建國中學 2012 年資訊能力競賽培訓講義

bobogei
Yuhsi Chaing
skg
STEP4
momohuang

December 20, 2012

目錄

1 簡介

1.1 作戰計劃

1.1.1 主線任務

時間	比賽	戰場	說明
9/18,10/1	建中校內資訊能力競賽	建國中學	前 12 名進入校隊，培訓開放旁聽
11 月	北市資訊能力競賽	師大分部	校隊參加，前十名進全國賽
12 月	全國資訊能力競賽	清華大學	前十名進入 TOI 一階
3 月	建中 TOI 校內補選	建國中學	遞補已進 TOI 一階的名額
3 月	TOI 入營考	師大分部	前 20 名進入 TOI 一階
4 月	TOI 一階	師大分部	2 次模考前 12 名進 TOI 二階
5 月	TOI 二階	師大分部	4 人成為 IOI 國手
5 月	APIO	師大分部	由 TOI 二階的選手參加
7/6~7/13	IOI	澳洲	為國爭光!!

1.1.2 支線任務

時間	比賽	戰場	說明
11 月	北市軟體競賽	松山工農	第 1 階段筆試第 2 階段上機
11~12 月	NPSC	台灣大學	同校三人組隊報名，分初賽和決賽，每校至多三隊進決賽，獎品豐富!!

1.1.3 競賽資源簡介

Online Judge

1. USACO Training (http://train.usaco.org)
2. Uva Online Judge (ACM) (http://uva.onlinejudge.org)
3. Temporary INFOR Online Judge (http://tioj.redirectme.net:8080/JudgeOnline)
4. Młodzieżowa Akademia Informatyczna (POI) (http://main.edu.pl/en)
5. PKU Online Judge (http://acm.pku.edn.cn/JudgeOnline)
6. STEP5 Online Judge (http://web2.ck.tp.edu.tw/~step5/)

線上賽

1. USACO (http://www.usaco.org)
2. Top Coder (http://www.topcoder.com)
3. CodeForces (http://www.codeforces.com)

網路資源

1. 校內培訓網站 (http://www.ck.tp.edu.tw/∼peng)
2. DJWS 演算法筆記 (http://www.csie.ntnu.edu.tw/∼u91029)
3. NPSC 補完計劃 (http://www3.tcgs.tc.edu.tw/npsc/index.php)

1.2 演算法概論

1.2.1 演算法

什麼是演算法

演算法 (algorithm)，簡單來說就是我們寫程式時撰寫程序的方法，當我們給程式一筆輸入 (input)，要讓程式能夠輸出我們想要的結果 (output)，靠的就是演算法。演算法的優劣會大大影響程式的執行效率，尤其在競賽中，越大量的資料會讓演算法之間的效率差距越明顯，如何設計好的演算法便成了重要的課題。

演算法的比較

為什麼我們要學演算法? 怎樣才算是一個好的演算法?

一個好的演算法，他的答案至少要保證是正確的 (廢話)，而除了答案正確以外，還要有好的效率 (efficiency)。效率包含了程式的執行時間 (runing-time) 和需要使用的記憶體空間 (memory space)。通常我們會先確保答案的正確性，再視題目對時間和空間的限制，使用最節省資源的演算法。

舉個例子，現在若我們要對 N 個數字做排序，最直接的想法就是把這 N 個數字的全排列都試一次，總共要做 $N!$ 次。而若是運用一點演算法，可以只用 N^2 次，甚至只要 $N \lg N$ 次 ($\lg N = \log_2 N$)。假設我們現在有一台電腦每秒可以處理 1 億 (10^8) 個指令，並用他來分別執行上述的三種排序法，當 N 的值很大如 $N = 10^6$ 時，這三種排序法的執行時間如下：

- 排序法 A：$10^6! \div 10^8 =$ 跑到天荒地老，把電腦砸了比較快 $= =$
- 排序法 B：$(10^6)^2 \div 10^8 = 10^4$(秒)
- 排序法 C：$(10^6 \times \lg 10^6) \div 10^8 \approx 0.5$(秒)

由此可見，選擇一個正確的演算法來解決問題是非常重要的。

時間複雜度

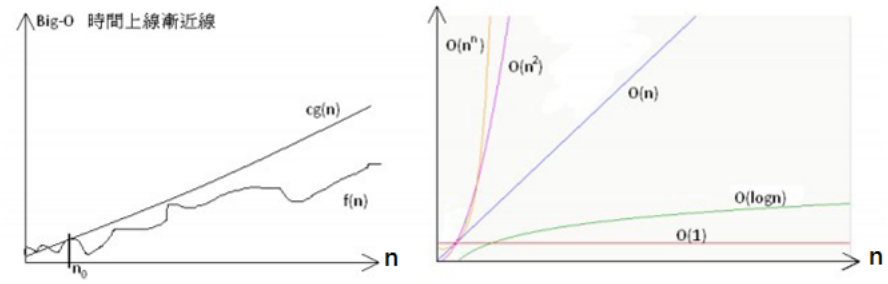
我們通常用 Big-O 記號來表示一個演算法的上限時間複雜度，Big-O 的嚴格定義如下：

$$f(N) = O(g(N)), \quad \text{if} \quad \exists N_0, c > 0, \quad \forall N > N_0, \quad |f(N)| \leq c|g(N)|$$

也就是說存在一個固定的常數 c，對於夠大的 N，$f(N)$ 都不大於 c 倍的 $g(N)$。

常見的時間複雜度：

$$O(1) < O(\lg N) < O(N) < O(N \lg N) < O(N^k) < O(k^N) < O(N!) < O(N^N)$$



1.2.2 排序與二分搜

排序

排序顧名思義就是把資料按照順序排好，以便利用資料的單調性來做處理，以下介紹幾種常見的排序法。

Insertion Sort(插入排序)

時間複雜度：$O(N^2)$

Algorithm 1 Insertion Sort

```
1:  procedure INSERTION_SORT(A[], N)
2:      for i ← 1 to N − 1 do
3:          k ← A[i]
4:          for j ← i down to 1 do
5:              if A[j − 1] > k then
6:                  A[j] ← A[j − 1]
7:              else
8:                  break
9:              end if
10:         end for
11:     end for
12: end procedure
```

Bubble Sort(泡沫排序)

時間複雜度：$O(N^2)$

Algorithm 2 Bubble Sort

```
1: procedure BUBBLE_SORT(A[], N)
2:     for i ← N − 1 down to 1 do
3:         for j ← 1 to i do
4:             if A[j − 1] > A[j] then
5:                 exchange A[j − 1] ⟷ A[j]
6:             end if
7:         end for
8:     end for
9: end procedure
```

Merge Sort(合併排序)

時間複雜度：$O(N \lg N)$

若我們有兩段已經排序過的序列，我們可以在 $O(N)$ 的時間內將他們合併成一段排序過的序列，於是合併排序法產生了!! 只要每次將序列分成等長的兩段遞迴處理，那我們排序長度為 N 的序列的時間就是 $T(N) = 2T(\frac{N}{2}) + O(N)$，解遞迴式可以得到 $T(N) = O(N \lg N)$。

Quick Sort(快速排序)

時間複雜度：平均 $O(N \lg N)$，最差 $O(N^2)$

若我們將序列分成兩個部份，並花 $O(N)$ 的時間重新排列，使左邊那段的元素都小於右邊那段的元素。不斷遞迴處理左右兩段，直到序列長度為 1，快速排序就完成了。但要注意的事，若測資經過設計，在最差的狀況下複雜度是 $O(N^2)$，且過度頻繁的切割序列、遞迴反而容易使速度變慢。

Counting Sort (基數排序)

時間複雜度：$O(N + C)$

若輸入的數值範圍 C 不大，我們可以直接計算每個數字的出現次數。假設數值皆為 0 到 $C - 1$ 之間的整數。

二分搜尋法 (Binary Search)

時間複雜度：$O(\lg N)$

我們要在一串已排序的序列中搜尋一個索引值或最接近的值，我們可以每次取中間的值將尋找的範圍縮小一半，得到複雜度是 $O(\lg N)$ 的算法。

Algorithm 3 Merge Sort

```
1: procedure MERGE_SORT(A[], L, R)
2:     if L < R then
3:         M ← (L+R)/2
4:         MERGE_SORT(A[], L, M)
5:         MERGE_SORT(A[], M + 1, R)
6:         MERGE(A[], L, M, R)
7:     end if
8: end procedure
9: procedure MERGE(A[], L, M, R)
10:    create B[R − L + 1]
11:    i ← L, j ← M + 1, k ← 0
12:    while i ≤ M and j ≤ R do
13:        if A[i] < A[j] then
14:            B[k] ← A[i]
15:            i ← i + 1
16:        else
17:            B[k] ← A[j]
18:            j ← j + 1
19:        end if
20:        k ← k + 1
21:    end while
22:    while i ≤ M do
23:        B[k] ← A[i]
24:        i ← i + 1, k ← k + 1
25:    end while
26:    while j ≤ R do
27:        B[k] ← A[j]
28:        j ← j + 1, k ← k + 1
29:    end while
30:    for i ← 0 to k − 1 do
31:        A[L + i] ← B[i]
32:    end for
33: end procedure
```

Algorithm 4 Quick Sort

```
1: procedure QUICK_SORT(A[], L, R)
2:     if L < R then
3:         M ← PARTITION(A[], L, R)
4:         QUICK_SORT(A[], L, M − 1)
5:         QUICK_SORT(A[], M + 1, R)
6:     end if
7: end procedure
8: function PARTITION(A[], L, R)
9:     pivot ← A[R]
10:    split ← L
11:    for i ← L to R − 1 do
12:        if A[i] ≤ pivot then
13:            exchange A[split] ⟷ A[i]
14:            split ← split + 1
15:        end if
16:    end for
17:    exchange A[split] ⟷ A[R]
18:    return split
19: end function
```

Algorithm 5 Counting Sort

```
1: procedure COUNTING_SORT(A[], N, C)
2:     initialize cnt[C] ← 0
3:     for i ← 0 to N − 1 do
4:         cnt[A[i]] ← cnt[A[i]] + 1
5:     end for
6:     k ← 0
7:     for i ← 0 to C − 1 do
8:         for j ← 1 to cnt[i] do
9:             A[k] ← i
10:            k ← k + 1
11:        end for
12:    end for
13: end procedure
```

Algorithm 6 Binary Search

```
1: function BINARY_SEARCH(A[], L, R, x)
2:     if L > R then
3:         return false
4:     end if
5:     M ← (L+R)/2
6:     if x = A[M] then
7:         return true
8:     else if x < A[M] then
9:         return BINARY_SEARCH(A[], L, M − 1, x)
10:    else
11:        return BINARY_SEARCH(A[], M + 1, R, x)
12:    end if
13: end function
```

1.2.3 雜項

大數運算

當數值的範圍很大，超過電腦內建的數值型態限制，無法讓程式直接運算，這時候就要用到小時候學過的直式加減乘除。大數運算的想法是開陣列記下每一位數，然後用直式的方法去做運算。常見的優化有兩種：

1. 記下大數的位數
2. 一個 int 中多存位數 (以不會溢位為原則)

離散化

當題目的數值範圍很大但實際上並不會全都用到時，我們可以使用離散化，將數值範圍縮小到比較好處理的範圍，通常是離線預處理。

1.2.4 Exercise

1. < TIOJ 1080 逆序數對 >
   對一個數列 S 來說，若 S 的第 i 項 S_i 與第 j 項 S_j 符合 $S_i > S_j$，並且 $i < j$ 的話，那麼我們說 (i, j) 是一個逆序數對。給定 S，請問總共有多少個逆序數對?

2. < TIOJ 1364 蛋糕內的信物 >
   給定 n 個數字，請在期望 $O(n)$ 的時間內求出第 k 大的數

3. < UVaOJ 10487 Closest Sums >
   給定一個 n 個數的集合 S，共有 m 個詢問，對於每個詢問給的數 k，請找出集合中兩個數的和，使其最接近 k。$(1 < n \leq 1000, 0 < m < 25)$

4. < NPSC 2003 年決賽 pA , TIOJ 1045 細菌培養 >
   在 $10^5 \times 10^5$ 的方格中，每個方格一開始都有一隻細菌。題目會做 T 次操作，每次操作會把一個矩形區域的細菌數倍增。所有操作都結束後的總細菌數為?$(T \leq 200)$

2 資料結構

2.1 資料結構

資料結構是指資料在電腦中儲存的某種特定方式，設計或選擇適當的資料結構能夠節省記憶體空間並提升演算法的效率。常見的資料結構包含 Array、Linked List、Stack、Queue、Tree、Graph、Heap、Hash、Disjoint Sets …等等。

2.1.1 鏈結串列 (Linked List)

鏈結串列是由許多節點連接而成的，一個結點中通常包含資料欄位以及指標，若是單向串列 (Singly Linked List) 則只須記錄指向前一個或下一個節點的指標，而雙相串列 (Doubly Linked List) 則必須同時記錄前一個及下一個節點。另外還有一種環狀串列 (Circular Linked List) 則是將同尾兩個節點相連。

鏈結串列與陣列 (Array) 都是很基本的資料結構，而它與陣列不同的地方在於 Linked List 可以在常數時間內插入或刪除元素，不過要查找第 k 個元素則需要線性的時間。

實作的方法可以直接使用指標，或是用陣列來模擬。

2.1.2 堆疊 (Stack)

堆疊可以想像成是一疊盤子，每次可以在這疊盤子的最上端放上新的盤子或拿走一個盤子，而這就分別對應到堆疊的兩種基本操作：push 和 pop。

push 是從堆疊頂端加入一個元素 (element)，而 pop 則是從堆疊頂端取出一個元素。堆疊的特性是 LIFO(Last In First Out, 後進先出)，即愈晚加入的元素會愈早被取出。

堆疊通常會使用陣列來實做，另外用一個 top 變數指向頂端元素的索引值。另外，若使用鏈結串列也是可行的，但由於這種實作方式沒有太大的優點，因此極少人使用。

Exercise

1. < Uva 514 Rails >

   某地之火車站長得如右圖一般。火車車廂由 A 處依 $1, 2, 3, \cdots, n$ 的順序進站。假設 station 處能停放無限多節車廂，如今給定一個 1 到 n 的重排，試問是否能在 station 處經過一些調度，使火車依照這樣的重排順序出站。

2. < TIOJ 1176 Cows >

   給一長度為 n 的數列，對於數列中的每個數字，請求出右邊第一個比它大的數字與它的距離。

3. < 括弧匹配 >
   首先定義:(1) 空字串是合法的。(2) 如果 A 是合法的，則 (A) 是合法的。(3) 如果 A 和 B 都是合法的，則 AB 是合法的。給定一個由左括弧與右括弧組成的字串，請判斷是否合法。
4. < TIOJ 1378 司令部的危機 >
   給一個包含 +、−、∗、/、及 () 的運算式的中序表示，請求出他的後序表示。
5. < HOJ 2 要我寫毛阿 >
   給 N 條線段 $[X_i,Y_i]$，問當中是否存任意兩條線段 $[X_a,Y_a]$ 符合 $[X_b,Y_b]$，$X_a < X_b < Y_a < Y_b$。(保證任意座標值皆不相同。)
6. < TIOJ 1106 遇見一株樹 >
   用一個包含 ∗ 與 () 的字串表示一棵樹，每對 () 代表一個樹枝的分叉點，每個 ∗ 代表一片葉子。求這棵樹的葉子數、最大深度及最多分岔的數量。
7. < TIOJ 1063 最大矩形 >
   給一個 $M \times N$ 的 0, 1 矩形，問由 1 組成的最大矩形面積為多少。
8. < TIOJ 1637 我愛台灣 >
   給定相鄰兩點的障礙值，並定義 a, b 兩點間能通訊的條件為電波強度大於 a, b 中間所有障礙值。問要使得任意兩點間皆能通訊，電波強度總和至少是多少?
9. < TIOJ 1549 笛卡爾樹 (Cartesian Tree) >
   給定陣列 $A[1\cdots n]$。你希望建立一棵二元樹滿足：
   1. 這棵樹的根是 $A[1\cdots n]$ 中最小的元素 (假設是 $A[k]$)
   2. $A[1\cdots k-1]$、$A[k+1\cdots n]$ 也都是笛卡爾樹。

2.1.3 佇列 (Queue)

佇列不同於 stack, 是一個先進先出 (FIFO, First In First Out) 的資料結構, 就像是排隊買飯, 愈先排的人可以愈先買。queue 支援兩種操作, 從尾端 (rear, tail) 插入資料 (enqueue) 及從前端 (front, head) 取出資料 (dequeue)。

實做時通常用陣列或串列來實現, 並用兩個變數來記錄前端和尾端。

但仔細思考就會發現當 front 一直往後移動時，前端許多的空間都浪費掉了，為了避免這種情況，我們可以使用環狀佇列，將原本 queue 的頭尾接在一起，便能解決浪費的問題。實做的時候通常是開一個大小為 QSIZE 的陣列，運用模運算來達成。

例題

1. < TIOJ 1489 核心字串 >
   給一個字串，請找出一個最短的區間包含 $a \sim z$ 所有字母。

2. < NTUJ 0839 Finding Seats >
   在一個 $R \times C (R, C \leq 300)$ 的地圖中，每個座標不是'.' 就是'x'。請找出面積最小的一個區域，使得這個區域中最少有 K 個'.'。輸出該區域的面積。

2.1.4 雜湊 (Hash)

在許多情況下，我們常會用一張表格來儲存我們需要資訊，但如果索引值是字串，int 範圍內的數等不能使用正常陣列來處理時，我們就會考慮使用 hash。

hash 的想法是通過一個雜湊函數 h() 將原本的索引值轉換成一個數字 k，於是我們便可以使用 data[k] 來儲存這個資料，每個操作期望的複雜度是 $O(1)$。

hash 函數通常是將原先的索引做某些神祕的處理後在模一個數，例如以下便一個以字串為 input 的 h 函數:

$\text{hash(str)} = (\text{str}[0] * 2 + \text{str}[1] * 3 + \text{str}[2] * 5 + \text{str}[3] * 7) \mod 11$

hash 函數沒有特定的寫法，需要發揮一些想像力及創意，不過要特別注意兩件事：計算的複雜度以及碰撞。碰撞的發生即是因為我們的 hash 函數極有可能不是一對一的函數，也就是說不同的索引值 hash 出來的結果可能相同。

碰撞處理

以下介紹幾種常見的碰撞處理方式。

1. ~~RP~~

2. 開散列法

又稱為拉鏈法 (chaining)，即用 linked list 將碰撞的資料存在同一個欄位中，是最常見的處理方法，但在取值時需要對同一格的 node 逐一比對。

3. 線性探測

當 hash 完後發現該格已經被使用的話，便往後尋找下一個空位來填入，實作相當容易，但若 hash 出來的值相當集中很可能造成所有的資料都集中於某一個區間內，同時也會大大增加操作的複雜度。

4. 多次 hash

對於同一個索引使用多個 h 函數來檢查，這樣可以大大降低碰撞的機會，並且在 h 函數不多的情況下做到 $O(1)$ 的複雜度。

Exercise

1. < TIOJ 1160 動態眾數問題 >
   給定 n 個數字，每讀到一個數字 k，便需輸出當前出現最多次的數字。

2. < TIOJ 1302 撿鞋運動 >
   給定以下三種操作:
   (1) add - 加入一筆資料，包含兩個字串 s_1, s_2
   (2) chk - 查詢。可能由 s_1 查詢 s_2 亦可由 s_2 查詢 s_1。
   (3) del - 刪除一筆資料 (以 s_1 或 s_2 做 key)。

2.2 Disjoint Sets

Disjoint Sets(並查集) 是用來處理多個不相交集合的資料結構。Disjoint sets 具有兩種操作，Union 是將兩個集合合併，Find 則是查詢某個元素所在的集合。

並查集用樹狀結構來表達一個集合，以樹的樹根來代表該集合。實際上我們只需記錄每個元素的父元素即可，當要查詢元素所在集合時，便相當於查詢其父元素所在的集合，如此遞迴下去，直到找不到父元素為止，就代表找到了樹根。而合併兩個集合就是將其中一個集合的樹根之父元素指向另一集合的樹根。

Algorithm 7 Disjoint Sets

```
1: procedure FIND(x)
2:     if x.parent ≠ x then
3:         return FIND(x.parent)
4:     else
5:         return x
6:     end if
7: end procedure
8: procedure UNION(x, y)
9:     xRoot ← FIND(x)
10:    yRoot ← FIND(y)
11:    xRoot.parent ← yRoot
12: end procedure
```

以上的樸素的作法在最差的情況下 (一條鏈)，Find 的複雜度會變為 $O(N)$，與暴力搜索的複雜度一樣，因此我們應該想辦法優化。以下介紹兩種常見的優化方式：

2.2.1 Path Compression

Path Compression 是在遞迴的過程中將所經過元素的父元素皆改為樹根，如此一來下次再查詢時便可節省許多時間。此優化的速度非常快，但複雜度不好估計。

Algorithm 8 Path Compression

```
1: procedure FIND(x)
2:     if x.parent ≠ x then
3:         x.parent ← FIND(x.parent)
4:         return x.parent
5:     else
6:         return x
7:     end if
8: end procedure
```

2.2.2 Union By Rank

對於每個集合，多記錄一個 rank 值代表其最大深度。在合併時，比較兩個集合的 rank，將 rank 較小的合併到 rank 較大的集合中。若 rank 相同則可任意可併，但合併後的集合 rank 須 +1。

Algorithm 9 Union By Rank

```
1:  procedure UNION(x, y)
2:      xRoot ← FIND(x)
3:      yRoot ← FIND(y)
4:      if xRoot.rank < yRoot.rank then
5:          xRoot.parent ← yRoot
6:      else if yRoot.rank < xRoot.rank then
7:          yRoot.parent ← xRoot
8:      else
9:          xRoot.parent ← yRoot
10:         yRoot.rank ← yRoot.rank + 1
11:     end if
12: end procedure
```

使用這個優化後，Find 的複雜度將變為 $O(\lg N)$。

若同時使用以上兩種優化, 則總複雜度會變成 $O(N \times \alpha(N))$。由於 $\alpha(N)$ 的值在競賽領域中 n 的範圍內幾乎可視為常數，故可假設所有對並查集的操作加總為 $O(N)$。

2.2.3 Exercise

1. 〈TIOJ 1312 家族〉
   給你很多組數字 a, b，代表 a 與 b 在同個家族。
   現在給你一個數字 k，求 k 所在的家族中編號最小的是?

2. 〈POJ 2492 雌蟲雄蟲〉
   給你很多對關係 x, y，代表蟲 x 跟蟲 y 為異性。
   請輸出是否有任何矛盾 (也就是有蟲為雌雄同體)。

3. 〈POJ 1182 食物鏈〉
   總共有 A、B、C 三種動物，A 吃 B，B 吃 C，C 吃 A。
   依序給你很多組關係 d, x, y，若 $d = 1$ 代表 x 跟 y 是同類，$d = 2$ 則代表 x 吃 y。
   若當前給的關係與之前的關係矛盾，代表此組關係是假的。
   求總共有多少組關係是假的?

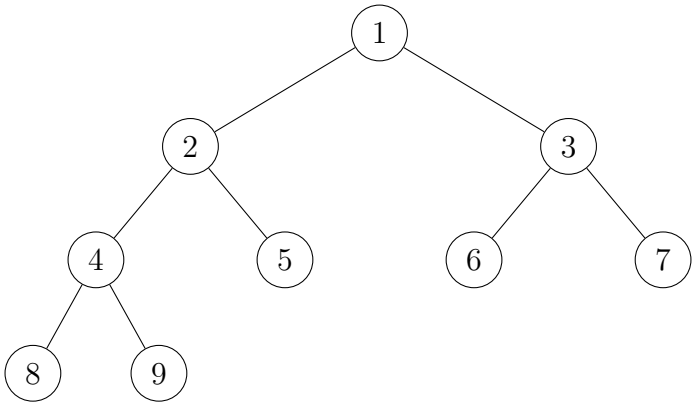
2.3 Binary Tree

2.3.1 定義

Binary Tree(二元樹) 是一種樹狀資料結構 $G = (V, E)$，滿足以下條件：

- G 為一棵有根的樹。
- 對於所有的 $v \in V$，v 至多只有 2 個子節點。

在二元樹裡，一個點的深度即為他到根節點的距離，二元樹的高度即為最深的節點的深度。葉子即為沒有子節點的節點，而我們說兩節點為兄弟節點表示他們的父節點相同。一個二元樹是完整的，表示二元樹的節點由上而下，再由左而右填滿整個樹。一個二元樹是滿枝的，表示二元樹除了深度為 H 的節點為葉子之外，其他的節點都洽有兩個子節點。



2.3.2 二元樹的儲存

如果二元樹非常接近完整二元樹，我們可以如下編號，另樹根的編號為 $1(0)$，每一個節點 x 的左子節點的編號為 $2x(2x+1)$，右子節點的編號為 $2x+1(2x+2)$，然後直接開一個陣列以編號當索引紀錄節點的資訊。另外當然也可以直接用儲存一般圖的方法，這個後面會提到。

2.3.3 二元樹的走訪

我們定義了三種二元樹的走訪方式： 可以知道「前」、「中」、「後」分別代表在拜訪左節

```
1: function PRE-ORDER(v)
2:     if v is not Null then
3:         print v
4:         PRE-ORDER(Left child of v)
5:         PRE-ORDER(Right child of v)
6:     end if
7: end function
```

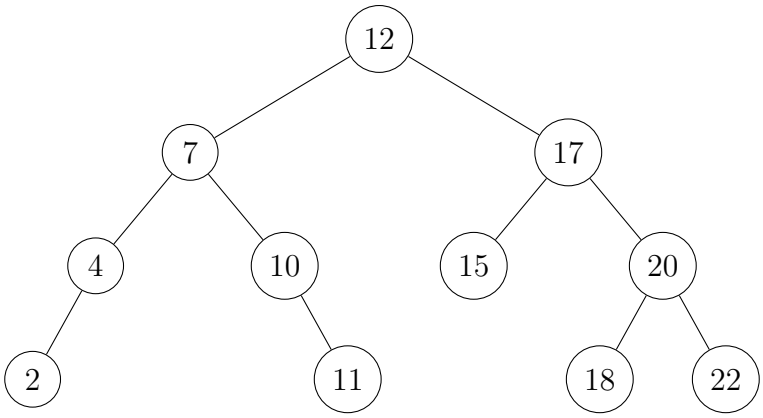
```
1: function IN-ORDER(v)
2:     if v is not Null then
3:         IN-ORDER(Left child of v)
4:         print v
5:         IN-ORDER(Right child of v)
6:     end if
7: end function
```

```
1: function POST-ORDER(v)
2:     if v is not Null then
3:         POST-ORDER(Left child of v)
4:         POST-ORDER(Right child of v)
5:         print v
6:     end if
7: end function
```

點前、拜訪左節點和右節點中間和拜訪右節點後將頂點的標號列印出來。

2.3.4 Binary search tree

Binary search tree(二元搜尋樹) 是一個特別的二元樹，滿足對任意節點 v，左子樹任意節點的鍵值都比 v 的鍵值小，右子樹任意節點的鍵值都比 v 的鍵值大。



這樣當我們要做操作時，我們只需要按照其定義操作即可。比如我們要查詢一個鍵值 k，便從樹根開始，如果當前節點 v 的鍵值 $v_{key} = k$ 即找到了，$v_{key} < k$ 則往左子節點走，$v_{key} > k$ 則往右子節點走。插入和刪除的動作基本上差不多。

而這些的操作時間複雜度和二元樹的高度 H 有關，大部分都是 $O(H)$，在最好或是平均的狀況之下，二元樹可以達到 $H = O(\lg N)$，但在最差的情況下卻有可能 $H = O(N)$。例如插入的數字恰好是由小到大的，此時二元樹會退化為一值鏈。為了解決這個問題，便衍生出許多自平衡的二元搜尋樹，我們將在後面介紹。

2.4 Heap

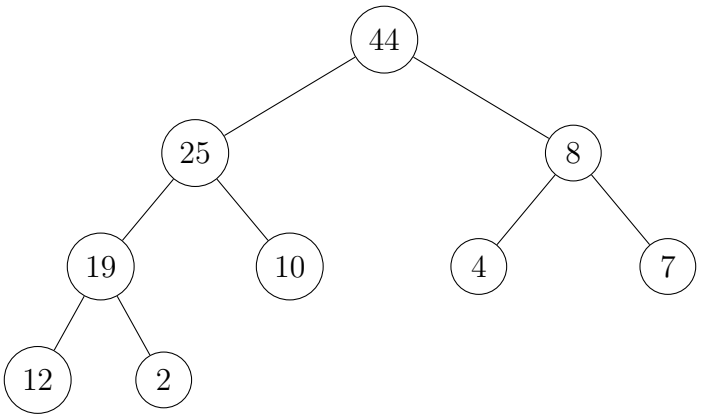
Heap 是一種資料結構，可以快速地維持、查詢極值。

	查詢最大值	插入元素	刪除元素	合併
Binary Heap	$O(1)$	$O(\lg n)$	$O(\lg n)$	–
Leftist Heap	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
Fibonacci Heap	$O(1)$	$O(1)$	$O(\lg n)$	$O(1)$

在這裡主要介紹 Binary Heap。Leftist Heap 可以說是可併堆中較好實作的一個，Fibonacci Heap 雖然理論複雜度優但實作複雜，而且實際上常數頗大的。
雖然 Binary Heap 幾乎可以被樹狀資料結構取代，但因實作容易，如果只需要查詢最大值的話不失為一個不錯的資料結構，並且 STL 中有 priority_queue 為一 max-heap 可供直接使用。

2.4.1 Binary Heap

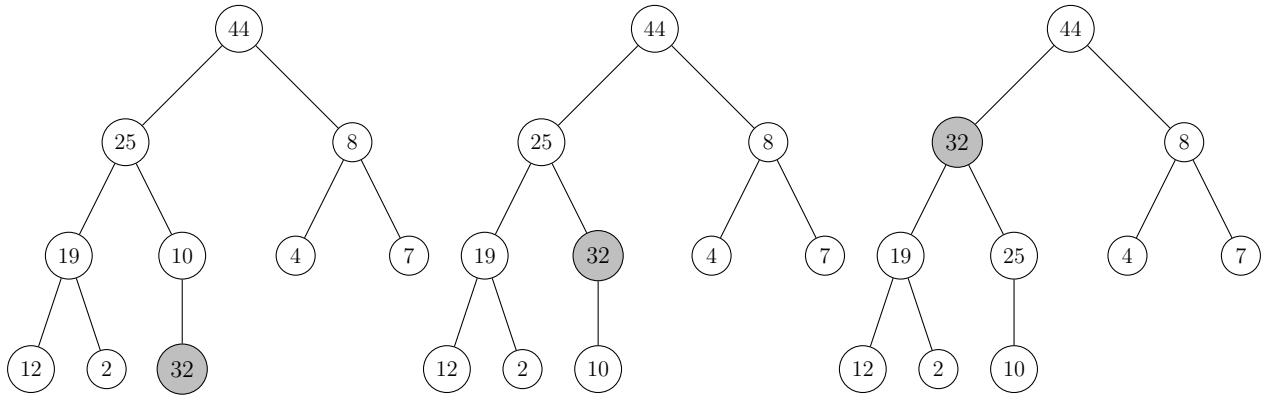
Binary Heap 是一個完整的二元樹，必須要滿足一個結構，任何一個節點的鍵值都要大於等於其子樹中任意節點的鍵值。也因為 Binary Heap 是一個完整的二元樹，我們通常直接用一個陣列儲存。



