (int i=0; i<n; i++) { // for each vertex

```

```

27     int v=i;
28     while (v>=0) { // add itself to all its ancestors
29         num[v]++;
30         v=parent[v];
31     }
32 }
33 // find median
34 bool flag[N]={false}; // if having a child >=n/2;
35 for (int i=1; i<n; i++)
36     if (num[i]>=n/2) flag[parent[i]]=true;
37 int median=0;
38 for ( ; median<n; median++) {
39     if (num[median]>=n/2 && !flag[median])
40         break;
41 }
42 // compute d[v] the distance from median to v
43 int total=0;
44 dfs(median, -1);
45 for (int i=0; i<n; i++)
46     total += d[i];
47 printf("%d\n%d\n", median, total);
48 return 0;
49 }

```

這個程式的主要缺點是算得太慢，複雜度太差，如果樹的高度是 h ，需要的時間是 $O(nh)$ ，所以在樹高是 $O(n)$ 時，它的時間複雜度是 $O(n^2)$ 。其實，根據定義： $num(v) = 1 + \sum_{u \in CHILD(v)} num(u)$ ，因此計算 $num(v)$ 不過就是個類似於前綴和的簡單 DP，只要走一個 bottom-up 的順序，當一個點的子樹大小算好了之後，把它加到 parent 的子樹大小中就可以了。這樣做足夠改善複雜度，但是程式還是比較複雜囉嗦，子樹大小都算好了，是不是可以用來計算第三步驟要算的距離呢？

若 m 是 median，我們要算 m 到所有點的距離和，直接的方法當然是把每一條路徑都算出來再加總，其實在樹上有另外一種計算方式。考慮任一條邊 (u, v) ，其中 v 是 u 的 parent，邊 (u, v) 把樹分成兩塊，姑且稱為下方與上方，點 m 在 (u, v) 的某一方，那麼，哪些點到 m 的路徑會經過 (u, v) 呢？當然是 m 另外一方的點才需要跨過此邊，所以我們只要知道另外一方有幾點，就知道 (u, v) 在距離總和中會被計算多少次，對每個邊都這樣計算後加總就是距離總和了。那麼， m 在哪一方？根據 median 的性質， m 一定在點多的一方！！我們在計算 $num[]$ 時算出了每一個子樹的大小，所以我們知道 $num[u]$ ，不管 m 在上方或下方，這條邊被計算的次數是 $\min(num[u], n - num[u])$ 。有了這個了解，我們就可以直接算距離總和了。以下是用 bottom-up 順序做的範例程式。

輸入的時候順便計算每個點的孩子數量 $deg[]$ ，第 15 行準備一個佇列 Q 放可以出列的點，初始時把葉節點 ($deg[] = 0$) 放入 Q 中，然後跑一個 while 迴圈做 bottom-up 走訪。每次迴圈從佇列中拿出一點 v 來，因為它一定在它的所有孩子之後出列，所以它的

孩子們的子樹大小都已經加到它的 `num[v]` 中了，因為我們尚未算 `v` 自己，所以第 22 行將 `num[v]++`，第 23 行檢查它是否是第一個超過 $n/2$ 的子樹，初始我們將 `median` 設為 `-1`，所以第 23 行的 `if` 成立時，`v` 就是最低的（也就是我們要找的）`median`。第 25 行我們計算到 `median` 的距離總和中這個邊會被計入的量，接著，將它的 `num[v]` 加入 `parent` 的子樹大小中，並檢查 `parent` 是否要進入 `Q` 中。這個程式要簡潔多了，時間複雜度是 $O(n)$ 。

```

00 // P-8-3 tree median, Bottom-up method, O(n)
01 #include <bits/stdc++.h>
02 using namespace std;
03 #define N 100010
04
05 int main() {
06     int n, parent[N], leng[N];
07     int deg[N]={0}, num[N]={0};
08     scanf("%d", &n);
09     for (int i=1;i<n;i++) {
10         scanf("%d%d", &parent[i], &leng[i]);
11         deg[parent[i]]++;
12     }
13     parent[0]=-1; // root has no parent
14     int median=-1, total=0;
15     queue<int> Q;
16     for (int i=0; i<n; i++)
17         if (deg[i]==0) Q.push(i);
18     // bottom-up traversal
19     while (!Q.empty()) {
20         int v=Q.front();
21         Q.pop();
22         num[v]++; // add itself
23         if (median<0 && num[v]>=n/2)
24             median = v; // first >=n/2 is median
25         total += min(num[v], n-num[v]) * leng[v];
26         // check parent
27         if (parent[v]<0) break; // root
28         num[parent[v]] += num[v];
29         if (--deg[parent[v]] == 0)
30             Q.push(parent[v]);
31     }
32     printf("%d\n%d\n", median, total);
33     return 0;
34 }

```

如果用遞迴當然程式碼更簡潔一點，遞迴函數中，記得要先呼叫孩子後，才能把它的值拿來使用。

```

// P-8-3 tree median, dfs method, O(n)

```

```

#include <bits/stdc++.h>
using namespace std;
#define N 100010
vector<pair<int,int>> child[N];
int n, num[N]={0}, median=-1, total=0;
void dfs(int v) {
    for (auto e: child[v]) {
        int u=e.first, w=e.second;
        dfs(u);
        total += min(num[u], n-num[u]) * w;
        num[v] += num[u];
    }
    num[v]++; // add itself
    if (median<0 && num[v]>=n/2)
        median = v;
    return;
}

int main() {
    scanf("%d", &n);
    for (int i=1;i<n;i++) {
        int v, w;
        scanf("%d%d", &v, &w);
        child[v].push_back({i,w});
    }
    dfs(0);
    printf("%d\n%d\n", median, total);
    return 0;
}

```

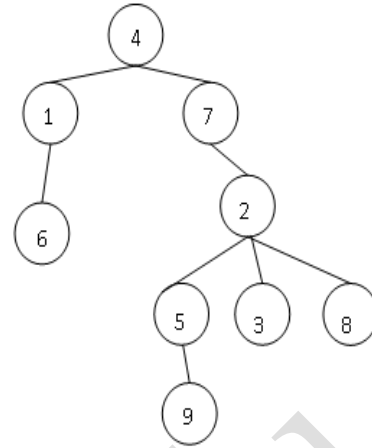
8.3. 例題與習題

在這一節中我們介紹一些典型的樹狀圖的題目，不外乎貪心法或遞迴，也有些是模擬或走訪。下一題曾經出現在 P-3-1，當時是介紹由下而上的走訪方式，而且還沒介紹什麼是 DP，在學過 DP 以及樹的 DFS 之後，這一題應該變得簡單很多了。

P-8-4. 樹的高度與根 (同 P-3-1) (APCS201710)

有一個大家族有 N 個成員，編號 $1 \sim N$ ，我們請每個成員寫下此家族中哪幾位是他的孩子。我們要計算每個人有幾代的子孫，這個值稱為他在這個族譜中的高度，也就是說，編號 i 的人如果高度記做 $h(i)$ ，那麼 $h(i)$ 就表示在所有 i 的子孫中，最遠的子孫與他隔了幾代。本題假設除了最高的一位祖先 (稱為 root) 之外，其他成員的 parent 都在此家族中 (且只有一個 parent)。本題要計算所有成員高度的總和以及根的編號。

右圖是一個例子，每個成員是一個圈起來的號碼，劃線相聯的表示上面的點是下面點的 parent。編號 4 是根，有兩個孩子 1 與 7。編號 6, 9, 3, 8 都沒有孩子， $h(6)=h(9)=h(3)=h(8)=0$ ，此外 $h(2)=2$ 因為 9 與他隔兩代。你可以看出來 $h(5)=h(1)=1$ ， $h(7)=3$ ， $h(4)=4$ ，所以高度總和是 11。



Time limit: 1 秒

輸入格式：第一行有一個正整數 N 。接下來有 N 行，第 i 行的第一個數字 k 代表 i 有 k 個孩子，第 i 行接下來的 k 個數字就是孩子的編號。每一行的相鄰數字間以空白隔開。 $N \leq 1e5$ 。

輸出：第一行輸出根的編號，第二行輸出高度總和。

範例輸入：

