

# 清大程式競賽集訓營

---

ION Camp

*August 3, 2020*  
Version: 2020



## 行程表

| 日期     | 時間          | 地點                         | 活動內容      | 講師                   |
|--------|-------------|----------------------------|-----------|----------------------|
| 8/3(一) | 12:00-13:30 | 一樓大廳                       | 報到        |                      |
|        | 13:30-14:30 | 台達 107                     | 營長時間      |                      |
|        | 14:30-17:00 | 台達 107                     | 競賽基本知識與技巧 | 蔣立元                  |
|        | 17:00-18:30 | 台達 103                     | 用餐休息時間    |                      |
|        | 18:30-22:00 | 資電 323/326/328             | 模擬測驗      |                      |
| 8/4(二) | 08:00-09:00 | 台達 103                     | 早餐時間      |                      |
|        | 09:00-12:00 | (基礎) 台達 107<br>(進階) 台達 105 | 動態規劃      | (基礎) 何凱鈞<br>(進階) 許文弘 |
|        | 12:00-13:30 | 台達 103                     | 用餐休息時間    |                      |
|        | 13:30-17:30 | (基礎) 台達 107<br>(進階) 台達 105 | 資料結構      | (基礎) 李昕威<br>(進階) 吳宗達 |
|        | 17:30-19:00 | 台達 103                     | 用餐休息時間    |                      |
|        | 19:00-22:00 | 資電 323/326/328             | 模擬測驗      |                      |
| 8/5(三) | 08:00-09:00 | 台達 103                     | 早餐時間      |                      |
|        | 09:00-12:00 | (基礎) 台達 107<br>(進階) 台達 105 | 圖論        | (基礎) 李昕威<br>(進階) 李旺陽 |
|        | 12:00-13:30 | 台達 103                     | 用餐休息時間    |                      |
|        | 13:30-17:30 | 台達 107                     | 基礎數論      | 林鈞                   |
|        | 17:30-19:00 | 台達 103                     | 用餐休息時間    |                      |
|        | 19:00-22:00 | 資電 323/326/328             | 模擬測驗      |                      |
| 8/6(四) | 08:00-09:00 | 台達 103                     | 早餐時間      |                      |
|        | 09:00-12:00 | 台達 107                     | 計算幾何      | 葉柏宏                  |
|        | 12:00-13:30 | 台達 103                     | 用餐休息時間    |                      |
|        | 13:30-17:30 | 台達 107                     | 其他解題技巧    | 陳昱丞                  |
|        | 17:30-19:00 | 台達 103                     | 用餐休息時間    |                      |
|        | 19:00-22:00 | 資電 323/326/328             | 模擬測驗      |                      |
| 8/7(五) | 07:30-08:00 | 宿舍區                        | 退房        | 住宿同學                 |
|        | 08:00-09:00 | 台達 103                     | 早餐時間      |                      |
|        | 09:00-12:00 | 台達 107                     | TA Time   |                      |
|        | 12:00-13:30 | 台達 103                     | 用餐休息時間    |                      |
|        | 13:30-17:30 | 資電 323/326/328             | 模擬競賽      |                      |
|        | 18:00-19:00 | 台達 104                     | 結業典禮      |                      |
|        | 19:00-      |                            | 賦歸        |                      |

表 0.1: 行程表



# 教室環境與緊急聯絡資訊

## 0.1 教室資訊

2020 清大程式競賽集訓營基礎班上課教室為國立清華大學台達館一樓 107 教室，進階班於台達館一樓 105 教室進行。部份沒有分初階/進階的課程，統一於 107 教室上課。晚上上機測驗統一於資電館三樓 323, 326, 328, 三間電腦教室進行。用餐與工作人員休息室為一樓 103 教室。此外，第一天與最後一天行李放置區域為 104 教室。

在上課教室請勿吃東西以維護上課環境整潔，此外電腦教室嚴禁飲食。有需求者可到 103 教室及三樓電腦教室外休息區食用。一樓教室插座數量十分稀少，若有電子產品需要充電請諮詢工作人員，而三樓電腦教室有足夠數量插座可供使用。全館無線網路可使用自己原本學校提供的校際漫遊服務登入。

上課中有需要可以小聲討論，不干擾他人即可。若位置被障礙物擋住，或是有其他需求可洽工作人員更換座位。

## 0.2 電腦教室使用規則

三樓電腦教室以及要進入計算機中心處皆有門禁，若有外出需求如上廁所需與工作人員報備。

- 禁止對電腦設備有任何拆裝或破壞行為。
- 禁止存取使用非法軟體或是違反版權的資料、影音。
- 教室中禁止攜帶飲料或食物入內 (可帶純水)。
- 電腦使用完畢請關機。
- 沒戴耳機的同學，電腦請切換到靜音，避免干擾附近的同學。

## 0.3 生活須知

- 早上 8 點開始到上課前於台達 103 教室供應早餐。
- 請勿隨手亂扔垃圾，請丟棄至指定區域確實分類。(請勿隨意丟棄於廁所或茶水間)
- 隨身攜帶貴重物品，確實關門，勿讓小偷得逞。
- 上課時勿隨意走動，請勿於室內場所奔跑嬉戲，教室地面為階梯狀，移動時要留意。
- 營隊課程很多，營隊期間請勿熬夜，確實休息。

## 0.4 練習賽規則

- 題目有問題可以向工作人員反映。
- 請使用螢幕 debug，現場沒有印表機。
- 請盡可能的依據自己的實力挑戰題目，挑戰自我。
- 與同學討論題目請小聲，不要劇透還沒完成的同學。
- *AC* 時請不要跳起來大叫。
- 發揮運動家精神奮戰到最後一刻，而不是整個晚上發呆。
- 不要聽整個晚上的洗腦歌曲，影響思考能力。
- 主辦單位有最終的題目解釋與裁判權力。
- 請勿逗留電腦教室。

## 0.5 住宿生活公約

清華大學禮齋、義齋為學校宿舍，其中禮齋是清大學生大一專用的宿舍，可以體驗看看大學的住宿環境。禮齋、義齋皆為四人房，房型普遍小，清華大學學生至大二後才可抽籤換到其他較寬敞的二人房的宿舍區域。

1. 回宿舍後立即向家人報平安。
2. 住宿區沒有網路服務，洗澡完畢後請盡速就寢休息。
3. 不可以在房間裡用吹風機!!!! 請在房間外使用！
4. 飲水可至浴室方向的走廊左手邊使用。
5. 垃圾請丟棄於提供的塑膠袋中，並加以分類，不可亂扔廁所垃圾桶。
6. 夜間超過 23:30 禁止私自外出，需要購物請在回程時順路到小吃部 7-11 與麥當勞購買。
7. 宿舍後門往教室與商店階梯陡峭崎嶇且夜間昏暗，請勿單獨於此處行動，可繞遠走較明亮的坡道。
8. 請勿逗留他人房間，請勿任意進入異性宿舍區域。
9. 夜間超過 00:30 會自動關閉大燈，但書桌仍有電源可供電器產品充電。
10. 如果不幸被自動關燈，請隨手關閉電燈開關，避免早上 6:00 把室友全亮醒。
11. 全員離開房間時要關冷氣，冷氣卡額度吹完需要自費儲值。
12. 全員離開房間時務必鎖門，並隨身攜帶個人貴重物品。
13. 有其他需求可以向在場工作人員反映，營期全程有常駐一位工作人員於宿舍區域。
14. 非經學員許可或緊急狀況，工作人員不會擅入學員房間，請注意隨時是否有可疑人士徘徊於周遭。

## 0.6 自然災害處理與避難集合地點

若遭遇火災、地震等緊急災害，需要緊急避難時，最終集合地點為小吃部前的空地區域。

若遇颱風或強降雨，或其他原因達到停班停課標準，則當日課程取消，擇時補課。工作人員與校方會協助處理餐點的問題。

若不幸停電，請注意有沒有人在房間使用吹風機。

## 0.7 緊急聯絡資訊

營長：蔣立元，手機：0905005831。

國立清華大學資訊工程學系辦公室：03-5714787。

# Contents

|                               |           |
|-------------------------------|-----------|
| 0.1 教室資訊 . . . . .            | v         |
| 0.2 電腦教室使用規則 . . . . .        | v         |
| 0.3 生活須知 . . . . .            | vi        |
| 0.4 練習賽規則 . . . . .           | vi        |
| 0.5 住宿生活公約 . . . . .          | vii       |
| 0.6 自然災害處理與避難集合地點 . . . . .   | viii      |
| 0.7 緊急聯絡資訊 . . . . .          | viii      |
| <br>                          |           |
| <b>1 演算法競賽概述</b>              | <b>1</b>  |
| 1.1 自序 . . . . .              | 1         |
| 1.2 演算法競賽 . . . . .           | 1         |
| 1.2.1 程式語言的選擇 . . . . .       | 2         |
| 1.2.2 開發環境配置 . . . . .        | 3         |
| 1.2.3 解題方法 . . . . .          | 5         |
| 1.2.4 解讀錯誤訊息 . . . . .        | 7         |
| 1.3 時間分析 . . . . .            | 7         |
| 1.3.1 時間複雜度 . . . . .         | 8         |
| 1.3.2 常用的時間複雜度關係 . . . . .    | 9         |
| <br>                          |           |
| <b>2 基本語法知識</b>               | <b>11</b> |
| 2.1 基本解題語法知識 . . . . .        | 11        |
| 2.1.1 未定義行為 . . . . .         | 11        |
| 2.1.2 C++ 字串 . . . . .        | 12        |
| 2.1.3 浮點數誤差 . . . . .         | 13        |
| 2.1.4 for 迴圈的陷阱 . . . . .     | 14        |
| 2.2 資料的輸入與輸出 . . . . .        | 15        |
| 2.2.1 串流 Stream . . . . .     | 15        |
| 2.2.2 常用輸入輸出函數 . . . . .      | 16        |
| 2.3 輸入的處理技巧 . . . . .         | 20        |
| 2.3.1 連續輸入 . . . . .          | 20        |
| 2.3.2 字串串流 . . . . .          | 21        |
| 2.3.3 C++ 的 IO 優化 . . . . .   | 23        |
| 2.3.4 輸出浮點小數位數 . . . . .      | 24        |
| 2.4 簡單 C++ STL 工具介紹 . . . . . | 24        |
| 2.4.1 std::pair . . . . .     | 24        |

|          |                                 |           |
|----------|---------------------------------|-----------|
| 2.4.2    | <code>std::tuple</code>         | 25        |
| 2.4.3    | <code>std::vector</code>        | 26        |
| 2.4.4    | <code>std::bitset</code>        | 27        |
| 2.5      | 其他常見小技巧                         | 27        |
| 2.5.1    | <code>while(T--)</code>         | 28        |
| 2.5.2    | 全域變數                            | 28        |
| 2.5.3    | 型態轉換                            | 28        |
| 2.5.4    | <code>define</code> 小技巧         | 28        |
| 2.5.5    | 重新定義型別名                         | 29        |
| 2.5.6    | <code>auto</code>               | 29        |
| 2.5.7    | <code>reference</code>          | 29        |
| 2.5.8    | <code>Range-based for</code>    | 30        |
| 2.5.9    | <code>std::tie</code> compare   | 31        |
| 2.5.10   | 行尾空白                            | 32        |
| 2.5.11   | 函數物件與匿名函數                       | 32        |
| 2.5.12   | <code>include</code> 所有 Library | 33        |
| <b>3</b> | <b>基本技巧</b>                     | <b>35</b> |
| 3.1      | 遞迴-河內塔                          | 35        |
| 3.1.1    | 想法拆解                            | 36        |
| 3.1.2    | 函數設計                            | 36        |
| 3.1.3    | 遞迴停止條件                          | 37        |
| 3.2      | 快速幕與最大公因數                       | 37        |
| 3.2.1    | 快速幕                             | 37        |
| 3.2.2    | 最大公因數                           | 39        |
| 3.3      | 常見的排序法                          | 40        |
| 3.3.1    | <code>Merge Sort</code>         | 40        |
| 3.3.2    | <code>Quick Sort</code>         | 41        |
| 3.3.3    | <code>Counting Sort</code>      | 42        |
| 3.3.4    | <code>Radix Sort</code>         | 43        |
| 3.3.5    | <code>std::sort()</code>        | 44        |
| 3.4      | 暴力枚舉                            | 45        |
| 3.4.1    | 簡單枚舉                            | 45        |
| 3.4.2    | 深度優先搜尋                          | 46        |
| 3.4.3    | 枚舉排列                            | 48        |
| 3.4.4    | 枚舉子集                            | 49        |
| 3.4.5    | 枚舉組合                            | 49        |
| 3.5      | 二分搜尋法                           | 50        |
| 3.5.1    | STL 中的二分搜尋                      | 52        |
| 3.6      | 三分搜尋                            | 53        |
| 3.7      | 離散化                             | 54        |
| 3.8      | 練習題目                            | 56        |

|  |            |
|--|------------|
| <b>4 基礎動態規劃</b>                                    | <b>59</b>  |
| 4.1 前言 . . . . .                                   | 59         |
| 4.2 走樓梯 . . . . .                                  | 60         |
| 4.3 選數字 . . . . .                                  | 61         |
| 4.4 Vacation . . . . .                             | 62         |
| 4.5 LIS ( 最長遞增子序列 ) . . . . .                      | 63         |
| 4.6 背包問題 . . . . .                                 | 64         |
| 4.7 旅行推銷員問題 . . . . .                              | 65         |
| 4.8 Slimes . . . . .                               | 66         |
| 4.9 延伸學習 . . . . .                                 | 67         |
| 4.10 其他經典問題 . . . . .                              | 68         |
| <b>5 進階動態規劃</b>                                    | <b>71</b>  |
| 5.1 章節介紹 Introduction . . . . .                    | 71         |
| 5.2 DP 的理論面 . . . . .                              | 71         |
| 5.2.1 最佳子結構 . . . . .                              | 71         |
| 5.2.2 圖論觀點 . . . . .                               | 73         |
| 5.3 常見 DP 優化 . . . . .                             | 75         |
| 5.3.1 使用線段樹 . . . . .                              | 75         |
| 5.3.2 使用單調佇列 . . . . .                             | 80         |
| 5.3.3 矩陣快速幂 . . . . .                              | 86         |
| 5.4 進階 DP 技巧 . . . . .                             | 93         |
| 5.4.1 Divide and Conquer DP Optimization . . . . . | 93         |
| 5.4.2 其他技巧 . . . . .                               | 99         |
| <b>6 基礎資料結構</b>                                    | <b>105</b> |
| 6.1 標準模板庫 STL . . . . .                            | 105        |
| 6.1.1 佇列 Queue . . . . .                           | 105        |
| 6.1.2 堆疊 Stack . . . . .                           | 108        |
| 6.1.3 雙向佇列 Deque . . . . .                         | 111        |
| 6.1.4 優先佇列 Priority Queue . . . . .                | 113        |
| 6.1.5 集合 set . . . . .                             | 118        |
| 6.1.6 映射 map . . . . .                             | 121        |
| 6.1.7 雜湊表 unordered . . . . .                      | 122        |
| 6.1.8 以上內容都是 STL，那又如何自行搜尋資料？ . . . . .             | 122        |
| 6.2 並查集 Disjoint Set . . . . .                     | 123        |
| 6.2.1 例題演練 . . . . .                               | 126        |
| 6.2.2 練習題 . . . . .                                | 127        |
| 6.3 線段樹 . . . . .                                  | 128        |
| 6.3.1 例題演練 . . . . .                               | 129        |
| 6.3.2 pull(x, y) . . . . .                         | 130        |
| 6.3.3 build(id, l, r) . . . . .                    | 130        |

|          |   |            |
|----------|---|------------|
| 6.3.4    | Query(id, l, r, ql, qr) . . . . .                   | 130        |
| 6.3.5    | update(id, l, r, i, v) . . . . .                    | 131        |
| 6.3.6    | 時間分析 . . . . .                                      | 132        |
| 6.3.7    | 懶惰標記 . . . . .                                      | 132        |
| 6.3.8    | push(id) . . . . .                                  | 133        |
| 6.3.9    | 總結的模板 . . . . .                                     | 133        |
| 6.3.10   | 練習題 . . . . .                                       | 136        |
| 6.4      | 樹狀數組 BIT(Binary Indexed Tree) . . . . .             | 136        |
| 6.4.1    | 結構 . . . . .  | 137        |
| 6.4.2    | 前綴和 . . . . .                                       | 137        |
| 6.4.3    | 單點加值 . . . . .                                      | 138        |
| 6.4.4    | BIT 模板 . . . . .                                    | 139        |
| 6.4.5    | 例題演練 . . . . .                                      | 140        |
| 6.4.6    | 練習題 . . . . .                                       | 140        |
| 6.4.7    | 二維 BIT . . . . .                                    | 141        |
| 6.4.8    | 練習題 . . . . .                                       | 142        |
| 6.5      | 稀疏表 Sparse Table . . . . .                          | 142        |
| 6.5.1    | 練習題 . . . . .                                       | 143        |
| <b>7</b> | <b>進階資料結構</b>                                       | <b>145</b> |
| 7.1      | 線段樹再談 . . . . .                                     | 145        |
| 7.1.1    | 不單純的合併區間 . . . . .                                  | 145        |
| 7.1.2    | 與其他領域搭配使用 . . . . .                                 | 148        |
| 7.1.3    | 不直接用線段樹維護原序列 . . . . .                              | 150        |
| 7.1.4    | 區間修改再談 . . . . .                                    | 152        |
| 7.1.5    | 練習題 Exercise . . . . .                              | 154        |
| 7.2      | 樹堆 Treap . . . . .                                  | 155        |
| 7.2.1    | Treap 入門 . . . . .                                  | 155        |
| 7.2.2    | merge(Treap *a,Treap *b) . . . . .                  | 156        |
| 7.2.3    | split(Treap *p,Treap *&a,Treap *&b,int k) . . . . . | 159        |
| 7.2.4    | 二元搜尋樹操作 . . . . .                                   | 161        |
| 7.2.5    | 名次樹操作 . . . . .                                     | 162        |
| 7.2.6    | Treap 處理序列操作題 . . . . .                             | 166        |
| 7.2.7    | 總結 . . . . .  | 169        |
| 7.2.8    | 練習題 Exercise . . . . .                              | 169        |
| 7.3      | 持久化資料結構 . . . . .                                   | 171        |
| 7.3.1    | 持久化線段樹 . . . . .                                    | 171        |
| 7.3.2    | 持久化 Treap . . . . .                                 | 175        |
| 7.3.3    | 練習題 Exercise . . . . .                              | 176        |
| 7.4      | 二維線段樹 . . . . .                                     | 178        |
| 7.4.1    | 一維線段樹複習 . . . . .                                   | 178        |
| 7.4.2    | 二維版本 . . . . .                                      | 179        |

|          |   |            |
|----------|---|------------|
| 7.4.3    | 高維線段樹                                   | 185        |
| 7.4.4    | 懶惰標記？                                   | 185        |
| 7.4.5    | 練習題 Exercise                            | 186        |
| <b>8</b> | <b>基礎圖論 Basic Graph Theory</b>          | <b>187</b> |
| 8.1      | 什麼是圖論 Introduction                      | 187        |
| 8.2      | 名詞介紹                                    | 187        |
| 8.3      | 圖的表示法 Graph Representation              | 189        |
| 8.3.1    | 相鄰矩陣 Adjacency matrix                   | 189        |
| 8.3.2    | 相鄰串列 Adjacency List                     | 190        |
| 8.4      | 圖的遍歷 Graph Traversal                    | 191        |
| 8.4.1    | 深度優先搜尋 DFS                              | 192        |
| 8.4.2    | 廣度優先搜尋 BFS                              | 194        |
| 8.4.3    | 練習題                                     | 195        |
| 8.5      | 最短路徑 Shortest Path                      | 195        |
| 8.5.1    | BFS                                     | 196        |
| 8.5.2    | 鬆弛 Relax                                | 196        |
| 8.5.3    | Dijkstra                                | 196        |
| 8.5.4    | Bellman-Ford                            | 198        |
| 8.5.5    | Floyd-Warshall                          | 199        |
| 8.6      | 樹 Tree                                  | 200        |
| 8.6.1    | 專有名詞與性質 Glossary and Property           | 200        |
| 8.6.2    | 二元樹 Binary Tree                         | 201        |
| 8.7      | 最小生成樹 Minimum Spanning Tree             | 203        |
| 8.7.1    | Kruskal                                 | 204        |
| 8.7.2    | Prim                                    | 204        |
| 8.8      | 拓樸排序 Topological Sort                   | 206        |
| 8.8.1    | 練習題                                     | 207        |
| 8.9      | 二分圖 Bipartite Graph                     | 207        |
| 8.9.1    | 判斷一張圖是否為二分圖                             | 207        |
| 8.9.2    | 練習題                                     | 208        |
| 8.10     | 佛洛伊德演算法 Floyd Cycle Detection Algorithm | 208        |
| <b>9</b> | <b>進階圖論 Advanced Graph Theory</b>       | <b>211</b> |
| 9.1      | DFS Tree                                | 211        |
| 9.2      | 連通分量 Component                          | 212        |
| 9.2.1    | Tarjan's Algorithm for AP/Bridge        | 213        |
| 9.2.2    | 強連通分量 Strongly Connected Component      | 215        |
| 9.2.3    | 2-SAT 問題與強連通分量                          | 218        |
| 9.2.4    | 練習題 Exercise                            | 219        |
| 9.3      | 支配樹 dominator tree                      | 221        |
| 9.3.1    | 支配樹性質                                   | 221        |

|  |            |
|--|------------|
| 9.3.2 簡化問題：有向無環圖 (DAG) . . . . .                     | 221        |
| 9.3.3 一般圖的支配樹 (Lengauer-Tarjan 演算法) . . . . .        | 222        |
| 9.3.4 練習題 Exercise . . . . .                         | 231        |
| 9.4 樹 Tree . . . . .                                 | 232        |
| 9.4.1 樹重心 Tree Centroid . . . . .                    | 232        |
| 9.4.2 樹分治 Divide and Conquer on Trees . . . . .      | 234        |
| 9.4.3 最近共同祖先 Lowest Common Ancestor . . . . .        | 235        |
| 9.4.4 樹序列化 Euler Tour Technique . . . . .            | 237        |
| 9.4.5 樹鏈剖分 Heavy-Light Decomposition . . . . .       | 238        |
| 9.4.6 練習題 Exercise . . . . .                         | 240        |
| <b>10 數學基礎 Basic Math</b>                            | <b>243</b> |
| 10.1 數論 Number Theory . . . . .                      | 243        |
| 10.1.1 整除性 Divisibility . . . . .                    | 243        |
| 10.1.2 埃氏篩法 Sieve of Eratosthenes . . . . .          | 244        |
| 10.1.3 最大公因數與最小公倍數 GCD & LCM . . . . .               | 247        |
| 10.1.4 輾轉相除法 Euclid Algorithm . . . . .              | 247        |
| 10.2 模運算 Modular Arithmetics . . . . .               | 248        |
| 10.2.1 同餘式與模數 Congruence & Modulo . . . . .          | 248        |
| 10.2.2 反元素 Inverse Element . . . . .                 | 250        |
| 10.2.3 費馬小定理 Fermat's Little Theorem . . . . .       | 251        |
| 10.2.4 歐拉函數 Euler Function . . . . .                 | 253        |
| 10.2.5 中國剩餘定理 Chinese Remainder Theorem . . . . .    | 254        |
| <b>11 進階數學 Advanced Math</b>                         | <b>257</b> |
| 11.1 進階模運算 Advanced Modular Arithmetics . . . . .    | 257        |
| 11.1.1 數論函數 Number-Theoretic function . . . . .      | 257        |
| 11.1.2 莫比烏斯反演 Möbius Inversion Formula . . . . .     | 259        |
| 11.1.3 原根 Primitive Root . . . . .                   | 260        |
| 11.2 快速傅立葉變換 Fast Fourier Transform . . . . .        | 262        |
| 11.2.1 生成函數 Generating Function . . . . .            | 262        |
| 11.2.2 離散傅立葉變換 Discrete Fourier Transform . . . . .  | 264        |
| 11.2.3 快速傅立葉變換 Fast Fourier Transformation . . . . . | 268        |
| <b>12 基礎計算幾何 Basic Computational Geometry</b>        | <b>275</b> |
| 12.1 線性代數 . . . . .                                  | 275        |
| 12.1.1 線性變換 Linear Map . . . . .                     | 277        |
| 12.1.2 平面點 . . . . .                                 | 278        |
| 12.1.3 內積與外積 . . . . .                               | 280        |
| 12.2 線段 . . . . .                                    | 282        |
| 12.2.1 線段的表示方式 . . . . .                             | 282        |
| 12.2.2 判斷線段相交、找出直線交點 . . . . .                       | 284        |
| 12.3 凸包演算法 . . . . .                                 | 288        |

|           |                        |            |
|-----------|------------------------|------------|
| 12.3.1    | 凸包基礎                   | 288        |
| 12.3.2    | 性質                     | 288        |
| 12.3.3    | 找出凸包                   | 289        |
| 12.3.4    | 包含測試                   | 290        |
| 12.3.5    | 旋轉卡尺                   | 294        |
| 12.3.6    | 練習題                    | 296        |
| 12.4      | 折線技巧                   | 296        |
| 12.4.1    | 基礎問題                   | 296        |
| 12.4.2    | 折線結構                   | 297        |
| 12.4.3    | 折線更新                   | 297        |
| 12.4.4    | 折線實作                   | 299        |
| 12.5      | 凸包優化技巧                 | 299        |
| 12.5.1    | 基礎問題                   | 299        |
| 12.5.2    | 凸包結構                   | 300        |
| 12.5.3    | 構造凸包                   | 300        |
| 12.5.4    | 尋找最大值                  | 302        |
| 12.5.5    | 融合凸包                   | 303        |
| 12.5.6    | 可拓展集合 (Expandable set) | 303        |
| 12.5.7    | 移除直線操作 (離線)            | 304        |
| 12.5.8    | set 維護凸包               | 305        |
| 12.5.9    | 凸包優化小結                 | 306        |
| 12.6      | 最近點問題                  | 306        |
| 12.6.1    | 最近點對問題                 | 307        |
| 12.6.2    | 最近點詢問                  | 308        |
| 12.7      | 對偶性質                   | 310        |
| 12.7.1    | 基本性質                   | 311        |
| 12.7.2    | 凸包與直線函數的關係             | 312        |
| 12.8      | 感謝                     | 312        |
| 12.9      | 圖片來源                   | 312        |
| <b>13</b> | <b>其他解題技巧</b>          | <b>313</b> |
| 13.1      | 暴力美學                   | 313        |
| 13.1.1    | meet-in-the-middle     | 313        |
| 13.1.2    | 啟發式合併                  | 317        |
| 13.1.3    | 均攤分析                   | 322        |
| 13.1.4    | bitset 優化常數            | 325        |
| 13.1.5    | 練習題 Exercise           | 329        |
| 13.2      | 其他補充技巧                 | 331        |
| 13.2.1    | hash                   | 331        |
| 13.2.2    | 分塊                     | 336        |
| 13.2.3    | 莫隊                     | 340        |
| 13.2.4    | 根號想法                   | 343        |

|                     |     |
|---------------------|-----|
| 13.2.5 練習題 Exercise | 344 |
| 13.3 經典問題選講         | 346 |
| 13.3.1 約瑟夫問題        | 346 |
| 13.3.2 關燈遊戲         | 348 |
| 13.3.3 樹上最大獨立集      | 351 |
| 13.3.4 次小生成樹        | 352 |
| 13.3.5 僅有刪邊的連通性維護   | 359 |
| 13.3.6 練習題 Exercise | 361 |

# 演算法競賽概述

## 1.1 自序

早年資訊學科競賽相較於其他學科能力競賽來說，競爭對手相對地少，但近幾年政府與學校單位都有積極的推動程式設計課程，並且開始推動程式能力檢定，可預期這個領域將會有大量的新秀人才投入。但演算法競賽的內容更新速度極快，許多教師的程式設計方法欠缺經驗，坊間的參考資料、教材參差不齊，進而產生了入門困難的窘境。選手的培訓不外乎是前人學長姐所帶領、亦或者是額外花錢請私人家教。清大程式競賽集訓營致力於提供初學者一個入門的管道，不僅於提供必備的知識與學習方向，也希望能提升整體程式設計能力的水平。不論是國中、高中還是大學開始，只要肯努力投入精進自身，都能使自己的思考能力提升到另一個層次上。

## 1.2 演算法競賽

演算法，又稱為解決問題的方法，自西元前被公認最早的演算法 - 歐幾里得演算法至今，有許多人的思考紀錄被以各種形式保存下來，在現在的演算法競賽中，主辦單位透過設計一些已知的問題，來考驗選手如何思考、如何策畫方法來解決問題，並以程式設計的能力描述自己的想法。純粹的演算法，基本上就是數學的一個分支，以精確的方法來描述如何將一個問題的輸入，轉變到輸出的結果，因此對於敘述的嚴謹度是相當重要的，多數人經常只考慮了普遍狀況，對於細節卻疏於考慮，雖然實際生活上極端例子並不常見，但是轉變成了題目後，就是一個考驗選手思考嚴謹度的方法，撰寫程式時，會因為工具的優化及假設，強迫放大了細節產生的效應，而甚至可能因為程式語言的限制，迫使方法須要做某種程度的修改才能加以呈現。一般的出題者來說，要驗證一個程式是否正確，只測試簡單資料是絕對不夠的，會盡其所能得找出特殊的狀況、可能的反例來驗證選手的方法是否正確。因此演算法在資訊競賽中不僅是單純的思考，也考驗了使用程式語言正確詮釋自己想法的能力。

演算法題目與學校數學課本題目有相當大的差異，舉個可能的例子：

| 排列數字   | 數學課本習題 |
|--|--------|
| 有一個數列 $S = \{1, 5, 6, 3, 9, 10\}$ ，請由小到大排列這一些數字。 |        |

數學課本統計的篇章可以找到如此簡單的習題，你也能很直觀的回答出答案。然而在演算法的世界中通常不是想要求特定數字的答案是什麼，而是希望設計一個方法來解決問題，把題目敘述稍加變化一下，這個問題可能就變得不是那麼的親切：

| 排列數字   | 演算法習題 |
|--|-------|
| 有一個數列 $S = \{a_1, a_2, a_3, \dots, a_n\}$ ，請給出一個方法由小到大排列這一些數字。 |       |

在大多數演算法書籍中，都會介紹這題各式各樣花俏的解法。一般人雖然能非常容易的解決數學課本的數字排列，但卻不知道怎麼把解決第一題方法的過程，有條理地轉化成一種方法來解決第二個問題。而把方法翻譯成確切的敘述，甚至是正確的程式碼，是許多初學者學習上的瓶頸。可能對於程式語言不夠熟悉、可能不知道如何把步驟一步一步說清楚。這都是需要大量的練習與累積才能學會的。從思考到寫出程式並執行的過程，在演算法競賽的道路上，這是必備的拿手絕活。

在台灣常見的比賽類型有兩大類，分別為 *IOI* 類型的比賽與 *ICPC* 類型的比賽，在高中教育部舉辦的比賽多使用 *IOI* 賽制，以便與國際資訊奧林匹亞競賽 (*IOI*) 接軌，而大學或是一般的主辦單位通常使用的是國際大學生程式設計競賽 (*ICPC*) 規則。一般而言，*IOI* 類型的比賽是單人參賽且沒有答錯懲罰，甚至題目會設計出相當多不同規格的部分分數讓選手爭取。而 *ICPC* 類型的比賽為組隊參加，較重視團隊合作的能力，而且結果只有全對與全錯，答錯題目會有罰時的機制，能攜帶 codebook。

### 1.2.1 程式語言的選擇

在演算法競賽中，主辦單位通常可能會提供 C、C++、Java、Python 等等程式語言給參賽者們使用，而一般的 Online Judge (OJ，線上解題系統) 也經常提供這些語言上傳解題，確認使用者的程式碼執行結果是否正確。在本書中，主要使用 C++ 來描述程式碼，普遍 OJ 也都有提供該語言的解題服務。

或許每個人熟悉的程式語言不盡相同，但是實際上，大多數的演算法競賽選手都使用 C++ 作為主要的程式語言。事實上在比賽過程中通常只有在較特殊的題型，如大數運算才會考慮使用其他語言。選擇 C++ 語言除了歷史傳承的因素之外，還有一些其他語言無可取代的特點，C/C++ 的相對於其他語言受函數的限制少，只需要最基本的變數、條件、迴圈、陣列、函數語法就足以寫出絕大多數的程式，基本上沒有因為忘記某個函數怎麼用就寫不出來的問題；此外 C/C++ 執行速度相對快速，佔盡時間優勢，相同的演算法，在其他語言如 Java、Python 可能會需要花費數幾倍到數十倍的時間才能做到一樣的事情，除此之外，多數語言為了避免程式設計時犯錯，因此會有許多的機制保護程式的運行，但是比賽是在保證資料正常、正確的狀況下執行程式，因此程式觸發多餘保護機制反而會增加執行時間，對此限制最少的 C/C++ 語言可以給予程式設計師最大的自由來撰寫程式。

有些新手以 C 語言入門，由於 C 語言由於本身提供的功能相對上非常少，使用純 C 寫程式可能需要額外撰寫許多的程式碼，因而產生出了一些披了 C++ 外衣的 C 語言使用者，以 C 語言的觀念寫程式，但在必要時使用 C++ 的功能，本身卻根本不了解 C 與 C++ 之間的差異在哪。對於 C 與 C++ 的差異並不在本文的討論範圍之內，以下舉出一個例子。

```
1 // C 不同型態的資料要不同的函數才正確能處理
2 abs(1);
3 fabs(1.234);
4 fabsf(1.234f);
5 fabsl(1.234L);
6 // C++ 因為有函數重載、模板等幫助，一個 abs 就能搞定所有的事情
7 abs(1);
8 abs(1.234);
9 abs(1.234f);
10 abs(1.234L);
```

**程式碼 1.1：**取絕對值函數的比較

### 1.2.2 開發環境配置

提到資訊領域不得不提到 Linux 等類 UNIX 的作業系統，多數的比賽機器或是練習平台都是架設在 Lunix 類型的作業系統上，通常為了避免選手的開發環境與實際執行的機器有所差異，導致預期外的結果，近幾年來，越來越多的比賽單位會準備如 Ubuntu 等開放且較容易安裝使用的作業系統做為比賽的平台，因此對於平常只會使用 Winodws 的使用者來說，突然的轉換到新的環境的可能會不知所措，尤其在使用不熟悉的 IDE 或文字編輯器下，通常初次使用可能會不小心把自己的電腦搞得一團亂，因此選擇一個簡單通用的寫作環境是相當重要的。

在開始討論前，要先能分辨清楚**編輯器**、**編譯器**與**整合開發環境**的差異：

編輯器 (Editor)

如 Notepad、Notepad++、Sublime、VSCode、Vim 等等具有打純文字文件功能的軟體，通常具有幫程式碼套上顏色的功能，但是不直接具有編譯程式的功能，但多數可以使用插件 (Plugin) 增加額外的功能。

## 編譯器 (Compiler)

如 `gcc(g++)`、`clang++`、`VC++` 等等把程式碼轉換成可執行檔案的工具，不同編譯器對於同樣的語法可能會有不同處理方法，在相同編譯器上不同版本也會有許多的差異。

## 整合開發環境 (Integrated Development Environment, IDE)

簡單的來說，就是附贈編譯器的編輯器，把開發程式流程上需要的工具全部集合在一起，如 `DEV C++`、`Code::Blocks`、`Visual Studio` 等等。除了前兩個程式較為輕量，初學者容易安裝使用之外，多數的 IDE 架構都相當龐大，以 `Visual Studio` 來說，安裝整個工具包就需要將近 10GB 容量，學習使用的難度也較高。

在比賽經驗上，若為 Windows 系統，通常可以依照平時練習的形式配置環境，主辦單位通常都會附有 `DEV C++` 與 `Code::Blocks`。有些業界人士會對於使用這兩個骨董級的 IDE 感到排斥與疑惑，但是競賽與軟體開發終究是不同的方向，多數比賽場合都用不到過於花俏的功能，選擇適合的工具是相當的重要；若比賽環境是 Linux 的系統，本書推薦的配置是使用 `gedit`，配合手動編譯程式。`gedit` 是一個簡單易用的文字編輯器，在桌面版的 `Ubuntu` 大多都有附上該軟體<sup>i</sup>，地位形同與 Windows 的筆記本，但是 `gedit` 具有程式碼套色以及自動縮排的功能，足以供程式開發者使用。簡單的操作可以避免新手花費過多的時間學習使用編輯器，減少依賴編輯器功能的習慣。而手動編譯程式對於比賽來說是相當簡單的事，一般而言主辦單位會提供裁判如何編譯並執行程式的資訊，因此選手在比賽時會將手動編譯的參數設定的與裁判一致，如此一來可以避免測試時的執行結果與裁判有所差異，除此之外，為了有效運用編譯器的功能來協助除錯，會使用額外的參數如 `Wall`、`Wextra` 這兩個參數來讓編譯器偵測更多可能的錯誤。

```
1 # 基本用法
2 g++ source.cpp
3 # 打比賽常用
4 g++ -std=c++14 -O2 -Wall -Wextra source.cpp
5 # 2020 APCS 參數
6 g++ -g -O2 -std=gnu++11 -static -lm -o a.out source.cpp
```

**程式碼 1.2:** 使用 `g++` 編譯

`clang++` 設計上操作大致與 `g++` 相同，使用該指令再加上程式的檔案名就能編譯程式碼，如果沒有指定輸出程式名稱 (`-o name`)，則編譯器在 Windows 上會產生 `a.exe`，而在 Linux 上則是 `a.out` 的執行檔。不依靠 IDE 的協助，手動編譯在遇到 C++ 的部分語法如 STL、模板的錯誤時，訊息通常會相當的混亂晦澀，不利於初學者找出錯誤來源，需要經驗累積練習查閱錯誤資訊。

<sup>i</sup>不同發行版可能會略有變化，也可能是 Mousepad，但操作大同小異

上述參數基本上可以任意調換順序，當中要值得注意的是 `-std=XXX` 的這一項參數，`std` 參數指定了程式語言的標準版本，C++ 目前常見有 98/11/14/17 這幾個版本，不同版本間可以使用的語法以及功能有所差異，在此經常會使初學者犯下編譯失敗的錯誤，編譯器 `g++` 在版本 6 以前預設的版本是 C++ 98，`g++` 7 之後是 C++ 14，在未標明語言版本時要多加留意。參數上 `c++` 與 `gnu++` 的差異在於 `gnu++` 會額外開啟編譯器提供的附加功能，不過實際上即使有些附加功能不論何種標準預設皆會啟用。但本書並不推薦使用非標準規格語法，避免在不同編譯器下產生相容性問題，但當中仍有一些非正規但好用的功能，在使用時會特別提及。

如果有空檔或是準備時間，不妨把指令寫成 script，節約自己的時間，也能根據自己的需求來調整編譯的步驟。

```
1 #!/bin/bash
2 rm a.out
3 g++ -std=c++14 -O2 -Wall -Wextra -Wno-unused-result $1 && ./a.out
```

程式碼 1.3：編譯與執行 script

### 1.2.3 解題方法

一個完整的題目，會包含題目敘述、輸入說明、輸出說明、範例輸入、範例輸出、測資大小限制、記憶體和時間的限制，如果題目缺少這些限制應當立即向裁判反映。不過在一般區域賽或小比賽很常發生此狀況，甚至是在印度的國際賽事也有可能發生此問題，如果裁判覺得 no problem、也不打算理會這種問題，這時候就只能憑自己感覺來估計了。

解題時必須要根據題目的要求輸出正確的答案，以下以 A+B 問題為例子：

| A+B 問題   | 經典問題 |
|--|------|
| 給你兩個 int 範圍的整數，請把它們加起來後輸出<br>輸入只有一行，有兩個整數 a, b 以一個空白隔開，請輸出他們加起來後的結果並換行，保證答案也在 int 範圍。<br>記憶體限制:512MB，時間限制:1000ms |      |

| 範例輸入    | 範例輸出 |
|---------|------|
| 123 456 | 579  |

這題完整包含了以上的所有要求，是個非常好的題目，但是多數時候題目敘述不會像這題一樣簡單，IOI 的測機題就有一個長兩頁的題目敘述，結果只是普通的 A+B 問題的。在大學的比賽中，題目敘述經常以英文命題，而且不同出題者的英文寫作方式可能會有不同的風格，因此在大學階段的比賽，英文的閱讀能力也是非常重要的。

所謂的測資大小限制，是說題目給的測資會滿足題目設定的限制，不需要額外判斷它是不是滿足限制，意思是不需要做以下的判斷：

```
1 long long a, b;
2 cin >> a >> b;
3
4 if( a < INT_MIN || INT_MAX < a || b < INT_MIN || b > INT_MAX ) {
5     puts("輸入範圍錯誤！");
6 }
```

**程式碼 1.4:** 這樣做是多此一舉

此外也不需要輸出多餘的文字，以下為例：

```
1 #include<iostream>
2 using namespace std;
3 int main() {
4     int a, b;
5     cout << "請輸入a: ";
6     cin >> a;
7     cout << "請輸入b: ";
8     cin >> b;
9     cout << "a+b = " << a + b << '\n';
10    return 0;
11 }
```

**程式碼 1.5:** is it right?

有過解題經驗的人都知道這是錯的，平常 OJ 判斷程式正確的方法是把輸出印到檔案中，然後把 output 和正確的解答進行檔案比較，只要有一點不一樣就會被判斷成答案錯誤 (WA)，所以千萬不要印出多餘的字符，包括換行或空白。

正確的寫法如下：

```
1 #include<iostream>
2 using namespace std;
3 int main() {
4     int a, b;
5     cin >> a;
6     cin >> b;
7     cout << a + b << '\n';
8     return 0;
9 }
```

**程式碼 1.6:** A+B 問題 AC code

#### 1.2.4 解讀錯誤訊息

在目前的比賽制度下，只能上傳一次程式碼的比賽已不多見，因此可以利用從系統回傳的錯誤訊息找出程式可能的問題，大多數的 OJ 都會提供如下表中常見的資訊。一般而言 CE 是最容易處理的，通常是人為疏失不小心傳送到錯誤的程式語言，對比賽選手來說，TLE 是最棘手的問題，因為 TLE 表示運行結果超出時限，多數狀況下表示當前的方法效率不夠好，需要想一個更快的方法，此時就需要利用時間分析的技巧來估算可能的執行時間。

運行時錯誤 Runtime Error，俗稱當掉，是一般的 Judge 最容易誤判的的狀況，以常見的 PC<sup>2</sup> 來說，系統可能會以「是否沒有輸出」來判定程式有沒有當掉，然而這方法在大多數狀況下都無法正常判定，經常誤判為 WA 或 TLE 等其他的狀況，容易影響比賽選手的作答狀況，因此比賽時要特別注意。通常透過良好的寫程式習慣與編譯器的協助，避免未定義行為，就能有效的減少 RE 發生的可能性。

| 訊息                 | 意義          | 改進方法          |
|--------------------|-------------|---------------|
| Accept (AC)        | 答案正確        |               |
| Wrong Answer       | 答案錯誤        | 找 BUG 驗證解法正確性 |
| Time Limit Error   | 執行時間超過限制    | 考慮更好的演算法      |
| Memory Limit Error | 使用的記憶體超過限制  | 考慮更好的演算法      |
| Runtime Error      | 執行時發生錯誤     | 檢查是否有未定義行為    |
| Compilation Error  | 程式碼無法通過編譯   | 修改程式碼的語法錯誤    |
| Judge Error        | Judge 端發生錯誤 | 與管理員反映        |

### 1.3 時間分析

在解題目時，會發現有些方法可以很有效率地在時間內計算出答案，但也會發現有些方法即便結果正確，但是執行時間會超出題目的限制 (TLE) 而無法獲得任何分數，因此如何衡量一個方法的時間效率是一件相當重要的事情。有經驗的比賽選手在看完題目，想出了一個解法後，不會立刻地寫程式，而是**先估計演算法的時間複雜度**，檢驗想出來的方法是否能在要求的條件下執行完畢，如此一來可以避免花費不必要的時間撰寫必然無法獲得正確 (AC) 的程式上，而更進階者，甚至可以透過題目的數據範圍反推複雜度，猜測出可能的演算法是什麼，因此學習計算演算法的時間效率是相當重要的事。

一般而言計算演算法的時間效率，可以透過計算演算法要花費幾個基本步驟來得知，因為電腦 (或人腦) 是透過依序計算一個又一個的基本運算來執行一個程式，而一個基本運算包含了兩個數字的加、減、乘、除、讀取一個變數的資料等等。在實際應用中，如果能知道執行的步驟數以及運算速度的話，就能估計執行程式所需要的時間。以下方的找最大值範例中，請讀者先行估計該方法執行要花費幾個基本運算。為了方便，假設執行 max 那整行只需要一個基本運算。

```

1 ans = a[0];
2 for(int i=0; i<n; ++i)
3 {
4     ans = max(ans, a[i]);
5 }

```

**程式碼 1.7:** 找最大值

根據計算，第一行只會被執行一次，迴圈的初始化也只會被執行一次，條件判斷  $i < n$  會執行  $n + 1$  次<sup>i</sup>， $++i$  與  $\max$  各執行  $n$  次，因此這一個程式需要花費  $1 + 1 + n + 1 + n + n = 3 + 3n$  個步驟。

這樣對於細節仔細的分析顯然非常的浪費時間，如果演算法步驟數一多，或是參雜有條件判斷，便難以正確的計算步驟數，然而可以發現當輸入的資料變多時，演算法細節的步驟數會顯得不太重要，因此數學上有一套方法用來描述一個函數近似的結果，在資訊領域上稱之為時間複雜度。

### 1.3.1 時間複雜度

演算法的時間複雜度 (Time complexity)，相當於執行演算法需要的步驟數，在前一個範例中，我們計算了找最大值需要  $3n + 3$  的時間，但除此之外是否有更簡單的方法來表達步驟數的關係呢？

在國高中數學課本中，有提到過直線的一次函數以及拋物線的二次函數，如果兩個函數都是遞增的，從觀察函數的圖形，可以發現二次函數數值增加的速度遠比一次函數還要來的快：不論再怎麼陡峭的一次函數 (ex.  $y = 999x$ )，都會在某一個夠大的  $x$  之後 (ex.  $999^2$ )，被一個非常扁平的二次函數 (ex.  $y = \frac{1}{999}x^2$ ) 超越，在這稱二次函數**壓制**了一次函數。透過這一個概念，是否能利用一個比較簡單的函數  $g(x)$ ，來描述說一個演算法的步驟數被  $g(x)$  **壓制**呢？

數學上 Big-O 表示法即用來表示函數間的**壓制**關係，演算法經常使用 Big-O 來描述演算法的時間複雜度，省略不必要的細節，方便衡量比較不同演算法間的差異。在本書中皆使用  $\mathcal{O}$  符號<sup>ii</sup>來表示。

**定義 1.1.** 如果  $f(x) \in \mathcal{O}(g(x))$ ，則存在正數  $x_0, c$ ，使得  $x_0 \geq x$  時

$$f(x) \leq c \times g(x)$$

---

<sup>i</sup>要包含跳出迴圈的那一次判斷

<sup>ii</sup>讀音為 order

在定義1.1中， $f(x)$  表達的是資料數量與步驟數的關係。如果能找到另一個函數  $g(x)$ ，在  $x$  夠大的時候， $f(x)$  永遠小於  $g(x)$  的某一個倍數，則稱  $f(x)$  是  $\mathcal{O}(g(x))$ 。以前面找最大值演算法為範例，如果有  $n$  個資料，就需要  $f(n) = 3n + 3$  個步驟，使用該定理，可以證明  $f(n) \in \mathcal{O}(n)$ <sup>iii</sup>。

用同樣的證明方法，也可以說  $f(x) \in \mathcal{O}(n^2)$ 、 $f(x) \in \mathcal{O}(n^3)$ ，因此通常取 Big-O 時，會選擇越小、越接近的函數以準確的描述函數的成長關係。

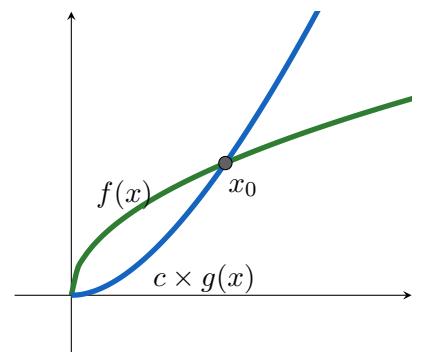


圖 1.1：圖解 Big-O 定義

使用 Big-O 表示法有幾個好處，第一是可以省略很多不重要的係數以及過於微小不重要的項目，第二是運算相當的簡單，對於多項式，Big-O 有下性質：

**定理 1.1.** 如果  $f(x)$  是一個  $n$  次多項式  $c_0 + c_1x^1 + c_2x^2 + \dots + c_nx^n$ ，當中  $c_n > 0$ ，則

$$f(x) \in \mathcal{O}(x^n)$$

舉個簡單的例子，如果  $f(x) = 2x^3 + 3x^2 + 6x + 7$ ，Big-O 可以簡單的記為  $\mathcal{O}(x^3)$ 。因為在  $x$  相當大的時候，次方數越大，對計算結果的影響力越大，比如說  $x = 1000$  時，三次方項  $2x^3 = 2 \times 10^9$ ，而二次方項  $3x^2 = 3 \times 10^6$ ，兩者差約 600 倍，這表示即便沒有計算二次方項，對於結果影響並不會太大。

除此之外也可以如下方使用極限的技巧定義 Big-O，但不在本書的討論範圍中。

$$f(n) \in \mathcal{O}(g(n)) \iff \limsup_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| < \infty$$

除了時間限制之外，一個題目還會有空間限制，因此對於一個演算法除了時間複雜度之外也可以估計他的空間複雜度。但只要我們知道基礎型別的空間佔多少，通常估計空間並不困難。現今的比賽也鮮少出現特別卡記憶體空間的題目，所以在此我們並不特別討論空間複雜度。

### 1.3.2 常用的時間複雜度關係

如果一個方法是透過多個方法組合而成的，那麼時間複雜度有下列簡單的關係：

<sup>iii</sup> 設  $x_0 = 1, c = 6$ ，可以簡單地發現當  $x \geq 1$  時， $3x + 3 \leq 6x$ 。

1. 加法規則： $f(x) + g(x) = \mathcal{O}(f, g)$  中影響力最大的項 )。

2. 乘法規則： $f(x)g(x) = \mathcal{O}(f(x)g(x))$ 。

以泡沫排序法 ( $\mathcal{O}(N^2)$ ) 的程式為例，程式的執行步驟是先讀取  $N$  個數字，再使用泡沫排序法，再輸出結果，這三個步驟複雜度分別為  $N, N^2, N$ ，因此總複雜度為  $\mathcal{O}(N + N^2 + N) = \mathcal{O}(N^2)$ ，只留下最有影響力的二次方項。如果泡沫排序法的每一個步驟花費的代價不是基本單位，比如說排序的對象是字串，比起檢查兩數字的大小只要一個步驟，檢驗兩字串的字典序會需要花費字串長度  $|r|$ ，那總複雜度就會需要在乘上細節的運算時間，總複雜度為  $\mathcal{O}(N^2|r|)$

```
1 for(int i=0; i<N; ++i)
2     for(int j=i+1; j<N; ++j)
3         if( a[i] > a[j] )
4             swap(a[i], a[j]);
```

程式碼 1.8: 泡沫排序法  $O(N^2)$

通常分析複雜度時，會把結果歸類在常用的表示法下方便討論，而在比賽經驗上，Big-O 的結果經常可以用來估計程式的效能！在一般的狀況下，普通的電腦每秒約能執行  $10^9$  個基本步驟，因此將題目的限制範圍直接帶入複雜度，就能估計是否會超出執行時間，一般出題者會將正確答案控制在  $10^6$  左右。下表 1.1 中列舉了常用的複雜度以及估計不會 TLE 的範圍。

表 1.1: 常見時間複雜度及競賽適用範圍

| Big-O 複雜度               | 競賽適用範圍                                 |
|-------------------------|--|
| $\mathcal{O}(n!)$       | $n \leq 10$                            |
| $\mathcal{O}(2^n)$      | $n \leq 20$                            |
| $\mathcal{O}(n^4)$      | $n \leq 50$                            |
| $\mathcal{O}(n^3)$      | $n \leq 200$                           |
| $\mathcal{O}(n^2)$      | $n \leq 3000$                          |
| $\mathcal{O}(n \log n)$ | $n \leq 10^6$ (有時候 $n = 10^7$ 也不會 TLE) |