接著我們考慮其基本操作。

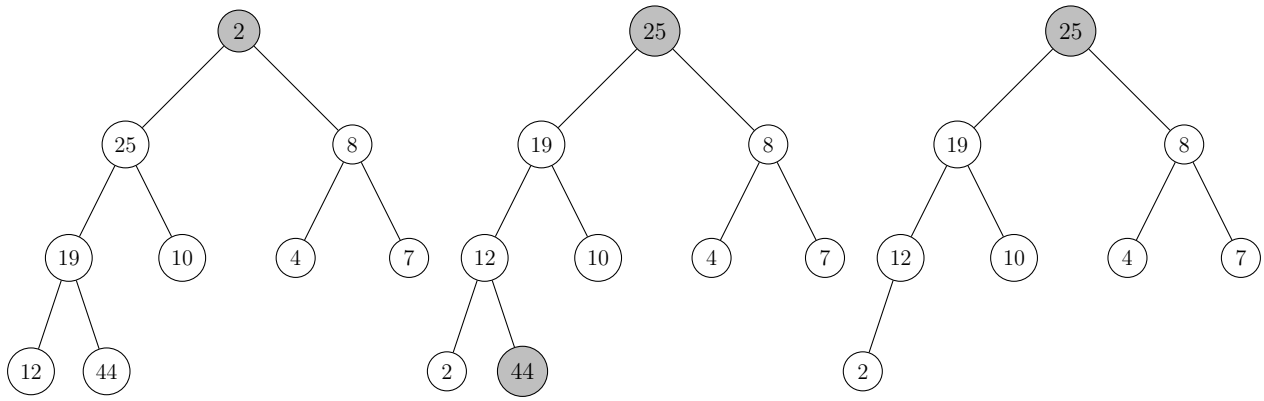
插入元素

當我們要插入一個元素時我們把新的節點加進最後的位置，並且如果新的節點的值比其父節點大，便將他往上調整，即和他父節點交換，直到他成為樹根或是其父節點比他還大為止。
可以知道此操作的複雜度為 $O(H) = O(\lg N)$，其中 H 為 heap 的高度。



刪除最大元素

與插入元素類似，我們直接將樹根還有最尾端的元素交換，並將新的樹根往下調整。此操作的複雜度為 $O(H) = O(\lg N)$，其中 H 為 heap 的高度。



查詢最大值

這個 easy，就是根節點的鍵值。

2.5 C++ Standard Library

2.5.1 簡介

C++ 與 C 語言在架構上基本相同，主要的差異在於 C++ 多了一套標準程式庫 (Standard Library)，整合了許多實用的功能。相較於手動實作，在編程上方便了許多，但代價是執行速度可能稍微慢一些。(部份函式的執行時間可能慢 2 倍左右，競賽題除非時限卡很緊，否則通常都能接受)

C++ 標準程式庫主要包含 IOStream、String、STL 容器、演算法及其他常用函式等，這次主要針對 IOStream 及 STL 作介紹。

2.5.2 String

string 用來儲存字串，功能類似 char[]，但好處為 string 是一個物件，作處理時比較方便，也比較不容易存取越界導致 Runtime Error，並且可以直接以 cin/cout 輸出入。需要引入標頭檔 <string>。

使用範例：

```
string s1,s2; //建立新字串
s1 = "Test"; //賦值
s2 = s1;
s1 = s1 + " And " + s2; //字串連接
int len = s1.length(); //取得長度
char c = s1[5]; //取得某個位置的字元
cout << s1 << endl; //可直接cout輸出
printf("%s\n", s1.c_str()); //轉為字元陣列
```

2.5.3 IOStream

IOStream 中文為輸入/輸出串流，是 C++ 處理輸出入的新方式。主要分為 iostream、fstream 及 stringstream。

iostream

包含 cin 及 cout，處理標準輸出入，功能類似 C 語言中的 printf() 及 scanf()。不同的是，cin/cout 本身是一個物件，可以執行更多元化的操作，並且支援 class 型別的輸出入。換行時，請輸出 endl。需要引入標頭檔 <iostream>。

使用方法：cin >> x; (讀入到 x)，cout << x << endl; (將 x 輸出並換行)，getline(cin, str); (讀入一整行到 string 型別的字串)

cin/cout 經常因為效率不足而不被建議使用，但實際上這可以在程式開頭呼叫以下函式以解決：

```
ios_base::sync_with_stdio(0);
```

實測結果：某題目，使用 printf/scanf 耗時 0.6s；使用 cin/cout 耗時 3.2s，優化後耗時 0.6s。

fstream

fstream 主要用來存取檔案，功能類似 freopen。用法類似 cin/cout，並且可以同時存取多個檔案，以及不與 cin 衝突。需要引入標頭檔 <fstream>。

使用範例：

```
fstream fin;
int a,b;
fin.open("test.txt", fstream::in);
fin >> a >> b;
fin.close();
```

stringstream

sstream 主要用來處理 string 型別字串，功能類似 sscanf()/sprintf()。用法類似 cin/cout，但把字串當作標準輸出入來處理，注意「輸出到字串」意指「寫入到字串」。需要引入標頭檔 <sstream>。

使用範例：

```
stringstream ss(stringstream::in | stringstream::out);
int a,b;
ss << "12 34";
ss >> a >> b;
```

2.5.4 STL Container

簡介

STL Container(STL 容器) 包含了許多不同的資料結構模板，並且支援相關的演算法，若能有效地利用，可以大大減低程式碼的長度。大致可分為三大類：Sequence Container(序列式容器)、Container Adaptor(容器配接器)、Associative Container(關聯式容器)，主要包含有 vector、deque、list、stack、queue、priority_queue、set、multiset、map、multiset 等等。

而 STL Container 的元素存取主要靠 Iterator(迭代器)，根據不同的 Container 有不同的型態。

Iterator

Iterator(迭代器) 是用來存取 Container 裡面的元素，類似 Array 的 Pointer。對於每個 Container 有不同的型別，例如 `vector<int>::iterator`。

Iterator 支援幾種運算子，如下範例：

```
vector<int> vi;
vector<int>::iterator it; //建立新Iterator
it = vi.begin(); //賦值
int i = *it; //提領
vi++; //往前進一格
vi += 10; //往前進指定距離
if(it < vi.end()) //比較大小，也可用!=比較是否相等
{
    *it=5; //提領並賦值
    it++;
}
```

Vector

vector 簡單來說就是動態版的 array，不用預先決定大小。常用於不確定元素的數量，或者元素的數量會隨時變化的情況。一般來說，剛建立的 vector 裡面是空的，由使用者一個一個將元素加入，通常 vector 的大小隨時都為裡面的元素數量。

vector 本身是一個模板，需要套用一種型別進去，換句話說，就是要設定你這個 vector 裡面裝的是「什麼東西」(所有的 STL Container 都是如此)。`vector<int>` 才是一個完整的型別，代表這個動態陣列裡面裝的是 `int` 型別；而其 iterator 的型別為 `vector<int>::iterator`。

vector，以及稍後介紹的 deque、list，屬於 Sequence container(序列式容器)，裡面的元素具有一定的順序。

使用時需引入標頭檔 `<vector>`。

- `vector<int> vi` 建立一個 int 型別的 vector
- `vi.clear()` 清空 vector
- `vi.size()` 取得元素數量
- `vi.reserve(x)` 預先將大小設為 x，新增的元素以 default constructor 初始化
- `vi.empty()` 判斷是否為空
- `vi.push_back(i)` 從後面加入一個元素，值為 i
- `vi.pop_back()` 從後面刪掉一個元素
- `vi[i]` 取得第 i 個元素 (從 0 開始計算)
- `vi.begin()` 取得指向第一個元素的 iterator
- `vi.end()` 取得指向最後一個元素後面一格的 iterator
- `vi.front()` 取得第一個元素的值
- `vi.back()` 取得最後一個元素的值

Deque

deque 全名為 double-ended queue(雙端佇列)，是一種可以從前端或後端新增元/刪除元素的資料結構 (因此同時具備 stack 和 queue 的功能)。

deque 也具有 vector 的功能，不過缺點是 deque 的 iterator 在插入新元素後會失效。

使用時需引入標頭檔 `<deque>`。

- `deque<int> dq` 建立一個 int 型別的 deque
- `dq.clear()` 清空 deque
- `dq.size()` 取得元素數量
- `dq.empty()` 判斷是否為空
- `dq.push_front(i)` 從前面加入一個元素，值為 i
- `dq.pop_front()` 從前面刪掉一個元素
- `dq.push_back(i)` 從後面加入一個元素，值為 i
- `dq.pop_back()` 從後面刪掉一個元素
- `dq[i]` 取得第 i 個元素 (從 0 開始計算)
- `dq.begin()` 取得指向第一個元素的 iterator
- `dq.end()` 取得指向最後一個元素後面一格的 iterator
- `dq.front()` 取得第一個元素的值
- `dq.back()` 取得最後一個元素的值

List

即如同剛才介紹的 list 資料結構，由一堆元素串在一起所形成。適合從中間插入值和分割、合併、區段拼接等，可以在 $O(1)$ 時間完成。但在隨機查詢元素時缺乏效率，需要 $O(N)$ 的時間。

使用時需引入標頭檔 `<list>`。

- `link<int> lk` 建立一個 int 型別的 link
- `lk.clear()` 清空 link
- `lk.size()` 取得元素數量
- `lk.empty()` 判斷是否為空
- `lk.push_front(i)` 從前面加入一個元素，值為 i
- `lk.pop_front()` 從前面刪掉一個元素
- `lk.push_back(i)` 從後面加入一個元素，值為 i
- `lk.pop_back()` 從後面刪掉一個元素
- `lk.begin()` 取得指向第一個元素的 iterator
- `lk.end()` 取得指向最後一個元素後面一格的 iterator
- `lk.front()` 取得第一個元素的值
- `lk.back()` 取得最後一個元素的值
- `lk.insert(it, x)` 在 it 這個 iterator 所指的元素前面，插入一個值為 x 的元素
- `lk.insert(it, first, last)` 在 it 所指的元素前面，插入 first 到 last 中的所有元素 (first、last 為 iterator)

- `lk.erase(it)` 刪除 it 所指的元素
- `lk.erase(first, last)` 刪除 first 到 last 中的所有元素 (first、last 為 iterator)

Stack

即如同剛才介紹的 stack 資料結構，具有先進後出的特性，因此只能由後端新增/刪除元素。

stack，以及稍候介紹的 queue、priority_queue，屬於 Container adaptor(容器配接器)，是基於一種 Sequence container 實作而得。你可以選擇使用 vector、deque 或 list 作為底層的容器，預設為 deque。

使用時需引入標頭檔 `<stack>`。

- `stack<int> st` 建立一個 int 型別的 stack
- `stack<int, vector<int> > st` 以 vector 作為底層容器建立 stack
- `sk.size()` 取得元素數量
- `sk.empty()` 判斷是否為空
- `sk.push(i)` 從頂端加入一個元素，值為 i
- `sk.pop()` 刪除頂端元素
- `sk.top()` 取得頂端元素 (即最後一個元素)

Queue

即如同剛才介紹的 queue 資料結構，具有先進先出的特性。因此只能由一端加入，另一端刪除元素。

由於 queue 需要從前端刪除元素，故 vector 不符合要求，只有 deque 和 list 可以作為底層容器的選擇，預設為 deque。

使用時需引入標頭檔 `<queue>`。

- `queue<int> q` 建立一個 int 型別的 queue
- `queue<int, deque<int> > q` 以 deque 作為底層容器建立 queue
- `q.size()` 取得元素數量
- `q.empty()` 判斷是否為空
- `q.push(i)` 從後端加入一個元素，值為 i
- `q.pop()` 從前端刪除一個元素
- `q.front()` 取得前端元素 (即 out 端第一個元素)
- `q.back()` 取得後端元素 (即 in 端第一個元素)

Priority Queue

priority queue 事實上是下次會介紹的 heap 資料結構。支援在 $O(\lg N)$ 時間加入新元素，$O(1)$ 時間取得最大元素的值，$O(\lg N)$ 時間刪除最大元素。

由於 list 不支援隨機存取，故只有 vector 和 deque 可以作為底層元素，預設為 deque。另外 priority queue 需要一個比較函式，以作為比較大小的依據。C++ 的模板大於函

式 greater<int>、小於函式 less<int> 皆可以使用，或者也可以使用自定函式，預設為 less<int>。欲取最大值時使用小於函式，取最小值時使用大於函式。

使用時需引入標頭檔 <queue>。

- priority_queue<int> pq 建立一個 int 型別的 queue
- priority_queue<int, deque<int>, greater<int> > pq 以 deque 作為底層容器建立 queue，以 greater 作為比較函式 (也就是使頂端為最小元素)
- pq.size() 取得元素數量
- pq.empty() 判斷是否為空
- pq.push(i) 加入一個元素，值為 i
- pq.pop() 刪除最大元素
- pq.top() 取得最大元素的值

Set, Multiset, Map, Multimap

set、multiset、map 和 multimap 屬於 Associative container(關聯式容器)，即以 Key(索引) 來存取、查詢元素 (而 Sequence container 以相對位置來存取元素)。內部的結構是一棵平衡二元樹。這部份往後會再作介紹。

在這些容器中，每個元素有一個 Key 值，並且所有的元素依照 Key 值排序，支援 $O(\lg N)$ 時間插入/刪除元素、查詢指定 Key 值的元素、某 Key 值存不存在、某 Key 值以上有幾個元素等操作。

這 4 種容器的運作方式其實相當類似，差別只在於：

- set、multiset：元素的值本身即為 Key 值。
- multiset 容許重複的元素值，而 set 元素值為唯一。
- map、multimap：元素的值與 Key 不同，Key 值是另外指定的。
- multimap 容許同一 Key 值有多個元素，而 map 中一個 Key 對應到唯一的元素。

因為 map、multimap 的元素包含了 Key 值和元素值兩個資訊，故以 pair 的形式儲存。如 Key 型別為 char，元素值型別為 int，則元素的型別為 pair<char, int>。

set、multiset 需要引入標頭檔 <set>，map、multimap 需要引入標頭檔 <map>。

使用方式：

- set<int> s 建立元素為 int 的 set
- multiset<int> ms 建立元素為 int 的 multiset
- map<int, string> m 建立 Key 為 int，元素值為 string 的 map
- multimap<int, string> mm 建立 Key 為 int，元素值為 string 的 multimap
- size() 取得元素數量
- empty() 判斷是否為空
- clear() 清空
- insert(i) 新增一個值為 i 的元素 (set、multiset)
- insert(make_pair(i,x)) 新增一個 Key 值為 i，元素值為 x 的元素 (以 pair 形式插入) (map、multimap)

- `erase(i)` 刪除值為 i 的元素
- `erase(it)` 刪除 it 所指的元素
- `begin()` 取得指向第一個元素的 iterator
- `end()` 取得指向最後一個元素後面一格的 iterator
- `find(i)` 取得 Key 值為 i 的元素的 iterator(若找不到則回傳 end())
- `count(i)` 取得 Key 值為 i 的元素的數量
- `lower_bound(i)` 取得第一個 Key 值大於等於 i 的元素的 iterator
- `upper_bound(i)` 取得第一個 Key 值大於 i 的元素的 iterator
- `m[i]` 取得 Key 值為 i 的元素的值 (map)
- `*it` 取得 it 所指的元素的值 (set、multiset), Key-Value Pair(map、multimap)
- `it->first` 取得 it 所指的元素的 Key 值 (map、multimap)
- `it->second` 取得 it 所指的元素的元素值 (map、multimap)

Container 操作時間複雜度

	Sequence containers			Associative containers		Container Adaptors		
Members	vector	deque	list	set	map	stack	queue	p_queue
`operator=`	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$			
`begin()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$			
`end()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$			
`size()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`empty()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`front()`	$O(1)$	$O(1)$	$O(1)$				$O(1)$	
`back()`	$O(1)$	$O(1)$	$O(1)$				$O(1)$	
`top()`						$O(1)$		$O(1)$
`operator[]`	$O(1)$	$O(1)$			$O(\lg N)$			
`insert()`	$O(N)$	$O(N)$	$O(1)$	$O(\lg N)$	$O(\lg N)$			
`erase()`	$O(N)$	$O(N)$	$O(1)$	$O(\lg N)$	$O(\lg N)$			
`clear()`	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$			
`push_front()`		$O(1)$	$O(1)$					
`pop_front()`		$O(1)$	$O(1)$					
`push_back()`	$O(1)$	$O(1)$	$O(1)$					
`pop_back()`	$O(1)$	$O(1)$	$O(1)$					
`push()`						$O(1)$	$O(1)$	$O(1)$
`pop()`						$O(1)$	$O(1)$	$O(1)$
`find()`				$O(\lg N)$	$O(\lg N)$			
`count()`				$O(\lg N)$	$O(\lg N)$			
`lower_bound()`				$O(\lg N)$	$O(\lg N)$			
`upper_bound()`				$O(\lg N)$	$O(\lg N)$			

2.5.5 STL Algorithm

STL 裡面除了一些資料結構以外，還內建有一些常用的演算法以及實用函式，在此稍作介紹。

需要引入標頭檔 <algorithm>。

sort()、stable_sort()

sort 即是排序演算法，複雜度為 $O(N \lg N)$。分為兩種版本，sort() 及 stable_sort()，後者會維持同樣大的元素的順序。

```
sort(first, last);
    //將[first,last)間的元素排序(first和last為iterator或pointer)
    //注意last本身不算，使用vector的話一般會將first=v.begin(), last=v.end()
    //使用array的話將first=arr, end=arr+N (N為陣列大小)
sort(first, last, cmp);
    //以自訂比較函式cmp來排序
```

reverse()

將序列的元素倒轉。例如 $\{1,2,3,4\} \rightarrow \{4,3,2,1\}$。時間複雜度 $O(N)$。

注意不要把 reverse() 跟 vector 的 reserve() 搞混了。

```
reverse(first, last);
    //將[first,last)間的元素倒轉
```

lower_bound()、upper_bound()

lower_bound() 以二分搜方式取得第一個大於等於 (upper_bound() 只能大於) 指定值的元素的 iterator，而複雜度為 $O(\lg N)$。注意搜尋的區間必須為已排序，否則結果無法預期。

```
vector<int>::iterator it = lower_bound(first, last, i);
    //在[first,last)間找第一個大於等於i的元素，注意last本身不算
it = lower_bound(first, last, i, cmp);
    //使用自訂cmp比較函式
int a = *it;
    //lower_bound()回傳的是iterator，需要提領才能得到元素值
```

binary_search()

也是二分搜，但只查詢指定元素是否存在，回傳型態為 bool。同樣地序列須為已排序。複雜度為 $O(\lg N)$。

```
bool b = binary_search(first, last, i);
    //在[first,last)區間內查詢是否有等於i的元素
b = binary_search(first, last, i, cmp);
    //使用自訂比較函式cmp
```

fill()

將一個序列的所有值初始化為指定值。效率較 for 迴圈為高，且適用範圍比 memset() 廣一些 (例如適用於 class 型別)。

```
fill(first, last, i);
    //將[first,last)間的所有元素設定為i
```

mismatch()

取得兩個序列第一個不相等的元素，回傳值為一個 pair，包含分別指向兩個序列該位置的 iterator。也常用於 string 字串。時間複雜度 $O(N)$。

```
pair<vector<int>::iterator, vector<int>::iterator> = mismatch(first1, last1, first2);
    //將序列[first1,last1)與序列[first2,...比對，回傳第一個不同的位置
```

next_permutation()

取得下一個排列方式，並將它寫入原本的序列。$N!$ 爆搜的時候很常用 (?) 例如 $\{1,2,3\}\to\{1,3,2\}\to\{2,1,3\}\to\cdots$

成功的時候回傳 true，失敗則回傳 false。時間複雜度 $O(N)$。

```
next_permutation(first, last);
    //將[first,last)間的序列換成下一個排序方式。
```

random_shuffle()

將序列隨機重新排列。例如 $1,2,3,4,5,6,7\to 6,1,5,7,4,2,3$

```
random_shuffle(first, last);
    //將[first,last)間的元素隨機排列
```

3 常見演算法

3.1 搜索

搜索是最基本的演算法，也就是俗稱的「暴力」法，嘗試所有的可能性以求得答案。縱使如此，方法的不同還是會有速度上的差異。

3.1.1 枚舉

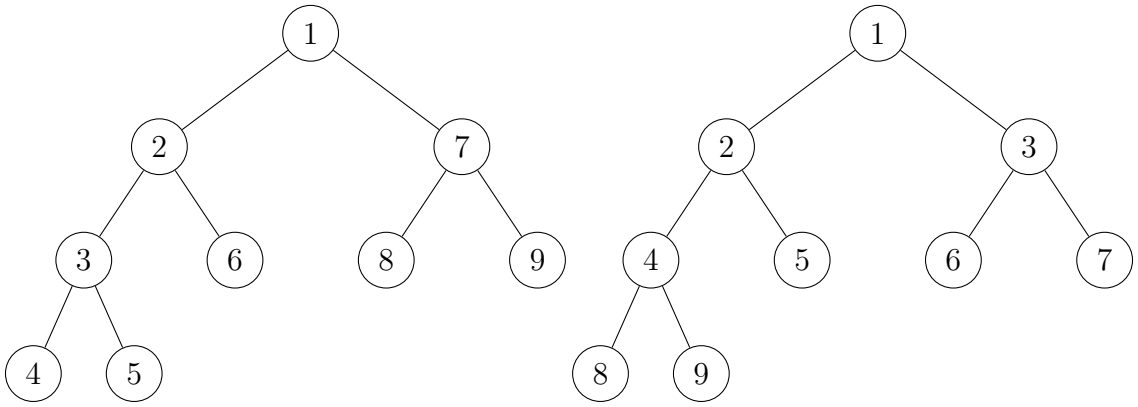
枚舉每種可能的答案。
經過一番枚舉之後... 沒辦法我還是只想到這兩行可以打。

3.1.2 狀態空間搜索

當我們在搜索的時候，可以想成我們在一張圖上行走，對問題在某一時刻狀況的數學描述即稱為狀態，可以想像成圖上的一個點，一開始我們會在起始狀態，而我們最終希望能到達結束狀態。從一個狀態轉換到另外一個狀態的操作就稱為狀態轉移，即為圖上的一條邊。所有狀態的集合就稱為狀態空間。
只不過我們可能不知道圖上所有的邊，不知道圖上的兩點是否可以通行，甚至不能確定圖上有那些點，因此我們要進行搜索，即嘗試走過圖上的點，盡可能快地達到終點。以下介紹不同的搜索方式。

3.1.3 DFS 與 BFS

DFS(Depth First Search) 深度優先搜索，即依照深度優先，盡可能地往深度越深的地方搜索，若走到死路即回頭，通常使用遞迴或堆疊來實現。他的優點是實現簡潔，而且佔用空間少 (空間複雜度僅 $O(d)$, 其中 d 為最大深度)，但沒有辦法在深度無上界的狀態空間下搜索。BFS(Breath First Search) 廣度優先搜索，就有點像是「地毯式」搜索，一層一層優先擴展與初始狀態較近的節點，通常使用佇列實做。可以在深度無上界的狀態空間下搜索，但需要的記憶體往往是很可怕的。




```
1:  function DFS(Init_State)
2:      Push Init_State Into Stack
3:      while Stack Is not Empty do
4:          Current_State ← Pop Stack
5:          Mark Current_State As visited
6:          for each v adjacent to Current_State do
7:              if v Is not Visited then
8:                  Push v Into Stack
9:              end if
10:         end for
11:     end while
12: end function
```

```
1:  function BFS(Init_State)
2:      Push Init_State Into Queue
3:      Mark Init_State As visited
4:      while Queue Is not Empty do
5:          Current_State ← Pop Queue
6:          for each Next_State adjacent to Current_State do
7:              if Next_State Is not Visited then
8:                  Push Next_State Into Queue
9:                  Mark Next_State As visited
10:             end if
11:         end for
12:     end while
13: end function
```

3.1.4 BiBFS

假設我們使用 BFS 進行搜索，並且每個狀態都有 n 種轉移，那當我們在搜索第 d 層的時後便會有 $O(n^d)$ 種狀態，往往會讓記憶體無法負荷。這時候如果狀態轉移的動作是可逆的，並且我們明確的知道結束狀態，我們其實可以一邊從起點搜索下去，另一邊從終點倒推回去，而讓搜索的層數降低為 $d/2$。

3.1.5 IDDFS

BFS 雖能保證找到狀態轉移次數最少的解，但是太佔空間。因此，我們可以用迭代加深的 DFS 來模擬廣度優先搜索。
IDDFS 即是每次限制 DFS 的最大深度 limit，如果 DFS 的深度超過 limit 就不再繼續搜索下去。當我們在 limit 之內找不到解時，加大 limit 再進行一次 DFS。
雖然乍看之下好像做了許多重複搜索的工作，但是因為搜索樹的每一層級往往擴展節點數量差異極多，因此迭代加深的工作效率仍然不錯。IDDFS 的空間複雜度僅僅是 $O(d)$。

3.1.6 A*

A* 是一種啟發式演算法，主要的想法很簡單，即是改變搜索的順序，讓我們可以盡快地走到終點。以直覺來說我們每一次都應該選擇「現在看似離終點最近的狀態」繼續走下去，而這時候就必須有一個估計函數估計目前狀態到終點的距離是多少。
另 $h(x)$ 為狀態 x 的估計函數，$h'(x)$ 為狀態 x 到終點的實際距離，$g(x)$ 為狀態 x 從起點到現在的距離，我們便用 $f(x) = g(x) + h(x)$ 來決定擴展節點的順序。
但當然，好的估計函數會影響到搜索的速度，甚至影響搜索的正確性。一個好估計函數應該要滿足以下條件。

- $h(x) \leq h'(x)$，這個不等式保證了估計函數的正確性。
- $h(x)$ 越大越好，$h(x)$ 越大 *A 算法便會越根據你的估計函數來決定搜索順序。

在實作上通常會使用 priority queue 來維護頂點 $f(x)$ 值的大小順序。

3.1.7 IDA*

IDA* 的缺點就是記憶體需求太大，因此與 IDDFS 類似，我們設定一個限制，直接忽略 $f(x)$ 值大於深度限制的節點，並以迭代加深的方式搜索。

Algorithm 10 IDA*

```
1:  function IDDFS(State,g,limit)
2:      f ← g + h(State)
3:      if f > Limit then
4:          return false
5:      end if
6:      if S thentate Is Goal_State
7:          return True
8:      end if
9:      for each Next_State adjacent to State do
10:         ret ← IDDFS(NextState,g + 1, limit)
11:         if r thenet Is not False
12:             return ret
13:         end if
14:     end for
15: end function
16: function IDA*(Init_State)
17:     Limit ← 0
18:     repeat
19:         Limit ← Limit +1
20:         ret ← IDDFS(Init_State ,0 , Limit)
21:     until ret is False
22: end function
```

3.1.8 Exercise

1. 〈Packing Rectangles (USACO 1-4, packrec)〉
   給定四個矩形的長、寬。矩形可以任意移動、順/逆時針旋轉 90 度，但是不能重疊。如果要把四個矩形包在一個矩形當中，則該矩形的面積最小是多少? 請輸出所有解。

2. 〈經典問題〉
   給一個 $M \times N$ 的格子矩形，每個格子不是 1 就是 0，你每次可以對一個格子進行操作，把那個格子和他周圍的格子 1 變成 0、0 變成 1，問你至少要做幾次才能使所有格子都變成 0。$(M, N \leq 15)$

3. 〈經典問題〉
   輸出數獨問題的所有解。

4. 〈射手座之日 (NPSC 2006 決賽, pB)〉TIOJ 1099
   對一個三維空間的座標 (x, y, z)，你可以做它兩件事：

   (a) 把 x, y, z 任意調換

   (b) 把 (x, y, z) 變成 $(2y-x+1, 2x-y-1, z)$

   問有沒有可能把座標從 (x_1, y_1, z_1) 變成 (x_2, y_2, z_2)。

5. 〈Snake(Codeforces 225D)〉
   你現在在一個 $M \times N$ 有障礙物的格子上玩貪吃蛇，你蛇的長度為 L，並且有個格子上有食物，你每次頭可以往前、左或是右移動，你某格身體便會移動到前面那一格的身體的位置。注意到你的頭不可以撞到任何障礙物或是身體，問你至少要動幾步。$(M, N \leq 15, L \leq 9)$

6. 〈15-Puzzle〉TIOJ 1573
   給一個 4×4 的盤面，當中填有 $1, 2, \cdots, 15$ 的數字，並有一個空位。每次你可以移動一個與空位相鄰的數字到空位去，請問最少要幾部才能到達指定的盤面？

3.2 Greedy

3.2.1 Greedy Method (貪婪法)

Greedy Method(貪婪法) 顧名思義就是貪心!! 在每一步選擇都選在當前狀態下最好的選擇，是最常應用在生活上的演算法。Greedy 可以幫助我們解決一些最優化問題或是具有最佳子結構的問題。最佳子結構意思是局部最優解能推出全局最優解，也就是能把一個問題分成幾個子問題，而子問題的最優解能遞推出問題的最優解。Greedy 常常不能得到正確的答案，所以得小心證明，而一旦問題可以用 Greedy 解決，那通常會是解決這個問題的最好方法。也因為能在短時間內求出近似的答案，Greedy 在競賽中常成為假解的好方法。

$$\text{Greedy} = \text{觀察} + \text{假設} + \text{證明}$$

當然 Greedy 的問題不一定都這麼單純，有時還會配上其他演算法或資料結構甚至是數學，讓 Greedy 不容易被想到，接著就直接來做些練習吧!!