```

9
1 6
3 5 3 8
0
2 1 7
1 9
0
1 2
0
0

```

範例結果：

```

4
11

```

因為一個點的高度，是他所有孩子最高的高度加一，這是一個遞迴定義，使用 DFS 來做就非常簡單，對每一個孩子呼叫一次高度計算，然後在所有孩子中取最大值就可以了。

```

// tree height, dfs version, using vector
#include <bits/stdc++.h>
using namespace std;
#define N 100010
int height[N]; // height of node i
vector<int> child[N]; // children of node i

```



```

// recursively find height of r
void findh(int r) {
    height[r] = 0;
    for (int v: child[r]) {
        findh(v); // recursive call
        height[r] = max(height[r], height[v]+1);
    }
}

int main() {
    int i, j, n, root;
    int parent[N]={0}; // parent of node i
    scanf("%d", &n);
    for (i=1; i<=n; i++) {
        int deg, ch;
        scanf("%d", &deg);
        for (j=0; j<deg; j++) {
            scanf("%d", &ch);
            child[i].push_back(ch);
            parent[ch] = i;
        }
    }
    // find root, the node without parent
    for (root=1; root<=n; root++)
        if (parent[root] == 0) break;
    findh(root);
    int total_h = 0;
    for (i=1; i<=n; i++) total_h += height[i];
    printf("%d\n%d\n", root, total_h);
    return 0;
}

```

如果只做純遞迴，時間複雜度會變成 $O(nh)$ ，其中 h 是樹的高度，下面是一個寫壞的方式。

```

// tree height, puring recursion, O(nh)
#include <bits/stdc++.h>
using namespace std;
#define N 100010
//int height[N]; // height of node i
vector<int> child[N]; // children of node i
// recursively find height of r
int findh(int r) {
    int h=0;
    for (int v: child[r])
        h = max(h, findh(v)+1); // recursive call
    return h;
}

int main() {
    int i, j, n, root;

```

```

int parent[N]={0}; // parent of node i
scanf("%d", &n);
for (i=1; i<=n; i++) {
    int deg, ch;
    scanf("%d", &deg);
    for (j=0; j<deg; j++) {
        scanf("%d", &ch);
        child[i].push_back(ch);
        parent[ch] = i;
    }
}
// find root, the node without parent
for (root=1; root<=n; root++)
    if (parent[root] == 0) break;
int total_h = 0;
for (i=1; i<=n; i++)
    total_h += findh(i);
printf("%d\n%d\n", root, total_h);
return 0;
}

```

純遞迴的時間複雜度會差，是因為我們沒有把做過的事情用表格記錄下來，如果用 top-down DP 的觀念，使用表格紀錄，即使對每個點呼叫一次，時間複雜度依然只有 $O(n)$ 。下面是懂樹但是懂 top-down DP 的寫法。

```

// tree height, top-down DP, O(n)
#include <bits/stdc++.h>
using namespace std;
#define N 100010
int height[N]; // height of node i
vector<int> child[N]; // children of node i
// top-down memo DP to find height of r
int findh(int r) {
    if (height[r]>=0) return height[r];
    height[r]=0;
    for (int v: child[r])
        height[r] = max(height[r], findh(v)+1); // recursive call
    return height[r];
}

int main() {
    int i, j, n, root;
    int parent[N]={0}; // parent of node i
    scanf("%d", &n);
    for (i=1; i<=n; i++) {
        int deg, ch;
        scanf("%d", &deg);
        for (j=0; j<deg; j++) {
            scanf("%d", &ch);
            child[i].push_back(ch);
        }
    }
}

```

```

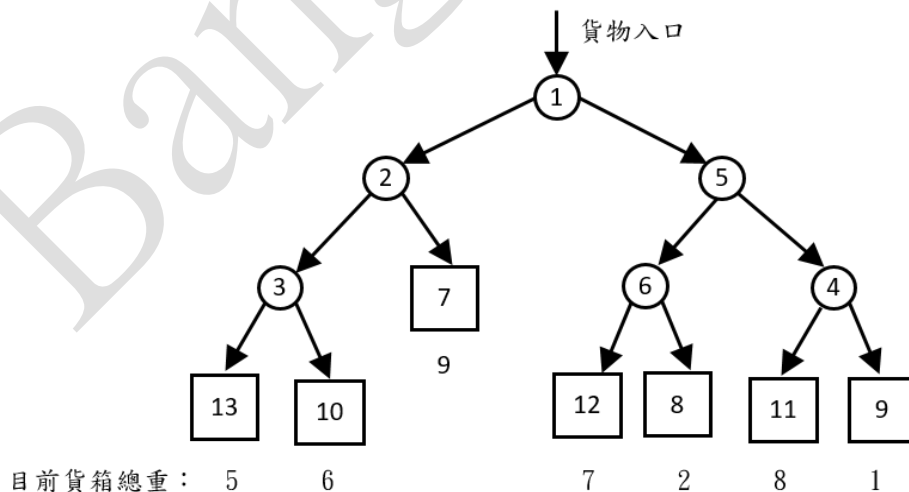
    parent[ch] = i;
  }
}
// find root, the node without parent
for (root=1; root<=n; root++)
  if (parent[root] == 0) break;
int total_h = 0;
for (int i=1; i<=n; i++)
  height[i]=-1; // not computed
for (i=1; i<=n; i++)
  total_h += findh(i);
printf("%d\n%d\n", root, total_h);
return 0;
}

```

就解題的目的，DFS 版本最簡單，一個問題我們提出了多個寫法，目的是讓讀者了解其中的道理。下一個問題也出自 APCS 的考試中。

P-8-5. 自動分裝 (APCS202002)

某工程公司設計了一個自動分裝的系統。貨物會一個一個的被送進此系統，經過一些切換器的轉送後，會被輸送到 n 個貨箱。系統有 $n-1$ 個切換器與 n 個貨箱。每個切換器有一個入口以及左右兩個出口，切換器的出口會接到其他切換器或是連接到貨箱，貨箱則只有入口。下圖是一個 $n=7$ 的例子，圓代表切換器，方形代表貨箱，請注意編號 $1 \sim n-1$ 的裝置是切換器，貨箱編號是 $n \sim 2n-1$ ，且貨物入口一定是 1 號切換器的入口。



每一個切換器會分別記錄左右兩個出口所通往貨箱的總重量，當貨物進入此切換器時，切換器會將貨物轉送到「貨箱總重量比較輕的那個出口」，如果兩邊一樣重，則送往左邊。以上圖的例子來說，假設每一個貨箱目前的重量如各矩形下方的標示，下一個到達的貨物的運送過程如下：一號切換器左邊出口是連到 {13, 10, 7} 三個

貨箱，目前總重 $5+6+9=20$ ；右邊出口連到的是 $\{12, 8, 11, 9\}$ 四個貨箱，目前總重 $7+2+8+1=18$ ，因此貨物會由右邊出口送到 5 號切換器。5 號切換器的左邊與右邊的貨箱總重是一樣的 ($7+2=8+1$)，因此貨物由左出口送至 6 號切換器。最後，6 號切換器將貨物送到 8 號貨箱 ($7>2$)。貨物進入貨箱後就存放在該貨箱，該貨物的重量就會加入該貨箱的總重，因此可能影響下一個貨物的運送路徑。

輸入此系統的連接架構與貨箱目前的重量，以及接下來依序進入的 m 個貨物的重量，請計算這 m 個貨物分別會被送到哪一個貨箱。

Time limit: 1 秒

輸入格式：第一行為兩個正整數 n 和 m ，其中 $n \leq 1e5$ 且 $m \leq 100$ 。第二行有 n 個非負整數，依序是編號為 $n \sim 2n-1$ 各個貨箱初始的重量。第三行是 m 個正整數，代表接下來依序進入的貨物重量。全部貨箱初始的重量與貨物重量總和不會超過 10^9 。第四行開始有 $n-1$ 行，這些是系統架構的資訊：每一行有三個整數 p, s 與 t ，代表裝置 p 的左右出口分別接到裝置 s 與 t ，其中 p 一定是一個切換器的編號 ($1 \leq p < n$)。同一行數字之間以空白間隔。

輸出：輸出一行有 m 個整數，依序代表接下來進入系統的 m 個貨物所進入到的貨箱編號，數字之間以一個空白間隔。

<p>範例一：輸入</p> <pre> 4 5 0 0 0 0 5 3 4 2 1 1 2 3 2 4 5 3 6 7 </pre>	<p>範例一：正確輸出</p> <pre> 4 6 7 5 5 </pre>
<p>範例二：輸入</p> <pre> 7 2 9 2 1 6 8 7 5 2 3 1 2 5 2 3 7 3 13 10 4 11 9 6 12 8 5 6 4 </pre>	<p>範例二：正確輸出</p> <pre> 8 7 </pre>

範例二的架構即是題目中的圖。

本題所描述的系統其實是一個所謂的二元樹(binary tree)的結構，每一個切換器是一個中間節點(internal node)，每一個貨箱是一個葉節點(leaf node)。題目的要求是模擬每一個貨物進入後的流程，要知道貨物在每個切換器會流向左方或右方，我們需要知道左邊與右邊貨箱**總重量**。假設知道每個切換器左右的總重量之後，我們就可以依照題意模擬貨物流向，而且根據流向來更新左右總重，所以在下一個貨物進入時，就可以繼續模擬的動作。

所以解這個題目需要會做 3 件事：(1). 儲存此樹狀結構；(2). 會計算初始每個中間節點左右的總重量；(3) 模擬流程更新重量紀錄。這一題是二元樹，只要存每點的左右孩子編號就可以了。對於一棵有根樹的某個中間節點，其下的所有節點稱為子樹(subtree)，因此我們需要計算的其實是以每個點為根的子樹重量。計算子樹重量的方法我們在 P-8-3 算 median 的時候已經看過了，這一題的中間節點本身是沒有重量，所以是計算子樹中的葉節點重量總和，但做法沒有不同。

我們來看看模擬的流程。對於每個中間點 v ，令 $w[v]$ 為其下貨箱總重， $LC[v]$ 與 $RC[v]$ 分別是他的左右子節點的編號。根據題目的定義，中間節點的編號皆小於 n ，因此我們的流程可以描述如下：