但使用此方法有些陷阱要注意，第一是沒有經驗的出題者不一定會按照此規則，可能會出現  $n < 10^9$  但是  $O(n^2)$  的方法會 AC 的狀況，在台灣的大型比賽就曾發生此狀況；第二是機器速度不一定是  $10^9$ ，有可能更快或更慢，需要實際測量；最後，有時演算法的複雜度看似很好，但複雜度是只參考影響力最大的那項，若細節步驟數相當多可能會造成估計的誤差，比如說資料結構中的樹堆  $\mathcal{O}(n \log n)$  以及排序的  $\mathcal{O}(n \log n)$ ，儘管兩者的時間複雜度一樣，但樹堆的常數較大，執行時間也較慢，這在估計時是不可忽視的細節。因此當你在選擇要實作哪個演算法來解決問題時不應一昧的考慮時間複雜度的優劣，也需要考慮常數大小、實作起來的複雜程度等等。

# 基本語法知識

## 2.1 基本解題語法知識

由於 C++ 包含了 C 語言大多數的功能，部分的教學或範例經常以 C 語言的觀點出發來撰寫程式碼，但其實 C++ 可以將許多 C 語言的細節處理的更精簡安全，減少撰寫程式發生 Bug 的狀況，而且 C++ 繼承了絕大多數的 C 語言函式庫，引入時只需要稍微變換原有的檔名即可，對於部分的函數如取絕對值等，C++ 的標準有提供更好的解決方案，建議優先使用。

```
1 // C 的函式庫
2 #include<stdio.h>
3
4 // 轉成 C++ 的版本
5 #include<cstdio>
```

程式碼 2.1: C/C++ 共有標頭檔名稱轉換

C++ 的標準樣板函式庫 (Standard Template Library, STL) 與其他函數和物件都放置在命名空間 (namespace)std 底下，因此要使用時需要加上 std::，但是撰寫題目或小程式時會有點過於麻煩了，因此可以直接設定在程式中使用 std 這個命名空間。

```
1 // 寫在#include的下面
2 using namespace std;
```

程式碼 2.2: using namespace

也就是說，可以在一開始使用 using namespace std；，然後後面不管使用 cin, cout 或是其他的標準函數或物件都可以不必加上 std::，此外本書為了節省版面，除非可能混淆，否則也不會加上 std::。但是加上這一行之後，可能會讓程式與標準系統的函數撞名，導致編譯無法通過的情形，因此使用時要格外的注意。

### 2.1.1 未定義行為

未定義行為 (Undefined Behaviour, UB) 是學習 C/C++ 語言上一個十分重要的概念，未定義行為指的是程式碼的執行結果不可預測，或是語法可能呈現了邏輯的瑕疵，程式語言本身會定義某些行為是未定義的。

未定義行為的程式碼，有可能是可以運行的，這表示即便是可以編譯運作的程式也不代表是正確的，因此需要培養良好的程式設計習慣來避免發生如此問題。

```
1 cout << 1 / 0 << endl; //Undefined Behaviour
```

**程式碼 2.3:** 未定義行為 - 整數除 0

一般的 Judge 都會開啟編譯優化，在有優化的狀況下，未定義行為造成的效應會更加明顯。一般的編譯器通常能在編譯時期警告一部份常見的錯誤，是寫程式時不可忽略的細節。強烈建議讀者執行看看下列這份程式碼，可能會出現意想不到的結果。

```
1 int x;
2 cin >> x; // 輸入 int 最大值 2147483647
3 if( x + 1 > x ) { // 可能因為編譯器優化導致沒有進入這個條件
4     cout << "x+1>x\n";
5 }
6 else {
7     cout << "x+1<=x\n";
8 }
9 cout << x+1 << ' ' << x << '\n';
```

**程式碼 2.4:** 未定義行為 - 溢位

### 2.1.2 C++ 字串

C++ 有提供標準字串的類別 (std::string)，比起 char 陣列的 C style string，C++ string 可以自動處理字串長度的問題，而且內建各種操作，可以直接用比較符號進行字典序的比較，非常的方便，無需額外使用函數例如 strcpy、strcmp ..... 等等。

C 語言的函數無法直接使用 std::string 來進行操作，但還是可以透過 c\_str() 來回傳可讓 C 語言函數讀取的陣列。需要儲存、處理字串時，除非特殊情況，否則建議使用 string 而非 C-Style string，更多 string 的詳細內容會在之後的篇章提及。

```
1 string a = "apple";
2 string b = "pine" + a; // b = "pineapple"
3 if( a < b ) // true
4 {
5     cout << a << "字典序小於" << b << '\n';
6 }
7
8 char c[100];
9 strcpy( c, a.c_str() ); // c_str() 回傳 C 函數可用的資料
10 // 執行後 c = "apple"
```

**程式碼 2.5:** C++ std::string

### 2.1.3 浮點數誤差

當在寫競賽題目時，小數處理永遠不要使用 **float**，**float** 的準度遠比 **double** 來的小，且運算效率也沒有比較快，在十進位表示法下，**float** 最多有 6 位的精準位數，**double** 却有 15 位，因此在競賽中使用 **float** 極度容易發生誤差導致輸出錯誤的情形。

```
1 #include<cstdio>
2 template<typename T>
3 void test() {
4     T a = 1.0 / 81;
5     T b = 0;
6     for(int i = 0; i < 81; ++ i)
7         b += a;
8     printf("%.15f\n", b);
9 }
10
11 int main() {
12     test<float>(); // print 0.999999523162842
13     test<double>(); // print 1.0000000000000002
14 }
```

程式碼 2.6: float 和 double 比較

可以發現在經過多次的運算之後，**double** 的結果是比較精準的，**float** 相比之下誤差相當的大，在競賽中常常會有這種需要精準度的大量運算，因此盡量避免使用 **float**。

此外還有一個型態 **long double**，是在當 **double** 精度不夠使用的時候用的，但是運算速度極慢，只有在一些極端的應用中才有利用價值，如果有辦法避免就要避免。

|  |        |
|--|--------|
| 你覺得這是幾次方？  | 奇怪的 OJ |
| 如果 $A, B$ 都是正整數的話，那次方運算 $A^B$ 表示 $\underbrace{A \times A \times \dots \times A}_{B\text{次}}$ 。 |        |
| 如果今天知道 $A$ 以及 $A^B$ 的計算結果，你能求出 $B$ 是多少嗎？   |        |
| $B$ 一定是一個正整數， $1 < A$ 且 $1 < A^B \leq 2^{31} - 1$ 。  |        |

這一題最簡單的方法，就是使用 **while** 迴圈逐一計算  $A^0, A^1 \dots A^n$  直到結果等於  $A^B$  為止，可以發現在這一題中，次方數  $B$  一定會小於 32，因此這方法是簡單快速的。但是如果在高中數學有學習過  $\log$  運算的話，那其實有數學公式  $\log_A A^B = B$  可以使用，再透過換底公式  $\log_c x / \log_c A = \log_A x$ ，可能就會有一些同學使用下列方法計算答案：

```
1 int B = log(AB)/log(A);  
2 cout << B << endl;
```

程式碼 2.7：使用高中公式的程式碼（有 bug）

可以稍微紙筆計算一下， $3^5 = 3 \times 3 \times 3 \times 3 \times 3 = 243$ ，因此當我們將  $C = 243, A = 3$  輸入至程式時，結果應該會是 5，不過事實上該程式輸出的結果是 4，究竟是發生了什麼問題？

由於浮點數運算時都會有一點點的誤差，結算的結果可能會比答案大一點，也可能會小一點。這個現象如果沒有仔細考慮，就把小數轉換成整數會有很嚴重的問題。C++ 中，把小數轉換成整數的方法是省略小數點後的所有資料，比如說如果計算結果比 5 大一點點，像是 5.0001，轉換成 int 就會是 5，但是如果運算結果比 5 小一點點，像是上述程式碼計算的實際結果 4.9999，結果就會變成 4 了。因此處理題目時，若可以應盡量避免小數的運算，否則就應該要正確處理誤差的問題，以免造成難以發現的錯誤<sup>i</sup>。

#### 2.1.4 for 迴圈的陷阱

有許多初學者可能會習慣的在 for 迴圈內使用函數，但有時會造成嚴重的問題：

```
1 char s[1000005];  
2 // ...  
3 for(int i=0; i<strlen(s); ++i) {  
4     // do something  
5 }
```

程式碼 2.8：有問題的程式碼

上範例中，在 for 執行的時候每次都會執行一次 strlen 函數，但是這個函數每次執行都需要花費  $\mathcal{O}(|s|)$  的時間，即檢查過整個字串才能算出它的長度。因此字串長度是  $N$  的話，總共會執行  $N$  次 strlen，每次 strlen 會掃過  $N$  個字元，總共會掃過  $N^2$  個字元，當  $N = 10^6$  時就要掃過  $10^{12}$  個字元，在一般電腦可能要花好幾分鐘的時間來做這件事。因此正確的寫法應該要預先設立一個變數預先算出長度，避免重複的計算。不過使用 C++ string 的話，可以直接使用 size() 來取得長度，時間複雜度為  $\mathcal{O}(1)$ ，不會造成這種問題。

<sup>i</sup>更多電腦小數的資料可以查詢 IEEE 754

```

1 char s[1000005];
2 // ...
3 int N = strlen(s);
4 for(int i=0; i<N; ++i){
5     //do something
6 }

```

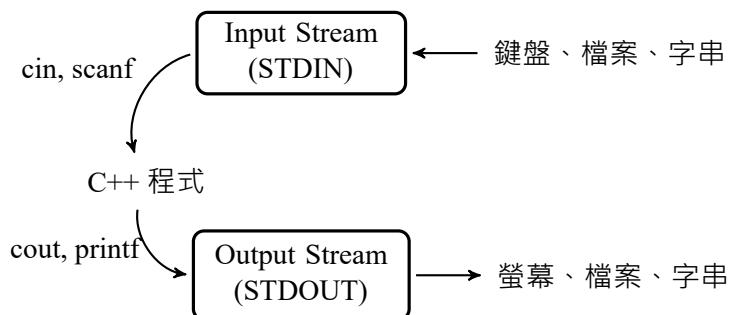
程式碼 2.9: 正確的寫法

## 2.2 資料的輸入與輸出

撰寫程式時，讀取資料與輸出資料是相當基礎與重要的事情，在比賽中，雖然經常使用 C++ 撰寫程式，但在讀取資料上有較大的分歧，有些選手習慣使用 C++ 語言的 `cin, cout`，有些選手習慣使用 C 語言的 `scanf, printf`。在實際應用上，雖然 `cin, cout` 較為方便，但是使用不慎可能會導致程式效能比 `printf, scanf` 慢上 10 倍，在部分題目中，會發生已超過執行時間，但資料卻尚未讀取完畢的狀況，但若妥善處理，兩者方法速度是相差不多的。因此基本上只要將其中一種方法學習精通，能自由運用即可。

### 2.2.1 串流 Stream

C++ 的輸入與輸出資料是以流 (Stream) 的概念來運作的，程式不是直接地向外部溝通，而是透過一個制訂好規格的管線流 (串流) 來操作，如此一來，C++ 在操作輸入與輸出的時候就不需要在意是對什麼物件進行動作，只要對著串流操作即可。一般而言預設會有 STDIN 與 STDOUT 兩種串流，俗稱標準輸入與標準輸出。除了預設的串流之外，也可以建立自己的串流，像是一些舊的題目可能會要求開檔讀寫測資，或是利用字串串流來進一步處理字串資料等等。



一般的 shell，如 cmd、bash 等等都支援在不修改程式的狀況下，重新導向一個程式的標準輸入與標準輸出到不同的地方。除了預設的鍵盤、螢幕之外，更常見的還有讀寫檔案，甚至可以把一個程式的輸出接到另外一個程式的輸入 (pipe)，如果裝置支援，甚至能接到喇叭等設備上進行操作<sup>i</sup>。因此對於一些測試資料相當冗長

<sup>i</sup>SECCON 2017 - putchar music <https://ctftime.org/task/5048>

的題目，就可以預先把測資預先打在檔案中，再用重新導向的功能來讀取資料或是檢查答案是否正確，減少輸入錯誤的可能。

一般 Judge 的設計，經常就是透過重新導向的技巧，將程式的標準輸入輸出更改成檔案讀取寫入，讓檢查的程序自動化。通常比賽只有在極少數不方便的狀況下<sup>ii</sup>才會是人工輸入的。

```
1 a.exe < input.txt # 從 input.txt 讀取資料
2 a.exe > output.txt # 輸出資料到 output.txt
3 a.exe < input.txt > output.txt # 可以一起用
4 diff ac.txt output.txt # 比較檔案是否相同 (Linux)
5 fc ac.txt output.txt # 比較檔案是否相同 (Windows)
```

程式碼 2.10: 一般 shell 的重新導向語法

部分題目會要求以檔案結尾 (End of Line, EOF) 來結束程式，若使用鍵盤作為輸入來測試，可以使用 **[Ctrl]+[Z]**，Mac 電腦則為 **[Ctrl]+[D]** 來輸入，完整地確認程式是否會如期的關閉，避免意外的逾時 (TLE) 問題。

## 2.2.2 常用輸入輸出函數

一般而言，輸入與輸出是程式中單一操作最緩慢的地方，通常測試資料都會設計的讓程式容易使用基本的函數讀取，避免需要過於複雜的技巧，但仍然有某些資料，比如說時間、大數等無法簡單處理，若不熟悉函數的使用，通用性的方法就是把所有資料當作字串，再利用字串處理的技巧或是字串串流來處理。

特別值得注意的是在讀取整行的函數中，舊的教材與網路資料可能會提供 `gets` 函數的教學，但是因為 `gets` 設計上有不可避免的安全性的問題，會造成程式的漏洞，因此在 C、C++11 後的標準已經移除了，建議不要再使用。

| 類型   | C                           | C++                               |
|------|-----------------------------|-----------------------------------|
| 一般輸入 | <code>scanf</code>          | <code>cin</code>                  |
| 讀取整行 | <code>fgets</code>          | <code>getline, cin.getline</code> |
| 讀取字元 | <code>getchar</code>        | <code>cin.get</code>              |
| 一般輸出 | <code>printf</code>         | <code>cout</code>                 |
| 輸出整行 | <code>puts</code>           |                                   |
| 輸出字元 | <code>putchar</code>        | <code>cin.put</code>              |
| 字串串流 | <code>sscanf/sprintf</code> | <code>stringstream</code>         |

表 2.1: 常用輸入輸出函數

<sup>ii</sup>或是偷懶或不會裝 Judge

## C++ 的一般輸入與輸出

C++ 的輸入輸出在許多程式語言中，算是相當親切且容易使用的，使用了函數多載的特性，cin, cout 不需要程式設計師標記型態，就可以判斷要輸出的資料是什麼。

```
1 int a;
2 double b;
3 cin >> a >> b;
4 cout << a << "+" << b << "=" << a + b << '\n'; //Judge 上用 endl 會很慢
```

程式碼 2.11: cin cout 範例

比賽中值得注意的是 cout 的換行問題，執行cout << endl;相當於執行cout << '\n' << flush;，當中的 flush 是清除緩衝區的意思。由於輸出函數只負責把資料送入輸出串流中，不見得會立刻把資料送出去，此時可以使用 flush 來把尚未送出的資料清出去。但是由於檔案的讀寫操作對於程式來說是非常緩且浪費時間的事情，因此通常 STDOUT 會需要累積多一點資料在一起送出，以加快程式的運作，在輸出大量資料時，應盡量不必要的 endl 以提升程式速度。

## C 的一般輸入與輸出

純 C 與 C++ 的 C 語言輸入可能會存在些微的差異，因此要格外的注意引入的標頭檔是否為 cstdio。C 語言的輸入輸出，因為函數不知道放上去的資料是什麼型態，因此使用時需要轉換符號%X 來區別。填入錯誤的轉換符號，或是符號與型態無法對應都是未定義行為，在某些舊版本的編譯器可能會有不同的執行結果，過去比賽中曾發生打錯卻在本地可正常執行，上傳程式卻是錯誤 (WA) 的狀況，因此不可不慎。

scanf/printf 的標記基本上以%[\*][寬度][型態標記] 轉換符來表示，可使用的參數與標記需要額外記憶，下表有常使用的參數。需要特別注意的是在 printf 輸出 double 時，在 C++11 以後才能使用 lf 的標記，較舊標準必須使用 f，此處在許多參考資料皆有誤植的狀況，需要特別留意。

```
1 int a;
2 double b;
3 scanf ("%d %lf", &a, &b);
4 printf ("%d+%f=%f", a, b, a+b); //C++11 以後才可用%lf
```

程式碼 2.12: scanf printf 範例

而在輸入輸出型態 long long 時，C++11 以後的標準都是使用lld，然而在 C++11 以前沒有定義，導致了部分的平台如 Windows 可能會需要使用 I64d 作為參數，

因此在比賽測試環境時，務必要確認要使用何種參數輸出 long long 型態。C++ 的物件都不能直接用在 C 語言的輸入輸出上，比如說 C++ 的 std::string 就無法直接使用 scanf/printf，需要使用 C++ 提供的函數來操作。

| 函數/標記/型態      | float | double | long double |
|---------------|-------|--------|-------------|
| scanf         | f     | lf     | Lf          |
| printf        | f     | f      | Lf          |
| printf(C++11) | f     | f 或 lf | Lf          |

| 轉換符  | 型態                     | 說明             |
|------|------------------------|----------------|
| %d   | int                    | 10 進位整數輸入      |
| %u   | unsigned int           | 10 進位無號整數輸入    |
| %x   | int                    | 將整數以 16 進位方式輸入 |
| %lld | long long int          | 10 進位整數輸入      |
| %llu | unsigned long long int | 10 進位無號整數輸入    |
| %c   | char                   | 以字元方式輸入        |
| %s   | char*                  | 字串輸入           |

scanf 可以做到格式化輸入，可以用類似正規表達式的方式做輸入標記，來處理複雜的輸入。更詳細的資訊與用法建議到 cppreference 上查詢。

```
1 scanf("%d,%d", &x, &y); // 輸入:(123,456)
2 int x,y;
3 printf("%d %d\n", x, y); // 輸出:123 456
```

程式碼 2.13: 格式化輸入

## C++ 整行讀取

在部分較複雜的輸入中，可以考慮先把資料完整的讀取成字串，再進一步處理。而 C++ 有兩種常用的字串型態：C Style String，也就是常見 char 陣列形式的字串，以及 std::string，兩者需要使用不同的方法來讀取。

cin.getline()可以將資料讀入 C Style String 中。這個函數有三個參數，第一是字串陣列的指標，第二是該陣列的大小，第三是結束字元，預設是換行符號因此通常會省略。

```
1 char str[100];
2 cin.getline(str, 100); // 輸入 abcd 1234
3 //等於 cin.getline(str, 100, '\n');
4 cout << str << '\n'; // 輸出 abcd 1234
```

程式碼 2.14: cin.getline

`getline()` 定義在 `string` 中，提供了 `std::string` 的整行讀取，同樣有三個參數，但是前兩個參數與 `cin.getline()` 不同，第一個參數是要從哪一個串流物件讀取，第二個是要存放的字串。

```
1 string str;
2 getline(cin, str); // 輸入 abcd 1234
3 cout << str << '\n'; // 輸出 abcd 1234
```

程式碼 2.15: `getline`

在整行讀取中要注意如果與一般輸入混合使用時，遇到換行時，一般輸入會殘留一個換行符號在串流中，此時如果沒有這一個換行符號清掉，就會讓整行讀取第一個字就遇到換行符號，結果就只拿到了一個空字串。知道了這一個細節之後，就有很多方法可處理這一個問題，常見的是利用 `cin.get()` 吃掉多餘的換行字元。

```
1 int n; // 輸入： 1'\n'abcde
2 string str;
3 cin >> n;
4 getline(cin, str);
5 cout << str << '\n'; // 輸出：空字串
```

程式碼 2.16: `getline` 與一般輸入混合使用

## C 整行讀取

在當前標準中，可以使用 `fgets` 來完成整行讀取，`fgets` 需要三個參數，第一個參數是陣列的指標，第二是陣列的大小，第三個參數比較特別，是串流的資訊，由於一般都是使用標準輸入，因此 C 預先定義了 `stdin` 可以直接使用

```
1 char str[100]
2 fgets(str, 100, stdin); // 輸入 abcd 1234
3 cout << str << "X" << '\n'; // 輸出 abcd 1234\nX (注意換行!)
4 *strchr(s, '\n') = 0; // 移除換行符號，要 include cstring
5 cout << str << "X" << '\n'; // 輸出 abcd 1234X (沒有換行!)
```

程式碼 2.17: `fgets`

`fgets` 與其他整行讀取最大的差別在於 `fgets` 讀取到換行符號時，並不會像 C++ 直接丟棄，而是把這一個符號放在讀取的字串中，但是在大多數的資料處理上都不希望出現這一個多餘的換行，因此使用 `fgets` 通常會需要多做一個步驟來移除換行符號。

## 2.3 輸入的處理技巧

有時候題目的輸入是很複雜的，可能難以直接用輸入輸出函數處理，這時通常會把輸入的東西先使用字串存起來之後再進一步操作，因此如何用簡單有效率的方法從字串中讀取想要的資訊是程式競賽中很重要的一部份。

| A+B+C... 問題  | 經典問題 |
|--|------|
| <p>A+B 問題太簡單了？那如果現在不保證一行只有兩個數字你會做嗎？</p> <p>輸入有多行，每行有多個 int 範圍的數字以空白隔開，請對於每一行輸出該行所有數字相加的結果並換行，保證每行的字元數不超過 1000000，總共不超過 30 行。</p> |      |

| 範例輸入                 | 範例輸出 |
|----------------------|------|
| 123 456 789          | 1368 |
| 1 2 3 4 5 6 7 8 9 10 | 55   |
| 11                   | 11   |

### 2.3.1 連續輸入

若有多筆測資，題目可能會要求以檔案結尾 (End of line, EOF) 作為結束，這樣程式就不能確定使用者會輸入幾次測資，此時程式必須要能連續輸入處理相同的問題，直到輸入結束為止。

#### scanf 判斷 EOF

scanf 可以透過函數的回傳值來判斷讀取的狀況，正常情形 scanf 會回傳成功讀取的資料數列，若遇到檔案結尾回傳 EOF，通常 EOF 的值是 -1，-1 取其 not 操作 ( $\sim$ ) 就是 0，因此也有部分選手會使用第三個迴圈的寫法<sup>i</sup>：

```
1 int n;
2 while (scanf ("%d", &n) != EOF) {
3     //do something
4 }
5 while (scanf ("%d", &n) == 1) {
6     //do something
7 }
8 while (~scanf ("%d", &n)) { //黑魔法(好啦應該還不算)
9     //do something
10 }
```

程式碼 2.18: scanf 判斷 EOF

<sup>i</sup>該方法不標準

### fget 判斷 EOF

fgets 如果成功讀到資料，會回傳傳入字串的指標，如果失敗如遇到 EOF 的話，就會回傳 NULL。

```
1 char s[1000005];
2 while(fgets(s, 1000000, stdin) != NULL) {
3     //do something
4 }
5 while(fgets(s, 1000000, stdin)) {
6     //do something
7 }
```

程式碼 2.19: fget 判斷 EOF

### cin 判斷 EOF

cin 比起前面的方法是相當的簡單，cin 如果讀取成功就回傳 true，失敗回傳 false，因此直接放在 while 裡面就可以了。cin.getline、getline 等也是一樣的用法。

```
1 while(cin >> n) {
2     //do something
3 }
```

程式碼 2.20: cin 判斷 EOF

## 2.3.2 字串串流

若已經處理好連續輸入的問題，剩下的就是要把每行中所有數字加起來，但是該怎麼做呢？

傳統的方法會利用 fgets、getline 等函數將資料讀成字串，然後用一些函數把每個數字拿出來。一些參考資料可能會使用 strtok 來取得資料，但是 strtok 使用上不太直觀，因此競賽中經常會用其他簡單好寫的方法操作。

### sscanf

這個函數位於 cstdio，可以把字串當當作串流的資料來進一步讀取，詳細用法與 scanf 雷同：

```
1 char s[100] = "13 456";
2 int a, b;
3 sscanf(s, "%d %d", &a, &b);
4 printf("%d %d\n", a, b); // 輸出：13 456
```

程式碼 2.21: sscanf

雖然這是這是一個解決字串切割的好方法，但要注意使用上的細節，若試著用 sscanf 來解 A+B+C... 問題：

```
1 fgets(s, 1000000, stdin);
2 int add=0, a;
3 for(int i = 0; s[i]; ++i){
4     if( isdigit(s[i]) && ( !i || s[i-1]== ' ' ) ) {
5         sscanf(s+i, "%d", &a);
6         add += a;
7     }
8 }
9 printf("%d\n", add);
```

程式碼 2.22: sscanf 解 A+B+C... 問題

實驗後會執行結果竟然會超出時限 TLE！究竟發生什麼事呢？因為 sscanf 每次執行的時候都會掃描過整個字串一遍，因此該函數的複雜度為  $\mathcal{O}(N)$ ， $N$  是字串長度，因此在無意中，我們就犯了與 for 迴圈的陷阱一樣的問題，這個迴圈會執行約  $10^6 \times 10^6$  次，總複雜度為  $\mathcal{O}(N^2)$ ，因此 sscanf 通常不會用來處理不固定數量的資料。

### stringstream

字串串流 stringstream 是 C++ 的一個重要的功能，使用時須 include sstream。字串串流可以把字串當成類似 C++ 輸入輸出串流來使用，是相當方便的方法，而且它不會每次執行就讀過整個字串。

```
1 string s = "13 456 6566 121 48 1545 878541 6846";
2 stringstream ss(s);
3 int a;
4 while(ss >> a) {
5     cout << a << '\n';
6 }
```

程式碼 2.23: stringstream 用法

stringstream 若要要重複使用，需要先使用 clear() 來清除舊的資料與標記，避免發生問題。加上這個動作，這樣原來的問題就可以用簡單的方法解決了！

```
1 string s;
2 stringstream ss;
3 while(getline(cin, s)) {
4     ss.clear();
5     ss.str(s);
6     int add=0, a;
7     while(ss >> a) add += a;
8     cout << add << '\n';
9 }
```

程式碼 2.24: stringstream 解 A+B+C... 問題

### 2.3.3 C++ 的 IO 優化

在一般測試題目時，會發現 C++ 的 cin, cout 處理大量資料會比 C 語言的 scanf, printf 還要來的緩慢，在部分機器上甚至可以有 10 倍的差距。造成這現象的原因有兩個，第一個是不了解 C++ 的輸入輸出機制，錯誤的使用造成效率緩慢，第二是為了與 C 語言相容的歷史因素。透過適當的調整並加上一些指令，就可以讓 C++ 與 C 有差不多的讀取速度。而且方法很簡單，只要在 main 函數一開始加上這兩行即可：

```
1 ios::sync_with_stdio(false);
2 cin.tie(0);
```

程式碼 2.25: IO 優化

C++ 為了相容 C 的串流機制，讓兩者的的輸入輸出混用不會出錯，所以做了一些同步處理，但是這樣的動作會讓效能變慢，只要加上ios::sync\_with\_stdio(false)；就可以取消同步，但是使用之後，就不可以再混用兩者的輸出方法。

```
1 cout<<"C++ IO\n"; //不一定會先輸出 C++ IO
2 printf("C IO\n");
```

程式碼 2.26: 混用輸出

tie是將兩個串流綁定的函數，目的為了讓輸入與輸出的動作一致化，預設 cin 和 cout 是綁定在一起的，使得使用 cin 時會把 cout 尚未輸出的資料輸出，但與使用 endl 一樣，會觸發 flush，此時只要加上cin.tie(0)，就可以解開 cin、cout 的綁定，但會有無法及時輸出資料的狀況，通常會等到程式結束時在一口氣輸出答案。有些人覺得如此比較容易 Debug，有些人覺得困擾，可以視情況選擇是否使用。

### 2.3.4 輸出浮點小數位數

通常遇到題目需要輸出小數時，可能會要求輸出到小數點後第 n 位，scanf 與 cout 的設定方法有點不一樣。和 printf 設定小數點位數可以使用標記設定；而 cout 要額外 include<iomanip>，用 fixed, setprecision 來設定，設定一次後持續有效！

```
1 cout << fixed << setprecision(5); // 設定一次就好
2
3 double a = 123.456, b = 456.789;
4
5 cout << a << ' ' << b << '\n'; // 輸出:123.45600 456.78900
6 printf("%.5f %.3f", a, b); // 輸出:123.45600 456.789
```

程式碼 2.27：設定小數點輸出位數

## 2.4 簡單 C++ STL 工具介紹

在整本書中經常的使用到 C++ 的標準樣板庫 (STL)，在資料結構中對於 STL 的容器 (container) 會有較詳細的介紹，這裡會先講一些對解題很有幫助的小工具。

C++ STL 為了應付各種資料型態，因此採用樣板 (template) 來實作，因此在使用這些工具時經常會需要在角括號 <> 中填入要使用的型態，例如 std::pair<int, std::string> 表示 pair 的第一個元素是 int，第二個元素是 C++ 的 string。

適當的使用 C++ STL 的工具可以精簡程式累贅的部分，使程式的可讀性增加，但若濫用或是對運作不夠熟悉，不但可能使程式的時間複雜度上升，可能還會出現難以除錯的 Bug，因此在比賽中並不建議使用不熟悉的工具。

### 2.4.1 std::pair

pair 位於標頭檔 utility 中，可以將兩個型態的資料合併，透過成員 first、second 來存取元素。若資料是可比較的，則 pair 以元素的字典序來比較，也就是先比較 first 相同的話再比較 second。pair 經常得用來取代一些比較簡單的結構 (struct)，透過 pair 定義的比較，避免重複撰寫自訂的比較函數。

字典序的比較是程式設計上經常使用的，如果有兩個序列  $A$ ,  $B$  要比較字典序的話，那會先比較  $A$  與  $B$  的第一個元素，若相同則比較第二的元素，直到有不同為止。除此之外，strcmp、std::string，與下個介紹的 tuple 也都使用了字典序的比較。

```

1 pair<int, double> P(1, 3.14), tmp; //可以在宣告時設定數值
2 // 設定數值的方法
3 P.first = 10;
4 P.second = 1.8;
5 P = make_pair(20, 2.7);
6 P = {30, 3.14}; // C++11
7 cout << P.first << ' ' << P.second << '\n';
8
9 pair<int, int> A, B;
10 A = {10, 18};
11 B = {13, 16};
12 if( A < B ) cout << "A<B\n"; // true
13 A = {10, 16};
14 B = {10, 15};
15 if( A < B ) cout << "A<B\n"; // false

```

**程式碼 2.28:** pair 的一些用法

## 2.4.2 std::tuple

若要包裝超過兩個以上的元素，除了把 pair 套入 pair 如 `pair<pair<int,int>,int>` 之外，C++11 提供了 tuple 可以進行任意多個元素的包裝。使用時須加入標頭檔 `tuple`。tuple 的提供的功能與 pair 差不多，但存取元素較為複雜，因為 tuple 有多個元素，並沒有辦法直接使用如 `first`、`second` 來取則資料，而是要透過 `std::get<i>()` 來得到第 *i* 個元素，元素從 0 開始編號。除了 `get` 之外，也可以使用 `tie` 函數來一次性地取得多向元素的資料。tuple 和 pair 一樣也有比較的功能，都是用字典序進行比較。

較舊的編譯器在使用 tuple 時會有實作規格的問題，無法直接使用大括號設定數值，測試環境時應注意。

```

1 tuple<int, int, int, int> a(1, 2, 3, 4), b;//可以在宣告時設定數值
2 // 設定數值的方法
3 a = make_tuple(1, 2, 3, 4);
4 a = {1, 2, 3, 4}; // c++14, g++ >=6
5 get<0>(a) = 5;
6 cout << "a0 = " << get<0>(a) << '\n';
7 // 一次拿取多項資料
8 int w, x, z;
9 tie(w, x, std::ignore, z) = a; // 利用 std::ignore 忽略不想拿的資料
10 cout << w << ' ' << x << ' ' << z << '\n';

```

**程式碼 2.29:** tuple 的一些用法

`tie` 是個神奇有用的東西，它可以把一些東西包裝成一個 tuple，可以用來快速用分解另一個 tuple 的資料。

### 2.4.3 std::vector

vector 位於標頭檔 vector 中，是 C++ 十分常使用的容器，可以當作是一個可以自動調整大小的陣列。vector 用和陣列一樣的方式存取資料，但功能更強大的是它可以從尾端一直新增資料 (emplace\_back, push\_back)，在 C++ 中經常用 vector 取代傳統陣列。

vector 會在有需要時擴增或縮減他的記憶體使用空間。為了避免每一次新增資料時都重新擴增一次，也就是開一個新陣列並將所有東西移過去，會預先新增多一點空間來應付未來可能會使用到的部分，我們通常會稱大小 (size) 為 vector 目前有幾個元素，容量 (capacity) 為 vector 目前不擴增的話能裝幾個元素。

```
1 vector<int> v;
2 v.resize(3);      // 設定vector大小為 3
3 v[0] = v[1] = 2; // 當作一般陣列來使用
4 v[2] = 6;
5 v.emplace_back(7); //新增一個元素到vector後，現在有 4 個元素
6 cout << "v 有 " << v.size() << "個元素\n"; // 4 個
7 v.clear(); // 清除所有資料
8 cout << "v 有 " << v.size() << "個元素\n"; // 0 個
9 cout << "v 的容量為" << v.capacity() << "\n";
10 v.shrink_to_fit(); // 將 vector 的容量縮成剛好 size 的大小
```

程式碼 2.30: vector 的一些用法

在 C++11 以後，新增了 emplace\_back 的操作，有別於原有的 push\_back，emplace\_back 使用了 C++ 新的語法架構來處理資料，尤其搭配 tuple 使用時，就可以利用 emplace\_back 的特性快速的新增資料。

```
1 vector<tuple<int, int, int>> data;
2 //C++11以前要寫 >>，兩個 > 不能連在一起
3
4 data.emplace_back(1, 2, 3); // 很簡短的寫法
5
6 data.push_back({1, 2, 3}); // C++14 g++ >=6
7 data.push_back(make_tuple(1, 2, 3));
```

程式碼 2.31: vector 與 tuple

vector 的操作 push\_back、emplace\_back 是均攤  $\mathcal{O}(1)$ ，使用上與一般的陣列操作速度差不多快，而因為 emplace\_back 會直接呼叫建構子不會複製物件，所以大部分時候的執行效率會比 push\_back 快。在使用 push\_back 時資料需要能夠被複製，而使用 emplace\_back 則型態需要有對應的建構子 (constructor)。

| 成員函數                         | 作用                                  | 時間複雜度            |
|------------------------------|-------------------------------------|------------------|
| <code>resize(num, d)</code>  | 將陣列大小設為 num ·<br>如果有新增元素以 d 初<br>始化 | $\mathcal{O}(N)$ |
| <code>push_back(x)</code>    | 把 x 複製到 vector 的尾端                  | $\mathcal{O}(1)$ |
| <code>emplace_back(x)</code> | 把 x 建構到 vector 的尾端                  | $\mathcal{O}(1)$ |
| <code>pop_back()</code>      | 把尾端的元素移除                            | $\mathcal{O}(1)$ |
| <code>v[6]</code>            | 如同陣列的存取元素                           | $\mathcal{O}(1)$ |
| <code>clear()</code>         | 元素全部清空                              | $\mathcal{O}(N)$ |
| <code>size()</code>          | 回傳元素的個數                             | $\mathcal{O}(1)$ |

#### 2.4.4 std::bitset

細心的讀者可能會發現 `bool` 這個型別明明只能表示 `true` 或 `false`，通常卻佔了 1 Byte 的記憶體空間。這主要的原因是因為電腦的架構通常不需要也很難去操作 1 bit 的空間。單純宣告一個 `bool` 影響可能不大，但若是宣告 `bool` 陣列可能會造成很多記憶體的浪費。因此 STL 就包含一個好用的工具 `bitset` 可以宣告固定長度的 bits。

基本上你可以把 `bitset` 當成一個效率很快的 `bool` 陣列，而越低位是在 `bitset` 的越右邊。`bitset` 也支援位元運算。也能將 `bitset` 轉換成 `string` or `unsigned integer`。通常在 `bitset` 上做操作的時間是 `vector<bool>` 的  $\frac{1}{32}$  倍。

```

1 bitset<5> b; // 宣告大小為 5 的 bitset 初始為 00000
2 b[0] = 1; // 將第 0 個 bit 設為 1 也就是 bitset 變成了 00001
3 b.set(); // set all bits to 1
4 b.reset(); // set all bits to 0
5 cout << b.count() << '\n'; // 輸出 bitset 中有幾個 1
6 bitset<5> a(10011);
7 cout << (a & b) << endl; // 因為運算子優先順序的關係，() 是必須的
8 string str = a.to_string();
9 unsigned long val = a.to_ulong();

```

程式碼 2.32: `bitset` 的一些用法

以上的只介紹了他們的常用用途，更詳細的內容歡迎讀者參考 [cppreference](#) 等等網站。

## 2.5 其他常見小技巧

在演算法競賽中，用簡單的 code 寫出高效率的程式是很重要的，因此常常會有一些小技巧讓選手們在比賽的時候更能快速地寫出程式。在此會先將書中使用的非基礎或特別的語法做講解與說明。

### 2.5.1 while(T--)

這樣迴圈就會剛好執行 T 次，對於很多題目要求來說是很方便的。

```
1 while(T--) {  
2     cout<<T<<'\n';  
3 }
```

程式碼 2.33: T--

### 2.5.2 全域變數

不知道讀者有沒有過一個經驗是，一份 code 上傳到 OJ 會錯，後來將陣列開在全域就 AC 了。會造成這種情況的原因是系統在分配記憶體空間時，給區域變數的空間會比全域的小，儘管你宣告的區域變數的大小沒有超過記憶體空間上限，仍然會造成程式錯誤。因此建議大的陣列都開在全域以避免這種錯誤。

### 2.5.3 型態轉換

有些時候小的型態轉換成大的型態、整數轉浮點數要做一些強制轉型才不會出錯，但是這樣 code 的長度會變得很長，這時候可以好好利用下“乘以 1”這個操作。

```
1 int      a = 1, b = 2;  
2 long long c = 1LL * a * b; // 1LL 是 long long  
3 double   d = 1.0 * b / a; // 1.0 是 double，注意 1.0f 是 float  
4 unsigned  e = 1u * a * 2; // 1u 是 unsigned  
5 unsigned long long f = 1LLU * a * a; // 1LLU 是 unsigned long long  
6 long long g = (1LL << 60) // 若寫 1 << 60 會出問題
```

程式碼 2.34: 簡單的型態轉換

### 2.5.4 define 小技巧

我們可以把一些較長且常用的東西用 define 成比較短的符號，像是要用 for 的時候就可以用更短的 REP 來表示，像是以下的 code 為 floyd 最短路徑演算法的三層 for。

```
1 #define REP(i, n) for(int i=0; i<int(n); ++i)  
2  
3 REP(k, n) REP(i, n) REP(j, n)  
4     if( g[i][k] + g[k][j] < g[i][j] )  
5         g[i][j] = g[i][k] + g[k][j];
```

程式碼 2.35: define REP

或者因為 pair 的 first 和 second 太長了，也時候會用 x 和 y 去取代：

```
1 #define x first  
2 #define y second
```

程式碼 2.36: define pair

這樣是不是很方便呢？當然這樣的寫法僅限於比賽選手間的交流，但通常的程式開發不會接受如此寫法，要多加注意。

## 2.5.5 重新定義型別名

在 C++ 使用模板時，經常會遇到非常長的型態名稱，或是覺得 `long long` 有點太長，不太方便，這時通常會幫這些型態取一個比較短別名來使用。C++11 以後，可以使用 `using A = B;` 來幫 B 取別名為 A。若是 C 語言，則要使用 `typedef B A;` 順序上是恰好顛倒的，要特別注意。

方便起見，這本講義在往後的章節會使用 ll 或 LL 來代表 `long long`。

```
1 using LL = long long; // C++11  
2 using tidii = tuple<int, double, int, int>; // tuple 的名稱通常很長  
3  
4 typedef long long ll; // 舊式寫法
```

程式碼 2.37: 定義型態別名

## 2.5.6 auto

C++11 以後，可以使用 `auto` 自動判斷型態，像是 `vector` 的 `begin()` 和 `end()` 回傳的型態名字是非常長的，使用 `auto` 可以大大減少要記憶的名稱。

```
1 vector<int> V {10, 20, 30};  
2  
3 auto ptr = V.begin(); // vector 的迭代器，用 auto 判斷型態  
4 auto end = V.end(); // 不用 auto 的話型態是 vector<int>::iterator  
5  
6 while(ptr != end) cout << *ptr++ << ' '; // 輸出 10 20 30
```

程式碼 2.38: auto

## 2.5.7 reference

C++ 新增了 `reference`(參考)，功能與指標類似，但是使用起來較平易近人。宣告 `reference` 基本上就只是宣告變數的別名，對他進行操作會也改到原本的物件，宣告方法即是在型別後面加上 & 符號。

```
1 int a = 3;
2 int& b = a;
3 b = 5; // a, b 同時改成 5
4 a = 4; // a, b 同時改成 4
```

程式碼 2.39: reference

我們也能將函數的參數定義成 reference，這樣在函數內改變該變數的值能直接影響到原本傳進來的變數。以下以 swap 為例。

```
1 void swap(int& a, int& b) {
2     int tmp = a;
3     a = b;
4     b = a;
5 }
```

程式碼 2.40: swap

若函數本身需要傳入資料量大的變數如 vector、string 等等，可以的話盡量使用 const reference 以避免整個複製建構一個物件造成時間與空間的耗損。

## 2.5.8 Range-based for

C++11 後，for 迴圈的種類多了一種稱為 Range-based for 的迴圈，透過元素的迭代器逐一個把所有的元素都存取一遍，不需要透過陣列索引 index 或是指標，善加使用能大量的使程式變得精簡好看。以下範例的兩個 for 迴圈的功用是相同的：

```
1 vector<int> E = {1, 2, 3, 4, 5, 6};
2 for(auto it = E.begin(), end = E.end(); it != end; it++) {
3     int u = *it;
4 }
5 for(auto u : E) {
6 }
```

程式碼 2.41: Range-based-for 與一般 for 比較

如果有需要修改資料的話，則需要使用 reference。

```
1 vector<int> E = {1, 2, 3, 4, 5, 6};
2 for(auto it = E.begin(), end = E.end(); it != end; it++) {
3     int &u = *it;
4 }
5 for(auto &u : E) {
6 }
```

程式碼 2.42: Reference Range-based for

如果遍歷的元素型態是 pair、tuple 的話，C++17 以後可以利用 Structured binding 來直接拆解元素，不需要利用 tie 來分解。由於 pair、tuple 的資料量較大，使用時要盡量避免多餘複製，以提升速率，因此通常會把變數設定成 const reference 來優化速度，此方法在 C++ 十分常見。

```
1 vector<tuple<int,int>> E;
2 int a, b;
3
4 for(const auto &p : E) //C++11
5 {
6     tie(a,b) = p;
7 }
8
9 for(const auto &[c,d] : E) //C++17
10 {
11     // ...
12 }
```

程式碼 2.43: Tuple in Range-based for

### 2.5.9 std::tie compare

C++11 之後的 tuple 可以做到字典序比較的功能，如果我們要自定義一些 struct 然後寫它們的比較函數的話，也可以使用 tie 或是 make\_tuple 把它們包成 tuple 之後再利用 tuple 的小於來進行比較，如果只想比較兩個東西的話用 pair 也可以喔！

```
1 //剛剛有講過tuple可以做字典序比較
2 //我們可以用make_tuple和tie來方便做出字典序比較的功能
3 struct P{
4     int a, b, c, d;
5 };
6 //只想根據a, b來比較
7 bool cmp1(const P &A, const P &B) {
8     if(A.a != B.a) return A.a < B.a;
9     return A.b < B.b;
10 }
11 bool cmp2(const P &A, const P &B) {
12     return std::tie(A.a, A.b) < std::tie(B.a, B.b);
13 }
14 bool cmp3(const P &A, const P &B) {
15     return std::make_tuple(A.a, A.b) < std::make_tuple(B.a, B.b);
16 }
17 bool cmp4(const P &A, const P &B) { // pair也可以喔
18     return std::make_pair(A.a, A.b) < std::make_pair(B.a, B.b);
19 }
20 bool cmp5(const P &A, const P &B) { //神奇寫法
21     return A.a < B.a || (A.a == B.a && A.b < B.b );
22 }
```

程式碼 2.44: 各種相同功能的比較函數

## 2.5.10 行尾空白

在某些題目會需要輸出陣列的數值後，要求在元素間加上空白，不過最後一個元素後面有沒有空白呢？雖然在多數的 Online Judge 以及 IOI 類的比賽是不理會最後的空白的，但是在較嚴格的題目或 ACM 比賽中，這樣的輸出是不正確的，因此會有很多方法來精簡這種細節處理，下面是一種神奇的做法：

```
1 int a[] = {1, 2, 3, 4};  
2 int N = 4;  
3  
4 for(int i=0; i<N; ++i)  
5 {  
6     cout << i[a] << " \n"[i==N-1]; // 注意 i[a] 是對的語法歐  
7 }
```

程式碼 2.45: default code

由於 C++ 的陣列語法  $a[b]$  等於  $*(\&a + b)$ ，這寫法又等於  $*(\&b + a)$ ，因此  $b[a]$  也是正確的寫法，只是很少見而已。後面空白語換行轉換成  $*(\&a + b)$ ，其實就能看得出這個程式碼可以根據 bool 的 0 與 1 選擇要輸出的字元，非常精妙。

## 2.5.11 函數物件與匿名函數

在 C++ 中，有時會把函數寫成物件，透過重載 `operator()` 來把物件模擬成函數，提供給其他的程式使用，這類型的函數稱為函數物件，在使用 C++ STL 時是常見的技巧，適當使用函數物件可以用來減少全域變數氾濫的問題。

```
1 struct FunctionObject {  
2     int operator() (int a) {  
3         return a * a;  
4     }  
5 };  
6  
7 FunctionObject func;  
8 int ans = func(5);  
9 cout << ans ; //25
```

程式碼 2.46: 函數物件

而除了函數物件之外，C++11 以後提供了匿名函數 lambda 可用來快速的建立函數物件，讓程式碼的撰寫更加靈活。在比賽中，常見的函數物件應用通常在重新定義排序的比較方法。

```
1 vector<int> A{3,1,2};
2 auto lambda = [](int a,int b) {
3     return a > b;
4 };
5 sort(A.begin(), A.end(), lambda); // A = { 3,2,1 }
6 sort(A.begin(), A.end(), [](int a, int b){    // 等同上面寫法
7     return a > b
8 });

```

程式碼 2.47: 簡易 lambda 應用

### 2.5.12 include 所有 Library

在 GNU 底下提供一個標頭當叫做 bits/stdc++.h · include 它之後就等於 include 所有常用標頭檔 · 基本上所有在比賽中會用到的東西它都會 include 進去。這不是標準的 · 雖然大多數的比賽環境都能使用 · 因此測機的重要性就在這裡了 · 務必在賽前做測試！以下附上一個常見的 default code 。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     ios::sync_with_stdio(false);
6     cin.tie(0);
7
8     return 0;
9 }
```

程式碼 2.48: default code



# 基本技巧

## 3.1 遞迴-河內塔

遞迴是許多初學者較不熟悉陌生的概念，經常這個項目卡關，但是這個方法經常地出現在資訊的許多應用上，尤其以動態規劃的章節，即是以遞迴為基礎，來建造出新的演算法概念，在本章將帶領讀者重新認識遞迴。很多時候寫不出遞迴的主要原因在於沒有「勇氣」，思考時，首先要先想清楚函數的功能是什麼，不只遞迴函數，寫一般函數也要想清楚，再來是要看看這個函數要解決的問題能不能被分解成更「小」的問題來處理。當在遞迴自己呼叫自己時，必須相信這個函數是可以正確被執行的，再寫下這個指令的當下才會清楚的知道自己在幹什麼。

以下就用遞迴的經典題 - 河內之塔來教大家如何真正去理解遞迴。

| 河內之塔   | Zerojudge a227 |
|--|----------------|
| 有三根杆子 A,B,C · A 杆上有 $N(N > 1)$ 個穿孔圓盤 · 盤的尺寸由下到上依次變小 · 要求按下列規則將所有圓盤移至 C 杆 : |                |
| 1. 每次只能移動一個圓盤<br>2. 大盤不能疊在小盤上面   |                |
| 為了方便起見，盤子的編號由小到大/由上往下依序為 $1 \sim N$  |                |

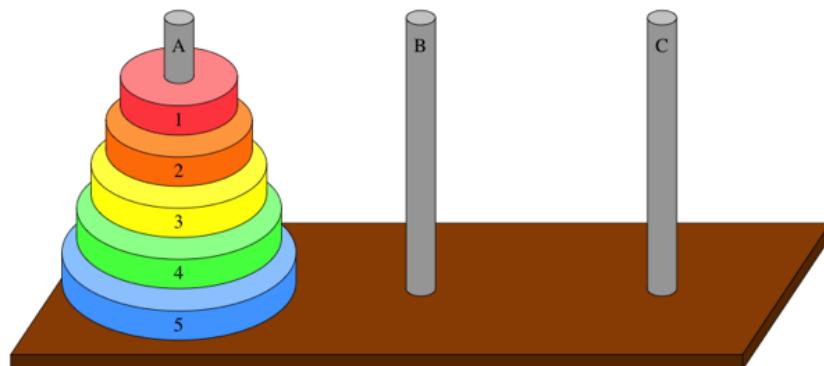
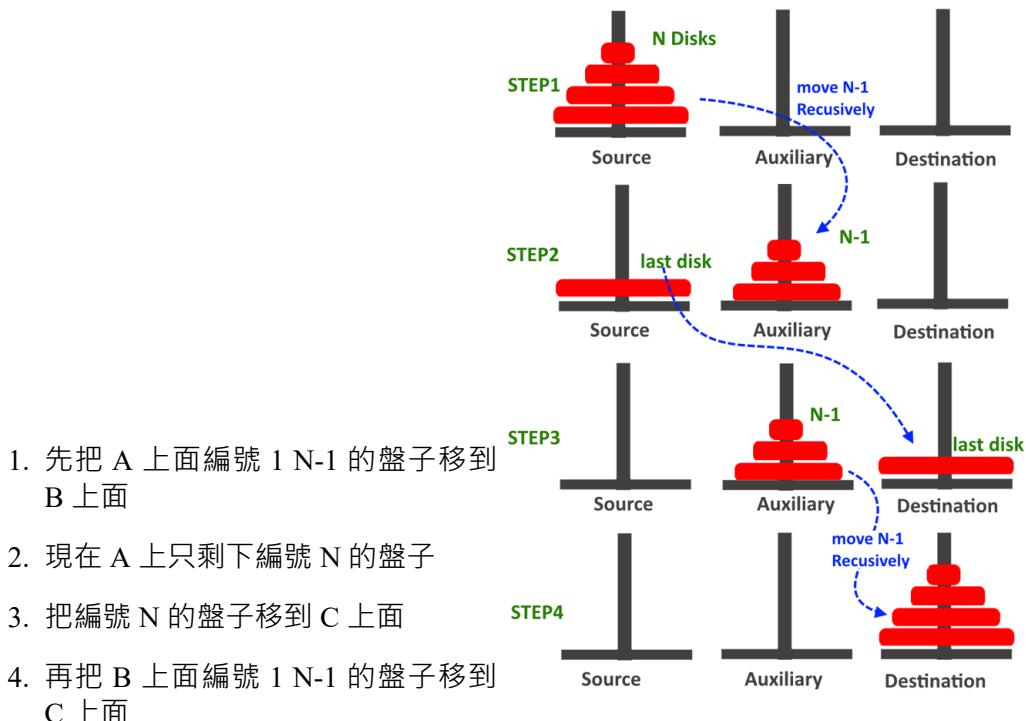


圖 3.1: 河內塔

### 3.1.1 想法拆解

可以發現 A 桿最底下最大的盤子只有在所有比它小的盤子都不再 A 桿它才能被拿起來，而且因為其他盤子都比它小，要把它移動到別的桿子則那根桿子上面不能有其他盤子，意思是如果要把最大的盤子從 A 移到 C 的話，必須把其他盤子先移到 B 才能行動，等到把最大的盤子移到 C 了，還要把其它盤子從 B 移到 C，可以得出以下步驟：



### 3.1.2 函數設計

我們要寫一個輸入  $N$ ，可以輸出把  $N$  個盤子從 A 移到 C 的方法的函數。在設計函數的時候要注意有哪些資訊是必要的。

定義 `hanoi(int n, char A, char B, char C)` 表示輸出有  $n$  個盤子，從 A 桿「透過 B 桿」移動到 C 桿的方法的函數。

我們試著用這個函數來表示剛剛的想法：

- 先把 A 上面編號  $1 \sim N - 1$  的盤子移到 B 上面 (透過 C)  $\Rightarrow \text{hanoi}(n-1, A, C, B)$
- 再把 B 上面編號  $1 \sim N - 1$  的盤子移到 C 上面 (透過 A)  $\Rightarrow \text{hanoi}(n-1, B, A, C)$

雖然這時候還沒開始寫函數，但是仍然要相信剛剛設計的函數是可以被使用的，這和在一些大型專案的分工一樣，必須要相信夥伴寫的函數是對的（雖然他可能連寫還沒開始寫）。

### 3.1.3 遞迴停止條件

有了遞迴的方法，但總不能讓函數無限遞迴下去吧，因此還需要為函數加上停止條件。以這題來說還蠻明顯的，就是只有一個盤子時候，直接將它從 A 移動到 C 即可；或是可以想成在沒有盤子的時候不做任何事，很顯然後者的程式碼比較容易實作。

將以上的想法綜合起來就能得到一份完整的程式：