3.2.2 Exercise

1. 線段覆蓋
   給你一維座標上 N 條直線的起點和終點 $x_i, y_i (x_i < y_i)$，問最多能選幾條線段並使他們互相不重疊。

2. < NPSC2005 pB > 惱人的零錢
   東東想要買一個價格 C 的物品，告訴你東東和老闆各別擁有的一元、五元、十元、二十元、五十元硬幣的數量，東東想要盡量減少他身上的硬幣數量但一定要讓老闆有得找，請輸出他身上最少能剩下多少硬幣？

3. < NPSC2005 pA > 誰先早餐
   給你 n 個要吃早餐的人和一個廚師。當然那些人可以同時一起吃，但廚師一次只能做一道菜。給定每個人要吃的菜需要做多久，以及那個人需要吃多久，試問至少要多少時間，才能讓全部人都吃完？

4. < STEP5 0021 > 背包問題 (建中 100 校內初賽 pB)
   有兩個背包容量分別為 N, M，現在要將 $N + M$ 件物品裝進背包，每件物品都有他的重量 w_i，求字典序最小的放法使兩個背包的重量平均數加起來最小。

5. < STEP5 0031 > 蘿莉切割問題 (建中 101 校內初賽 pA)
   你要將一塊長度 L 的木板切成 N 段，每段長度是 $A_i (A_1 + A_2 + ...A_n = L)$，而將一塊長度 X 的木板切開需要 X 的花費，求花費最小的切割方式。

6. < POI XII > Toy Cars
   有一個小屁孩想玩車車，但車車放在架子上要請他媽媽幫他拿，小屁孩總共有 N 台車

車，地上總共只能放 K 台，現在給你小屁孩玩 p 次車車的順序，問媽媽最少要幫小屁孩拿幾次車？$(1 \leq k \leq n \leq 100,000; 1 \leq p \leq 500,000)$

7. <POI X> Chocolate
   你要將一塊 $M \times N$ 的巧克力切成 1×1 的大小，直向有 $M-1$ 個切點分別需要花費 $X_1, X_2, ...X_{M-1}$，橫向有 $N-1$ 個切點分別花費 $Y_1, Y_2...Y_{N-1}$，問最小的花費是多少。

8. < TIOJ 1432,1465 > 骨牌遊戲, H 遊戲秘笈
   給定一個由 N 個數字組成的序列，求使分成 K 段之後每段總合的最大值最小的分法。

9. < TIOJ 1406 > FISH
   有一條大馬路，馬路上有 N 個城鎮，每個城鎮都有一些魚。現在我們要運送這些魚使每個城鎮的都有大於等於 Y 噸的魚。然而每運送一公里就會被搶走一噸的魚。想請問你 K 最大可為多少？

3.3 Divide and Conquer

Divide and Conquer(分而治之, D&C , Fun and J*zz) 主要的想法即是將一個大問題切成數個相同的子問題，然後遞迴下去解決這些子問題，而關鍵即是怎麼將這些子問題合併成原本問題的答案，合併的演算法的效率往往會影響整個演算法的時間複雜度。

3.3.1 一些重要的遞迴式

這裡我們列出了一些重要的遞迴式的時間複雜度。

- $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \Leftarrow T(n) = O(n \lg n)$。
- $T(n) = 3T\left(\frac{n}{2}\right) + O(n) \Leftarrow T(n) = O(n^{\lg 3}) \sim O(n^{1.58})$。
- $T(n) = 2T\left(\frac{n}{2}\right) + O(\lg n) \Leftarrow T(n) = O(n \lg^2 n)$。
- $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2) \Leftarrow T(n) = O(n^2)$。

其他的可以使用遞迴樹計算。

3.3.2 Exercise

1. 〈河內塔, 經典問題〉
   現在有三根柱子，一開始在第一根柱子上有 n 個圓盤，由大到小疊好，請印出所有搬動圓盤步驟，把所有圓盤從 1 號柱搬到 3 號柱子，且移動過程中大圓盤不得疊在小圓盤上。

2. 〈平面最近距點對, 經典問題〉
   給定平面上的 n 個點，尋問 (直線距離) 最近的兩個點的距離。($n \leq 10^5$)

3. 〈快速多項式乘法, 經典問題〉
   給兩個 n 次的多項式 f, g，計算 fg。($n \leq 10^5$)

4. 〈經典問題〉
   給你一棵 n 個點的樹，詢問樹上距離小於等於 k 的點對的數量。($n \leq 10^5, m \leq 10^9$)

5. 〈計步器 (TOI 2011)〉HOJ 113
   給你一棵 n 個點的樹，詢問樹上兩點間路徑上的邊的權重和 $\bmod m$ 的最大值。($n \leq 10^5, m \leq 10^9$)

3.4 Dynamic Programming

Dynamic Programming(動態規劃) 是以數個相關問題的答案推導出原本問題的答案，我們把原本的問題稱作母問題，用來推導出母問題的相關問題稱作子問題。雖然這樣看似遞迴就可以做到，但動態規劃將重複子問題的答案記下來節省時間。我們通常會用一些數字來描述抽象的問題，稱作狀態，而我們用子問題來解決母問題的數學方法就稱為轉移，通常用動態規劃解決的問題，狀態都是要可枚舉的且數量在一定範圍以下，且每個狀態的轉移是相似的。但當然，我們要用動態規劃來解決問題必須滿足一些條件，才能保證正確性。

- 最佳子問題
  對一個最佳解而言，他的任意一個局部都要是該子狀態的最佳解。
- 無後效性
  若轉移到子狀態時的決策會影響到子狀態轉移到母狀態時的決策選項，我們說這個這組狀態有後效性。我們無法在一個有後效性的狀態下定義最佳解，自然無法符合最佳子問題。所以當狀態有後效性時，我們就可能得以別種狀態來描述問題。

3.4.1 實作方式

通常有兩種實作的方式

- Bottom-Up
  Bottom-Up 顧名思義就是由下而上的 DP。在計算一個狀態之前，先將轉移會用到的所有狀態算出。優點是通常可以用迴圈來跑，不會有太大的常數，並且有可能可以安排 DP 的順序節省記憶體。缺點是可能會計算到許多不會用到的狀態。
- Top-Down
  Top-Down 在計算一個狀態時，如果發現有某個需拿來轉移的狀態尚未被計算，就遞迴下去先行算出該子狀態的答案並記起來。相對 Bottom-Up 比較好實作，而且沒用到的狀態也不會被計算到。然而呼叫遞迴會消耗時間，所以在狀態幾乎都用到的狀況下，此種方法會比 Bottom-Up 還慢。另外，使用這種方法有時需注意遞迴過深的問題，而且記憶體通常是需要開滿的。

3.4.2 Exercise-基本 DP

在許多求極值的問題也應該想一想如果題目要求輸出一組解的話要怎麼處理。

1. 〈IOI 1994 The Triangle〉 TIOJ 1288
   給你一個數字構成的邊長為 n 的正三角形，對於每個位置你接下來可以走到他左下方或是右下方的數字，問你從頂點走到底邊的路徑上數字總和的最大值。($O(n^2)$)
2. 〈小豬 Piggy (TOI 2004 初選 problem 2)〉 TIOJ 1196
   給你一張 $n \times m$ 的地圖，每格會有一個值或是 X 表示不可通行。現在你每次只能往右或往下走，求從最左上走到最右下時路徑上格子總和的最大值。($n, m \leq 1000$)

3. 〈營地〉TIOJ 1097
   給你一個 $m \times n$ 的 $0-1$ 矩陣，找一個最大的正方形使得正方形內都是 0。$(n, m \leq 1000)$

4. 〈經典問題〉
   給你一棵 n 個點的樹，詢問樹上距離小於等於 k 的點對的數量。$(n \leq 10^5, m \leq 10^9)$

5. 〈兔子跳鈴噹〉TIOJ 1019
   給你 n 個鈴鐺的水平位置。你每次可以從第 x 個鈴鐺跳到第 $x+1$ 或第 $x+2$ 個鈴鐺。求從第一個鈴鐺跳到第 n 個鈴鐺所需的最小移動水平距離 $(n \leq 10^5)$

3.4.3 Exercise-經典 DP

1. 〈Longest Common Subsequence(LCS, 最長共同子序列), 經典問題〉
   給你兩個長度分別為 l_a, l_b 的字串 a, b，求他們的最長共同子序列。$(l_a, l_b \leq 1000)$

2. 〈Edit distance(編輯距離), 經典問題〉
   給你兩個長度分別為 l_a, l_b 的字串 a, b，你每次編輯可以刪掉一個字元，插入一個字元或是改變一個字元，問你從 a 到 b 至少要編輯幾次。$(l_a, l_b \leq 1000)$

3. 〈最大連續和, 經典問題〉
   給你一個長度為 n 的整數數列，找一段最大的連續和。$(n \leq 10^6)$

4. 〈背包問題, 經典問題〉
   你有一個最多能承載重量 W 的背包，現在你有 N 個物品分別有重量、價值和數量 w_i, c_i, a_i，選一些物品裝入背包內使總價值最大。$(O(WN \lg(\sum c_i)))$

5. 〈A 遊戲 (96 建中校內資訊能力競賽 problem 6), 經典問題〉TIOJ 1029
   給你一個有 n 個整數的序列，雙方每次可以從左邊或右邊拿走一個數字，最後的分數即是你拿走的數字的總和，問在雙方都使用最佳策略下，先手至少可以贏後手多少分數。

6. 〈最佳順序矩陣乘法, 經典問題〉
   有 N 個矩陣相乘的式子，矩陣乘法是有結合率的，即對於矩陣 A, B, C，$(AB)C = A(BC)$，並且 $k \times m$ 的矩陣乘上 $m \times n$ 的矩陣需要 kn 次乘法，問你在適當的安排乘法順序下，最好需要幾次乘法。$(N \leq 300)$

3.4.4 DP on tree

1. 〈樹分治求重心, 經典問題〉
   給你一顆有 n 個節點的樹，請找出一個點使得去除這個點後，剩下的樹中最大值最小。$(l_a, l_b \leq 10^5)$

2. 〈題目忘了〉TIOJ 1766
   給你一顆有 n 個節點的樹，樹上的邊為無向的且有流量限制，請問你從某一個非葉子節點當作源點，所有葉子當作匯點的最大流的最大值。$(n \leq 10^5)$

3. 〈猴子點技能, 經典問題〉NTUJ 1616
   有 N 種技能，對於一種技能有 C_i, L_i 兩個整數代表花費和喜好程度，而且技能可能有依賴技能 P_i，請找一種總花費不超過 K 元的購買技能方案，你所購買的技能其喜好程度總和最大。($n \leq 10^6$)

3.4.5 需要想一下的 DP

1. 〈N 箱 M 球〉TIOJ 1291
   求將 M 個相同/相異的球放入 N 個相同/相異的箱子的方法數。

2. 〈Oil(Apio 2009)〉
   給你一個 $m \times n$ 的 $0-1$ 矩陣，找 3 個沒有交集且邊長為 k 的正方形使得 3 個正方形內的數字總和最大。($k \leq n, m \leq 2000$)

3. 〈???〉codeforces ??
   給你一張 $n \times m$ 的地圖，每個格子會有一個整數 (可能為負)。現在有兩個人從左上角出發，每次只能往右或往下，最後到達右下角，求被至少一個人走過的格子裡的數字總和的最大值。($n, m \leq 300$)

3.4.6 狀態壓縮 DP

有時候也必須仔細思考一個好的方法將每個問題對映到狀態。

1. 〈漢米頓路徑, 經典問題〉TIOJ 1291
   求一個圖上 (權值最大) 的漢米頓路徑 (圈)。

2. 〈打地鼠 (95 建中校內培訓模擬試題)〉TIOJ 1014
   有 n 個從左而右排成一列的地鼠洞，相鄰兩個洞距離皆為 1。在每個洞各有一出現周期固定的地鼠，且地鼠被打一次之後就不會再出現。你在第一個洞的左邊距離 1 處，你每秒可走 1 的距離。現在給你每個地鼠的周期，求打完所有地鼠的最少需花時間。($n \leq 15$)

4 數學方法

4.1 數論

4.1.1 質數

定義

若一個正整數只有 1 和自己兩個正因數，那麼稱這個數為質數。

質數檢查

一個合數 N 不可能沒有 $\leq \sqrt{N}$ 的因數。將所有 $1 \sim \sqrt{N}$ 的質數一個一個拿來除以 N，若皆不能整除，代表 N 為質數。
時間複雜度：$O(\sqrt{N})$

質數篩法

目的：判斷 $1 \sim N$ 中有哪些質數。

A___A 算法　把 $1 \sim N$ 的每一個數，都用上述方法判斷是否為質數。
時間複雜度：$O(N\sqrt{N})$

快速篩法　先建立一個 $1 \sim N$ 的表，記錄每個數有沒有可能是質數。
首先把 1 刪掉。接著找出最小的還沒刪的數 (目前來說是 2)，把所有 2 的倍數都刪掉。此時再找最小還沒刪的數 (3)，把所有 3 的倍數都刪掉，再找最小還沒刪的數 (5)…這樣下去，刪到 $\sqrt{N}$ 為止。餘下的所有還沒被刪的數皆為質數。
優化：事實上選 k 的倍數來刪的時候，k^2 以下的數，必定有其他更小的質因數，所以早就被刪光了，所以從 k^2 開始往上刪 k 的倍數就行了。
時間複雜度：$O(N \lg \lg N) \approx O(N)$

質因數分解

將一正整數 N 分解成多個相異質數的冪次相乘的形式：

$$N = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

算法 先建立 $1 \sim \sqrt{N}$ 的質數表，由最小的開始除 N，直到除不盡時就換一個。所有質數都除完後，若 N 還沒被除成 1，而還有剩下一個數，則那個數也是一個質因數。(若 N 很小，為了方便，可以直接把所有 $\sqrt{N}$ 以下的數都拿來除)
時間複雜度：$O(\sqrt{N})$

正因數

若 N 的質因數分解為：

$$N = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$$

則其正因數個數為

$$(e_1 + 1)(e_2 + 1) \cdots (e_k + 1)$$

而正因數總和為

$$(1 + p_1 + \cdots + p_1^{e_1})(1 + p_2 + \cdots + p_2^{e_2}) \cdots (1 + p_k + \cdots + p_k^{e_k})$$

質數的性質

對於任一質數 p：

1. $(p-1)! \equiv -1 \pmod{p}$
2. (費馬小定理)$a^{p-1} \equiv 1 \pmod{p}$
3. 對於質數 $p > 2$，$1 \sim p$ 的整數平方中，只有 $1^2, (p-1)^2 \equiv 1 \pmod{p}$。取一整數 x 的平方，計算 $x^2 \mod p$，可判斷 p 是否為質數 (有時會有例外！所以要取多一點 x 值取得錯誤的機率減小)。

4.1.2 輾轉相除法

目的

求出兩正整數 a、b 的最大公因數 $\gcd(a, b)$。

原理

給定兩個正整數 a、b $(a \geq b)$，令 r 為 a 除以 b 的餘數，則 $a = kb + r$，其中 k 為整數。則 $\gcd(a, b) = \gcd(r, b)$。如此可以再將 b、r 代替原本的 a、b 再運算，其最大公因數也不會改變。如此循環到最後，若其中一數變成 0，則另一數則為原欲求的最大公因數 (因為 $\gcd(a, 0) = a$)。

最小公倍數可利用以下性質求得：$\operatorname{lcm}(a, b) \gcd(a, b) = ab$。
時間複雜度：$O(\lg(\max(a, b)))$

Algorithm 11 GCD

```
1: function GCD(a, b)
2:     if a < b then
3:         Swap(a, b)
4:     end if
5:     if b = 0 then
6:         return a
7:     end if
8:     r ← a mod b
9:     return GCD(b, r)
10: end function
```

二元一次方程式

輾轉相除法也能用來求方程式 $ax + by = \gcd(a, b)$ 的整數解。

算法： 將 $1a + 0b = a$ 與 $0a + 1b = b$ 兩個元素作輾轉相除法，到最後等式右邊會變成 $gcd(a, b)$，等式左邊的係數即為 x、y 的解。

Algorithm 12 Solve Linear Equation

```
1: function IntSolution(a, b)
2:     if a = b then
3:         return (0, 1)
4:     end if
5:     (x, y) ← IntSolution(b, a mod b)
6:     t ← a/b
7:     return (y, x − ty)
8: end function
```

時間複雜度：$O(\lg(\max(a, b)))$

4.1.3 同餘

定義

若整數 a、b 符合 $a = b + km$(其中 k 為整數，m 為正整數)，即 a 與 b 除以 m 的餘數相同，則稱 $a \equiv b \pmod{m}$。

模運算

1. $a \equiv b \pmod{m}, r \equiv d \pmod{m} \Longrightarrow a \pm b \equiv r \pm d \pmod{m}$

2. $a \equiv b \pmod{m}, r \equiv d \pmod{m} \Longrightarrow ar \equiv bd \pmod{m}$

3. $a \equiv b \pmod{m} \Longrightarrow a^k \equiv b^k \pmod{m}$

模逆元

若整數 x 滿足 $ax \equiv 1 \pmod{m}$，則稱 x 為 a 模 m 的模逆元，記作 $x = a^{-1}$。
若 a 與 m 互質，則：

$$ab \equiv r \pmod{m} \Longrightarrow b \equiv ra^{-1} \pmod{m}$$

模逆元計算方法

1. 以輾轉相除法解 $ax + my = 1$，得到 $ax \equiv 1 \pmod{m}$，即 $x = a^{-1}$。
   時間複雜度：$O(\lg m)$

2. 若 p 為質數，由費馬小定理 $a^{p-1} \equiv 1 \pmod{p} \Rightarrow a^{-1} \equiv a^{p-2} \pmod{p}$。
   時間複雜度：$O(\lg p)$

快速冪

上面提到模逆元的第 2 種計算方式，需要計算 a^{p-2}，如果 p 很大的時候，計算將會非常耗時。有一種簡化計算方式：
要計算 $a^x \pmod{p}$，先把 $a, a^2, a^4, a^8, \cdots, a^{\lfloor \lg x \rfloor} \pmod{p}$ 計算出來 (一直平方即可)，然後將 x 表示成相應的二進位值，以剛才計算好的次方湊出 x。
時間複雜度：$O(\lg x)$
Ex: 計算 $3^{13} \pmod{11}$
先算出 $3^1 \equiv 3, 3^2 \equiv 9, 3^4 \equiv 4, 3^8 \equiv 5 \pmod{11}$，以及 $13 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1$。
所以 $3^{13} = 3^8 \times 3^4 \times 3^1 \equiv 5 \times 4 \times 3 \equiv 5 \pmod{11}$。

同餘的應用

在資訊的題目中，經常出現答案非常大的狀況 (Ex: 10^{10^5} 的數量級)，不可能將數字直接表示出來，此時題目多會要求你輸出答案 $\bmod p$ 的餘數 (此處 p 通常是一個很大的質數，常為 1000000007)。這時候就需要運用模運算的四則運算，取代整數的四則運算。
Ex: 計算 $123456! \pmod{1000000007}$。

4.1.4 Exercise

1. 分組編隊 < 建國中學 99 年校內培訓 contest #1 prob 3> [TIOJ 1668]
   求在 $[L, R]$ 中有幾個數是質數 $(1 \le L \le R \le 2^{31} - 1, R - L \le 2 \times 10^5)$

Algorithm 13 Fast Exp. (a^b) In Modula p, And Mod Inverse ($a^{-1} \mod p$)

```
1: function FastExp(a, b, p)
2:     t ← a
3:     W ← 1
4:     while b > 0 do
5:         if b&1 then
6:             W ← (W × t) mod p
7:         end if
8:         b ← b >> 1
9:         t ← t^2 mod p
10:    end while
11:    return W
12: end function
13: function ModRev(a, p)
14:    return FastExp(a, p − 2, p)
15: end function
```

2. 完全子圖 < NPSC2007 初賽 (prob B)> [TIOJ 1459]
   對所有 $1 \le k \le N$，已知 $x \equiv f_k \pmod{w_k}$ 的解，求 x 通解或是判斷無解。

3. 互質任務 < NPSC2005 初賽 (prob C) > [TIOJ 1069]
   有 N 個人站成一列，每個人有一個 $0 \sim 9$ 的數字，最多能挑多少人依原本順序站成一排形成一個新的數且和 M 互質。($N \le 10^3, M \le 10^4$)

4. 橫掃射擊場 <TIOJ 1514 Skyly & Shik Contest #1 pD.>
   你在 $(0, 0)$，給定正方形的大小，有多少個看不到的點? 兩個點看得到的條件是它們之間沒有其他點。如左圖就看得到。

4.2 矩陣

4.2.1 n 元一次方程組

$n = 2$:

$$\begin{cases} ax + by &= c \\ dx + ey &= f \end{cases}$$

1. 代入消去法
2. 兩式相減求解
3. 公式解：$x = \dfrac{cd - bf}{ad - be}, y = \dfrac{af - ec}{ad - be}$

一般情況：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{cases}$$

要如何解出 $x_1, x_2, \cdots, x_n$?

1. 代入消去法 (變數太多了，Q___Q)
2. 公式解 (有，可是複雜度很差)
3. 兩式相減求解 (高斯消去法)

4.2.2 高斯消去法

先從第一條方程式開始消去，將其他方程式的第一個變數都減成 0，再換第二條方程式往下消去…以此類推。必要時可交換方程式的順序。

可用操作：

1. 將一條方程式，乘以一個係數，在另一條方程式中減去
2. 將一條方程式乘以一個係數
3. 將兩條方程式交換

時間複雜度：$O(n^3)$

※ 注意浮點數誤差！

$$(\text{原方程組})\begin{cases} 3x + 2y + z = 2 & (1) \\ 2x - 3z = -7 & (2) \\ -x + y + 2z = 3 & (3) \end{cases}$$

$$\Downarrow$$

$$(\text{向下消去})\begin{cases} 3x + 2y + z = 2 & (1) \\ -\frac{4}{3}y - \frac{11}{3}z = -\frac{25}{3} & (4) = (2) - \frac{2}{3} \times (1) \\ \frac{5}{3}y + \frac{7}{3}z = \frac{11}{3} & (5) = (3) + \frac{1}{3} \times (1) \end{cases}$$

$$\Downarrow$$

$$(\text{三角化})\begin{cases} 3x + 2y + z = 2 & (1) \\ -\frac{4}{3}y - \frac{11}{3}z = -\frac{25}{3} & (4) \\ -\frac{27}{12}z = -\frac{81}{12} & (6) = (5) + \frac{5}{4} \times (4) \end{cases}$$

$$\Downarrow$$

$$(\text{調整係數})\begin{cases} 3x + 2y + z = 2 & (1) \\ 4y + 11z = 25 & (7) = -3 \times (4) \\ z = 3 & (8) = -\frac{12}{27} \times (6) \end{cases}$$

$$\Downarrow$$

$$(\text{向上代入})\begin{cases} 3x + 2y = -1 & (9) = (1) - (8) \\ 4y = -8 & (10) = (7) - 11 \times (8) \\ z = 3 & (8) = -\frac{12}{27} \times (6) \end{cases}$$

$$\Downarrow$$

$$(\text{向上代入})\begin{cases} 3x = 3 & (11) = (9) - \frac{1}{2} \times (10) \\ 4y = -8 & (10) \\ z = 3 & (8) \end{cases}$$

$$\Downarrow$$

$$(\text{完成})\begin{cases} x = 1 & (12) = \frac{1}{3} \times (11) \\ y = -2 & (13) = \frac{1}{4} \times (10) \\ z = 3 & (8) \end{cases}$$

解的狀況：

1. 消去完後結果包含 $0 \neq 0$：無解
2. 消去完後結果包含 $0 = 0$，並且沒有出現 $0 \neq 0$：無限多組解
3. 消去完後，左式皆不為 0：唯一解

4.2.3 矩陣

線性組合與線性變換

二元一次聯立方程組

$$\begin{cases} 2x+y &= 3 \\ 3x-2y &= -1 \end{cases}$$

可以用另一種方式來理解：

方程式的左邊，我們把 x、y、這兩個變數，做了一些「加加乘乘」的動作，變成了 $2x+y$、$3x-2y$。而我們現在希望能找出一組 x、y，使得他們經過「第一種操作」之後會變成 3，經過「第二種操作」之後會變成 -1。

仔細觀察之後，可以發現這些「加加乘乘」的動作，可以更明確地表達為：將 x、y 兩個變數，分別乘上不同的係數之後，再相加。以第二條方程式來說，先將 x 乘以 3、y 乘以 -2，再加起來變成 $3x-2y$。這種動作稱為 x、y 的「線性組合」。

回到原本的方程組，可以發現方程組左邊所做的事其實是：
將 x、y 兩個變數，用兩種不同的線性組合方式，製造出新的兩個變數 x'、y'。這種使用 x、y 的多種「線性組合」，將 $(x,y) \to (x',y')$ 的操作，稱為「線性變換」。

於是原本的方程組可以解釋成：請找出適當的 (x,y)，使得他們在經過給定的「線性變換」之後，會變成 $(3,-1)$。

矩陣表示法

為了方便起見，我們把這個「線性變換」和「被變換的數組」分開來寫：

$$\begin{bmatrix} 2 & 1 \\ 3 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

最左邊 2×2 的長方形稱為「矩陣」，代表所給定的「線性變換」；而 2×1 的則稱為「數組」(事實上正式的名稱為「向量」，但為了避免與後面計算幾何的向量混淆，先不採用這個名稱)。

也可以簡寫成：

$$Av = b$$

其中 A 為矩陣，$v=(x,y)$，$b=(3,-1)$。

在這種形式下，看起來很像「A 乘以 v 等於 b」。事實上一個矩陣「乘以」一個數組，即是將一個線性變換「操作」在一個數組上。而這就是矩陣乘法定義的來源：

$$\begin{bmatrix} \mathbf{2} & \mathbf{1} \\ 3 & -2 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{2}\times\mathbf{x}+\mathbf{1}\times\mathbf{y} \\ 3\times x+(-2)\times y \end{bmatrix}$$

這種表示方式會帶來許多計算上的方便。

定義及操作

1. 一個 $m \times n$ 的矩陣，有 m 橫排以及 n 直列：

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

2. 兩個相同大小的矩陣可相加 (減)：每個元素直接相加 (減)

3. 一個矩陣乘以一個係數 r：每個元素都乘以係數 r

4. 矩陣乘數組：根據前面線性變換的定義，一個矩陣乘以一個數組，會經過線性變換，得到另一個數組

   新數組的第 i 個元素，等於矩陣第 i 橫排與被乘數組，各元素的相應乘積的和

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_n \end{bmatrix}$$

   時間複雜度：$O(n^2)$

5. 單位矩陣 I：對任何數組 v，符合 $Iv = v$，也就是 I 是一個「操作完還是自己」的線性變換。根據乘法定義：

$$I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

6. 矩陣乘矩陣：

   假設有一個數組 v，經過一個線性變換 A 之後變成 v'，而 v' 再經過另一個線性變換 B 之後變成 v''，是否存在一個大的線性變換 C，直接將 v 變成 v''？

   以矩陣形式來表示，$v' = Av, v'' = Bv'$，所以 $v'' = B(Av) = Cv$。這時定義 $C = BA$，即矩陣乘矩陣的乘法。

   規則跟矩陣乘數組時相似，新矩陣 C 的第 i 橫排第 j 直列的元素，等於 A 的第 i 橫排整排、和 B 的第 j 直列整列，元素兩兩相乘後的總和

$$\begin{bmatrix} & \cdots & \\ a_{i1} & a_{i2} \cdots & a_{in} \\ & \cdots & \end{bmatrix} \begin{bmatrix} & b_{1j} & \\ & b_{2j} & \\ \vdots & \vdots & \vdots \\ & b_{nj} & \end{bmatrix} = \begin{bmatrix} & \vdots & \\ \cdots & c_{ij} & \cdots \\ & \vdots & \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

例如：

$$\begin{bmatrix} 2 & 3 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & -3 \end{bmatrix} = \begin{bmatrix} 2\times 1+3\times 2 & 2\times 2+3\times(-3) \\ 3\times 1+(-1)\times 2 & 3\times 2+(-1)\times(-3) \end{bmatrix} = \begin{bmatrix} 8 & -5 \\ 1 & 9 \end{bmatrix}$$

時間複雜度：$O(n^3)$

另外，由於單位矩陣代表「沒有變換」，因此 I 乘以任何矩陣，無論先乘後乘，都會得到自己 ($IA = A$，$AI = A$)。

7. 矩陣乘法具有結合律：根據以上的定義，$v'' = B(Av) = Cv = (BA)v$，也就是說，矩陣乘法的前後括號順序可以任意地決定。

8. 矩陣乘法不具有交換律：在某些情況下，先套用 A 變換再套用 B 變換，以及先套用 B 變換再套用 A 變換，所得到的結果是不同的！也就是說，$AB \neq BA$。

9. 單向分配律：$A(B+C) = AB + AC$，$(A+B)C = AC + BC$ (注意乘的先後順序不能變！)

反矩陣

線性方程組

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{cases}$$

可以化成矩陣型式：

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

在右邊可以乘上一個單位矩陣 I(反正不影響)：

$$Ax = Ib$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

套用高斯消去法，將左邊的矩陣 A 三角化，再變成對角矩陣，最後變成單位矩陣 I。在消去的過程中，同時對右邊的矩陣做一樣的事情，最後右邊的單位矩陣 I 會被消去成另一個矩陣 A^{-1}。

$$Ix = A^{-1}b$$

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}^{-1} & a_{12}^{-1} & \cdots & a_{1n}^{-1} \\ a_{21}^{-1} & a_{22}^{-1} & \cdots & a_{2n}^{-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{-1} & a_{n2}^{-1} & \cdots & a_{nn}^{-1} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

但 $Ix = x$，所以我們得到 $x = A^{-1}b$，也就是說解出 x 了！

這個新產生的矩陣 A^{-1} 稱為 A 的反矩陣，也就是說 $AA^{-1} = A^{-1}A = I$。從線性變換的觀點來說，一個數組 v 經過 A 變換成為 v'，而 v' 再經過反變換 A^{-1} 之後會回到 v。

因此我們可以得到：$Ax = b \to A^{-1}Ax = A^{-1}b \to x = A^{-1}b$。

這種方法與前面所說的高斯消去法解聯立方程組，其實是同樣的意思。所以時間複雜度同樣為 $O(n^3)$。

矩陣的冪次

將一個變換 A 連續套用到 v 上 k 次，寫成矩陣乘法型式就是：

$$v' = AAAA \cdots Av$$

由於矩陣乘法具有結合律，因此我們可以任意決定先乘什麼，而最後答案都會一樣。因此可以定義矩陣冪次：

$$v' = AAAA \cdots Av = A^k v$$

$$AAAA \cdots A = A^k$$

計算矩陣冪次最簡單的方法，即是照定義一個一個乘，將 A 自乘 k 次就得到 A^k。時間複雜度：$O(n^3k)$

事實上矩陣的冪次與整數的次方一樣，可以使用快速冪的技巧加速：先計算出 $A, A^2, A^4, A^8, \cdots$，再湊出所需要的 A^k，這樣只需要進行 $O(\lg k)$ 次的乘法，總時間複雜度為 $O(n^3 \lg k)$。

4.2.4 Exercise

1. 幼稚國王的獎賞 [TIOJ 1094] <NPSC2006 初賽 (prob C)>
   給你 N 個數 x_i，問你能 XOR 出最大的值 ($N \leq 1000, x_i \leq 10^{18}$)

2. <UVa 10689>
   $f(0) = a, f(1) = b, f(n) = f(n-1) + f(n-2)$，求 $f(n)$ 的末 m 位數
   ($0 \leq a, b \leq 100, 0 \leq n \leq 10^9, 1 \leq m \leq 4$,10000 筆測資)

3. 熱鍋上的螞蟻 [TIOJ 1136] <96 TWN Practice Contest 1>
   有 N 個鍋子，兩個鍋子上可能有筷子相通，有隻螞蟻從某個起點開始爬，他爬到與某個鍋子相鄰的所有鍋子機率相等，問 K 步後處在某個鍋子的機率 ($N \leq 100, K \leq 10^9$)

4. 城市旅遊問題 [TIOJ 1778] <99 建中校內資訊能力競賽 (prob 3)>
   給你 V 個點和 M 條邊 (可能有重邊)，問你從 S 點走 K 條路恰好到達 T 點的方法數有幾種，邊可以重複走來走去都不會仆街。($N \leq 100, M \leq 50000, K \leq 10^{15}$)

4.3 組合

4.3.1 基礎組合

1. 加法原理：兩個互相獨立 (互不影響) 的選擇，只能選其中一邊中的一種，其總方法數為兩者相加。
2. 乘法原理：兩個互相獨立 (互不影響) 的選擇，兩邊各要選一種，其總方法數為兩者相乘。
3. 階乘 $n! = 1 \times 2 \times 3 \times \cdots \times n$：(同時也是 m 個相異物排列成不同順序的方法數)。
4. 排列數 $P_n^m = \dfrac{m!}{(m-n)!}$：$m$ 個相異物，選其中 n 個來排 (順序有差) 的方法數。
5. 組合數 $C_n^m = \dfrac{m!}{n!(m-n)!}$：$m$ 個相異物，選其中 n 個出來 (順序不計) 的方法數，同時也是 n 個 A 物和 $m-n$ 個 B 物排成一列 (順序有差) 的方法數。
6. 重覆組合 $H_n^m = C_n^{m-n+1}$：m 種不同的東西無限供應，總共選 n 個出來 (可以重覆，順序不計) 的方法數。同時也是方程式 $x_1 + x_2 + \cdots + x_m = n$ 的非負整數解組數。
7. 二項式定理：$(x+y)^n = \sum_{k=0}^{n} C_k^n x^{n-k} y^k$，即二項式展開的係數為組合數。
8. 巴斯卡定理：$C_n^m = C_n^{m-1} + C_{n-1}^{m-1}$。
9. 組合數經常會超出 `long long` 範圍，因此需要配合同餘運算。

4.3.2 遞迴式

定義

將數列 (或函數) 的任何一項以前一 (幾) 項表示的方法。

例：數列 $\{a_n\}, a_i = a_{i-1} + a_{i-2}$，即代表這個數列中每一項皆為其前兩項的和。

初始條件

若要以遞迴式確定其數列 (或函數)，經常需要幾個初始條件 (數量通常等於為遞迴式的階數)。

例：上述數列若加上條件 $a_0 = 0, a_1 = 1$，即可確定其整個數列：

$a_0 = 0, a_1 = 1, a_2 = 1, a_3 = 2, a_4 = 3, a_5 = 5, a_6 = 8, \cdots$ (費氏數列)

計算方法

- DP
- 矩陣法 (線性遞迴式適用，參考：數學方法 I)
- 解出一般式 (不一定適用於所有遞迴式，且計算過程和結果可能非常複雜)

4.3.3 卡特蘭數

定義

卡特蘭數 C_n：在 $n \times n$ 的方格中，一次只能往右或往上走一格，從左下角 $(0,0)$ 走到右上角 (n,n)，且不能越過對角線的方法數。

$$C_n = \{1, 1, 2, 5, 14, 42, 132, \cdots\}$$

遞迴式

設第一次碰到對角線時的座標是 (i,i)，則可分割為兩個部份：

1. $0 \sim i-1$ (還沒碰到對角線之前的部份)：
   相當於從 $(1,0)$ 走到 $(i,i-1)$，不越過對角線的走法數，也就是卡特蘭數 C_{i-1}。
2. $i \sim n$ (已經碰過對角線的部份)：
   相當於從 (i,i) 走到 (n,n)，不越過對角線的走法數，也就是卡特蘭數 C_{n-i}。

由於上面兩部份互不相干，所以這兩個方法數以乘法原理連結：於 (i,i) 第一次碰到對角線的方法數為 $C_{i-1}C_{n-i}$。

所以總方法數為：

$$C_n = \sum_{i=1}^{n} C_{i-1}C_{n-i}$$

或

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$$

此即為卡特蘭數的遞迴式。單次遞迴時間複雜度為 $O(n)$；以 DP 法計算 C_n，總時間複雜度為 $O(n^2)$。

一般式

事實上，卡特蘭數可以寫成一般式：(暫時不證明)

$$C_n = \frac{C_n^{2n}}{n+1}$$

計算時間複雜度為 $O(n)$。

應用

以下各種情況的方法數皆為卡特蘭數 C_n

1. 邊長為 n 的正方形不跨越對角線的捷徑走法數。
2. n 個節點可產生的二元樹個數。
3. 在正 n 邊形中，以三角形分割多邊形的方法數。
4. n 對括號可產生的合法匹配數。(合法運算式)