```
// 進入貨物的重量為 x，計算其最後到達的貨箱
v = 1; // 進入點(root)的編號
while (v < n) // 尚未到達貨箱
    if w[LC[v]] <= w[RC[v]] then
        v = LC[v]; // 左邊較輕往左流
    else
        v = RC[v]; // 往右流
    end if;
    w[v] = w[v] + x; // 更新總重
end while;
輸出 v; // 到達的貨箱編號
```

下面的範例程式中，第 5~9 行的副程式 `find_w()` 是以 DFS 寫法來計算 v 點以下葉節點的總重，因為是二元樹，所以遞迴呼叫左右孩子就可以，重點是要用表格紀錄。主程式在輸入資料後，第 24 行從 `root` 呼叫計算重量會把所有節點的重量全部算完，然後第 26 行開始模擬貨物進來的流程。

```

00 // P-8-5 arbitrary binary tree, DFS DP
01 #include <stdio.h>
02 #define N 200010
03 int n, w[N]={0}, lc[N], rc[N];
04 // recursively compute the weight
05 int find_w(int v) {
06     if (v>=n) return w[v]; // leaf
07     w[v] = find_w(lc[v]) + find_w(rc[v]);
08     return w[v];
09 }
10
11 int main() {
12     int m, i, in[110];
13     scanf("%d%d", &n, &m);
14     for (i=n; i<n*2; i++)
15         scanf("%d", &w[i]);
16     for (i=0; i<m; i++)
17         scanf("%d", &in[i]);
18     for (i=0; i<n-1; i++) {
19         int v;
20         scanf("%d", &v);
21         scanf("%d%d", &lc[v], &rc[v]);
22     }
23     // find initial weight of internal node
24     w[1] = find_w(1); // DFS from root=1
25     // start simulation
26     for (i=0; i<m; i++) {
27         int v=1;
28         while (v<n) { // until reaching leaf
29             if (w[lc[v]] <= w[rc[v]])
30                 v = lc[v]; // go left
31             else v = rc[v]; // go right
32             w[v] += in[i];
33         }
34         if (i>0) printf(" "); // output space
35         printf("%d", v);
36     }
37     printf("\n");
38     return 0;
39 }

```

下面一題是一個裸題，要算樹上所有點到所有點的距離。 $O(n^3)$ 當然不能接受， $O(n^2)$ 也不行，需要一個 $O(n)$ 的算法。這一題當做習題，給一個提示：請看看 P-8-3 題中計算 median 到所有點的距離和所使用的方法。

Q-8-6. 樹狀圖的距離總和

輸入一個樹狀圖，請計算所有點到所有點的距離總和。本題假設編號 1 是根節點。

Time limit: 1 秒

輸入格式：第一行是正整數 n ，代表點數，點以 $1 \sim n$ 編號，第二行有 $n-1$ 個正整數， $p(2), p(3), \dots, p(n)$ ，依序是編號 $2 \sim$ 編號 n 各點的 parent，第三行有 $n-1$ 個正整數，依序代表 i 從 2 到 n ，邊 $(i, p(i))$ 的長度。 n 不超過 $1e5$ ，每條邊長度是不超過 1000 的正整數。

輸出：各點到各點的距離總和。請注意 i 到 j 與 j 到 i 都要納入總和，如範例。

範例一輸入： 4 1 2 3 10 20 30	範例一輸出： 400
範例二輸入： 5 1 1 1 2 20 20 30 30	範例二輸出： 880

P-8-7. 寶石的顏色 (108 全國高中賽)

尋寶之旅的遊戲有一個地圖，地圖上有 n 個站，以 0 到 $n - 1$ 編號，此外有 $n - 1$ 條道路，這些道路都是單向的，遊戲固定從 0 號站出發，且已知從 0 號站出發可以直接或間接到達任何其他站。每一個站都有一顆寶石，寶石有分為多種顏色，第 i 站存放的寶石顏色為 $c(i)$ 。出發之前，你可以選定一種顏色的寶石收集箱，路途中遇到與你的收集箱相同顏色的寶石就可以收集(包含起點)，請你計算最多可以收集到多少顆同色寶石。

Time limit: 1 秒

輸入格式：第一行有是一個正整數 n ， $n \leq 2e5$ ，代表地圖上有 n 個站。第二行是 n 個非負整數，依序代表每一站的寶石顏色號碼 $c(0), c(1), \dots, c(n - 1)$ ；寶石的顏色號碼不超過 $1e9$ 。接著有 $n-1$ 行，每行兩個以空白間隔的整數 s 與 t ，表示有一條 s 到 t 的道路。

輸出：輸出為一整數，代表最多可能收集到的寶石數量。

範例一輸入： 6 0 0 0 0 0 0 0 1 1 2 0 3 1 4 1 5	範例一輸出： 3
範例二輸入： 10 5 3 3 1 4 0 3 4 5 0 0 1 5 2 7 3 5 4 0 5 4 6 1 7 0 8 4 9	範例二輸出： 2
(測資檔案中，第一筆測資只有一色，第二筆測資不超過 10 個顏色)	

樹的每個點上有一個顏色，要選擇一個顏色，並且找出一條由根走到某個葉子的路徑，經過所選顏色的點數越多越好。先想比較簡單的例子，如果只有一種顏色，那就是找一個最長的路徑，因為只能由根往下走，也就是找出根的高度。在 P-8-4 我們做過如何計算樹的高度。把計算高度的程式稍微修改一下，只有同色的點才計算，很容易就可以計算一條路徑上某個顏色的點有多少，如下面的做法。