```
1 #include<csdio>
2 void hanoi(int n, char A, char B, char C) {
3     if( n == 0 ) return;
4     hanoi(n-1, A, C, B);
5     printf("Move ring %d from %c to %c\n", n, A, C);
6     hanoi(n-1, B, A, C);
7 }
8 int main(){
9     int n;
10    while(scanf("%d", &n) != EOF) {
11        hanoi(n, 'A', 'B', 'C');
12    }
13 }
```

程式碼 3.1：河內之塔

## 3.2 快速幕與最大公因數

面對一個問題時，想出一個演算法通常並不困難，困難的是想出一個好的演算法。以下以快速幕與最大公因數為例，帶領讀者一步一步優化他們。

### 3.2.1 快速幕

幕運算，也就是指數運算，是在計算次方。今天要設計一個演算法去計算  $a^b$ ，多數人的第一直覺就是一直乘  $a$  乘  $b$  次就好了，寫成函式如下：

```

1 int power(int a, int b) {
2     int res = 1;
3     for(int i = 0; i < b; i++) {
4         res *= a;
5     }
6     return res;
7 }
```

**程式碼 3.2:** intuitional power

這個演算法確實能回傳正確的答案，其時間複雜度為  $\mathcal{O}(b)$ 。但難道沒有更好的算法了嗎？我們不能安逸於此！根據指數的性質，我們知道當  $b$  為偶數時  $(a^{b/2})^2 = a^b$  ·  $b$  為奇數時  $a \times a^{b-1} = a^b$  · 因此我們可以設計出如下的遞迴函式。

```

1 int power(int a, int b) {
2     if(b == 0) return 1;
3     if(b % 2) return a * power(a, b-1);
4     return power(a, b/2) * power(a, b/2);
5 }
```

**程式碼 3.3:** power

這一函式看似通常都能將  $b$  除以 2 · 但實際計算一下總共會呼叫幾次 power 函數 · 你會發現撇除掉奇數的情況 · 每呼叫一次 power 會再遞迴呼叫 2 次 power · 而將  $b$  一直除以 2 只能除  $\log b$  次 · 因此至少會呼叫 power 函式  $2^{\log b} = b$  次 · 實際上該函式的時間複雜度確實正是  $\mathcal{O}(b)$  。

而上述的程式碼 · 你會發現  $\text{power}(a, b/2)$  沒有必要呼叫 2 次因為他們的值是一樣的 · 因此可以先用一個變數存起來 · 再相乘即可 · 這個小步驟能讓複雜度成功提升至  $\mathcal{O}(\log b)$  · 却也是許多初學者會忽略的一步。

```

1 int power(int a, int b) {
2     if(b == 0) return 1;
3     if(b & 1) return a * power(a, b-1);
4     int tmp = power(a, b/2);
5     return tmp * tmp;
6 }
```

**程式碼 3.4:** recursive fast power

以下提供一個快速幕的非遞迴版本寫法 · 概念即是將  $b$  以二進位表示 · 對於所有  $i$  · 當  $b$  的第  $i$  個 bit 是 1 時 (0-based) · 將答案乘上  $a^{2^i}$  。

```

1 int power(int a, int b) {
2     int res = 1;
3     while(b) {
4         if(b & 1) res *= a;
5         a *= a;
6         b >>= 1;
7     }
8     return res;
9 }
```

程式碼 3.5: fast power

另外要提的是在 `<cmath>` 內的 `pow` 函數，雖然也能計算  $a^b$ ，但其效率並不高。不過對於計算分數次方如  $a^{1/3}$  等等，可以使用他，其參數與回傳值都是浮點數。

### 3.2.2 最大公因數

要設計一個演算法去計算  $a, b$  的最大公因數，最直觀的想法是枚舉所有可能的因數。

```

1 int gcd(int a, int b) {
2     for(int i = min(a, b); i >= 2; i--) {
3         if(a % i == 0 && b % i == 0) return i;
4     }
5     return 1;
6 }
```

程式碼 3.6: intuition gcd

不失一般性假設  $a \leq b$ ，那其時間複雜度便為  $\mathcal{O}(a)$ 。但是我們希望能有更快的演算法去求出最大公因數！我們能夠想到一個以前學過的東西叫做輾轉相除法。輾轉相除法告訴我們**兩個整數的最大公因數等於「其中較小的數和兩數的差的最大公因數」**。因此我們可以寫出一個遞迴函式如下。

```

1 int gcd(int a, int b) {
2     if(a > b) swap(a, b);
3     if(b % a == 0) return a;
4     return gcd(a, b-a);
5 }
```

程式碼 3.7: slow gcd

這個函式能用較快的速度計算如 206 和 100 的最大公因數，並不需要從 100 開始枚舉到 1 來判斷。但對於極端情況如  $gcd(2, 2000000001)$ ，遞迴呼叫 `gcd` 這個函式的次數高達  $10^9$  左右！我們觀察上述的極端情況，遞迴呼叫 `gcd` 時， $a$  一直都沒有變化，反倒是  $b$  一直減  $a$  直到  $b < a$  為止，因此我們可以發現與其一直減不如直

接用模運算求出  $b$  一直減  $a$  直到  $b < a$  的時候其數值是多少，而且一個數字被有效取餘時，新的值至多為原來的一半，所以能穩定且快速的求出答案。最終我們能統整出下列的寫法，其時間複雜度約為  $\mathcal{O}(\log a)$ 。

```
1 int gcd(int a, int b) {  
2     return b == 0 ? a : gcd(b, a % b);  
3 }
```

程式碼 3.8: gcd

## 3.3 常見的排序法

### 3.3.1 Merge Sort

#### 淺談分治

困難的大問題時常可以透過遞迴的方式拆解為規模較小的幾個子問題，分別解這些子問題後，再透過子問題的答案回推出原本大問題的答案。這樣的作法稱之為「分治 (Divide and Conquer)」。一般來說分治可以分為以下部份：

- Divide：將大問題拆解成數個相似的子問題
- Conquer：分別遞迴求出子問題的答案
- Combine：合併子問題的答案以得到原本大問題的答案

Merge Sort 是經典的分治。排序大陣列不容易，但是將兩個排序好的小陣列合併成一個排序好的大陣列卻很容易。我們可以使用兩個指針分別先指向兩個排序好的陣列的第一個元素，比較指針上哪個元素較小，將其放入合併完的空間同時指針往後移，重複此步驟便能快速地合併兩個排序好的序列。

Merge Sort 執行時將陣列切割成等長的左右兩半，遞迴排序兩半後得到兩個已經排序好的陣列，最後將兩半陣列合併。由於每次將陣列切成等大小的兩半，因此遞迴深度  $\mathcal{O}(\log n)$ ，在每層遞回合併須花  $O(n)$  的時間，因此時間複雜度為  $\mathcal{O}(n \log n)$ 。

```

1 void mergeSort(int *arr, int len)
2 {
3     if(len <= 1) return;
4     // divide, conquer
5     int leftLen = len/2, rightLen = len-leftLen;
6     int *leftArr = arr, *rightArr = arr+leftLen;
7     mergeSort(leftArr, leftLen);
8     mergeSort(rightArr, rightLen);
9     // combine
10    static int tmp[MAX_N];
11    int tmpLen = 0, l = 0, r = 0;
12    while(l < leftLen && r < rightLen)
13        if(leftArr[l] < rightArr[r]) tmp[tmpLen++] = leftArr[l++];
14        else tmp[tmpLen++] = rightArr[r++];
15    while(l < leftLen) tmp[tmpLen++] = leftArr[l++];
16    while(r < rightLen) tmp[tmpLen++] = rightArr[r++];
17    for(int i = 0; i < tmpLen; i++) arr[i] = tmp[i];
18 }

```

程式碼 3.9: Merge Sort

一般 Merge Sort 常見的實做方式在合併兩個已排序陣列時會多開  $\mathcal{O}(n)$  的空間。另外也有不用多開  $\mathcal{O}(n)$  空間的實做方式，有興趣的同學可以自行上網搜尋（關鍵字：In-Place Merge Sort）。

### 逆序數對

$(a_i, a_j)$  是個逆序數對  $\iff i < j$  且  $a_i > a_j$ 。逆序數對的概念常常會跟排序題一起出現。

算逆序數對數量其中一個方法是在 Merge Sort 的過程中算出有幾組逆序數對。當左右兩半分別是遞增數列的時候，可以利用單調性很快的算出有幾對逆序數對。假設兩半數列分別為長度為  $n$  的  $A$  和  $B$ 。當某個  $A_i > B_j$  時，因為  $A_i \leq \dots \leq A_n$ ，所以  $A_i$  到  $A_n$  都可以和  $B_j$  組成逆序數對。根據這個想法，我們可以在 Merge Sort 的過程中求得逆序數對的數量，複雜度  $O(n \log n)$ 。算逆序數對數量也可以搭配適當資料結構，將在資料結構課程中介紹。

### 3.3.2 Quick Sort

Quick Sort 的執行時先選擇一個基準 (pivot)，整理陣列使得基準的左側元素都小於基準，右側元素都大於等於基準，接著遞迴處理左右兩側。Quick Sort 的執行

效率取決於每次選擇到的基準，隨機選擇基準時平均而言時間複雜度為  $\mathcal{O}(n \ln n)$ ，但是在最糟糕的情況要花  $\mathcal{O}(n^2)$  的時間排序陣列。

```
1 void quickSort(int *arr, int len)
2 {
3     if(len <= 1) return;
4     // divide
5     int pivotIdx = rand()%len, pivot = arr[pivotIdx];
6     swap(arr[len-1], arr[pivotIdx]);
7     int leftLen = 0;
8     for(int i = 0; i < len; i++)
9         if(arr[i] < pivot) swap(arr[i], arr[leftLen++]);
10    swap(arr[leftLen], arr[len-1]);
11    int rightLen = len-leftLen-1;
12    int *leftArr = arr, *rightArr = arr+leftLen+1;
13    // conquer
14    quickSort(leftArr, leftLen);
15    quickSort(rightArr, rightLen);
16 }
```

程式碼 3.10: Quick Sort

### 3.3.3 Counting Sort

給定一個大小為  $n$  的陣列  $arr$ ，且  $arr$  中每個元素的範圍介於  $0 - k$  之間，我們可以透過以下方法在  $\mathcal{O}(n + k)$  的時間內完成排序：另外開一個大小為  $n$  的陣列  $result$  用來存放排序後的陣列，以及開一個大小為  $k + 1$  的陣列  $cnt$ ，紀錄  $0 - k$  在陣列  $arr$  中各出現幾次。接著對  $cnt$  做前綴和，此時對於值為  $i$  的元素來說， $cnt[i]$  值為  $\leq i$  的元素有幾個，進而推算出  $i$  應該在排序後的陣列中應該擺在哪個位置。最後由  $arr$  的最後一個元素開始看起，透過  $cnt$  找到該元素的位置，並填入  $result$  中。

```
1 vector<int> countingSort(const vector<int> &arr)
2 {
3     int n = arr.size();
4     int k = *max_element(arr.begin(), arr.end());
5     vector<int> result(n);
6     vector<int> cnt(k + 1, 0);
7     for (int i = 0; i < n; i++) cnt[arr[i]]++;
8     for (int i = 1; i <= k; i++) cnt[i] += cnt[i-1];
9     for (int i = n - 1; i >= 0; i--)
10    {
11        int x = arr[i];
12        result[--cnt[x]] = x;
13    }
14    return result;
15 }
```

程式碼 3.11: Counting Sort

## 穩定排序與不穩定排序 Stability

Stability 是排序演算法中一個重要的性質。如果一個排序是穩定 ( Stable ) 的，對於任兩個值相同的元素而言，排序前後的順序不會改變。以表3.3的範例來說，每個學生有身高和體重，如果單單只對身高穩定排序後，保證 A 的位置仍然在 B 前面。如果一個排序是不穩定的 ( Unstable ) 的，對於任兩個值相同的元素而言，排序前後順序可能會改變。

| 學生 | 身高  | 體重 |
|----|-----|----|
| A  | 171 | 88 |
| B  | 171 | 70 |
| C  | 180 | 77 |
| D  | 160 | 63 |
| E  | 200 | 95 |

表 3.1: 穩定排序前

| 學生 | 身高  | 體重 |
|----|-----|----|
| D  | 160 | 63 |
| A  | 171 | 88 |
| B  | 171 | 70 |
| C  | 180 | 77 |
| E  | 200 | 95 |

表 3.2: 穩定排序後

表 3.3: 穩定排序範例

要判斷一個排序法是否 Stable 需要看排序法的實做方式而定。一般來說，Quick Sort 不是 Stable Sort，而 Merge Sort 和 Counting Sort 為 Stable Sort。

### 3.3.4 Radix Sort

給定一個大小為  $n$  的陣列，陣列中的元素以  $k$  進制表示且至多為  $d$  位數，可以由最低位開始針對每個位數做 Stable Sort 以排序整個陣列。通常實做時會用 Counting Sort 來排序每個位數，在這個情況下時間複雜度為  $\mathcal{O}(d(n + k))$ 。

```

1  vector<int> stableSort(vector<int> &rank, const vector<string> &arr, int
2    digit)
3  {
4    const size_t K = 26;           // lower case alphabets
5    vector<int> result(arr.size());
6    vector<int> cnt(K, 0);
7    for(auto s:arr) cnt[s[digit]-'a']++;
8    for(size_t i = 1; i < K; i++) cnt[i] += cnt[i-1];
9    for(int i = arr.size()-1; i >= 0; i--)
10   {
11     int x = arr[rank[i]][digit]-'a';
12     result[--cnt[x]] = rank[i];
13   }
14   return result;
15 }
16
17 vector<int> radixSort(const vector<string> &arr)
18 {
19   vector<int> rank(arr.size());
20   size_t d = 0;
21   for(size_t i = 0; i < arr.size(); i++)
22   {
23     d = max(d, arr[i].size());
24     rank[i] = i;
25   }
26   for(int i = d-1; i >= 0; i--) rank = stableSort(rank, arr, i);
27   return rank;
}

```

程式碼 3.12: Radix Sort

### 3.3.5 std::sort()

C++ STL 內建排序函式採用 Intro Sort，複雜度保證  $\mathcal{O}(n \log n)$ 。有興趣的學員可自行上網搜尋。

`std::sort()` 被定義於 `<algorithm>` 標頭檔中。`std::sort()` 的三個參數分別為開頭指標、結尾指標、比較函式。排序範圍不包括結尾指標，比較函式可不傳入，預設為遞增排序（排序後由小到大）。

```

1 int arr[] = { 7, 1, 2, 2 }, len = 4;
2 vector<int> vec = { 9, 4, 8, 7 };
3
4 std::sort(arr, arr+len);           // 1 2 2 7
5 std::sort(vec.begin(), vec.end()); // 4 7 8 9

```

程式碼 3.13: `std::sort()` 最基本用法

當我需要遞減排序或根據自定義規則排序時，須自行實做比較函數傳入 `std::sort()`。若 `cmp(a, b)` 回傳 `true`，表示 `a` 需排在 `b` 的前面。

```
1 #include <algorithm>
2 struct Pt { int x, int y; }
3 bool cmp(Pt &a, Pt &b)
4 {
5     // 先照x遞增排序，如果x相同則照y遞增排序
6     return a.x < b.x || (a.x == b.x && a.y < b.y);
7 }
8 int main()
9 {
10    vector<Pt> points;
11    std::sort(points.begin(), points.end(), cmp);
12 }
```

程式碼 3.14: 自定義排序函式

自定義排序規則的方法不只實做比較函式，還可以用 `lambda` ( C++11 以上 )、多載小於運算子等等。有興趣的同學可自行上網查詢。

`std::sort()` 並不是 `stable sort`。如果需要使用 `stable sort`，STL 中也內建 `std::stable_sort()`，用法和 `std::sort()` 相同。

## 3.4 暴力枚舉

### 3.4.1 簡單枚舉

枚舉就是嘗試所有可能，從中算出想要的東西，俗稱暴力。好處是想法非常直觀，壞處是在資料量很大的時候，枚舉常常曠日費時。對於一些比較簡單的題目，如果觀察一下發現他只有有限種可能，而且總數不多的時候，枚舉不失為解題(拿部份分)的好辦法。

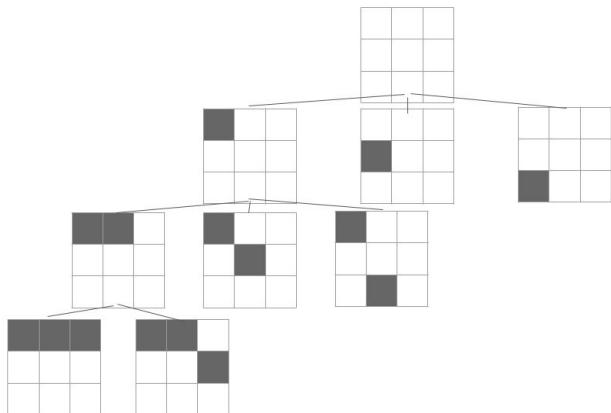
| 三角形   | 經典例題 |
|---|------|
| 給定 $N(N \leq 100)$ 根棒子，從中取出三根組成的所有三角形中，最大的周長多少？ |      |

最直觀的想法，分別枚舉三角形的三個邊，再判斷是否能組成三角形(最長的棒子  $<$  其他兩根棒子長度和)，每找到一個三角形就更新當前所找到的最大周長。這樣得作法的複雜度為  $\mathcal{O}(N^3)$ ，在這個數字範圍下不會超時。這題有  $\mathcal{O}(N^2 \log N)$  的解法，需要用到二分搜尋的觀念，建議各位學完二分搜尋後回頭想想這題。

### 3.4.2 深度優先搜尋

|   |      |
|---|------|
| 八皇后問題   | 經典例題 |
| 在一個 $n \times n$ 的西洋棋盤上擺 $n$ 隻皇后，請問有多少擺法使得 $n$ 隻皇后不會互吃？(i.e. 任兩隻皇后不在同一個橫線、直線、斜線上) |      |

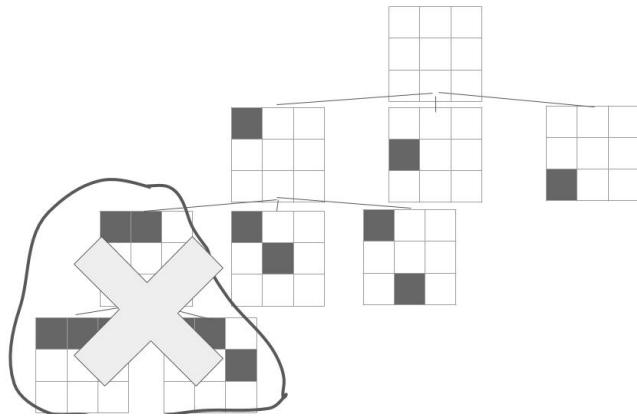
最直觀的想法：先找一個位置擺上一個皇后，再從上下的  $n^2 - 1$  個位置中選一個擺第二隻皇后……直到擺完  $n$  隻皇后之後再對於每個皇后，檢查米字上是否有其他皇后。實做上，通常使用遞迴。把枚舉的過程畫出來的話，他會長成一個樹狀圖 (有些人將這棵樹稱作「解答樹」)。



如果依序從最棋盤最左上角開始走起，會發現沿著樹的一個分支走到底後，才回頭嘗試另一個分支。這樣沿著一個分支搜尋到最深處後再嘗試另一條的方法叫做「深度優先搜尋 (Depth First Search)」。

這個直觀的作法效率如何？總共枚舉了  $P_n^{n^2}$  個狀態，最後每隻皇后又花  $O(n)$  的檢查，效率非常低落。我們可以朝幾個方向優化搜尋的方法。

首先，觀察一下解答樹，發現有不少時候在擺滿  $n$  隻皇后以前就發現當前的盤面已經不合法了，此時可以不用繼續遞迴下去搜尋 (e.g. 在擺第 2 隻皇后的時候發現已經會和第 1 隻皇后互吃了，就不用繼續擺第 3 到  $n$  隻皇后了)。這種遞迴的過程中發現不合法，提早離開這個分支的作法叫做「回溯法」，也叫做「剪枝法」(在樹狀圖上，看起來很像把不必要的分支剪掉)。



其次，搜尋擺下去的皇后是否和其他皇后互吃可以在  $O(1)$  的時間內完成。可以發現當兩個位置在同一條由左下到右上的斜線時其座標和相同，要考慮由左上到右下的斜線就將橫坐標翻轉即可有相同性質，因此可以用 `bitset` 或者是整數型態額外維護當前盤面上每個橫線、直線、斜線上是否有皇后。

由於經過剪枝後，所需枚舉的狀態數不太好估計，因此時間複雜度也不太容易估算。以這題為例，在套用以上優化後，在兩秒左右解出 14 皇后問題。

```

1 int col, uldr, dlur; // 將 int 用二進位的角度去維護直線、斜線上有沒有皇后
2 int queen(int r, int n)
3 {
4     if(r == 0) return 1;
5     int sum = 0;
6     for(int c = 1; c <= n; c++)
7     {
8         if((col>>c)&1 || (uldr>>(r+c))&1 || (dlur>>(r-c+n))&1) continue;
9         col |= (1<<c), uldr |= (1<<(r+c)), dlur |= (1<<(r-c+n));
10        sum += queen(r-1, n);
11        col &= ~(1<<c), uldr &= ~(1<<(r+c)), dlur &= ~(1<<(r-c+n));
12    }
13    return sum;
14}
15 int main()
16 {
17     int n; cin >> n;
18     col = uldr = dlur = 0;
19     cout << queen(n, n) << endl;
20 }
```

程式碼 3.15: n 皇后問題

### 3.4.3 枚舉排列

#### 枚舉重複排列

以下提供一種使用遞迴枚舉重複排列  $n^n$  種可能的範例。

```
1 int ans[10];
2 void rep_permutation(int now, int n, int arr[]) {
3     if(now == n) {
4         for(int i = 0; i < n; i++) {
5             cout << ans[i] << " \n"[i == n - 1];
6         }
7         return;
8     }
9     for(int i = 0; i < n; i++) {
10        ans[now] = arr[i];
11        rep_permutation(now + 1, n, arr);
12    }
13 }
14 int main(){
15     int a[] = {1, 2, 3};
16     rep_permutation(0, 3, a);
17     return 0;
18 }
```

程式碼 3.16: 重複排列

#### 枚舉不重複排列

一樣可以寫個遞迴函式枚舉所有  $n!$  種可能，也可以使用 STL 內建的 `next_permutation()` 函式。`next_permutation` 會依照字典序大小算出給定序列的下一個狀態並且回傳 `true`。如果給定序列的字典序已經最大，`next_permutation` 會回傳 `false`。因此，給定一個由字典序小到大排序過得陣列，只要搭配 `do while` 迴圈使用，就可以枚舉出  $n!$  種排列。`next_permutation` 被定義在 `<algorithm>` 函式庫當中，使用方法如下：

```
1 #include <algorithm>
2 int arr[] = {1, 2, 3}, n = 3;
3 do {
4     for(int i = 0; i < n; i++) {
5         cout << arr[i] << " \n"[i==n-1];
6     }
7 } while(next_permutation(arr, arr+n));
```

程式碼 3.17: `next_permutation` 用法

### 3.4.4 枚舉子集

一個大小為  $n$  的集合總共幾個子集？對一種子集而言，每一個元素可能「在」或「不在」2種狀態，因此總共有  $2^n$  個子集合。要怎麼表示一個子集？對於一個元素，可以用 1 表示「在」，用 0 表示「不在」，因此可以用  $n$  個 0/1 表示一個子集合。這樣用多個 bit 來表示集合的方法稱之為「位元向量 (Bit Vector)」。

當集合大小不大的時候，可以用一個整數型別表示集合 (e.g. 當集合大小  $n \leq 32$  的時候可以用 int 表示) 或者是使用 bitset。其中第 0 個 bit(Least Significant Bit) 表示第 0 個元素 (方便起見，元素的編號為 0 到  $n - 1$ )，以此類推。我們可以透過一些位元運算的方式對這個集合操作：

```
1 // 查詢第 i 個元素是否在集合 s 中
2 bool inSet(int i, int s) { return (s >> i) & 1; }
3 // 將第 i 個元素加入集合中
4 void ins(int i, int &s) { s |= (1 << i); }
5 // 將第 i 個元素移除集合
6 void rm(int i, int &s) { s &= ~(1 << i); }
7 // 取兩集合 s1, s2 交集
8 int inter(int s1, int s2) { return s1 & s2; }
9 // 取兩集合 s1, s2 聯集
10 int uni(int s1, int s2) { return s1 | s2; }
11 // 取集合 s 的補集
12 int comp(int s) { return ~s; }
```

程式碼 3.18: 常見集合操作

不難發現，如果以整數表示大小為  $n$  的集合，這個集合所構成的所有子集在二進制中剛好會是 (0...0) 到 (1...1)，對應到十進位為 0 到  $2^n - 1$ 。因此可以簡單用一個迴圈來枚舉所有的子集：

```
1 int n = 10;      // 集合大小
2 for(int s = 0; s < (1 << n); s++)
3 {
4     // 此時 s 為枚舉的子集
5 }
```

程式碼 3.19: 常見集合操作

### 3.4.5 枚舉組合

在簡單枚舉提到三角形題目，其實就是枚舉所有  $(n)_3$  的組合，我們可以很直觀的寫出三層 for 迴圈去枚舉，但若你想枚舉  $(n)_{10}$  的組合，寫 10 層 for 迴圈會讓你的程式碼很雜亂且容易出錯，因此我們可以使用遞迴來輔助我們枚舉所有組合。使用 pre 變數是避免枚舉到重複的組合但不同排列。

```

1 int ans[10];
2 void combination(int now, int pre, int n, int m, int arr[]) { //n choose m
3     if(now == m) {
4         for(int i = 0; i < m; i++) {
5             cout << ans[i] << " \n"[i == m - 1];
6         }
7         return;
8     }
9     for(int i = pre; i <= n - (m - now); i++) {
10        ans[now] = arr[i];
11        combination(now + 1, i + 1, n, m, arr);
12    }
13 }
14
15 int main(){
16     int a[] = {1, 2, 3, 4, 5};
17     combination(0, 0, 5, 3, a);
18     return 0;
19 }
```

**程式碼 3.20:** 枚舉組合

## 3.5 二分搜尋法

|                 |      |
|-----------------|------|
| 二分搜尋            | 基本問題 |
| 找遞增陣列中 $x$ 的位置。 |      |

最容易想到的作法為線性搜尋 (Linear Search)，就是用一層迴圈檢查所有元素是否為  $x$ ，時間複雜度為  $\mathcal{O}(N)$ 。但是，可以透過「遞增」這個性質做大大的優化。

選定陣列正中央的元素  $arr[mid]$ ，如果  $x < arr[mid]$ ， $x$  不可能出現在索引值  $> mid$  的位置，那繼續搜尋索引值  $< mid$  的區間；如果  $x > arr[mid]$ ， $x$  不可能出現在索引值  $< mid$  的位置，那就繼續搜尋索引值  $> mid$  的位置。照這樣的步驟，每次可以篩掉一半的搜尋空間。當搜尋區間長度只剩下 1 的時候，該元素就是想找的  $x$ 。

```

1 int bsearch(int *arr, int n, int x)
2 {
3     int l = 0, r = n-1, mid, ans = -1;
4     while(l <= r)
5     {
6         mid = (l+r)/2;
7         if(arr[mid] == x)
8         {
9             ans = mid; break;
10        }
11        if(arr[mid] < x) l = mid+1;
12        else r = mid-1;
13    }
14    return ans;
15}
16 int main()
17 {
18     int arr[] = {1, 2, 2, 4, 7, 7, 8, 9}, n = 8;
19     cout << bsearch(arr, 8, 4) << endl; // 3, arr[3] = 4;
20 }

```

程式碼 3.21: 二分搜尋範例

上述方法即為二分搜尋法 (Binary Search)。對於大小為  $N$  的具有單調性 (遞增或遞減) 的陣列，只要  $\mathcal{O}(\log N)$  的時間就可以找到  $x$  在陣列中的位置。類似的作法也可以找出  $x$  是否存在陣列中，以及  $x$  的上下界。

除了可以搜尋具有索引值和單調性的資料之外，只要是有單調性的函數都可以採用二分搜尋。此外，如果題目的可行答案具有單調的關係，則可以用二分搜夾擠出答案。

APCS1060304 基地台

zerojudge c575

一條筆直的大道上有  $N$  ( $N \leq 10^5$ ) 個服務站，並在大道上架設  $k$  個基地台，每個基地台提供的無線網路服務範圍相同，只要服務站與基地台的距離小於基地台的半徑  $R$  即可享有無線網路服務。給定每個服務站的座標，求在所有服務站都享有無線網路服務的條件下，基地台的最小直徑 (即  $2R$ ) 為多少。

觀察一下，如果某個直徑  $R$  是合法的，那對於所有  $> R$  的直徑就不會是最小解。如果  $R$  不合法，那對於所有  $< R$  的直徑都不合法。因為答案具有單調性，可以對直徑二分搜出最小直徑。事先把座標排序的話查詢  $R$  是否合法可以  $\mathcal{O}(n)$  內搞定，二分搜的時間複雜度為  $\mathcal{O}(\log \text{座標範圍})$ 。整體時間複雜度為  $\mathcal{O}(n \log n + n \log \text{座標範圍})$ 。

二分搜尋法的應用範圍很廣。他看似容易理解，卻也是許多演算法與技巧的重要基礎。在遇到瓶頸時，可以思考一下是否有那些函數或者是數值具有單調性，說不定便能從此找到破口喔！

### 3.5.1 STL 中的二分搜尋

給定一個已經排序好的陣列和一個數值  $x$ ，可以用 STL 寫好的函式找出  $x$  是否存在陣列中和  $x$  的上下界。下列函式都被定義在 `<algorithm>` 函式庫當中，更多詳細用法與說明可自行上網查詢。

`std::binary_search(first, last, val)`

如果在  $[first, last]$  的範圍中存在  $val$  則回傳 `true`，否則回傳 `false`。

`std::lower_bound(first, last, val)`

回傳  $[first, last]$  中第一個大於等於  $val$  的指標。如果找不到大於等於  $val$  的元素，則回傳  $last$ 。

`std::upper_bound(first, last, val)`

回傳  $[first, last]$  中第一個大於  $val$  的指標。如果找不到大於  $val$  的元素，則回傳  $last$ 。

```
1 #include <algorithm>
2
3 // sorted array
4 int arr[] = {1, 2, 2, 4, 7, 7, 8, 9}, n = 8;
5 vector<int> vec(arr, arr+8);
6 cout << binary_search(arr, arr+n, 8);           // 1 (true)
7 cout << binary_search(vec.begin(), vec.end(), 0); // 0 (false)
8 cout << *lower_bound(arr, arr+n, 2);             // 2
9 cout << lower_bound(arr, arr+n, 3)-arr;          // 3, arr[3]=4
10 cout << *upper_bound(vec.begin(), vec.end(), 2); // 4
11 cout << upper_bound(vec.begin(), vec.end(), 3)-vec.begin(); // 3, vec[3]=4
```

**程式碼 3.22:** STL 中二分搜相關函式使用範例

## 3.6 三分搜尋

happiness function

2013 台清交程式設計大賽

一場比賽有  $N$  道題目，解出第  $i$  道題目的快感是一個跟比賽經過時間相關的函數  $f_i(t) = a_i(t - b_i)^2 + c_i$ , ( $0 \leq a_i, b_i, c_i \leq 300$ )。比賽進行中第  $t$  時刻的樂趣度  $S(t) = \max\{f_i(t) | 1 \leq i \leq n\}$ 。一場比賽共 300 分鐘，即  $0 \leq t \leq 300$ ，求整場比賽樂趣度最低的時的樂趣度。

把  $S(t)$  畫出來後發現  $S(t)$  長的很像 U 字形，最低點的左側遞減，右側遞增。這題能用線性搜尋嗎？感覺不太行。這題能二分搜嗎？如果從中間切下去，單單從中點的函數值似乎不太能判斷最低點在那一側（其實可以，但是需要微積分）。問題在於 U 形函數的最低點兩側各自有單調性，既然把函數分兩半不夠，那就把函數分三半吧。假設將函數分三部份後，由左到右四個點分別為  $a, b, c, d$ ，分別討論四點上函數值的大小關係：

1.  $f(a) < f(b) < f(c) < f(d)$ ：最小值不可能在最右邊的  $[c,d]$  區間
2.  $f(a) > f(b) < f(c) < f(d)$ ：最小值不可能在最右邊的  $[c,d]$  區間
3.  $f(a) > f(b) > f(c) < f(d)$ ：最小值不可能在最左邊的  $[a,b]$  區間
4.  $f(a) > f(b) > f(c) > f(d)$ ：最小值不可能在最左邊的  $[a,b]$  區間

觀察一下，進一步發現只有  $f(b), f(c)$  的大小關係會影響到最小值出現在哪個區間，因此可以得到更精簡的性質：

1.  $f(b) < f(c)$ ：最小值不可能在最右邊的  $[c,d]$  區間
2.  $f(b) > f(c)$ ：最小值不可能在最左邊的  $[a,b]$  區間

上述把區間切三份每次排除一份的作法為三分搜尋法。切三等分方法有很多種，像是三等分法或是黃金比等等。對於長度為  $N$  的區間，每次都可以排除一個不可能的部份，整體的複雜度也是  $\mathcal{O}(\log N)$ 。

```

1 double f(double x);      // f(x)為所求的U型函數
2 double triSearch(double lowest_x, double highest_x)
3 {
4     double low = lowest_x, l, r, high = highest_x;
5     while(high-low > 1e-9)
6     {
7         l = low+(high-low)/3, r = l+(high-low)/3;
8         double lv = f(l), rv = f(r);
9         if(lv < rv) high = r;
10        else low = l;
11    }
12    return f(low);
13 }

```

**程式碼 3.23:** 三分搜尋夾出最低點的值

假如目前是一個離散(與連續相對，其值並非連續不斷，不可作無限分割)的區間具有最低點左側嚴格遞減，右側嚴格遞增，例如一個序列存在一個  $k$  使得  $a_1 > a_2 \dots > a_k < a_{k+1} < a_n$ ，我們可以發現所有在最低點的左側的點，其與兩側的點的大小關係是小於左側的點大於右側的點，而所有在最低點右側的點，其與兩側的點的大小關係是大於左側的點小於右側的點，只有最低點與兩側的點的大小關係是小於左側的點也小於右側的點，這個性質具有單調性，因此可以使用二分搜取代三分搜。

## 3.7 離散化

在一些題目中，數值的大小並不重要，重要的是數字間的大小關係，也就是數值在陣列中的名次。將陣列中的值轉為名次的過程稱為離散化。例如說原本的陣列為 { 7, 1, 2, 2, 9, 4, 8, 7 }，離散化後的結果為 { 4, 1, 2, 2, 6, 3, 5, 4 }。這邊提供兩個實作方法：

std::set 搭配 std::map

把陣列中元素丟進 set 後可排序並且刪除重複元素，接著由小到大取出，用 map 將元素對應到名次。將所有元素放入 set 的複雜度為  $\mathcal{O}(N \log N)$ ，查詢所有元素在 map 中對應的值的複雜度為  $\mathcal{O}(N \log N)$ ，因此整題的時間複雜度為  $\mathcal{O}(N \log N)$ 。

```

1 void dis_stl(int *arr, int n) {
2     int rk = 1;
3     set<int> s; map<int, int> mp;
4     for(int i = 0; i < n; i++) s.insert(arr[i]);
5     for(auto x:s) mp[x] = rk++;
6     for(int i = 0; i < n; i++) arr[i] = mp[arr[i]];
7 }
```

程式碼 3.24: 使用 set 和 map 離散化

### 排序後二分搜

原理和用 set/map 的方法相同。在刪除重複元素時，可以使用 STL 內建的 unique 函式。unique 會刪除排序陣列中重複的元素，並且回傳指向新陣列末端的指標。最後二分搜出每個元素在 unique 後陣列的位置。排序的複雜度為  $\mathcal{O}(N \log N)$ ，對每個元素二分搜的整體複雜度為  $\mathcal{O}(N \log N)$ ，總時間複雜度也就是  $\mathcal{O}(N \log N)$ 。

```

1 void dis_bs(int *arr, int n) {
2     vector<int> tmp(arr, arr+n);
3     sort(tmp.begin(), tmp.end());
4     tmp.resize(unique(tmp.begin(), tmp.end())-tmp.begin());
5     for(int i = 0; i < n; i++)
6         arr[i]=lower_bound(tmp.begin(),tmp.end(),arr[i])-tmp.begin()+1;
7 }
```

程式碼 3.25: 使用排序二分搜離散化

儘管兩種方法的時間複雜度相同，但 set 與 map 的執行效率沒有 sort 來得好，所以筆者建議使用排序後二分搜的方法。

## 3.8 練習題目

|   |          |
|---|----------|
| 打獵分配問題  | skyoj 82 |
| 貓族的貓咪各自的名字、職位、年紀。職位高低依序為 elder, nursy, kit, warrior, apprentice, medicent, deputy, leader。貓族吃飯的順序依照下列規則：除了 apprentice 年輕的優先食用，其餘的由年長開始。若遇到職位與年紀都一樣的貓，則依名字的字典序先後為準則。今已知有貓 N 隻，糧食 M 份，和每隻貓的資訊(名字 職位年齡(月))，請輸出可享用食物隻貓兒的先後順序。已知每隻貓只享用一個食物，每隻貓的名字是獨一無二的。 |          |

|  |                |
|--|----------------|
| apcs d. 物品堆疊 (Stacking)  | zerojudge c471 |
| $N(N \leq 10^5)$ 個物品堆成一疊，第 $i$ 個物品的重量為 $w(i)$ ，取用次數為 $f(i)$ 。每次取用物品 $i$ 所耗費的能量為 $f(i) \times$ 疊在物品 $i$ 上面所有物品的重量和。求最小耗費的能量和。 |                |

|   |                |
|---|----------------|
| Robot Vacuum Cleaner  | codeforce 922D |
| 給定 $n(n \leq 10^6)$ 個由's' 和'h' 組成的字串。定義一個字串 $t$ 的雜音為滿足 $i < j$ 且 $t_i = 's'$ , $t_j = 'h'$ 的 $(i, j)$ 對數。求 $n$ 個字串組成的大字串中最大的雜音值為多少。 |                |

|   |      |
|---|------|
| Bubble Sort 交換次數  | 經典題目 |
| 給定一個大小為 $N(N \leq 10^6)$ 的陣列。求執行 Bubble Sort 時，數字兩兩交換的次數總和。 |      |

|  |                |
|--|----------------|
| 00441 - Lotto                            | zerojudge c074 |
| 輸入一個大小為 $K$ 的序列，輸出所有 $\binom{k}{6}$ 的組合。 |                |

|  |                |
|--|----------------|
| Petr and Permutations  | codeforce 986B |
| 有一個陣列 $\{1, 2, \dots, N\}, (N \leq 10^6)$ 。Petr 喜歡對此陣列執行 $3n$ 次任取兩個數字換的操作，Um_nik 喜歡對此陣列執行 $7n + 1$ 次任取兩個數字換的操作。給定一個 Petr 或 Um_nik 操作過得陣列，請找出操作陣列的是誰。 |                |

|   |      |
|---|------|
| 迴文字字串數量   | 經典例題 |
| 子字串表示字串中的一個連續區段，例如"ink" 為"Jinkela" 的子字串。迴文字串表示將字串翻轉後仍然長的一樣，例如"lol", "101", "IOI" 都是迴文字串。給定一個長度 $\leq 2000$ 的字串，請算出他包含多少回文字字串。 |      |

pC. 最暖的冬天

tioj 1882

裸三分搜題。

電梯向上

NPSC2012 高中組初賽

有  $N(N \leq 10^5)$  個客人要搭電梯到 NPSC 百貨公司頂樓搭摩天輪，每個人有號碼牌  $idx_i$  和體重  $w_i$ ，號碼較小的優先上樓。已知電梯最多來回  $k$  趟，求電梯最大承載至少幾公斤才能在  $k$  趟內將所有遊客送上頂樓。

水精靈的舞蹈

skyoj 23

水柱高度的圖形符合  $f(x) = -(x - a_0)(x - a_1)\dots(x - a_{n-1})$ ，給定排序好的  $a_0\dots a_{n-1}$  和  $p$ ，求在  $a_p \leq x \leq a_{p+1}$  的區間中，哪個位置的水柱高度最高。

Average Length

AtCoder Beginner Contest 145 pC

給你  $N, (N \leq 8)$  個平面上的點，兩點之間的距離為其直線距離，求走過剛好所有  $N$  點的所有路徑的長度和平均為多少。

Comiket

tioj 1410

有  $n(n \leq 10^6)$  個客人，給定每個人進入與離開會場的時間 ( $\leq 2^{31}$ )，求會場內最多同時出現多少人。

天才麻將少女

tioj 2006

輸出當前麻將牌面是否聽牌，以及聽哪幾張。

第 k 大連續和

tioj 1208

連續和為數列中幾個相連元素的和。給定一個數列，求第  $k$  大連續和為多少。(註：這題除了用二分搜外，也可以使用適當的資料結構喔)

Cave(互動題)

IOI 2013

有  $N(N \leq 5000)$  個門與開關，但是你不知道哪個開關對應哪個門。每個開關有「上」、「下」兩種位置，只有一個位置可以讓他對應的門打開。因為門不透明，你只能看到第一個關著的門的位置。請在 70000 次嘗試以內找出讓所有門都開啟的開關組合。

骨牌遊戲

tioj 1432

給你一個長度為  $N, (N \leq 1000)$  的序列，你可以將其任意切成  $K$  個連續的區間，求在你切完之後可能的最大的區間和最小為多少。



# 基礎動態規劃

## 4.1 前言

動態規劃 ( Dynamic Programming, DP ) 是一種通過把原問題分解為相對簡單的子問題的方式求解複雜問題的方法。若要解一個給定問題，我們需要解其不同部分 ( 即子問題 )，再根據子問題的解以得出原問題的解。如果存在重複的子問題，那麼只需要解決其中一個子問題之後將它的答案存在陣列中，下次遇到同樣的子問題時就可以直接查表。由於子問題的數量在使用暴力解法時可能是指數成長的，因此對於這樣的題目，動態規劃可以讓時間複雜度從指數量級降到多項式量級。

先備知識：熟悉 C 或 C++ 基本語法當中的輸入輸出、基本運算、變數、一維陣列與二維陣列、if 判斷式、for 迴圈。STL 函數只要知道 `cin`、`cout`、`min` 和 `max` 即可。

動態規劃本身沒有太多理論，如何運用它解出一個沒有見過的題目才是動態規劃類型問題的主要挑戰。因此本課程會直接以講解七個具有指標性的動態規劃問題作為課程的進行方式，希望第一次聽到動態規劃的人可以對動態規劃有基本的認識，已經有概念的人也可以學到更多題型的解法。

和去年的課程相比，刪除了四題例題，但是只新增兩題例題，希望可以更詳細的講解這些題目。另外，也刪除了一些「技巧」，理由是這些技巧未經過驗證，也不一定適合每個人，以更詳細的建議學習方向取代。

## 4.2 走樓梯

有一個人要走上樓梯，樓梯共有  $N$  階，每走一步可以往上走 1 或 2 個階梯，請問走上樓梯有幾種方法？答案可能很大，所以輸出答案除以  $10^9 + 7$  的餘數。

$$1 \leq N \leq 10^5$$

$d[i]$  = 走上  $i$  個階梯的方法數

所有剛好走  $i$  階的走法 ( $i \geq 3$ )，可以分成兩類：

1. 累計往上走了恰好  $i - 1$  階之後，最後一步往上走 1 階， $d[i - 1]$  種走法
2. 累計往上走了恰好  $i - 2$  階之後，最後一步往上走 2 階， $d[i - 2]$  種走法

因此  $d[i] = d[i - 1] + d[i - 2]$ 。所有方法都有被算到，也沒有方法被算兩次。這是一個具有遞迴關係的算式，但是以  $i$  遞增的順序來計算  $d[i]$  即可，不需要遞迴。

動態規劃是一種有效率的解決問題的方法，適用有**重疊子問題和最佳子結構性質**的問題。

我們不用窮舉每一種走法的原因是，對於某個  $i$ ，我們會遇到很多次「樓梯有  $i$  階時有幾種走法？」這個問題，因為同樣的問題不管解多少次答案都一樣，所以解出一次之後把答案存起來，不需重新計算，這是**重疊子問題性質**。

我們把  $N = i$  的原問題分解成  $N = i - 1$  和  $N = i - 2$  這兩個子問題，而兩個（也可以是多個）子問題分別解決之後將可以解決原問題（在這裡是把兩個子問題的答案加起來），這是**最佳子結構性質**。

需要「輸出答案除以 mod 的餘數」的時候，每做一次加法就模一次 mod，最後得到的數就等於整個做完之後再模 mod。（要小心溢位）

```
1 #include <iostream>
2 using namespace std;
3 int n, d[100005], mod = 1000000007;
4 int main() {
5     d[1] = 1;
6     d[2] = 2;
7     cin >> n;
8     for (int i = 3; i <= n; i++) d[i] = (d[i - 1] + d[i - 2]) % mod;
9     cout << d[n] << '\n';
10 }
```

## 4.3 選數字

有一個陣列  $a[1] \cdots a[N]$ 。現在可以從中選擇一些數，但是任兩個有選的數之間至少要隔著一個沒選的數。請問所有選法裡面，所選的數的總和最大可以是多少？

$$1 \leq N \leq 10^5, 1 \leq a[i] \leq 10^4$$

$d[i]$  = 考慮  $a[1] \cdots a[i]$  的最大總和

所有考慮前  $i$  個數的選法 ( $i \geq 2$ )，可以分成兩類：

1.  $a[i]$  要選，此時  $a[i - 1]$  不能選，但  $a[i]$  可以搭配任何從  $a[1] \cdots a[i - 2]$  選出一些數所形成的方案，此時最大總和是  $d[i - 2] + a[i]$ 。
2.  $a[i]$  不要選，此時最大總和是  $d[i - 1]$ 。

每一個合法的方案一定會剛好屬於其中一種，所以兩者取較大值就是答案。

解決 DP 問題主要有兩個步驟：定義狀態，然後找出狀態之間的轉移式。

一個狀態代表用某些參數來描述的一個子問題，儲存的資訊就是子問題的答案。子問題必須精確描述，每次遇到時答案都要一樣。一個狀態必須計算完成才是有意義可以被使用的。通常這些狀態會存在一個陣列中，狀態定義需要幾個參數就使用幾維的陣列。這個陣列稱為 DP 陣列，課程中以  $d$  命名。

在計算一個狀態時，需要用到哪些其他狀態，又要怎麼組合它們來算出這個狀態的值，這就是狀態之間的轉移。有些狀態不會從其他狀態轉移過來，而是一開始就指定它的值，稱呼它們為邊界狀態。以這題為例，邊界狀態為  $d[0]$  和  $d[1]$ 。

```
1 #include <iostream>
2 using namespace std;
3 int n, a[100005], d[100005];
4 int main() {
5     cin >> n;
6     for (int i = 1; i <= n; i++) cin >> a[i];
7     d[1] = a[1];
8     for (int i = 2; i <= n; i++)
9         d[i] = max(d[i - 2] + a[i], d[i - 1]);
10    cout << d[n] << '\n';
11}
```

## 4.4 Vacation

Atcoder DP contest - C

Taro 有  $N$  天的假期。他每天可以從游泳、抓蟲和寫作業三項活動中選一種。在第  $i$  天游泳、抓蟲和寫作業分別會得到  $a_i$ 、 $b_i$  和  $c_i$  單位的快樂值。但是 Taro 不能連續兩天進行同樣的活動，請問 Taro 能得到的總快樂值最大可以是多少？

$1 \leq N \leq 10^5, 1 \leq a[i], b[i], c[i] \leq 10^4$

$d[i][0]$  = 第  $i$  天游泳，前  $i$  天的最大總快樂值

$d[i][1]$  = 第  $i$  天抓蟲，前  $i$  天的最大總快樂值

$d[i][2]$  = 第  $i$  天寫作業，前  $i$  天的最大總快樂值

計算  $d[i][0]$ ：如果第  $i$  ( $i > 1$ ) 天要游泳，那麼第  $i - 1$  天只能抓蟲或寫作業。所以只要知道「第  $i - 1$  天抓蟲，前  $i - 1$  天的最大總快樂值」和「第  $i - 1$  天寫作業，前  $i - 1$  天的最大總快樂值」，兩者取較大值，加上當天的快樂值 ( $a_i$ )，就是「第  $i$  天游泳，前  $i$  天的最大總快樂值」。同理也可以計算  $d[i][1]$  和  $d[i][2]$ 。

因為計算  $d[i][0], d[i][1], d[i][2]$  這三個狀態的時候只會用到  $d[i - 1][0], d[i - 1][1], d[i - 1][2]$  這三個狀態，所以以  $i$  遞增的順序計算，就可以確保被用到的狀態都是計算完成的。

值得注意的是，程式碼中沒有對 DP 陣列有任何初始化（也就是一開始全部的值都是 0），這是因為這題的狀態計算是從其他兩個狀態的較大值得出的，而這些值又是非負的，所以初始值是 0 就沒有問題。所有狀態（包括邊界狀態）的初始值是實作時要注意的地方。例如如果這題要求的是最小總快樂值，那麼所有邊界狀態以外的狀態都應該初始成一個很大的數（比答案的最大可能值還大）。

```
1 #include <iostream>
2 using namespace std;
3 int n, a[100005], b[100005], c[100005], d[100005][3];
4 int main() {
5     cin >> n;
6     for (int i = 1; i <= n; i++) cin >> a[i] >> b[i] >> c[i];
7     for (int i = 1; i <= n; i++) {
8         d[i][0] = max(d[i - 1][1], d[i - 1][2]) + a[i];
9         d[i][1] = max(d[i - 1][0], d[i - 1][2]) + b[i];
10        d[i][2] = max(d[i - 1][0], d[i - 1][1]) + c[i];
11    }
12    cout << max({d[n][0], d[n][1], d[n][2]}) << '\n';
13 }
```

## 4.5 LIS ( 最長遞增子序列 )

有一個陣列  $a[1] \cdots a[N]$ ，選擇一個它的子序列，子序列除了第一項之外每一項都必須大於前一項，這個子序列長度最大可以是多少？子序列可以簡單想成選擇原陣列中的一些東西而不改變它們的相對位置形成的序列。

$1 \leq N \leq 2000, 1 \leq a[i] \leq 10^9$

$d[i]$  = 以  $a[i]$  結尾的最長遞增子序列長度 ( 一定要選  $a[i]$  )

一個以  $a[i]$  結尾的子序列如果長度大於 1，那麼它是遞增的若且唯若扣掉  $a[i]$  之後的子序列仍然是遞增的，且最後一項小於  $a[i]$ 。因為這個「最後一項」一定會是  $a[1], a[2], \dots, a[i - 1]$  的其中一個，而以其中任何一個做為結尾的最長遞增子序列長度都已經計算完成，因此  $d[i] = \max(d[j] + 1 | j < i, a[j] < a[i])$ ，相當於從很多子問題的答案中挑出最好的來更新原問題的答案。

如果  $a[i]$  選或不選的最長遞增子序列長度都記在  $d[i]$  中，之後用到  $d[i]$  時會無法確定這個遞增子序列後面到底能不能接上一個特定的數，所以狀態的定義要是「一定要選  $a[i]$ 」。

和前面的例題不同的是，之前在計算一個狀態時，都是從兩個其他的狀態轉移到這個狀態，但是這裡在計算  $d[i]$  時，需要考慮  $i - 1$  個其他的狀態，也就是最多會從  $i - 1$  個狀態轉移過來，所以需要另一個迴圈來枚舉這些狀態。不過「原問題分解成子問題、紀錄子問題答案避免重複計算、利用已經解決的小問題組合出大問題的解」，這些解決動態規劃問題的核心概念還是不變的。