廣義卡特蘭數

定義廣義卡特蘭數為

$$C_{k,n} = \frac{C_n^{kn}}{(k-1)n+1}$$

此數等於有 n 個節點的 k 元樹個數。

4.3.4 Exercise

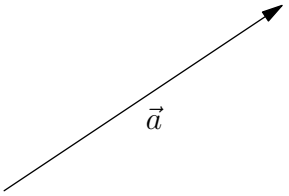
1. 山稜線 <TIOJ 1607>
   對 T 筆詢問，輸出長度為 $2N$ 的山稜線的種類數 mod 1000000007 的值。
2. 燈泡問題 <TIOJ 1770>
   問有幾種方法點燈泡 M 秒鐘且沒有超過連續 N 秒燈是亮的
3. 放錯的信封 < 94 建中校內資訊能力競賽 > <TIOJ 1086>
   問你 N 個人都拿到不該拿的東西的方法有幾種。
   輸出末 8 位數：$O(N)$；輸出前 8 位數：$O(1)$
4. 海藻 < 98 北市賽 (prob 1) > <TIOJ 1677>
   有一個神奇觸手，一開始是 (綠) 的，接者每天 (綠) 會變成 (黃)，而 (黃) 會變成 (綠黃)，求第 N 天由左往右第 K 個觸手是甚麼顏色

4.4 計算幾何

4.4.1 向量

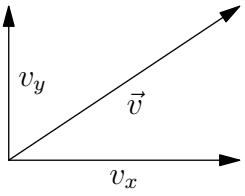
定義

在二維 (或三維) 空間中，一「有大小」、「有方向」的量。(純量則只有大小)
常將向量上方加一個箭頭 (例如：$\vec{a}$)，以便辨認。



二維平面上的向量，可以分解成直角座標的 x、y 分量：

$$\vec{v} = (v_x, v_y)$$

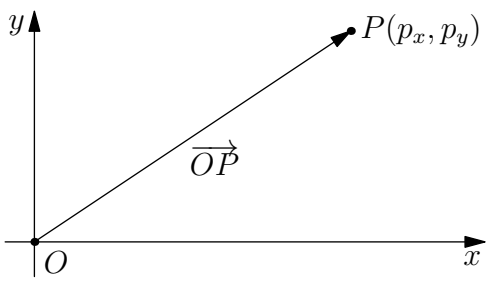


座標平面上兩個點 $P(p_x, p_y)$ 和 $Q(q_x, q_y)$，定義向量 $\overrightarrow{PQ}$ 為由 P 點指向 Q 點的向量：

$$\overrightarrow{PQ} = (q_x - p_x, q_y - p_y)$$

注意到，平面上的每個點 $P(p_x, p_y)$，可以用從原點 $O(0,0)$ 指向 P 的向量來代表。因此

$$\vec{P} = \overrightarrow{OP} = (p_x, p_y)$$



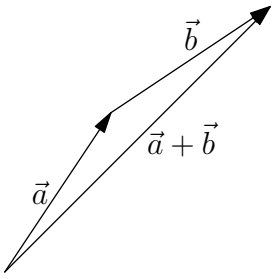
將平面上的點以向量型式表示，有利於計算 (向量有很多好用的運算和性質)

基本運算

1. 加減法：頭尾相連形成新向量。以分量形式計算的話，直接將各分量相加/減。

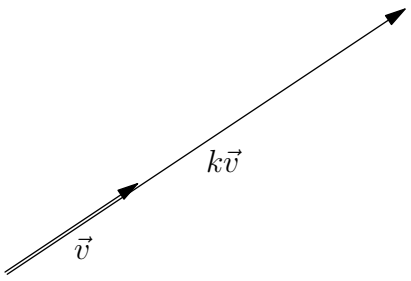
$$\vec{a} = (a_x, a_y), \vec{b} = (b_x, b_y)$$

$$\vec{a} \pm \vec{b} = (a_x \pm b_x, a_y \pm b_y)$$



2. 係數乘積：將一向量乘以 k 倍，相當於把向量延長 k 倍 (若 $k < 0$，則向反向延伸 $|k|$ 倍)。分量形式計算的話，將各分量分別乘以 k。

$$k\vec{v} = (kv_x, kv_y)$$



3. 絕對值：就是向量的長度。向量 $\vec{v}$ 的絕對值記為 $|\vec{v}|$，以分量形式計算的話是

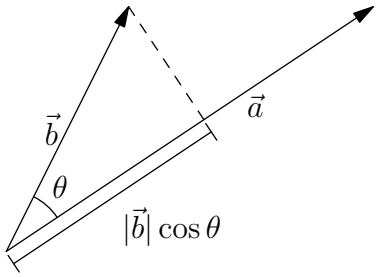
$$|\vec{v}| = \sqrt{v_x^2 + v_y^2}$$

4. 內積：向量 $\vec{a}$ 與向量 $\vec{b}$ 的內積，記為 $\vec{a} \cdot \vec{b}$。內積的結果為一純量，等於 $\vec{a}$ 的長度，乘以「$\vec{b}$ 在 $\vec{a}$ 方向的投影」的長度。分量形式計算的話，將 x、y 分量分別相乘後，再相加。

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y$$

另外，如果 $\vec{a}$ 和 $\vec{b}$ 的夾角為 θ，則內積也可以表示成：

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$$



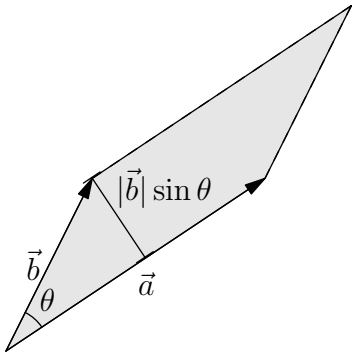
因此，自己跟自己的內積是長度的平方：

$$\vec{a}\cdot\vec{a}=|\vec{a}|^2$$

由 $\cos\theta$ 的正負號可知，當兩向量平行時，內積最大 (等於兩向量長度乘積)。當兩向量夾銳角時，內積為正；夾鈍角時，內積為負；兩向量垂直時，內積為 0(通常判斷垂直，即是判斷兩向量內積是否為 0)。

5. 外積：相當於兩個向量圍出的平行四邊形面積，以 $\vec{a}\times\vec{b}$ 表示。若由 $\vec{a}$ 轉到 $\vec{b}$ 的夾角為 θ(右手定則：逆時針為正，順時針為負)，則其外積記為：

$$\vec{a}\times\vec{b}=|\vec{a}||\vec{b}|\sin\theta=\begin{vmatrix}a_x & a_y\\ b_x & b_y\end{vmatrix}=a_xb_y-b_xa_y$$

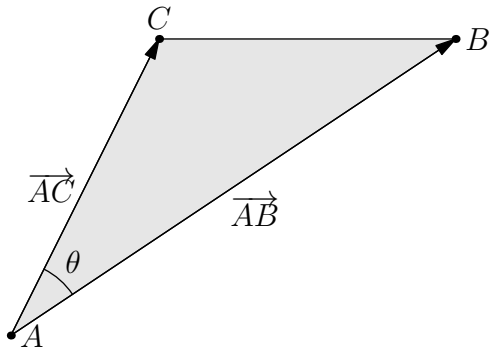


當兩向量平行時，外積為 0；兩向量垂直時，外積的絕對值最大。
另外若 $\vec{b}$ 在 $\vec{a}$ 的逆時針方向 (即左邊)，那麼 $\vec{a}\times\vec{b}>0$。外積的正負號，可以用來判斷兩向量之類的順逆時針關係。

另外，根據定義，平面上三點 A,B,C 所圍成的三角形面積為

$$\frac{1}{2}|\overrightarrow{AB}\times\overrightarrow{AC}|$$

※ 事實上，在三維空間中，兩向量的外積會得到另一個與原本兩個向量皆垂直的向量，並且計算更為複雜，有興趣的同學可以上網查一下資料。



6. 向量的旋轉：一個向量 $\vec{a}=(a_x,a_y)$，逆時針旋轉 θ 角度，會得到一個新的向量 $\vec{a'}$：

$$\vec{a'}=(a'_x,a'_y)=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}a_x\\ a_y\end{bmatrix}=(a_x\cos\theta-a_y\sin\theta,a_x\sin\theta+a_y\cos\theta)$$

可以驗證，新的向量長度依然不變：

$$|\vec{a'}|=\sqrt{(a_x\cos\theta-a_y\sin\theta)^2+(a_x\sin\theta+a_y\cos\theta)^2}=\sqrt{a_x^2+a_y^2}=|\vec{a}|$$

浮點數誤差 (EPS)

在平面幾何的運算中，並沒有什麼理由限制座標必需為整數，因此常常牽涉到浮點數的運算。

浮點數一般使用 `double` 或 `long double` 型別來儲存，其精確度有一定的上限值。因此不可避免地，在浮點數的四則運算當中，會產生一些誤差。例如電腦可能計算 $1.0+1.0=1.9999999999999999$，而這時電腦會認為 $1.0+1.0\neq 2.0$，導致程式出現問題。

因此在比對兩數值是否相等時，應容許一些很小的誤差。例如令 $\mathrm{EPS}=10^{-10}$，而兩數差的絕對值 $\leq \mathrm{EPS}$ 時，判斷兩數相等，如此即可解決大部份誤差問題。

4.4.2 極座標

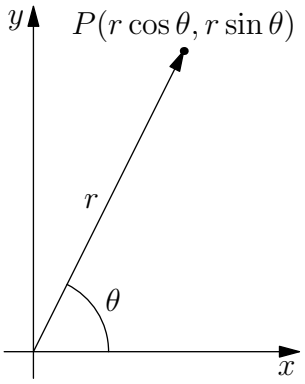
給定一個點作為原點 O，則平面上所有點的位置，除了可以用直角座標 (x,y) 來表示，也可以用極座標 (r,θ) 來表示。其中 r 代表與原點的距離 (也就是從原點到那點的向量長度)；θ 代表與 $+x$ 軸的夾角 (逆時針為正)。因此有以下關係：

$$\begin{array}{rcl} r & = & \sqrt{x^2+y^2} \\ \theta & = & \tan^{-1}\left(\frac{y}{x}\right)\end{array}\quad , \quad \begin{array}{l} x=r\cos\theta \\ y=r\sin\theta\end{array}$$

有些時候會需要以 θ 來排序，以使點照逆時針的順序排好 (因 $\tan^{-1}$ 函數無法分別第一和第三象限的點，因此在做環狀排序的時候，先把左右 (或上下) 兩邊排好，再合併)。另外也可以用外積的正負號作為比較函式來排序。

4.4.3 線段相交

給定兩線段 $\overline{P_1P_2},\overline{Q_1Q_2}$ 四個端點 P_1,P_2,Q_1,Q_2 的座標，判斷這兩線段是否相交。

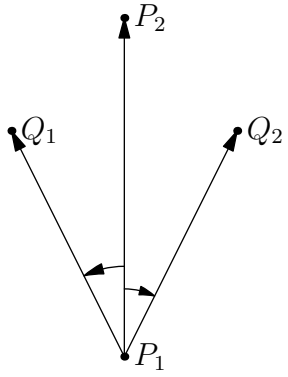


1. 可直接求出兩直線方程式，找出交點，再判斷是否在兩線段範圍內。(誤差大，麻煩)

2. 跨立檢驗：若 P_1, P_2 在 $\overline{Q_1Q_2}$ 異側，且 Q_1, Q_2 在 $\overline{P_1P_2}$ 異側，則此兩線段相交。
   即判斷下列兩式是否皆成立：

$$(\overrightarrow{P_1P_2} \times \overrightarrow{P_1Q_1})(\overrightarrow{P_1P_2} \times \overrightarrow{P_1Q_2}) < 0$$

$$(\overrightarrow{Q_1Q_2} \times \overrightarrow{Q_1P_1})(\overrightarrow{Q_1Q_2} \times \overrightarrow{Q_1P_2}) < 0$$



至於判斷交點可以求出面積比，再用分點公式，比直接解方程式稍微精確一點。

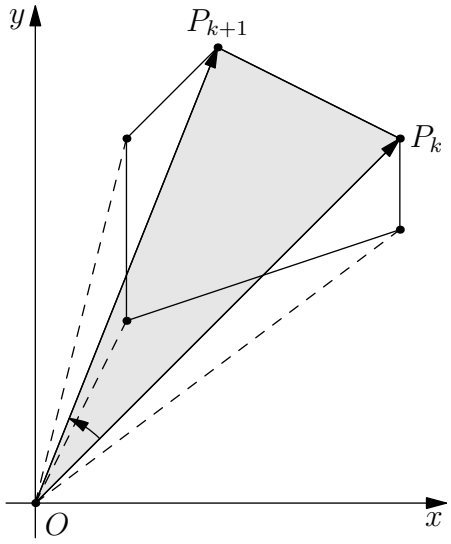
4.4.4 多邊形面積

給定一無邊相交的多邊形 $P_1P_2P_3\cdots P_n$，可計算此多邊形面積：(為了方便，令 $P_{n+1} = P_1$)

$$\frac{1}{2}\left|\sum_{k=1}^{n} \overrightarrow{OP_k} \times \overrightarrow{OP_{k+1}}\right| = \frac{1}{2}\left|\sum_{k=1}^{n} \left(P_{kx}P_{(k+1)y} - P_{ky}P_{(k+1)x}\right)\right|$$

將四邊形分割成許多三角形 (面積可為負) 即可導出此式。

時間複雜度：$O(n)$



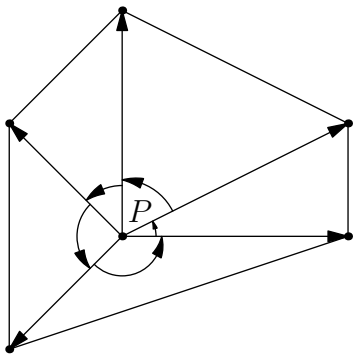
4.4.5 判斷點在多邊形之內外

給定一個點 A 和一個多邊形 $P_1P_2P_3\cdots P_n$，判斷 A 是否在多邊形內。

1. 凸多邊形：若 A 在多邊形內，則多邊形的順序正好是由 A 為參考點的極角排序。因此，所有的

$$\overrightarrow{AP_k} \times \overrightarrow{AP_{k+1}}$$

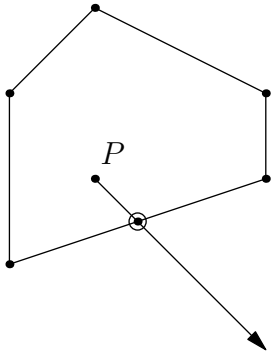
皆同號。這些外積中，若有異號代表 A 在凸多邊形外；若有一個 0，代表 A 在某一條邊上；若有兩個 0，代表 A 在某一頂點上。



時間複雜度：$O(n)$

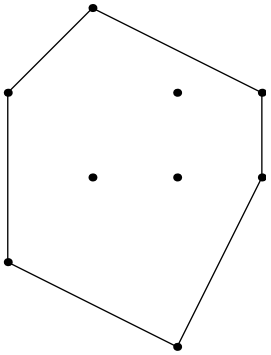
2. 任意邊不相交的多邊形：由 A 點往外作射線，若此射線交多邊形的邊奇數次，則 A 在多邊形內若交偶數次，則 A 在多邊形外 (需先判掉 A 在頂點或邊上的狀況)。射線方向儘量隨機，以避免剛好交到頂點等例外狀況。

時間複雜度：$O(n)$ (若為凸多邊形，可以用二分搜之類的加速)



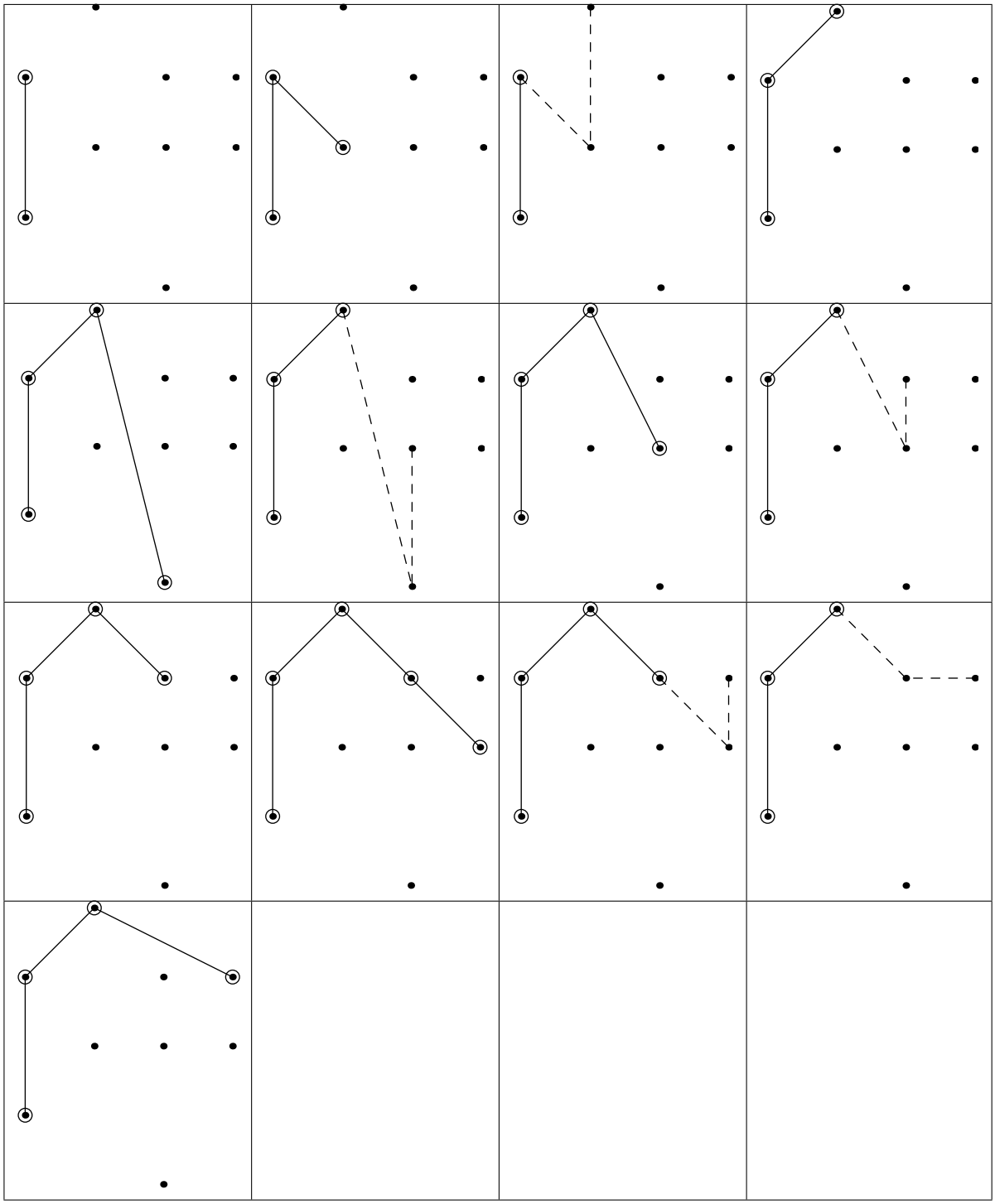
4.4.6 凸包

對於一個點集，找出其中一些點，使得點集中的所有點，都在由這些選中的點所圍成的凸多邊形之內或邊界上。



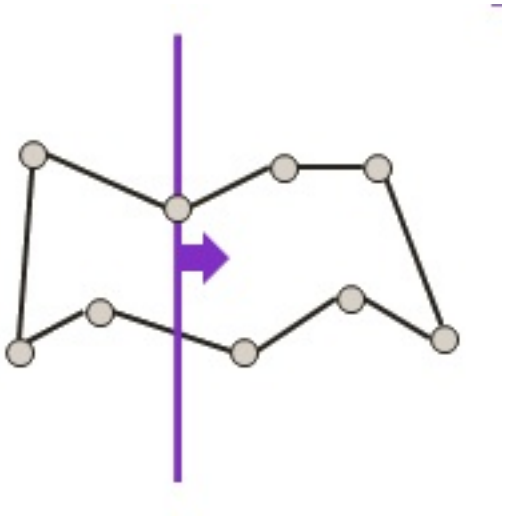
算法 將凸包分成上下兩部份分開求。
首先將所有點依 x 座標排序，x 座標相同時照 y 座標排序。排序完後的第一個點 (最左，最下) 和最後一個點 (最右，最上) 都必在凸包上。
先從第一個點 (最左，最下) 開始求上半部的凸包。開一個 stack 存放屬於上半部凸包的點。首先把排序前 2 個點丟進去。隨時考慮 stack 中倒數 2 個點 $A, B(B$ 為最後一個點) 以及排序中的下一個點 C。如果 $A–B–C$ 為一逆時針轉動，則表明 B 不在凸包上，故把 B 刪掉。若 stack 剩不到 2 個點或 $A–B–C$ 為一順時針轉動，則將 C 放進 stack 中，並從排序中移除。一直重覆此動作，直到排序被清空，此時 stack 中的點就是上半部的凸包。以相同的方式從最後一個點開始，求出下半部的凸包，再合併後可得到完整的凸包。

時間複雜度：$O(n \lg n)$ (若已排序好，則可做到 $O(n)$)



4.4.7 掃描線 (Sweep Line)

想像有一條延伸到無線遠的直線，朝其法線方向前進，掃過所有的點。



實作方式為將所有點依照 x(或 y) 座標排序，再依序處理所有的點。

4.4.8 判斷線段相交

上一次已經教過了判斷兩線段是否相交的方法，但若是有 N 條線段的狀況呢？

判斷是否有任何線段相交

先判掉有鉛直線以及共同交點的狀況。

這裡利用到了兩個性質：

1. 在判斷的過程中，可以假設所有考慮過的線段皆不相交 (否則就 return false 了)
2. 一些不相交的線段，其上下順序不會隨著 x 座標而改變。
3. 若兩條線段相交，則必定在某個 x 座標區間，兩條線段中沒有其他線段。

因此建立一條從左到右的掃描線，當經過一條線段的左端點時，判斷自己和上下兩條線段是否相交；經過一條線段的右端點時，判斷上下兩條線段是否相交。

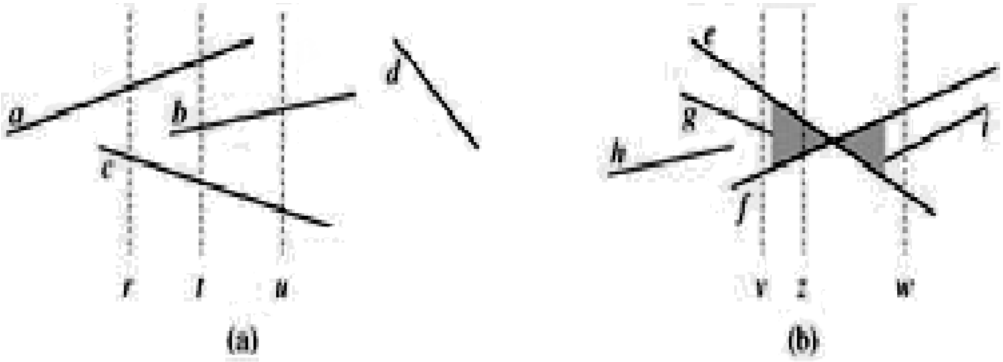
實作上會建立一個集合，存放還在掃描線範圍內的所有線段，並依與掃描線的交點由上往下排序。這個集合可以使用平衡樹 (或 `std::map`) 來實作。

排序 $O(N \lg N)$。總共 N 條線段，每條線段判斷相交 $O(N)$，插入/刪除 $O(\lg N)$。總時間複雜度：$O(N \lg N)$。

找出所有的線段交點

N 條線段最多有 $\frac{N(N-1)}{2} = O(N^2)$ 個交點，那還不如直接枚舉比較快…

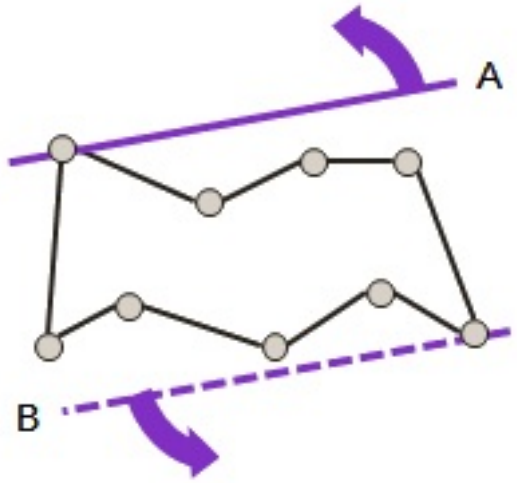
當然如果已知交點數目 K 不大的話，可以將上面的演算法稍微修改一下：當兩線段有相交的時候，記錄起來，並交換順序。



時間複雜度：$O((N+K)\lg N)$。(注意到如果線段全部兩兩相交時，甚至會比枚舉還慢)

4.4.9 旋轉卡尺 (Rotating Caliper)

以兩條會旋轉的平行線夾住圖形。



事實上會被旋轉卡尺切到的點、邊必定是凸包上的點、邊，所以經常先做完一次凸包。

實作上會記錄兩條線分別交圖形於哪兩個點。枚舉一個點 (主點) 旋轉，另一個點 (副點) 跟著轉。

至於要推進主點或副點，可以由斜率來決定。假設主點為 P_i，副點為 P_j，則當外積

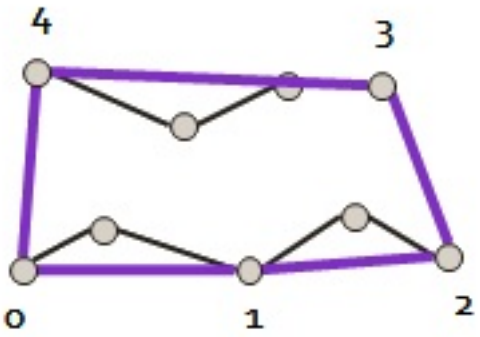
$$\overrightarrow{P_iP_{i+1}} \times \overrightarrow{P_{j-1}P_j} > 0$$

時，代表推進副點，否則推進主點。可以寫成雙重迴圈的型式。

4.4.10 最遠點對

一個點集中最遠的點對，必定會被旋轉卡尺夾到 (且會讓卡尺的寬度最大)。因此利用旋轉卡尺繞一圈，隨時記錄被夾到的點的距離，取最大值即可。

時間複雜度：$O(N\lg N)$。



4.4.11 最小包含圓

給定一個點集，找出一個最小的圓，使得點集內的所有點都在這個圓的內部/邊界上。

縮小圓算法 首先將圓設得很大，包含所有的點。接著慢慢縮小圓，直到圓周上有 2 個以上的點為止。此時找到一個 $> 180\deg$ 的弧 AB，將圓以 AB 為一弦向另一邊小，直到有另一點也在圓周上，或者 AB 成為直徑。重複這個動作，直到不能做為止。

時間複雜度：$O(n^2)$

A___A 算法 先找一個很接近 1 但小於 1 的漸近係數 ϵ，如 $\epsilon = 0.999$。隨便找一個靠近點集中心 (例如重心) 的點 P，找距離 P 最遠的點 Q，找到點 P 使得 $\overrightarrow{PP'} = \epsilon\overrightarrow{PQ}$。把 P 點當作新的中心，找距離 P' 最遠的點 Q'，再找點 P' 使得 $\overrightarrow{P'P''} = \epsilon^2\overrightarrow{P'Q'}\cdots$直到 $\epsilon^n < 10^{-20}$ 為止。這個點即為最小包含圓的圓心。
(這個演算法雖然無法證明正確性，但 RP 爆發時可以通過所有測資)

隨機增量法 考慮一個問題，令 C_n 為 $P_1, P_2, \cdots, P_n$ 的最小包含圓，可以知道如果 P_k 不在 C_{k-1} 內或圓上，則 P_k 在 C_k 圓周上。
接著如果知道一個點在最小包含圓上，令 D_h 為 $P_1, P_2, \cdots, P_h (h < k)$，且已知 P_n 在圓上的最小包含圓，可以知道如果 P_{k+1} 不在 D_k 內或圓上則 P_{k+1} 在 D_{k+1} 圓周上。確定了圓周上的兩個點後，我們便可以跟前面所有點做一次最小包含圓 (3 個點可以確定圓了)，而最大的圓即為最小包含圓。

4.4.12 Exercise

1. 給定兩凸多邊形，在兩凸多邊形中各選一點形成一點對，求這種點對的最短/最長距離。
2. 運河 <TIOJ 1041> <NPSC2003 初賽 (prob D)>
   你在玩世 J122 帝國，你和你的敵方共蓋了 N 個城堡，現在你要挖一個無限延伸的護城河讓你的城堡都和他的城堡隔開 (以免他海哥德衛隊來)，運河寬度越大越好，問你運河寬度。

3. 朝向控制室的咆哮 <TIOJ 1765> <2011 快樂暑假營第二次模擬賽 pJ>
   給你 N 個點，選 3 個點讓他圍出的三角型面積最大。
4. 矩包
   求出包含平面上所有點，且面積最小的矩形。
5. 給定兩個凸包，求這兩個凸包的交集。
6. 給定 N 個半平面，出所有半平面的交集。

5 圖論

5.1 圖論簡介

圖論是數學的一個分支，顧名思義是一門以圖為研究對象的學問。但這邊的圖和我們一般的圖有點不同，在這個圖上我們用點代表不同的元素，比如每個點可能代表不同的小蘿莉，而如果這些點之間有關係，那我們就連一條邊，比如說兩隻小蘿莉互相是朋友，那我們就用一條「友誼」線連接兩者。從這個例子我們可以知道我們只注意元素還有他們之間的關係而已，點的大小，位置，或是邊的長度等等都不是我們關切的。

5.1.1 定義

- Graph，圖：即代表元素和他們之間的關係的集合，記做 $G = (V, E)$。其中 V 代表這個圖點的集合，E 則代表圖中邊的集合。
- Vertex,Node，(頂) 點：即我們關心的元素的集合，圖 G 的點集記作 $V(G)$。點的數量 $|V|$ 又稱為階。
- Edge，邊：即代表關係的集合，圖 G 的邊集記作 $E(G)$。兩個點 v_1, v_2 之間的邊通常用 (v_1, v_2) 或是 $e(v_1, v_2)$ 表示。關係如果是雙向的，我們就稱為雙向邊 (Undirected edge, Edge)，即 $e(v_1, v_2) = e(v_2, v_1)$；而關係如果是單向的，我們就稱作單向邊或是弧 (Directed edge, Arc)，此時 (v_1, v_2) 會表示一條方向從 v_1 到 v_2 的有向邊。
- Undirected Graph，有向圖：邊皆為無向邊的圖。
- Directed Graph，有向圖：邊皆為有向邊的圖。
- Mixed Graph，混合圖：同時存在無向邊和有向邊的圖。
- Adjacent，相鄰：表達點之間的關係，當我們說 v_1 與 v_2 相鄰若且為若
  1. 存在至少一條無向邊 $(v1, v2) = (v2, v1)$。
  2. 存在至少一條有向邊 $(v2, v1)$。
- Degree，自由度：一個點 v 的自由度代表他與多少邊相鄰，記做 $\deg(v)$。
- In-Degree，入度：一個點 v 的入度代表他與多少指向他的有向邊相鄰，記做 $\deg^+(v)$。
- Out-Degree，出度：一個點 v 的入度代表他與多少指出去的有向邊相鄰，記做 $\deg^-(v)$。

Question

- **請說明一個圖中所有頂點的度數總和 $\sum_{v\in V(G)}\deg(v)$ 與邊的數量 $|E|$ 的關係。**
- **請說明一個圖中所有頂點的入度數總和 $\sum_{v\in V(G)}\deg^{+}(v)$ 與所有頂點的出度數總和 $\sum_{v\in V(G)}\deg^{-}(v)$ 的關係。**

5.1.2 圖的儲存

接下來我們應該考慮的問題是如何儲存，一般有以下幾種儲存的方式。

1. Adjacent Matrix，相鄰矩陣：以一個矩陣 $A=[a_{ij}]_{n\times n}$，而 a_{ij} 則記錄邊 (v_i,v_j) 的資訊，比方說如果 $(v_1,v_2)\in E(G)$，則令 $a_{ij}=1$，否則 $a_{ij}=0$。
   這樣是最直觀的儲存方法，但缺點是沒辦法處理重邊，而且空間和時間複雜度會變成 $O(V^2)$。