```

// max num of nodes of color col in a path
int dfs(int v, int col) {
    int nc = 0;
    for (int u: child[v]) {
        nc = max(nc, dfs(u,col));
    }
    return nc + (c[v]==col);
}

```


在多種顏色的狀況下，一個簡單的做法是把每一種顏色都算一算，但是時間複雜度會是 $O(Cn)$ ，其中 C 是顏色種類數量。同樣的路走那麼多次實在不聰明，我們可以準備一張表格，紀錄目前的路徑上，各種顏色的數量。DFS 過程中，每次走到一點，就將該點顏色數量加一；當 DFS 返回 parent 時，我們可以把該點顏色數減一，以表示回復沒有走到他的狀態。這樣看起來表格很容易維護，但是要如何找最多顏色呢？

如果我們在表格上找最大值，又需要花 $O(C)$ 的時間，如果我們聰明一點只有在葉節點才計算最大值，可能可以減少一點時間，但是葉節點的數量可能會高達 $O(n)$ ，所以還是不好。其實很簡單，在每個節點時，我們不必去看所有的顏色，因為只有該點的顏色數量有變動，所以只要檢查該點的顏色數量有沒有超過最大值就可以了。

最後還有一個問題，顏色的編號並非由 0 或 1 連續編號，而是 10 億以內的非負整數，這表格要如何做呢？其實我們前面已經看過了，這就是要做離散化 (P-2-2)。有多種方式可以做離散化，先將他們轉換成 $0 \sim C-1$ 也可以，下面的範例程式我們使用一個 map 當做顏色編號的字典，我們並不先去置換顏色編號，而是要用到時再去查詢編號。

```
// rooted tree, find a path with max same color
#include<bits/stdc++.h>
using namespace std;
#define N 200010
int n, ans=0, c[N]; // color of node
vector<int> child[N];
map<int,int> D; // a dictionary for id of color
int counter[N]={0}; // counter of each color

void dfs(int v) {
    int cc=D[c[v]]; // find id of color
    counter[cc]++; // add color of v
    ans = max(ans,counter[cc]); // check answer
    for (int u: child[v]) {
        dfs(u);
    }
    counter[cc]--; // recover
    return;
}

int main() {
    int i;
    scanf("%d",&n);
    for (i=0; i<n; i++) scanf("%d",c+i);
    for (i=0; i<n-1; i++) {
        int v,p;
        scanf("%d%d",&p,&v);
        child[p].push_back(v);
    }
    // discretization
    for (i=0;i<n;i++) D.insert({c[i],0});
    int cnt=0; // id of color
```

```

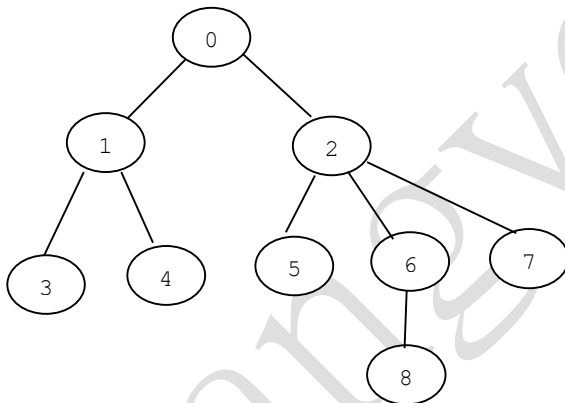
for (auto &v:D) {
    v.second=cnt;
    cnt++;
}
dfs(0); // dfs from root
printf("%d\n",ans);
return 0;
}

```

在圖論中，一群不相連的點稱為獨立集 (independent set)，在一般圖上找獨立集是件困難的事情，但是在樹上找獨立集並不困難。下面是一個相關的題目。

P-8-8. 樹的最大獨立集

輸入一棵有 n 個點的樹，我們要挑選一群彼此不相鄰的點，而且挑選的點越多點越好。請計算最多可以挑多少點。以下圖為例，我們可以挑 $\{0, 3, 4, 5, 6, 7\}$ 共 6 點，沒有辦法挑出更多的不相鄰的點。



Time limit: 1 秒

輸入格式：第一行是正整數 n ，代表點數，點以 $0 \sim n-1$ 編號， 0 是根，第二行有 $n-1$ 個正整數， $p(1), p(2), \dots, p(n-1)$ ，依序是編號 $1 \sim n-1$ 各點的 parent。 n 不超過 $1e5$ ，相鄰數字以空白間隔。

輸出：最多可以挑選幾點。

範例一輸入： 9 0 0 1 1 2 2 2 6	範例一輸出： 6
範例二輸入： 8 0 1 2 3 3 2 2	範例二輸出： 5

若 T 是我們要處理的樹，樹一定有葉，假設 v 是一個葉。我們宣稱：一定有一個最大獨立集挑選了 v 而沒有挑選 $\text{parent}(v)$ 。證明：假設某最大獨立集 S 沒有挑選 v ，那麼一定有挑 $\text{parent}(v)$ ，否則可以將 v 加入 S 中變成更大的獨立集。我們可以用 v 點取代 S 中的 $\text{parent}(v)$ 得到另外一個獨立集 S' ，就是我們宣稱包含 v 的最大獨立集。這樣的替換不可能造成不合法的解，因為 v 點是葉子， $\text{parent}(v)$ 是它唯一的鄰居。

根據以上的證明，我們可以得到一個貪心法：對於一棵樹，挑選所有的葉節點，捨棄這些葉節點的 parent ，重複此步驟直到樹被拔光為止。以題目中的樹為例，我們會先挑 $\{3, 4, 5, 8, 7\}$ ，捨棄 $\{1, 2, 6\}$ ；然後挑 0 ，結束。所得的獨立集是 $\{0, 3, 4, 5, 8, 7\}$ ，與題目中的解雖然不一樣，但都是最大獨立集。

實作時，可以一次拔掉全部的葉子，也可以一點一點拔，因為從葉子下手，所以我們需要一個由下而上的順序，當然也可以用 DFS 取代。以下是範例程式，程式的重點在於走到每一個點時，必須知道他的孩子中是否有人被挑選，我們用一個陣列 $\text{chosen}[]$ 紀錄每個點是否可以選，初始的時候全設為 1 ，當一個點被挑時，將它的 parent 的 chosen 設為 0 。這貪心法中我們沒有用表格紀錄而是用遞迴函數值接回傳結果，主程式中則只有讀取輸入與呼叫 $\text{dfs}(0)$ 而以。有一點要提醒， $\text{dfs}(r)$ 裡面會對 $\text{parent}[r]$ 做動作，但如果走到根節點時， $\text{parent}[r]$ 的存取有可能會造成傷害，所以我們 $\text{parent}[0]$ 設為不會出現的 n ，當然也有別種寫法。

```
// P-8-8 unweighted max independent set of a tree
// top-down implement
#include <bits/stdc++.h>
using namespace std;

#define N 100020
vector<int> child[N];
int parent[N], chosen[N]; // if chosen
// return max independent set, un-chosen its parent if chosen
int dfs(int r) {
    int num=0;
    for (int s:child[r]) {
```

```

        num += dfs(s);
    }
    if (chosen[r]) { // r is chosen
        chosen[parent[r]] = 0; // un-chosen its parent
        return num + 1;
    }
    return num; // r is not chosen
}

int main() {
    int i, n;
    scanf("%d",&n);
    for (i=1; i<n; i++) scanf("%d",&parent[i]);
    for (i=1; i<n; i++)
        child[parent[i]].push_back(i);
    for (i=0; i<n; i++) chosen[i]=1;
    parent[0] = n; // dummy
    printf("%d\n", dfs(0)); // root = 0
    return 0;
}

```

樹的貪心法用 bottom-up 其實比較自然，只是 bottom-up 的順序需要多一個 queue 來處理，以下是 bottom-up 的寫法。程式架構就是一個標準的由下而上的走訪，我們用了一個陣列 mark[] 來標記不可以挑選的點，當走到一點時，如果他沒有被標記，就挑選他並標記他的 parent。我們把 root 放在迴圈結束後處理，以免在迴圈內多寫 if。

```

// P-8-8 unweighted max independent set of a tree
// greedy, choosing leaf and mark its parent
// bottom-up implement
#include <bits/stdc++.h>
using namespace std;
#define N 100020
int parent[N], deg[N]={0};

int main() {
    int i, n, total=0;
    scanf("%d",&n);
    for (i=1;i<n;i++) {
        scanf("%d",&parent[i]);
        deg[parent[i]]++;
    }
    int mark[N]={0}; // cannot be chosen
    // bottom-up traversal
    queue<int> que;
    for (i=0;i<n;i++)
        if (deg[i]==0) que.push(i);
    i=0;
    while (! que.empty()) {
        int v=que.front();
        que.pop(); i++;
    }
}

```