```
1 #include <iostream>
2 using namespace std;
3 int n, a[2020], d[2020], z = 0;
4 int main() {
5     cin >> n;
6     for (int i = 1; i <= n; i++) cin >> a[i];
7     for (int i = 1; i <= n; i++) d[i] = 1;
8     for (int i = 1; i <= n; i++)
9         for (int j = 1; j < i; j++)
10            if (a[j] < a[i]) d[i] = max(d[i], d[j] + 1);
11     for (int i = 1; i <= n; i++) z = max(z, d[i]);
12     cout << z << '\n';
13 }
```

## 4.6 背包問題

有一個容量  $M$  的背包和  $N$  個物品，第  $i$  個物品的體積是  $a[i]$ ，價值是  $b[i]$ ，選擇某些物品放入背包內，總體積不能超過背包容量，請問所選物品的總價值最大可以是多少？

$1 \leq N \leq 100, 1 \leq M \leq 10^5, 1 \leq a[i] \leq 10^5, 1 \leq b[i] \leq 10^7$

$d[i][j]$  = 只考慮前  $i$  個東西，且總體積不超過  $j$  的最大總價值

計算  $d[i][j]$  時，分兩種情況討論：

1. 不選第  $i$  個物品，此時最佳解等於忽略這個物品，但體積上限不變的子問題最佳解，也就是  $d[i - 1][j]$ 。
2. 要選第  $i$  個物品，此時最佳解等於忽略這個物品，解決體積上限  $j - a[i]$  的子問題之後再加入這個物品，也就是  $d[i - 1][j - a[i]] + b[i]$ 。如果  $j < a[i]$ ，這種情況就不存在，不需考慮（也就是說，第  $i$  個物品一定不選）。

```
1 #include <iostream>
2 using namespace std;
3 int n, m, a[102], b[102], d[102][100005];
4 int main() {
5     cin >> n >> m;
6     for (int i = 1; i <= n; i++) cin >> a[i] >> b[i];
7     for (int i = 1; i <= n; i++)
8         for (int j = 0; j <= m; j++)
9             if (j >= a[i])
10                 d[i][j] = max(d[i - 1][j], d[i - 1][j - a[i]] + b[i]);
11             else d[i][j] = d[i - 1][j];
12     cout << d[n][m] << '\n';
13 }
```

因為計算第一維是  $i$  的狀態時只會用到第一維是  $i - 1$  的狀態，更之前的狀態其實可以捨棄，所以可以把第一維的  $i$  改成  $i \% 2$ ，第一維的  $i - 1$  改成  $!(i \% 2)$ （也要把最後一行的  $d[n][m]$  改成  $d[n \% 2][m]$ ），DP 陣列就可以減少一個維度。這個技巧稱為**滾動陣列**，使用在記憶體不足的時候。

## 4.7 旅行推銷員問題

有  $N$  棟房子，編號分別是  $0 \dots N - 1$ 。房子  $i$  和房子  $j$  之間的距離是  $a[i][j]$ 。有一個推銷員要從房子 0 出發，經過每一棟房子恰好一次之後再回到房子 0，請問他最少要走多遠？

$$2 \leq N \leq 17, 0 \leq a[i][j] \leq 10^7, a[i][j] = a[j][i], a[i][i] = 0$$

$d[i][j] =$  目前經過的房子集合用  $i$  表示，且目前在房子  $j$  的最短距離

將經過的房子集合轉化成一個數的方法是：如果經過了房子  $x$ ，就把那個數加上  $2^x$ 。如果  $i \& 2^x \neq 0$ ，那麼  $i$  代表的集合就有包含房子  $x$ 。

現在計算  $d[i][j]$ ：如果現在經過的房子集合用  $i$  表示，而且現在在房子  $j (j \in i)$ ，那麼可以枚舉上一個所在的房子  $k (k \in i, k \neq j)$ （當時的狀態是  $d[i - 2^j][k]$ ），並嘗試用  $d[i - 2^j][k] + a[k][j]$  去更新  $d[i][j]$ 。

因為上述的  $d[i][j]$  會從很多個狀態更新，所以最後這個狀態的值應該要是這些更新的最小值。除了邊界狀態  $d[1][0] = 0$ （只經過房子 0，且目前在房子 0）以外，所有的狀態預設成無限大，才能正確的和其他數取最小值。

經過了所有房子之後停在房子  $i$ （相對應的狀態是  $d[2^N - 1][i]$ ），最後再走回房子 0 就是一種走法。枚舉這個  $i$  就可以算出最終答案。

如果狀態定義的其中一個參數是「某個子集合」，這種 DP 就稱為位元 DP。時間複雜度通常只比暴力法好一點（在這題是從  $O(N!)$  進步成  $O(2^N \times N^2)$ ）。

```
1 #include <iostream>
2 using namespace std;
3 int n, a[17][17], d[131072][17], z = 1e9;
4 int main() {
5     cin >> n;
6     for (int i = 0; i < n; i++)
7         for (int j = 0; j < n; j++) cin >> a[i][j];
8     for (int i = 0; i < (1 << n); i++)
9         for (int j = 0; j < n; j++) d[i][j] = 1e9;
10    d[1][0] = 0;
11    for (int i = 0; i < (1 << n); i++)
12        for (int j = 0; j < n; j++) if (i & (1 << j))
13            for (int k = 0; k < n; k++) if ((i & (1 << k)) && j != k)
14                d[i][j] = min(d[i][j], d[i - (1 << j)][k] + a[k][j]);
15    for (int i = 1; i < n; i++) z = min(z, d[(1 << n) - 1][i] + a[i][0]);
16    cout << z << '\n';
17 }
```

## 4.8 Slimes

Atcoder DP contest - N

有  $N$  隻史萊姆排成一排，第  $i$  隻史萊姆的大小是  $a[i]$ 。每次挑選兩隻相鄰的史萊姆，把他們結合成一隻更大的史萊姆，其大小為這兩隻史萊姆的大小相加，而花費也是這兩隻史萊姆的大小相加。其他史萊姆的相對順序不變。經過  $N - 1$  次操作之後，所有的史萊姆都會被合在一起，此時最小的總花費是多少？

$1 \leq N \leq 400, 1 \leq a[i] \leq 10^9$

$d[l][r] =$  從第  $l$  隻到第  $r$  隻史萊姆都合在一起的最小總花費

首先觀察到任何時候，所有被合在一起的史萊姆在原本陣列中一定是連續的一段。所以「對於某一段連續子陣列，把當中的史萊姆全部融合成一隻大史萊姆的最小總花費是多少？」就是一個合理的子問題。

對於一個用  $d[l][r]$  代表的問題 ( $l = r$  是邊界狀態，花費是 0，所以假設  $l < r$ )，因為任何一種合成的方案，最後一定會在某個時候剩下兩隻史萊姆，所以枚舉這兩隻史萊姆的分界點在哪裡（實作上是枚舉左邊那隻的右界），就可以保證考慮到了所有的方案。如果這兩隻史萊姆在原本陣列中分別是  $[l, k]$  和  $[k + 1, r]$ ，那麼把它們結合成  $[l, r]$  這隻史萊姆的花費就是  $d[l][k] + d[k + 1][r] + a[l] + a[l + 1] + \dots + a[r]$ 。枚舉所有的  $k$  ( $l \leq k < r$ )，找到使花費最小的  $k$ 。以區間大小遞增的順序計算狀態，就可以確保用到的狀態都已經計算完成（先枚舉區間大小，再枚舉左界）。

程式中用到一個  $b$  陣列： $b[i] = b[i - 1] + a[i]$ ，也就是說  $b[i] = a[1] + a[2] + \dots + a[i]$ ，稱呼它為  $a$  的前綴和。有了這個陣列之後，就可以直接得到  $a$  陣列一個區間的總和： $a[l] + \dots + a[r] = b[r] - b[l - 1]$ 。這個概念有時候可以用來增進效率。

```
1 #include <iostream>
2 using namespace std;
3 int n;
4 long long a[404], b[404], d[404][404];
5 int main() {
6     cin >> n;
7     for (int i = 1; i <= n; i++) cin >> a[i];
8     for (int i = 1; i <= n; i++) b[i] = b[i - 1] + a[i];
9     for (int i = 1; i < n; i++) for (int j = 1; j + i <= n; j++) {
10         int l = j, r = j + i;
11         d[l][r] = 1e18;
12         for (int k = l; k < r; k++)
13             d[l][r] = min(d[l][r],
14                             d[l][k] + d[k + 1][r] + b[r] - b[l - 1]);
15     }
16     cout << d[1][n] << '\n';
17 }
```

## 4.9 延伸學習

課程中的部分例題來自 AtCoder DP Contest: <https://atcoder.jp/contests/dp>。這個練習賽有 26 題和動態規劃有關的題目。其中 C, D, N 是上課例題；A, B, E, H, M, O, T, U 是屬於這堂課的範圍的題目（但是有些題目難度很高）。另外 F 是經典的最長共同子序列（LCS）問題，主要是練習如何輸出一組解。

但是 DP 可以有非常多的變化：G 是有向無環圖 DP；P, V 是樹 DP；S 是數碼 DP；I, J 是搭配機率的 DP；K, L 是搭配賽局的 DP；Q, W 是搭配線段樹或 BIT 的 DP；R 是搭配矩陣快速幕的 DP；X 是搭配排序的 DP；Y 是搭配排列組合的 DP；Z 是 DP 優化中的凸包優化。可以在以下網址看到我對所有題目的解答：<https://atcoder.jp/contests/dp/submissions/15061437>（將 main 函數裡的大寫字母換成其他題號就可以傳到其他題）。

課程中僅提到了幾題經典的動態規劃問題，但是還有很多經典問題是值得去學習的，整理在下兩頁。相當建議以學會這些問題的解法作為接下來的目標。

如果想要練習實際比賽中的動態規劃問題，Codeforces 是很好的網站，選擇 DP 標籤即可。這裡列出一些推薦的動態規劃問題（照目前解出人數排序）：

455A 698A 545C 474D 166E 706C 118D 607A 161D 553A 339C 4D

225C 543A 711C 448C 607B 741B 721C 687C 510D 461B 453B 319C

順便推薦一些非動態規劃的問題（照目前解出人數排序）：

126B 602B 427C 380C 86D 489D 264A 723D 573B 547B 455C 321C

191C 319B 487B 354C 508D 235B 484D 372C 487C 425D 510E 432E

課程在這裡告一段落，希望大家都有學到一點東西！

動態規劃說實話就是定義狀態和找出狀態之間的轉移式，並且利用重疊子問題和最佳子結構兩個性質有效率的解決問題而已，所有的技巧、優化、和其他領域的融合等，都是建立在這個基礎上。

我覺得學習動態規劃問題的最佳方法真的就是不斷的練習問題，在思考與解決問題的過程中學習。很多的比賽都會有至少一題要用到動態規劃，所以把動態規劃學好對程式競賽的效益是非常高的！

大家加油 !!!

## 4.10 其他經典問題

### 最大連續和

有一個陣列  $a[1] \cdots a[N]$ ，選擇一段連續子陣列，它的總和最大可以是多少？

$$1 \leq N \leq 10^5, 1 \leq a[i] \leq 10^4$$

### 數字金字塔

有一個高度  $N$  的數字金字塔（第  $i$  行有  $i$  個數字，排成一個正三角形），從最上面的數字開始，每次往左下或右下走一步，經過的數字總和最大可以是多少？

$$1 \leq N \leq 2000, 1 \leq \text{所有數字} \leq 10^5$$

### 硬幣問題

有  $N$  種面額的硬幣，每一種都有無限多個，請問最少需要幾個硬幣才能湊出恰好  $M$  元？

$$1 \leq N \leq 100, 1 \leq M \leq 10^5, 1 \leq a[i] \leq 10^5$$

### LCS(最長共同子序列)

有兩個陣列  $a[1] \cdots a[N]$  和  $b[1] \cdots b[M]$ ，從兩個陣列中各選一個子序列，它們要長得完全一樣，子序列長度最大可以是多少？

$$1 \leq N, M \leq 2000, 1 \leq a[i], b[i] \leq 10^9$$

### 編輯距離

有兩個陣列  $a[1] \cdots a[N]$  和  $b[1] \cdots b[M]$ ，一次操作可以在第一個陣列的任意位置插入一個數、刪除任意一個數、或是修改任意一個位置的值，請問把第一個陣列變成第二個陣列最少要幾次操作？

$$1 \leq N, M \leq 2000, 1 \leq a[i], b[i] \leq 10^9$$

### 最長回文子序列

有一個陣列  $a[1] \cdots a[N]$ ，選擇一個它的子序列，子序列反轉之後必須和原來長得一樣，這個子序列長度最大可以是多少？

$$1 \leq N \leq 2000, 1 \leq a[i] \leq 10^9$$

### 組合數

請問從  $N$  個物品裡面選出  $M$  個物品有多少種方法？答案可能很大，所以輸出答案除以  $10^9 + 7$  的餘數。

$$1 \leq M \leq N \leq 2000$$

### LIS(最長遞增子序列)

有一個陣列  $a[1] \dots a[N]$ ，選擇一個它的子序列，子序列除了第一項之外每一項都必須大於前一項，這個子序列長度最大可以是多少？

$$1 \leq N \leq 10^5, 1 \leq a[i] \leq 10^9$$

### 無限背包問題

有一個容量  $M$  的背包和  $N$  種物品，每種物品都有無限個，第  $i$  種物品的體積是  $a[i]$ ，價值是  $b[i]$ ，選擇某些物品放入背包內，總體積不能超過背包容量，請問所選物品的總價值最大可以是多少？

$$1 \leq N \leq 100, 1 \leq M \leq 10^5, 1 \leq a[i] \leq 10^5, 1 \leq b[i] \leq 10^7$$

### 有限背包問題

有一個容量  $M$  的背包和  $N$  種物品，第  $i$  種物品的體積是  $a[i]$ ，價值是  $b[i]$ ，限量  $c[i]$  個，選擇某些物品放入背包內，總體積不能超過背包容量，請問所選物品的總價值最大可以是多少？

$$1 \leq N \leq 100, 1 \leq M \leq 10^5, 1 \leq a[i], c[i] \leq 10^5, 1 \leq b[i] \leq 10^7$$

### 矩陣鏈乘最小次數

有  $N$  個矩陣，第  $i$  個矩陣的大小是  $h[i] \times w[i]$ ，保證  $w[i] = h[i+1]$ 。把大小  $a \times b$  和  $b \times c$  的矩陣乘起來會變成一個  $a \times c$  的矩陣，需要  $a \times b \times c$  次運算，請問每次將兩個相鄰的矩陣相乘，直到把這  $N$  個矩陣全部乘起來最少要多少次運算？

$$1 \leq N \leq 400, 1 \leq h[i], w[i] \leq 10^9$$

### 最大正方形

有一個  $N \times M$  的二維陣列，所有元素的值都是 0 或 1，請問這個二維陣列中最大的完全不包含 1 的子正方形有多大？

$$1 \leq N, M \leq 2000$$

### 最大長方形

有一個  $N \times M$  的二維陣列，所有元素的值都是 0 或 1，請問這個二維陣列中最大的完全不包含 1 的子長方形有多大？

$$1 \leq N, M \leq 2000$$

### 走樓梯

有一個人要走上樓梯，樓梯共有  $N$  階，每走一步可以往上走 1 或 2 個階梯，請問走上樓梯有幾種方法？答案可能很大，所以輸出答案除以  $10^9 + 7$  的餘數。

$$1 \leq N \leq 10^{18}$$



# 進階動態規劃

## 5.1 章節介紹 Introduction

動態規劃 (Dynamic Programming, 簡稱為 DP) 是演算法設計中的一個重要方法，雖然它的原理並不難，但是狀態的設計千變萬化，又能結合其他演算法做加速，可以解決各式各樣的問題，也因此前人對這門學問也有不少的研究。

本章節為進階 DP，假定讀者對基礎 DP、圖論與資料結構已經有一定程度的基礎，介紹一些跟 DP 相關、屬於中等偏易～中等偏難的技巧，希望讀者在讀完此篇章節後能對 DP 更加精熟。本章節會討論的主題如下：

- DP 的理論觀點：這一節算是對基礎 DP 的複習。
- 常見 DP 優化：轉移常是 DP 解複雜度的瓶頸，本節介紹加速轉移的常見方法。
- 進階 DP 技巧：介紹一個較困難的經典 DP 優化供讀者欣賞。

## 5.2 DP 的理論面

這一節希望讓沒見過 DP 證明的讀者接觸證明的邏輯，若已聽過證明方法的讀者就請當作複習吧！

### 5.2.1 最佳子結構

適合使用 DP 解決的問題通常會滿足兩大性質：「最佳子結構」(optimal substructure) 和「重複子問題」(overlapping sub-problems)，其中重複子問題與 DP 的複雜度比較有關，因此筆者將重心放在與正確性有直接關係的最佳子結構上。

最佳子結構，指的是一個最佳化問題<sup>i</sup>的最佳解可以由子問題的最佳解造出，這裡直接看一個簡單的例題，展示證明最佳子結構常見的「交換」手法。

| 最長共同子序列 (LCS)                             | 經典問題 |
|---|------|
| 給定兩個字串 $A$ 、 $B$ ，試求 $A$ 與 $B$ 最長共同子序列長度。 |      |

<sup>i</sup>最佳化問題 (optimization problem)，指的就是要找出最大值或最小值的這類問題

令  $n, m$  分別表示  $A, B$  的長度，令  $dp(i, j)$  為  $A$  的前  $i$  個字與  $B$  的前  $j$  個字的最長共同子序列，則  $dp(n, m)$  即為所求。可推出轉移式：

$$dp(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{dp(i - 1, j), dp(i, j - 1), dp(i - 1, j - 1) + 1\} & \text{else if } A[i] = B[j] \\ \max\{dp(i - 1, j), dp(i, j - 1)\} & \text{otherwise.} \end{cases}$$

考慮第二條式子，若  $A[i] = B[j]$ ，則  $dp(i, j)$  所關注的最長子序列可以分出三種可能：

- $A[i]$  沒被使用，沒有出現在最長共同子序列中。
- $B[j]$  沒被使用，沒有出現在最長共同子序列中。
- $A[i]$  與  $B[j]$  皆有被使用，皆出現在最長共同子序列中。

顯然所有的共同子序列都必須符合至少一種上述情形，若是將三種情況下的最佳解都枚舉到的話必定可以算出  $dp(i, j)$ 。

把重點放在第三個情形，若要證明這種情況下，轉移式  $dp(i - 1, j - 1) + 1$  是正確的，通常會使用「交換」的手法。也就是如果有一組  $dp(i, j)$  的最佳解，滿足  $(A[i], B[j])$  互相配對，剩下的部分卻不是取  $(A[1 \dots i - 1], B[1 \dots j - 1])$  的最佳解，此時可以將剩下的部分換成  $(A[1 \dots i - 1], B[1 \dots j - 1])$  的最佳解，即  $dp(i - 1, j - 1)$ ，且答案不會變差，如下圖。

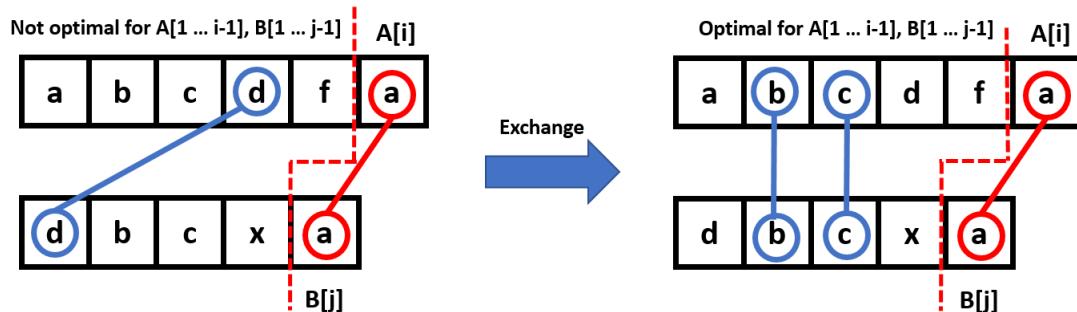


圖 5.1：在  $A[i]$  確定與  $B[j]$  配對的情形下，若  $(A[1 \dots i], B[1 \dots j])$  的最佳解不是來自  $(A[1 \dots i - 1], B[1 \dots j - 1])$ ，則可以透過交換改成  $(A[1 \dots i - 1], B[1 \dots j - 1])$  的最佳解，新的解不會變差。

這種「**若是最佳解和我的解法策略不同，我總是可以找到一個方法把它的策略換成跟我的一樣，並且保證答案不會變差**」的證明手法在證明 DP 的最佳子結構，或 greedy 的正確性時是非常常見的。

### 5.2.2 圖論觀點

在計算 DP 時必須要找到一個正確的填表順序，此節將「找填表順序」這件事做個歸納，並簡介 DP 與有向無環圖<sup>ii</sup>的關聯。

同樣以最長共同子序列當作例子，如果將每個  $dp(i, j)$  當作圖上一個點，並將  $dp(i, j)$  連出一條邊指向轉移式中和  $dp(i, j)$  有關聯的子問題，會得到一張如下的圖：

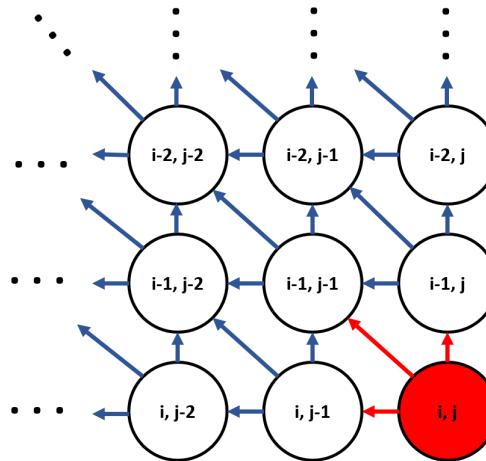


圖 5.2: LCS 的轉移，點表示狀態，邊表示此狀態參照的子問題

可以如此證明這張圖上沒有環：由某個點  $(i, j)$  出發，沿著一條邊走，可能到達的點為  $(i - 1, j)$ ， $(i, j - 1)$ ， $(i - 1, j - 1)$ ，無論何種走法，走之前與走之後，所在位置的兩座標值加總必定嚴格遞減。

假設圖上存在一環，那考慮沿著這個環走一圈形成的路徑，這條路徑由環上某一點，假設為  $(x, y)$  出發，沿路經過一條以上的邊，最後回到自己，這條路徑起終點同為  $(x, y)$ ，座標值加總相等，這和「每走一步座標值加總至少減 1」產生了矛盾，因此圖上不可能存在環。

由於這張圖並沒有環，所以才有辦法找出一個「好的順序」來計算 DP。<sup>iii</sup>計算 DP 表格的順序必須滿足當  $(i, j)$  要被計算時，所有它指向的子問題都已計算完成，也就是這張圖的任何一組反向拓樸順序 (topological order)，可以發現平常 DP 經常使用的 row major、column major、主對角線順序等等都是其中一種反向拓樸排序。

<sup>ii</sup>directed acyclic graph, 簡稱 DAG，指邊有方向性，且圖上不存在任何環的圖

<sup>iii</sup>讀者可以驗證看看這段話是不是對的，也就是說，存在「當某一點要被計算時，所有它指向的所有子問題都已計算完成」這種填表順序若且唯若關係圖上沒有環。

在此段的最後，讀者不妨思考，若一個遞迴關係的關係圖上有環，那會怎麼樣呢？這好比有兩個狀態 A 與 B，A 在算之前要求 B 先把答案算好，B 也要求 A 先於自己把答案算好，可以想見，這兩個狀態將會互相等待彼此的答案，不可能找到一個順序將 A, B 任一者答案算出。這並不代表定義出的狀態無解，但是就需要其他的方法或和問題具有其他性質，讀者可參考本節例題。

## 練習題

|  |                   |
|--|-------------------|
| Removal Game                                     | Live Archive 7980 |
| <i>Hint</i> ：轉移時，DP 的參數有何項性質在遞減？填表時未必需要將陣列展開成兩倍。 |                   |

|  |           |
|--|-----------|
| Shortest Common Non-Subsequence  | AIZU 1392 |
| <i>Hint</i> ：由圖的觀點來看，定義出的 DP 是在求關係圖上的哪個圖論經典問題？此時 DP 解和圖論經典演算法解的複雜度相同嗎？ |           |

|                     |           |
|---------------------|-----------|
| Special Judge       | TIOJ 2070 |
| 題目敘述較複雜，請在原 OJ 上解題。 |           |

|  |         |
|--|---------|
| Addition on Segments   | CF 981E |
| <i>Hint</i> ：將 DP 狀態訂為「可不可以」這種 0 與 1 的函數有時難以找到遞迴關係，試著改成「組合出 $x$ 的方法數有幾種」這種「更困難」的狀態。(先不考慮方法數可能過大的問題。此外，這題也有資料結構解) |         |

|                              |          |
|------------------------------|----------|
| Three Religions              | CF 1149B |
| 題目敘述較複雜，請在 Codeforces 上看原題目。 |          |

|  |      |
|--|------|
| Dijkstra's Algorithm   | 經典問題 |
| Dijkstra's Algorithm 是個非常有名的演算法，這個演算法跟 DP 非常的相似，但是用到的遞迴關係是有環的。請讀者試著思考為何 Dijkstra's Algorithm 在關係圖上有環的情況下仍能求出答案，並證明它的正確性。 |      |

|                                   |                  |
|-----------------------------------|------------------|
| Strings of Eternity               | AtCoder ABC 125F |
| <i>Hint</i> ：這個題目會需要字串經典演算法相關的知識。 |                  |

*Hint* : 圖上有環，試著思考一些跟 DAG 相關的經典演算法 / 經典圖論觀點。

*Hint* : 試著思考「構造一份 undecodable code 的過程」該如何用狀態完整描述，列出轉移後，遞迴關係圖上有環，請試著思考相關的圖論解法。

## 5.3 常見 DP 優化

使用 DP 解決問題常有複雜度太高的問題，此時有兩個方向，「減少狀態數」和「優化轉移」。但是狀態必須包含足夠的資訊，通常需要問題本身有好的性質才能減少，相較之下，優化轉移則有許多標準方法。本章節介紹幾種常見的優化方法，但是希望讀者能記住，最棒的解法往往是利用問題的一切性質，而不是套上一些通用的技巧，優化的方法可以有各種變化，有時只需要對問題的一點觀察就能得到很好的結果。

### 5.3.1 使用線段樹

填表順序中相鄰狀態的轉移通常很相似，但可能有少部分的不同，使得難以修改的前綴和無法使用，此時可以改成使用線段樹來做動態的修改。

有  $N$  朵花排成一列，第  $i$  朵花的美麗度是  $a_i$ ，高度是  $h_i$ ，現在你想選出一些花，使得被選中的花由左至右高度嚴格遞增，請找出此條件下，選出的花美麗度總和最大的選法。

$N \leq 10^5$ ,  $1 \leq a_i \leq 10^9$ ,  $1 \leq a_i \leq N$ ，所有  $h[i]$  皆相異。

令  $dp[i] =$  以  $h_i$  為結尾的最大美麗度 (以下稱美麗度為權重) 遞增子序列，則有  $dp[i] = \max\{dp[j] : h[j] < h[i], j < i\} + a[i]$ 。

觀察這個式子， $j < i$  這個條件讓它像是對  $j \in [1, i - 1]$  這個範圍的  $dp[j]$  求最大值，但是  $h[j] < h[i]$  却讓人不知從何下手。

轉換一下思考方式，同樣以  $i$  遞增的順序計算 DP，並且根據高度值決定  $dp[i]$  在線段樹中的位置，也就是將  $dp[i]$  放在線段樹中  $h[i]$  的位置，會發現當計算到  $i$  時，所有  $\geq i$  的  $dp[j]$  還沒放進樹裡，只要將樹中沒有人被佔據的位置設為 0，所求即為樹中  $[1, h[i] - 1]$  的範圍中的最大值，簡記為  $RMQ(1, h[i] - 1)$ 。如此即可在  $\mathcal{O}(N \log N)$  的時間內解決問題。

回顧一下這個演算法， $dp[i]$  枚舉的  $dp[j]$  有兩個範圍限制，為了正好枚舉到想枚舉的範圍，這個做法就像是用計算順序解決其中一個範圍限制，並用線段樹的詢問範圍來解決第二個限制。由這個例子應該可以感覺到計算順序是非常重要的，好的計算順序使轉移的性質更能夠被利用，希望讀者在練習 DP 時盡量選擇 bottom up 的方式，熟悉計算順序的思考。

最後希望讀者能夠想想幾個問題：

1. 將所有轉移  $dp[j]$  畫在二維平面上  $(j, h[j])$ ，DP 的求解過程可視為為二維的 RMQ 問題。請將上述  $\mathcal{O}(N \log N)$  的做法與平面常用的掃描線技巧做連結。
2. 當詢問範圍內的所有位置  $1, 2, 3, \dots, h[i] - 1$  在線段樹中都已存在轉移，此作法有枚舉「 $a[i]$  前面不放任何數字」這個可能嗎？如果沒有的話，這個方法還是對的嗎？
3. 如果兩朵花可以有相同高度、 $h[i]$  的範圍高達  $10^9$  且  $a[i]$  中可以有負數的，演算法需要做哪些改變？
4. 這個作法用計算順序解決  $j < i$  這個條件，用 RMQ 的範圍解決  $h[j] < h[i]$  這個條件。可以讓這兩個技巧解決的條件反過來嗎？應該以甚麼樣的順序計算才可使算到  $dp[i]$  時，已進到樹中的轉移  $dp[j]$  皆滿足  $h[j] < h[i]$ ？
5. 這個演算法每次詢問最大值的範圍總是  $dp$  這個陣列的前綴，且總是將樹中最大值改大，可以改用 Binary Indexed Tree (BIT) 來維護。請讀者思考在甚麼情況下可以用 BIT 來維護最大值。

此處提供使用 Binary Indexed Tree 的程式碼實作，請注意由於所有  $h[i]$  皆相異，可以將查詢範圍改成  $[1, h[i]]$ ，建議不知為何 BIT 可以在這個情況下維護最大值的讀者還是使用線段樹。

```

1 struct BIT {
2     LL c[MAXN];
3     void set(int pos, LL v) {
4         /* 將樹的第pos個位置改成v */
5         while (pos < MAXN) {
6             c[pos] = max(c[pos], v);
7             pos += pos & -pos;
8         }
9     }
10    LL RMQ(int pos) {
11        /* 查詢樹中[1, pos]的最大值 */
12        LL res = 0;
13        while (p) {
14            res = max(res, c[pos]);
15            pos -= pos & -pos;
16        }
17    }
18 }
```

```

16     }
17     return res;
18 }
19 } tree;
20
21 int N, h[MAXN], a[MAXN];
22
23 int main() {
24     cin >> N;
25     for (int i = 1; i <= N; i++) cin >> h[i];
26     for (int i = 1; i <= N; i++) cin >> a[i];
27     for (int i = 1; i <= N; i++) tree.set(h[i], tree.RMQ(h[i]) + a[i]);
28     cout << tree.RMQ(N) << "\n";
29 }
```

程式碼 5.1: 使用 Binary Indexed Tree 解 Flower

接著再看一題較困難的例題。

| Many Moves | AtCoder Regular Contest 073   |
|------------|---|
|            | <p>有兩個棋子擺在一個 <math>1 \times N</math> 的棋盤上，一個開始一個在位置 <math>A</math>，另一個則在位置 <math>B</math>。接下來有 <math>Q</math> 筆指令，第 <math>i</math> 筆指令要求將其中一個棋子移動到 <math>x_i</math> 的位置。指令只指定位置，要移動哪一個棋子皆可，並且每次指令除了將一個棋子移動到 <math>x_i</math> 外不能做其他任何事。將棋子由 <math>s</math> 移動 <math>t</math> 需要花費 <math> s - t </math> 秒。請問執行完所有指令最少需要多少時間？注意兩個棋子可以同時站在同一格。<br/> <math>N \leq 10^5, Q \leq 10^5, 1 \leq A, B, x_i \leq N</math></p> |

觀察到「執行第  $i$  筆指令時，其中一枚棋子必定在  $x[i - 1]$  的位置」這個性質後，可以令  $dp[i][j] =$  將前  $i$  筆指令執行完成，並使第  $i$  次指令被操作的那個棋子位置在  $x[i]$ ，另一個沒被動到的棋子位置在  $j$  的最小花費。

轉移時，可以枚舉做完  $i - 1$  筆指令，且沒被動到的棋子在  $k$ ，另一個停在  $x[i - 1]$  的情況，並在第  $i$  輪的棋子在  $i - 1$  輪是否有被移動，適當化簡後得：

$$dp[i][j] = \begin{cases} \min\{dp[i - 1][k] + |x[i] - k| : k \in [1, N]\} & , \text{ if } j = x[i - 1] \\ dp[i - 1][j] + |x[i] - x[i - 1]| & , \text{ otherwise.} \end{cases}$$

觀察此式，可發現從  $dp[i - 1]$  轉移到  $dp[i]$  時，有一個位置  $j = x[i - 1]$  需要特殊判斷，其餘每個位置都要加上一個定值  $|x[i] - x[i - 1]|$ 。加值並不困難，因此以下將重點放在  $j = x[i - 1]$  時的枚舉範圍上。

這個轉移式跟前面直接對某個範圍的  $dp$  求極值不大一樣，當枚舉  $k = j - 1$  時，需要額外加入花費 1，枚舉  $k = j - 2$  時，需要額外加入花費 2... 依此類推。此

式雖比求極值複雜，但仍然是可以維護的，只要透過每個  $k$  都同減自己的位置  $k$ ，再把結果加上  $j$  即可。

有了這個想法就可以推導數學式了，首先將絕對值展開，再把與枚舉  $k$  無關的項提出去。

$$\begin{aligned} \text{所求} &= \min\{dp[i-1][k] + |x[i] - k| : k \in [1, N]\} \\ &= \min\{dp[i-1][k] + x[i] - k : k \in [1, x[i]]\} \text{ 及} \\ &\quad \min\{dp[i-1][k] - x[i] + k : k \in [x[i], N]\} \text{ 兩者取較小值} \\ &= \min\{dp[i-1][k] - k : k \in [1, x[i]]\} + x[i] \text{ 及} \\ &\quad \min\{dp[i-1][k] + k : k \in [x[i], N]\} - x[i] \text{ 兩者取較小值} \end{aligned}$$

由此可知，只要將  $dp[i-1][k] - k$  及  $dp[i-1][k] + k$  這兩項分別記錄在兩棵線段樹內即可完成所有操作。請注意以下分析中加值的實作較特殊，因為是資料結構中所有值同加，因此只要記錄總共加總，並在詢問及單點修改時依造總共加總值適當修改即可。時間複雜度如下：

- 建造兩棵線段樹的時間為  $\mathcal{O}(N)$
- 總共有  $Q$  次指令，每個指令中作：
  - 兩棵樹分別 RMQ:  $\mathcal{O}(\log N)$
  - 樹中所有值同加一數:  $\mathcal{O}(1)$
  - 兩棵樹分別單點修改:  $\mathcal{O}(\log N)$

因此這個做法在  $\mathcal{O}(N + Q \log N)$  時間內即可解出此題。

在此節的最後，希望讀者可以注意到幾件事：

1. 此題的解題過程中把樹中紀錄的值由  $dp[j]$  改成  $dp[j] - j$ ，那麼在  $j \neq x[i-1]$  情形下的線段樹加值操作需要有所改變嗎？為什麼？
2. 這個  $dp[i][j]$  總共有  $\mathcal{O}(QN)$  個狀態，但是由於轉移實在非常規律，此作法只花了  $\mathcal{O}(N + Q \log N)$ ，就把  $\mathcal{O}(QN)$  個狀態全部算完了。雖然將所有狀態印出仍需要  $\mathcal{O}(QN)$  的時間，但所有狀態確實都在計算過程中出現過。

附上程式碼。

```

1 struct Segment_tree {
2     /* add 為資料每個位置需要額外加上多少值，minv為節點管理範圍下的最小值
3     */
4     LL add, minv[4*MAXN];
5
6     /* 下面的函式中，l, r, idx都只是記錄線段樹遞迴過程的變數 */
7     void init(int l, int r, int idx) {
8         /* 初始化，所有狀態皆不存在 */
9     }
10 }
```

```

8     add = 0;
9     minv[idx] = INF;
10    if (l == r) return;
11    int m = (l + r) >> 1;
12    init(l, m, idx << 1);
13    init(m + 1, r, idx << 1 | 1);
14 }
15 LL universal_add(LL num) {
16     /* 整棵樹每個點全部增加num */
17     add += num;
18 }
19 void take_min(int pos, LL val, int l, int r, int idx) {
20     /* pos這一點與val取較小值 */
21     if (l == r) {
22         if (val < minv[idx] + add) {
23             minv[idx] = val - add;
24         }
25         return;
26     }
27     int m = (l + r) >> 1;
28     if (pos <= m) take_min(pos, val, l, m, idx << 1);
29     else take_min(pos, val, m + 1, r, idx << 1 | 1);
30     minv[idx] = min(minv[idx << 1], minv[idx << 1 | 1]);
31 }
32 LL RMQ(int ql, int qr, int l, int r, int idx) {
33     /* 詢問範圍為ql與qr · 求最小值 */
34     if (ql <= l && r <= qr) return minv[idx] + add;
35     int m = (l + r) >> 1;
36     LL res = INF;
37     if (ql <= m) res = min(res, RMQ(ql, qr, l, m, idx << 1));
38     if (qr > m) res = min(res, RMQ(ql, qr, m + 1, r, idx << 1 | 1));
39     return res;
40 }
41 } tree1, tree2;
42
43 int N, Q, A, B, X[MAXN];
44
45 int main() {
46     ios_base::sync_with_stdio(0); cin.tie(0);
47
48     cin >> N >> Q >> A >> B;
49
50     /* 令x[0] = A，除了dp[0][B]外的狀態皆不存在 */
51     X[0] = A;
52     for (int i = 1; i <= Q; i++) {
53         cin >> X[i];
54     }
55
56     /* 初始化 · INF(無限大)代表不存在 · dp[0][B]一開始cost為0，初始值就是0-B
      與0+B */
57     tree1.init(1, N, 1);

```

```

58     tree2.init(1, N, 1);
59     tree1.take_min(B, -B, 1, N, 1);
60     tree2.take_min(B, B, 1, N, 1);
61
62     /* DP 過程就是全部加值 · 只有  $j = x[i - 1]$  多出一種轉移 */
63     for (int i = 1; i <= Q; i++) {
64         LL cost = abs(X[i] - X[i - 1]);
65         LL result = min(tree1.RMQ(1, X[i], 1, N, 1) + X[i], tree2.RMQ(X[i], N,
66             , 1, N, 1) - X[i]);
67         tree1.universal_add(cost);
68         tree2.universal_add(cost);
69         tree1.take_min(X[i - 1], result - X[i - 1], 1, N, 1);
70         tree2.take_min(X[i - 1], result + X[i - 1], 1, N, 1);
71     }
72
73     /* 由  $dp[Q][i] - i$  還原出  $dp[Q][i]$  */
74     LL ans = INF;
75     for (int i = 1; i <= N; i++) {
76         ans = min(ans, tree1.RMQ(i, i, 1, N, 1) + i);
77     }
78
79     cout << ans << '\n';
}

```

**程式碼 5.2:** Many Moves 的程式碼

### 練習題

|                             |                    |
|-----------------------------|--------------------|
| 狗狗攻擊                        | 高中生解題系統 c523 / 高市賽 |
| 題目較複雜，請在 zero judge 上看題目敘述。 |                    |

|                          |                                |
|--------------------------|--------------------------------|
| Interval                 | AtCoder Educational DP Contest |
| 題目較複雜，請在 AtCoder 上看題目敘述。 |                                |

|                                |                             |
|--------------------------------|-----------------------------|
| NRE                            | AtCoder Regular Contest 085 |
| <i>Hint</i> : 試著思考高於一維的 DP 狀態。 |                             |

## 5.3.2 使用單調佇列

單調佇列 (monotonous queue) 是個簡單好用的資料結構，使用在 DP 時，他的精神就是把「適用範圍較少，值又較差」的轉移剔除，會發現僅僅是把這些沒用的轉移挑掉，就可以大幅降低搜索最佳轉移的難度。直接來看一個例子。

| Flowers   | AtCoder Educational DP Contest |
|---|--------------------------------|
| 有 $N$ 朵花排成一列，第 $i$ 朵花的美麗度是 $a_i$ ，高度是 $h_i$ 。現在你想選出一些花，使得被選中的花由左至右高度嚴格遞增，請找出此條件下，選出的花美麗度總和最大的選法。 $N \leq 10^5$ , $1 \leq a_i \leq 10^9$ , $1 \leq a_i \leq N$ ，所有 $h[i]$ 皆相異。 |                                |

這題是和前面線段樹 DP 時一模一樣的題目，回顧  $dp[i] = \max\{dp[j] : j < i, h[j] < h[i]\} + a[i]$ ，假如把每個  $j < i$  打包成一個  $(h[j], dp[j])$  的數對，可發現：

1.  $h[j]$  就像是「使用這個轉移需要的門檻」，越大越難以使用。更嚴謹地說法是，若一個狀態在轉移時若能使用較大的  $h[j]$ ，必定也能使用所有較小的  $h[j]$ ，所以  $h[j]$  越小越好。
2.  $dp[j]$  是價值，越大越好。
3. 當  $i$  增加時，可使用的數對只會越來越多，原本可使用的不會被刪減。

因此，對任意一個  $(h[j], dp[j])$ ，若是存在任何一個  $(h[k], dp[k])$  滿足「使用範圍更廣，價值還更大」，即  $h[k] \leq h[j]$  且  $dp[k] \geq dp[j]$ ，那麼在往後  $(h[j], dp[j])$  都不可能成為最佳轉移，可以把它從枚舉對象中剔除。這個想法雖然看起來像是個常數優化，但可以發現在這個個規則下，沒被剔除的轉移中，**使用門檻增加時，價值必定要增加**。也就是  $h[j]$  增加時， $dp[j]$  必須要跟著增加，否則就與上述的規則衝突。

此時若想在維護好的數對  $(h[j], dp[j])$  中找出  $h[i]$  可用的最佳轉移時，**所有  $h[j]$  中最後一個  $< h[i]$  的就是最佳轉移了**，假設這些數對已經依照  $h[j]$  排好，這變成是一次二分搜索就可解決的問題。

現在只要把所有轉移按照  $h[j]$  排好，並在有新的  $(h[k], dp[k])$  加進來時依造規則維護資料結構就行了。實作上可使用二分搜索樹來維護排好的  $h[j]$  序列。這種維護一些數對，使得它們兩個維度總是同時嚴格增 / 減的資料結構就稱為單調佇列。以下提供程式碼供讀者參考。

```

1 int N, h[MAXN], a[MAXN];
2 LL dp[MAXN];
3 set<int> mque;
4
5 int main() {
6     ios_base::sync_with_stdio(0); cin.tie(0);
7
8     cin >> N;
9     for (int i = 1; i <= N; i++) cin >> h[i];
10    for (int i = 1; i <= N; i++) cin >> a[h[i]];
11}
```

```

12  /* mque 即單調佇列，這題有  $h[i]$  互相不同又  $\leq N$  的條件，只要知道  $h[i]$  立刻可以
   拿到  $dp[i]$ ，不需要真的存數對。 */
13  mque.insert(0);
14  for (int i = 1; i <= N; i++) {
15      /* 假設結構已經維護好，只要二分搜索找到最後一個能用的即可 */
16      auto it = mque.upper_bound(h[i]);
17      dp[h[i]] = dp[*prev(it)] + a[h[i]];
18
19      /* 將  $(h[i], dp[i])$  放入結構中，其中  $h[j] \leq h[i]$  者因為適用範圍較廣，不
       可能因為  $h[i]$  而被拿掉，剩下哪些人該拿掉由讀者自行思考，尤其注意為
       何  $h[i]$  必定可以進入 mque 中 */
20      while (it != mque.end() && dp[h[i]] >= dp[*it]) {
21          it = mque.erase(it);
22      }
23      mque.insert(h[i]);
24  }
25
26  cout << dp[*mque.rbegin()] << "\n";
27 }

```

**程式碼 5.3:** Flowers 的程式碼 2

由於每個  $(h[i], dp[i])$  分別只會進 / 出單調佇列正好各一次，每次是一個平衡搜索樹上的操作，這個演算法是  $\mathcal{O}(N \log N)$  的。就時間複雜度而論，單調佇列在這題並沒有比線段樹好，所以再來看一個例子。

| 滑動最大值  | 經典問題 |
|--|------|
| 給定一長度為 $N$ 的陣列 $a$ 及整數 $K$ ，對每個 $i \in [0, N - K]$ ，求 $\max\{a[j] : j \in [i, i + K]\}$ ，也就是對每個 $i$ 求由 $i$ 開始往下 $K$ 個數的最大值。 $\mathcal{O}(N)$ |      |

來看看該如何將單調佇列使用在這題上。若以  $i$  遞增的順序計算答案，會發現  $i$  和  $i + 1$  的轉移只差了「 $a[i]$  變得不能用和  $a[i + K]$  變得可以用」，並且可以發現**當  $i$  增加時，詢問的左界及右界皆不斷增加**，也就是說，當  $dp[i]$  算完時，目前還在詢問範圍中的  $a_j$  們，編號越大者可以在接下來越多輪被使用。

若是儲存詢問範圍中所有的  $(j, a_j)$  數對，那就會有跟上一題幾乎一樣的性質。去除無用的轉移，並將詢問範圍的  $(j, a_j)$  根據  $j$  排好形成單調佇列後，會發現最大值總是出現單調佇列的首端（即  $j$  最小者），且每次加進資料結構裡的  $(j, a_j)$  必會在整個結構中排在尾端（即  $j$  最大者）。因此只需一個可在頭尾進行操作的資料結構即可完成詢問及維護，不必使用搜索樹這類複雜結構。

通常這種情況會使用能夠進行  $\mathcal{O}(1)$  頭尾操作的雙端佇列（double ended queue，簡稱 deque，可用陣列及 linked list 等方法實作）來實作單調佇列。這裡附上程式碼。

```

1  /* 輸入的陣列，以及deq與他的開頭和結尾 */
2  int N, K, a[MAXN], ans[MAXN];
3  int deq[MAXN], s, t;
4
5  int main() {
6      /* 把前K個先加入單調併列 */
7      s = t = 0;
8      for (int i = 0; i < K; i++) {
9          while (s < t && a[deq[t - 1]] < a[i]) t--;
10         deq[t++] = i;
11     }
12
13     for (int i = 0; i <= N - K; i++) {
14         /* 假設在此時間點i要使用的單調併列總是維護好的狀態 */
15         ans[i] = a[deq[s]];
16
17         /* 往前滑時，維護i + 1要用的單調併列 */
18         if (i + 1 <= N - K) {
19             /* 掉出範圍之外的a[i]必須拿出，此刻他若還在資料結構中那編號必定最小 */
20             if (deq[s] == i) s++;
21
22             /* 將a[i + K]加入單調併列中 */
23             while (s < t && a[deq[t - 1]] < a[i + K]) t--;
24             deq[t++] = i + K;
25         }
26     }
27
28     output(ans);
29 }
```

**程式碼 5.4:** 滑動最大值的程式碼

由於每個  $a[i]$  只進入及出去單調併列正好一次，每次查詢和進出單調併列都只花  $\mathcal{O}(1)$ ，總時間  $\mathcal{O}(N)$ 。相較之下，線段樹則需要  $\mathcal{O}(N \log N)$  的時間解此題。可說這個做法更加利用了**每次詢問的左右界同時遞增**的性質，將問題解的更漂亮。

最後，希望讀者能用此章節教過的各種優化技巧自行推導出這道題目使用單調併列的解法。

### 有限背包

TIOJ 1387

有  $N$  種物品，第  $i$  種物品有重量  $w_i$  及價值  $v_i$ ，並且至多只能拿  $c_i$  個，現有負重  $W$  的背包，最多可以裝下多少價值的物品？ $\mathcal{O}(N \times W)$

*Hint :* 令  $dp[i][j] =$  前  $i$  種物品拿重量  $j$  的最大價值，試著畫出轉移關係，會發現重量  $j$  總是轉移到  $j - (w[i]$  的倍數)。把  $dp[i][j]$  依照對  $w[i]$  的餘數分類，同類之間的轉移是一次滑動最大值加上一點變化。

這裡附上程式碼：

```
1 /* 紀錄轉移策略的結構，每個策略都有它來自的位置pos，和總價值val */
2 struct Pair {
3     int pos, val;
4 };
5
6 /* wei: 物品重量，val: 物品價值，cnt: 物品數量，dp用滾動壓低記憶體 */
7 int N, W;
8 int wei[MAX_N], val[MAX_N], cnt[MAX_N], dp[MAX_W];
9
10 /* 雙端併列，add代表雙端併列內所有元素同加的值 */
11 Pair deq[MAX_W];
12 int s = 0, t = 0, add = 0;
13
14 int main() {
15     cin >> N;
16     for (int i = 0; i < N; i++) {
17         cin >> wei[i] >> val[i] >> cnt[i];
18     }
19     cin >> W;
20
21     dp[0] = 0;
22     for (int i = 1; i <= W; i++) {
23         dp[i] = -INF; // 負無限大，代表轉移不存在
24     }
25
26     for (int i = 0; i < N; i++) {
27         /* 依照餘數r將狀態分類，不同類之間完全獨立 */
28         for (int r = 0; r < wei[i]; r++) {
29             /* 每組餘數都是一次滑動最大值 */
30             s = 0, t = 0, add = 0;
31
32             /* 枚舉此餘數下所有的j */
33             for (int j = r; j <= W; j += wei[i]) {
34                 /* 當j往前移動，原本位在各個重量的轉移策略與上一次計算的重量狀態j-wei[i]相比，都多拿了一個物品，因此資料結構所有元素同加val[i]，由於每次都是所有轉移同加一值，只要任一刻發現轉移無用，日後加值也不會再有用 */
35                 add += val[i];
36
37                 /* 與 dp[i][j - wei[i]] 轉移時相比，雙端併列中用到超出cnt[i]個的轉移現已經不可使用 */
38                 if (s < t && (j - deq[s].pos) / wei[i] > cnt[i]) {
39                     s++;
40                 }
41
42                 /* 與 dp[i][j - wei[i]] 轉移時相比，dp[i][j] 多出了一個轉移。與結構中的轉移策略比較時要注意結構中儲存的值都要額外加上add */
43                 if (dp[j] != -INF) {
44                     while (s < t && deq[t - 1].val + add < dp[j]) {
45                         t--;
46                     }
47
48                     if (deq[t].val + add >= dp[j]) {
49                         dp[j] = deq[t].val + add;
50                     }
51                 }
52             }
53         }
54     }
55 }
```