2. Adjacent List，相鄰串列：對每一個點 v_i 開一個串列 L_i 紀錄這個點連到那些其他的頂點。
   這樣記憶體減少為 $O(E)$，也解決了重邊的問題。

3. Foward Star，前向星：將所有的邊儲存在一個陣列中，並且依照某種順序排序，使得每一個點連 v_i 連出去的邊都恰好在一個範圍 $[s_i,e_i]$ 內，通常對於一個邊 (v_i,v_j) 我們先按照 v_i 大小排序，如果相同在按照 v_j 大小排序。
   算是一個完美的結構，幾乎所有操作都是線性的，只是實作稍嫌麻煩。

Question

- **在相鄰矩陣 $A=[a_{ij}]_{n\times n}$ 中如果令 a_{ij} 代表有幾條邊連接 (v_i,v_j)，請說明 A^k 所代表的意義。**

5.1.3 Exercise

- 證明：在一個有 6 個頂點的圖 G 中，一定可以找到 3 個點他們兩兩皆有邊相連，或著可以找到 3 個點他們兩兩皆沒有邊相連。
- 證明：在一個有 6 個頂點的圖 G 中，一定可以找到 2 組 3 個點使他們兩兩皆有邊相連，他們兩兩皆沒有邊相連。
- 如果有 3 個點，他們兩兩皆有邊相連，我們就稱作一個三角形。證明：在一個有圖 G 中，至少有 $\frac{4|E|^2-|E||V|^2}{3|V|}$ 個三角形。
- 請給出一個儲存有向圖的方法，滿足：
  1. 只需要 $O(|V|+|E|)$ 的記憶體。
  2. 查詢一個有向邊的反邊只需要 $O(1)$ 的時間。

5.2 子圖與特殊的圖

5.2.1 定義

- Subgraph，子圖：如果兩個圖 $G=(V,E), G'=(V',E')$，如果 $V'\subseteq V, E'\subseteq E$，那我們說 G' 是 G 的子圖。
- Induced Subgraph，導出子圖：如果圖 G' 是 G 的子圖，且對於任兩個頂點 $u,v\in V(G')$，$(u,v)\in E' \iff (u,v)\in E$，那我們說 G' 是 G 的導出子圖。
- Spanning Subgraph，生成子圖：如果圖 G' 是 G 的子圖，$V(G)=V(G')$，那我們就說 G' 是 G 的生成子圖。
- Compliment Graph，補圖：圖 G 的補圖 G^c，為滿足 $V(G^c)=V(G)$ 且 $e\in E(G) \iff e\notin E(G^c)$ 的圖。
- Path，路徑：圖 G 中的一個頂點的序列 $(v_1,v_2,\cdots,v_n)$ 滿足 $v_i\in V(G), (v_i,v_{i+1})\in E(G)$。$v_1$ 稱作路徑的起點，v_n 為路徑的終點。
- Simple Path，簡單路徑：點都不重複的路徑。
- Cycle，環：一個起點與終點的路徑。
- Simple Cycle，簡單環：點都不重複的環。

5.2.2 一些特殊的圖

簡單圖

我們把一條邊 (v,v)，即邊的兩端連接同一個頂點稱作自環。而如果有兩個邊 e_1,e_2 同時連接相同的兩個頂點 (v_1,v_2)(在有向途中則如果同時有兩個有向邊的起點為 v_1，終點為 v_2)，我們就稱作重邊。
一個簡單圖即是沒有上述兩種邊的圖。

Question

- 給你一個有 n 個點的度數的序列 $\deg(v_1),\deg(v_2),\cdots,\deg(v_n)$，請判斷是否存在一個簡單圖使得圖中點的度數序列恰為此序列。(<TIOJ 1210> [圖論之簡單圖測試],$O(n\lg n)$)

連通圖

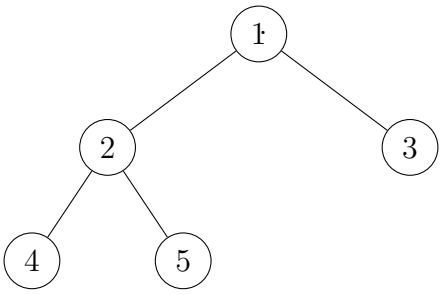
如果一個圖 G 滿足圖中任兩個點 v_i,v_j 都有一個路徑連接兩點，那我們就說 G 是連通圖 (Connected Graph)。

樹

如果一個無向的簡單圖 G 滿足 G 是沒有環的連通圖，那我們就稱 G 為一棵樹 (Tree)。而在有向圖 G 中，如果存在一點 v 使得 $\deg^+(v) = 0$，且任何其他點 u 滿足 $\deg^+(u) = 1$，並且從 v 到 u 有且僅有一條簡單有向路徑相連，我們就說 G 是一個以 v 為根的有向樹 (Directed Tree)。

Question

- **請說明以下的定義和樹的定義等價。**
  1. **G 沒有環，但是在 G 添加任意一條邊就會形成一個環。**
  2. **G 是連通的，但是如果去掉任意一條邊就會使得 G 不連通。**
  3. **G 中任意兩個頂點能被唯一一條簡單路徑連通。**
  4. **G 是連通的，且 $|E| = |V| - 1$。**
- **如果一個圖 G 滿足 $|E| = |V|$，請問這個圖會有甚麼性質？**



完全圖

如果無向圖 G 滿足任意兩個頂點都有一條邊相鄰，那我們說 G 是完全圖 (Complete Graph)，有 n 個頂點的完全圖是唯一的，記作 K_n。

二分圖

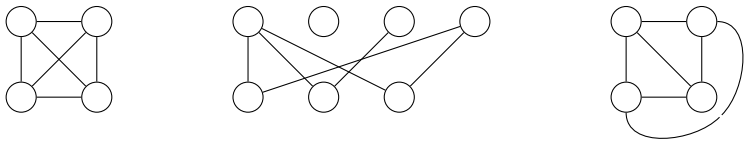
如果無向圖 G 滿足可以將 G 中的頂點分成兩個集合 V_1, V_2，使得任意 $(v_1, v_2) \in E(G)$，v_1, v_2 不會在同一個集合 V_1 或是 V_2 內，我們就說 G 是二分圖 (Bipartite graph)。

Question

- **請判斷一個圖 G 是不是二分圖。(<TIOJ 1209> [圖論之二分圖測試],$O(|V|)$)**

平面圖

如果圖 G 可以畫在平面上使得他的邊僅在端點處相交，我們就說 G 是平面圖 (Planar Graph)。一個平面圖的邊會把平面區隔成不同的區域稱作面，記作 F。平面圖中有著名的歐拉定理 $|V| - |E| + |F| = 2$



5.2.3 Exercise

- 給你一個圖 $G = (V, E)$，其中 $|E| = |V|$，找出這個圖中最長的一條簡單路徑。(<IOI 2008> [Island], $|V| \leq 3000$)
- 如果一個圖 G 的頂點可以被分成兩個集合 C, I 滿足 C 中的任兩點都有邊相鄰，而 I 中的任兩點都沒有邊相鄰，那我們說 G 是一個 Split Graph。給你一個簡單圖 G，請問他是不是 Split Graph。(<POI XVIII> [Conspiracy], $O(|V| + |E|)$)
- 如果一個有向圖 $G = (V, E)$ 滿足任兩個點 v_1, v_2，$(v_1, v_2) \in E$ 或是 $(v_2, v_1) \in E$ 恰有一個成立，那我們就說這是一個競賽圖 (Tournament Graph)。給你一個競賽圖 G，請找出一個 3-環 (長度為 3 的簡單環) 或是輸出無解。(<CodeForces 117C> [Cycle], $|V| \leq 3000$)
- 證明：任何一個競賽圖 G，一定可以找到一個點 v，滿足對於其他所有的點 u，使得以下兩點至少有一點成立：
  1. $(u, v) \in E(G)$。(注意競賽圖是有向的)
  2. 存在另一個點 w 滿足 $(u, w), (w, v) \in E(G)$。
- 給你一個圖 $G = (V, E)$，其中 $|E| = |V|$，找出這個圖中最長的一條簡單路徑。
- 給你一個圖 $G = (V, E)$，其中 $V = 3k$，已知這個圖中有一個子圖為 $2k$ 階完全圖 K_{2k}，請找出這個圖的一個子圖 G' 滿足 G' 為 k 階完全圖 K_k。(<POI XVIII> [Party], $|V| \leq 3000$)
- 證明：如果 G 是一個平面圖，那麼 $|E| \leq 3|V| - 6$。
- 給你一個樹 $T = (V, E)$ 和平面上的 $|V|$ 個點 V'，請用這些點構造一個平面圖 $G' = (V', E')$ 滿足 T 和 G' 同構。即用平面上的這些點畫原本所給的樹 T。(<CodeForces 196C> [Paint Tree], $|V| \leq 10^5$)

5.3 圖的遍歷與時間戳記

圖的遍歷 (Traversal) 是指一張圖從某個點 v 開始，依照某種順序拜訪與已拜訪的點相鄰的點，最終拜訪過圖內所有頂點。而如果我們令 T 為包含拜訪過的節點和拜訪節點所經過的邊的子圖，可以知道 T 會形成一棵 (有向) 樹。我們把樹 T 稱作搜索生成樹。

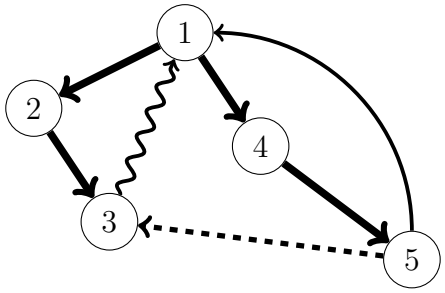
5.3.1 深度優先搜索

所謂的深度優先搜索 (Depth-First Search) 即是搜索順序以一個點的子節點優先而非兄弟節點。

而深度優先搜索的搜索生成樹即稱為 DFS-Tree。

Algorithm 14 Depth-First Search

```
1: function DFS(v:current vertex,S:set of visited vertices)
2:     S ← S ∪ v
3:     for each u adjacent to v do
4:         if v ∉ S then
5:             DFS(u,S)
6:         end if
7:     end for
8: end function
```

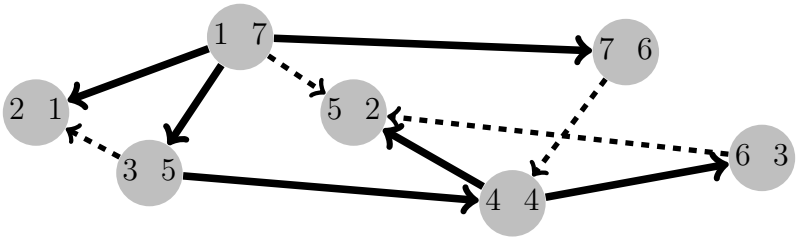


而經過深度優先搜索後，我們便可以將原本圖上的邊分為以下四種：

1. Tree-Edge: 即 DFS-Tree 上的邊。
2. Back-Edge: 連到自己祖先且非 DFS-Tree 上的邊。
3. Foward-Edge: 連到自己子孫且非 DFS-Tree 上的邊。
4. Cross-Edge: 不屬於以上三者的邊。

5.3.2 時間戳記

在搜索的過程中我們可以記下進入或是離開點的順序，分別稱作進入時間戳記和離開時間戳記。而如果是以深度優先搜索，我們便稱作 DFS 時間戳記



這個方法雖然簡單，卻可以幫助我們解決不少問題。

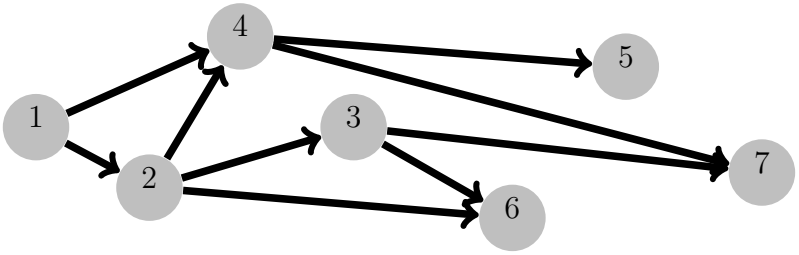
Topological Sort

拓樸排序 (Topological Sort) 是希望能在有向圖 G 找到一個頂點排列的順序 $\{v_1, v_2, \cdots, v_n\}$，使得對於兩點 v_i, v_j，如果 $i < j$ 那麼 $(v_j, v_i) \notin E(G)$。也就是說如果一條有向邊 (v_i, v_j) 代表 v_i 在序列中要排在 v_j 的前面，那一個拓樸排序就會是一個合理的順序。

一個圖可能有不只一個拓樸排序，或是沒有拓樸排序。可以知道一個有向圖如果有環那是不可能有拓樸排序的，我們把這種圖稱為有向無環圖 (DAG, Directed Acyclic Graph)。

接下來的問題便是要如何找到一個拓樸排序了。一個很直觀的想法是找目前圖上入度為 0 的點，並且將這個點和這個點連出去的邊從圖中拔掉，遞迴做下去直到圖上的所有點都被拔掉，那點被拔掉的順序就會是一個合法的拓樸排序。事實上這個做法仔細思考便可以做到 $O(|V| + |E|)$ 的時間複雜度，但我們要介紹一個更簡潔的作法。

考慮一下 DFS 的離開時間戳記，容易證明離開時間戳記較小者不可能有邊連向離開時間戳記較大者，因此 DFS 離開時間戳記的相反順序便是一個合理的拓樸排序。這樣時間複雜度也是 $O(|V| + |E|)$。

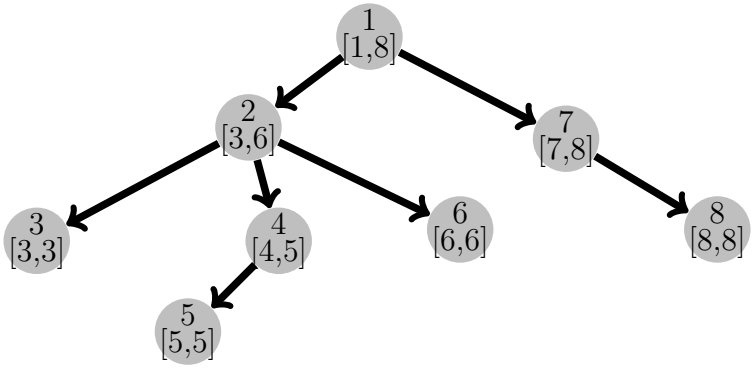


樹的區間表示

在一個樹 T 上，有時候我們會需要查詢某個點 v 的子節點有哪些或對他所有子節點進行操作。事實上我們可以找到一個序列 $\{v_1, v_2, \cdots, v_n\}$ 使得某個點的所有子節點會對應到序列

中的一個區間，即這個序列給我們一個將樹壓扁的方法。

考慮一下 DFS 的進入時間戳記，可以知道在樹 T 上一個點 v 的任何子節點的進入時間戳記，必定會小於 v 的進入時間戳記，並且對於任和 v_i, v_j 為 v 的子孫，如果 $i < k < j$ 那 v_k 也必定會是 v 的子孫，否則我們便進出點 v 兩次了，因此 DFS 進入時間戳記恰好是一個合法的序列。



5.3.3 Exercise

- 有一個不知道順序的字元集。給你很多這個字元集組成的字串，並告訴你這些字串是依照這個字元集的字典順序排好的。請輸出每個字符的大小關係。(<UVa 200> [Rare Order])
- 給你一個有向圖，還有一個序列 $a_1, a_2, \cdots, a_n$，其中 a_i 可能是數字或是問號，請問存不存在一個拓樸排序使得如果 a_i 不是問號的話，v_i 恰好為拓樸排序的第 a_i 個。(<NTU 1637> [Generalized Pizza Book], $|V|, |E| \leq 10^5$)
- 有一個序列 $1, 2, \cdots, N$，現在有 M 組關係 (a_i, b_i) 表示如果數字 a_i, b_i 在序列中相鄰的話就可以交換，問你經過合法的交換後字典序最大的序列。($|N|, |M| \leq 10^5$)
- 給一個有向圖 $G = (V, E)$，其中 v_1 是頂點，並且每條邊都有 1 或是 0 的權重，你要選一些邊使得可以從 v_1 走到任何頂點，並且選出的權重和最小。(<Codeforces 240E> [Road Repairs], $|V|, |E| \leq 10^5$)

5.4 連通元件

5.4.1 連通元件

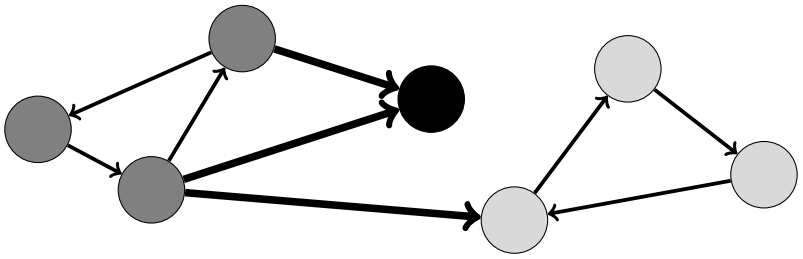
如果圖 G 的一個子圖 G' 滿足任兩個在 G' 內的點都連通，且在加入任意一個不再集合內的點到集合裡就會破壞這個性質，我們就稱 G' 為圖 G 的一個連通元件 (Connected Component)。我們主要的目的在於將圖分成數個連通元件，利用同一個連通元件內的性質解決問題。事實上我們還可以定義連通的條件，而有不同種的連通元件。

弱連通元件

我們說兩個點 u, v 為弱連通的，若且為若存在從 u 到 v 或 v 到 u 的路徑。而一個弱連通元件則是與此連通的定義關聯的元件。
但事實上在無向圖中弱連通與一般的連通是等價的，而在有向圖中容易發現弱連通元件常有重疊的現象，並沒有特別的性質，因此通常不討論。

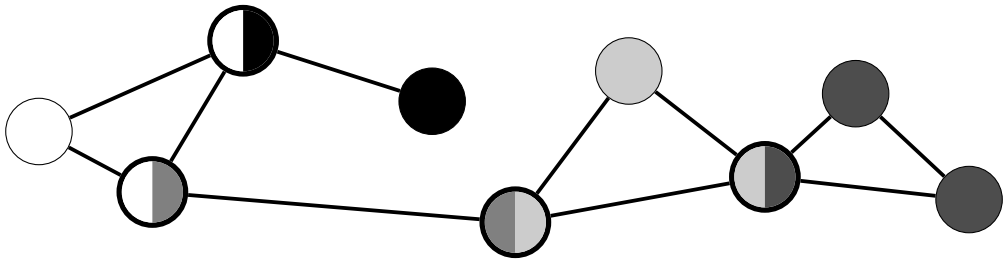
強連通元件

我們說兩個點 u, v 為強連通的，若且為若同時存在從 u 到 v 和從 v 到 u 的路徑。
在無向圖中強連通與一般的連通也是等價的，所以通常在有向圖中討論。強連通元件會把原本的圖 G 中的點集 V 分成不同的不重疊元件 $V_1, V_2, \cdots, V_n$，即 $V_i \cap V_j = \varnothing$ 對任何 $i \neq j$。由定義可以知道圖 G 上的環會屬於同一個連通元件，因此如果我們令 $V_1, V_2, \cdots, V_n$ 為新的圖 G' 的頂點，而 $(V_i, V_j) \in E(G')$ 若且為若 $\exists v_i, v_j, v_i \in V_i, v_j \in V_j \Rightarrow (v_i, v_j) \in E(G)$，即是將原圖依照強連通元件「縮點」，新的圖 G 會是一個有向無環圖。
這個「縮點」的步驟可以幫我們解決很多問題，比如說我們要在有向圖上 DP，如果因為有環無法 DP，我們便可以先縮點，每個強連通元件做個別處理後，就可以在縮點後的圖上 DP。



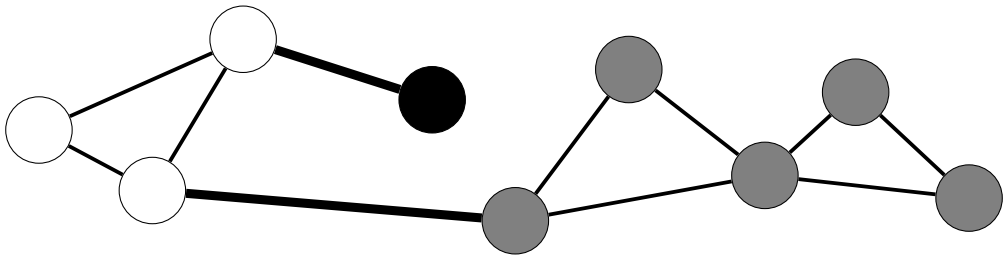
點雙連通元件

通常我們會在無向圖討論點雙連通，我們說兩個點 u, v 為點雙連通的，若且為若拔掉任意其他 u, v 以外的點，還是存在從 u 到 v 或 v 到 u 的路徑。
雙連通元件會把原本的圖 G 中的點集 V 分成不同的元件 $V_1, V_2, \cdots, V_n$，而 $|V_i \cap V_j| \leq 1$，即兩個點雙連通元件至多重疊到一個點，這個點即是拔掉後會造成不連通的點，稱作割點 (AP, Articulation Point)。



邊雙連通元件

同樣的，通常我們會在無向圖討論邊雙連通，我們說兩個點 u, v 為邊雙連通的，若且為若拔掉任意一條圖上的邊，還是存在從 u 到 v 或 v 到 u 的路徑。邊雙連通元件與有向圖上的強連通元件類似，會把原本的圖 G 中的點集 V 分成不同的不重疊元件 $V_1, V_2, \cdots, V_n$。由定義可以知道圖 G 上的環會屬於同一個邊強連通元件，因此我們也可以進行縮點變成圖 G'。而在 $E(G')$ 中的邊即是在原圖上拔掉會造成點對不連通的邊，稱作橋 (Bridge)。



Question

- **在一個無向圖 G 中，對於任何一個橋 (v_1, v_2)，v_1, v_2 是否都必為割點？**
- **在一個無向圖 G 中，對於任兩個割點 v_1, v_2，如果 $(v_1, v_2) \in E(G)$，(v_1, v_2) 是否必為橋？**

5.4.2 Tarjan's algorithm

Tarjan's algorithm 是一個能在線性時間 ($O(|V| + |E|)$) 找出以上不同的連通元件的演算法。其中最主要的精隨就是 Low 值還有 Dfn 值的計算。

Low 值和 Dfn 值

首先我們對一個圖 G 做一次深度優先搜索，之後我們便可定義 Low 值和 Dfn 值：

- $\text{Dfn}(v) \equiv v$ 在 DFS 生成樹中的深度。
- $\text{Low}(v) \equiv \min_{u,w} (\text{Dfn}(v), \text{Dfn}(u), \text{Low}(w))$，其中 u 為所有以 v 為起點的 Back-Edge 所連到的點，w 為 u 的所有子節點。

而一個點 v 的 Low 值的意義即為 v 和 v 的子孫可以藉由 Back-Edge 回到多久以前的子孫。

應用

考慮有向圖中的強連通元件，可以知道在 DFS-Tree 上如果存在一個點 v 使得 $\text{Low}(v) = \text{Dfn}(v)$，代表 DFS-Tree 中所有 v 的子節點最多只能藉由 Back-Edge 回到 v 而已，即 v 沒有路徑可以回到任何 v 的祖先，因此 v 和尚未屬於任何強連通元件的所有 v 的子節點會屬於一個強連通元件。

而無向圖中的橋與之類似，如果在 DFS-Tree 上存在一個點 v 和他的子節點 u 使得 $\text{dfn}(v) < \text{Low}(v)$，那 (u, v) 便是一條橋。而圖便會被橋分割成不同的區塊，每一塊都會恰好是一個邊雙連通元件

無向圖中的割點則要分類討論，如果 v 是在 DFS-Tree 上的根，且 v 有兩個以上的子結點，那 v 便會是割點；而如果 v 非 DFS-Tree 上的根，且存在一個 v 的子結點 u 使得 $\text{dfn}(v) \leq \text{Low}(v)$，那 v 便會是割點。同樣的圖便會被割點分割成不同的區塊，每一塊都會恰好是一個點雙連通元件，而割點便會是兩個點雙連通元件的交集。

5.4.3 2-SAT

考慮一個特殊的布林運算式

$$(v_{11} \vee v_{12}) \wedge (v_{21} \vee v_{22}) \wedge \cdots \wedge (v_{m1} \vee v_{m2})$$

其中 v_{ij} 為某個 $x_1, x_2, \cdots, x_n$ 或是某個 $\neg x_1, \neg x_2, \cdots, \neg x_n$，注意到任何 x_i 或是 $\neg x_i$ 可以在 v_{ij} 中重複出現。現在我們要判斷是否能設定每個 x_i 為 true 或是 false，使得布林運算式的結果為 true。

觀察可以發現因為括號是以 and 連接，所以如果整個運算式要是 true，那所有 $v_{11} \vee v_{12}$ 都必須為 true，即 v_{11}, v_{12} 至少有一個是 true，也就代表如果 v_{11} 為 false，那麼 v_{12} 就必須為 true，反之亦然。

所以便有一個這樣的解法，我們將 $x_1, x_2, \cdots, x_n$ 還有 $\neg x_1, \neg x_2, \cdots, \neg x_n$ 視為圖上的點，而如果有一個括號內的式子，比如說是 $x_1 \vee x_2$，那我們就建一條從 $\neg x_1$ 到 x_2 的有向邊，代表如果你選擇將 x_1 設為 false，那你就必須將 x_2 設為 true；同樣道理再建一條從 $\neg x_2$ 到 x_1 的有向邊。而當我們將圖建完了以後，如果存在一個 i 值使得 x_i 和 $\neg x_i$ 在同一個強連通元件裡，代表同時存在 x_i 到 $\neg x_i$ 和 $\neg x_i$ 到 x_i 的路徑，也就是不管將 x_i 設為 true 或是 false，都會導致另一個條件讓你非得將 x_i 設為相反的值，即不管如何都會導致矛盾，此時即無解。

因此由上述的方法，我們便成功的將原問題化為圖論上的問題。

Question

- **請問對於下列為一個括號內的式子，我們要如何在圖上建邊？(以下 $i \neq j$)**
  1. $(x_i \vee x_j)$
  2. $(x_i \vee \neg x_j)$
  3. $(\neg x_i \vee \neg x_j)$
  4. $(x_i \vee x_i)$
  5. $(x_i \vee \neg x_i)$
- **如果所有 $x_i, \neg x_i$ 都不在同一個強連通元件裡，那是否一定有解？如何找出一組解？**

5.4.4 Exercise

- 給你一個無向圖 $G = (V, E)$，問你其補圖 G^c 有多少連通元件。(<POI XIV> [Office], $|V| \leq 10^5, |E| \leq 10^6$)
- 給你一個有向圖 $G = (V, E)$，其中每個點 v_i 都有銀行搶一次會讓你得到 a_i 元，並且有些點還有酒吧，你要跟你同伴從一個點出發去搶錢，最後到一個酒吧喝酒享受吃牢飯錢的歡愉，問你最多能搶到多少錢。(<APIO 2009> [ATM], $|V|, |E| \leq 5 \times 10^5$)
- 給你一個無向圖 $G = (V, E)$，你的家在 $v_1, v_2, \cdots v_k$(你是富二代 2^k，所以你有一堆房子)。很不幸的有人要在圖上辦一個腳踏車賽，任何環都有可能是腳踏車賽舉辦的路徑，好險你是富二代你可以拆路，問你至少要拆幾條路。(<POI XIX> [Tour de Byteotia], $k \leq |V|, |E| \leq 10^6$)
- 給你一個無向圖 $G = (V, E)$，你要將每一個點定向，即將原圖變成有向圖，滿足任兩點 v_1, v_2，同時存在 v_1 到 v_2 和 v_2 到 v_1 的路。(<CodeForces 118E> [Bertown roads], $|V| \leq 3000$)
- 給一張有向圖，選一些起點使得所有的點都可以被到達，問最少要選幾個點？(<TIOJ 1451> [八卦傳播系統], $O(|V| + |E|)$)
- 給定一張無向圖和一個起始點。請拔掉一個頂點使得拔掉後起始點能到達的點的數量最少。(<TOI 2009 入營考 prob.5> [謠言問題], $O(|V| + |E|)$)
- 給一張無向圖 $G = (V, E)$，有 Q 比詢問求拔掉某一個邊後點 a_i, b_j 是否連通？(<NPSC 2011> [整修中的道路], $|V| \leq 32000, |E| \leq 514000, Q \leq 201112$)
- 給一張有向圖 $G = (V, E)$，如果 $(v_i, v_j) \in E(G)$ 代表 i 必定贏 j 否則雙方都有可能獲勝，現在要辦一個單淘汰賽，而你可以隨意安排賽程，問你那些人有可能獲勝？(<POI XI> [The Tournament], $|V| \leq 10^5, |E| \leq 10^6$)
- 給一個 $M \times N$ 的地圖，你可以任意轟炸一行或是一列，但有些格子一定要被炸到兩次，有些格子至少要被炸到一次，有些格子至多被炸到一次，有些格子完全不能被炸到，問你可不可以辦到。(<NPSC 2010> [轟炸任務])

5.5 圖論上的問題 (I)

圖論延伸出了許多不同的問題，我們將在以下一一討論。

5.5.1 歐拉路徑與歐拉迴路

一個跡 (Trail) 為圖上的一個路徑，且路徑所經過的邊都不重複。而歐拉路徑 (Eulerian trail) 則是希望能找出圖上的一個跡，恰好將圖上的所有邊都走過一次。而歐拉迴路 (Eulerian circuit) 還要求起點和終點為同一個點。我們分無向圖和有向圖討論。

無向圖中的歐拉路徑

如果一個點 v 的度數 $\deg(v)$ 為奇數，我們就稱此點為奇點，否則稱為偶點。
可以知道對於一條起點終點不同的路徑 $v_1, v_2, \cdots, v_n (v_1 \neq v_n)$，除了 v_1, v_n 為奇點之外其他都會偶點，不管點有沒有被重複經過。而一個環 $v_1, v_2, \cdots, v_n, v_1$ 上的所有點都會是偶點，因此我們可以總結以下：

- 如果一個連通圖 G 有歐拉迴路，則圖 G 沒有奇點。
- 如果一個連通圖 G 有歐拉路徑，則圖 G 有 0 或是 2 個奇點。且如果圖上有 2 個奇點，那路徑一定是從其中一個奇點出發終於另一個奇點。

而奇妙的是這些條件不僅是必要條件也是充分條件。

而我們可以一以下方法在一個沒有奇點的圖找到一個歐拉路徑。從任意一點開始進行 DFS，將經過的邊紀錄下，直到走到死路，開始進行回溯。在回溯過程中將所經過之邊丟入序列，回溯至尚未結束拜訪之點再繼續搜索。最後將序列中的邊依序印出。若圖有兩個奇點，則可先加一條邊連結這兩點，將整張圖變為沒有奇點的圖。做完尤拉迴路之後再把該邊拆掉即可。

有向圖中的歐拉路徑

與無向圖的情況類似，對於所有聯通圖 G 我們可以分以下討論：

- 若所有點 v 滿足 $\deg^+(v) = \deg^-(v)$，則一定存在尤拉迴路。
- 若恰有兩點 v_1, v_2 滿足 $\deg^+(v_1) = \deg^-(v_1) + 1, \deg^+(v_2) = \deg^-(v_2) - 1$，且其他點 v 滿足 $\deg^+(v) = \deg^-(v)$，則一定存在尤拉迴路。

5.5.2 漢米頓路徑與迴路

漢米頓路徑為圖上的一個簡單路徑，且路徑所經過的點都不重複，且恰好經過所有的點。漢米頓迴路則為起點終點相同的漢米頓路徑。

這個問題雖然乍看之下跟尤拉路徑很像，而且只需要經過所有的點即可，不需要經過所有的邊。但這個問題到現在都還沒有非常好的解法！事實上這個問題是 NP-Complete，目前最好的做法是使用狀態壓縮 DP 達到 $O(|V|^2 2^{|V|})$ 的時間複雜度。