```

    if (v==0) break;
    if (mark[v] == 0) {
        total++;
        mark[parent[v]] = 1;
    }
    if (--deg[parent[v]] == 0)
        que.push(parent[v]);
}
if (mark[0]==0) total++; // if root is chosen
printf("%d\n",total);
fprintf(stderr," visited nodes =%d\n",i);
return 0;
}

```

下面兩個問題都是與 P-8-8 類似的題目，我們當做習題。

Q-8-9. 服務中心選位置

有 n 個小村莊以 $1 \sim n$ 編號，這些村莊以 $n-1$ 條道路連接，每條道路都是雙向通行且連接兩個不同的村莊，已知從任兩個村莊之間都可以經由這些道路通達，兩個村莊稱為鄰近的村莊，如果它們之間有一條道路直接相連。現在要選一些村莊成立服務中心，因為預算的關係無法每個村莊都成立一個服務中心，政府的政策決定：如果某個村莊沒有設服務中心，那麼他一定有一個鄰近的村莊有服務中心。也就是說，任何一個村莊最多只要經過一條道路就可以到達一個有設服務中心的村莊。請你計算至少需要成立幾個服務中心。

Time limit: 1 秒

輸入格式：第一行是正整數 n ， $n > 1$ ，村莊以 $1 \sim n$ 編號。接下來有 $n-1$ 行是道路的資料，每一行有兩個正整數 u 與 v ，代表有一條道路連接 u 與 v 。 n 不超過 $1e5$ 。

輸出：最少需要成立的服務中心數量。

範例一輸入： 5 1 2 5 1 5 3 5 4	範例一輸出： 2
範例二輸入： 7 1 2 2 3 4 3 5 4 5 6 7 6	範例二輸出： 3

範例一說明：設在{5, 2}或{5, 1}皆符合要求。

範例二說明：這是 7 個點排成一條路徑，選{2, 5, 7}是一個解，兩個點一定無法滿足需求。

Q-8-9 提示：我們說一個點可以管轄自己與它的鄰居，一個圖上選一些點可以管轄所有的點，這樣的集合稱為管轄集 (dominating set)，本題要計算樹的最小管轄集。類似於 P-8-8，我們可以想一想：如果 v 是一個葉節點，是否一定有最佳解選了或沒選 v ？當決定選擇某點時，要將它鄰近的點做一些設定。

Q-8-9 是比較容易寫壞的題目，下面這一題比較容易。

Q-8-10. 竊聽間諜網

有 n 個間諜，他們的 ID 以 $0 \sim n-1$ 編號，間諜頭子的 ID 是 0，除了他以外每個間諜都有一個領導，每個間諜只會與他的領導相互通話，現在我們要竊聽間諜們彼此的通話，所以要在一些間諜的通話裝置上安裝竊聽器，要竊聽到某個間諜與他領導的通話，必須至少在他們其中一人的通話裝置上安裝竊聽器。如果不想漏掉任何通話，至少要安裝幾台竊聽器。

Time limit: 1 秒

輸入格式：第一行是正整數 n ，代表間諜數，每個間諜的 ID 必定大於他領導的 ID。第二行有 $n-1$ 個整數分別是 $t(1), t(2), \dots, t(n-1)$ ，其中 $t(i)$ 就是 i 的領導。 n 不超過 $1e5$ 。

輸出：最少的竊聽器數量。

<p>範例一輸入：</p> <p>5 0 1 1 3</p>	<p>範例一輸出：</p> <p>2</p>
<p>範例二輸入：</p> <p>7 0 1 1 1 1 1</p>	<p>範例二輸出：</p> <p>1</p>

範例一說明：安裝在 1 號與 3 號間諜。

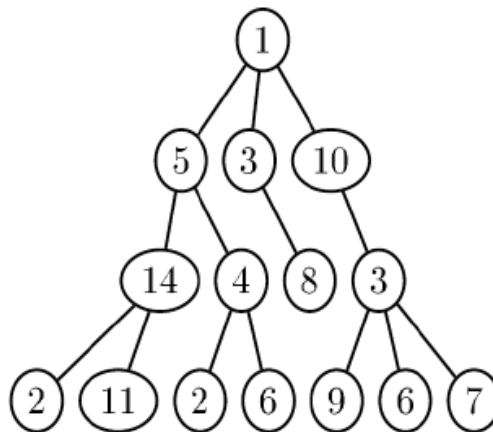
範例二說明：安裝在 1 號間諜。

我們說一個點可以 cover 它所接的邊，在一個圖中選一些點 cover 所有的邊，這樣的點集合稱為 vertex cover，也就是每一個邊的兩端點都至少被選中一點，本題是樹的無權重的 minimum vertex cover，與前兩題類似，所以不予提示。

P-8-11. 公司派對 (NCPC)

有一個公司有 n 個成員，成員以 $1 \sim n$ 編號，每個人都有一個領導，除了編號 1 是公司的總裁，他沒有領導。現在要辦一個公司派對，想要邀請一些人來參加，如果邀請了編號 i 的人來參加，就會有 $r(i)$ 的歡樂指數，可是，這個公司的任何人都不能跟他的領導一起參加派對，請你幫忙計算要邀請那些人來參加，才可以讓參加者的歡樂指數的總和最大。

右圖是一個例子，每個圓圈代表一個人，圓圈內的數字是他的歡樂指數，每個人上方與他連線的就是他的領導，你可以看到，總裁的歡樂



指數是 1，也全公司最低的。我們可以看到此例中有兩個人的歡樂指數是 14 與 11，可惜 14 是 11 的領導，所以不能同時邀請這兩人參加。這個例子的最大歡樂指數總和是 66，扣除歡樂指數分別為 1, 3, 3, 4, 14 那 5 個人，其餘的人都參加。

Time limit: 1 秒

輸入格式：第一行是兩個正整數 n 與 $r(1)$ ，接下來有 $n-1$ 行，每行兩個正整數，由 $i=2$ 到 n ，依序是 $p(i)$ 與 $r(i)$ 。 n 不超過 $1e5$ ， $r(i)$ 不超過 1000 的正整數。

輸出：輸出參加派對的最大歡樂指數總和。

範例一輸入： 4 7 1 5 2 6 3 8	範例一輸出： 15
範例二輸入： 15 1 1 5 1 3 1 10 2 14 2 4 3 8 4 3 5 2 5 11 6 2 6 6 8 9 8 6 8 7	範例二輸出： 66

範例一說明：選編號 1 與 4，歡樂指數為 $7+8=15$ 。

範例二說明：題目敘述中的例子。

這一題其實是 P-8-8 的加權版本，是最大權重獨立集的問題。無加權時要挑選最多不相鄰的點，加權時要挑選總和最大的不相鄰的點。無加權時，貪心法則可以找到最多的點，但是在有加權時，貪心的方法並無法找到最佳解，我們嘗試用 DP 的觀點來找答案。對任一點 v ，令 $f(v)$ 代表以 v 為根的子樹的最佳解，也就是在 v 的後代(含 v)中，不相鄰的點最大可能的權重和，但在寫狀態轉移式時候遇到困難，因為 v 點是否有選會影

響他的孩子是否可以挑選。碰到類似狀況時，把解分類是通常的解決方法，我們把解分成兩類：

$f_0(v)$ 是以 v 為根的子樹中，不挑選 v 的狀態下之最佳解；

$f_1(v)$ 是以 v 為根的子樹中，挑選 v 的狀態下之最佳解。

我們可以得到以下遞迴式：

$f_0(v) = \sum(\max(f_0(u), f_1(u)) : \text{for all children } u \text{ of } v);$

$f_1(v) = r(v) + \sum(f_0(u) : \text{for all children } u \text{ of } v);$

也就是說，當 v 不選的時候，他的孩子可以挑也可以不挑；而當 v 被挑選時，他的孩子都不可以挑。根據這個遞迴式，搭配由下而上的 DP 順序，就可以很容易求解，最後的答案在 $\max(f_0(r), f_1(r))$ ，其中 r 是根。實作時一樣有 bottom-up 與 top-down 兩種寫法，以下先揭示非遞迴的寫法，走訪的方式前面已經看過很多次了，就不再多解釋說明，時間複雜度是 $O(n)$ 。