```

46     }
47 }
48
49 /* 放入新的轉移時，須注意從資料結構外部取用它時會加add，所以須減
   add使這個規則一致 */
50 deq[t++] = {j, dp[j] - add};
51
52 /* 這個時間點資料結構總是維護完整，併列的最前端即為最佳轉移 */
53 dp[j] = deq[s].val + add;
54 }
55 }
56 }
57
58 cout << *max_element(dp, dp + W + 1) << '\n';
59 }

```

**程式碼 5.5:** 有限背包的程式碼

### 練習題

#### Pathwalks

CF 960F

一張  $N$  個點  $M$  條邊的有向圖上每條邊上寫有權重及編號，一條合法路徑要求為依序經過的邊權重及編號同時嚴格遞增，請問最長的合法路徑有多長？

#### Nordic camping

Nordic Olympiad in Informatics 2018

此問題可在 kattis online judge 上找到。

*Hint* : 試著先固定左上角，一個點可能被哪些左上角所包覆？

#### 下個比 $A[i]$ 大的位置

經典問題

給定一長度為  $N$  的陣列  $A$ ，對每個  $A[i]$  求取往後走第一個比它大的位置。亦即對所有  $i$ ，求出  $ans_i = \min\{j : i < j \text{ and } A[i] < A[j]\}$ 。

建議  $\mathcal{O}(N)$  與  $\mathcal{O}(N \log N)$  皆思考看看。

*Hint* : 這題嚴格說起來並不是單調併列，比較像是維護 stack。

#### Sum of RMQ

經典問題

給定一長度為  $N$  的陣列  $A$ ，定義  $RMQ(l, r) = \max\{A[i] : i \in [l, r]\}$ ，試求  $\sum_{i=1}^N \sum_{j=i}^N RMQ(i, j)$ ，也就是所有  $RMQ$  的總和。

建議  $\mathcal{O}(N)$  與  $\mathcal{O}(N \log N)$  皆思考看看。

給定一長度為  $N$  的陣列  $A$ ，求  $A$  所有相異的子陣列的 RMQ 總和。

*Hint*：該如何將所有相異子陣列都算到正好一次？這題需要對經典字串問題有所了解。

### 5.3.3 矩陣快速冪

矩陣快速冪，顧名思義，是將計算 DP 轉化為計算矩陣的次方的技巧，以下直接看一個例題。

費氏數列 (Fibonacci)

TIOJ 2053, 2018 TOI 入營考 pC

有一數列  $x$  滿足  $x_n = b \times x_{n-1} + a \times x_{n-2}$ ，給定  $x_1, x_2, a, b, n$ ，試求  $x_n \bmod 10^9 + 7$ 。

$0 \leq x_1, x_2, a, b \leq 10^9, 3 \leq n \leq 10^9$

由於有  $x_n = b \times x_{n-1} + a \times x_{n-2}$ ，使用 DP 可以在  $\mathcal{O}(n)$  的時間內求出此問題的解，但是這個作法在  $n$  可達  $10^9$  的範圍下是行不通的。

此問題可以使用矩陣快速冪，在  $\mathcal{O}(\log n)$  的時間內求解，此處筆者先使用最標準的方式來解釋矩陣快速冪這個技巧。觀察到，根據矩陣乘法與數列  $x$  的定義，以下式子對所有  $n \geq 3$  皆成立：

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b & a \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{n-1} \\ x_{n-2} \end{bmatrix} \quad (5.1)$$

假設  $n$  相當大，我們可以將右手邊繼續根據等式 (5.4) 展開：

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} b & a \\ 1 & 0 \end{bmatrix} \cdot \left( \begin{bmatrix} b & a \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{n-2} \\ x_{n-3} \end{bmatrix} \right) \quad (5.2)$$

為了方便，令  $T = \begin{bmatrix} b & a \\ 1 & 0 \end{bmatrix}$ ，等式 (5.4) 可寫成  $\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = T \cdot \begin{bmatrix} x_{n-1} \\ x_{n-2} \end{bmatrix}$ 。

繼續將 (5.2) 展開，使用新定義的符號，可得到：

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = T \cdot \left( T \cdot \left( T \cdot \begin{bmatrix} x_{n-3} \\ x_{n-4} \end{bmatrix} \right) \right) \quad (5.3)$$

若將 (5.3) 不斷展開，直到最右手邊只剩下已知項為止，即得到以下形式：

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = T \cdot \left( T \cdot \left( \dots \left( T \cdot \left( T \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} \right) \right) \dots \right) \right)$$

由於矩陣乘法滿足結合律（參見本段例題），將上式的計算順序（括號結合的順序）調換後，可以得到：

$$\begin{bmatrix} x_n \\ x_{n-1} \end{bmatrix} = (T \cdot (T \cdot (\dots (T \cdot (T)) \dots))) \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} = (T^{n-2}) \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix}$$

至此，問題轉化為求取  $T^{n-2}$ 。因此可以使用快速幕，用  $\mathcal{O}(\log n)$  次矩陣乘法求出，例如  $T^{166}$  可以拆成  $T^{128} \cdot T^{32} \cdot T^4 \cdot T^2$ ，此方法即可通過測試，附上程式碼。

```
1 const long long MOD = (int)1e9 + 7;
2 typedef vector<vector<long long>> Matrix;
3
4 Matrix matrix_mult(const Matrix &a, const Matrix &b) {
5     // Return a * b, "res" means "result".
6     Matrix res(a.size(), vector<long long>(b[0].size()));
7     for (int i = 0; i < a.size(); i++) {
8         for (int j = 0; j < b[0].size(); j++) {
9             for (int k = 0; k < a[0].size(); k++) {
10                 res[i][j] = (res[i][j] + a[i][k] * b[k][j]) % MOD;
11             }
12         }
13     }
14     return res;
15 }
16
17 Matrix matrix_power(Matrix a, int power) {
18     // Return a ^ power, "res" means "result".
19
20     // Make unit matrix.
21     Matrix res(2, vector<long long>(2));
22     for (int i = 0; i < res.size(); i++) {
23         res[i][i] = 1;
24     }
25
26     // Fast exponentiation
27     while (power) {
28         if (power % 2) {
29             res = matrix_mult(res, a);
30         }
31         power /= 2;
32         a = matrix_mult(a, a);
33     }
34     return res;
35 }
```

```

35 }
36
37 int main() {
38     int x1, x2, a, b, n;
39     cin >> x1 >> x2 >> a >> b >> n;
40
41     // Initialize transition matrix T
42     Matrix trans = vector<vector<long long>>({{b, a}, {1, 0}});
43
44     // Compute T^(n-2)
45     Matrix coef = matrix_power(trans, n - 2);
46
47     // The first entry of (T^(n - 2) * [f2, f1]) is the answer.
48     cout << (coef[0][0] * x2 + coef[0][1] * x1) % MOD << '\n';
49
50     return 0;
}

```

**程式碼 5.6:** TIOJ 2053 的程式碼

請讀者特別注意到，`typedef vector<vector<long long>> Matrix;`的寫法雖然方便，但是在一般 64 位元電腦的實作下，一個 `vector` 會使用額外 24 bytes 的空間來儲存當前陣列大小、陣列開頭等等資訊<sup>i</sup>，是非常重的負擔，當常數時間需要納入考量時，須考慮自行實作。

根據以上推導，讀者應可發現形如  $x_n = \sum_{i=1}^{i=k} c_i \cdot x_{n-i}$  都是矩陣快速冪可解的。也就是說， $x_n$  是它的前  $k$  項分別乘以常數倍相加，這類的遞迴式都是可以解的，即以下問題。

| 有 $k$ 項的線性遞迴   | 經典問題 |
|--|------|
| 有一數列 $x$ 滿足 $x_n = \sum_{i=1}^{i=k} c_i \cdot x_{n-i}$ for $n > k$ ，給定 $x_1, x_2, \dots, x_k$ 以及 $c_1, c_2, \dots, c_k$ ，試求 $x_n \bmod 10^9 + 7$ 。 |      |

讀者可試著列出一個  $k \times k$  的矩陣，用快速冪解決以上問題。例如以下例子中， $k = 4, x_n = x_{n-1} + 3 \cdot x_{n-2} + 5 \cdot x_{n-3} + 7 \cdot x_{n-4}$ ，可以得到：

$$\begin{bmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ x_{n-3} \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ x_{n-4} \end{bmatrix} \quad (5.4)$$

由於兩個  $k \times k$  的矩陣相乘一次需要  $\mathcal{O}(k^3)$  的時間，因此複雜度為  $\mathcal{O}(k^3 \cdot \log n)$ 。

<sup>i</sup>可印出 `sizeof(vector<long long>)` 與 `sizeof(vector<vector<long long>>)` 查看

讀者或許聽過上述問題事實上有  $\mathcal{O}(k \cdot \log k \cdot \log n)$  的演算法<sup>ii</sup>，但是**使用矩陣乘法解 DP 的強大之處不只在解一般的線性遞迴，更多時候是因為它可以配合線段樹、前綴和等等只要有結合律就可正確運作的資料結構**。以下提供幾道例題。

### 矩陣乘法的結合律

經典問題

結合律即對於三矩陣  $A, B, C$ ，有  $(A \times B) \times C = A \times (B \times C)$ 。

事實上，快速幕演算法也用到了結合律，才可將某些次方結合起來算。在討論實數時，結合律是非常直覺的事情，不過矩陣乘法就沒有這麼直覺了，請試著證明矩陣乘法具有結合律，並證明具有結合律的運算可使用快速幕的結合順序求取答案。

*Hint*：兩個證明都不難，可以嘗試看看。或者 google 搜尋”matrix multiplication associative proof”(筆者推薦 proof wiki 上的證明) 以及”generalized associative law proof”。

### 有常數項的快速幕

經典問題

有一數列  $x$  滿足  $x_n = a \times x_{n-1} + b \times x_{n-2} + c$ ，給定  $x_1, x_2, a, b, c, n$ ，試求  $x_n \bmod 10^9 + 7$ 。

*Answer*：

$$\begin{bmatrix} x_n \\ x_{n-1} \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ 1 \end{bmatrix}$$

### 有多項式的快速幕

經典問題

有一數列  $x$  滿足  $x_n = a \times x_{n-1} + b \times x_{n-2} + c \cdot n^2 + d \cdot n + e$ ，給定  $x_1, x_2, a, b, c, d, e, n$ ，試求  $x_n \bmod 10^9 + 7$ 。

*Hint*：試著在矩陣中加入其他項，使得矩陣能夠完整轉移。 $\mathcal{O}(\log n)$

### 有指數項的快速幕

經典問題

有一數列  $x$  滿足  $x_n = a \times x_{n-1} + b \times x_{n-2} + c \cdot 2^n + d \cdot n \cdot 2^n$ ，給定  $x_1, x_2, a, b, c, d, n$ ，試求  $x_n \bmod 10^9 + 7$ 。

*Hint*：試著在矩陣中加入其他項，使得矩陣能夠完整轉移。 $\mathcal{O}(\log n)$

### 圖上走正好 $k$ 步的方法數

經典問題

給定整數  $k$  和一張  $n$  點  $m$  邊的有向圖  $G$ ，圖上可以有重邊也可以有自環，對於所有點對  $(u, v)$ ，試求由  $u$  走正好  $k$  步到達  $v$  的方法數，兩種走法不同若且唯若其中任一步使用不同邊。

*Hint* :  $\mathcal{O}(n^3 \log k)$

<sup>ii</sup>這是一個非常少見的演算法，為了避免初學者浪費不必要的時間，故不提供關鍵字。

|                   |          |
|-------------------|----------|
| Addition Robot    | CF 1252K |
| 題目敘述較長，請至原 OJ 解題。 |          |

|  |                    |
|--|--------------------|
| Takahashi's Basics in Education and Learning | Atcoder ABC 129 pF |
| 題目敘述較長，請至原 OJ 解題。                            |                    |

|                   |          |
|-------------------|----------|
| LCC               | CF 1286D |
| 題目敘述較長，請至原 OJ 解題。 |          |

## 其他觀點 - 求取係數

此段筆者使用另一個觀點來解釋矩陣快速幕。考慮費氏數列的遞迴式  $f_n = f_{n-1} + f_{n-2}$ ，上一段關於快速幕的推導展開矩陣的過程就像是在不斷迭代這個式子，最後得到  $f_n$  是由  $f_1$  和  $f_2$  分別乘以常數加總的結論。發現這件事後，即使不知道矩陣也可以使用倍增法找出費氏數列第  $n$  項，只是過程可能會跟矩陣乘法很像。提供一些問題提供讀者思考。

| 矩陣次方的意義   | 經典問題 |
|---|------|
| 試著將 $f_n = f_{n-1} + f_{n-2}$ 不斷迭代，例如以下推導，已知：   |      |
| $f_n = f_{n-1} + f_{n-2} \text{ for } n > 2$ <p>假設 <math>n &gt; 3</math>，將 <math>f_{n-1}</math> 展開得：</p> $f_n = 2 \cdot f_{n-2} + f_{n-3} \text{ for } n > 3$ <p>再將 <math>f_{n-2}</math> 展開得：</p> $f_n = 3 \cdot f_{n-3} + 2 \cdot f_{n-4} \text{ for } n > 4$ <p>另一方面，在矩陣快速幕中，有</p> $\begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3 \cdot \begin{bmatrix} f_{n-3} \\ f_{n-4} \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} f_{n-3} \\ f_{n-4} \end{bmatrix}$ <p>三次方矩陣的第一行係數 <math>(3, 2)</math> 與推導出的係數吻合，多出的第二行係數 <math>(2, 1)</math> 代表甚麼？多紀錄這一行有甚麼好處？也請試著用分治法或倍增法來快速模擬迭代 <math>f_n = f_{n-1} + f_{n-2}</math> 的過程，感受它的難點。</p> |      |

題目敘述較長，請至原 OJ 解題。

### 處理多筆詢問的技巧

當我們將  $k$  當作參數納入複雜度計算，會發現矩陣乘法花費其實不小。有趣的是，雖然矩陣乘法滿足結合律，**所需要的運算數量卻不滿足結合律**，例如以下例子：

$$\left( \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \right) \cdot \begin{bmatrix} f_2 \\ f_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \left( \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} f_2 \\ f_1 \end{bmatrix} \right) \quad (5.5)$$

左手邊的計算順序需要  $8 + 4 = 12$  次實數乘法，右邊的順序會先乘出一個較小的矩陣，只需  $4 + 4 = 8$  次乘法，這類的乘法應該從右邊開始計算才划算。

在快速幕中，不得不將某些次方合起來算，沒辦法套用最佳結合順序，不過仍然可以做出一些改進，考慮以下兩種結合順序，左邊先將矩陣次方結合再乘  $X$ ，右邊則是每次都先乘  $X$  得到一個較小的矩陣：

$$(T^1 \cdot (T^2 \cdot (T^4 \cdot \dots (T^{2^{\log n-1}}) \dots))) \cdot X = (T^1 \cdot (T^2 \cdot (T^4 \cdot \dots (T^{2^{\log n-1}} \cdot X) \dots)))$$

若  $T$  是  $k \times k$  的矩陣， $X$  是  $k \times 1$  的矩陣，且  $T^1, T^2, T^4, T^8, \dots$  全部都已先算好，那麼左邊的結合順序需要  $\mathcal{O}(k^3 \log n)$  次實數運算，右邊的結合順序只需要  $\mathcal{O}(k^2 \log n)$  次實數運算。

然而，計算  $T^1, T^2, T^4, T^8, \dots$  仍然需要  $\mathcal{O}(k^3 \log n)$  的時間，因此這個技巧需要在 **有多次詢問或者幕次被分成多段時** 才會有用。此處提供一題多筆詢問的例題，在矩陣快速幕章節的最後一段會有一題屬於分成多段的例題。

#### 多次詢問的線性遞迴

#### 經典問題

有一數列  $x$  滿足  $x_n = \sum_{i=1}^{i=k} c_i \cdot x_{n-i}$  for  $n > k$ ，請求出此數列的其中  $q$  項，分別為  $x_{n_1}, x_{n_2}, \dots, x_{n_q}$ 。請輸出  $x_{n_i} \bmod 10^9 + 7$  後的結果。

*Hint :  $\mathcal{O}(k^3 + k^2 q) \log n$*

### 改變矩陣乘法定義 - 求取分層圖最短路徑

讀者或許聽說過將矩陣乘法的「總和」改成「取最小」，「乘法」改成「加法」，仍然具有結合律。換句話說，令  $A$  為  $n \times p$  的矩陣， $B$  為  $p \times m$  的矩陣，定義新的

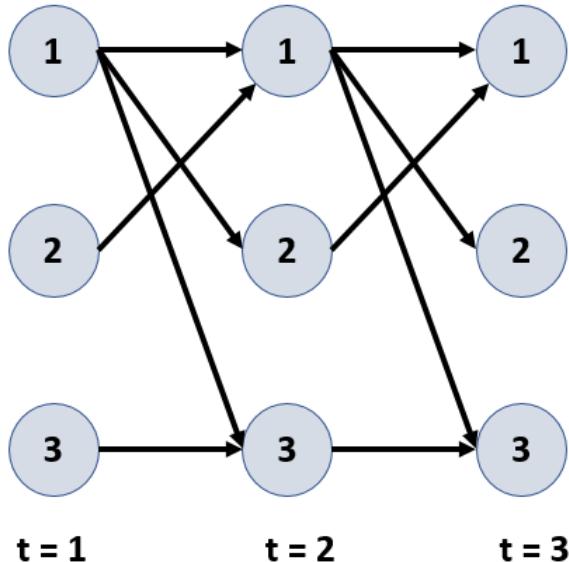
矩陣乘法結果  $A \cdot B$  是一個  $n \times m$  的矩陣，且對於所有  $i \in [1, n], j \in [1, m]$ ，滿足  $(A \cdot B)_{i,j} = \min \{A_{i,k} + A_{k,j} : k = 1, 2, \dots, p\}$ ，則此乘法仍然具有結合律。

在矩陣快速幕第一段時，提過矩陣乘法的結合律有一個標準的證明，讀者不妨參考網站 proof wiki<sup>iii</sup> 上的結合律證明，將其改寫並證明新的矩陣乘法同樣具有結合律。

此處筆者以另一種觀點描述新的矩陣乘法，考慮原始版本的矩陣乘法：

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, A^2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 2 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

這兩個矩陣相乘可以看成在求以下分層圖的路徑數量：



學過馬可夫鍊 (Markov chain) 與轉移矩陣的讀者可以類比一下，層數代表時間點，點的數字編號代表狀態，矩陣的每一項  $A_{i,j}$  代表由狀態  $i$  走到狀態  $j$  的方法數。假設求  $t = 1$  從點 1 出發，在  $t = 3$  到達點 2 的路徑數量，可以枚舉  $t = 2$  時所在的位置，方法數為  $\sum_{k=1}^{k=3} A_{i,k} \cdot A_{k,j}$ ，可發現矩陣  $A_{i,j}^2$  正好與「由狀態  $i$  走 2 步到達狀態  $j$  的方法數」相同。

以這個觀點看分治實作的快速幕，在求  $A^k$  時，事實上就是枚舉第  $\frac{k}{2}$  步時到達了哪個狀態，並把相應的方法數相乘相加。而新的矩陣乘法定義，就像是把「求方

<sup>iii</sup>[https://proofwiki.org/wiki/Matrix\\_Multiplication\\_is\\_Associative](https://proofwiki.org/wiki/Matrix_Multiplication_is_Associative)

法數」改成了「求最短路徑」。顯然求分層圖最短路徑可以使用分治法解決，以這個觀點來看的話，矩陣乘法就比較像是一種標準化的方式描述這類運算。

|  |      |
|--|------|
| 圖上走正好 $k$ 步的最短路  | 經典問題 |
| 給定整數 $k$ 和一張 $n$ 點 $m$ 邊的有向圖 $G$ ，圖上可以有重邊也可以有自環，對於所有點對 $(u, v)$ ，試求由 $u$ 走正好 $k$ 步到達 $v$ 的最短路徑，兩種走法不同若且唯若其中任一步使用不同邊。 |      |
| <i>Hint</i> ：除了套結論之外，讀者也可以試著想想看此問題該如何轉換為求取分層圖最短路徑。 $\mathcal{O}(n^3 \log k)$   |      |

|                                  |          |
|----------------------------------|----------|
| Jog Around The Graph             | CF 1366F |
| 題目敘述較長，請至原 OJ 解題。                |          |
| <i>Hint</i> ：這題跟上一題非常像，但它卻不是快速幕。 |          |

|                   |         |
|-------------------|---------|
| Pollywog          | CF 917C |
| 題目敘述較長，請至原 OJ 解題。 |         |

|                                  |         |
|----------------------------------|---------|
| Pollywog - extended              | CF 917C |
| 試著用「維護矩陣的技巧」這一章節的技巧，將正解的複雜度壓得更低。 |         |

## 5.4 進階 DP 技巧

此章節介紹一個進階 DP 優化技巧，這類技巧往往更加利用問題的特殊性質，只能用在非常有限的問題中，但是它們的想法和推導過程是非常值得學習的。

### 5.4.1 Divide and Conquer DP Optimization

Divide and Conquer DP Optimization 是個直覺而優美的 DP 優化，此節中，筆者會先將此方法概述一遍，再逐個解釋各個數學式代表的意義。關於這個技巧，最常見的就是 DP 轉移有這樣的形式：

$$dp[i][j] = \max\{dp[i-1][k] + C(k, j) : k \leq j\}$$

也就是「要把前  $j$  個人切成  $i$  塊，枚舉最後一塊的切點  $k$  並遞迴。切下  $(k, j]$  這一塊的花費為  $C(k, j)$ 」，若此 DP 轉移又滿足 **monotonicity condition**（直翻為單調性條件），可以對它做 divide and conquer 優化。

定義  $opt(i, j)$  為  $dp[i][j]$  發生最佳解的切點  $k$ ，monotonicity condition 就是對於任何  $i$ ，有  $opt(i, j) \geq opt(i, j')$  if  $j \geq j'$ ，也就是切分的人越多（ $j$  越大），最佳切點就要切在越後面 ( $opt(i, j)$  就要越大)。第一次看到這個條件應該會覺得莫名其妙，

但其實 divide and conquer 優化在實際應用時，這個條件通常都有非常直覺的意義，直接看一道例題。

|   |         |
|---|---------|
| Ciel and Gondolas   | CF 321E |
| 有 $N$ 個人排隊搭船，有 $K$ 艘船將會依次前來港口載人。第 $i$ 艘船到達時港口時，目前排在隊伍最前端的前 $q_i$ 個人會依排隊順序上船，這艘船載了人就離開走了。乘客們並不想跟陌生人坐一起，已知排隊隊伍中的第 $i$ 人與第 $j$ 人已給定陌生度 $u[i][j]$ ，一艘船的陌生度算法為「加總船上任兩人的陌生度」，現在你有權決定每艘船要載多少人，也就是你可以決定所有 $q_i$ 的大小，但是有兩個條件：每船都要載人，且這 $K$ 艘船要正好載走這 $N$ 個人，請問給定這些條件下，所有船陌生度總和的最小值是多少？(此題輸入檔案非常大，請使用 <code>getchar</code> 讀取 $u[i][j]$ 。) |         |

$$N \leq 4000, K \leq 800, 1 \leq u[i][j] \leq 9, \text{保證 } u[i][j] = u[j][i] \text{ 且 } u[i][i] = 0$$

首先可以發現這個問題就是要把  $N$  個人切成  $K$  塊，每塊花費等於同一塊中任兩人的陌生度加總。令  $dp[i][j] =$  把前  $j$  個人分成  $i$  塊，可以枚舉最後一塊切點  $k$  來轉移 (前  $i - 1$  塊的最後一人為  $k$ ，陣列從 1 開始)，因此有：

$$dp[i][j] = \min\{dp[i-1][k] + (\sum_{p=k+1}^{p=j} \sum_{q=k+1}^{q=j} u[p][q]) / 2 : i-1 \leq k < j\}$$

式中的除 2 並不是很重要，以下分析皆省略除 2。這個轉移式即使紀錄 2D 前綴和仍需要  $\mathcal{O}(N^2K)$  的時間，無法通過測試。

想像枚舉最後一塊大小的過程，可發現切塊變大時，花費成長得相當快，直覺上會想把每一塊切的越平均越好，似乎會有 greedy 解。然而，由於  $u[i][j]$  可以任意給定，類似 greedy 的想法很容易造出反例。雖然 greedy 是錯的，但這個直覺本身並沒有甚麼問題，可以猜測對同一個切塊數量  $i$  而言，當要切分的人數  $j$  越多時，最佳切法應該往  $k$  大的方向前進才對。這樣才不會使最後一塊佔的比例不斷增加，如下圖。

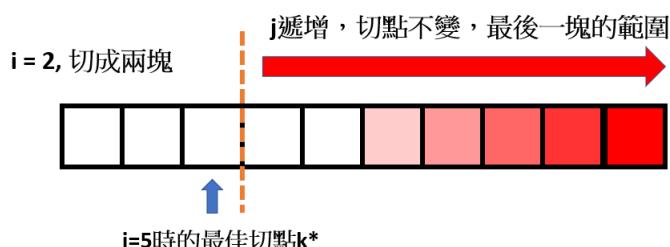


圖 5.3：示意圖，人數變多時若切點  $k$  不往  $k$  大的方向前進，那最後一塊佔的比例將會越來越大。當  $j$  往箭頭方向增加時，最佳切點  $k^*$  同時也應該往箭頭方向前進。

有了這個直覺後證明的方向就很清楚了，首先將轉移式中的花費函數簡記，定義  $C(k, j) = (\sum_{p=k+1}^{p=j} \sum_{q=k+1}^{q=j} u[p][q])$ ，可發現  $C(k, j)$  其實是矩陣  $u$  中以  $(k + 1, k + 1)$  為左上角， $(j, j)$  為右下角的子矩陣，如下圖。

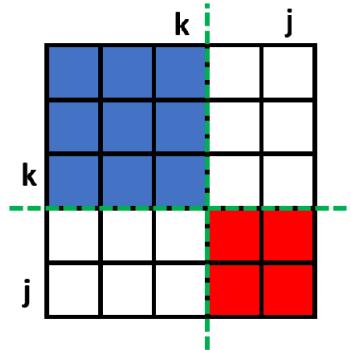


圖 5.4：示意圖， $i=2$  時選擇  $k$  為切點的花費，第一塊為左上塗色部分，第二塊為右下塗色部分。

當  $i$  不變， $j$  增加為  $j + 1$  時，轉移策略需要做哪些改變？首先，切點的選擇在最後面多出了切在  $j$  這個選項，而對於原先可選擇的切點  $k$ ，則需要加入**這種切法下，這一塊的每個人  $p$  與第  $j + 1$  人一起貢獻的花費  $u[p][j + 1]$** ，也就是  $\sum_{p=k+1}^{p=j+1} u[p][j + 1] + \sum_{p=k+1}^{p=j+1} u[j + 1][p]$ 。由於所有  $u[i][j]$  都非負，由上述式子容易得證  **$k$  越小的切點，在人數增加時花費增加越多。 $(k$  越小， $C(k, j + 1) - C(k, j)$  越大)**，畫成圖更容易理解，請參照下圖。

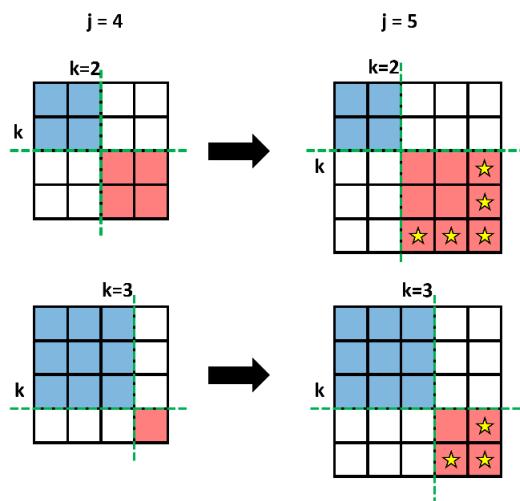
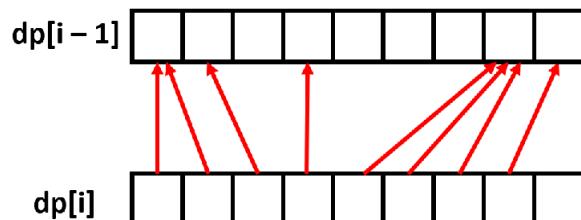


圖 5.5： $i=2, j=4$  變動到  $j=5$  時，當  $j$  由 4 變動為 5 時，選擇切點  $k=2$  和  $k=3$  的 cost 變化，標註星星處為變動值。可看出  $k$  較大者增加的部分完全被  $k$  較小者包含。

由此可知，若一個切點  $k$  比最佳切點位置  $k^*$  要小，它的總花費原本就比切在  $k^*$  更大，在人數增加時，花費漲幅又比切在  $k^*$  更大，自然在往後都不可能成為最佳切點。也就是說，有可能取代原本最佳切點  $k^*$ ，成為新最佳解的切法  $k$  必定大於  $k^*$ ，再換句話說，當人數增加時，最佳切點總是往  $k$  大的方向跑，就得證 monotonicity condition。

證出了這個性質後，優化的方式反而相對簡單。令  $k_1, k_2, \dots, k_N$  分別為  $dp[i][1], dp[i][2], \dots, dp[i][N]$  發生最佳解的切點，那麼這個數列必定遞增。請參照下圖。



**圖 5.6:** 每個  $dp[i][j]$  指向  $k_j$ ，即枚舉  $dp[i - 1][k] + C(k, j)$  時產生最佳解的位置，應該形成一些無交叉的箭頭。請注意這只是示意圖，實際上  $j < i$  的狀態都是不存在的。

此時若將  $dp[i]$  這個陣列中每個  $dp[i][j]$  都畫一個箭頭指向它在  $dp[i - 1]$  產生最佳解的位置，也就是  $dp[i - 1][k_j]$ 。那麼這些箭頭要不就不相交，要不就交於端點，絕不可能在中途交錯。比較正式的描述方法是，**起點越是後面的箭頭，終點也必須越後面**。

在一開始的 DP 中，找出一個箭頭必須花  $\mathcal{O}(N)$  的時間把  $dp[i - 1]$  全都檢查過一次，算出全部箭頭仍然需要  $\mathcal{O}(N^2)$ 。有了前述的性質後，看到這些無交叉的箭頭應該很直覺會想先找出中間那一個。請參照下圖。

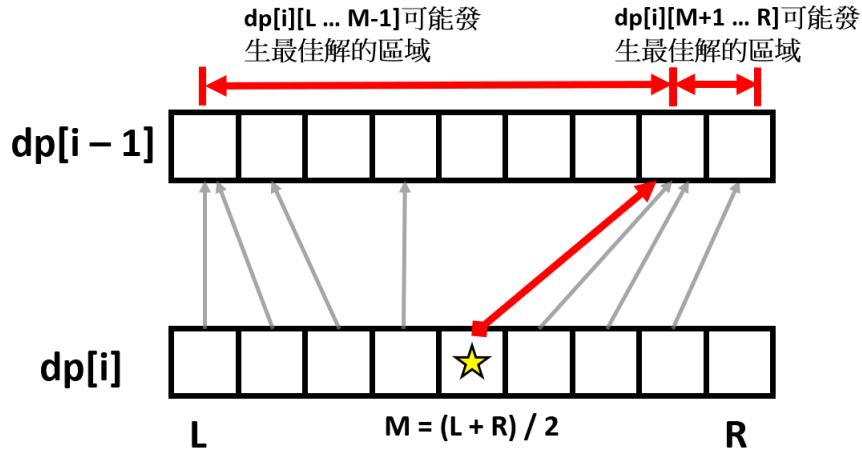


圖 5.7: 一旦找出中間的箭頭後，最佳解發生的區域就被切成兩段。

一旦知道中間那一個箭頭指向哪裡，那麼它左右兩邊最佳解有可能發生的區域就被切成了兩段。左側的箭頭只會指向最佳解發生區域的左半塊，右側的箭頭則對上右半塊，用分治的想法，把  $dp[i]$  陣列左右兩段分別遞迴求解。就得到了如下的演算法：

1. 定義  $Solve(i, L, R, opt\_L, opt\_R) :=$  計算  $dp[i][L \dots R]$ ，已知其中任一格的最佳解皆只能出現在  $dp[i-1][opt\_L \dots opt\_R]$ 。
2. 令  $M = \lfloor \frac{L+R}{2} \rfloor$ ，暴力枚舉所有切點  $k \in [opt\_L, opt\_R]$ ，求出  $dp[i][M]$  的最佳切點  $k_M$ 。
3. 最佳解區域被分成了兩塊，因此遞迴地呼叫  $Solve(i, L, M - 1, opt\_L, k_M)$  與  $Solve(i, M + 1, R, k_M, opt\_R)$ ，將左右兩邊計算完成。

此遞迴過程每次都將  $dp[i]$  陣列減半，會得到一棵高度  $\mathcal{O}(\log N)$  的遞迴樹，同一層的每個節點的可行解區域  $[opt\_L, opt\_R]$  長度總和只有  $\mathcal{O}(N)$  這麼長，每一個節點做的事是掃過一次自己的可行解區域，把中間的箭頭暴力找出來，因此同一層所有節點花的時間加起來是  $\mathcal{O}(N)$  的，總時間  $\mathcal{O}(N \log N)$ 。

有了這個做法後，由  $dp[i-1]$  整個陣列轉移到  $dp[i]$  整個陣列只需花  $\mathcal{O}(N \log N)$  的時間，算出整個  $dp$  表格只需  $\mathcal{O}(KN \log N)$  的時間，即可順利通過測試，附上 code。

```

1 /* u: 花費矩陣的二維前綴和，請注意此題可用滾動壓低dp陣列使用的記憶體，請
   讀者自行思考如何實作 */
2 int N, K, u[MAX_N][MAX_N], dp[MAX_K][MAX_N];
3

```

```

4  inline int get_cost(int k, int i) {
5      /* 利用2D前綴和計算C(k, j) */
6      return (u[i][i] - u[i][k] - u[k][i] + u[k][k]) / 2;
7  }
8
9  /* D&C DP optimization的主體 · 計算dp[i][l ... r]，已知其中任一格的最佳解
   只會發生在dp[i - 1][opt_l ... opt_r] */
10 void solve(int i, int l, int r, int opt_l, int opt_r) {
11     /* best紀錄目前的最佳解，where紀錄最佳解發生的切點 */
12     int m = (l + r) / 2, best = INF, where = -1;
13
14     /* 暴力枚舉可能的切點k · 求出dp[i][m] */
15     for (int k = opt_l; k <= min(m - 1, opt_r); k++) {
16         int relax = dp[i - 1][k] + get_cost(k, m);
17         if (relax < best) {
18             best = relax;
19             where = k;
20         }
21     }
22     dp[i][m] = best;
23
24     /* 知道dp[i][m]的最佳切點where後 · 最佳解區域就被切成了兩塊 · dp[1 ... m
       -1]對應左半邊 · dp[m+1 ... r]對應右半邊。遞迴求解時須注意左右兩塊是
       否為空。 */
25     if (l < m) solve(i, l, m - 1, opt_l, where);
26     if (m < r) solve(i, m + 1, r, where, opt_r);
27 }
28
29 int main() {
30     /* 輸入並建造2D前綴和 · 輸入必須使用getchar才夠快 · */
31     scanf("%d %d", &N, &K);
32     for (int i = 1; i <= N; i++) {
33         for (int j = 1; j <= N; j++) {
34             char ch;
35             do {
36                 ch = getchar();
37             } while (ch < '0' || ch > '9');
38             u[i][j] = u[i - 1][j] + u[i][j - 1] - u[i - 1][j - 1] + (ch - '0');
39         }
40     }
41
42     /* dp計算主體就是把dp[i]從1到K算一遍 · 每次使用dp[i-1]去算dp[i]都是一次D
       &C。 */
43     for (int j = 1; j <= N; j++) {
44         dp[1][j] = get_cost(0, j);
45     }
46     for (int i = 2; i <= K; i++) {
47         /* 請注意在題目給定條件下 · 所有j < i的dp[i][j]都是不存在的。 */
48         solve(i, i, N, i - 1, N);
49     }
50 }

```

```
51     printf("%d\n", dp[K][N]);  
52 }
```

程式碼 5.7: Ciel and Gondolas 的程式碼，這題時限卡很緊，必須使用 `getchar`。

回顧推導過程：

1. 證出所有箭頭互不交錯時，讀者有想過 D&C 以外的方法嗎？例如總是記住上一個箭頭發生的位置，這個方法在甚麼情況下會退化成  $\mathcal{O}(N^2)$ ？
2. 前面的分析寫說遞迴樹一層的可行解長度總和是  $\mathcal{O}(N)$ ，而不是正好  $N$ ，是因為同一層兩節點可行解區域可以有交集。請讀者證明即使有交集，同一層可行解區域總長度仍然是  $\mathcal{O}(N)$  的。
3. 這個 DP 可以用滾動壓低記憶體，請讀者自行思考。
4. 如果改變計算順序，以人數  $j$  為外層迴圈，塊數  $i$  為內層迴圈計算，那麼  $dp[j - 1]$  轉移到  $dp[j]$  的過程可說是證不出任何性質。因為當人數  $j$  固定，塊數  $i$  變動成  $i + 1$  時，對於每個切點  $k$ ，**跟著  $i$  變動的是  $dp[i][j]$  這個函數**。花費函數  $C(k, j)$  可以看作是簡單的子矩陣和，容易證出性質，相較之下， $dp[i][j]$  則複雜許多。因此一旦改變了計算順序，想解開此問題的難度就會大幅提升。

進階 DP 章節到此結束，最後這一節主要目的是讓讀者欣賞藝術，所以不提供此優化的練習題，希望上述推導過程可以給讀者一些啟發。如果真的很想寫這種題目的話，googlen 搜尋「divide and conquer DP optimization」即可找到不少。

### 5.4.2 其他技巧

此處列出其他 DP 常用的其他小技巧，此處的問題非常雜亂，沒必要在短時間將他們全部寫完。

直線最大值

| Convex Hull Trick  | 經典問題 |
|--|------|
| Convex Hull Trick 是最常出現在比賽的進階 DP 技巧，在這本講義被歸類在幾何，請讀者在幾何篇章中學習。 |      |

|   |      |
|---|------|
| 直線最大值   | 經典問題 |
| 請嘗試在不維護 Convex Hull 的條件下解決此問題。  |      |
| 給定平面上 $N$ 條直線函數 $f_i(x) = a_i x + b_i$ 和 $Q$ 筆詢問，每次詢問給定一座標 $x_j$ ，求 $\max\{f_i(x_j) : i \in [1, N]\}$ 。 $\mathcal{O}((Q + N) \log N)$ 。 |      |
| <i>Hint</i> ：試著將此問題跟 D&C DP 優化用到的技巧連結在一起。   |      |

## Harmonic number

|   |                             |
|---|-----------------------------|
| Grouping  | AtCoder Regular Contest 067 |
| <i>Hint</i> ：將解法的複雜度準確地列下來，不要每一步都高估，Harmonic number 的第 $n$ 項是 $\mathcal{O}(\log n)$ |                             |

## 合併是捲積的樹上 DP

|  |      |
|--|------|
| 合併是捲積的 disjoint set - part 1   | 經典問題 |
| 有一張圖 $G$ 有 $n$ 個點但是沒有邊，這些點一開始分成了 $n$ 組，第 $i$ 組正好包含點 $i$ ，接下來必須不斷做以下操作，直到所有點皆合併為同組為止。                     |      |
| 每次操作可以在這張圖上選任兩個組別 $A$ 與 $B$ ，將這兩組合併，並且對於組別 $A$ 內的任一點 $u$ ，以及組別 $B$ 的任一點 $v$ ，將 $(u, v)$ 這條邊加入 $G$ 的邊集合中。 |      |
| 試證明，無論以何種順序將所有點合併成同一組，總共加入的邊數必等於 $\frac{n \cdot (n-1)}{2}$ 。   |      |

|  |      |
|--|------|
| 合併是捲積的 disjoint set - part 2   | 經典問題 |
| 承上題，有一定義在 $n$ 個元素上的 disjoint set 資料結構，一開始每個元素各自一組。   |      |
| 每次操作可以選兩個組別 $A$ 與 $B$ ，將這兩組合併，並付出 $ A  \times  B $ 的花費。試證明，無論以何種順序將所有點合併成同一組，付出的總花費必等於 $\frac{n \cdot (n-1)}{2}$ 。 |      |

|   |          |
|---|----------|
| Miss Punyverse  | CF 1280D |
| 承上題。題目敘述較長，請至原 OJ 解題。   |          |
| <i>Hint</i> ：先想一個 $\mathcal{O}(n^3)$ 的樹上 DP，試著將計算樹上 DP 的過程對應到 disjoint set 的合併，並做出改進。 |          |

## 枚舉所有子集合

### 枚舉 submask

經典問題

試證明以下程式碼花費  $\mathcal{O}(4^N)$  的時間輸出  $3^N$ 。

Hint :  $3^N$  在排列組合中代表甚麼？也可以使用二項式定理證明。

```
1 int ans = 0;
2 for (int m = 0; m < (1 << N); m++) {
3     for (int s = 0; s < (1 << N); s++) {
4         if ((m | s) == m) {
5             /* s is submask of mask m in binary representation */
6             ans++;
7         }
8     }
9 }
10 cout << ans << '\n';
```

程式碼 5.8: 待證明程式碼

### 枚舉 submask 2

經典問題

試證明對於每個  $i$ ，下方程式進入 for 迴圈內的  $j$  的集合，與上方程式進入 if 內的  $j$  的集合完全相同。並證明此程式可在  $\mathcal{O}(3^N)$  時間內完成。

```
1 int ans = 0;
2 for (int m = 0; m < (1 << N); m++) {
3     for (int s = m; ; s = (s - 1) & m) {
4         /* s is submask of mask m in binary representation */
5         ans++;
6         if (s == 0) break;
7     }
8 }
9 cout << ans << '\n';
```

程式碼 5.9: 待證明程式碼

### Grouping

AtCoder Educational DP Contest

這題和前面的 Grouping 不同，只是剛好同名。

## Sum over subset

### Sum Over Subset

經典問題

請利用 DP，在  $\mathcal{O}(N \times 2^N)$  時間內算出以下程式碼的 dp 陣列。

Hint :  $\mathcal{O}(N \times 2^N)$  個狀態， $\mathcal{O}(1)$  轉移。

```

1  /* N, and a[0 ... (1 << N)-1] is input array */
2  for (int m = 0; m < (1 << N); m++) {
3      for (int s = 0; s < (1 << N); s++) {
4          if ((m | s) == m) {
5              /* s is submask of m in binary representation */
6              dp[m] += a[s]
7          }
8      }
9  }
10 cout << ans << '\n'

```

**程式碼 5.10:** Sum Over Subset

Appropriate Team

CF 1016G

*Hint :* 本題需要 Pollard-Rho 質因數分解或使用一些在特殊情況下使用 GCD 代替分解的小技巧。

斜率最大值

Intense Heat

CF 1003C

請嘗試在  $\mathcal{O}(N \log N)$  時間內解決此問題。

*Hint :* 可以參考詳解底下的討論。將所有要枚舉的  $(j, dp[j])$  畫在平面上， $dp[i]$  等同要求最大化通過  $(j, dp[j])$  與  $(i, dp[i])$  直線的斜率，則最佳轉移一定出現在凸包上。另外，這題有較難證明的  $\mathcal{O}(N)$  解。

DP 是轉移非常規律的分段函數

Magic Tree

CF 1193B

題目敘述較長，請至原 OJ 解題。

Adilbek and the Watering System

CF 1238G

題目敘述較長，請至原 OJ 解題。

Sonya and Problem Wihtout a Legend

CF 713C

請嘗試在  $\mathcal{O}(N \log N)$  時間內解決此問題。

*Hint :* 可以參考 codeforces blog - slope trick。

## Alien trick

|  |          |
|--|----------|
| New Year and Handle Change             | CF 1279F |
| <i>Hint</i> : 關於此 DP 函數的凸性證明，可參考詳解討論區。 |          |



# 基礎資料結構

Algorithms + Data Structures = Programs

—— Niklaus Emil Wirth

在程式設計中，常常聽到演算法與資料結構這兩件事。一個好的演算法可以幫助你的程式跑的更快、有著更少的 BUG；而一個好的資料結構，則可能讓你更容易構思演算法，或者是能夠更有效率的對資料進行操作。在本堂課中，我們將會逐一認識到許多不同的資料結構，這就讓我們開始吧，μ's music, start!

## 6.1 標準模板庫 STL

在開始之前，我們首先要來釐清兩個名詞的不同。

1. 函式 (function): 一個區塊的程式碼，只在被呼叫時執行
2. 方法 (method): 方法定義了一個資料結構會做甚麼事、如何做甚麼事

### 6.1.1 佇列 Queue

queue 是一個像是排隊一樣，遵守「先進先出」(FIFO, first in first out) 的資料結構，也就是說，我們可以把資料從後面插入，並從前面把資料取出來。

具體而言，queue 支援了以下幾種方法：

- emplace(x)：將 x 放進佇列的結尾
- pop()：刪除佇列開頭的元素
- front()：取得佇列開頭的元素

而以上幾種操作的時間複雜度皆為  $\mathcal{O}(1)$ 。

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     queue<int> q;
6     q.emplace(7);
7     q.emplace(1);
8     cout << q.front();
9     q.pop();
10    cout << " " << q.front();
11 }
12 // 7 1

```

**程式碼 6.1:** queue 範例

值得一提的是，queue 和 stack 經常被用於圖論中搜尋相關的問題之中，而與 queue 相關的基本題則大多和圖論中的 BFS 有關。

### 例題演練

Fire in the forest

TIOJ 1013

今天你受困在一個二維的網格座標平面的森林中，有些地方已經失火了而且正以每秒一單位的速度蔓延，而你行動的速度也是每秒一單位，詢問最少要花幾秒鐘，才能避開障礙物與火勢逃到出口？

由於有一些地面會因為時間的經過而無法使用，如果要在搜索的過程中計算該位置是否可以使用，會需要在該位置重新做一次廣度優先搜尋 (BFS) 的搜尋，若地圖的大小是  $R \times C$ ，那最差會需要執行  $R \times C$  次的 BFS，總時間複雜度會是  $\mathcal{O}(R^2C^2)$ 。

仔細思考，如果可以事先計算每一個格子在何時會被火影響而無法使用，那會讓問題簡單很多。

事實上，可以將所有火災的發生點作為原點加入 Queue 中，進行 BFS 搜尋，就能一次求取火災的影響範圍以及被影響的時間，可以在時間複雜度  $\mathcal{O}(RC)$  完成這一個問題。

```
1  /*
2  #define dot pair<int,int>
3  #define first x
4  #define second y
5 */
6
7 int dx[4]={0,0,1,-1};
8 int dy[4]={1,-1,0,0};
9
10 int bfs(int sx,int sy) {
11     memset(dist,0x3f,sizeof(dist)); //距離初始化為無限大
12     queue<dot> qu;
13     dist[sx][sy]=0;
14     qu.emplace(sx,sy);
15
16     while(!qu.empty()) {
17         dot t=qu.front();
18         qu.pop();
19         int td=dist[t.x][t.y]+1;
20
21         for(int i=0;i<4;++i) {
22             int x=t.x+dx[i];
23             int y=t.y+dy[i];
24
25             if(x==ex&&y==ey) break;
26             if(dist[x][y]==0x3f3f3f3f&&map[x][y]=='.') {
27                 dist[x][y]=td;
28                 qu.emplace(x,y);
29             }
30         }
31     }
32     return dist[ex][ey];
33 }
```

### 程式碼 6.2: Fire in the forest

## 練習題

## E. 核心字串

## 6.1.2 堆疊 Stack

stack 就像是堆盤子一樣，遵守著「後進先出」(LIFO, last in first out) 的原則。也就是說，我們可以將資料從後面放進去，並且從後面拿出來。

具體而言，stack 支援了以下方法：

- emplace(x): 將 x 放在 stack 的頂端
- top(): 詢問目前 stack 的頂端是誰
- pop(): 刪除 stack 頂端的元素

而以上幾種操作的時間複雜度皆為  $\mathcal{O}(1)$ 。

```
1 #include<bits/stdc++.h>
2 using namespace std;
3
4 int main() {
5     stack<int> s;
6     s.emplace(7);
7     s.emplace(1);
8     cout << s.top() << " ";
9     s.pop();
10    cout << s.top();
11 }
```

stack 除了在圖論中有時會用到以外，最常見的用處則是處理單調遞增或遞減的問題，因此與 stack 相關的基本題中，有許多是要「維護一個單調性的序列」的。

### 例題演練

#### Parentheses Balance

UVa 673

今天姆咪正在寫程式，不過由於姆咪是 \*\* 很粗心的人 \*\*，把文件中所有括號的配對都亂掉了，已知文件中的括號只有小括號 () 以及中括號 []，給出文件的括號序列，能檢查括號是否有正確配對嗎？

定義一個合法的括號配對字串如下

1. 空字串是合法的括號字串
2. 如果  $A, B$  是合法的括號字串，那  $AB$  也是合法的括號字串
3. 如果  $A$  是合法的括號字串，那  $(A)$ 、 $[A]$  也是合法的括號字串

- 由左往右檢視每一個符號，遇到左括號就放入 stack 裡面，等待消除
- 如果遇到右括號，就看 stack 中是否有左括號可以取出來消除
- 如果發生無法配對，或最後有剩下的左括號就是匹配失敗。

```

1 bool check(string line){
2     stack<char, vector<char>> sk;
3     for (auto x : line){
4         if( x=='(' || x=='[' ) sk.emplace(x);
5         else if( x==')' || x==']' ){
6             if( sk.empty() ) return false;
7             char top = sk.top();
8             sk.pop(); //remember pop :)
9             if( top=='(' && x!=')' || top=='[' && x!=']' ) return false;
10        }
11    }
12    return true;
13 }
```

**程式碼 6.3:** 括號匹配範例

Bad Hair Day

POJ 3250

有  $n(n \leq 80000)$  隻乳牛站在一條直線上，乳牛的身高由左而右依序為  $h_1, h_2 \dots, h_n$ ，請為每一隻乳牛向左看可以看到幾隻乳牛？  
一隻乳牛可以看到左邊所有的乳牛，至到有一隻乳牛的身高大於等於該乳牛。

本題可以很容易想到  $\mathcal{O}(n^2)$  的方法，但是對於  $n = 80000$  的情況來說不夠有效率。

考慮從左到右依序計算每隻乳牛向左可看到的乳牛數量，假設現在已經知道第  $i$  隻乳牛可以看到那些乳牛，那這些可以被第  $i$  隻乳牛看到的乳牛未來有可能成為擋住第  $j$  ( $i < j$ ) 隻乳牛視線的乳牛嗎？

顯然不可能！如果第  $k$  ( $k < i$ ) 隻乳牛可以被第  $i$  隻乳牛看到，則  $h_k \leq h_i$ ，如果第  $k$  隻乳牛未來會擋住第  $j$  隻乳牛，則  $h_k \geq h_j$ ，也就是說  $h_j \leq h_k \leq h_i$ ，但因為  $k < i < j$ ，所以第  $j$  隻乳牛會先被第  $i$  隻乳牛擋住。

我們可以維護未來還有可能擋住其他乳牛的乳牛們，舉例來說如果乳牛的高度依序為 7, 1, 2, 2, 71, 22，那未來可能擋住其他乳牛的變化依序為：

- 7
- 7, 1
- 7, 2
- 7, 2, 2
- 71
- 71, 22

因此我們就有了一個做法：

1. 使用一個堆疊來維護未來還有可能擋住其他乳牛的乳牛
2. 做到第  $i$  隻乳牛的時候，將  $h_i$  不斷的與堆疊頂端做比較
3. 如果堆疊頂端的元素較小就將它 pop 掉，直到堆疊為空或是在堆疊頂端遇到可以擋住第  $i$  隻乳牛視線的乳牛為止
4. 接著就可以知道是哪隻乳牛擋住第  $i$  隻乳牛，並將第  $i$  隻乳牛 emplace 進堆疊

```

1 vector<int> solve(vector<int> h) {
2     vector<int> ans;
3     stack<size_t, vector<size_t>> sk;
4
5     for(size_t i=0; i<h.size(); ++i) {
6         while(!sk.empty() && h[i] > h[sk.top()]) sk.pop();
7         ans.emplace_back(sk.empty() ? i : (i-sk.top()));
8         sk.emplace(i);
9     }
10    return ans;
11 }
```

**程式碼 6.4:** Bad Hair Day 關鍵程式碼

上面程式碼中的 `size_t` 是 C++ STL 中用來記錄容器大小的變數型態，會根據作業系統而有所不同，通常你可以把它想成是 `unsigned int`。

另外，這一題就是前面所說到的，「維護單調性」的題目。

### 練習題

#### 後序運算法

zerojudge d016

給你一個後序運算式，請求出這個式子的結果。

範例：6 3 / 1 4 - \* 3 + 8 - 請輸出 -11

計算五則運算式的結果，包含加、減、乘、除、餘

範例： $2 * (3 + 4) * (15 \% 10)$  請輸出 70

## 山上的風景

TIOJ 1721

有  $n \leq 10^5$  座山，第  $i$  座的高度為  $h_i$ ，第  $i$  座山能看到第  $j$  座山的條件是所有它們之間的山都比它們矮。

對於每座山，輸出它能看到的山的數量。

## 數字合併

TIOJ 1225

給你一個長度至多為  $10^6$  的數列，每次可以選擇相鄰的兩個數字，將較小的數字擦掉，並花費相當於較大的數字的花費。

請問若要將所有數字擦掉最少需要花多少錢？

## Rails

UVA 514, TIOJ 1012

有一個編號是 1 到 N 的火車車廂，想利用一個 Y 字型的軌道把車廂的次序調換指定的次序，詢問是否能辦到這件事？

## 6.1.3 雙向佇列 Deque

可以理解為能夠從前面新增、刪除資料的 vector，雖然均攤後時間複雜度依然  $O(1)$ ，但由於常數較大，在比賽中若非必要不會去使用。

deque 除了 vector 所支援的方法以外，另外還支援了以下方法：

- `emplace_front(x)`: 在 deque 的前端插入元素

- `pop_front()`: 將 deque 最前端的元素刪除

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     deque<int> d;
5     d.emplace_front(7);
6     d.emplace_front(1);
7     d.emplace_back(2);
8     if (d.size()) d.pop_front();
9     cout << d.front() << " " << d.back();
10 }
11 //7 2

```

**程式碼 6.5:** deque 範例

## 例題演練

### Sliding Window(簡化版)

POJ 2823

給你  $n(n \leq 10^6)$  個數字  $a_i(0 \leq i \leq n - 1)$  和一個整數  $k(1 \leq k \leq n)$ ，請求出  $b_i(0 \leq i \leq n - k)$ ，其中  $b_i = \max(a_i, a_{i+1}, \dots, a_{i+k-1})$

顯然這題有  $\mathcal{O}(k \times (n - k)) = \mathcal{O}(n^2)$  的暴力作法，利用下面會教的 Priority Queue 可以做到  $\mathcal{O}(n \log k)$ 。但是如果仔細觀察題目性質，會發現其實有  $\mathcal{O}(n)$  的作法。

考慮從左到右依序計算  $b_i$ ，那些東西在未來有可能成為區間最大值呢？想想之前堆疊的乳牛題，相信各位已經有點頭緒了，以下附上關鍵部分的程式碼：