5.5.3 Exercise

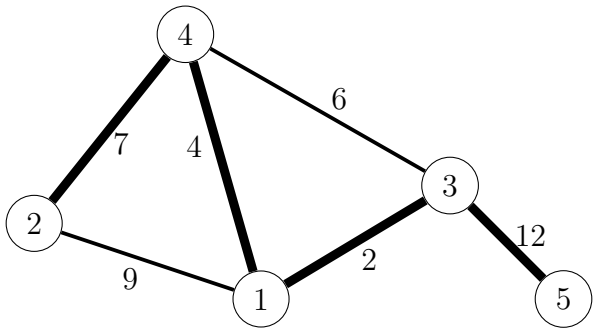
- 找出圖上字典序最小的尤拉回路/路徑。(<TIOJ 1084> [一筆畫問題])
- 給定一個無向圖 $G = (V, E)$，找一個邊權和最小的漢米頓路徑/迴路。([旅行推銷員問題], $O(|V|^2 2^{|V|})$)
- 給定一個無向圖 $G = (V, E)$，找一個經過所有邊的路徑/迴路，並且邊可以重複經過。(<99,100 年全國賽 > [洗街車路線問題, 城市馬拉松], $|V| \leq 1000, |V_{\text{odd}}| \leq 20$)
- 給定一個有向圖 $G = (V, E)$。每條邊有 0 或 1 兩種狀態，並且每條邊都有個兩個值 a_i, b_i 分別表示這條邊一開始的狀態和結束時要求的狀態。現在你可以開一台垃圾車，垃圾車行經路徑為一個簡單環，一條道路每被垃圾車經過一次就會從狀態 1 變成 0，或者從狀態 0 變成 1。求一組垃圾車的設置方案，使得每條路的狀態達到最終希望的狀態，並且行經總距離 $\leq 5|E|$。(<POI XVIII > [Garbage], $|V| \leq 10^5, |E| \leq 10^6$)
- 給出一個想法判斷一個混合圖是否有歐拉迴路。
- 證明：一個無向圖 $G = (V, E)$ 滿足對於所有相異的兩點 u, v 皆有 $\deg(u) + \deg(v) \geq |V| - 1$。那 G 必定存在漢米頓路徑。

5.6 圖論上的問題 (II)

有時候圖的邊會有權重 (Weight)，即關於這條邊的一個數字，代表的可能是邊的長度，花費等等，我們用 $w(e)$ 代表 e 的權重，而這一節我們主要即是探討邊上有權重的問題。

5.6.1 最小生成樹

給你一個無向圖 G，邊上有權重，求出一個邊權總和最小的生成樹 (Minimum Spanning Tree)，即選 $|V|-1$ 條邊將所有的點連起來，並要求權重和最小。



Prim's Algorithm

從任一個節點當作樹根開始逐步擴張樹，每次選擇從樹上一點連到非樹上一點的邊中權值最小的擴張，直到所有的點都被加入，即擴張 $|V|-1$ 次後結束。

Algorithm 15 Prim's Algorithm in MST

```
1: function PRIM(G: graph, T:current tree)
2:     T ← arbitrary vertex v ∈ G
3:     for i = 1to |V| − 1 do
4:         pick minimum e = (u, v) satisfied u ∈ T, v ∉ T.
5:         T ← T ∪ {e, v}
6:     end for
7: end function
```

最直觀的實做，開一個陣列紀錄每個點距離目前樹的距離。每次就選擇一個距離最小的點 v 加入樹中，並且用 $w(v,u)$ 更新所有 $(v,u) \in E(G)$ 的距離。
而因為每次只要查詢最小值，我們可以用 Heap 維護，達到 $O((|E|+|V|)\lg|V|)$ 的複雜度。

Question

- 請證明 Prim's Algorithm 的正確性。
- Prim's Algorithm 可以在有負權的圖上運作嗎？

Kruskal's Algorithm

把圖上所有邊依照小到大排好，從小開始加入，每次加入檢查是否形成環。若會形成環則跳過，依序進行到加入 $|V|-1$ 條邊為止。在實作通常用並查集 (Disjoint Set) 來判斷是否會形成環，複雜度為 $O(|E|\lg|E|+|E|\alpha(|V|))$

Question

- Kruskal's Algorithm 可以在有負權的圖上運作嗎？

5.6.2 單源最短路徑

給你一個圖 G，可能是無向也可能是有向圖，邊上有權重代表長度，求出一條起點到終點長度最短的路徑，一條路徑的長度即定義為所有邊權的總和。
首先我們應該考慮邊權正負的問題，如果有負權那可能會出現比較特殊的情況，比如說如果圖上有負環，即權重和為負的環，那最短路徑可能不存在，因此我們不考慮有負環出現的情況。在圖上沒有負環的情況下，可以知道最短路徑一定是簡單路徑。

Dijkstra's Algorithm

和 Prim's Algorithm 的想法類似，Dijkstra's Algorithm 是一個可以處理邊權皆為正的圖的演算法。從起點當作樹根開始逐步擴張最短路徑樹 T，每次選擇距離最短的一點擴張，這裡一個點的距離 $d(v)$ 就定義為 $d(v)=\min_{u\in T}(d(u)+w(u,v))$，直到所有的點都被加入，即擴張 $|V|-1$ 次後結束。實作也和 Prim's Algorithm 類似，用 Heap 維護可達到 $O((|E|+|V|)lg|V|)$ 的時間複雜度。

Question

- Dijkstra's Algorithm 可以在有負權的圖上運作嗎？

- 如果圖 G 上有負環，且邊權最小的權值為 $w_{\min}$，我們可以將所有的邊權加上 $w_{\min}$ 後當作邊權皆為正的圖去求單源最短路徑嗎？

Bellman-Ford's Algorithm

先定義一個動作叫做鬆弛 (relax)：我們用 $d(v)$ 代表目前從起點到 v 的最佳路徑，而如果存在兩個點 u, v 使得 $d(u) > d(v) + w(v, u)$ 則我們可以用 $d(v) + w(v, u)$ 來更新 $d(u)$，這個動作就稱作以 v 對 u 作一次鬆弛。
可以證明對於任何一點最多被鬆弛 $|V| - 1$ 次，不論有沒有負權存在，只要圖上沒有負環即可。Bellman-Ford's Algorithm 的想法即是將所有的邊依照任意順序鬆弛，重複 $|V| - 1$ 次。如果重複 $|V|$ 次以上還有邊可以鬆弛表示圖上有負環，因此 Bellman-Ford's Algorithm 也是一個可以偵測圖上有無負環的演算法。

Shortest Path Fast Algorithm

Shortest Path Fast Algorithm(SPFA) 其實只是 Bellman-Ford's Algorithm 的一個優化，一個點 v 要被鬆弛過才可能以 v 鬆弛其他的點，因此我們可以用單調對列 (如 Queue) 來維護目前被鬆弛的點，而每一次就從單調對列最前端取出點 v，之後更新所有與 v 相鄰的點的距離，如果我們以 v 去鬆弛某一點 u 了就把 u 丟進單調對列中，直到沒有點可以鬆弛了。
理論上最差時間複雜度 Bellman-Ford's Algorithm 一樣為 $O(VE)$，但是在純亂數測試中，它可以達到一個近乎 $O(2E)$ 的理想時間複雜度。

5.6.3 All-Pair Shortest Path

給你一個圖 G，可能是無向也可能是有向圖，求出任一兩點間的最短路徑。
當然我們可以對所有點做單源最短路徑，而且時間複雜度也很好，但我們要介紹簡潔的 Floyd-Warshall's Algorithm。

Floyd-Warshall's Algorithm

Floyd-Warshall's Algorithm 是一個基於動態規劃的演算法。我們用 $D_{i,j,k}$ 代表從 i 到 j，且除了這兩點外路徑上只經過編號在 $[1, k]$ 中的點的最短路徑，可以知道有轉移式

$$D_{i,j,k} = \min\left(D_{i,k,k-1} + D_{k,j,k-1}, D_{i,j,k-1}\right)$$

且邊界條件為如果 $(i, j) \in E(G)$，$D_{i,j,0} = w(i, j)$ 否則 $D_{i,j,0} = \infty$，而最後 i 到 j 的最短路徑即為 $D_{i,j,|V|}$。

Algorithm 16 Floyd-Warshall's Algorithm

```
1: function FLOYD-WARSHALL(G = (V, E): graph, D: Dynamic Programming Array)
2:     for k = 1to |V| do
3:         for i = 1to |V| do
4:             for j = 1to |V| do
5:                 D_{i,j,k} = min(D_{i,k,k-1} + D_{k,j,k-1}, D_{i,j,k-1})
6:             end for
7:         end for
8:     end for
9: end function
```

5.6.4 Exercise

- 有 N 個蘿莉 M 個正太，招募一個人入伍需要 10000 元。蘿莉和正太之間有 R 條關係。每條關 (x, y, d)，表示蘿莉 x 和正太 y 的關係為 d，代表如果其中一人已被你招募，則可以用 $(10000 - d)$ 的費用招募另一人入伍。求將所有人招募的最少花費。(<POJ 3723> [Conscription] , $N, M \leq 10^4, R \leq 10^5$)
- 給你一個圖 $G = (V, E)$，其中每條邊有非 1 即 0 的權值，求一個生成樹使得其所有邊的權重和恰為 K，輸出任意組解或是輸出無解。(<APIO 2008> [Roads] , $|V| \leq 2 \times 10^5, |E| \leq 10^6$)
- 給你一個圖 $G = (V, E)$，你要判斷每一條邊是否必會出現在最小生成樹中，或是必不會出現在最小生成樹中，或是兩者皆不是。(<Codeforces 160D> [Edges in MST] , $|V|, |E| \leq 10^5$)
- 給你一個圖 $G = (V, E)$，求出其次小的生成樹。([次小的生成樹] , $O(|V|^2)$)
- 給你一個圖 $G = (V, E)$，如果限制特定一點在生成樹上的最大度數，求出最小生成樹。([頂點限制度數最小生成樹] , $O(|V|^2)$)
- 給你一個圖 $G = (V, E)$，每個邊 e 上有兩個權值 $a(e), b(e)$，求出一個生成樹 T 使得 $\dfrac{\sum_{e \in T} a(e)}{\sum_{e \in T} b(e)}$ 最小。([最優比率樹] , $O(|V|^2)$)
- 給你 N 個在座標平面上相離的圓，求出從點 $(0, 0)$ 到點 $(1000, 1000)$ 的最短路徑長度，使得路徑不經過任何圓的內部。(<NPSC 2010> [耶誕老人要回家])
- 給你一個圖 $G = (V, E)$，邊上有權代表長度，你要從一個點 v 走到終點 u，不過有一隻很煩人的鱷魚，他會在你每動一次後選一條邊霸佔著不讓你過去，問在最差的情況下你至少要走多少路才能到達終點。(<IOI 2011> [Crocodile])

6 字串

6.1 字串

6.1.1 名詞解釋

觀看前人的編排，他們都先做了一些名詞的解釋，我也仿效一下：

1. 字元 (character)：
   孤單的一個符號。'7','1', '阿', '2', '局', '漬'
2. 字元集 (Alphabet)：
   由字元組成的集合，據說用 Σ 表示
3. 字串 (String)：
   由字元集中的字元構成的序列。"7122"
4. 子字串 (Substring)：
   字串中的一段連續字元。"71" in "7122"
5. 子序列 (Subsequence)：
   字串中不需連續的一斷字元。"72" in "7122"
6. 前綴 (Prefix)：
   一個子字串包含第一個字元。"7", "71", "712", "7122" in "7122"
7. 後綴 (Suffix)：
   一個子字串包含最後一個字元。"2", "22", "122", "7122" in "7122"
8. 字典序 (Alphabetical Order)：
   定義字串間的大小，先定義字元間的大小：' ' < 'a' < 'b' < 'c' < 'd' < ⋯< 'z'
   通常就是照著 ASCII 碼，比較要注意的是空字元比其他都還要小
   接下來從第一個位置一位一位比對，先比對方小的就是比較小的字串
9. 後綴數組 (Suffix Array)：
   將一個字串的所有後綴，照字典序排序後，所得的名次陣列。
   Sa[i]: 第 i 名是第幾個後綴
10. 排名數組 (Rank Array)：
    為後綴數組的逆數組。
    Ra[i]: 第 i 個後綴是第幾名

11. 最長共同前綴 (Longest Common Prefix)：
    兩個字串，從第一位一位一位比對，直到不一樣就停止
    EX: '712221212' 和'712222222' 的最長共同前綴：'71222'

12. lcp(I, J)：
    對於一個字串，他的第 I 個後綴和第 J 個後綴的 LCP 有多長

13. LCP(I, J)：
    對於一個字串，他的第 I 名後綴與第 J 名後綴的 LCP 有多長

14. height$[i]$：
    對於一個字串，LCP$(i-1, i)$

15. h$[i]$：
    對於一個字串，LCP$(\text{Ra}[i]-1, \text{Ra}[i])$

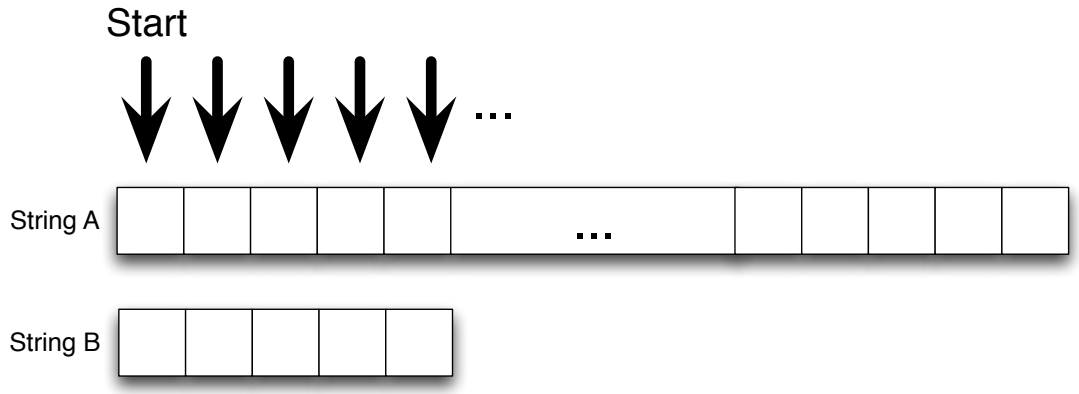
6.1.2 匹配問題 (String Matching)

字串的問題千奇百怪，其中最樸實 (?!) 的就是匹配問題，因為作法的種類多的太誇張了，所以我就挑幾個比較實用的作法來講解。

問題定義 對兩個字串 A 和 B，從 A 裡找出所有的 B。$n = \text{len}(A), m = \text{len}(B)$

天真算法

萬物的根源，枚舉 A 中所有的起點，一位一位跟 B 比對。時間複雜度：$O(nm)$



RK 算法 (Robin-Karp Algorithm)

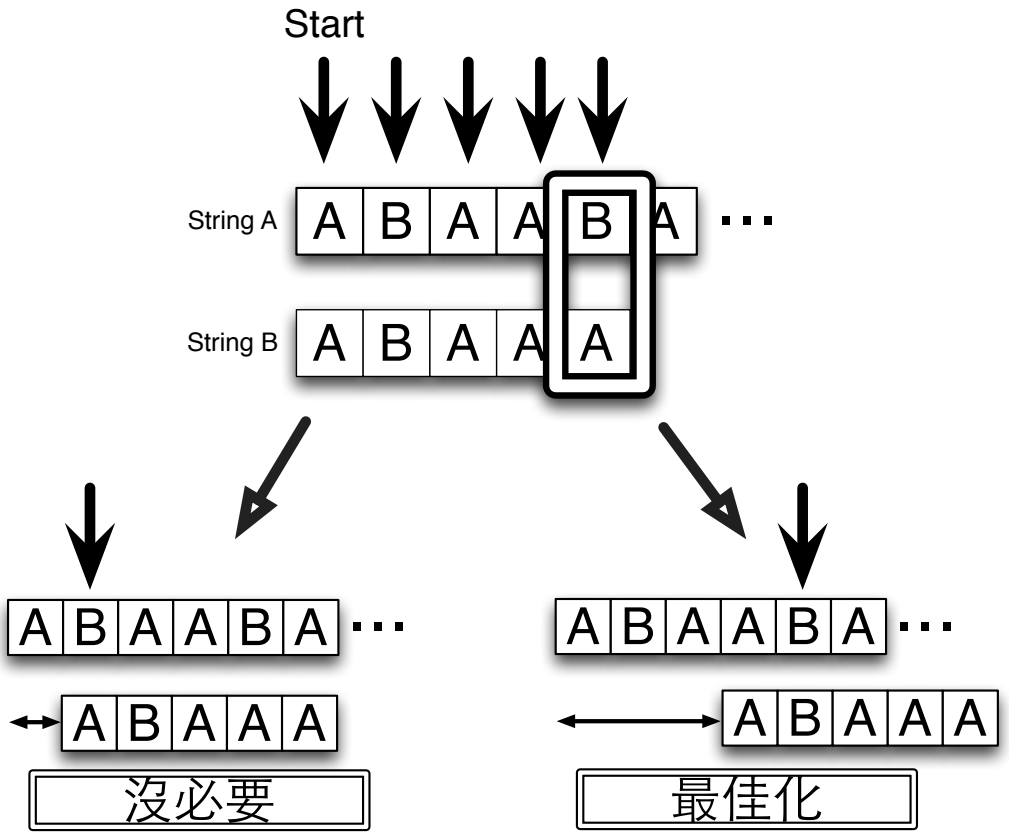
Hanhan 的最愛，他是天真算法的改進，把 A 每 m 個用 rolling hash 做一個 key 值。如果跟 B 的 key 值相同，才要一位一位比對：

$$H = s[0] * X^{m-1} + s[1] * X^{m-2} + \cdots + s[m-2] * X^1 + s[m-1] * X^0$$

但是有些情況下，會發生跟天真算法一樣悲劇的情形，所以可以改成計算多個 Key 值，然後就不再做一位一位的比對了，雖然有些許的危險性在。(所以人稱『RP 算法』) 時間複雜度：$O(n+m)$

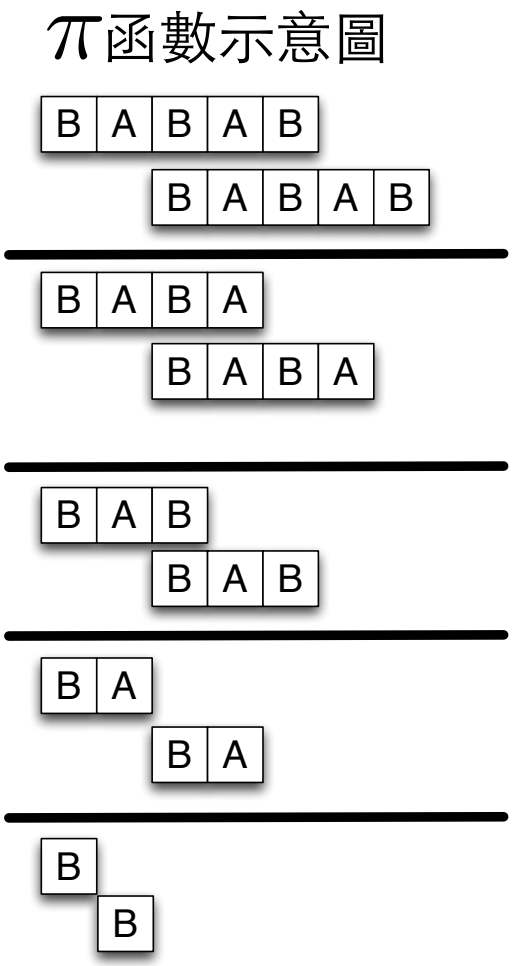
KMP 算法 (Knuth-Morris-Pratt Algorithm)

人總是不斷追求極致，於是發展出神奇的 KMP 算法。接下來讓大家對他有多一分的了解吧！他的複雜度 $O(n+m)$ 且 100% 安全。



這是一個充分利用已有資訊去做出判斷的好算法。如果依照天真算法，你就會做出如上圖左邊的動作，但其實這是沒有必要的，因為你既然已經知道前四格都跟自己相同，你可以『直接』把 String B shift 到一個『確定可以的地方』再比對其下一個字元，又不行則再 shift。也就是先預處理出『自己的『每個前綴』的『後綴與自己的前綴』最長是哪個前綴』的一個神祕函數，稱為 π 函數 (如下圖所示)

假如你有了這個函數，你就可以做出如同上面那張圖右邊的異能動作，不斷的比對直到你配對成功，成功後再一次做出位移的動作，然後繼續比對。你可以知道你要比對的次數就只會有 n 次，但每次比對都會配上 B 字串位移的次數，乍聽之下，好像是 $O(nm)$，但仔細想想後，會發現 B 字串位移的總次數最多只會是 n 次，所以比對複雜度 $O(n)$。最後的問題



在於這個 π 函數是要如何求得，其實求他的方式就跟比對的方式幾乎一樣，增加出一個後，不斷的位移，直到他的下一個跟自己的下一個相同為止，因為是相同的手法，複雜度 $O(m)$。這就是有名的複雜度 $O(n+m)$ 的 KMP 算法，不清楚可以去 Momo's Weird Code 搜尋 KMP 有 code 跟彩色圖解。

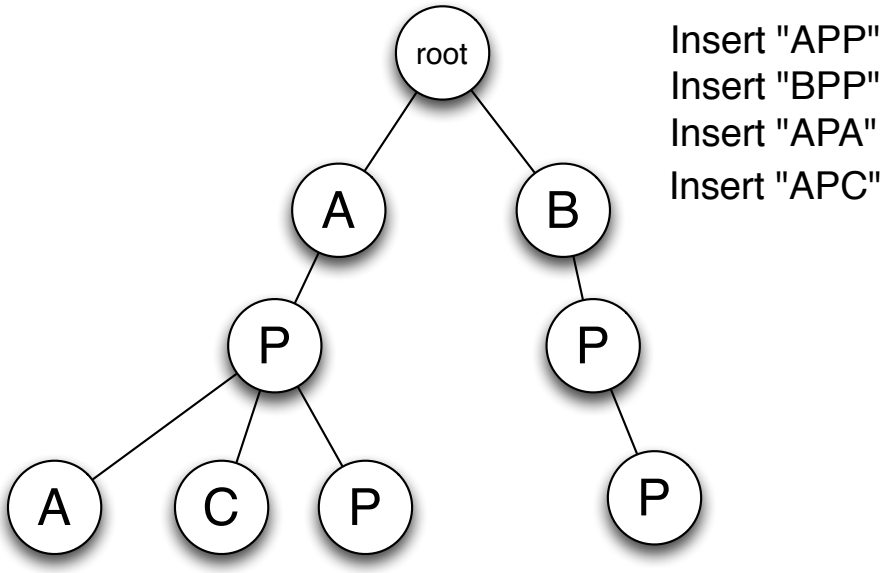
小插曲 — 字典樹 (Trie) 就是用來保存多個字串的資料結構，放個圖相信大家就會了解了，因為著實不難，只是一棵樹而已，不具巧妙之處。

用 Pointer 阿，array 阿都可以實踐，唯一缺點是記憶體用量大，因為他的記憶體用量為輸入字串的總字元數量 $\times 26$ 或 $\times 52$ 之類的，因為對於所有的節點都無可避免的開出無謂的大小 26 陣列。一般來說他可以高效的性能判斷某個字串是否出現過，也可以藉由 LCA 算法，快速的求出兩個字串的 LCP。

Aho-Corasick 算法

這時你碰到一個需要匹配多個字串的問題，就是給你一個字串 A 和多個字串 B_i，要你看有沒有匹配之類的問題。先給出大致的算法構想。定義 $n = \text{len}(A), m = \sum \text{len}(B_i)$。

算法步驟



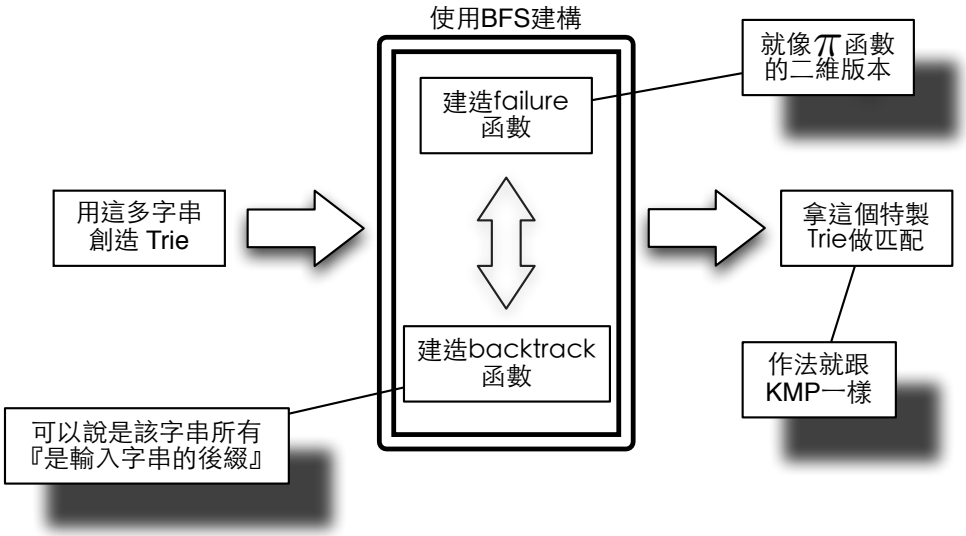
1. 把它給你的多個字串，做成一棵可愛的字典樹，可以做 pointer 版或是 array 版，我比較喜歡的是 array 版，因為我不太會指標，array 版的話就是把節點離散化 (?) 的感覺，pointer 就是記錄記憶體位置那樣。

2. 接下來就是要做 KMP 的 π 函數了，也就是我們這裡稱的 failure function(應該說論文裡稱的，簡稱 f 函數)，其實概念就跟 KMP 一樣啦，只是有一個重點：那就是一定要透過 BFS 進行，為什麼呢？不能用 DFS 嘛？原因是你求得的那個前綴的 f 函數亦必須是已知的。各位還記得 KMP 嘛？如果沒試成，那就得再前往更前面的 f 函數，而且可以知道自己 f 函數的那一層一定比自己那一層淺，所以從 root 開始 BFS 是萬無一失的，時間複雜度 $O(m)$。

3. 與 failure function 同步進行的是 backtrack function(之後簡稱 b 函數)，其實他就是代表你的後綴且『同樣是輸入進來的字串』的函數，因為並不是每一個節點都是輸進來的字串的結尾，其實 b 函數就等於 (『若 f 函數是某字串結尾』? f 函數: f 函數的 b 函數)。所以對於不是某字串的結尾也要做一個 b 函數，這樣編程時將較方便，時間複雜度 $O(m)$。

4. 終於到了最後匹配的步驟了，其實就是 KMP，一模一樣，時間複雜度 $O(n)$。只差在當你發現某個節點是某個字串的結尾時 (也就是匹配成功)，你必須透過 b 函數遞迴一下，因為那個字串匹配成功則他的所有後綴都會匹配成功，可是這樣就會在時間複雜度裡增加了一個 $O(k)$ 代表匹配成功的數量，這是很可怕的，因為如果匹配了太多次，電腦就跑不動了。所以你可以 DP 一下，就是把匹配成功的節點做記錄，到全部都跑完再一次整理。不過還是仍得端看題目，有時候他根本就不在乎有哪些人被匹配到，那麼 b 函數就只是沒有用的渣渣了。

5. 接下來就等著 AC 吧！(或陷入無限 debug 迴圈，code 比上面的多 $==$)



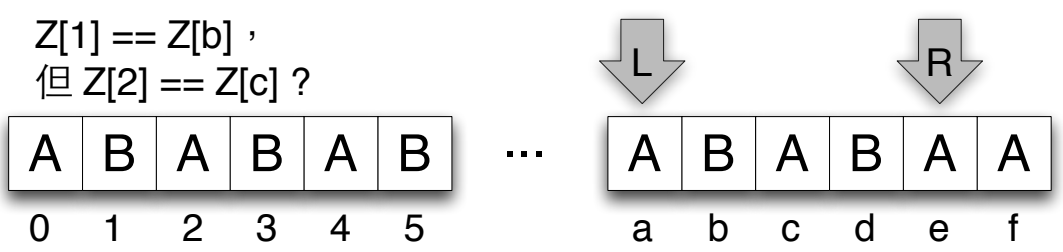
Gusfield Algorithm(俗稱 Z algorithm)

各位是否覺得 KMP 太過於困難，太過於複雜，可以試試看簡單易懂的 Z 算法，這是獻給如我一般每次都記不住 KMP 到底在幹嘛的大家，也是字串匹配的最終章。(code 12 行)

算法用途 回傳一個 Z 陣列，Z[x] 代表從此字串 x 開始的後綴與此字串的 LCP，$O(n)$。(把字串做成 BA，用一下 Z 算法就可以知道有沒有字串匹配了)

其實這個也是充分利用先前資訊的一個演算法，在字串演算法中，最重要的概念就是要『充分利用』，想想看這題可以怎麼做吧！

Hint 1: 朝讓上面的比對指針只會前進不必後退之類思考，這樣才能 $O(n)$。
Hint 2: 說不定要記錄兩個指針！譬如說一個是比對到哪，一個是是哪個後綴比的
Hint 3: 如果一個位置 X 位於之前比過的那段 $[L, R]$ 當中，他是否跟 $X - L$ 相同呢？
Hint 4: 接續 Hint 3，搞不好可以用討論的，就可以解決這個困難



可以知道 $Z[0]$ 是 n，所以從第一位開始比對。其實只要分成三種情形：

1. 要比的後綴根本不在以前比過的範圍內 → 就去比吧！

2. 要比的後綴在以前比過的範圍但長度未知 → 還是去比吧！
3. 要比的後綴在以前比過的範圍但長度已知 → 直接記錄囉！

```
void Z_maker( int z[], char s[], int n ){
    z[0] = n;
    int L = 0, R = 0, i, x;
    for( i = 1 ; i < n ; i++ ){
        if( R < i || z[i-L] >= R-i+1 ){
            R < i ? x = i : x = R+1;
            while( x < n && s[x] == s[x-i] ) x++;
            z[i] = x-i; if( i < x ){ L = i; R = x-1; }
        }
        else z[i] = z[i-L];
    }
}
```

字串匹配章，終。

6.2 SA 數組、LCP 定理

不知道到這個時候，還有沒有人記得前面的名詞介紹，不記得請翻一翻前面。這裡要介紹一個可以求出 SA 數組世界第二快的算法。(想知道最快的是什麼嘛？依據蝴蝶的說法：有興趣自虐的人可以咕狗一下 DC3 算法)

6.2.1 Doubling Algorithm(倍增算法)

$\mathrm{SA}[i]$ = 第 i 名是此字串的哪一個後綴，$\mathrm{RA}[i]$ = 第 i 個後綴是此字串的第幾名 (依據字典序)，大家想到的方法很可能都脫離不了 $O(n^2 \lg n)$ 之類的。像先把所有後綴都拿出來，然後再做個 sort，因為有 n 個後綴所以 $O(n \lg n)$？不，因為任兩個比較的平均時間 $O(n)$，所以仍是 $O(n^2 \lg n)$。

雖然不知道發明這個算法的人是誰，不過他真的突破天際，想到了一個 $O(n \lg n)$ 的算法。像前面所說的，字串題就是要充分利用所擁有的資訊，創造最大的利益 (?) 接下來就來看看他想到的作法究竟是什麼：

前面提到的樸素作法，顯然沒有利用到他們都是同一個字串的這個性質，其實這是很棒的性質，假如你知道每一個一個字的 SA 數組了，是否可以 $O(n)$ 得到每兩個兩個字的 SA 數組呢？如果第一個字一樣就比第二個字，所以其實你可以用兩個一個字組成兩個字，很直覺的性質吧 (?!) 不過還記得 Counting Sort 嘛？用剛剛的性質，你先對第二個字做 CS，接著再對第一個字做 CS，就得到每兩個兩個字的 SA 數組了。透過這個步驟一直合併合併，只需要 $O(\lg n)$ 次就得到最終的 SA 數組了。

這時不知道有沒有人發現，最後一組兩個字並不具有兩個字，這並不是 bug，不過你要記得先預處理掉那些不具有第二個字的字，並把他們都排到第一順位 (因為是依照字典序，而且務必要記得 co 這段)，當然這也是為什麼到最後每一個字串都會是不同長度的原因。

至於想要看實際的 code，可以去看 momo's weird code 4 月 20 日那篇。

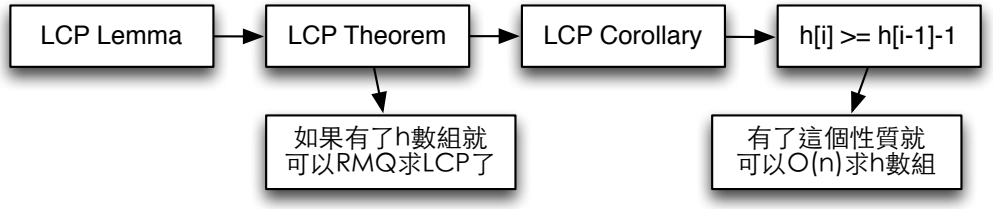
6.2.2 Longest Common Prefix (LCP) 定理

為了要將 SA 數組發揮到極致，我們還要能在很快時間找出任兩個 Suffix 的 LCP。我覺得這段非常有趣但也挺複雜的，要多多複習名詞，並請細細品味 XD

(P.S. 多加一個表示法：$=_x$ 代表兩個字串前 x 位相同)