```
// Tree, Max weight independent set, bottom-up DP
#include <bits/stdc++.h>
using namespace std;
#define N 100010
// f0[], f1[]: opt without/with root
// f0[r]=sum(max(f0[v],f1[v])), f1[r]=w[r]+sum(f0[v])
int main() {
    int parent[N],n, f0[N]={0}, f1[N], deg[N]={0};
    scanf("%d%d",&n,&f1[1]);
    for (int i=2;i<=n;i++) {
        scanf("%d%d",&parent[i],&f1[i]); // weight[i] in f1[i]
        deg[parent[i]]++;
    }
    queue<int> Q;
    for (int i=1; i<=n; i++)
        if (deg[i]==0) Q.push(i);
    while (!Q.empty()) {
        int v=Q.front();
        Q.pop();
        if (v==1) break;
        int r=parent[v];
        f0[r] += max(f0[v],f1[v]);
        f1[r] += f0[v];
        if (--deg[r] == 0) Q.push(r);
    }
    printf("%d\n",max(f0[1],f1[1]));
    return 0;
}
```

使用 top-down 的 DFS 時，輸入時先將各點的 child 設定好，這題有給 root，所以不必特別去找。程式的架構之前也看過多次，也不多做解釋了。

```
// Tree, Max weight independent set, top-down DP
#include <bits/stdc++.h>
using namespace std;
#define N 100010
int f0[N],f1[N],w[N];
vector<int> child[N];

void dfs(int r) {
    f0[r]=0;
    f1[r]=w[r];
    for (int c:child[r]) {
        dfs(c);
        f0[r]+=max(f0[c],f1[c]);
        f1[r]+=f0[c];
    }
    return;
}

int main() {
    int parent,n;
    scanf("%d%d",&n,&w[1]);
    for (int i=1;i<=n;i++) child[i].clear();
    for (int i=2;i<=n;i++) {
        scanf("%d%d",&parent,&w[i]);
        child[parent].push_back(i);
    }
    dfs(1);
    printf("%d\n",max(f0[1],f1[1]));
    return 0;
}
```

回顧一下在第六章的 P-6-2 (不連續的表演酬勞)，其實上那就是一個一維陣列的最大獨立集問題，而一維陣列是樹的一種特殊形式，所以也可以仿效 P-6-2 的解法來想這一題的 DP。如果用一維陣列時的解法，還有一種寫法是只設一個遞迴函數 $f(v)$ 是 v 子樹的最佳解， v 不挑時，選各子樹的解相加；而 v 挑選時，將 v 的孫子 (孩子的孩子) 為根的子樹的解拿來相加。因為一個點只有一個祖父，這樣做的複雜度還是 $O(n)$ ，所附範例程式中有一支是用這個方法寫的，請自行參考。

下一題在一維陣列中也有類似的題目，當做習題，提示是：請參考 P-4-13。

Q-8-12. 佔領連續的城鎮

某個遊戲中有 n 個城鎮以 $1 \sim n$ 編號，這些城鎮以 $n-1$ 條道路連接，每條道路都是雙向通行且連接兩個不同的城鎮，已知從任兩個城鎮之間都可以經由這些道路通達。現

在你要佔領一些城鎮來獲取最大的利益，佔領編號 i 的城鎮可以獲得 $w[i]$ 的利益，但是你佔領的城鎮必須內部可以互相通達，也就是說，若 i 與 j 都是你的領地，則從 i 出發可以只經過你自己的城鎮到達 j 。請計算你最多可以獲得多少總和利益。

Time limit: 1 秒

輸入格式：第一行是正整數 n ， $n > 1$ ，村莊以 $1 \sim n$ 編號。第二行有 n 個整數，依序是 $w[1], w[2], \dots, w[n]$ 。接下來有 $n-1$ 行是道路的資料，每一行有兩個正整數 u 與 v ，代表有一條道路連接 u 與 v 。 n 不超過 $1e5$ ，每個城鎮利益的絕對值不超過 $1e4$ 。

輸出：最多可以獲得的總和利益。

<p>範例一輸入：</p> <pre>5 -2 3 -1 3 -4 1 2 5 1 5 3 5 4</pre>	<p>範例一輸出：</p> <pre>3</pre>
<p>範例二輸入：</p> <pre>7 2 -1 4 -3 5 -3 2 1 2 2 3 4 3 5 4 5 6 7 6</pre>	<p>範例二輸出：</p> <pre>7</pre>

範例一說明：選 $\{2\}$ 或 $\{4\}$ 都是 3。

範例二說明：這是 7 個點排成一條路徑，選 $\{1, 2, 3, 4, 5\}$ ，總和是 7。

下面一題看起來也是類似的題目，他是 Q-8-9 的加權版本，但這一題比較困難。

P-8-13. 不同成本的購物中心 (@@)

有 n 個小村鎮以 $1 \sim n$ 編號，這些村鎮以 $n-1$ 條道路連接，每條道路都是雙向通行且連接兩個不同的村鎮，已知從任兩個村莊之間都可以經由這些道路通達，兩個村莊稱

為鄰近的村莊，如果它們之間有一條道路直接相連。現在要選一些村莊成立服務中心，因為預算的關係無法每個村莊都成立一個服務中心，政府的政策決定：如果某個村莊沒有設服務中心，那麼他一定有一個鄰近的村莊有服務中心。也就是說，任何一個村莊最多只要經過一條道路就可以到達一個有設服務中心的村莊。每個村莊設立服務中心的成本不同，在編號 i 的村莊成立服務中心需要的成本是 $w[i]$ ，我們希望以最少的總成本設置服務中心，請你計算最少的總成本。

Time limit: 1 秒

輸入格式：第一行是正整數 n ， $n > 1$ ，村莊以 $1 \sim n$ 編號。第二行有 n 個正整數，依序是 $w[1], w[2], \dots, w[n]$ 。接下來有 $n-1$ 行是道路的資料，每一行有兩個正整數 u 與 v ，代表有一條道路連接 u 與 v 。 n 不超過 $1e5$ ，每個村莊設置成本不超過 1000 。

輸出：最少需要成立的服務中心數量。

<p>範例一輸入：</p> <pre>5 2 4 1 3 7 1 2 5 1 5 3 5 4</pre>	<p>範例一輸出：</p> <pre>6</pre>
<p>範例二輸入：</p> <pre>7 1 3 3 1 2 2 1 1 2 2 3 4 3 5 4 5 6 7 6</pre>	<p>範例二輸出：</p> <pre>3</pre>

範例一說明：設在 $\{1, 3, 4\}$ 成本 6。

範例二說明：這是 7 個點排成一條路徑，每個點成本皆相同。選 $\{1, 4, 7\}$ 成本是 3。

這一題是要計算 minimum weight domination set，與最大獨立集類似的：無加權時可以貪心，加權時必須用 DP 來解。最大獨立集的 DP 遞迴式比較容易，這一

題的困難點並不是程式難寫或是需要什麼特別的資料結構與演算法，而是 DP 遞迴式很容易弄錯。子問題都定義在以任一點 r 為根的子樹的解，我們將子問題分成三類：

- $d1(r)$ ：在 r 被挑選下的最佳解 (domination set 的最小權重)。
- $d01(r)$ ：在 r 不被挑選但 r 已被管轄下的最佳解，也就是至少有一個 r 的孩子被挑選。
- $d00(r)$ ：在 r 不被挑選且 r 不一定有沒有被管轄之下的最佳解。這個值未必是個合法的解，根 r 要靠上面的點管轄。

子問題的遞迴式如下 (v 皆為對所有 r 的孩子)：

- $d1(r) = w(r) + \sum(\min(d00(v), d1(v)) : \text{all } v \text{ in child}(r))$ 。也就是對所有子樹的最小值加總後加上 r 的權重，因為 $d00(v) \leq d01(v)$ ，所以只需要取 $\min(d00(v), d1(v))$ 。
- $d00(r) = \sum(\min(d1(v), d01(v)) : \text{all } v)$
- $d01(r) = \sum(\min(d1(v), d01(v)) : \text{all } v) + \min_diff$ ；其中 \min_diff 代表 $\max(0, \min(d1(v) - d01(v) : \text{all } v))$ ，也就是對所有孩子 v 找出 $d1(v) - d01(v)$ 的最小值，這個值是正的就要加上，如果是負的就不加。遞迴式的意義在各孩子的最小合法解之總和，但是不能全部的孩子都沒有挑選。

程式並不特別難，一樣有 top-down 與 bottom-up 兩種寫法，以下是 top-down 的寫法，註解快要比程式碼還多了，另外一種寫法類似，請自行參考。