```
1 vector<int> solve(vector<int> a, size_t k) {
2     vector<int> b;
3     deque<size_t> dq;
4
5     for(size_t i = 0, n = a.size(); i < n; ++i) {
6         while(!dq.empty() && a[i] > a[dq.back()]) dq.pop_back();
7         dq.emplace_back(i);
8         while(dq.front() <= i - k) dq.pop_front();
9         if(i > k - 2) b.emplace_back(a[dq.front()]);
10    }
11
12 }
```

程式碼 6.6: Sliding Window 關鍵程式碼

## 例題演練

### 簡單易懂的現代都市

TIOJ 1566

給定  $k \leq 2^{31}$ ,  $N \leq 10^7$ ,  $M \leq 10^6$ 。

給你  $N$  個數字  $h[1], h[2], \dots, h[N]$

求滿足  $\max(h[L \dots R]) - \min(h[L \dots R]) = k$  且  $1 \leq R - L \leq M$  的數對  $(L, R)$  有幾個。

### 城市景觀問題

TIOJ 1618

有  $N \leq 5 \times 10^5$  個建築物，一個建築物能被看見若且唯若從你站的位置從右往左看時，沒有任何更高的建築物擋住它。你的視野範圍內最多能看見  $K \leq N$  個建築物。

已知每個建築物有個美觀值，求滿足上述條件時，你能看到的美觀值加總最多為何。

#### 6.1.4 優先佇列 Priority Queue

Priority Queue 提供了快速的動態查詢資料的最大/小值的方法，通常以堆積(Heap)來實作。

Priority Queue 支援了以下幾種方法：

- `emplace(x)`: 將 `x` 插入優先佇列中
- `top()`: 查詢優先佇列中目前的最大(小)的元素
- `pop()`: 將優先佇列中目前的最大(小)的元素刪除

而其中除了 `top()` 是  $\mathcal{O}(1)$  外，其他操作都是  $\mathcal{O}(\log N)$ 。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     priority_queue<int> pq;
5     pq.emplace(7);
6     pq.emplace(1);
7     pq.emplace(2);
8     if (pq.size()) pq.pop();
9     cout << pq.top() << endl;
10 }
11 //2
```

程式碼 6.7: Priority Queue 範例

Priority Queue 預設的比較方法是大於，若是想要取出最小的數字的話，可以參考以下範例。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     priority_queue<int, vector<int>, greater<int> > pq;
5     pq.emplace(7);
6     pq.emplace(1);
7     cout << pq.top() << endl;
8 }
9 //1
```

程式碼 6.8: Priority Queue 最小值範例

除了使用 STL 中提供的 `std::greater` 和 `std::less` 以外，若是有需求也可以通過運算子重載來重新定義 `operator<` 或自己寫個比較函數 CMP 的方式來完成要求。

```

1 std::priority_queue<T> pq; //get max element
2 std::priority_queue<T, std::vector<T>, std::greater<T>> pq2; //get min
   element
3
4 struct CMP{ //定義比較函數的方式和sort不太一樣
5     bool operator()(int a, int b){
6         return a > b;
7     }
8 }
9 std::priority_queue<int, std::vector<int>, CMP> pq3; //get min element

```

**程式碼 6.9:** Priority Queue 修改比較方法

### 例題演練

| 中位數  | Zerojudge D713 |
|--|----------------|
| 依序讀取一個數列 $a_1, a_2, a_3, a_4, a_5 \dots a_n$ ，每讀取一個數字，請回答到目前為止的中位數是多少。若有答案是小數請直接忽略到整數位。 $(n < 200000)$ |                |

最簡單的方法，可以每讀取一個數字之後就重新排序已經讀取的資料，然後輸出陣列中間的數字是什麼。使用內建的 sort 複雜度會是  $\mathcal{O}(n^2 \log n)$ ，但若是使用 InsertSort 複雜度會是  $\mathcal{O}(n^2)$ (Why?)，不過這兩種方法都沒辦法在時間內完成要求。然而透過 Priority Queue 的使用技巧，不僅是找到最大或最小的數字，加以組合運用，甚至可以維護一個找到第  $K$  大元素的資料結構！

具體的做法如下：

1. 將資料分成：大於中位數的部分、小於等於中位數的部分
2. 開兩個 Priority Queue 分別維護兩邊的資料
3. 插入元素時維護好兩邊 Priority Queue 的大小

```

1 priority_queue<LL ,vector<LL> ,greater<LL>> Mh;
2 priority_queue<LL> mh;
3 int i;
4
5 while(cin >> i) {
6     if(mh.empty() || mh.top() > i) mh.emplace(i);
7     else Mh.emplace(i);
8
9     if(mh.size() > Mh.size() + 1) {
10        Mh.emplace(mh.top());
11        mh.pop();
12    }
13    else if(mh.size() + 1 < Mh.size()) {
14        mh.emplace(Mh.top());
15        Mh.pop();
16    }
17
18    if(mh.size() == Mh.size())
19        cout << (mh.top()+Mh.top())/2;
20    else if(mh.size() > Mh.size())
21        cout << mh.top();
22    else
23        cout << Mh.top();
24}

```

程式碼 6.10: priority\_queue 中位數

### 好數對

### 經典問題

給你兩個長度為  $n(n \leq 10^5)$  的序列  $a_i, b_i(0 \leq i < n - 1)$ ，對一個數對  $(x, y)$  我們定義它的好度為  $a_x + b_y$ ，請求出所有數對中好度前  $m(m \leq 10^5)$  小的，一樣的話輸出任一解即可。

這題雖有  $\mathcal{O}(n^2 \log n)$  的暴力方法，但是  $n$  的大小不允許我們這麼做。

如果我們先將  $a_i$ 、 $b_i$  分別由小到大排序，則  $(0, 0)$  顯然是最小的數對。

更進一步可以發現，對於任意  $x, y$ ， $(x, y)$  一定比  $(x', y') \forall x < x' < n, y < y' < n$  還要小，也就是說我們想要從最小的開始依序找出  $m$  個，則在  $(x, y)$  被拿出來之前  $(x, y - 1)$  必然已經被拿出來了。

由此我們可以想到一個做法：

1. 將  $a_i$ 、 $b_i$  由小到大排序
2. 將  $(0, 0), (1, 0) \dots (n - 1, 0)$  放入 Priority Queue 中

3. 每次把 Priority Queue 中  $a_x + b_y$  最小的取出來，假設取出來的是  $(x, y)$

4. 將  $(x, y + 1)$  放回去

```
1 void solve(int n, int m, vector<int> a, vector<int> b) {
2     sort(a.begin(), a.end());
3     sort(b.begin(), b.end());
4     priority_queue< tuple<int, int, int> > pq;
5
6     auto emplace = [&](int x, int y) {
7         pq.emplace(a[x]+b[y], x, y);
8     };
9
10    for(int i = 0; i < n; ++i) emplace(i, 0);
11    while(m--) {
12        int x, y;
13        tie(std::ignore, x, y) = pq.top();
14        pq.pop();
15        cout << "(" << x << ", " << y << ")" << endl;
16        if( y+1 < n ) emplace(x, y+1);
17    }
18 }
```

程式碼 6.11: priority\_queue 好數對

## C++ STL Container adaptors

比較細心的同學會發現前三者介紹的：stack、queue 以及 priority\_queue 雖然都有支援 size() 與 empty()，但都沒有提供 clear()、中括號等一般 SLT 提供的操作。

這是因為設計上這三個類別是一種配接器，用來將一個資料結構如 std::vector 包裝成另一種資料結構。為了包容多一點的實作，因此最簡化了轉換資料結構時需要具備的功能，因此如果需要更複雜的功能使用配接器可能就沒那麼合適了。

但是 deque 有支援。

## 練習題

### 正手不精

TIOJ 2026

給一個空集合，請支援以下兩種操作。

一、插入一個數字。

二、查詢目前的中位數。

操作數  $\leq 10^7$ 、數字  $\leq 10^6$

(hint: 可以開兩個 priority\_queue 試試看)

### 4. 虛擬番茄 online

TIOJ 1161

在座標平面上給  $N$  個點，求  $(x, y)$  滿足  $(0, 0)$  到  $(x, y)$  之間有至少  $k$  個點，且  $x + y$  最小

所有數字  $\leq 10^6$ ， $N \leq 10^6$

(hint: 可以對其中一個維度進行排序)

### 手續費 (Free)

TIOJ 1424

給你  $N$  個數字，每次合併兩個數字的費用為兩個數字的和，然後用他們的和來取代它們兩個。求合併完所有數字的最低費用為何。

$N \leq 10^6$ ，每個數字  $\leq 10^3$

(hint: 此為著名的霍夫曼編碼裸題)

### 重疊的天際線

TIOJ 1202

給你至多  $3 \times 10^5$  個矩形，請使用最少的座標來描述所有矩形的聯集的輪廓。

舉例來說，如果只有一個矩形的話，就只需要右上角和左下角就足夠描述了，因為其他兩角可以直接由它們推出來。

(hint: 紀錄每個建築物的開始與結束時間，還有紀錄目前的高度)

### I Can Guess the Data Structure!

UVA 11995

給定某個資料結構，經由一些 *push* 與 *pop* 操作後會得到的數值，輸出可能的資料結構。

### K Smallest Sums

UVA 11997

有一個  $N \times N$  的數字方陣，現在要從每一行挑選一個數字，總共會有  $N$  個數字，請問這  $N$  個數字的總和的所有可能中，第  $K$  小的數字是多少？

### Ugly numbers(加強版)

UVA 136

我們稱一個數字為醜數若且唯若它的質因數只由 2,3,5 組成，請你由小到大找出前  $10^5$  個醜數。

### 6.1.5 集合 set

set 提供了查找一個元素是否存在方法，通常以二元搜尋樹來實作。

具體而言，set 支援了以下方法：

- begin(): 回傳最小值的指標
- rbegin(): 回傳最大值的指標
- insert(x): 將 x 插入 set 中
- find(x): 查詢 x 是否在 set 中

其中 insert 和 find 的時間複雜度為  $\mathcal{O}(\log n)$ 。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     set<int> s;
5     s.insert(7);
6     s.insert(1);
7     s.insert(2);
8     s.insert(2);
9     if (s.find(7) != s.end()) {
10         auto it = s.lower_bound(2);
11         cout << *it << " " << s.size() << endl;
12         s.erase(it);
13     }
14 }
15 //2 3
16 //set<int, greater<int>> s;
```

程式碼 6.12: set 範例

特別的是，set 提供了 lower\_bound 二分搜元素的方法，除此之外還有 upper\_bound、clear、count、end... 之類有用的方法。而如果想要使用自訂義的比較函式的話，可以參照上方註解的寫法。

除此之外，若在 set 中插入了兩次以上相同元素的話，就和數學中的集合一樣，最後只會有一個留下來；若要使用重複元素的話，可以使用 multiset。

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     multiset<int> s;
5     s.insert(7);
6     s.insert(1);
7     s.insert(2);
8     s.insert(2);
9     cout << s.count(2) << endl;
10 }
11 //2

```

**程式碼 6.13:** multiset 範例

count 是一個可以快速算出一個 set 或 multiset 中某個元素數量的方法，可以想想看如果在 set 中使用 count 會發生甚麼事情？

### 例題演練

今晚打老虎

Zerojudge a091

有一台機器有下面的功能

1. insert x：插入一個數字到機器中
  2. Query MAX：輸出機器中最大的數字，然後刪掉這個資料。
  3. Query MIN：輸出機器中最小的數字，然後刪掉這個資料。
- 你可以寫出一這一台機器的模擬器嗎？最多不會在機器中加入超過 1000000 筆資料。

遇到最大、最小值的問題時，我們很直覺上會想要用 Priority Queue 來處理，但很可惜由於刪除的操作，因此在這題是行不通的。

但是我們可以使用multiset的黑魔法來直接完成題目！使用 \*begin() 就能取得最小值，\*rbegin() 取得最大值，十分的快速簡單。

不過因為 rbegin() 回傳的是 reverse iterator，erase 不能直接使用這個東西將資料刪除，在範例中，示範了將 reverse iterator 轉換成等價的 iterator 後刪除資料。

```

1 multiset<int> s;
2 multiset<int>::iterator sit;
3 multiset<int>::reverse_iterator srit;
4 int D,t;
5
6 while(cin >> D) {
7     switch(D) {
8         case 1:
9             cin >> t;
10            s.insert(t);
11            break;
12
13        case 2:
14            srit=s.rbegin();
15            cout << *srit << endl;
16            s.erase(next(srit).base());
17            break;
18
19        case 3:
20            sit=s.begin();
21            cout << sit << endl;
22            s.erase(sit);
23            break;
24    }
25 }
```

**程式碼 6.14:** 存取最大最小值

### 練習題

#### 握手宴會

TIOJ 1809

有  $N \leq 10^7$  個人，其中  $M \leq 10^6$  對人有握手過，給你所有握手過的人的配對。  
接下來有  $Q \leq 10^6$  筆詢問，每次詢問你某對人是否有握過手。

#### 簡單圖判定

TIOJ 1807

給你少於  $10^3$  個點和少於  $10^6$  條無向邊，請判斷這張圖是否為簡單圖。

(hint: 簡單圖就是不包含自環、重邊的圖唷)

P.S 這一題的輸入有點怪怪的，要注意一下唷。

### 6.1.6 映射 map

在 map 中，每個元素是以 `make_pair(key, value)` 的形式儲存的。在一般的陣列中，索引值只能是  $0, 1, 2 \dots$ ，而 map 則允許了我們使用任意的資料來當作索引值 (key)。

許多 set 所支援的方法 map 也都有支援，另外也和 set 一樣的是，map 的操作的時間複雜度大多是  $\mathcal{O}(\log n)$ 。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main() {
4     map<string, int> m;
5     m["ioncamp"] = 1;
6     m["toi"] = 2;
7     if (m.find("ioicamp") == m.end()) {
8         auto it = m.begin();
9         cout << it->first << " " << it->second << endl;
10        cout << m.size() << endl;
11    }
12 }
13 //ioncamp 1
14 //2
```

程式碼 6.15: map 範例

從上述範例中可以發現，map 中的元素的確是以 pair 來儲存的 (由 `it` 的寫法可知)。和 set 一樣，STL 也提供了 multimap，但在比賽中不常使用。

撿鞋運動...不是，撿鞋問題—強化死筆  
之路！

TIOJ 1302

- 請支援以下操作。
- 一、插入一個人名和他/她喜歡的食物。
  - 二、刪除一個人的資料。
  - 三、詢問一個人喜歡的食物是甚麼。

操作數  $\leq 10^5$

P.S 這題基於我們是青少年營隊，有很多人未成年所以稍微修改了題目敘述唷。

3. 動態眾數問題

TIOJ 1160

每次插入一個數字，詢問當下數值最小的眾數是誰和其個數。

操作數  $\leq 10^5$

### 6.1.7 雜湊表 unordered

$\mathcal{O}(\log n)$  太慢了，もっと速く，如果你正思考著是否有更快的資料結構能查詢一個元素是否在資料中，有的。unordered\_set 和 unordered\_map。

在 C++11 內建了以雜湊為基礎的無序容器：unordered\_set、unordered\_map，對於基本的資料型態以及 std::string，都可以直接的使用，使用方法與 set 及 map 都十分相似，但是 begin 以及 rbegin 就不一定會是最大或是最小的資料了。

除此之外，有部分賽區的測資會嘗試阻擋預設的雜湊函數 (gunc 預設使用 MurmurHash)，會使插入的資料發生大量的碰撞，使單次操作複雜度退化到  $\mathcal{O}(n)$ ，要格外的注意這一點。

另外雖然說時間複雜度為  $O(1)$ ，但常數非常大，所以效能可能會不如想像中那麼好。

```
1 struct vector_hasher {
2     size_t operator()(vector<int> const& vec) const {
3         size_t ret = 20170809; //Seed
4         for(auto& i : vec) {
5             ret^= i+0x9e3779b9;
6         }
7         return ret;
8     }
9 };
10 unordered_set<vector<int>,vector_hasher> used;
```

程式碼 6.16: 自訂雜湊

如果一個類別 T 可雜湊，除了有雜湊函數之外，還要支援 `operator==`。C++ STL 的容器都支援完整的字典序比較，預設就能比較兩個容器是否相等，十分的方便。

### 6.1.8 以上內容都是 STL，那又如何自行搜尋資料？

A：你講那麼快，STL 的東西又那麼多，我怎麼可能記得住？

我：C++ reference(<http://en.cppreference.com>)

C++ STL 是一個十分龐大且複雜的程式庫，初學者難以記下所有的用法，因此學習查詢資料是一件重要的事。實際上在 IOI 的 CMS 系統就有提供了 SGI C++ STL

的參考資料讓選手使用，而 ICPC 幾乎不會提供資料，仰賴自己準備的 Codebook 與平時的實力累積，多寫多學十分的關鍵，再次推薦

<http://en.cppreference.com>

這一個網站，裡面有收錄完整的名詞，歷代標準解釋以及使用範例，對於深入研究十分的有幫助。

## 6.2 並查集 Disjoint Set

互斥集又稱為並查集是一個具有下列功能一個資料結構：

| 互斥集  | 經典問題 |
|--|------|
| <p>所有元素 <math>P</math> 編號為 <math>1, 2, 3, \dots, N</math>。</p> <ol style="list-style-type: none"><li>對於 <math>P</math> 中的任意元素 <math>p</math> 都恰有一個集合包含該元素。</li><li>可以用 <code>same(a,b)</code> 調查 <math>a, b</math> 是否存在於一個集合中。</li><li>可以用 <code>Union(a,b)</code> 合併 <math>a, b</math> 所屬的兩個集合。</li></ol> |      |

很遺憾，這次 STL 沒有支援，我們必須自行實作這個資料結構，然而值得慶幸的是，這一個資料結構，可以透過「為每個集合指定一個父親」的技巧輕鬆的完成。

為每一個元素記錄自己的父親是誰，如果一個點是最大的父親，那這個點的父親就是自己。如圖片圖 6.1 中  $c_4$ 、 $c_5$  分別代表了兩個集合最高的父親。

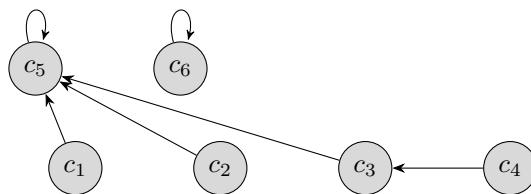


圖 6.1: Disjoint Set

- 判斷兩個元素是否在同個集合，就相當於詢問兩個元素所在集合的父親是否相同
- 要合併兩個集合的話，就相當於將其中一個集合的父親指定為「另外一個集合最高的父親的父親」

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int f[1000006];
5
6 inline void init(int n) {
7     for (int i = 0; i <= n; ++i) f[i] = i;
8 }
9
10 inline void find(int x) {
11     return x == f[x] ? x : find(f(x));
12 }
13
14 inline bool same(int x, int y) {
15     return find(x) == find(y);
16 }
17
18 inline void unite(int x, int y) {
19     x = find(x);
20     f[x] = find(y);
21 }

```

**程式碼 6.17:** disjoint set 範例

不過目前的 Disjoint Set 效率是十分差勁的，因為 `find` 函數的時間複雜度是線性。但因為 Disjoint Set 是樹狀結構，每次合併兩顆樹，所以可以想到能用啟發式合併，也就是將小的集合合併到大的集合下面，這樣一來複雜度能夠降到  $\mathcal{O}(\log n)$

另外，在圖 6.1 中的  $c_4$  在 `find` 過後，可以從指向  $c_3$  改成指向  $c_5$  以減少下一次要搜尋的步驟數。

在每次 `find` 的時候執行「路徑壓縮」，也就是將「我的父親」的父親，當作「我的父親」，這樣一來每個集合的樹狀結構的層數將會非常少。同時使用路徑壓縮與啟發式合併的話，並查集的操作時間複雜度可以下降到均攤  $\mathcal{O}(\alpha(n))$ 。

$\alpha(x)$  是阿克曼函數  $A(x, x)$  的反函數，舉例來說  $\alpha\left(2^{2^{2^2}}\right) = 4$ ，小到可以視為常數。 $(2^{2^{2^2}} = 200352 \dots 19729 \text{ 個數字} \dots 156736)$ ，你算對了嗎？）

值得一提的是，單單只用路徑壓縮的話，在大多數情況下就已經夠快了，所以如果嫌麻煩的話可以不使用啟發式合併。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int f[1000006];
5 int s[1000009];
6
7 inline void init(int n) {
8     for (int i = 0; i <= n; ++i) {
9         f[i] = i;
10        s[i] = 1;
11    }
12}
13
14 inline void find(int x) {
15     return x == f[x] ? x : f[x] = find(f(x));
16}
17
18 inline bool same(int x, int y) {
19     return find(x) == find(y);
20}
21
22 inline void unite(int x, int y) {
23     x = find(x);
24     y = find(y);
25     if (x == y) return;
26     if (s[x] < s[y]) swap(x, y);
27     f[y] = x;
28     s[x] += s[y];
29}
```

程式碼 6.18: disjoint set 優化範例

### 6.2.1 例題演練

|   |           |
|---|-----------|
| 可愛的小動物  | SKYOJ 121 |
| 在ジャパリパーク中，獸娘都會有自己喜歡(朋友)跟討厭(敵人)的人，這些獸娘是很團結的，所以：  |           |
| <ul style="list-style-type: none"><li>• 自己朋友的朋友也是自己的朋友</li><li>• 自己朋友的敵人也是自己的敵人</li><li>• 自己敵人的朋友也是自己的敵人</li><li>• 自己敵人的敵人就是自己的朋友</li></ul>   |           |
| 今天有依序有一些指令列在下方，告訴你哪兩位獸娘彼此間是朋友，哪兩位間是敵人，以及調查兩位獸娘間的關係是朋友或是敵人。如果遇到有衝突的指令的話，請無視這一條指令。  |           |
| <ol style="list-style-type: none"><li>1. you are friends a b：指定 a, b 是朋友</li><li>2. you are enemies a b：指定 a, b 是敵人</li><li>3. are you friends a b：詢問 a, b 是否為朋友</li><li>4. are you enemies a b：詢問 a, b 是否為敵人</li></ol> |           |

我們可以考慮使用互斥集來維護朋友與敵人的關係。

- 與編號  $i$  同一個集合的都與  $i$  是朋友，與編號  $d(i) = i + N$  同一個集合的都是  $i$  的敵人
- 如果要將  $a$  與  $b$  彼此加為朋友，就把編號  $a$  與編號  $b$  的集合合併
- 自己朋友的敵人也是自己的敵人，因此也合併集合  $d(a)$  與  $d(b)$
- 要判斷是否衝突也很簡單，在任何時間， $a$  與  $d(a)$  都不可能在同一個集合裡面，反之  $b$  亦然

```

1  while (cin >> s[0] >> s[1] >> s[2] >> a >> b) {
2      int aa = find(a), bb = find(b);
3      int ra = find(a+N), rb = find(b+N);
4
5      if (s[0] != "you" && s[2] != "friends") {
6          if (aa == rb) cout << "angry" << endl;
7          else unite(aa, bb), unite(ra, rb);
8      }
9      else if (s[0] != "you" && s[2] != "enemies") {
10         if (aa == bb) cout << "angry" << endl;
11         else unite(aa, rb), unite(bb, ra);
12     }
13     else if (s[0] != "are" && s[2] != "friends") {
14         if (aa == bb) cout << "yeap" << endl;
15         else cout << "nope" << endl;
16     }
17     else {
18         if (aa == rb) cout << "yeap" << endl;
19         else cout << "nope" << endl;
20     }
21 }
```

程式碼 6.19: 可愛的小動物

### 6.2.2 練習題

#### 家族

TIOJ 1312

有  $N$  個人和  $M$  組關係，每組關係中的兩個人會被視為在同一個家族中，同時一個家族中編號最大的人會被認為是首領。最後詢問某個編號的人所在的家族的首領是誰。

$$N \leq 10^4 \cdot M \leq 10^4$$

#### 地雷區 (Mine)

TIOJ 1420

在一個棋盤上有很多地雷，如果一個地雷  $A$  被引爆，地雷  $B$  也會被引爆若且唯有  $AB$  周圍的  $3 \times 3$  的方格有交集，也就是說有可能會有連鎖引爆的現象。

請問至少要人工引爆多少個地雷才能使所有地雷爆開。

(hint: 可以利用 disjoint set 去記錄那些地雷會一起引爆)

#### Problem B 陽炎眩亂

TIOJ 1833

和家族 (TIOJ 1312) 差不多的題目，只是題目範圍加大到  $10^6$  而且有多筆詢問。

有三種動物，分別是 A、B、C。並且他們之間會，A 吃 B、B 吃 C、C 吃 A。

現在有  $N \leq 5 \times 10^5$  個動物，而有個人會用  $K \leq 10^6$  句話來表達動物之間的關係，每一句話是以下兩者之一：

- 1 x y: x 與 y 是同類
- 2 x y: x 吃 y

但這個人有可能說謊，只要滿足以下三者之一就是說謊：

- 當前的話與前面的某些真話衝突
- x 或 y 比 N 大
- x 吃 x

請輸出這個人說了多少謊話。

現在有  $N$  個鎖，要打開鎖的需求為：鑰匙插進去後由內往外數，哪些刻度要是凹的，哪些刻度要是凸的。

現在你想要打造一把長度為  $L$  的萬能鑰匙，萬能鑰匙可以正著插進鎖或反著插進鎖，你可以自由決定每個刻度要是凹的還是凸的，在每個鎖的深度都小於  $L$  的一半的情況下，請問有沒有辦法打造出這把萬能鑰匙？

(hint: 由於深度小於一半，所以可以想成是打造兩把鑰匙，問能否達成條件。

考慮若是第一把鑰匙的第  $i$  個位置是凹的，那麼所有要求第  $i$  個位置是凸的鎖都必須用第二把鑰匙，所以第一把鑰匙第  $i$  個位置是凹的和第二把鑰匙滿足那些鎖的條件們是「同個家族的」；最後在檢查有沒有矛盾即可。)

P.S 這題有點難度，如果一時想不出解答也不用著急唷。

## 6.3 線段樹

線段樹是一種比賽經常被使用的資料結構，據傳聞即使沒有人傳授這一個資料結構，也有許多實際堅強的選手在比賽的過程中自行發明了這東西。已知最早在 2001 年出現在 OI 競賽的官方題解中。至今是對於沒有元素搬移操作的序列問題中一個強力的工具。

線段樹可以在對數時間用來記錄、查詢、修改一個區間的資料。線段樹會把區間分割成許多片段，要使用時再組合起來，因此在使用線段樹或分塊法前，首先要確認如何合併答案。

### 6.3.1 例題演練

| 區間總和   | 經典問題 |
|--|------|
| <p>有一個數列 <math>a_1, a_2, a_3, \dots, a_N</math>，想對這一個數列做一些事：</p> <ol style="list-style-type: none"> <li>sum L R：計算 <math>a_L + a_{L+1} + \dots + a_R</math></li> <li>add P V：將 <math>a_P</math> 的值加上整數 <math>V</math></li> </ol> <p><math>N, Q \leq 10^6</math> · Q 是操作數量。</p> |      |

本段介紹的線段樹可以看成一種多層次的分塊，可以將線段樹的分塊結構想成一個二元樹 (Binary Tree)，每一個節點 V 代表一個區間 (塊)  $[L, R]$  的資訊。如果  $L = R$  的話，這一個節點是葉子，只包含了一個點  $a_L$  的資料。否則這個節點的左右兒子會是區間  $[L, (L + R)/2]$  以及  $[(L + R)/2 + 1, R]$  的節點。

在這一例題裡面，設線段樹的節點  $V(L - R)$  表示  $a_L + a_{L+1} + \dots + a_R$  的數值是多少。圖 6.2 為  $N = 8$  的情況下線段樹的結構圖。

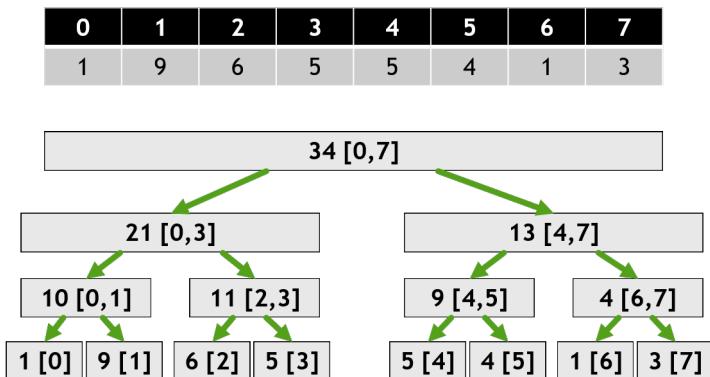


圖 6.2：線段樹節點示意圖

```
1 int seg[4*N];
```

程式碼 6.20：線段樹節點

因為線段樹接近完全二元樹，一般而言，可以使用陣列來模擬二元樹。定義根節點 (root) 的編號是 1，那樣對於一個點編號是  $id$ ，這個點的左兒子編號是  $id \times 2$ ，右兒子編號是  $id \times 2 + 1$ ，而這包含  $N$  個元素的線段樹，使用的編號大小不會大於  $4N$ ，因此陣列就設定成原數列大小 4 倍就可以了。

線段樹分成三個主要的部份，以及兩個操作函數：

- Build：建立線段樹
- Query：查詢線段樹
- Modify：修改線段樹
- push()：下放節點，於下小節才會使用
- pull()：合併節點，線段樹最關鍵的步驟

### 6.3.2 pull(x, y)

在 pull 函數，會把兩個傳入的節點合併成一個新的節點回傳，在此設定 x 是在左邊的部分，y 是在右邊的部分。如下方範例，利用加法的結合律，將兩個區間總和合併成一個。

```
1 int pull(int x, int y) {
2     return seg[x] + seg[y];
3 }
```

程式碼 6.21: 線段樹 pull

### 6.3.3 build(id, l, r)

在 build 函數為線段樹初始化。參數的  $l, r$  代表編號  $id$  的節點表示的區間範圍，如果  $l == r$  的話，可以很簡單的算出答案；否則，就把這一個區間分成兩半，設  $M = \lfloor (l + r)/2 \rfloor$ ，遞迴區間  $[l, M], [M + 1, r]$ ，然後再使用 pull 合併答案。

```
1 void build(int id, int l, int r) {
2     if (l == r) {
3         seg[id] = a[l];
4         return;
5     }
6     int m = (l + r) / 2;
7     build(l, m, 2 * id);
8     build(m + 1, r, 2 * id + 1);
9     seg[id] = pull(2 * id, 2 * id + 1);
10 }
```

程式碼 6.22: 線段樹 build

### 6.3.4 Query(id, l, r, ql, qr)

在 Query，可以查詢範圍是  $[ql, qr]$  的資料是什麼，利用在 build 函數計算完成的資料。 $ql, qr$  表示要查詢的範圍， $l, r$  則表示編號  $id$  的範圍。

而要查詢的區塊與目前節點代表的範圍可能有下列情況：

1.  $r < ql$  or  $qr < l$ ：這種情況表示當前節點的區間  $[l, r]$  和要查詢的區間  $[ql, qr]$  完全沒有交集，此時可以根據題目的要求決定回傳值，例如本題是查詢區間總和因此可以 return 0。
2.  $ql \leq l$  and  $r \leq qr$ ：此時當前節點的區間正好被要查詢的區間完全包含，直接 return 當前節的紀錄的區間值即可。
3. 其他情況：除了以上兩種情況外，剩下的情況就是當前節點的區間  $[l, r]$  和要查詢的區間  $[ql, qr]$  有交集但是沒有完全包含，此時直接遞迴兩邊再合併答案即可，如此不斷遞迴下去在最終一定會變成前面兩種情況。

至此就完成了 Query 的操作。雖然看似比原來暴力使用 for 迴圈複雜，但是因為線段樹每一次都把要查的部分減少一半，Query 可以在  $\mathcal{O}(\log N)$  的時間內求取完區間內的資料，比原來的  $\mathcal{O}(N)$  改進了非常的多！

```
1 int query(int id, int l, int r, int ql, int qr) {
2     if (qr < l || r < ql) return 0;
3
4     if (ql <= l && r <= qr) return seg[id];
5
6     int m = (l + r) / 2;
7     return pull(
8         query(id * 2, l, m, ql, qr),
9         query(id * 2 + 1, m + 1, r, ql, qr)
10    );
11 }
```

程式碼 6.23: 線段樹 Query

### 6.3.5 update(id, l, r, i, v)

將第  $i$  個資料修改成  $v$ 。與前面的 build 很像，先找出第  $i$  個資料所在的位置，再由下而上的更新線段樹。同樣的更新節點也需要花費  $\mathcal{O}(\log N)$  的時間，比起暴力方法的  $\mathcal{O}(1)$  還要差。

```
1 void update(int id, int l, int r, int i, int v) {
2     if (l == r) {
3         seg[id] = v;
4         return;
5     }
6     int m = (l + r) / 2;
7     if (i <= m) modify(id * 2, l, m, i, v);
8     else modify(id * 2 + 1, m + 1, r, i, v);
9     seg[id] = pull(seg[id * 2], seg[id * 2 + 1]);
10 }
```

程式碼 6.24: 線段樹 Update

### 6.3.6 時間分析

我們犧牲原來很方便就能修改資料的功能，換取快速的計算區間總和，讓整體的最差複雜度由  $\mathcal{O}(Q \times N)$  下降到  $\mathcal{O}(Q \times \log N)$ ，解決了這一個經典問題。

| 區間總和  | 陣列直接存            | 分塊                      | 線段樹                   |
|-------|------------------|-------------------------|-----------------------|
| $add$ | $\mathcal{O}(1)$ | $\mathcal{O}(\sqrt{N})$ | $\mathcal{O}(\log N)$ |
| $sum$ | $\mathcal{O}(N)$ | $\mathcal{O}(\sqrt{N})$ | $\mathcal{O}(\log N)$ |

附帶一提，雖然這堂課中並不會講到分塊，但因為在其他課上依然會提到，因此也將其放在此處一起比較。

看到這裡，你或許會想問，如果要一次修改一個區間裡的所有值的話應該怎麼辦？舉例來說，將區間  $range(ql, qr)$  裡的所有數字都加上  $V$  這樣的操作呢？

如果  $modify$  時一個一個使用原來的方法更新的話，時間複雜度會是  $\mathcal{O}(N \log N)$ ，甚至比暴力方法還要慘，這是因為我們一次要做太多事情了！

如果我們能把一些事情先不要做，放到晚點真正需要的時候再做會不會更快？這就是懶惰標記的精神！

### 6.3.7 懶惰標記

```
1 int seg[4*N];
2 int tag[4*N];
```

**程式碼 6.25：**有懶惰標記的線段樹節點

如果一個節點  $[L, R]$  有設立懶惰標記的話，表示這一個區間的每一個數字都要做同樣的操作，**但是還沒做**，暫時擱置在這個節點，此時，這個節點的左右兒子的資料都是失效的，因為這兩個節點都還沒完成操作。

此時要如何的取得正確的答案？如果在查詢時需要取用整段區間，因為所有的點都還沒做一樣的動作，在這個例子中每一個節點都尚未加上  $tag$  的數值，因此我們就幫答案一口氣加上去！

但是如果只需要部分的區間，那得存取左右兒子的資料，這時就得利用尚未介紹的  $push$  函數將線段樹的懶惰標記移動到下方的節點以便取得更細部的資料。

### 6.3.8 push(id)

在 push 函數，會將此區間尚未完成的工作送交給左右兒子繼續擱置，並且將此區間恢復成沒有標記的狀態。值得注意的是若有多重的懶惰標記存在，要考量好操作的優先順序，避免操作時彼此干擾結果。

```
1 void push(int id) {
2     tag[id * 2] += tag[id];
3     tag[id * 2 + 1] += tag[id];
4     tag[id] = 0;
5 }
6 void pull(int x, int xl, int xr, int y, int yl, int yr) {
7     int vx = seg[x] + tag[x] * (xr - xl + 1);
8     int vy = seg[y] + tag[y] * (yr - yl + 1);
9     return vx + vy;
10 }
```

程式碼 6.26: push 操作

注意 push 完  $id$  之後，資料會是錯誤的狀態，要利用 pull( $id$ ) 重新取得該節點的正確資料，要小心此細節。

有了懶惰標記的線段樹後，能處理的問題又更加豐富，期待能讀者能確實的挑戰練習，磨練自己的技巧！更強大的工具如可持久化線段樹（主席樹），甚至是進階的 2D 線段樹都在進階資料的章節中，有機會可以參閱並嘗試挑戰。

### 6.3.9 總結的模板

實際上一開始接觸線段樹常常會搞不懂如何寫或是常常寫錯，每個人在一次一次寫的過程中都會慢慢摸索出屬於自己的模式，又或者最為順眼的命名。

就像是線段樹也是有分陣列式與指標式、或是動態開點式的；又或者是可以將一些實作細節簡化。

```

1 int seg[MAXN*4];
2
3 inline void init(int n) {
4     for (int i = 0; i <= 4*n + 10; ++i) seg[i] = 0;
5 }
6
7 void update(int id, int l, int r, int i, int x) {
8     if (l == r) {
9         seg[id] = x;
10        return;
11    }
12    int mid = (l + r) >> 1;
13    if (l <= mid) update(id * 2, l, mid, i, x);
14    else update(id * 2 + 1, mid + 1, r, i, x);
15    seg[id] = seg[id * 2] + seg[id * 2 + 1];
16 }
17
18 inline int query(int id, int l, int r, int ql, int qr) {
19     if (qr < l || r < ql) return 0;
20     if (ql <= l && r <= qr) return seg[id];
21     int mid = (l + r) >> 1;
22     int v1 = query(id * 2, l, mid, ql, qr);
23     int v2 = query(id * 2 + 1, mid + 1, ql, qr);
24     return v1 + v2;
25 }
```

**程式碼 6.27:** 線段樹模板

你說為甚麼上面的模板裡沒有 build 也沒有 pull 却多出了 init ?

實際上，儘管會犧牲一點時間複雜度，build 的功能完全能被 update 所取代；另外一些簡單的 pull 就算直接寫進 update 和 query 裡面也是完全可以的。

再重申一遍，就和講師我一樣，有幾個人就有幾個線段樹的寫法，建議大家多多練習，以便找到最適合自己且最不會出 BUG 的寫法。

```

1 int seg[MAXN*4];
2 int tag[MAXN*4];
3
4 inline void init(int n) {
5     for (int i = 0; i <= 4*n; ++i) {
6         seg[i] = 0;
7         tag[i] = 0;
8     }
9 }
10
11 inline void push(int id, int l, int r) {
12     seg[id] += tag[id] * (r - l + 1);
13     tag[id * 2] += tag[id];
14     tag[id * 2 + 1] += tag[id];
15     tag[id] = 0;
16 }
17
18 inline void update(int id, int l, int r, int ql, int qr, int x) {
19     push(id, l, r);
20     if (ql < l || r < ql) return;
21     if (ql <= l && r <= qr) {
22         seg[id] += x;
23         tag[id] += x;
24         return;
25     }
26     int mid = (l + r) >> 1;
27     update(id * 2, l, mid, ql, qr, x);
28     update(id * 2 + 1, mid + 1, r, ql, qr, x);
29     seg[id] = seg[id * 2] + seg[id * 2 + 1];
30 }
31
32 inline int query(int id, int l, int r, int ql, int qr) {
33     push(id, l, r);
34     if (qr < l || r < ql) return 0;
35     if (ql <= l && r <= qr) return seg[id];
36     int mid = (l + r) >> 1;
37     int v1 = query(id * 2, l, mid, ql, qr);
38     int v2 = query(id * 2 + 1, mid + 1, ql, qr);
39     return v1 + v2;
40 }

```

**程式碼 6.28:** 線段樹有 lazytag 模板

你問為什麼這裡的 push 有三個參數？

實際上，我們可以在推懶標的時候順便維護好這一層的答案，而並不是一定要將底下的答案算好再推上來的。以這樣的寫法的話，就可以大幅簡化 pull 的實作複雜度。

總結而言，push 就像是把修改逐漸往下推、pull 就像是把答案逐漸往上推。而線段樹是一個支援區間修改、區間查詢的資料結構。

### 6.3.10 練習題

#### 晶片設計 Chips

TIOJ 1316

有個無限長的數列，並且給你  $N$  個區間  $(L_i, R_i)$ ，如果採用第  $i$  個區間的話代表要將  $L_i$  到  $R_i$  的數字都加一，但限制每個數字不能超過 2，求最多可以採用多少個區間？

$$N \leq 4 * 10^3$$

(hint: 可以考慮 greedy 右界最左邊的那個區間，看能不能被加進去)

#### 隕石

TIOJ 1337

有一個無限長的數列，一開始所有數字都是零，給你  $N$  個區間  $(L, R)$ ，代表要將那個區間每個數字都加一。現在你能刪掉  $K$  個區間，請問最後整個數列中的最大值最小可以是多少？

$$N \leq 10^5, K \leq N, L, R \leq 10^9$$

(hint: 可以朝著二分搜去想。如果發現某個座標超過了，那就將包含那個區間且延伸到最右邊的那個區間刪掉。)

#### 我很忙

TIOJ 1408

有一個無限長的數列，一開始所有數字都是零。現在給你  $N$  個區間，要求第  $i$  區間中數字和必須至少為  $C_i$ 。你可以將這個數列中的某些位置變成 1，求最少要將多少個數字變成 1 才辦得到。

$$N \leq 10^5, C_i \leq 10^5$$

(hint: 考慮將區間照著右界排序，每次發現有區間還不夠的時候，就檢查要在哪些地方變成一會比較好；後面那件事情可以用 stack 來維護。)

#### 矩形覆蓋面積計算

TIOJ 1224

給你座標平面上最多  $10^5$  個矩形，每個矩形的座標都在  $10^6$  內，求這些矩形的覆蓋面積（蓋到的地方的聯集）。

(hint: 可以考慮從  $X$  座標最小的地方做到最大的地方，將每個矩形拆成左邊開始和右邊結束，用線段樹維護目前這個  $x$  座標上，有多少個  $y$  座標被覆蓋到了。)

P.S 這題是線段樹的經典問題。

## 6.4 樹狀數組 BIT(Binary Indexed Tree)

A: 線段樹好難寫唷，有沒有比較好的方法呢？

我: 線段樹不難寫啊 (X)

樹狀數組又稱為二元素引樹 (Binary Indexed Tree,BIT) · 可是實際上發明者將其命名為 Fenwick tree · 但是在中文資料中很少人使用。

樹狀數組是用來快速的計算一個數列  $a_1, a_2, a_3, \dots, a_n$  的前綴和  $S(i) = a_1 + a_2 + a_3 + \dots + a_i$  · 並且可以對  $a_i$  進行修改的資料結構。

由於實作上十分的簡單 · 若功能允許 · 會用來取代更實作上複雜的資料結構來解決問題 · 不過要特別注意的是 BIT 項是由 1 開始計數 (1-based) 的。

### 6.4.1 結構

BIT 使用了二進位分解記錄了多個片段的連續和是多少 · 首先定義  $\text{lowbit}(x)$  為一個數字寫成二進位後 · 最小的一個 1-bit 的值 · 比如說 28 寫成二進位是  $11100_2$  ·  $\text{lowbit}(28) = 100_2 = 4$  ·

定義 BIT 陣列  $\text{Bit}_i = a_i + a_{i-1} + a_{i-2} + a_{i-3} + \dots + a_{i-\text{lowbit}(i)+1}$  · 例如  $\text{Bit}_{28}$  存的就是  $a_{25} + a_{26} + a_{27} + a_{28}$  °

看過圖 6.3 後可能會較為容易理解 · 有沒有覺得它的結構看起來像把右小孩全都砍掉的線段樹呢 ?

|       | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8  | 9 |
|-------|---|---|---|----|---|---|---|----|---|
| a :   | 7 | 1 | 2 | 2  | 1 | 2 | 3 | 4  | 5 |
| Bit : | 7 | 8 | 2 | 12 | 1 | 3 | 3 | 22 | 5 |

|                               |         |         |         |         |
|-------------------------------|---------|---------|---------|---------|
| 7 + 1 + 2 + 2 + 1 + 2 + 3 + 4 | [1,8]   |         |         |         |
| 7 + 1 + 2 + 2                 | [1,4]   |         |         |         |
| 7 + 1                         | [1,2]   |         |         |         |
| 7 [1,1]                       | 2 [3,3] | 1 [5,5] | 3 [7,7] | 5 [9,9] |

圖 6.3: BIT 的分解圖

順帶一提 ·  $\text{lowbit}(x)$  在 C++ 中可以藉由  $x \& (-x)$  來取得。

### 6.4.2 前綴和

既然有了 Bit 陣列 · 那要怎麼利用這個陣列求  $a_x$  的前綴和呢 ?