LCP Lemma

$$\mathrm{LCP}(i,j) = \min(\mathrm{LCP}(i,k), \mathrm{LCP}(k,j)) \quad (i \le j \le k)$$



Pf: 反證法

1. 設 $x = \min(\mathrm{LCP}(i,k), \mathrm{LCP}(k,j))$, 則 $i =_x k$ 且 $k =_x j$, 所以 $\mathrm{LCP}(i,j) \geq x$
2. 設 $\mathrm{LCP}(i,j) > x$, 則 $i[x+1] = j[x+1]$
   因 $i \leq k \leq j$ 且 $i =_x j =_x k$, 故 $i[x+1] \leq k[x+1] \leq j[x+1]$
   因此 $i[x+1] = k[x+1] = j[x+1]$, 那怎麼 LCP 才 x, 矛盾！
3. 由 1. 和 2. 可知 $\mathrm{LCP}(i,j) = x = \min(\mathrm{LCP}(i,k), \mathrm{LCP}(k,j))$

LCP Theorem

$$\mathrm{LCP}(i,k) \leq \mathrm{LCP}(j,k) \quad (i+1 \leq k \leq j)$$

Pf: 數學歸納法

1. $j - i = 0$ 和 $j - i = 1$ 皆成立
2. 若 $j - i = m$ 成立，則 $\mathrm{LCP}(i,j) = \min(\mathrm{LCP}(k-1,k)) \quad (i+1 \leq k \leq j)$
   由 LCP Lemma,
   $\Rightarrow \mathrm{LCP}(i-1,j) = \min(\mathrm{LCP}(i-1,i), \mathrm{LCP}(i,j))$
   $\Rightarrow \mathrm{LCP}(i-1,j) = \min(\mathrm{LCP}(k-1,k)) \quad (i \leq k \leq j)$
   $\Rightarrow j - i = m + 1$ 成立

而 height 數組可以透過 h 數組求得，這也是我們為什麼要弄出一個 h 數組去折磨自己，全部都是為了可以用 RMQ (預處理 $O(n \lg n)$ / 詢問 $O(1)$) 求出任兩個後綴的 LCP 阿！

LCP Corollary

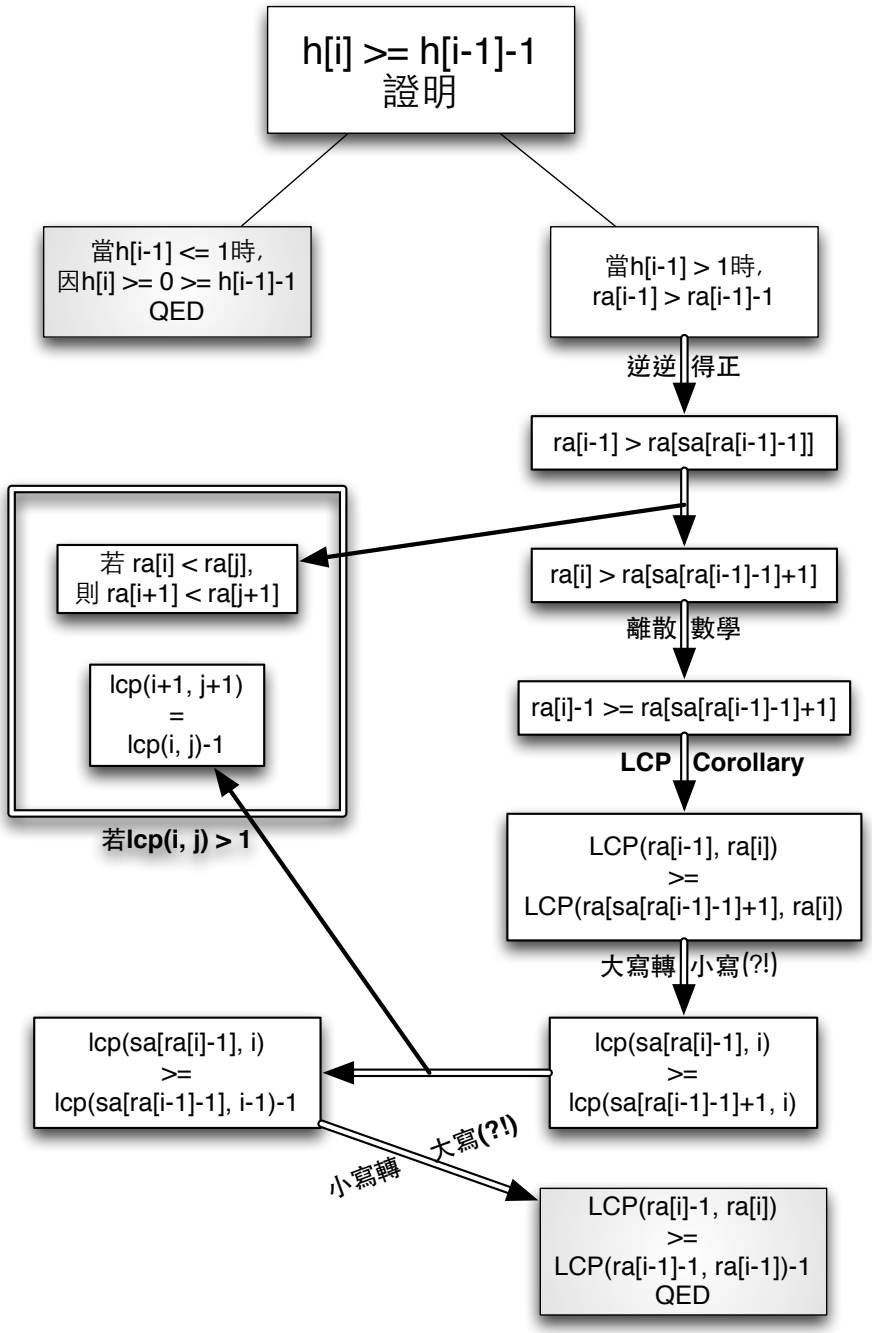
$$\mathrm{LCP}(i,k) \leq \mathrm{LCP}(j,k) \quad (i \leq j \leq k)$$

Pf: 遞推

$$\mathrm{LCP}(i,k) = \min(\mathrm{LCP}(i,j), \mathrm{LCP}(j,k)) \leq \mathrm{LCP}(j,k)$$

h 數組的性質

$$\mathrm{h}[i] \geq \mathrm{h}[i-1] - 1$$



因為你只需要從 $\mathrm{h}[i-1]-1$ 開始比對，小小的估一下複雜度，便會發現他是線性的，之後再透過 $\mathrm{height}[i] = \mathrm{h}[\mathrm{SA}[i]]$，我們便可以以 $O(n)$ 的時間求出 height 數組！

因為你必須要先求出 SA 數組，所以要求出 height 數組的整體的複雜度 $O(n \lg n)$，接下來要建一個 sparse table，時間複雜度 $(n \lg n)$，當建完 sparse table 之後你就可以 $O(1)$ 求出

任兩個後綴的 LCP 了！

總體複雜度 $O(n \lg n)$，但是常數非常大就是了。

6.2.3 後記

如果有興趣的話，可以參考 2004 年許智磊寫的後綴數組，我是從那裡學會的，不過我寫的份量跟他差不多。他後面有附上兩個例題的題解，但是第一個用前面教的 Aho-Corasick 算法更快，而第二題則可以參考參考。

最後再套許智磊說的話作為完結：『算法和數據結構都是死的，而運用他們的人才是真正的主角』。XD

7 $\lg N/\sqrt{n}$ 結構

7.1 概論

線段樹是一種樹狀資料結構，能支援區間的動態修改與查詢，是在競賽中十分有用的資料結構之一。由於線段樹有許多的變化，因此常常遇到可以使用線段樹的題目，另外線段樹在某些情況下也可以部分取代平衡二元搜尋樹。
線段樹是一棵二元樹，可以快速修改 (update) 與查詢 (query) 區間中的值。為了方便理解線段樹的結構及操作，我們先討論一個簡單的問題：

Dynamic Range Minimum Query(RMQ)
給定一個序列 $S[1,n]$, 以及以下兩種操作：
(1) 將 $S[i]$ 改為 k
(2) 查詢 $S[i,j]$ 中的最小值

7.1.1 塊狀數組

一個最簡單的想法就是每次直接 $O(1)$ 修改，查詢時 $O(n)$ 尋找最大值。然而我們可以稍微提高修改的複雜度，以加快查詢的效率。首先將數列分成 $\sqrt{n}$ 段，並分別記錄每段的最小值，修改時花 $O(\sqrt{n})$ 的時間更新覆蓋被修改元素的那一段。查詢時若詢問的區間恰好覆蓋某段，則可直接取得該段的最小值，不足的再逐一查詢。由於每此查詢最多覆蓋到 $O(\sqrt{n})$ 段以及 $O(\sqrt{n})$ 個不足一段的元素，因此查詢也可在 $O(\sqrt{n})$ 的時間內完成。這樣的結構即為塊狀數組。

7.1.2 線段樹

有了塊狀數組的想法後，很自然地會想到，是否能多加幾層使得速度更快。事實上線段樹就是用一個陣列紀錄每 2 個元素的最小值 (稱為第 2 層，原始序列稱為第 1 層)，並遞迴記錄該陣列每 2 段之最小值 (分別為第 3, 4 ... 層) 所形成最的一棵二元樹。因為每一層陣列的格數約為前一層的一半，因此其深度為 $O(\lg n)$。

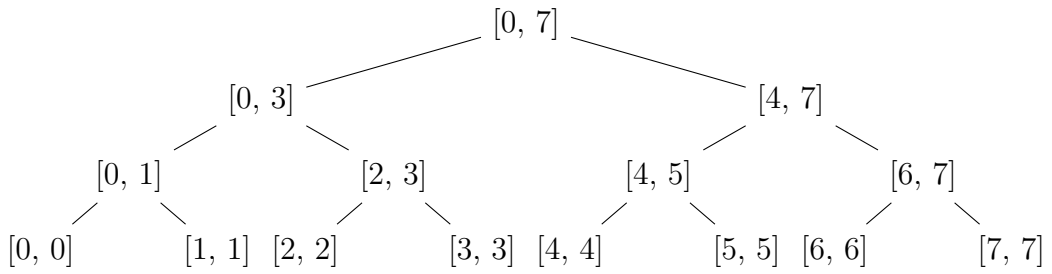


Figure 7.1: 線段樹及其節點之區間

觀察後可發現由於樹的深度為 $O(\lg n)$，修改時每個元素最多被 $O(\lg n)$ 段覆蓋，因此修改此值時最多只會影響 $O(\lg n)$ 段的最小值。查詢時，由於每一層最多會被完整覆蓋 2 段，所以總覆蓋最多 $O(\lg n)$ 段, 也就是查詢時最多只需檢查 $O(\lg n)$ 個段，複雜度亦為 $O(\lg n)$。

7.1.3 二元搜尋樹

若我們是離線處理問題，即已知道所有輸入的資料，則可以使用線段樹代替二元搜尋樹來使用，其優點是樹的結構能完全平衡。其做法是目前已出現數字之區間最小值，Insert 操作讓其從序列中出現，並更新所有覆蓋此點的區間之最小值，Delete 操作則和 Insert 操作相似.。

7.2 實作

線段樹可以簡單的使用陣列來儲存，將根節點編號為 1，而標號 i 的節點其左子節點標號為 $2i$，右子節點則為 $2i+1$。
其一般的操作如下所示:

```
1:  procedure OPERATION(l, r, i, T)
2:      if T[i].l ≠ l ∨ T[i].r ≠ r then
3:          m ← ⌊(T[i].l + T[i].r)/2⌋
4:          if r ≤ m then
5:              OPERATION(l, r, 2i, T)
6:          else if l ≥ m then
7:              OPERATION(l, r, 2i + 1, T)
8:          else
9:              OPERATION(l, m, 2i, T)
10:             OPERATION(m, r, 2i + 1, T)
11:         end if
12:     end if
13: end procedure
```

7.3 操作

在第 1 節間單的介紹了單一修改及區間修改的功能，事實上線段樹的功能還包含修改整個區間，以下舉兩種操作為例：

7.3.1 將一段數字加減一個值

對於樹上的每個節點我們多記錄一個 delta，代表更新值，修改時類似區間查詢，若有完整覆蓋的就直接更新他的 delta 值，否則就對於其左右節點遞迴下去更新。詢問時要特別注意，若某個節點的祖先中有節點的 delta 值不為 0 則代表他也要加上相同的 delta 值，因此必須要計算其祖先的 delta 值總和並加到 min 值上才是真正的最小值。

7.3.2 將一段數字全部改成一個數

其做法和前一個類似，我們可以同樣的多紀錄一個 cover 值，如果 cover $\neq \infty$ 則代表該節點以下所有值均為 cover。但是這樣會發現一個問題：如果我們先將 [1,2] 改成 1，再將 [1,1] 改成 2，這時候我們節點 [1,2] 的 cover 值會是 1，若我們詢問 [1,2] 時會以為 [1,2] 以下全都是 1，就發生錯誤了。解決這個錯誤的方法稱為「延遲標記」，當我們要在一個 cover $= k$ 的區間下改變值的時侯，那我們就先把它的 cover 值改成 ∞ 並把它的兩個子節點的 cover 值改為 k，即將這個區間拆成兩半，再繼續改變的動作，就可以避免以上情形發生。

至於線段樹的其他操作則請自行思考實作方式，其實大都不外乎在節點上多紀錄幾個值。

7.4 推廣與變形

7.4.1 Binary Indexed Tree

BIT 是線段樹的一種變形，他的實作較方便，但相對的功能也較侷限，其所支援的操做僅以下三種：

1. 單一修改, 區間 $(0, k]$ 查詢
2. 區間 $(0, k]$ 修改, 單一查詢
3. 可藉由排容原理所得到的值 (如總和)

因為其所支援的只有從頭開始的區間，所以只需維持一半的區間，則可將線段樹刪去一半的節點簡化為樹狀數組。此時會發現所維持的所有區間之後端點皆相異，故我們可以以後端點

的位置來表示一個區間，則可直接以一個索引值為 $[1,n]$ 的陣列來維持所有區間的結果。歸納其記錄的區間我們得到一個公式，索引值為 k 的節點，其所維護的區間是 $(k-2^i,k]$，i 稱為 k 的 lowbit，即為二進位表示中從最低位數來第一個非 0 的位數，lowbit 可以由以下的方法算出:

$$2^l = k\&(-k)$$

查詢 $(0,k]$ 區間時，其結果便是 $(k-\text{lowbit}(k),k]$ 和 $(0,k-\text{lowbit}(k)]$ 合併後的結果。前者恰好就是 BIT 在 k 所維護的區間，而後者只需遞迴下去計算即可。對於單一操作，則可以發現 $k_0=k, k_j=k_{j-1}+\text{lowbit}(k_{j-1}), j\in\mathbb{N}$ 恰好就是覆蓋 k 的所有區間。由於只須維持 $(0,k]$ 的區間，故兩者的複雜度皆為 $O(\lg n)$。

Algorithm 17 Single Update on Binary Indexed Tree

```
1: procedure UPDATE(i, T, n, v)
2:     while i ≤ n do
3:         T[i] ← T[i] + v
4:         i ← i+ lowbit(i)
5:     end while
6: end procedure
```

Algorithm 18 Range Query on Binary Indexed Tree

```
1: procedure QUERY(i, T)
2:     r ← 0
3:     while i > 0 do
4:         r ← r + T[i]
5:         i ← i− lowbit(i)
6:     end while
7:     return r
8: end procedure
```

7.4.2 K 維線段樹

對於一個 2 維的線段樹，其每個節點是一棵樹，第一層樹紀錄 x 坐標，而此 x 樹的節點是一個紀錄 y 坐標的樹，而 k 維則依此類推。不過我們通常不會用超過 2 維 (甚至連 2 維都很少用) 這是由於空間複雜度的限制。2 維線段樹有個致命傷，便是無法進行區塊修改 + 區塊詢問，因為在二維以上的區間，會出現兩區塊在不同維坐標互相包含的情況，此情形會造成同一個點被許多不相為父子節點的區間包含，因此無法實作。線段樹有另一個 generalization 叫做 Quadtree 和 Octree，它們操作的時間複雜度會比較高一些。

至於樹狀數組的高維度則比較容易實作，以二維為例只須建立一個二維陣列 S，其中 $S[i][j]$ 維護 $(i-\text{lowbit}(i),i]\times(j-\text{lowbit}(j),j]$ 區塊之值，其操作和一維的類似，只是複雜度變為 $O(\lg^2 n)$

7.5 應用

線段樹在實際應用上常會結合其他資料結構或演算法，以下介紹幾種常見的例子。

7.5.1 離散化 (Discretization)

有時候我們要紀錄的數字範圍太大，超出記憶體所能負荷，這時後，需要用一個函數 $f : \mathbb{N} \mapsto \mathbb{R}$ 來維持需要的位置，用小的值來當作索引位置。

7.5.2 掃描線 (Sweep Line)

線段樹經常結合掃描線演算法使用, 所謂掃描線, 就是用一條直線，朝與直線垂直的方向掃過去。可見，掃描線需要結合排序，得知所有點特定方向的坐標後才能得知所有點的順序。另外，掃描線也使用了離散化的概念，畢竟我們不能真的將整個 $\mathbb{R}$ 中的所有點都掃過。掃描線可以將線段樹降一維，是個十分有用的工具。

7.5.3 時間戳記 (Time Stamp)

在線段樹上，如要進行區段覆蓋時，可以用時間戳記紀錄更新時間，以便瞭解覆蓋的時間順序，才不會造成查詢時子節點有值，父節點也有值而不知道取用哪個的窘境。

7.5.4 樹上詢問 (Tree Queries)

將整棵樹 traverse 一遍，並紀錄所有節點 i 的進入時間 $\text{start}(a)$ 和離開時間 $\text{end}(a)$，就可以將一顆樹壓扁成一個序列，將樹上詢問轉為區間詢問。某節點的區間 $[\text{start}(a), \text{end}(a)]$ 即代表整棵由 a 為 root 的 tree。

7.6 Exercise

- 平面上有 N 個點，每個點有權重 w_i，輸出由點組成的序列，滿足 x, y 坐標皆非嚴格遞減的權重和最大值。(<POI XII> [The Bus] , $N \leq 10^5$)
- 給 $1, 2, 3$ 維坐標上 N 個點以及整數 R，輸出有幾個點對的曼哈頓距離不超過 R。(<IOI 2007> [Pairs])
- 給你船上 N 根排成一直線桅桿的高度 h_i 和需要掛的船帆數量 k_i，一根桅桿可以在高度為 $1, 2, \cdots, h_i$ 處掛上帆，而每面帆的反阻率是在其後方且與這面帆同高度的帆的數量，輸出在適當的安排下，反阻率總和的最小值。(<IOI 2007> [Sails] , $N \leq 10^5$)
- 給你平面坐標上 N 個長寬皆平行坐標軸的矩形，輸出所有矩形聯集的面積。($O(N \lg N)$)
- 現在有尺寸為 $[1, N]$ 的溜冰鞋各 k 雙，每個足尺寸為 r 的人可以穿尺寸界在 $[r, r + d]$ 的溜冰鞋，每個事件描述時間 i 有 r_i 個尺寸為 x_i 的人來或是離開，輸出每個事件後是否有方法分配溜冰鞋使得每個人都恰能分到一雙可穿的溜冰鞋。(<POI XVI> [Ice Skates] , $O(N \leq 10^5)$)
- 在 $M \times N$ 的平面上有 P 個矩形障礙，每個障礙有屬性 x_1, y_1, x_2, y_2, c 代表四邊的座標以及移除成本。若不能移除任何障礙，輸出可以最大的正方形。若有預算 B，輸出用這些預算移除某些障礙後最大能找到多大的正方形。(<IOI 2008> [Pyramid Base])
- 有一個數列 $A = \{a_1, a_2, \cdots, a_N\}$，請問你有幾對整數數對 (l, r)
使得數列 $B = \{a_1, a_2, \cdots, a_l, a_r, a_{r+1}, \cdots, a_N\}$ 的逆序數對不超過 K。(<Codeforces 220E> [Little Elephant and Inversions] , $N \leq 10^5$)
- 給你一個正整數數列，接著有 Q 比詢問問你 l_i 到 r_i 間不同的正整數有幾個。(<NTU 1419> [Fiasco on Fresco] , $N, Q \leq 10^5$)

8 進階主題

8.1 DP 優化

在前面的章節我們提到了用動態規劃 (Dynamic Programming) 來解決各式各樣的問題，而在這裡我們要討論一些神奇的優化方式。

8.1.1 單調隊列優化

先回憶一下雙向佇列 (Deque) 這個資料結構，他可以支援兩種操作：

- 從兩端 push 新的元素進去。
- 從兩端 pop 元素出去。

可以知道 Deque 完全涵蓋了 Stack 和 Queue 的操作。
而 Deque 通常用來優化 DP 中取極值的動作，尤其是要查詢多個範圍的極值，且這些範圍滿足一定的單調性，我們將在下面一一討論。

$$Ans_i = \min_{L_i \le j \le R_i} v_j \quad ; \quad L_i \le L_{i+1}, R_i \le R_{i+1}$$

簡單來說我們要依序查詢序列的某些區間的最大值，而且這些序列的區間左界和右界都是遞增的。假設序列的長度為 N, 如果我們對於每個區間都直接掃過一遍，時間複雜度會到 $O(N^2)$。
但仔細觀察可以發現，如果存在兩個元素 v_k, v_h 使得 $v_k \ge v_h$ 且 $k < h \le R_i$，那麼對於第 i 筆之後的詢問，v_k 絕不可能是區間的最小值。因此我們可以維護 Deque 從前面到後面是遞增的，即加入元素時保持單調性而從後面 pop 出元素，而當要查詢的時候先從前端 pop 出過期的元素後，極值即為 Deque 中最前端的元素。

$$Ans_i = \min_{L_i \le j \le i} v_j + (d_i - d_j) \quad ; \quad L_i \le L_{i+1}$$

對於這個情況，如果我們令 $a_i = v_i - d_i$，原本的式子就可以改寫成

$$Ans_i = \min_{L_i \le j \le i} a_j + d_i$$

而因為對於同一個 i 來說 d_i 是固定的，這樣就轉化成第一種情況了。這個式子可以讓我們加速背包問題的 DP，回憶一下背包問題的 DP 式，假設一個物品的重量為 w，價值為 v 且有 c 個：

$$D(n, m) = \max_{0 \le k \le c; 0 \le m - wk} D(n-1, m - wk) + vk$$

這個式子可以改寫為

$$D(n, wi + r) = \max_{0 \le j \le i} D(n-1, wj + r) + v(i - j)$$

可以看出 $D(n-1, wj+r)$ 即為 v_j，v_i 即為 d_i。這可以讓我們在 $O(WN)$ 的時間解決背包問題，其中 W 是背包的耐重，N 為物品的數量，是一個與同一個物品的數量 c 無關的算法。

$$Ans_i = \max_{L_i \le j \le i} d_j t_i + v_j \quad ; \quad L_i \le L_{i+1}, d_i \le d_{i+1}, t_i \le t_{i+1}$$

這個情況即是所謂的斜率優化，對於一個 j，我們可以把每一個 $d_j t_i + v_j$ 看做是的一條直線 $y = d_j x + v_j$，這些線的斜率都是遞增的，而在我們加入一條條的直線，維持凸包的單調性，也就是說我們在加入新的一條線 l 時，假設 Deque 中最後兩條線為 l_1, l_2，如果 l_1 超過 l_2 的 x 座標比 l 超過 l_2 的 x 座標大，那我們直接將 l_1 pop 出，而每次查詢從前面 pop 出過期或是已經被後面的線超過的元素。

8.1.2 四邊形優化

四邊形優化是一個非常恐怖的東西，當你對 DP 的熱愛程度爆表了可以考慮研究一下。首先我們先定義兩種 DP 問題。

1D/1D DP

這種式子的標準式

$$D_i = \min_{i \le k < i} D_j + w(j, i)$$

簡單來說就是 D_i 會它前面得出的答案 D_j 再加上一個轉移的函數 $w(i,j)$ 中的最小值。

1D/2D DP

標準式

$$D_{i,j} = \min_{0 \le k < i} D_{i,k} + D_{k+1,j} + w(i, j) \quad ; \quad D_{i,i} = 0$$

這種 DP 式的意義大概是一個區間 $[i,j]$ 的答案會是將這個區間分成兩塊 $[i,k], [k+1,j]$ 的分法中的最小值，再加上一個與 k 無關的權。

四邊形單調性

我們將單調性分成兩種。

- 凹四邊形單調性
  如果對於任何 $a < b, c < d$ 且 $F(a,c) \le F(b,c)$，就有 $F(a,d) \le F(b,d)$。
- 凸四邊形單調性
  如果對於任何 $a < b, c < d$ 且 $F(a,c) \ge F(b,c)$，就有 $F(a,d) \ge F(b,d)$。

但直接驗證 F 滿足這個性質往往是不容易的，因此我們常用以下不等式檢驗。

- 凹四邊形不等式
  如果對於任何 $a < b, c < d$ 都有 $F(a,c) + F(b,d) \geq F(a,d) + F(b,c)$，則 F 滿足凹四邊形單調性。

- 凸四邊形不等式
  如果對於任何 $a < b, c < d$ 都有 $F(a,c) + F(b,d) \leq F(a,d) + F(b,c)$，則 F 滿足凸四邊形單調性。

更進一步的結果，我們只需要檢查 $F(i,j) + F(i+1,j+1)$ 和 $F(i+1,j) + F(i,j+1)$ 的大小關係即可。

1D/1D 凹性優化

回憶一下 1D/1D 的 DP 式：

$$D_i = \min_{i \leq k < i} D_j + w(i,j)$$

如果我們令 $F(j,i) = D_j + w(j,i)$，代表用 j 轉移 i 的花費，而 D_i 就相當於 $F(i,k)$ 中的最小值。
此時可以知道如果 $w(j,i)$ 滿足凹四邊形單調性，$F(j,i)$ 也會滿足凹四邊形單調性。代表對於 $i, i+1, j, j+1$ 來說，一旦 $F(j,i) \leq F(j+1,i)$，也就是說如果用 j 來轉移 i 較 $j+1$ 來轉移 i 好的話，那麼必有 $F(j,i+1) \leq F(j+1,i+1)$，即接下來用 j 轉移 $i+1$ 也會較用 $j+1$ 來的好！換句話說如果我們用 k_i 代表用哪一個 j 值來轉移 i 會得到最小的值，必有 $k_i \geq k_{i+1}$。利用這個單調性，我們便可以用特殊的方法加速。我們使用 Stack 維護當前最佳解，對於 stack 裡的元素 s 我們紀錄 (L,R,p)，代表對於所有 $L \leq i \leq R$，我們用 $j = p$ 來轉移到 i 最佳。因此當我們要求 D_i 時只需先從 Stack pop 出過期的元素 (即 $R < i$)，之後把 Stack 的 top 元素計算 $F(p,i)$ 即可。而當我們要將 $j = i$ 加入 Stack 時，假設 Stack 的 top 元素為 s，有幾種情況：

1. $F(i, s.L) \geq F(s.p, s.L)$
   這代表用 p 轉移完勝 i，因此我們保留 s 在 Stack 中。

2. $F(i, s.R) \leq F(s.p, s.R)$
   與上面相反，這代表用 i 轉移完勝 p，因此我們直接將 s 給 pop 出。

3. $F(i, s.L) \leq F(s.p, s.L), F(i, s.R) \geq F(s.p, s.R)$
   這時候代表存在一個界線，使得在界線前 i 較 p 好，而在後面 p 較 i 好。由單調性我們可以使用二分搜找出這個界線 L'，並將 s 更新為 (L', R, p)。

最後我們再將 i 加入 Stack 中，即加入 $(i+1, s.L-1, i)$，便完成更新了。(當然有些時候我們根本無需加入，如 $s.L - 1 < i + 1$ 時。複雜度為 $O(N \lg N)$。

1D/1D 凸性優化

與凹性相反，我們用 k_i 代表用哪一個 j 值來轉移 i 會得到最小的值，此時則為 $k_i \le k_{i+1}$。這時候我們也用和剛才類似的方法優化，只不過我們將 Stack 改為 Deque，取值的時候先從前端 pop 出過期的元素，而加入新的值時，我們就從後邊刪除元素，最後一樣二分搜出確切位置。

2D/1D 凹性優化

2D/1D 凹性優化並沒有特別特殊的性質，我們可以考慮枚舉 i 後每次視為 1D/1D 的問題。

令 $F_i(j') = D_{i,i+j'}$，此時有

$$F_i(j') = \min_{0 \le k < j'} F_i(k) + u_i(k, j')$$

其中

$$u_i(k, j') = D_{i+k+1, j'-k-1} + w(i, i+j')$$

只要證明 $u_i(k, j')$ 有凹單調性，便可用前面所講的方法做到 $O(N^2 \lg N)$ 的複雜度。

2D/1D 凸性優化

相較於 2D/1D 凹性優化，2D/1D 凸性有更多的性質，因此有更好的優化方式。首先 2D/1D 凸性在驗證上也比較容易，我們有以下的引理：2D/1D 的 DP 式為

$$D_{i,j} = \min_{0 \le j < i} D_{i,k} + D_{k+1,j} + w(i,j) \quad ; \quad D_{i,i} = 0$$

如果 $w(i,j)$ 為凸單調性的且 $w(i,i+2) \ge \max(w(i,i+1), w(i+1,i+2))$，則 $D_{i,j}$ 也會符合凸單調性。

而且 2D/1D 凸性還有一個強大的性質：令 $K(i,j)$ 為使 $D_{i,j}$ 達到最小值的 k 值，即 k 為使 $D_{i,k} + D_{k+1,j} + w(i,j)$ 最小者，則有 $K(i,j-1) \le K(i,j) \le K(i+1,j)$。

也因為以上定理，我們 DP 時可以從 $j - i = 1, 2, \cdots$ 的順序開始 DP。當我們在求 $DP_{i,j}$ 時我們只需枚舉 $K(i,j-1)$ 到 $K(i+1,j)$ 即可，故對於所有 $j - i = c$ 的狀態，將他們都求出所需枚舉的狀態數只有：

$$\sum_{0 \le i < N-c} K(i+1,j) - K(i,j-1) = K(N-c,N) - K(0,c-1) + N - c = O(N)$$

故求出所有 DP 值只需要 $O(N^2)$。

8.1.3 插頭 DP

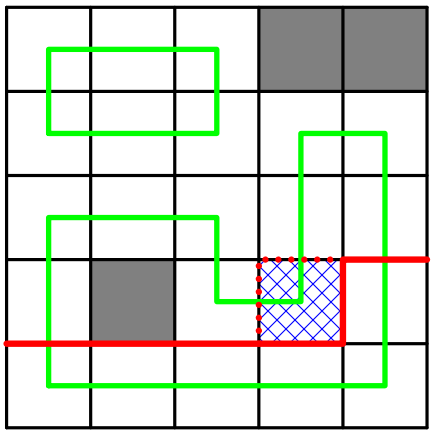
根據某篇文章，他又稱作「基於連通性狀態壓縮的動態規劃」。

插頭 DP 通常用來處理 $m \times n$ 方格的一些組合計數，尤其是計算不同的某種特定路徑覆蓋的

方法數。

就拿最基本的來舉例，假設我們有一個 $m \times n$ 的方格，有些方格可以通行有些不能，求用數個漢米頓圈覆蓋所有可以通行的方法樹有多少種？

而插頭 DP 的想法即是從右到左，從下到上依序討論每一個方格內路徑的樣式，即路徑要從方格的上下左右哪個相鄰方格進來/出去。雖然說我們要求漢米頓圈覆蓋覆蓋的方式，但我們可以想像成，我們先用路徑的方式討論，只要最後能把路徑「接起來」即可。



如圖所示，假設我們正在討論內部為交叉線條的那個方格，而我們 DP 的狀態便紀錄那條粗線的每個線段，是否有「插頭」，即是否有路徑通過那個位置，而當我們要遞推到下一格時，可以知道格子內路徑的樣式可以為那些只和格子右邊以及下邊的插頭有關，因此我們只需枚舉一些情況就可以了，而如果格子是不能通行的，那相當於右邊以及下邊都不能有插頭。