```
// P-8-13 min domination set of a tree
// top-down DP d1(r): min cost with root,
// d01(r): without root but root is dominated
// d00(r): without root and root is not necessarily dominated
// d1(r)=w(r)+sum(min(d00(v),d1(v)): all child v of r), d00<=d01
// d00(r)=sum(min(d1(v),d01(v)))
// d01(r)=sum(min(d1(v),d01(v)))+ max(0,min(d1[v]-d01(v)))
// not all d01
#include <bits/stdc++.h>
using namespace std;

#define N 100020
#define oo 1000000000
vector<int> adj[N];
int w[N], d1[N], d01[N], d00[N];
// set min domination set of subtree(r) with parent p
void dfs(int r, int p) {
    d1[r] = w[r], d01[r]=oo, d00[r]=0;
    for (int v: adj[r]) {
        if (v==p) continue;
```

```

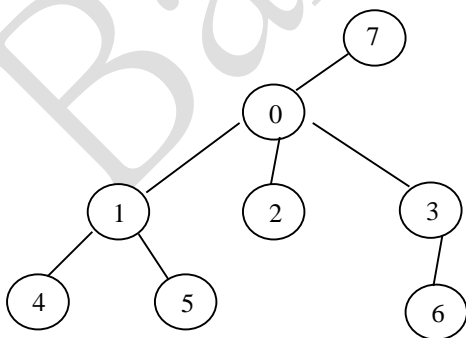
    dfs(v,r);
    d1[r] += min(d00[v], d1[v]);
    d00[r] += min(d1[v], d01[v]);
    d01[r] = min(d01[r], d1[v]-d01[v]);
}
d01[r] = d00[r] + max(0, d01[r]);
return;
}

int main() {
    int i, n;
    scanf("%d",&n);
    for (i=1; i<=n; i++) scanf("%d",&w[i]);
    for (i=1; i<n; i++) {
        int u,v;
        if (scanf("%d%d",&u, &v)!=2)
            fprintf(stderr,"input error\n");
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    // parent of root(1) is set to 0(dummy);
    dfs(1,0);
    printf("%d\n", min(d1[1], d01[1]));
    return 0;
}

```

P-8-14. 血緣關係 (APCS201603)

小宇有一個大家族。有一天，他發現記錄整個家族成員和成員間血緣關係的家族族譜。小宇對於最遠的血緣關係（我們稱之為"血緣距離"）有多遠感到很好奇。



上圖為家族的關係圖。0 是 7 的孩子，1、2 和 3 是 0 的孩子，4 和 5 是 1 的孩子，6 是 3 的孩子。我們可以輕易的發現最遠的親戚關係為 4 (或 5) 和 6，他們的"血緣距離"是 4 (4~1, 1~0, 0~3, 3~6)。我們將給你小宇家族的關係圖，請幫小宇找

出最遠的"血緣距離"。你可以假設只有一個人是整個家族成員的祖先，而且沒有兩個成員有同樣的小孩。

Time limit: 1 秒

輸入格式：第一行為一個正整數 n 代表成員的個數，每人以 $0 \sim n-1$ 之間惟一的編號代表。接著的 $n-1$ 行，每行有兩個以一個空白隔開的整數 a 與 b ($0 \leq a, b \leq n-1$)，代表 b 是 a 的孩子。 n 不超過 $1e5$ 。

輸出：輸出最遠"血緣距離"的答案。

範例一輸入： 8 0 1 0 2 0 3 7 0 1 4 1 5 3 6	範例一輸出： 4
範例二輸入： 4 0 1 0 2 2 3	範例二輸出： 3

範例一是題目中敘述的例子。

在一個圖中，最長的兩點最短路徑(距離)稱為直徑，直徑這個詞同時代表該路徑，也代表該路徑的長度。這個定義有點繞口，其實就像在平面幾何的圓中，直徑是圓上最遠的兩點的距離。如果對任意兩點 u, v 都求出兩點之間的最短距離 $d(u, v)$ ，那麼 $\text{diameter} = \max(d(u, v) : \text{all } u, v \text{ in } V)$ 。我們在前面的題目中已經知道，樹上求一點與其它點的距離可以在 $O(n)$ 時間內完成，那麼直接依照定義求出所有點到所有點的距離然後計算直徑可以在 $O(n^2)$ 時間完成。我們需要一個更有效率的解法。

這一題是無加權樹的直徑，算是樹的簡單 DP 中的經典題。先看用標準 top-down DP 的寫法。因為直徑必然由某個葉節點往上走到某個點 r 之後，再從 r 的另外一個孩子一路往下走，退化的狀態是 r 只有一個孩子，那就只走到 r 為止，我們稱這樣的路徑為以 r 為最高點的路徑。也可以換句話說，直徑一定是以某個 r 為最高點的最長路徑。我們

只要對每個 r ，計算出以 r 為最高點的最長路徑，也就是 r 往下的兩條最長路徑，然後對所有的 r 取最大值就可以了，但是要小心的是，這兩條路徑必須經由兩個不同的孩子。我們前面 P-8-4 學過樹中點的高度，事實上我們要找的就是 r 的兩個高度最高的孩子，把他們的高度相加再加 2 就是以 r 為最高點的最長路徑。

以 top-down 寫法的主架構當然還是 DFS，在對所有孩子遞迴呼叫後，我們的任務是找出深度最高的兩個孩子，這就像在一個數列中找出最大的兩個值一樣。如果用直接的寫法，通常需要用兩次迴圈，第一次找出最大值，第二次找出非最大值中的最大值；或者掃描一次但同時維護好第一名與第二名。這兩種寫法的時間複雜度是 $O(n)$ ，因為對每一個點 r ，它需要的時間複雜度是 $O(|\text{child}(r)|)$ ，而所有點的孩子數量總和不過是 $O(n)$ 。但不管如何，這兩種寫法都有點囉嗦。想想下面這招：假設數列是 S ， pm 是 prefix_max ，則最大兩個元素之和其實就是 $\max(S[i]+\text{pm}(i): \text{for all } i)$ 。

所以我們只需要一路掃描並維護好目前看到的最大值，每次將當下看到的值加上他之前的最大值相加即可。以下是範例程式，非常的簡潔漂亮。請留意，在 prefix_max 預設為 0 下，這個方法也適用只有一個孩子的情形。

```
// unweighted tree diameter, top-down DP
#include <bits/stdc++.h>
using namespace std;
#define N 100010
vector<int> child[N]; // list of children
int hei[N]={0}, diameter=0;
// find longest path whose top-vertex is r
void dfs(int r) {
    for (int v: child[r]) {
        dfs(v);
        diameter = max(diameter, hei[v]+1+hei[r]);
        // update parent's height
        hei[r] = max(hei[r], hei[v]+1);
    }
}

int main() {
    int i,j,k,n;
    bool not_root[N]={false};
    scanf("%d", &n);
    for (i=0; i<n-1; i++) {
        scanf("%d%d", &j, &k);
        child[j].push_back(k);
        not_root[k] = true;
    }
    // find root
    int root=0;
    while (not_root[root]) root++;
    // dfs to find diameter
```