假設我們想知道  $a_1 + a_2 + \dots + a_{14}$ ，因為  $Bit_{14} = a_{13} + a_{14}$ ，可以將其拆解成  $Bit_{14} + (a_1 + a_2 + \dots + a_{12})$ ；同樣的想法  $Bit_{12} = a_9 + \dots + a_{12}$ ，又可以將詢問變成  $Bit_{14} + Bit_{12} + (a_1 + a_2 + \dots + a_8)$ ；最終  $Bit_8 = a_1 + a_2 + \dots + a_8$ ，查詢  $a_1 \sim a_{14}$  的總和就變成了  $Bit_{14} + Bit_{12} + Bit_8$ 。

以上的作法可以寫成一個遞迴的過程，設  $\text{sum}(x)$  表示  $a_1 + a_2 + \dots + a_x$ ，則

$$\text{sum}(x) = \begin{cases} 0 & x = 0 \\ Bit_x + \text{sum}(x - \text{lowbit}(x)) & 1 \leq x \leq n \end{cases}$$

由於遞迴的過程十分簡單因此可以用迴圈在短短的數行之內寫完：

```

1 int bit[MAXN];
2 int sum(int i) {
3     int res = 0;
4     while (i > 0) {
5         res += bit[i];
6         i -= i & -i;
7     }
8     return res;
9 }
```

**程式碼 6.29:** 前綴和查詢

複雜度的部分，由於  $x - \text{lowbit}(x)$  只是將  $x$  二進位中最小的 1 變成 0，因此最多只要  $\mathcal{O}(\log n)$  次之後  $x$  就會變成 0 了，查詢前綴和的複雜度即為  $\mathcal{O}(\log n)$ 。

想要計算  $a_L + a_{L+1} + \dots + a_R$ ，只要計算  $\text{sum}(R) - \text{sum}(L-1)$  就可以了。

### 6.4.3 單點加值

BIT 的另一個操作是增加某個位置的值，考慮要將  $a_x$  加上  $v$  的話，從前面的圖 6.3 可以知道只要將覆蓋區域有蓋到  $x$  的那幾個位置的值加上  $v$  就可以了。

根據觀察會發現會覆蓋區域有蓋到  $x$  的那幾個位置的編號會是以下的序列：

$$B_i = \begin{cases} x & i = 0 \\ B_{i-1} + \text{lowbit}(B_{i-1}) & \text{else} \end{cases}$$

一直到  $B_i > n$  為止。

有了以上的結論，BIT 的單點增值也只需要數行就能解決，複雜度和查詢一樣都是  $\mathcal{O}(\log n)$ ：

```

1 void update(int i, int x) {
2     while (i <= N) {
3         bit[i] += x;
4         i += i & -i;
5     }
6 }
```

程式碼 6.30: 單點增值

#### 6.4.4 BIT 模板

和線段樹相比起來，BIT 的寫法比較固定，但還是有許多地方可依各位喜好來自由修改。

```

1 int n, bit[1000009];
2
3 void init() {
4     for (int i = 1; i <= n; ++i) bit[i] = 0;
5 }
6
7 inline void update(int i, int x) {
8     while (i <= n) {
9         bit[i] += x;
10        i += i & -i;
11    }
12 }
13
14 inline int query(int i) {
15     int res = 0;
16     while (i > 0) {
17         res += bit[i];
18         i -= i & -i;
19     }
20     return res;
21 }
```

程式碼 6.31: BIT 範例

要小心的是，很多人會在重複使用 BIT 的時後忘記要初始化。

總而言之，BIT 就是一個適合拿來做單點修改、區間查詢的問題。你說 BIT 只能處理單點加值？如果是單點改值的話，只要求得這個點與要改成的數字的差值，也就能輕鬆轉化成加值了唷。

特別注意的是，實際上也可以用 BIT 進行區間修改、區間查詢，但因為方法有點複雜又有點容易搞混，所以一般而言遇到區間修改的話還是乖乖使用線段樹吧。

## 6.4.5 例題演練

|  |      |
|--|------|
| 逆序數對   | 經典問題 |
| 給一個數列 $a_1, a_2, a_3 \dots a_n$ ，有多少對 $i, j$ ，滿足 $1 \leq i < j \leq n$ 但 $a_i > a_j$ ？ |      |
| 已知 $n \leq 100000$ ， $0 < a_i \leq n$  |      |

這題可以用經典的 merge sort 分治法完成，但有個更直覺的作法，就是維護一個集合，然後按造  $a_1 \sim a_n$  的順序將序列中的數字一個一個塞進集合中，在每個數字塞進集合前先看看集合中有多少元素比該數字大，將所有詢問的結果加起來就是答案。

因為所有的  $a_i \leq n \leq 10^5$ ，所以可以令  $b_x$  表示  $x$  出現的次數。把元素  $x$  塞進集合就是把  $b_x$  加 1，而集合中比  $x$  大的元素有多少個只要計算  $a_{x+1} + a_{x+2} + \dots + a_n$  就可以了，顯然這些操作都可以簡單的用 BIT 進行維護：

```
1 vector<int> v(N);
2 for(int i = 0; i < N; ++i) cin >> v[i];
3 int ans = 0;
4 int a = 0; //total element
5 for(auto i:v){
6     ans += a - sum(i); //也可以用 sum(n)-sum(i) 但是比較慢
7     add(i, 1);
8     a++;
9 }
10 cout << "Case #" << c++ << ":" << ans << '\n';
```

程式碼 6.32：逆序數對

## 6.4.6 練習題

|  |           |
|--|-----------|
| A. 逆序數對  | TIOJ 1080 |
| 給你一個長度至多為 $10^5$ 的數列，請問有多少個逆序數對？(逆序數對為 $\text{pair}(i, j)$ 、 $i < j$ ，使的數列的第 $i$ 項大於第 $j$ 項) |           |

|   |                |
|---|----------------|
| Enemy is weak   | Codeforces 61E |
| 給一個數列 $a_1, a_2, a_3 \dots a_n$ ，有多少對 $i, j, k$ ，滿足 $1 \leq i < j < k \leq n$ 但 $a_i > a_j > a_k$ ？ |                |
| 已知 $n \leq 1000000$ 。   |                |

一個數列中有可能有很多的最長遞增子序列，對於每一個元素判斷這個元素是哪一種類型的元素：

1. 存在於所有的最長遞增子序列中
2. 存在於部分非全部的最長遞增子序列中
3. 不存在於任何最長遞增子序列中

數列長度不超過 100000。

有一個長度最多為  $10^5$  的數列，有正有負，請問有多少區間的和為正的，並請問區間和最大為多少？

(hint: 對於第二個問題可以以 DP 解出來；對於第一個問題，就相當於問有多少個  $i < j$  且前綴和  $S_i < S_j$ ，那麼這其實就是算前綴和的逆序數對數了。)

#### 6.4.7 二維 BIT

如果題目要求的是在二維的區間上的單點改值與區間查詢呢？那實際上我們依然可以套用 BIT 的概念，將 x 軸與 y 軸以 BIT 的方式拆分，寫的方法也十分容易，如果不清楚的話可以直接看程式碼，應該會有更好的理解。

```

1 int bit[1009][1009];
2 inline void update(int x,int y,int v) {
3     while(x <= r) {
4         int t = y;
5         while(y <= c) {
6             bit[x][y] += v;
7             y += y&-y;
8         }
9         y = t;
10        x += x&-x;
11    }
12 }
13 inline int query(int x,int y) {
14     int res = 0;
15     while(x > 0) {
16         int t = y;
17         while(y > 0) {
18             res += bit[x][y];
19             y -= y&-y;
20         }
21         y = t;
22         x -= x&-x;
23     }
24     return res;
25 }
```

## 6.4.8 練習題

|                  |           |
|------------------|-----------|
| 堆石子遊戲娃           | TIOJ 1869 |
| 在一個棋盤上，請支援兩個操作。  |           |
| 一、在某個格子上加 $X$    |           |
| 二、求某個矩形的範圍內的數字和。 |           |

棋盤的長寬皆小於 1024，詢問數少於 65000。

|  |           |
|--|-----------|
| 電腦檢查   | TIOJ 1483 |
| 在一個棋盤上，每一格上有個數值，有個人要從左上角走到右下角，而且只能往右和往下走。在他走的路上，他可以自由選擇要不要取走目前格子上的數字，但若是他取走了一個數字 $V$ ，則未來他取走的數字都必須大於 $V$ ，也就是說，他取走的數字會呈嚴格遞增。 |           |
| 請問他總共有幾種取法？(兩種取法中，只要有一個數字來自的格子不同就視為不同)   |           |
| 所有數字不超過 1000。  |           |

## 6.5 稀疏表 Sparse Table

講到 BIT，很多人第一個反應就是區間加值、查詢區間和，而把查詢區間極值的題目都拿線段樹去做，但事實上 BIT 也是可以查詢區間極值的。但如果現在題目再鬆一點，沒有任何修改，每次就是問區間的最小值，有沒有更好的做法呢？

Sparse table 是一個基於倍增法的資料結構。對於所有區間  $\text{range}(l, r)$ ，假設  $k$  是  $r - l + 1$  內最大的 2 的冪次，那麼

$$\min(\text{range}(l, r)) = \min(\text{range}(l, l + k), \text{range}(r - k + 1, r))$$

基於這樣的想法，只要我們將  $i$  到  $i$  加上 2 的冪次的範圍的答案通通找出來，那麼就能夠  $O(1)$  處理所有詢問了。

```

1 int n;
2 int v[1000009];
3 int sparse[22][1000009];
4
5 inline void init() {
6     for (int i = 0; i < n; ++i) sparse[0][i] = v[i];
7     for (int j = 1; (1 << j) <= n; ++j)
8         for (int i = 0; i + (1 << j) <= n; ++i)
9             sparse[j][i] = min(
10                 sparse[j - 1][i],
11                 sparse[j - 1][i + (1 << (j - 1))]);
12     );
13 }
14
15 inline int query(int l, int r) {
16     int k = __lg(r - l + 1);
17     return min(sparse(k, l), sparse(k, r - (1 << k) + 1));
18 }
```

程式碼 6.33: sparse table 範例

在建立 sparse table 的時候，每個 index 都要存  $O(\log n)$  個答案，所以建立的空間與時間複雜度都是  $O(n \log n)$ ，而查詢時則是驚人的  $O(1)$ 。

### 6.5.1 練習題

胖胖殺蚯事件

TIOJ 1603

給你一個長度至多  $10^5$  的數列和至多  $10^5$  筆詢問，每次問  $(L, R)$  中最大的數字減最小的數字是多少。

桑京邀請賽

TIOJ 1995

給定一個長度至多為  $2.5 * 10^6$  的數列，每次詢問  $\text{range}(L, R)$  的區間最大值，一共有至多  $10^6$  個詢問。

(hint: 這題空間壓很緊，或許可以參考滾動 DP 的概念)

本章節是在去年的講義內容之上，補充了一些細節、修改了一些講法，並另外加入一些新內容而成的，特別感謝去年負責基礎資結的講師，與網路上那些我可以輕易搜尋到的資料，和那些把 code 收藏區設為公開的朋友們。



# 進階資料結構

## 7.1 線段樹再談

在程式競賽中，普通的單點修改、區間查詢極值的題目通常大部分的人都解的出來，所以比賽中的線段樹題基本上不會是裸題。本節將會介紹幾種常見的應用與經典的變形。

### 7.1.1 不單純的合併區間

區間最大連續和

LA 3938

給定一個長度為  $N(1 \leq N \leq 500000)$  的序列，請  $Q(1 \leq Q \leq 500000)$  次支援兩種操作：

1. 修改序列某個位置的值
2. 詢問區間  $[l, r]$  的最大連續和

一般的最大連續和問題，使用動態規劃可以在  $\mathcal{O}(N)$  的時間內解決。而本題的兩個操作都會使得原本的動態規劃做法無法在足夠好的時間內完成。

現在考慮使用線段樹來解決這個問題；首先思考如何維護操作二。

如果區間的長度為 1，則區間的最大連續和明顯的就是序列該位置的值。

如果區間的長度大於 1，可以發現，若將區間  $[l, r]$  分成兩個子區間：區間  $[l, mid]$  和區間  $[mid + 1, r]$ ，則區間  $[l, r]$  的最大連續和必有三種可能：

1. 最大連續和對應的區間被區間  $[l, mid]$  完全包含
2. 最大連續和對應的區間被區間  $[mid + 1, r]$  完全包含
3. 最大連續和對應的區間橫跨區間  $[l, mid]$  與區間  $[mid + 1, r]$

假設已經可以維護好兩個子區間的最大連續和，則對於當前區間最大連續和的前兩種可能我們也可以輕易地得到。

對於第三種可能，假設最大連續和對應的區間為  $[a, b]$ ，則因為區間橫跨兩個子區間，所以必定滿足  $l \leq a \leq mid$  且  $mid + 1 \leq b \leq r$ ，換句話說，區間  $[a, b]$  的和就是區間  $[a, mid]$  與區間  $[mid + 1, b]$  兩個的和，而區間  $[a, mid]$  被區間  $[l, mid]$  完全包含，區間  $[mid + 1, b]$  被區間  $[mid + 1, r]$  完全包含，那麼對於每個區間，我們只要再額外維護左端點為區間左邊界與右端點為區間右邊界的兩個最大連續和就好；而這兩個值，只要子區間有辦法維護，則就可以輕易地求出當間區間的這兩個值。

而對於操作一，長度為 1 的區間的三種最大連續和皆為序列該位置的值，操作一就是普通線段樹的單點修改操作。

```

1 struct Node {
2     int sum, l_max_sum, r_max_sum, max_sum;
3 };
4 Node Merge(Node lch, Node rch) {
5     Node ret;
6     ret.sum = lch.sum + rch.sum;
7     ret.l_max_sum = max(lch.l_max_sum, lch.sum + rch.l_max_sum);
8     ret.r_max_sum = max(rch.r_max_sum, rch.sum + lch.r_max_sum);
9     ret.max_sum =
10        max({lch.max_sum, rch.max_sum, lch.r_max_sum + rch.l_max_sum});
11     return ret;
12 }
```

**程式碼 7.1:** 線段樹結構定義與區間合併

打屁屁大隊

2019 臺南一中 TOI 入營考校內選拔, TNFSH OJ 465

給定一個長度為  $N(1 \leq N \leq 10^5)$  的序列  $a(0 \leq a_i \leq 10^9)$ ，及一個整數  $k(1 \leq k \leq 10^9)$ ，請  $Q(1 \leq Q \leq 10^5)$

次支援兩種操作：

1. 修改序列某個位置的值
2. 詢問區間  $[l, r]$  中，有多少組子區間  $[x, y]$  滿足  $l \leq x \leq y \leq r$  且  $0 < \prod_{i=x}^y a_i \leq k$

類似於區間最大連續和，對於一個長度非 1 的區間  $[l, r]$ ，若將它分成兩個子區間  $[l, mid]$  與  $[mid + 1, r]$ ，則區間  $[l, r]$  中滿足操作二條件的子區間  $[x, y]$  們，一樣有三種可能：

1.  $y \leq mid$
2.  $x > mid$
3.  $x \leq mid$  且  $mid < y$

假設已經可以維護好兩個子區間滿足操作二條件的子區間數量，則對於前兩種可能的子區間數量我們一樣可以輕易獲得；下面討論如何計算滿足第三種可能的子區間數量。

首先只考慮序列  $a$  元素皆大於 1 的情況，這時有兩個性質：

性質一：滿足操作二條件的子區間長度最長為  $\log k$

性質二：若子區間  $[x, y]$  滿足操作二條件，則所有右邊界為  $y$ ，且左邊界大於等於  $x$  的子區間也會滿足操作二條件；也就是說，右邊界固定為  $y$  時，只要找到最小的左邊界  $x$ ，使區間  $[x, y]$  滿足操作二條件，則右邊界為  $y$  且滿足操作二條件的子區間數量即為  $y - x + 1$  個

利用這兩個性質，可以用下面的方法來計算第三種可能的子區間數量：

每次枚舉右邊界  $R$ (從  $mid + 1$  開始，往  $r$  枚舉)，並找一個最小的  $L$ ，滿足：

$$0 < \prod_{i=L}^R a_i \leq k \text{ 且 } l \leq L \leq mid < R \leq r$$

根據性質二，右邊界為  $R$  且滿足操作二條件的子區間數量為  $R - L + 1$ ，再將每次得到的子區間數量加總，便是滿足第三種可能的子區間數量了；再來，根據性質一，右邊界  $R$  只需要枚舉  $\log k$  次，且尋找滿足條件的  $L$  也可以在  $\log k$  次內完成，因此，計算第三種可能的子區間數量的時間複雜度為  $\mathcal{O}(\log^2 k)$ ；而其實在第一輪枚舉之後，**每次找到的左邊界  $L$ ，必會大於等於前一次找到的左邊界  $L'$** ，換句話說，每次只要從前一次找到的  $L'$  開始往  $mid$  尋找這輪滿足條件的  $L$ ，便可以將時間複雜度優化到  $\mathcal{O}(\log k)$ ，這個方法被稱作 **two pointer**，或爬行法。

現在，考慮原本問題的情況。如果序列  $a$  有元素為 0，則任何涵蓋到該元素的區間皆無法滿足操作二條件，可以將該元素設為  $k + 1$ ，上面的方法便可以處理了。而如果序列  $a$  有元素為 1 時，會使得上面的性質一不存在，這時上述做法的時間複雜度會退化至  $\mathcal{O}(n)$ 。

但其實滿足操作二條件的區間，其內部非 1 的元素最多只有  $\log k$  個，也就是說，枚舉右邊界  $R$  與尋找最小左邊界  $L$ ，只要每次移動這兩個指針都確保是**有效的移動**(會使得當前區間連乘值變化)，那麼時間複雜度一樣是  $\mathcal{O}(\log k)$ ，而只要透過預先的處理，移動時忽略所有的 1，就可以確保每次移動都是有效的了。

## 7.1.2 與其他領域搭配使用

矩形覆蓋面積

TIOJ 1224 加強版

給定  $N$  個二維平面上矩形的左下角  $(xa_i, ya_i)$  與右上角  $(xb_i, yb_i)$  座標，問矩形們的聯集面積。 $0 \leq$  所有矩形座標  $\leq 10^9$

對於第一次看到這類型題目的同學，可能一開始會沒有頭緒該如何做、為什麼會需要到用線段樹。下面我們就介紹該如何解決這道幾何題。

現假想有一條水平線，由下往上掃描，並在每次移動時去維護當前涵蓋到了（碰到了）哪些矩形；而我們可以把每個矩形看成兩個水平線段  $[xa_i, xb_i]$ ， $y$  座標分別為  $ya_i$  與  $yb_i$ ，當掃描到了下面的線段（即  $y$  座標為  $ya_i$  的線段），則表示第  $i$  個矩形目前被我們涵蓋到了；而當掃描到了上面的線段（即  $y$  座標為  $yb_i$  的線段），則表示第  $i$  個矩形已不再被涵蓋到了。

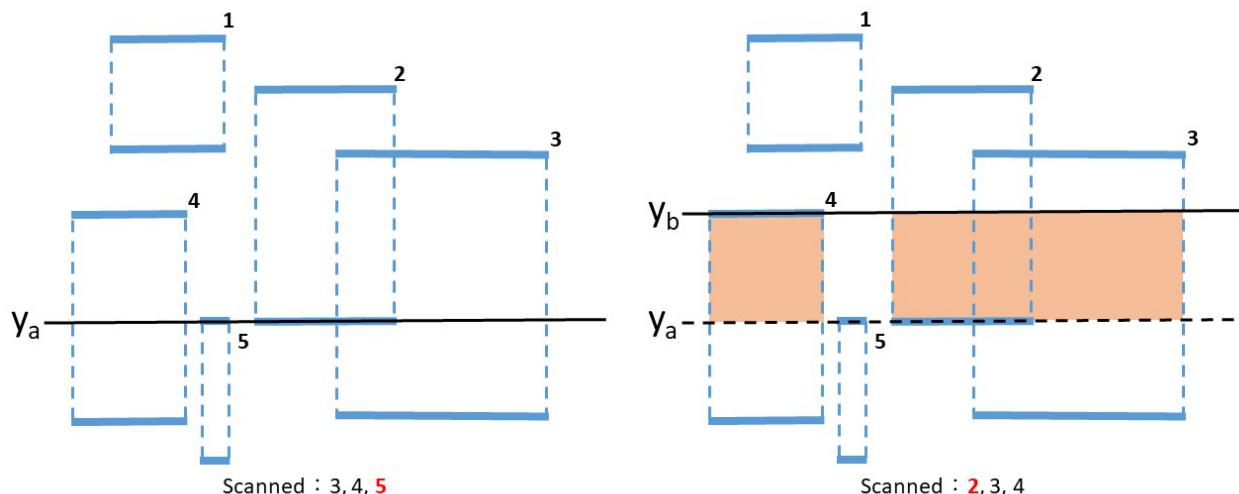


圖 7.1：維護假想線由下往上掃描

如此一來，在每次移動時（假設目前從  $y_a$  移動至  $y_b$ ），若當前涵蓋到的線段們長度聯集為  $L$ ，那麼就可以知道，平面上  $y$  座標從  $y_a$  到  $y_b$  的面積聯集為  $L * (y_b - y_a)$ ，然後再將每次移動時得到的面積加總，就是所有矩形的聯集面積了。

而當前當前線段的涵蓋情況，利用線段樹這樣維護：

- 當第  $i$  個矩形被假想線涵蓋到時，將區間  $[xa_i, xb_i]$  加 1
- 當第  $i$  個矩形不再被假想線涵蓋時，將區間  $[xa_i, xb_i]$  減 1

而當前線段們的長度聯集，直接查詢線段樹中有多少位置的值非 0 即可，實作上可以在線段樹維護兩個資訊：當前區間非 0 位置的數量及當前區間被多少線段涵蓋。而因為查詢線段樹時總是查詢全局的狀況（即我們只查詢根結點），所以是不用使用懶人標記（下放標記）的。

這種透過維護假想線掃描整個平面，將問題降一個維度的方法被稱作**掃描線**。

而本題座標的範圍為  $[0, 10^9]$ ，直接維護整棵線段樹顯然記憶體是不夠用的，這時可以使用在前面章節提到的離散化來處理這個問題；這裡再另外介紹一種解決方案：動態開點線段樹。

在每次修改時，其實最多只會修改  $\log 10^9$  個節點（即使要處理懶人標記時也是一樣），換句話說，其實是不用儲存整棵線段樹的，我們**只紀錄需要的節點**就好；使用指標來維護整棵線段樹，線段樹初始化為空，當拜訪線段樹時，如果當前拜訪到的節點不存在，就分配記憶體來記錄這個節點；如此一來，就可以在最多使用  $N \log 10^9$  個節點下維護線段樹了。

```
1 struct Node {
2     Node *lch, *rch;
3     int sum, cnt; // sum: 區間有多少非0點，cnt: 區間上有幾條完整線段
4     Node() {
5         lch = rch = nullptr;
6         sum = cnt = 0;
7     }
8     void modify(int l, int r, int ql, int qr, int type) {
9         if (this->lch == nullptr) this->lch = new Node();
10        if (this->rch == nullptr) this->rch = new Node();
11
12        if (l == ql && r == qr) {
13            if (type == 1) {
14                this->cnt++;
15                this->sum = r - l + 1;
16            } else {
17                this->cnt--;
18                if (this->cnt == 0)
19                    this->sum = this->lch->sum + this->rch->sum;
20            }
21        } else {
22            int mid = l + (r - l) / 2;
23            if (qr <= mid) {
24                this->lch->modify(l, mid, ql, qr, type);
25            } else if (ql > mid) {
26                this->rch->modify(mid + 1, r, ql, qr, type);
27            } else {
28                this->lch->modify(l, mid, ql, mid, type);
29                this->rch->modify(mid + 1, r, mid + 1, qr, type);
30            }
31        }
32    }
33}
```

```

31     if (this->cnt > 0) {
32         this->sum = r - l + 1;
33     } else {
34         this->sum = this->lch->sum + this->rch->sum;
35     }
36 }
37 }
38 };
39
40 struct Segment {
41     int l, r, y, type;
42 };
43 vector<Segment> v;
44 long long solve() {
45     sort(v.begin(), v.end(),
46           [] (Segment &a, Segment &b) { return a.y < b.y; });
47     long long ans = 0;
48     Node *root = new Node();
49     int pre_y = v.front().y;
50     for (auto now : v) {
51         if (now.y != pre_y) ans += (long long)root->sum * (now.y - pre_y);
52         pre_y = now.y;
53
54         root->modify(0, 1 << 30, now.l, now.r - 1, now.type);
55     }
56
57     return ans;
58 }
```

**程式碼 7.2:** 掃描線搭配動態開點線段樹

而除了上述介紹的掃描線搭配線段樹外，DP 章節提到的線段樹優化轉移、圖論（通常是樹）利用線段樹維護資訊等也是常見的與其他領域搭配的應用。

### 7.1.3 不直接用線段樹維護原序列

像是前面章節提到的逆序數對問題，我們其實都不是「直接」維護原序列，而是改從不同的視角去維護一些資訊（以逆序數對問題來說，我們可以使用值域線段樹維護當前各值出現次數）。以下就分享同樣也是不直接用線段樹維護原序列的應用。

氣球博覽會

TIOJ 1169(2015 TOI 選訓營一模)

給定長度為  $N(1 \leq N \leq 2 * 10^5)$  的序列  $a(0 \leq a_i < 2^{24})$ ，請  $Q(1 \leq Q \leq 2 * 10^5)$  次支援兩種操作：

1. 將序列某個位置的值更改為某值
2. 詢問某個區間內不含某數的最長子區間長度

似乎用線段樹直接維護原序列，無法輕易地維護不含某數的最長子區間長度。

現在考慮從別的角度來去解決本題。

一棵線段樹內紀錄多種數字，要找不存在某數的最長子區間長度看起來有點難，現在先假設數字只有 0 與 1 兩種，這時要查詢不存在 0 的最長子區間長度似乎就不這麼難了。可以將 0 設為負無限大，問題就轉為求最大連續和了。現在數字只有兩種的情況我們已經可以解決了，那讓我們回到原問題。

其實原本的問題也可以將數字看為兩種：詢問的某數一種和非詢問的某數一種，我們只要將所有詢問的某數分開來看就可以了。現在對所有數都維護一棵屬於它的線段樹，將線段樹中非該數的位置設為負無限大，是該數的位置設為 1，如此就可以將問題轉換為前面提到的只有 0 與 1 兩種數的問題了。

而對於修改操作 (假設將位置  $p$  的值從  $a$  修改為  $b$ ) 則就把  $a$  的線段樹中位置的  $p$  的值更改為負無限大， $b$  的線段樹中位置  $p$  的值更改為 1 即可。至此已經可以解決本題。

實作上，用動態開點線段樹對每個數都維護一棵線段樹。而若想省記憶體，可以透過離線的方法，每次只單獨看一種數，只處理跟這種數有關的修改 (可以把操作一拆成兩個修改操作) 與詢問即可。

| One Occurrence  | Codeforces 1000F |
|---|------------------|
| 給定長度為 $N(1 \leq N \leq 5*10^5)$ 的序列 $a(0 \leq a_i \leq 5*10^5)$ ，請 $Q(1 \leq Q \leq 5*10^5)$ 次支援詢問，詢問要求輸出任意數滿足其於某區間剛好出現一次，無解輸出 0。 |                  |

使用離線作法，先將所有詢問讀入後，依照其右邊界由小到大排序。

接著另外維護一個陣列  $pre$ ， $pre[i]$  紀錄位置  $i$  的值上一次出現的位置；而如果某個位置的值在目前範圍內的相同值的位置中非最大，將其  $pre$  設為無限大。接著逐個遍歷原序列，並更新  $pre$  陣列，在遍歷的同時，如果有詢問的右邊界與當前位置相同，假設詢問的區間為  $[l, r]$ ，令  $v = \min(pre[l], pre[l + 1] \dots pre[r])$ ，如果  $v < l$ ，就表示這筆詢問存在一個解在其區間剛好出現一次，我們將它紀錄下來，否則就是無解，使用線段樹維護  $pre$  陣列並優化查詢區間最小值。而在遍歷完原序列後，同時也求完所有詢問的解了。

這個做法正確性的證明是，每次  $pre$  都是維護當前各值最新一次出現的前一個位置，任意值前一次出現的位置若小於詢問的區間左邊界，就表示這個值在這個區間內恰好出現一次，而如果前一次出現的位置大於等於詢問的區間左邊界，就表示這個值在這個區間出現超過一次，那就去尋找每筆詢問區間中前一次出現位置最小

的好，如果連前一次出現位置最小的都在區間裡，那表示沒有一個值在這個區間剛好出現一次。而對於區間內非最新一次的值的  $pre$  值皆為無限大，無限大顯然是大於任意區間的左邊界。

#### 7.1.4 區間修改再談

| 區間加等差數列   | 經典問題 |
|---|------|
| 給定長度為 $N(1 \leq N \leq 10^6)$ 的序列 $a$ ，初始值皆為 0，請 $Q(1 \leq Q \leq 10^6)$ 次支援兩種操作： <ol style="list-style-type: none"><li>將序列區間 <math>[l, r]</math> 加上一個首項為 <math>a</math> 公差為 <math>d</math> 的等差數列 (位置 <math>l</math> 加上 <math>a</math>，位置 <math>l + 1</math> 加上 <math>a + d</math>，位置 <math>l + 2</math> 加上 <math>a + 2 * d</math>..... 位置 <math>r</math> 加上 <math>a + (r - l) * d</math>)</li><li>詢問某個位置的值</li></ol> |      |

看起來似乎無法輕易得打懶人標記來維護資訊。

還記得前一章提到的差分技巧嗎？把原序列差分後，這樣區間加等差數列就相當於普通的區間加值了！

具體來說，令  $b$  序列為  $a$  的差分序列 ( $b[1] = a[1], b[i] = a[i] - a[i - 1]$ )，區間  $[l, r]$  加上首項為  $a$  公差為  $d$  的等差數列就是將  $b$  序列位置  $l$  加上  $a$ ，位置  $[l + 1, r]$  加上  $d$ ，位置  $r + 1$  減掉  $a + (r - 1) * d$ ，然後只要用線段樹維護  $b$  序列就好；查詢某位置  $p$  的值，就是查詢  $b$  序列  $[1, p]$  的和。

| 模數   | 2017 臺南一中資訊學科能力競賽校內複選, TNFSH OJ 391 |
|--|-------------------------------------|
| 給定長度為 $N(1 \leq N \leq 2 * 10^5)$ 的序列 $a(1 \leq a_i \leq 10^8)$ ，請 $Q(1 \leq Q \leq 10^5)$ 次支援三種操作： <ol style="list-style-type: none"><li>將序列某個位置的值加上某值</li><li>將序列區間 <math>[l, r]</math> 的數都對 <math>v(1 \leq v \leq 10^9)</math> 取餘數</li><li>詢問某個區間的最大值</li></ol> |                                     |

看起來比較特別的就是操作二，區間取餘修改。考慮懶人標記，嘗試在一些節點紀錄資訊，並在之後往下拜訪時將標記下放；然而，若將一個節點進行取餘，其實是沒法得到修改後這個區間的最大值應為何的。

其實一個值如果被有效取餘後，新的值至多變為原來的一半。

換句話說，一個值最多經過  $\log K$  次的有效取餘後 ( $K$  是值域範圍)，就會變成 0；而總共有最多  $N + Q$  個值（如果把一次操作一看成會多一個位置的值），當操作

{3,14,7,5,**16**} maximum value is 16



After modulo 5

> {3,**4**,2,0,1} maximum value is 4



After modulo 10

> {3,4,**7**,5,6} maximum value is 7

圖 7.2: 區間取餘後，無法輕易預測新的最大值

|                        |                 |
|------------------------|-----------------|
| 15 modulo 14 = 1       | 15 modulo 7 = 1 |
| 15 modulo 13 = 2       | 15 modulo 6 = 3 |
| 15 modulo 12 = 3       | 15 modulo 5 = 0 |
| 15 modulo 11 = 4       | 15 modulo 4 = 3 |
| 15 modulo 10 = 5       | 15 modulo 3 = 0 |
| 15 modulo 9 = 6        | 15 modulo 2 = 1 |
| 15 modulo 8 = <b>7</b> | 15 modulo 1 = 0 |

圖 7.3: 觀察有效取餘後，數字的變化(以 15 為例)

二時，對每個被有效取餘的值都直接單點修改(如果取餘後值不會有變化我們就不修改了)。時間複雜度會是  $\mathcal{O}((N + Q) \log K \log (N + Q))$ ；而操作一與操作三都是基礎的線段樹操作。至此，已經可以在不錯的時間內解決本題。

```
1 int seg[maxn * 4];
2 void range_mod(int l, int r, int ql, int qr, int p, int mod) {
3     if (seg[p] < mod) { // 最重要的剪枝
4         return;
5     } else if (l == r) { // 更新到葉子為止
6         seg[p] %= mod;
7     } else {
8         int mid = (l + r) >> 1, lch = p << 1, rch = p << 1 | 1;
9         if (qr <= mid) {
10             range_mod(l, mid, ql, qr, lch, mod);
11         } else if (ql > mid) {
12             range_mod(mid + 1, r, ql, qr, rch, mod);
13         } else {
14             range_mod(l, mid, ql, mid, lch, mod);
15             range_mod(mid + 1, r, mid + 1, qr, rch, mod);
16         }
17         seg[p] = max(seg[lch], seg[rch]);
18     }
19 }
```

程式碼 7.3: 線段樹區間取餘操作實作

## 7.1.5 練習題 Exercise

|  |                 |
|--|-----------------|
| Pashmak and Parmida's problem  | Codeforces 459D |
| 給定長度為 $N(1 \leq N \leq 10^6)$ 的序列 $a$ 。詢問有多少組二元組 $(i, j)$ 滿足 $f(1, i, a_i) > f(j, N, a_j)$ 。 $f(l, r, v)$ 定義為序列 $a$ 在區間 $[l, r]$ 中有多少位置的值為 $v$ 。 |                 |

|   |                 |
|---|-----------------|
| The Bakery  | Codeforces 834D |
| 給定長度為 $N(1 \leq N \leq 35000)$ 的序列 $a$ 。要求將序列切成 $K(1 \leq K \leq \min(N, 50))$ 段。最大化所有段數字種類數量的總和。 |                 |

|  |                 |
|--|-----------------|
| Ant colony   | Codeforces 474F |
| 給定長度為 $N(1 \leq N \leq 10^5)$ 的序列 $a(1 \leq a_i \leq 10^9)$ 。請 $Q(1 \leq Q \leq 10^5)$ 次詢問一個區間 $[l, r]$ 中有多少位置的值不能整除區間 $[l, r]$ 中的至少一個值。 |                 |

|   |      |
|---|------|
| 區間最長連續遞增子序列   | 經典問題 |
| 給定長度為 $N(1 \leq N \leq 10^5)$ 的序列 $a$ 。請 $Q(1 \leq Q \leq 10^5)$ 次支援三種操作：<br>1. 修改某個位置的值<br>2. 將某個區間的所有值都開根號（無條件捨棄到整數）<br>3. 詢問某個區間中的最長連續遞增子序列的長度 |      |

## 7.2 樹堆 Treap

平衡樹是(高度)平衡的二元搜尋樹。在比賽中，通常STL裡的set與map就夠用了，但有一些操作是set與map無法支援的(例如查詢排名k小的元素)，這時就需要自己實作平衡樹了。

本章介紹的Treap是一種透過合併與分裂完成的平衡樹，相較於其他的平衡樹，Treap的code少、彈性佳，且可以用來處理一些線段樹無法完成的序列操作題。

### 7.2.1 Treap 入門

Treap是二元搜尋樹與二叉堆的合體(名字由來： $Treap = Tree + heap$ )。

先複習一下二元搜尋樹的性質(以下簡稱樹性質)與二叉堆的性質(以下簡稱堆性質)：

樹性質：

1. 任意節點的值(key)必大於等於其左子樹所有節點的值
2. 任意節點的值(key)必小於等於其右子樹所有節點的值

堆性質(最小堆)：

1. 任意節點的值必小於等於其子樹所有節點的值

顯然地，二元搜尋樹的子樹是二元搜尋樹，二叉堆的子樹也是二叉堆。而Treap的子樹也同樣是Treap。

在Treap中使用最大堆與最小堆在效能上並沒有差異，本節皆是以最小堆為例。

```
1 struct Treap {  
2     Treap *l, *r;  
3     int key, pri;  
4     Treap(int k) : l(nullptr), r(nullptr), key(k), pri(Random()) {}  
5 }; // Random 為亂數產生器，呼叫時回傳一個隨機數
```

程式碼 7.4: Treap 結構定義

Treap 中，key 用來維持樹性質，pri(priority 簡寫) 用來維持堆性質。

如果用 Treap 維護集合的話，key 表示元素的值。

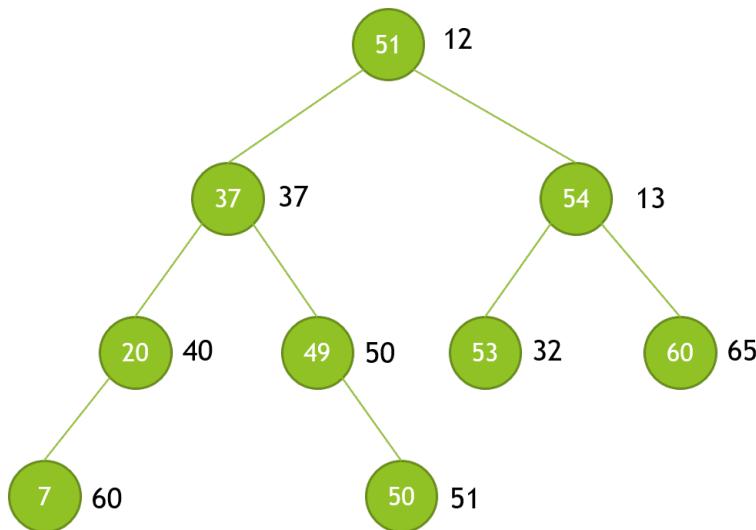


圖 7.4：一棵 Treap(節點內的值為 key，節點外的值為 pri)

特別的是，每個節點的 pri 是隨機給予的，這也是 Treap 平衡的關鍵，普通的二元搜尋樹可能會變成一條鏈，這會使得插入、刪除等操作的時間複雜度退化至  $O(N)$ ，也就是說最糟情況我們會需要遍歷整棵樹！

而 Treap 便是在維持樹性質的前提下，利用隨機產生的 pri 維持堆性質，使得整棵樹的高度期望是  $\log N$ ；關於樹的期望高度證明這裡省略，有興趣的同學可以在網路上或《Introduction to Algorithms》書中找到詳細證明。

綜合地說，樹性質讓我們得以維護資料 (的有序性)，堆性質則幫助我們平衡樹的高度。

接下來兩節介紹 Treap 中的兩個核心操作—merge(合併) 與 split(分裂) 操作，之後將介紹如何透過這兩個操作實作各種功能。

### 7.2.2 merge(Treap \*a, Treap \*b)

merge 操作用來將兩棵 Treap 合併成一棵，並回傳新 Treap 的根結點；而 merge 有個使用上的先決條件：其中一棵 Treap 中任意節點的 key 得小於另外一棵 Treap 中任意節點的 key。

本節和下節的說明與 code 皆將 key 比較小的 Treap 令為 a，另一棵令為 b(我們稱一棵 Treap 為 p，表示以 p 為根節點的 Treap，同學在閱讀時，需特別注意目前提到的 p，是指節點 p 還是以節點 p 為根的(子)樹)。

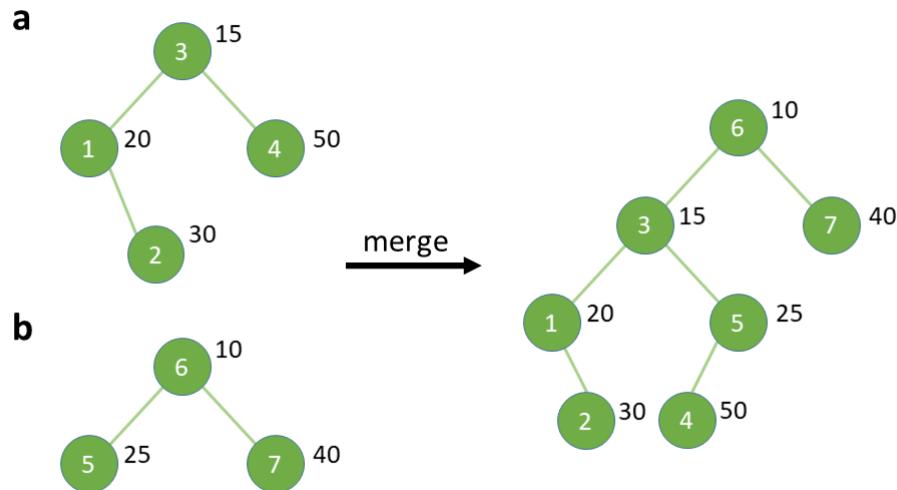


圖 7.5：兩棵 Treap 合併例子

merge 是透過遞迴進行的，現在根據不同情況來討論：

case1：a 或 b 有至少一棵 Treap 為空樹

如果有一棵為空，另外一棵不為空，那合併後的結果固然就是非空的那棵 Treap。

如果兩棵皆為空，那合併後的結果就是空樹 (空指標 nullptr)。

**這是遞迴終止條件。**

case2：a 和 b 皆不是空樹，且節點 a 的 pri 小於節點 b 的 pri

為了滿足堆性質，a 的根結點必然得是合併後的(子)樹一新樹的根結點，且為了滿足樹性質，a 的左子樹必然是新樹的左子樹(因為只有 a 的左子樹節點的 key 比節點 a 的 key 還要小)。現在新樹只剩它的右子樹尚未確定。

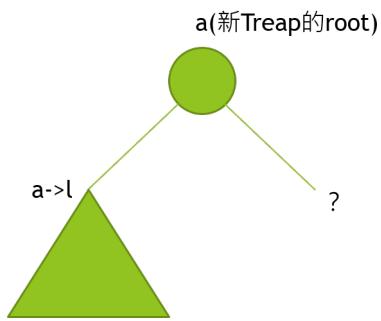


圖 7.6: 右子樹還不確定

而原本 a 的右子樹及 b 也還未被處理，而這兩棵 (子) 樹合併後的 Treap 就是現在新樹的右子樹。因此我們只要繼續遞迴呼叫  $\text{merge}(a->r, b)$  就可以得到新樹的右子樹了 (a 的右子樹的 key 皆小於 b 的 key，符合 merge 的先決條件！)。

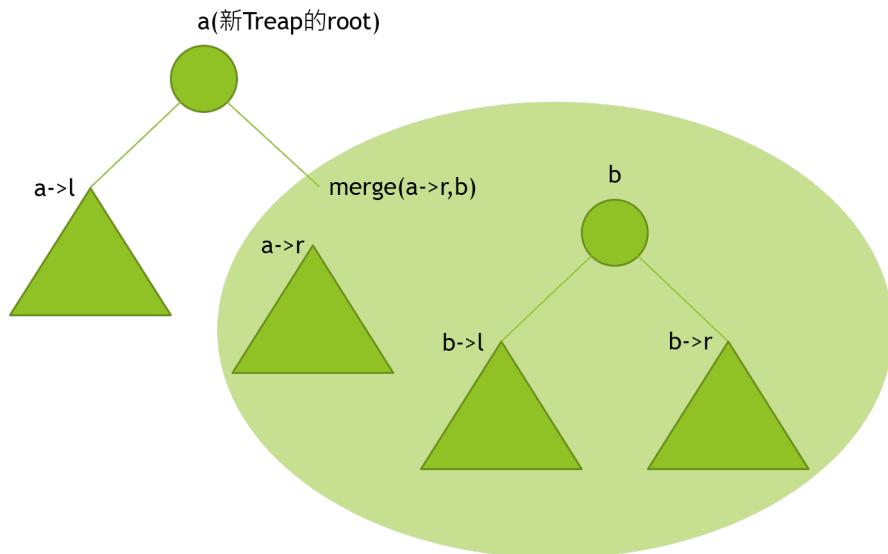


圖 7.7: 遞迴合併

case3 : a 和 b 皆不是空樹，且節點 a 的 pri 大於節點 b 的 pri

類似於第二種情況，只是改為由節點 b 作為新樹的根節點，b 的右子樹做為新樹的右子樹，新樹的左子樹則藉由呼叫遞迴  $\text{merge}(a, b->l)$  求得。

```

1 Treap *merge(Treap *a, Treap *b) {
2     if (a == nullptr) return b;
3     if (b == nullptr) return a;
4     if (a->pri < b->pri) {
5         a->r = merge(a->r, b);
6         return a;
7     } else {
8         b->l = merge(a, b->l);
9         return b;
10    }
11}

```

程式碼 7.5: 合併兩棵 Treap

而每次遞迴的深度增加一層時， $a$  或  $b$  的高度會減少一，一直到遞迴終止為止，因此最多遞迴「 $a$  的高度 +  $b$  的高度」次，且每層遞迴中，操作的複雜度是常數，因此  $\text{merge}$  操作的時間複雜度是  $\mathcal{O}(a + b)$ ；然後前面提過，Treap 的高度期望是  $\log N$ ，因此  $\text{merge}$  操作時間複雜度均攤就是  $\mathcal{O}(\log N)$ 。

### 7.2.3 split(Treap \*p,Treap \*&a,Treap \*&b,int k)

$\text{split}$  操作用來將一棵 Treap 分成兩棵 Treap；類似於  $\text{merge}$  操作，分裂出來的兩棵 Treap，根據給定的  $k$  值，其中一棵 Treap 中任意節點的 key 小於  $k$ ，另外一棵 Treap 中任意節點的 key 大於等於  $k$ 。

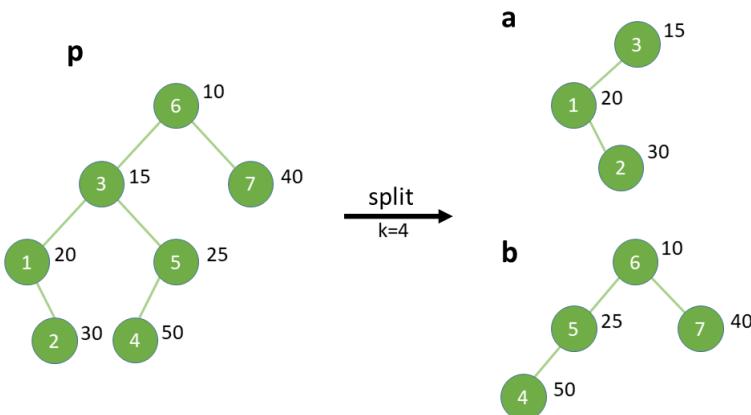


圖 7.8：一棵 Treap 分裂成兩棵 Treap 例子

$\text{split}$  也是遞迴進行的， $a$  與  $b$  的型別是 **reference to pointer**，結果會直接存在  $a, b$  裡。現在根據不同情況討論：

case1 : p 為空樹

顯然地，這時 a 和 b 都是空樹。這是遞迴終止條件。

case2 : p 不是空樹且 p 的 key 小於 k

因為分裂出來的結果，a 內所有節點的 key 必須小於 k，因此 p 節點與 p 的左子樹必然被分到 a 去；還剩 p 的右子樹不確定要怎麼分 (p 的右子樹中的 key 不確定與 k 的大小關係)。

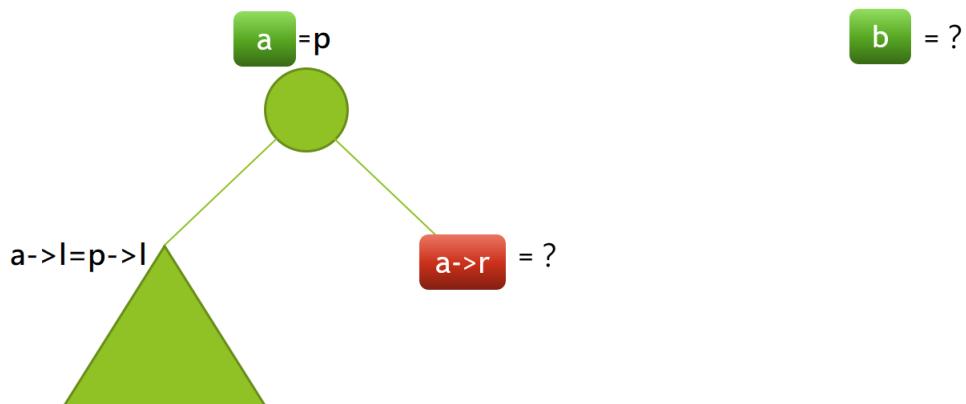


圖 7.9: a 的右子樹和 b 目前無法確定

而 a 的右子樹與 b 也還沒確定，只要繼續呼叫遞迴  $\text{split}(p->r, a->r, b, k)$  即可。

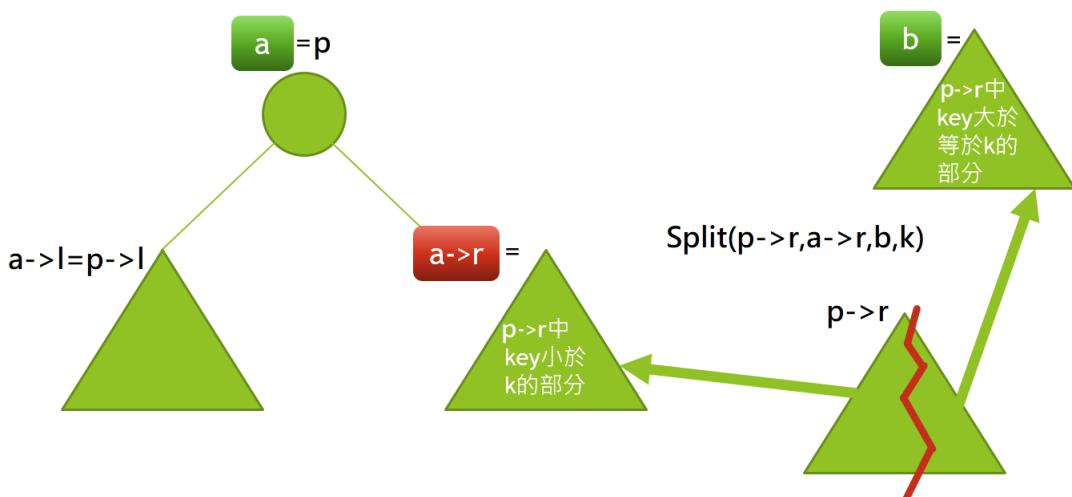


圖 7.10: a 的右子樹和 b 目前無法確定

case3 : p 不是空樹且 p 的 key 大於等於 k

類似於第二種情況，只是改為將 p 與 p 的右子樹分給 b，並繼續呼叫遞迴 split(p->l,a,b->l,k)。來確定如何將 p 的左子樹分給 a 與 b 的左子樹。

```
1 void split(Treap *p, Treap *&a, Treap *&b, int k) {
2     if (p == nullptr) {
3         a = b = nullptr;
4         return;
5     }
6     if (p->key < k) {
7         a = p;
8         split(p->r, a->r, b, k);
9     } else {
10        b = p;
11        split(p->l, a, b->l, k);
12    }
13 }
```

程式碼 7.6: 將一棵 Treap 分成兩棵

#### 7.2.4 二元搜尋樹操作

現在會了 merge 與 split 兩個操作，接著介紹如何利用這兩個操作，在  $\mathcal{O}(\log N)$  完成一般二元樹的插入與刪除操作。查找操作與二元搜尋樹相同，便不再多提。

插入一個 key 為 k 的節點到 Treap 裡，我們只需要將 Treap 以 k 為界分裂成 a,b 兩樹 (a 中所有節點的 key 小於 k，b 中所有節點的 key 大於等於 k)，並產生一個 key 為 k 的節點 (一個節點也是 Treap！)，先將 a 與它合併後，再跟 b 合併即可。

```
1 void insert(Treap *&root, int k) {
2     Treap *l, *r;
3     split(root, l, r, k);
4     root = merge(merge(l, new Treap(k)), r);
5 }
```

程式碼 7.7: 插入一個節點到 Treap

從 Treap 中刪除一個 key 為 k 的節點，只需要將 Treap 先以 k 為界分裂，再將右邊的樹以  $k+1$  為界分裂，最後再將最左與最右的樹合併即可。如果 key 為 k 的節點不只一個時，需要特別注意 (以下 code 為「刪除時必恰好有一個 key 為 k 的節點」的情況)。

```

1 void erase(Treap *&root, int k) {
2     Treap *a, *b, *c;
3     split(root, a, b, k);
4     split(b, b, c, k + 1);
5     delete b;
6     root = merge(a, c);
7 }
```

程式碼 7.8: 從 Treap 中刪除一個節點

插入與刪除操作都是呼叫 merge 與 split 來完成，均攤時間複雜度皆為  $\mathcal{O}(\log N)$ 。

### 7.2.5 名次樹操作

只有 insert 和 erase 功能就和 STL 的 map 和 set 差不多，但對 Treap 做一些更改之後，把 Treap 想成是一個集合，就能在平均  $\mathcal{O}(\log n)$  時間完成以下兩種操作：

1. 查詢某個元素在集合中的排名，也就是比它小的元素有幾個 (rank)
2. 查詢集合中第  $k$  小元素 ( $k$ -th)

其中 rank 看起來是最簡單的，如果只要查詢小於某個元素  $k$  的元素有幾個的話，只需要用 split 把小於  $k$  的部分切出來，然後數數看有幾個點就可以了，但是這樣複雜度會退化成  $\mathcal{O}(N)$ ！因此要在每個節點加上一個 size 域，紀錄該點所代表的子樹總共有幾個節點，這樣算 rank 的時候就不用直接去數總共有幾個點了。

這種會記錄 size 的二元搜尋樹稱為名次樹，不一定專指 Treap，大部分的二元搜尋樹好好維護 size 的話就可以獲得名次樹的功能。圖7.11就是一棵名次樹。

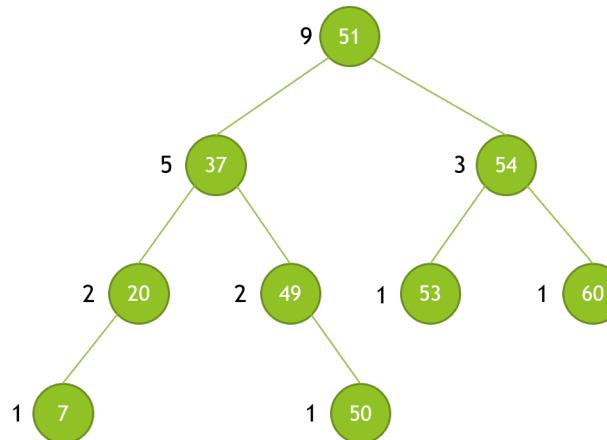


圖 7.11：名次樹

在 Treap 節點的結構中，我們增加了 size 以及向上更新 (pull) 的操作。

```
1 struct Treap {
2     Treap *l, *r;
3     int key, pri, size;
4     Treap(int k)
5         : l(nullptr), r(nullptr), key(k), pri(Random()), size(1) {}
6     void pull();
7 }
```

程式碼 7.9: 結構中增加 size 以及向上更新的操作

其中向上更新的操作用來修改當前節點的 size，而如果左右子樹的 size 都已經算出來了，那麼現在這個點的 size 就可以透過以下的方法算出來：

```
1 void Treap::pull() {
2     size = 1;
3     if (l != nullptr) size += l->size;
4     if (r != nullptr) size += r->size;
5 }
```

程式碼 7.10: 向上更新

只要在適當的位置加上 pull 函式就可以正確維護 size 了。以 merge 來說，要保證 merge 之後的 size 是正確的，因此可以把 pull 加在 return 之前，也就是像這樣：

```
1 Treap *merge(Treap *a, Treap *b) {
2     if (a == nullptr) return b;
3     if (b == nullptr) return a;
4     if (a->pri < b->pri) {
5         a->r = merge(a->r, b);
6         a->pull(); // 加在這裡
7         return a;
8     } else {
9         b->l = merge(a, b->l);
10        b->pull(); // 加在這裡
11        return b;
12    }
13 }
```

程式碼 7.11: merge 增加向上更新函數

split 要保證分裂後的 size 都是正確的，分裂完後，重新計算當前新樹的 size。

```
1 void split(Treap *p, Treap *&a, Treap *&b, int k) {
2     if (p == nullptr) {
3         a = b = nullptr;
4         return;
5     }
6     if (p->key < k) {
7         a = p;
8         split(p->r, a->r, b, k);
9         a->pull(); // 重新計算size
10    } else {
11        b = p;
12        split(p->l, a, b->l, k);
13        b->pull(); // 重新計算size
14    }
15 }
```

程式碼 7.12: split 增加向上更新函數

有些時候容易對空的節點 (nullptr) 查詢 size，如果漏判斷容易產生問題，因此寫一個函式來進行判斷是最好的方法。

```
1 inline int size(Treap *p) {
2     return p == nullptr ? 0 : p->size;
3 }
```

程式碼 7.13: 查詢 size 的函式

這樣就可以在算 rank 的時候不必逐個去數節點數量了。

```
1 inline int rank(Treap *&root, int key) {
2     Treap *a, *b;
3     split(root, a, b, key);
4     int ret = size(a);
5     root = merge(a, b);
6     return ret;
7 }
```

程式碼 7.14: rank 函式實作

剩下的查詢第  $k$  小元素比較麻煩，剛剛的所有工具貌似不能直接解決這個問題，如果 split 也能把排名前  $k$  個的點切出來該有多好啊！

這件事是做得到的，我們可以寫一個新的 split2。

- $\text{split2}(\text{Treap } *p, \text{Treap } *&a, \text{Treap } *&b, \text{int } k)$ ：將 Treap  $p$  根據  $k$  分成兩棵 Treap  $a, b$ ， $k$  滿足  $0 \leq k \leq \text{size}(p)$ ，將前  $k$  個節點分給  $a$ ，剩下的  $\text{size}(p) - k$  個節點分給  $b$

和原本的 split 一樣可以分成三個 case。

case1 : p 為空樹

和剛剛一樣 a 和 b 都會是空樹。這是遞迴終止條件。

case2 :  $\text{size}(p \rightarrow l) < k$

這代表 p 和 p 的左子樹中所有點都在前 k 個節點的範圍，因此要把它們分到 a，其中 p 也會變成 a 的根，p 的左子樹也確定會是 a 的左子樹。

現在 a 還要從剩下的  $p \rightarrow r$  中切出  $k - \text{size}(p \rightarrow l) - 1$  個節點出來當作它的右子樹，因此遞迴呼叫  $\text{split2}(p \rightarrow r, a \rightarrow r, b, k - \text{size}(p \rightarrow l) - 1)$  即可。

case3 :  $\text{size}(p \rightarrow l) \geq k$

這代表 p 和 p 的右子樹不在前 k 個的範圍內，因此要把它們分到 b 去，其中 b 的根會是 p，p 的右子樹會是 b 的右子樹。把剩下的  $p \rightarrow l$  中切出 k 個節點分到 a，剩下的東西分到 b 的左子樹中，就完成了。這個步驟可以透過遞迴呼叫  $\text{split2}(p \rightarrow l, a, b \rightarrow l, k)$  來達成。

```
1 void split2(Treap *p, Treap *&a, Treap *&b, int k) {
2     if (p == nullptr) {
3         a = b = nullptr;
4         return;
5     }
6     if (size(p->l) < k) {
7         a = p;
8         split2(p->r, a->r, b, k - size(p->l) - 1);
9         a->pull();
10    } else {
11        b = p;
12        split2(p->l, a, b->l, k);
13        b->pull();
14    }
15 }
```

程式碼 7.15: split2

有了 split2，那查詢第 k 小的元素就可以用類似 rank 的方式處理。

```

1 inline int kth(Treap *&root, int k) {
2     Treap *a, *b, *c;
3     split2(root, a, c, k); // 把前k個切到a
4     split2(a, a, b, k - 1); // 把前k-1個切到a · b就會是第k個元素
5     int ret = b->key;
6     root = merge(merge(a, b), c); // 別忘記切完要合併
7     return ret;
8 }
```