因此數個漢米頓圈覆蓋我們只需用 2 進位即可儲存狀態，但隨著題目越複雜，存的狀態可能也會越複雜，比如題目如果要求只用 1 個漢米頓圈覆蓋，那就需要用 3 進制儲存。詳細的話有興趣可以研究看看。

8.1.4 Exercise

1. 有 N 棟建築物，求出有多少個連續 M 棟建築的區間，使得區間內最高建築與最矮建築高度差不超過 k 公尺。(<TIOJ 1566> [簡單易懂的現代都市] , $N \leq 10^6$)
2. 求出一顆 N 個點的頂環樹上的最長路徑的長度。(<IOI 2008> [Island] , $N \leq 10^6$)
3. 給你一個序列 $x_1, x_2, \cdots, x_n$，你要將這個序列分成數塊，使得所有區塊的戰鬥力總和最大，假設一個區塊中所有數字的總合為 x，區塊的戰鬥力為 $Ax^2 + Bx + C$。(<APIO 2010> [Commando] , $N \leq 10^6; -5 \leq A \leq -1; |B|, |C| \leq 10^7; 1 \leq x_i \leq 100$)
4. 給你一個邊框，滿足 (1) 邊框可分為上邊框和下邊框。(2) 兩個邊框的起點為 $(0, 0)$，終點為 (x, y)。(3) 邊框是一條折線，即由 N 條水平線和鉛直線交替而成。(4) 上邊框永遠在下邊框上面。請你找出這個邊框中長寬平行於 x, y 軸的最大矩形。(<TIOJ ???> [超大?? 設置] $N \leq 10^5$)
5. 給一條線上的 N 個點，請你選出其中的 k 個點，使得所有點到這 k 個中最短的距離的總和最小。(<TIOJ 1449> [郵局設置問題] $N, k \leq 1000$)

6. 現在有一堆烏龜堆疊成一堆，由下到上每隻烏龜都有為和度 x_i，現在你可以選一段連續的烏龜融合起來：

   (a) 最多只能把 R 隻烏龜融合成一段。

   (b) 把一堆烏龜模型融合起來，他們的違和度會相加。

   (c) k 個模型合出的模型會有強度 k^2 的違和光芒。

   (d) 一隻違和度 x 的烏龜放在第 m 層會有 $c*(m-1)$ 的違和度。

   (e) 烏龜塔的違和度是所有烏龜的違和度減掉所有違合光芒的強度。

   求出烏龜塔的違和度的最大值 (<TIOJ ???> [烏龜疊疊樂] $N, R \leq 5 \times 10^5$)

7. 有一個 $M \times N$ 的方格，有些格子可以通行有些則不能，求一筆畫不重複經過格子並通過所有可通行的方格，再回到起點的方法數有多少種。

8. 有一個 $M \times N$ 的方格，有些格子可以通行有些則不能，求一筆畫不重複經過格子並通過所有可通行的方格的方法數有多少種。

8.2 其他的主題

8.2.1 最近共同祖先 (LCA)

最近共同祖先 (LCA) 要求在一棵樹上的其中兩點，他們的一個共同祖先，使得這個祖先的深度最深，而詢問通常都是多筆的。假設樹上有 N 個點，詢問數有 Q 個，那如果最普通的做法會得到 $O(QN)$ 的時間複雜度，在 Q, N 都很大的時候我們顯然需要更好的方法。

Tarjan's Algorithm

在許多領域都有 Tarjan 的演算法，在這裡也不例外。

Tarjan's Algorithm 是一個離線的演算法，亦即必須先將所有詢問讀入才開始計算。其原理為：考慮一個節點 v，對於其兩個不同子節點 u, w，u 的某個子孫和 w 的某個子孫的 LCA 必為 v。因此我們可以對樹 DFS，當我們 DFS 完一個節點的某個子樹後，就把這些點和 v union 起來，這個動作讓我們想到可以使用並查集優化。詳細的演算法在下面，時間複雜度為 $O((Q+N)\alpha(N))$。

Algorithm 19 Tarjan's LCA

```
1:  function DFS(Current vertex : v)
2:      ancestor[find(v)] ← v
3:      for each child vertex u of v do
4:          DFS(u)
5:          Union(u,v)
6:          ancestor[find(v)] ← v
7:      end for
8:      visited[v] ← true
9:      for each query LCA(v, w) do
10:         if visited[w] = true then
11:             LCA(v, w) ← ancestor[find(w)]
12:         end if
13:     end for
14: end function
```

樹的壓平與 RMQ

這裡我們要介紹一個將樹壓平，變成一個序列的方法。在一開始序列為空，接著我們從樹根開始 DFS，每次進入一個節點時我們就將這個點加到序列的最後端，而每次出去一個節點的時候，我們把他的父節點加到序列的最後端。對於一棵有 N 個節點的樹我們便會得到一個長度為 $2N-1$ 的序列。而可以發現，對於任兩個節點 u,v，假設他在序列中出現的位置分別在 i,j(如果出現多次任取一個)，那麼 LCA(u,v) 便會是序列在範圍 $[i,j]$ 中深度最小的一個點。因此我們便將 LCA 轉化為 RMQ 問題了，用一般線段樹的做法可以得到一個 $O(N \lg N)$ 的作法，事實上因為序列中相鄰兩點的深度差最多是 1，因此這其實一個特殊的 $\pm$1RMQ 問題，可以用特殊的方法做到 $O(N)$ 的時間複雜度，如有興趣可以自行研究。

倍增法

倍增法式一個巧妙的方法，對於每一個點，記錄其第 $1, 2, \cdots, 2^k$ 祖先為哪個節點，這樣如果我們要求 LCA(u,v)，我們便可以先將 u,v 往上調整至同一深度，之後再用類似 2 分搜的方式求出他們最近的共同祖先。

8.2.2 比率二分搜

有時候我們要處理最優比率的問題，比如說要求 $\frac{f(v)}{g(v)}$ 的最大值，這時候因為有分數往往不好處理。因此通常我們會用二分搜來將題目改成判斷性問題，首先我們設定一個猜測值 t，如果有一組解不小於 t，那麼必有 $\frac{f(v)}{g(v)} \geq t$，這相當於存在一組解使得 $f(v) - tg(v) \geq 0$，注意到在每次二分蒐的過程中我們只需要判斷是否有解滿足條件即可，並不需要直接求出最佳解，而當猜測值等於極值的時候自然會給出一組最佳的答案。

8.2.3 迭代

與二分搜的想法不同，一步步的增加猜測值 g 來求得最佳解，這是因為有的時候我們可以直接用當前的最佳解 g 篩掉許多資料，檢查是否存在 g' 比 g 更好，如果行得通就可以壓掉一個 $\lg N$。

8.2.4 合併演算法

如果我們有兩個演算法 A, B，而他們演算法的複雜度取決於不同的條件，那往往可以將兩個演算法合併得到更佳的演算法。比如輸入為一組滿足 $\sum x_i = n$ 的序列，$\{x_1, x_2, \cdots, x_k\}$，如果 A 演算法的複雜度取決於數字的個數 k，B 演算法的複雜度取決於數字中的最大值 $\max x_i$，可以知道單獨兩個演算法的最差時間複雜度都是 $O(N)$，但這時候如果我們把不超過 q 的數字挑出來給演算法 B 處裡，A 處理剩下的數字，可以知道 A 處裡的數字不會超過 N/q 個，因此我們的時間複雜度為 $q + N/q$，取 $q = \sqrt{N}$ 即得到一個 $O(\sqrt{N})$ 的演算法。

8.2.5 Exercise

- 給你一個圖 $G = (V, E)$，每個邊 e 上有一個權值 $a(e)$，要維護兩種操作：
  - 求出一條從 u 到 v 的路徑，使得路徑上邊權最大的一條邊最小。
  - 將一條邊的權值改小。
- 給一個 $N \times M$ 矩陣，數字為 1 到 NM。求一個長寬為 H, W 的矩形，使得矩形內的中位數最小。(<IOI 2010> [Quality of Living] , $M, N \leq 3000; H \equiv W \equiv 1 \pmod 2$)
- 給你一個圖 $G = (V, E)$，每個邊 e 上有兩個權值 $a(e), b(e)$，求出一個生成樹 T 使得 $\frac{\sum_{e \in T} a(e)}{\sum_{e \in T} b(e)}$ 最小。([最優比率樹] , $O(|V|^2)$)
- 給你一個圖 $G = (V, E)$，每個邊 e 上有權值 $w(e)$。給一個點 v 和一個正整數 k，求出一個生成樹 T 使得 v 在生成樹上的度數恰為 k 且 T 的權重總和最小。
- 給你長度為 N 的 $0-1$ 序列，找一個長度不少於 K 的區間使得 1 占的比率最高。(<TIOJ 1670> [新聞採訪] , $N \leq 10^6$)
- 給你長度為 N 的序列 $a_1, a_2, \cdots, a_n$，對於 Q 筆詢問 (s, d, t) 求出 $a_s + a_{s+d} + a_{s+2d} + \cdots + a_{s+td}$。(<Codeforces 103D> [Time to Raid Cowavans] , $N, Q \leq 80000$)
- 給你一個整數 N，求出 $\sum_{1 \leq x \leq N} (N \bmod x)$。(<TIOJ 1674> [新專輯] , $N \leq 10^12$)
- 給你長度為 N 的序列，對於 Q 筆詢問求出範圍內逆序數隊的數量。(<TIOJ 1694> [你的重生之旅] , $N, Q \leq 80000$)
- 給你長度為 N 的序列，對於 Q 筆詢問 (a, b) 求出數列中數字 a 和數字 b 最近的一對的距離。(<TIOJ ???> [Dice War] , $N, Q \leq 80000$)
- 有 N 隻大象站在一條直線上，你有一個照相機可以將長度為 x 中的所有大象都照下來，對於 Q 筆尋問求出一動一隻大象後你需要照幾次才能將所有大象照下來。(<IOI 2010> [Elephant] , $N, Q \leq 10^5$)
- 一個公司中有 N 個人，他們之間的上司關係恰成為一棵樹，並且每一個人來自一個地區 v_i，對於 Q 筆詢問 (a, b)，求出有多少對人 (x, y) 使得 x 是 y 在樹上的某個祖先並且 x, y 分別來自 a, b。(<IOI 2009> [Regions] , $N, Q \leq 80000$)

8.3 Flow & Cut

在生活中，我們常常會碰到有關輸送的問題，比如說我們要從 A 輸送物資到 B，而兩地之間有許多個中繼站，中繼站之間可以輸送，但有流量的限制，要請問從 A 輸送物資到 B 最大的流量是多少以及如何輸送。這種問題同樣的會在通訊網路、世紀帝國、BC2..... 以及在資訊題目時出現，所以當然非會不可。

8.3.1 定義

首先我們先將題目建成一個圖論的模型。給一個網路流圖 $G = (V, E)$，可以是有向、無向甚至是混合圖 (但為了方便以下沒有特別說明我們都討論有向圖的情況)。圖中有兩個特別的點，源點 s 以及匯點 t，並且有個函數 $c : E \mapsto \mathbb{R}^+$，即每一條邊都有一個權重，通常我們將 c 看作邊的容量。接著我們定義兩種不同的問題。

Maximum Flow

求出一個函數 $f : E \mapsto \mathbb{R}$，我們將 f 看作是邊的實際流量，也就是你要分派每一條邊一個固定的流量，而且 f 要滿足：

1. 斜對稱性：如果 u 到 v 有一個大小為 x 的流量，那我們也可以看作 v 到 u 有一個大小為 $-x$ 的流量。

$$\forall (u, v) \in E, \quad f(u, v) = -f(v, u)$$

2. 容量限制：流量不能超過容量限制，並且如果 $(u, v) \notin E$，則定義 $c(u, v) = 0$。

$$\forall (u, v) \in E, \quad f(u, v) \leq c(u, v)$$

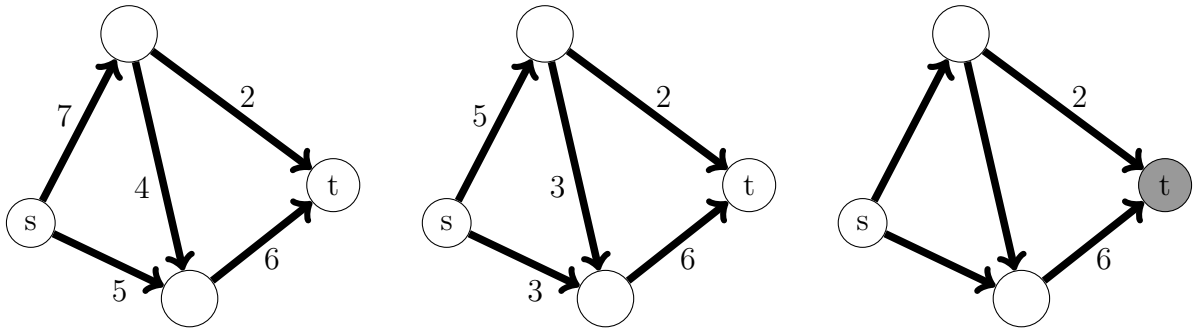
3. 流量守恆：除了源點只有流量出去，以及匯點只有流量進來以外，其他的點都要滿足進去的流量等於出去的流量。

$$\forall v \in E \setminus \{s, t\}, \quad \sum_{u \in V} f(v, u) = 0$$

且定義一個網路的流量 $|f| = \sum_{v \in V} f(v, t)$，也就是最後有多少流量流到匯點，Maximum Flow 即是要求一個 f 使得 $|f|$ 最大。

Minimum S-T Cut

接下來我們介紹另外一個完全不同的問題，S-T cut(S-T 割) 要將圖中的點分成兩個集合 S, T，使得 $S \cup T = V, S \cap T = \varnothing$，並且 $s \in S, t \in T$。接著定義 cut 的大小為 $\sum_{v \in S, u \in T} c(v, u)$，而最小割 (Minimum S-T cut) 即是要求 cut 大小的最小值。



Max-flow min-cut theorem

我們可以發現以上兩個問題有點像是兩個「相對」，但實質相同的問題，Max-flow 要求圖上的一個最大的流，而 min-cut 可以想成要求圖上的一個最小的瓶頸。事實上這兩個問題是線性規劃中的強對偶問題，Max-flow min-cut theorem 告訴我們 Max-flow 的大小恰等於 min-cut 的大小。

Question

- 給一個方法將無向圖和混和圖的流量網路轉化為有向圖流量網路處理。
- 請問一個 maximal flow 是否必為 maximum flow？
- 假設最大流的流量為 F，最小割的大小為 C。

  1. 證明 $F \leq C$。
  2. 證明 $C \leq F$。

8.3.2 Maximum flow Algorithm

解決最大流問題有許多有名的演算法，但基本上都是建立在剩餘流量網路上慢慢增加流量。

Residual network

首先我們定義一個網路流 $G = (V, E)$，他的剩餘流量網路 $G_f = (V, E_f)$，其中 $E_f = \{(u, v) : 0 < f(u, v) < c(u, v) \vee f(u, v) < 0\}$。
此外，定義邊 $(u, v) \in E_f$ 的剩餘容量 $c_f(u, v) = c(u, v) - f(u, v)$。
而如果當前的流量為 f，且 G_f 中存在一條從 s 到 t 的路徑，並且其中剩餘容量最小的一條邊為 $f' > 0$，那顯然我們可以將流量增加為 $f + f'$，我們把這條路徑稱作增廣路徑 (Augmenting path)。
Augmenting path theory 更告訴我們：一個網路流他達到最大流若且為若不存在一條增廣路徑。有了這個定理我們便可以求出最大流了。

Ford-Fulkerson's Algorithm

Ford-fulkerson's Algorithm 是 Augmenting path theory 立即的結果，即我們要求最大流，就從初始的剩餘網路開始不停的找任何一條增廣路徑增加流量，直到不存在增廣路徑為止。但這個演算法的時間複雜度最差為 $O(E|f|)$，即與最大流的值有關。一個最差的情況如下圖所示。也因此這個演算法一定要在容量限制皆為整數 (或是分數) 才可以運作，但事實上我們只要將每次的增廣路徑依照一定的方法尋找，就可以大大避免掉這些情況，因此有 Edmonds-Karp 演算法。

Edmonds-Karp's Algorithm

Edmonds-Karp's Algorithm 做的事情很簡單，只是將 Ford-fulkerson's Algorithm 中任意找一條增廣路徑改成每次都尋找最短的增廣路徑增廣，即可證明增廣次數不會超過 $O(VE)$，如果我們用 BFS 實作，則可以做到 $O(VE^2)$ 的時間複雜度。

Edmonds-Karp's Algorithm

Edmonds-Karp's Algorithm 做的事情很簡單，只是將 Ford-fulkerson's Algorithm 中任意找一條增廣路徑改成每次都尋找最短的增廣路徑增廣，即可證明增廣次數不會超過 $O(VE)$，如果我們用 BFS 實作，則可以做到 $O(VE^2)$ 的時間複雜度。

Improved Shortest Augmenting Path Algorithm

Edmonds-Karp's Algorithm 可以說是一個每次都找 Shortest Augmenting Path 來增廣的演算法，而 Ahuja 與 Orlin 更引入距離標號的概念，將之改進，複雜度為 $O(V^2E)$。

首先我們先定義距離標號，距離標號 d 給每一個點一個標號，代表這個點到匯點的距離下界，定義為 $d(t) = 0$ 且

$$d(u) \leq d(v) + 1, \quad \forall (u, v) \in E_f$$

顯然如果有一條從 s 到 t 的增廣路徑，那麼必有 $d(s) < |V|$，因此為了方便起見，我們定義那些在剩餘網路上無法連到匯點的點 v 的距離標號 $d(v) = |V|$。

接著我們定義可行弧，對於 $(u, v) \in E_f$，如果 $d(u) = d(v) + 1$，那我們就稱 (u, v) 為一條可行弧。而一條從 s 到 t、由可行弧組成的路徑必為一條最短增廣路徑，因此我們希望能不斷找由可行弧組成的增廣路徑來增廣，直到沒有增廣路徑為止。

但注意到 d 只是一個下界而已，甚至我們一開始只會將 d 初始化為 0 而已，因此我們要造出可行弧。定義兩種操作，Advance(v) 代表從 v 往下走一個可行弧，Relabel(v) 對 v 進行重新標號。

所以演算法要做的事情就很明確了，我們從源點 s 開始，假設現在到了點 v，先試著看能不能 Advance(v)，即往下走一個可行弧，如果不行我們就 Relabel(v)，而如果走到了 t，那就沿著走來的路徑增廣。一旦 $d(s) \geq |V|$ 那就確定我們已找到最大流了。

而這個演算法還有幾個重要的優化，如果在一次 Relabel 的時候我們發現距離標號 d 的值不連續，也就是說存在 $x_1 < x_2 < x_3$，使得有點的 $d(v) = x_1, d(u) = x_3$，但是沒有任何點的距

離標號為 x_2，這時候必定達到最大流了。

其次是 Relabel 的時候直接把 $d(v)$ 加一會比真的去取最小值還快一些。

Algorithm 20 Improved Shortest Augmenting Path Algorithm

```
1: function ADVANCE(v)
2:     if ∃u ⇒ (v, u) is an admissible arc then
3:         return u
4:     else
5:         return v
6:     end if
7: end function
8: function RELABEL(v)
9:     if ∃u ⇒ (v, u) ∈ E_f then
10:        d(v) ← min_{(v,u)∈E_f} d(u) + 1
11:    else
12:        d(v) ← |V|
13:    end if
14: end function
15: function *RELABEL(v)
16:    d(v) ← d(v) + 1
17:    if ∄u ⇒ d(u) = d(v) − 1 then
18:        return true
19:    end if
20:    return false
21: end function
22: function MAX_FLOW(Residual Flow network : G with source s and sink t)
23:    |f| ← 0
24:    d ← 0
25:    v ← s, π(s) ← s
26:    while d(s) < |V| do
27:        if ADVANCE(v) ≠ v then
28:            π(v) ← v
29:            v ← Advance(v)
30:            if v = t then
31:                |f| ← |f|+ AUGMENT(G,π)
32:            end if
33:        else
34:            if *RELABEL(v) = true then
35:                d(s) = |V|
36:            end if
37:        end if
38:    end while
39:    return |f|
40: end function
```

8.3.3 Minimum Cost Maximum Flow

假設我們現在流量網路的邊是有成本的，與流過邊的單位流量成正比。也就是說假設 e 的權重為 $c(e)$，如果有一條大小為 x 的流量流過，那我們便要負擔 $c(e)x$ 的成本，而一個流量網路的花費即是所有邊的花費總和。
最小花費最大流即是要求在流量網路上的一個最大流，且其花費最小。

Successive Shortest Path Algorithm

神奇的是，我們只要證明每次都拿剩餘網路中邊權總和最小的一條路徑增廣，那演算法結束後我們便會得到最小花費最大流。注意到這種題目通常會有負邊，所以必須用可以處理負權的演算法，如 SPFA。

但這樣還是沒辦法處理負環的情況，因此必須預處理，將圖中所有的負權環預先流滿，而在接下來便可以證明，如果一開始沒有負環，那在拿邊權總和最小的路徑增廣後也不會有負環。

8.3.4 最小割

如果要求一個最小割，我們只要先找出流量網路的一個最大流，最後在剩餘網路上從 s 開始可以走到的點的集合為 S，其他不可走到的點為 T，則 S, T 即為一個最小割。

8.3.5 建構模型

在上面我們會了解決最大流的演算法，現在我們要做的事就是把題目建模，成為網路流的問題。

基本流量建模

- 點有容量
  將點 v 拆成兩個點 v, v' 然後中間連一條限制容量的邊。

- 邊的容量沒有限制
  通常用很大的一個數字代替，只要保證操作不會溢位就可以。

- 要求整數解
  由 Augmenting path theory 可以知道只要邊是整數，那一定存在一組邊流量皆為整數的解，即使是最小花費最大流也一樣。

- 並不要求流量流滿，但少流一單位會有代價
  加一條成本為 c 且流量為無限大的邊。

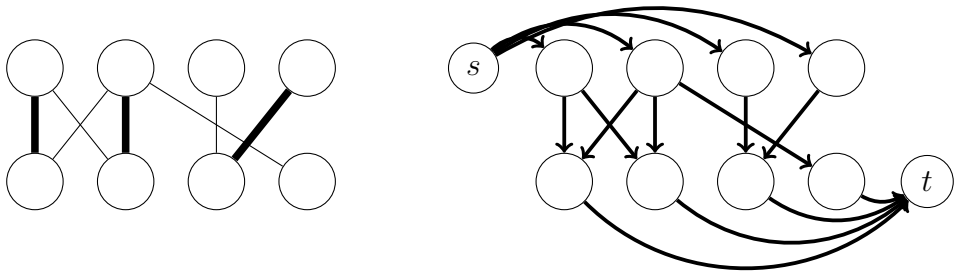
二分圖模型

回憶一下一個圖 $G = (V, E)$ 如果是二分圖，那就代表我們可以將點分成兩個集合 A, B 使得 $A \cap B = \varnothing$ 且任何邊 (u, v) 都滿足 $u \in A, v \in B$。

很多 NP 問題到了二分圖上都會有不錯的時間複雜度的解。

二分圖匹配是要求找出一個邊的子集 M，滿足其中的邊都不相鄰，並且我們希望邊的數量越多越好。可以想成我們用 M 中的邊將圖中的某些點兩兩匹配，並且一個點至多只能匹配另一個點。

而最大二分圖匹配可以很簡單的轉化為最大流模型，我們從源點把所有 $u \in A$ 都連一條容量為 1 的邊 (s, u)，把所有 $v \in B$ 都連一條容量為 1 的邊 (v, t) 到匯點，最後如果原圖有一條 $(u, v), u \in A, v \in B$ 的邊，那就在流量網路上加一條容量為 1 的邊 (u, v)，此時容易知道 s 到 t 的最大流即為二分圖的最大匹配數。



如果邊加上了花費 $-v(e)$，那我們可以想成選兩個點做配對會有價值 $v(u, v)$，要求價值最大的一組匹配，這可以轉化為最小花費最大流求解。而關於二分圖還有兩個重要的問題，最小點覆蓋還有最大獨立點集，這兩個問題恰好是相對的。

二分圖最小點覆蓋要求二分圖上的一個點的子集 C 使得所有的邊都與 C 中的某個點相鄰，而且希望 $|C|$ 越小越好，而最大獨立點集 I 則是希望能找到二分圖上的一個點的子集 I 使得 I 中任兩點都不相鄰。

一個巧妙的結果是，最小點覆蓋恰好會是最大獨立點集的補集，而且最小點覆蓋的大巧恰好會是最大匹配的大小。

而找一個最小點覆蓋的方法是這樣子的，先找出最大匹配後，如果沒有未匹配點，那隨便挑全部一側的點即會是一個點覆蓋，否則我們用以下方法將 V 分層。

- S_0 包含所有的未匹配點。(根據條件 $S_0 \neq \varnothing$)
- S_{2k-1} 包含所有與 S_{2k} 中其中一點相鄰，並且還不屬於前面任何一層的點。
- S_{2k} 包含所有與 S_{2k-1} 中其中一點以匹配邊相鄰，並且還不屬於前面任何一層的點。

- 如果 $S_{2k-1} = \varnothing$，任取一點丟進 S_{2k-1} 並繼續。

可以知道挑選出奇數層的點即可蓋住所有的邊，並且挑出的點不會超過最大匹配數。

Question

- 假設二分圖 $G = (V, E)$ 中 C 為任一個點覆蓋，證明 G/C 是一個獨立點集。
- 假設二分圖 $G = (V, E)$ 中 I 為任一個獨立點集，證明 G/I 是一個點覆蓋。
- 假設二分圖 $G = (V, E)$ 中 M 為一最大匹配，C 為一點覆蓋，證明 $|M| \leq |C|$。

下界流

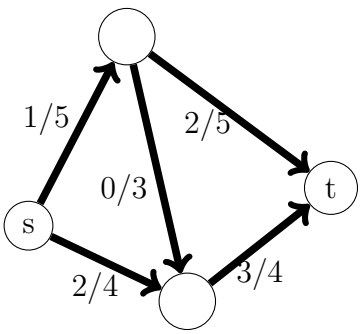
有時候我們會有流量下界的限制，即每條邊的流量除了容量為其上限以外，還不能低於一個值 $h(e)$。

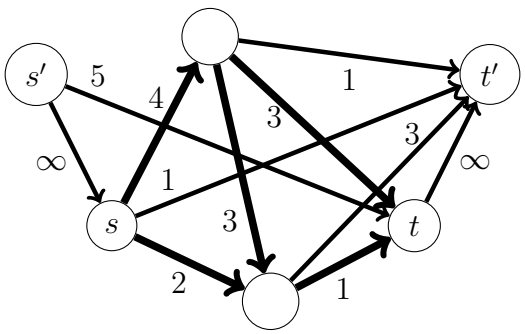
而要求一個滿足條件限制的下界流，想法是先把所有的邊都給一個等同於其下界的流量，這樣會使得一開始某些點進去和出來的流量不合，我們在用新增的源點和匯點來供給/接收多餘的流量。具體的寫法為：
新建一個流量網路 $G' = (V', E')$，其中 s', t' 為新的源點和匯點，並且 $V' = V \cup \{s', t'\}$。
令

$$f^+(v) = \sum_{(u,v)\in E} h(e), \quad f^-(v) = \sum_{(v,u)\in E} h(e)$$

- 對所有點 v 加一條容量為 $f^+(v)$ 的邊 (s', v)。
- 對所有點 v 加一條容量為 $f^-(v)$ 的邊 (v, t')。
- 對於原本存在的邊 e，將其容量減去 $h(e)$ 後加入 E'。
- 加一條容量為 ∞ 的邊 (s', s)。
- 加一條容量為 ∞ 的邊 (t, t')。

此時求 s' 到 t' 的最大流，如果 $|f| = \sum_{v\in V} f^+(v) = \sum_{v\in V} f^-(v)$ 那麼下界流存在，否則不存在。
而找到了一個下界流後，如果要找最大下界流，在已是合理的下界流 G' 的 $V' \setminus \{s', t'\}$ 中以 s, t 為原點匯點找最大流即可。





最小割模型

有時候會碰到的題目是，要把東西分成兩個集合，而屬於不同的兩個集合會有花費，或著是把某個集合選到其中一個集合會有花費等等，這時候可能可以化成最小割來解決，列出一些常見的建模方法。

- 一個點 v 被選到 S 會有花費 c
  連一條容量為 c 的邊 (v,t)，選到 T 的狀況同理。
- 如果某一個點 v 要在 T 中，那 u 也必須在 T 中
  連一條容量為 ∞ 的邊 (u,v)。

8.3.6 Exercise

- 有 N 個事件，每個事件會在 t_i 時間 (x_i, y_i) 位置發生。現在你要派出一些鎢絲處理這些事件，鎢絲一開始在原點，並且單位時間只能水平或垂直移動一單位，且移動速度是一格/單位時間。對每個事件都必須要在該事件發生前有至少一個鎢絲到達，問最少需要幾個鎢絲。(<NPSC 2005 初賽 pE> [魔法部的任務] , $N \leq 1000$)
- 給你一個混合圖 $G = (V, E)$，輸出任何一組歐拉迴路。(<Uva 10989> [Euler Circuit] , $|V| \leq 200$)
- 給你一個有向圖 $G = (V, E)$，求出最少要用多少有向路徑才能覆蓋所有的點。
- 現在你有數字 $1, 2, \cdots, N$，你希望把這些數字分成許多數列 $A_1, A_2, \cdots, A_k$，滿足對於每個數列 $A_i = \{a_1, a_2, \cdots, a_{m_i} =\}$ 來說都有 $a_i < a_{i+1}$ 且 $a_i + a_{i+1}$ 是質數，問你至少要分成多少數列。
- 在一個 $M \times N$ 的方格上，有些格子上有敵人。每次可以選擇炸一整行或一整列上的敵人，問至少要炸幾次。(<Uva 11419> [SAM I AM] , $M, N \leq 200$)
- 一間有 N 個員工的公司決定要裁員，每個人都有一個可正可負的貢獻值，可是也有許多影響關係：A 對 B 有影響的話，A 被裁掉 B 也會走人，問留下來的員工貢獻值和最大為多少。(<99 建中校內賽 p4> [房屋裁員] , $N \leq 500$)
- 在一張 $M \times N$ 的方格上，有些格子有障礙物。現在要用兩種蛇去把這張圖所有空格覆蓋恰好一次：1. 一個頭和尾接起來的蛇或是 2. 頭尾都貼在地圖邊界的蛇，問最少要幾種 2 型蛇才能達成目標。(<NPSC 2009 決賽 pF> [飛機上有蛇] , $N, M \leq 30$)
- 給一個序列 A 和 K，選出 K 條沒有重複數字的嚴格遞增子序列，求 K 條序列長度和最大值。(<NPSC 2009 決賽 pB> [多條嚴格遞增序列] , $|A|, |K| \leq 1000$)
- 數線上有 N 個點，現在要連其中 K 個點對，一個點最多連到一條線，求一個最短的連法。(<APIO 2007> [Backup] , $N \leq 10000$)
- 給你一個圖 $G = (V, E)$，求一個導出子圖 $G' = (V', E')$ 使其密度 $\frac{|E'|}{|V'|}$ 最大。

- 有 N 個儀器還有 M 種實驗，買每個儀器有花費 c_i，而做完每個實驗會有收入 b_i，但要做一個實驗就必須買特定的儀器。問你最多可以賺多少錢。($N, M \leq 1000$)
- 在一張 $M \times N$ 的方格上，有些格子有障礙物。現在 A,B 玩一個玩命遊戲，由 A 決定一個起點，之後由 B 先開始輪流開車，每次把車子開到上下左右相鄰的方格，並且走過的方格就會被放置一個地雷，誰被撞死或是炸死就輸了。問 A 選那些點可以獲勝。(<tioj ???> [棋盤策略遊戲] , $N, M \leq 300$)
- 現在有尺寸為 $[1, N]$ 的溜冰鞋各 k 雙，每個足尺寸為 r 的人可以穿尺寸界在 $[r, r + d]$ 的溜冰鞋，每個事件描述時間 i 有 r_i 個尺寸為 x_i 的人來或是離開，輸出每個事件後是否有方法分配溜冰鞋使得每個人都恰能分到一雙可穿的溜冰鞋。(<POI XVI> [Ice Skates] , $O(N \leq 10^5)$)