```

dfs(root);
printf("%d\n", diameter);
return 0;
}

```

上面的 DP 當然也可以寫成非遞迴的 bottom-up 形式，這已經出現很多次，就不再揭示這樣的寫法。樹的直徑有另外一個演算法，但必須根據圖論的一個性質：「距離樹上任一點最遠的點，一定是某個直徑的端點」。這個性質的證明並不算難，但我們只簡要說明，有興趣的讀者自己去證明。若 v 是任一點， u 是離 v 最遠的一點，而 D 是任一條直徑，則 v 到 u 的路徑與 D 必然相交；再則， u 到 D 的兩端較遠者，必然也是直徑。

根據上述的性質，可以得到一個很簡單的線性時間演算法，我們稱為 farthest of farthest 演算法：挑選任一點 v ，找離它最遠的一點 u ，再找離 u 最遠的一點 s ，則 u 到 s 就是直徑。以下是這樣的範例程式，我們以無根樹的方式用 BFS 來找距離最遠的點。因為要呼叫兩次，把 BFS 寫成函數，每次呼叫會算出起點到所有點的距離放在 $d[]$ 中，根據 $d[]$ 可以找出最遠的點與最遠的距離。

```

// unweighted tree diameter, farthest of farthest algorithm
// using bfs
#include <bits/stdc++.h>
using namespace std;
#define N 100010
vector<int> adj[N]; // adjacency list
int n, d[N];
// find all distances d[v] from r
void bfs(int r) {
    for (int i=0; i<n; i++) d[i]=-1; // unvisited
    queue<int> Q;
    Q.push(r);
    d[r] = 0;
    while (!Q.empty()) {
        int v=Q.front();
        Q.pop();
        for (int u: adj[v]) {
            if (d[u] >= 0) continue; // visited
            d[u] = d[v] + 1;
            Q.push(u);
        }
    }
    return;
}

int main() {
    scanf("%d", &n);
    for (int i=0; i<n-1; i++) {
        int j, k;
        scanf("%d%d", &j, &k);
        adj[j].push_back(k);
    }
}

```

```

    adj[k].push_back(j);
}
bfs(rand()%n); // random root
int farthest = max_element(d, d+n) - d;
bfs(farthest);
printf("%d\n", *max_element(d, d+n));
return 0;
}

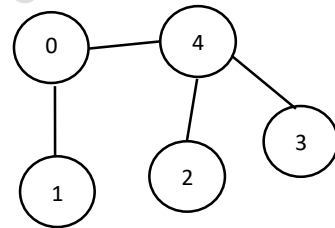
```

下面一題是與直徑有關的習題，提示是：想想 P-8-1 的解答說明。此外，這一題是加權版，也就是邊是有不同長度的。

Q-8-15. 樹上一位不回家的推銷員

有一個推銷員要走訪 n 個城市。這些城市以 $n-1$ 條道路連接，每條道路連接兩個不同的城市並且雙向可以通行，而且已知每一個城市都可以到達，不會有到達不了的狀況。輸入道路的資料，請你幫推銷員找出一個最短的路徑走訪所有的城市，推銷員可以從任何城市開始，也不必回到開始的城市，只要每個城市至少到一次就可以。

舉例來說，如右圖，有 5 個城市以 4 條道路連接，假設每條道路的長度都是 1。最短的拜訪路徑是 (1, 0, 4, 2, 4, 3)，長度為 5。



Time limit: 1 秒

輸入格式：第一行是正整數 n ，代表城市的數量，城市是以 $0 \sim n-1$ 編號，接下來有 $n-1$ 行是道路的資料，每行三個整數 a, b 與 w ，代表此道路連接城市 a 與 b ，道路的長度是 w 。 n 不超過 50000，每條道路長度是不超過 100 的正整數。

輸出：輸出最短的旅行距離。

範例一輸入： 5 1 0 1 0 4 1 2 4 1 4 3 1	範例一輸出： 5
範例二輸入： 3 1 2 5 0 2 3	範例二輸出： 8

Q-8-16. 病毒演化 (APCS202007)

在資訊科學上，RNA 病毒可以看成一個由 A、U、G、C 這四種字母構成的字串。病毒演化中，某些字元可能變異成其他字元，例如將 AAUGG 中的第 3 個位置的 U 換成 C，則得到 AACGG，變異可能發生在多個位置。有個實驗團隊取得了某種 RNA 病毒的數個 RNA 片段樣本，並且掌握到這些樣本演化的親緣關係，用

(樣本編號, 親代編號, RNA 字串)

的格式記錄，其中「樣本編號」與「親代編號」為兩個整數，若兩數字相同，則該樣本即為演化的源頭。例如，

(1, 1, AAAA)

(2, 1, GCAA)

表示樣本 1 的 RNA 字串為 AAAA，樣本 2 的為 GCAA，且樣本 2 是由樣本 1 在兩個位置發生變異演化而來，樣本 1 為演化的源頭。

然而，字串中每個位置的字元無法確定，實驗團隊以 @ 來表示這些字元，換言之，@ 可能為 A、U、G、C 中任何一個。請注意，一個字串中可能有多個 @，並非代表這些位置的字元是相同的。例如 A@C@ 可能是任何第一個字元為 A 且第三個字元為 C 的字串，像是 ACCU 或 AACC。團隊猜測演化過程發生的變異總數會盡可能的少。請你利用親緣關係，來計算最小的變異總數量。

Time limit: 1 秒

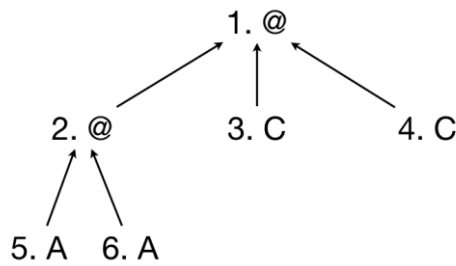
輸入格式：第一行有兩個正整數 n 與 m ，表示共有 n 個樣本，由 1 至 n 編號；每個樣本之 RNA 字串長度均為 m ，其中 $n \leq 1000$ 且 $m \leq 80$ 。接下來 n 行，每行包含以空白間隔的兩個正整數 i 與 j 以及一個 RNA 字串 s_i ，對應一個樣本 (i, j, s_i) ， s_i 由 A、U、G、C 與 @ 五種字元所組成。若 $i=j=1$ ，則該樣本即為演化的源頭，源頭以外的樣本皆恰有一個親代。

輸出：輸出最小可能的變異總數。

<p>範例一輸入：</p> <pre>2 3 1 1 AAC 2 1 A@@</pre>	<p>範例一輸出：</p> <p>0</p>
<p>範例二輸入：</p> <pre>6 1 1 1 @ 2 1 @ 3 1 C 4 1 C 5 2 A 6 2 A</pre>	<p>範例二輸出：</p> <p>1</p>

範例一說明：樣本 2 為 AAC 時，變異量為 0。

範例二說明：我們以一樹狀結構表示此例的親緣關係：



將樣本 1 置換成 C、樣本 2 置換成 A，則只有在樣本 1 與 2 之間有一個變異，其它皆無變異，故答案是 1。

Q-8-16 提示：從範例二看到的第一件事是這些 RNA 是個樹狀結構，另外一個很重要的事情是：雖然每個點是個字串，但是字串的每個位置是獨立的，也就是所有字串的第 1 個字元，要算一個最小變異量；所有字串的第 2 個字元，要算一個最小變異量；對每一個字串位置 i 都要算一個最小變異量，最後的答案是把他們加起來。所以，我們只要會解字串長度 $m=1$ 的狀況，最後套一個迴圈就好了。

如果沒有 '@'，所有的變異都是確定的，只要把每個點與它的 parent 比較是否相等就好了，所以問題在如何決定 '@' 變異量。如果葉節點是 '@'，我們可以讓它跟它的 parent 相同，這顯然不會有變異量，也就是說 '@' 的葉子根本可以無視它。為了計算方便，我們先將 AUCG 轉換為數字 0~3，以 DP 的觀點，對於每個節點 v ，我們算四個成本 $cost[v][i]$ ， i 從 0 到 3，分別代表如果 parent 的字元是 i 的時候 v 點以下子樹的總成本，然後去建構出遞迴式。

Bangye.Wu