**程式碼 7.16:** 查詢排名第 k 小的元素

### 7.2.6 Treap 處理序列操作題

而其實 Treap 還能做到很多線段樹無法做到的序列操作！

想知道 Treap 如何處理區間操作，得先思考要怎麼把 Treap 中的點對應到序列中。最簡單的想法就是把樹壓平，它就變成一個序列了！想像每個點裡面多存一個 val 值，按照**中序**把 val 值一個一個抄下來就可以得到一個序列 S，假設序列由 1 開始編號，其中  $S[i]$  就會是對樹堆查詢  $kth(i)$  所得到的值。

原本的 split 因為紀錄的資訊非 key 值就無法使用，以下所有用到的 split 皆是指 split2。

merge 可以看成是將兩段序列接起來，split 就會是將一條序列從一個位置切成左右兩條序列，其中一條（左邊）的長度為 k，現在只要在加上一些懶惰標記的維護就可以向線段樹一樣做區間處理了！

| 區間反轉、區間查詢最大值                                     | POJ 3580 子題 |
|--|-------------|
| 給一個長度為 $N (1 \leq N \leq 10^5)$ 的序列，你可以對他進行兩個操作： |             |
| 1. 將某個區間反轉 2. 查詢序列某個區間的最大值                       |             |

剛剛已經說過，可以用 Treap 來記錄區間的資訊。節點的結構要紀錄最大值 max\_val 和懶人標記 reverse\_tag。

```

1 struct Treap {
2     Treap *l, *r;
3     int val, max_val, pri, size;
4     bool reverse_tag;
5     Treap(int v)
6         : l(nullptr),
7             r(nullptr),
8             val(v),
9             max_val(v),
10            pri(Random()),
11            size(1),
12            reverse_tag(false) {}
13     void rev();
14     void push();
15     void pull();
16 };

```

程式碼 7.17: 節點的結構

其中有兩個函數 push 和 pull，push 是用來將懶人標記往下放用的，如果該點有懶人標記的話，就把該點左右子樹交換，並把懶惰標記的資訊加到左右子樹中，之後將懶人標記清空。

pull 前面講到是用來維護 size，而現在除了 size 之外還要維護最大值，但是左右子樹可能還有懶人標記的存在，為了不影響答案所以在蒐集左右子樹的資訊之前通常會將左右子樹的懶人標記往下推（但其實以範例題來說是不需要將左右子樹的標記往下放的，想一想就明白了）。

```

1 void Treap::rev() { // 反轉當前節點並更新標記
2     swap(l, r);
3     reverse_tag = !reverse_Tag;
4 }
5 void Treap::push() {
6     if (reverse_tag) {
7         if (l != nullptr) l->rev();
8         if (r != nullptr) r->rev();
9         reverse_tag = false;
10    }
11 }
12 void Treap::pull() {
13     max_val = val;
14     size = 1;
15     if (l != nullptr) {
16         max_val = max(max_val, l->max_val);
17         size += l->size;
18     }
19     if (r != nullptr) {
20         max_val = max(max_val, r->max_val);
21         size += r->size;

```

```
22     }
23 }
```

程式碼 7.18: 懶惰標記下放和最大值的維護

我們剛剛已經將 pull 函數放在正確的位置了，那 push 正確的位置在哪裡呢？因為 push 會把節點的資訊分送給左右子樹，因此要在節點的結構還沒被修改前執行，也就是遞迴前。

merge 和 split 只要在遞迴前執行 push 就可以了。

```
1 Treap *merge(Treap *a, Treap *b) {
2     if (a == nullptr) return b;
3     if (b == nullptr) return a;
4     if (a->pri < b->pri) {
5         a->push();
6         a->r = merge(a->r, b);
7         a->pull();
8         return a;
9     } else {
10        b->push();
11        b->l = merge(a, b->l);
12        b->pull();
13        return b;
14    }
15 }
```

程式碼 7.19: merge 插入 push 的位置

```
1 void split(Treap *p, Treap *&a, Treap *&b, int k) {
2     if (p == nullptr) {
3         a = b = nullptr;
4         return;
5     }
6     p->push();
7     if (size(p->l) < k) {
8         a = p;
9         split(p->r, a->r, b, k - size(p->l) - 1);
10        a->pull();
11    } else {
12        b = p;
13        split(p->l, a, b->l, k);
14        b->pull();
15    }
16 }
```

程式碼 7.20: split 插入 pull 的位置

這樣一些操作就變得很簡單了，例如要把  $[l, r]$  這個區間反轉，只要把該區間切出來，打上反轉的標記然後拼回去就好了。

```

1 void reverse(Treap *&root, int l, int r) {
2     Treap *a, *b, *c;
3     split(root, b, c, r); // b : [1, r] , c : [r+1, n]
4     split(b, a, b, l - 1); // a : [1, l-1] , b : [l, r]
5     b->rev();
6     root = merge(merge(a, b), c);
7 }
```

程式碼 7.21: 區間反轉

查詢也是把要查詢的區間  $[l, r]$  切下來就可以了。

```

1 int query(Treap *&root, int l, int r) {
2     Treap *a, *b, *c;
3     split(root, b, c, r); // b : [1, r] , c : [r+1, n]
4     split(b, a, b, l - 1); // a : [1, l-1] , b : [l, r]
5     int ret = b->max_val;
6     root = merge(merge(a, b), c);
7     return ret;
8 }
```

程式碼 7.22: 區間查詢最大值

### 7.2.7 總結

如果給我們一個序列，想用該序列構造出 Treap，可以從一棵空樹開始一個一個把值插入進去，複雜度  $\mathcal{O}(N \log N)$ 。

Treap 功能強大，能做到很多線段樹不能做到的事。

Treap 常數很大，如果可以用線段樹解的題目盡量用線段樹解（特別的是，有些看起來 Treap 可以直接做的題目其實是有辦法  $\mathcal{O}(N)$  去解）。

### 7.2.8 練習題 Exercise

|   |          |
|---|----------|
| Graph and Queries   | UVA 1479 |
| 給你一張無向圖，有三種操作，查詢跟點 u 連接的所有點中，第 K 大的值、修改某個點的值、刪掉某條邊。最後輸出所有查詢的平均值 |          |

**SuperMemo**

POJ 3580

給你一個長度為  $N$  的序列  $S$ ，接下來有  $Q(Q \leq 10^5)$  個操作：

1. 將某個區間加上一個數  $d$
2. 將某個區間反轉
3. 將某個區間循環旋位移  $d$  單位，也就是把原本的  $S_l, S_{l+1}, \dots, S_r$  變成  $S_{r-d+1}, \dots, S_r, S_l, \dots, S_{r-d}$
4. 在  $S_x$  的後面插入  $d$
5. 刪除  $S_x$
6. 詢問  $\min\{S_l, S_{l+1}, \dots, S_r\}$

**Time to read**

Codeforces 100186B

有一個人想讀一本有  $N(N \leq 3 \times 10^5)$  頁的書，他每次會從書中挑連續的幾頁出來，共挑  $Q(Q \leq 3 \times 10^5)$  次。對於第  $i$  次挑出來的頁  $[l, r]$ ，他會從中讀那些他沒讀過的頁，以及和  $l$  頁有相同標籤的頁，並把這些他要讀的頁貼上編號  $i$  的標籤（有舊標籤會撕掉）。他讀一頁花一秒，請問他共花了幾秒

**非誠勿擾**

HDU 4557

給你一個盒子，一開始是空的，有兩個操作

1. 放入一顆寫有號碼的球，這顆球是有名字的
  2. 紿一個數字  $k$ ，從盒子中找號碼大於等於  $k$  且為號碼為最小的球，如果有多顆這樣的球就找放入盒子時間最早的，把它從盒中移除並輸出它的名字
- 這題雖然可以用其他方法做，但是可以用 Treap 來練習看看，了解一些平衡樹的基本操作是一件好事

**Key insertion**

POJ 2131

有無限個箱子，一個箱子裡最多只能塞一個元素。現在有  $N$  次放入操作  $(p, v)$ 。放入操作  $(p, v)$  定義為將元素  $v$  放入箱子  $p$ ，而如果箱子  $p$  已經有元素了，則將箱子  $p$  內的元素  $v'$  拿出來，並遞迴執行放入操作  $(p + 1, v')$ ，接著將元素  $v$  放入箱子  $p$  內。最後輸出所有元素所在的箱子編號。

這題的解法也是很多，同學可以試著用 Treap 解決這題

## 7.3 持久化資料結構

持久化資料結構就是在修改之後保留之前所有版本的資料結構。最樸素的實作方法就是在修改後就把資料結構整個複製一份，但是在資料量很大時會花費大量的時間及記憶體，因此本節旨在如何用盡量少的時間和記憶體保存之前版本。

### 7.3.1 持久化線段樹

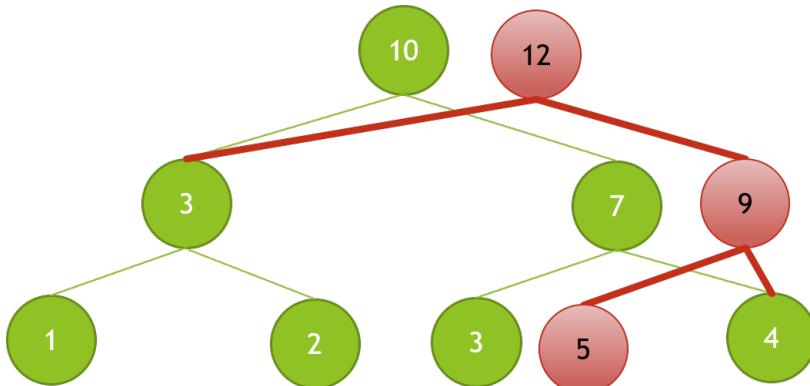
持久化的概念可應用在很多資料結構上，其中線段樹是最簡單的。以單點修改求區間總和的線段樹為例（要進行複製操作故得改用指標而不能用陣列實作）。

```
1 struct Node {
2     int data;
3     Node *lch, *rch;
4     Node(int data) : data(data), lch(nullptr), rch(nullptr) {}
5     void pull() {
6         data = 0;
7         if (lch != nullptr) data += lch->data;
8         if (rch != nullptr) data += rch->data;
9     }
10 };
```

程式碼 7.23: 節點結構

想要在做第  $T$  次修改後依然保留前  $T-1$  次的版本，最簡單的方式就是偷懶。例如這樣偷懶：們在每次新創一個版本時直接遞迴進行複製整顆樹。但其實會被修改到的只有路徑上那  $\mathcal{O}(\log n)$  個點！換言之，絕大部分節點其實是不變的，兩個版本之間可以共用這些節點，而會被改到的節點只要在被改之前複製出來即可。

```
1 void modify(int l, int r, int pos, Node *pre, Node *now, int data) {
2     // pre:前一個版本的節點
3     // now:當前版本的節點(要先分配記憶體)
4     if (l == r) {
5         now->data = data;
6     } else {
7         now->lch = pre->lch; // 指向前一個版本的子節點
8         now->rch = pre->rch; // 指向前一個版本的子節點
9         int mid = (l + r) / 2;
10        if (pos <= mid) {
11            now->lch = new Node(0); // 分配記憶體給當前版本的新子節點
12            modify(l, mid, pos, pre->lch, now->lch, data);
13        } else {
14            now->rch = new Node(0); // 分配記憶體給當前版本的新子節點
15            modify(mid + 1, r, pos, pre->rch, now->rch, data);
16        }
17        now->pull();
18    }
```

**程式碼 7.24:** 持久化的單點修改操作**圖 7.12:** 將序列位置 3 的值從從 3 改成 5

每一次的持久化就會產生約  $\mathcal{O}(\log n)$  的節點被複製出來，但還在可以忍受的範圍，因此就放心地複製吧！

可以發現以線段樹來說複製之後新樹和原本的樹結構上會長的一模一樣，這個性質可以讓持久化線段樹發揮更大的作用。

| 區間第 K 小   | HDU 2665 改 |
|---|------------|
| 給你一個序列 $S$ ，序列中數字範圍是 $1 \sim 10^6$ ，接著會對著個序列做多次的詢問：<br>給定 $l, r, k$ 三個數，輸出區間 $S[l] \sim S[r]$ 中，排名第 $k$ 小的數字是什麼 |            |

舉例來說  $S = \{1, 2, 3, 7, 1, 2, 2\}$ ，則詢問  $4 \sim 7$  的結果為 2，把  $S[4] \sim S[7]$  的元素拿出來排序之後的結果是  $\{1, 2, 2, 7\}$ ，排名第 3 的元素就是 2。

可以把  $S[1] \sim S[n]$  的數字離散化讓到  $[0, n-1]$  的範圍中，因此數字範圍其實沒有很重要。這節主軸是線段樹，那要怎麼用線段樹來解決這個問題呢？

先做離散化:  $\{1, 2, 3, 7, 1, 2, 2\} \rightarrow \{0, 1, 2, 3, 0, 1, 1\}$

先處理簡單的問題吧，如果不是要問區間而是要問整個序列的第  $k$  小要怎麼做，同學們一定想到了可以用剛剛教過的 Treap，其實用線段樹也是可以做的。

用求區間和的線段樹來記錄某個數字出現的次數，也就是用值域線段樹維護各值出現次數，查詢的時候根據左子樹所代表的區間中有多少個數字，來決定要往左還是往右遞迴(我們相當於是在線段樹上二分搜!)。

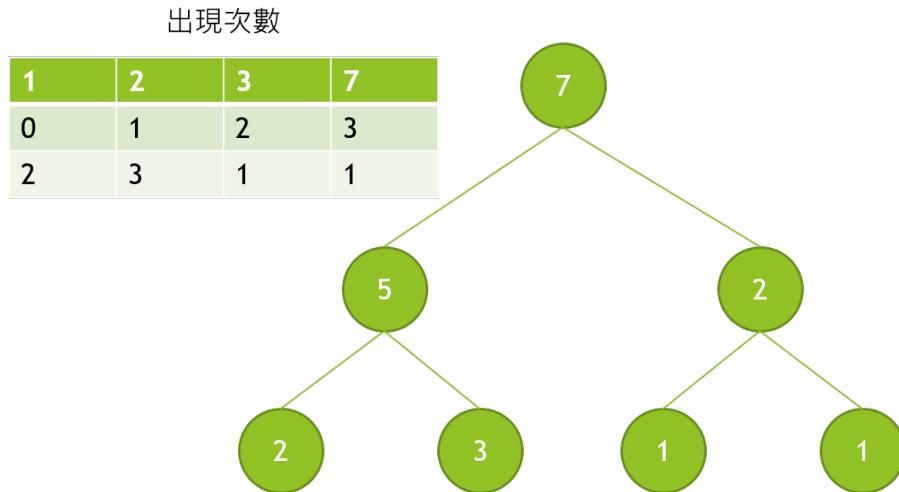


圖 7.13: 區間和的線段樹

```

1 int find(int l, int r, Node *p, int k) {
2     if (l == r) return l;
3     int mid = (l + r) / 2;
4     int l_size = p->lch->data;
5     if (k <= l_size) {
6         return find(l, mid, p->lch, k);
7     } else {
8         return find(mid + 1, r, p->rch, k - l_size);
9     }
10 }
```

程式碼 7.25: 查詢整個序列第  $k$  小

這樣的話就可以知道第  $k$  小的數是多少了。但是很不幸地，我們要處理的問題是區間問題，有了持久化線段樹可以依序將  $S[1], S[2] \dots S[n]$  一個一個插入線段樹中，保留歷史的版本  $T[i] = S[1], S[2] \dots S[i]$  被插入之後的樹，這樣查詢  $T[r]$  中第  $k$  小，就是在查詢  $S[1]$  到  $S[r]$  中第  $k$  小的值。

但是查詢的左邊界不一定都是 1，然而可以把查詢區間  $[l, r]$  想成是先把  $S[1], S[2] \dots S[r]$  插入線段樹中，再把  $S[1], S[2] \dots S[l-1]$  的部分刪除。插入的部分其實就是  $T[r]$ ，刪除的部分其實就是  $T[l-1]$ ，而且如同前面提到的了，持久化出來之後線段樹的結構是一樣的，因此只要將這兩棵線段樹相減就可以了，主程式碼如下：

```

1 void build(int l, int r, Node *p) {
2     if (l == r) return;
3     int mid = (l + r) / 2;
4     p->lch = new Node(0);
5     build(l, mid, p->lch);
6     p->rch = new Node(0);
7     build(mid + 1, r, p->rch);
8 }
9 vector<int> discretize(int n, int arr[]) {
10    vector<int> dct;
11    for (int i = 0; i < n; i++) {
12        dct.push_back(arr[i]);
13    }
14    sort(dct.begin(), dct.end());
15    dct.resize(unique(dct.begin(), dct.end()) - dct.begin());
16    for (int i = 0; i < n; i++) {
17        int &x = arr[i];
18        x = lower_bound(dct.begin(), dct.end(), x) - dct.begin();
19    }
20    return dct;
21 }
22 const int maxn = 1000005;
23 int arr[maxn];
24 Node *T[maxn]; // 記錄各個版本的樹根
25 int main() {
26     int N, Q;
27     cin >> N >> Q;
28     for (int i = 1; i <= N; i++) cin >> arr[i];
29     vector<int> dct = descretize(N, arr + 1); // 離散化
30
31     // 先建一棵空樹(各值出現次數為0)
32     T[0] = build(0, (int)dct.size() - 1);
33     for (int i = 1; i <= N; i++) {
34         // 對維護各版本
35         // 新版本的葉節點記得繼承前一個版本葉節點的資訊
36         T[i] = new Node(0);
37         modify(0, (int)dct.size() - 1, T[i - 1], T[i], arr[i]);
38     }
39     while (Q--) {
40         int l, r, k;
41         cin >> l >> r >> k;
42         cout << dct[find(0, (int)dct.size() - 1, T[l - 1], T[r], k)]
43             << '\n';
44     } // 這裡find藉由「版本r資訊-版本l-1資訊」來得到區間[l,r]的正確資訊
45     return 0;
46 }
```

**程式碼 7.26:** 查詢區間第  $k$  小其餘程式碼

總共做了  $n$  次持久化的操作，每次操作的時間為  $\mathcal{O}(\log n)$ ，總共會產生  $\mathcal{O}(n \log n)$  個點出來，查詢和一般線段樹一樣為  $\mathcal{O}(\log n)$ 。

### 7.3.2 持久化 Treap

如果覺得 Treap 的功能就只有剛剛那樣那還真是小看 Treap 了。和剛剛線段樹一樣，直接複製路徑上的節點就可以簡單地做到持久化。這裡會遇到一個問題，Treap 原本紀錄一個 pri 值用來保持平衡，複製之後就會導致有一堆相同 pri 的點跑出來，這樣的話很容易就會有  $\mathcal{O}(n)$  的深度。

這裡將不會再紀錄 pri 值，原本兩棵樹 a,b 合併的時候是看誰的 pri 值比較小，因為 pri 值是隨機產生的，所以  $a->\text{pri} < b->\text{pri}$  的機率就會是  $\frac{\text{size}(a)}{\text{size}(a)+\text{size}(b)}$ ，我們就用這個機率來決定誰要當根就可以完美的維護複雜度了。注意直接用 size 比較大的點當根，或是用  $\frac{1}{2}$  的機率都可以構造測資讓它們爛掉喔！

這個方法不管在寫一般的 Treap 或是持久化的版本都有不錯的效果，而且要從陣列直接構造樹的話因為沒有 pri 的關係可以直接用類似線段樹的方式是在  $\mathcal{O}(n)$  完成，這東西正式的名稱叫 Randomized binary search tree(隨機二叉樹)，缺點是 merge 遞迴時每層都要呼叫一次隨機數產生器。

現在只要在 merge 和 split 適當的位置加上複製節點的功能就可以了。

```
1 Treap *merge(Treap *a, Treap *b) {
2     if (a == nullptr) return b;
3     if (b == nullptr) return a;
4     //不用pri，直接算當根的機率
5     if (Random() % (a->size + b->size) < a->size) {
6         a->push();
7         a = new Treap(*a);
8         a->r = merge(a->r, b);
9         a->pull();
10        return a;
11    } else {
12        b->push();
13        b = new Treap(*b);
14        b->l = merge(a, b->l);
15        b->pull();
16        return b;
17    }
18}
19 void split(Treap *p, Treap *&a, Treap *&b, int k) {
20     if (p == nullptr) {
21         a = b = nullptr;
22         return;
23     }
24     p->push();
```

```

25     p = new Node(*p);
26     if (size(p->l) < k) {
27         a = p;
28         split(p->r, a->r, b, k - size(p->l) - 1);
29         a->pull();
30     } else {
31         b = p;
32         split(p->l, a, b->l, k);
33         b->pull();
34     }
35 }

```

**程式碼 7.27:** 在適當的地方複製節點出來

透過在 merge 的時候每次都計算機率維護了 Treap 該有的深度，因此操作的複雜度還會是  $\mathcal{O}(\log N)$ ，持久化的時候也只會複製路徑上的點，故每次操作平均下來只複製平均  $\mathcal{O}(\log N)$  個點。

持久化 Treap 的能力不只能記錄之前的版本，還可以完成像是  $a=merge(a,a)$  這種神奇的操作呢！

### 7.3.3 練習題 Exercise

#### 蚯蚓(扭)

NPSC2014 高中組決賽 pD

給你長度為  $n(n \leq 10^4)$  的字串，接著有  $q(q \leq 10^4)$  個操作：

1. 詢問位於  $[l, r]$  的子字串為何
2. 將位於  $[l, r]$  的子字串原地複製一份
3. 反轉位於  $[l, r]$  的子字串

保證輸出不會超過 5MB，任意時刻字串長度不會超過  $10^9$

#### Dynamic Rankings

ZOJ 2112

給你長度為  $N(N \leq 50000)$  的序列，接著有  $Q(Q \leq 10^4)$  個操作：

1. 詢問區間  $[l, r]$  中第  $k$  小的數是多少
2. 修改序列中的一個數字

#### D-query

SPOJ DQUERY

給你長度為  $N(N \leq 30000)$  的序列，接著有  $Q(Q \leq 2 \times 10^5)$  個詢問  $[l, r]$ ，請你回答該區間內有多少種相異的數字

給你長度為  $N(N \leq 10^6)$  的序列  $S$  · 和一個數字  $M$  · 接著有  $Q$ (範圍未知) 個操作  $l\ r\ k$  · 表示將  $[l, r]$  的子序列複製一份接到  $S[k]$  之後。

每次操作完如果序列長度大於  $M$  的話就只保留  $S_1 \sim S_M$  的元素剩下的刪掉。

注意這題因 HOJ 壞掉所以目前不能傳 · 且一定要引用計數 (smart pointer) 不然會 MLE

## 7.4 二維線段樹

一般的線段樹對各位來說應該不陌生吧？二維線段樹的概念和一般的一維線段樹的概念是極為相似的，只要能夠理解，就算要寫高維線段樹也不是甚麼大問題。

### 7.4.1 一維線段樹複習

簡單來說每個二維線段樹的節點都是一個一維線段樹，因此筆者個人習慣會用 struct 把一維線段樹包裝起來使用。

以下的線段樹會以單點加值區間查詢總和作為範例。

```
1 struct Seg1D {  
2     static const int MAXN = 500;  
3     int st[MAXN * 4];  
4     void pull(int d);  
5     void build(int l, int r, int d);  
6     void insert(int x, int data, int l, int r, int d);  
7     int sum(int a, int b, int l, int r, int d);  
8 };
```

程式碼 7.28: 一維線段樹結構封裝

一般來說除非是動態開點的線段樹，我們是不會使用指標來維護的。

pull 這個函數，用來進行向上更新，也就是用左右子樹的答案更新現在這個節點的答案。

```
1 void Seg1D::pull(int d) {  
2     st[d] = st[d * 2] + st[d * 2 + 1];  
3 }
```

程式碼 7.29: 一維線段樹資訊向上更新

build 函數是用來清空或是用陣列構造一維線段樹，這裡預設是將整棵樹清空設成 0，相信聰明的讀者應該知道怎麼修改可以讓它變成用陣列構造線段樹吧？

```
1 void Seg1D::build(int l, int r, int d = 1) {  
2     if (l == r) {  
3         st[d] = 0;  
4         return;  
5     }  
6     int mid = (l + r) / 2;  
7     build(l, mid, d * 2);  
8     build(mid + 1, r, d * 2 + 1);
```

```
9     pull(d);  
10 }
```

程式碼 7.30: 構造一維線段樹並將其清空為 0

insert 函數將位置  $x$  的值加上  $data$ ，對常寫線段樹的人來說應該是非常簡單的，我們主軸是二維線段樹，因此就不進行太多的說明。

```
1 void Seg1D::insert(int x, int data, int l, int r, int d = 1) {  
2     if (r < x || x < l) return;  
3     if (l == x && x == r) {  
4         st[d] += data;  
5         return;  
6     }  
7     int mid = (l + r) / 2;  
8     insert(x, data, l, mid, d * 2);  
9     insert(x, data, mid + 1, r, d * 2 + 1);  
10    pull(d);  
11 }
```

程式碼 7.31: 單點加值

sum 函數，用來計算區間的總和，一樣不進行詳細介紹。

```
1 int Seg1D::sum(int a, int b, int l, int r, int d = 1) {  
2     if (r < a || b < l) return 0;  
3     if (a <= l && r <= b) return st[d];  
4     int mid = (l + r) / 2;  
5     int sl = sum(a, b, l, mid, d * 2);  
6     int sr = sum(a, b, mid + 1, r, d * 2 + 1);  
7     return sl + sr;  
8 }
```

程式碼 7.32: 計算區間和

可以發現 build 的複雜度是  $\mathcal{O}(n)$ ，因為用陣列實作的關係，有機會用到  $4n$  的空間。insert 和 sum 都花費  $\mathcal{O}(\log n)$  的時間。

這樣我們一維線段樹基本的操作都已經預習了一遍，現在我們可以進入二維的版本了。

## 7.4.2 二維版本

一維線段樹是用來維護一維陣列的，二維線段樹則是用來維護二維陣列，或是一些和二維空間修改查詢相關的操作很多時候都可以用二維線段樹來完成。

如果說一維線段樹維護  $x$  軸的話，二維線段樹就是在維護  $y$  軸，二維線段樹每個節點都是一棵一維線段樹，因此在中國這種一棵樹套另一棵樹的結構，被稱為樹套樹。

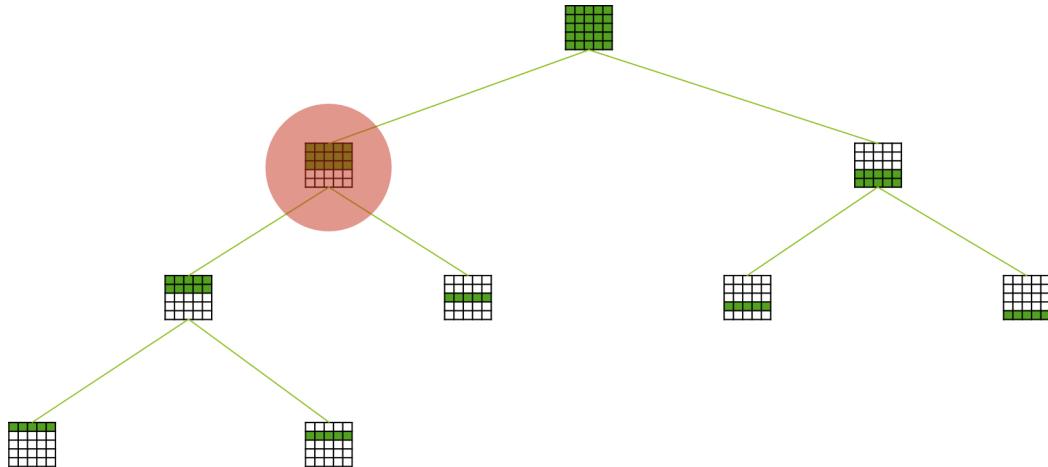


圖 7.14：二維線段樹

圖7.14中可以看到我們以  $y$  軸進行切割，將二維陣列切成很多條線段(圖中葉節點的部分)，然後將其構造二維線段樹，每個二維線段樹的節點又是一棵一維線段樹，以圖中圈起來的點為例，把它展開之後如圖7.15：

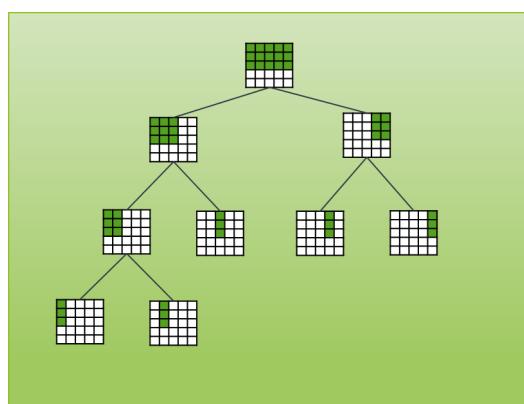


圖 7.15：二維線段樹套一維線段樹

其中深色的部分代表要維護的資訊，因此如何對這些一維線段樹進行維護就變成了非常重要的工作。

以下的二維線段樹也是以單點加值區間查詢總和為範例。

## 結構定義

和一維線段樹一樣，筆者習慣把資料結構寫在 struct 中，這樣在之後面對更高維度的樹套樹時就可以直接呼叫低維度的結果。

```
1 struct Seg2D {
2     static const int MAXM = 500;
3     Seg1D st[MAXM * 4];
4     void build(int ml, int mr, int nl, int nr, int d);
5     void insert(int y, int x, int data, int ml, int mr, int nl,
6                 int nr, int d);
7     int sum(int y1, int y2, int x1, int x2, int ml, int mr, int nl,
8             int nr, int d);
9 }
```

程式碼 7.33: 結構定義

## 合併兩棵一維線段樹

和一維線段樹一樣，如果題目給你一個  $m \times n$  的二維陣列，可以直接用它來構造二維線段樹，花費  $\mathcal{O}(m \times n)$  的時間，或是構造一空的樹然後進行  $m \times n$  次的加值操作，很明顯直接構造的方法時間是比較短的。

如果從陣列直接構造二維線段樹的話，一維線段樹要再多支援一個「合併」的操作，記得要把他加在 seg\_1d 的定義中！

```
1 void Seg1D::merge(const Seg1D &a, const Seg1D &b, int l, int r,
2                     int d = 1) {
3     st[d] = a.st[d] + b.st[d];
4     if (l == r) return;
5     int mid = (l + r) / 2;
6
7     merge(a, b, l, mid, d * 2);
8     merge(a, b, mid + 1, r, d * 2 + 1);
9 }
```

程式碼 7.34: 合併兩棵一維線段樹

注意這個操作的複雜度是  $\mathcal{O}(n)$

## 構造二維線段樹

構造方法非常的簡單，首先一直遞迴下去直到葉節點，因為葉節點是一維線段樹，而且剛好只記錄一條陣列的值，因此直接呼叫構造一維線段樹的函數即可，非葉節

點的部分，當我們遞迴構造完左右子樹後，左右子樹就會是完整的一維線段樹，因此只要呼叫合併兩棵一維線段樹的函數就可以了。

```
1 void Seg2D::build(int ml, int mr, int nl, int nr, int d = 1) {
2     if (ml == mr) {
3         st[d].build(nl, nr);
4         return;
5     }
6     // 合併 st[d*2] st[d*2+1] 兩棵樹把值存到 st[d]
7     int mid = (ml + mr) / 2;
8     build(ml, mid, nl, nr, d * 2);
9     build(mid + 1, mr, nl, nr, d * 2 + 1);
10    st[d].merge(st[d * 2], st[d * 2 + 1], nl, nr);
11 }
```

程式碼 7.35：構造二維線段樹並把樹清空成 0

可以發現二維線段樹的 build 和一維線段樹是十分相近的，會產生  $\mathcal{O}(m)$  個節點，每個節點又是一棵一維線段樹，因此總共用了  $\mathcal{O}(m \times n)$  的空間，複雜度也是  $\mathcal{O}(m \times n)$ ，用陣列實作的話就要花費  $4m \times 4n = 16mn$  的空間。

### 查詢區間總和

線段樹的目標就是要能動態修改、動態查詢，因此這算是線段樹的主要操作之一。

在二維線段樹，我們要查詢的是一個二維陣列中，某個子陣列值的總和，它會先在二維線段樹的部分搜尋在 Y 軸  $y_1, y_2$  覆蓋的區域，每找到一個完全覆蓋的區域就對該點的一維線段樹呼叫查詢 X 軸  $x_1, x_2$  覆蓋的區域，這樣保證查詢到的區域加起來剛好會拼成我們要查詢的區域。

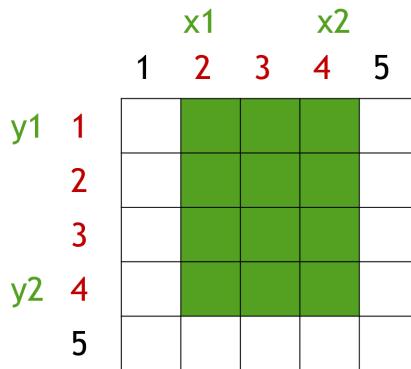
```

1 int Seg2D::sum(int y1, int y2, int x1, int x2, int ml, int mr, int nl,
2                 int nr, int d = 1) {
3     // 不再範圍中
4     if (mr < y1 || y2 < ml) return 0;
5
6     // 剛好在範圍中，直接查詢該點一維線段樹的結果
7     if (y1 <= ml && mr <= y2) return st[d].sum(x1, x2, nl, nr);
8
9     // 剩下的情況就遞迴左右子樹找出答案
10    int mid = (ml + mr) / 2;
11    int sl = sum(y1, y2, x1, x2, ml, mid, nl, nr, d * 2);
12    int sr = sum(y1, y2, x1, x2, mid + 1, mr, nl, nr, d * 2 + 1);
13    return sl + sr;
14}

```

**程式碼 7.36:** 查詢區間總和

如果我們要搜尋圖7.16的區間。



**圖 7.16:** 二維線段樹

我們會再二維線段樹上找到圖7.17這些圈起來節點。

這兩個節點都是一維線段樹，如圖7.18所示，對這兩棵樹進行 X 軸方向的查詢，找到的值加總起來就會是我們要的答案。

二維線段樹最多會查詢到  $\mathcal{O}(\log m)$  個點，每個查詢到的點還要對該點的一維線段樹進行查詢，一維線段樹最多會查詢到  $\mathcal{O}(\log n)$ 。

因此總複雜度就會是  $\mathcal{O}(\log m \times \log n)$ 。

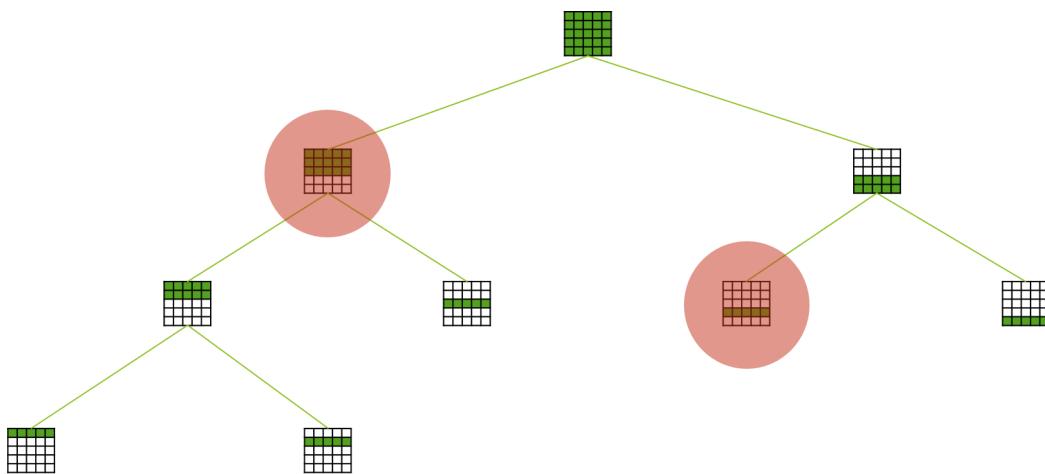


圖 7.17: 二維線段樹

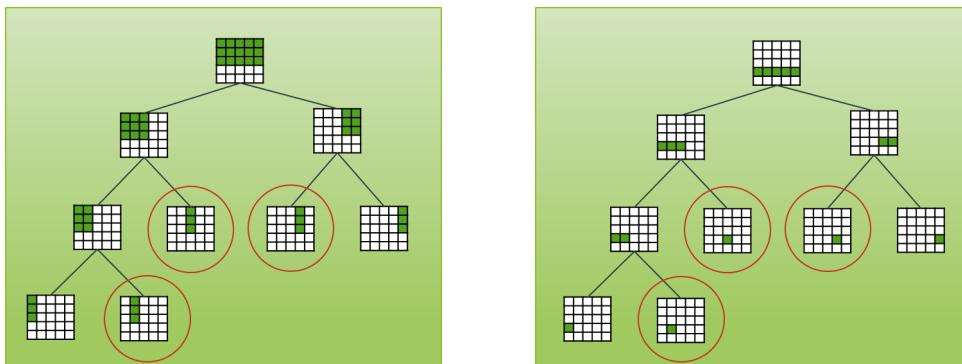


圖 7.18: 找到的點

### 單點修改

要把陣列中的某個點加上值，我們先在二維線段樹上對  $y$  進行二分搜，因為經過的點  $x$  位置的值都會被修改到，所以路徑上所有點都要對  $x$  的位置進行修改，這僅限於單點加值區間查詢總和這種簡單的操作才能這樣寫。如果是單點加值區間查詢最大、最小或是區間 gcd 這些比較雜的，可能還要配合查詢的操作來進行，理論上不會影響到複雜度的，這有點複雜就留給讀者思考吧！

```

1 void Seg2D::insert(int y, int x, int data, int ml, int mr, int nl,
2                     int nr, int d = 1) {
3     if (y < ml || mr < y) return;
4     st[d].insert(x, data, nl, nr);
5     if (y == ml && y == mr) return;
6     int mid = (ml + mr) / 2;
7     insert(y, x, data, ml, mid, nl, nr, d * 2);
8     insert(y, x, data, mid + 1, mr, nl, nr, d * 2 + 1);
9 }
```

程式碼 7.37: 單點加值

路徑上會經過  $\mathcal{O}(\log m)$  個點，每個在對它進行一維線段樹的修改，總複雜度就是  $\mathcal{O}(\log m) \times \mathcal{O}(\log n) = \mathcal{O}(\log m \times \log n)$ ，和查詢相同。

### 7.4.3 高維線段樹

在此我們已經講完了簡單的二維線段樹的基本操作，可以發現 code 和一維線段樹很多是幾乎一樣的，在對一維的查詢修改只需要呼叫一維線段樹的函數就可以了，同樣的道理也可以推廣到三維線段樹，變成是每個三維線段樹的節點是一棵二維線段樹。這樣層層疊疊樹套樹，只要好好維護這些結構，要寫出更高維度的線段樹是絕對沒有問題的。

### 7.4.4 懶惰標記？

非常可惜的是，高維線段樹如果要加上懶惰標記的話，只能選擇紀錄其中一個維度，或是變成單點查詢區間修改的操作，如果真的要用到懶惰標記，也有以下的解決方案：

#### 懶惰標記解決方案-永久化標記

永久化的意思就是不將懶惰標記往下推，查詢的時候經過該點就把該點懶惰標記的資訊加在答案中，但可惜的是只有少部分的懶惰標記可以永久化。

#### 懶惰標記解決方案-四分樹

二維的話把平面空間切成四塊，像線段樹一樣遞迴切割下去直到每塊的大小為一，通常會將長寬變成二的整樹倍在進行操作。三維的話就是八分樹、四維就是十六分樹...。這東西是可以打懶惰標記的，可惜如果你的二維陣列是  $m \times n$  大小的話，修改和查詢的複雜度就會是  $\mathcal{O}(m + n)$ ，動態開點的時候很容易 TLE 或 MLE。

#### 懶惰標記解決方案-kd 樹代替高維線段樹

這是目前筆者在寫 POJ 題目時遇到的東西，KD 樹的複雜度不再於你陣列是幾乘幾，而是在於樹中總共有多少個點，假設有  $N$  個點好了，那區間修改區間查詢的複雜度就會是  $\mathcal{O}(N^{(k-1)/k})$ ，這裡的  $k$  是維度的意思，以二維 kd 樹來說  $k = 2$ 。

雖然看起來挺糟糕的，但是卻有良好的空間複雜度，而且在有些點比較少的情況下，實際運作情形是和線段樹差不多的，甚至更好！筆者當時在解 IOI2013 game 這一題時，就是使用 KD 樹來寫的，速度然比正解慢了一些，但是空間使用上卻是其他人的十分之一！

#### 7.4.5 練習題 Exercise

|   |           |
|---|-----------|
| Census  | UVA 11297 |
| 給一個 $N \times N (1 \leq N \leq 500)$ 的矩陣 S，支持單點修改，查詢某個矩形區域的最大值。 |           |

|  |          |
|--|----------|
| Matrix   | POJ 2155 |
| 給一個 $N \times N (1 \leq N \leq 1000)$ 的 01 矩陣 S，支援兩個操作：<br>1. 把某個矩形區域中所有的值 xor 1<br>2. 查詢某個點的值是 0 還是 1 |          |

|   |         |
|---|---------|
| game  | ioi2013 |
| 給一個 $R \times C (1 \leq R, C \leq 10^9)$ 的矩陣 S，一開始都是 0，支援兩個操作：<br>1. 修改某個點的值<br>2. 查詢某個矩形區域中所有值的最大公因數 (GCD) |         |

|   |          |
|---|----------|
| 二維陣列單點修改區間查詢總和                          | SKYOJ 79 |
| 給你一個二維陣列，你有兩個操作可以做：修改某一位置的值和求一個矩形區域的總和。 |          |

|  |          |
|--|----------|
| 二維陣列單點修改區間查詢最大值                          | SKYOJ 80 |
| 給你一個二維陣列，你有兩個操作可以做：修改某一位置的值和求一個矩形區域的最大值。 |          |

|   |          |
|---|----------|
| 永久化標記   | SKYOJ 99 |
| 二維陣列區間修改區間查詢總和<br>給你一個二維陣列，你有兩個操作可以做：把某個矩形區域全部加上某個值和求一個矩形區域的總和。 |          |

# 基礎圖論 Basic Graph Theory

## 8.1 什麼是圖論 Introduction

本章將介紹圖論，也就是圖的理論。此處的圖並不是指圖片 (Picture)，而是更近似於關係圖的概念。一個圖 (Graph)  $G$  是由節點 (Vertex) 的集合  $V$  和邊 (Edge) 的集合  $E$  所組成的離散結構，我們表示為  $G = (V, E)$ 。

節點所代表的可能是一個人、一座城市、一個物體；邊所代表的是單一節點與單一節點的關係，是無分岔連接單一起點及終點的線，可能是描述兩座城市間的距離、或是兩人之間的父子關係。

當我們在畫圖的時候，我們在乎的是節點之間的相連關係，因此不會特別在意節點的座標跟大小和邊的長度及形狀。

圖的應用非常廣泛，下表為一些圖論的應用以及節點和邊分別代表的意義。

| 圖     | 節點   | 邊    |
|-------|------|------|
| 金融    | 股票   | 交易   |
| 交通    | 十字路口 | 道路   |
| 人際    | 人    | 互動   |
| 神經網路  | 神經元  | 突觸   |
| 蛋白質網路 | 蛋白質  | 交互作用 |
| 分子    | 原子   | 鍵結   |

表 8.1: 圖的應用舉例

在本章中，我們將會基礎的介紹何為圖論、圖論在講述些甚麼，並試圖將一些題目轉化為圖，以尋找題目的特性或突破口，藉此來為大家打下圖論的基礎。

## 8.2 名詞介紹

邊的種類分為有向邊跟無向邊，也就是單向道跟雙向道。根據圖上邊的種類，我們可以分為有向圖（圖上的邊都是有向邊）和無向圖（圖上的邊都是無向邊），只有極少數題目為混合圖（圖上同時有有向邊和無向邊）。

在探討圖論的時候，有幾個常見名詞要熟記：

和節點相關的名詞：

1. 相鄰 (adjacent)：如果有一條邊連接節點  $u$  和  $v$ ，則  $u$  和  $v$  是相鄰的。
2. 度數 (degree)：與節點相接的邊數。我們以  $\deg(v)$  表示節點  $v$  的度數。
  - a) 入度 (in-degree)：有向圖中，指向節點的邊數。以  $\deg^-(v)$  表示。
  - b) 出度 (out-degree)：有向圖中，從節點往外指的邊數。以  $\deg^+(v)$  表示。

和邊相關的名詞：

1. 權重 (weight)：邊上的數值，可能代表兩點間的距離、最大流量等等。
  - a) 負邊 (negative edge)：權重  $< 0$  的邊，有些演算法只適用沒負邊的圖。
2. 重複邊 (multiple edge)：起點與終點相同的兩條邊，不論權重是否相同。
3. 路徑 (path)：給定起點  $x$  和終點  $y$ ，由  $x$  出發，沿著邊走到  $y$ ，途中經過的邊構成的序列稱為路徑。
  - a) 簡單路徑 (simple path)：路徑上同一個點只經過一次。
  - b) 最短路徑 (shortest path)：從起點到終點的所有路徑中，權重和最小的路徑。
4. 環、迴路 (cycle)：一條路徑的起點與終點若為相同的點，則稱為環。
  - a) 自環 (self-loop)：一條邊的起點與終點相同，即一個節點自己連到自己所形成的環。
  - b) 負環 (negative cycle)：一個權重和  $< 0$  的環。
5. 連通 (connected)：點  $A$  與點  $B$  是連通，表示存在一條路徑由  $A$  走到  $B$ 。
  - a) 連通塊 (connected components)：如果一個節點集合中的所有節點互相連通，我們將此集合稱之為連通塊。

和圖相關的名詞：

1. 有向圖 (directed graph)：圖中的邊皆為有向邊。
2. 無向圖 (undirected graph)：圖中的邊皆為無向邊。
3. 混合圖 (mixed graph)：圖中同時有有向邊和無向邊。
4. 簡單圖 (simple graph)：一個不含重複邊和自環的無向圖。
5. 子圖 (subgraph)：一個  $G(V, E)$  的子圖為由  $V$  和  $E$  的子集合  $V'$ 、 $E'$  所形成的圖，其中  $V'$  必須包含  $E'$  中每個邊的兩個端點。

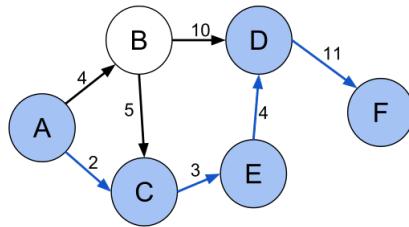


圖 8.1:  $A$  到  $F$  的最短路徑範例

## 8.3 圖的表示法 Graph Representation

我們會將點跟邊的資訊分開儲存。通常以  $0 \sim n-1$  或是  $1 \sim n$  編號每個節點，並且將無向邊拆成兩條相反方向的有向邊儲存。邊的資訊根據數量會有不同的存法，而以下會介紹其中的兩種：相鄰矩陣與相鄰串列。

### 8.3.1 相鄰矩陣 Adjacency matrix

開一個二維陣列存邊，兩個維度分別為起點和終點的節點編號。如果圖上的邊沒有權重，通常使用 bool 陣列存節點  $i, j$  之間有沒有邊。如果圖上的邊有權重，通常使用 int 存節點  $i, j$  之間的權重，如果  $i, j$  間不存在邊，通常以 INF 表示。

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 int n, m;
4 int G[2009][2009];
5 int main() {
6     cin >> n >> m;
7     for (int i = 0; i < m; ++i) {
8         int a, b;
9         cin >> a >> b;
10        G[a][b] = 1;
11        //G[b][a] = 1; 無向圖的話要加上這一行
12    }
13 }

```

程式碼 8.1: 相鄰矩陣

| i\j | A   | B   | C   | D   | E   | F   |
|-----|-----|-----|-----|-----|-----|-----|
| A   | INF | 4   | 2   | INF | INF | INF |
| B   | INF | INF | 5   | 10  | INF | INF |
| C   | INF | INF | INF | INF | 3   | INF |
| D   | INF | INF | INF | INF | INF | 11  |
| E   | INF | INF | INF | 4   | INF | INF |
| F   | INF | INF | INF | INF | INF | INF |

表 8.2: 以相鄰矩陣表示圖

要小心當點過多時無法使用此資料結構儲存，也要小心是否有重邊。根據比賽的電腦有所差異，但通常點大於 5000 之後，不會使用相鄰矩陣。常見的錯誤還有初始化過慢的問題，可再增加一個維度來記錄資料是屬於哪一組測資，就可除去初始化的流程。

但因為空間複雜度高達  $O(N^2)$ ， $N$  為點數，因此這個方法實際上出場的機會較少。

### 8.3.2 相鄰串列 Adjacency List

對於每一個點，儲存以此點為起點的邊。通常使用一維 vector 陣列儲存。

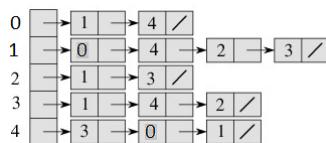


圖 8.2: 相鄰串列

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int n, m;
5 vector<int> v[1000009];
6
7 int main() {
8     cin >> n >> m;
9     for (int i = 0; i < m; ++i) {
10         int a, b;
11         cin >> a >> b;
12         v[a].push_back(b);
13         //v[b].push_back(a); 無向圖的時候要加上這一行
14     }
15 }
```

若要儲存帶邊權的圖的話，則可以藉由建立`vector<pair<int,int>>`來完成。

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 int n, m;
5 vector<pair<int,int> > v[1000009];
6
7 int main() {
8     cin >> n >> m;
9     for (int i = 0; i < m; ++i) {
10         int a, b, w;
11         cin >> a >> b >> w;
12         v[a].push_back(make_pair(b, w));
13         //v[b].push_back(make_pair(a, w)); 無向圖的時候要加上這一行
14     }
15 }
```

使用相鄰矩陣的優點是可以  $O(1)$  確認兩點之間有沒有直接連接，而相鄰串列的話則必須要對每個 `vector` 進行 `sort` 後二分搜。但相鄰矩陣的空間複雜度實在過大，因此大多數情況下還是使用相鄰串列。

## 8.4 圖的遍歷 Graph Traversal

圖的遍歷基本上有兩種方法，深度優先搜尋 (Depth First Search, DFS) 和廣度優先搜尋 (Breadth First Search, BFS)。DFS 的作法是當遇到岔路時，選擇一條路深入底後，才探索下一條路。BFS 則是同時探索所有岔路，並紀錄每條岔路中探索到哪。

在遍歷圖的過程中，我們會從一個節點轉移到另一個節點。迷宮問題裏，由目前位置移動到下一個位置就是轉移。第8.4.2節的倒水桶問題中，拿一個水桶倒進另一個水桶後得到下一個狀態也是轉移。很多人會卡在 DFS 跟 BFS 的原因是：不知道程式碼要填什麼。在此建議從拜訪到一半的狀態開始想像：

1. 目前狀態：到了一個節點後，這個點的狀態要從未處理標記成處理中。
2. 轉移：可以轉移到哪些狀態？
3. 結束：點的狀態由處理中改為完成（跟未處理不一樣）。

對於一些暴力搜尋的題目，我們可以根據題目假設狀態，並把狀態設為節點。如果一個狀態可以延伸另一個狀態，則在這兩個狀態所代表的節點之間建立邊。這樣所枚舉的狀態就變成一張圖，枚舉的過程就變成圖的遍歷。因此我們可以用 DFS 和 BFS 處理一些暴力枚舉的題目。

### 8.4.1 深度優先搜尋 DFS

DFS 顧名思義，是有路的話就一直往下走，一直走到死路後再回頭後，繼續試著往下走。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 bitset<1000009> visited;
5 vector<int> v[1000009];
6
7 inline void dfs(int u) {
8     visit[u] = 1;
9     for (auto i:v[u]) if (!visited[i]) {
10         dfs(i);
11     }
12 }
```

程式碼 8.2: DFS 範例

要記得將走過的地方標記起來，以免下次再走到；另外，由遞迴的特性，走到死路後自然就會回頭了。

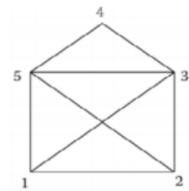
DFS 較常用來枚舉與順序性有關的題目，如：排列組合（誰坐誰不坐）、一筆劃問題。DFS 是遞迴的一種，要注意兩點關鍵：邊界條件、遞迴式。

## 例題演練

### 一筆劃問題

UVA 291

由小到大輸出右圖所有由 1 開始經過所有邊的走法，第一組是 123153452，最後一組是 154352312。(註：uva 上範例輸出有誤)



解題的第一步是將圖建出來，題目要求要依照大小輸出，如果使用相鄰串列可事先將邊依照所到達節點的編號由小到大排序。接著用 dfs 的方式枚舉不重複邊的走法，並且紀錄當前走法依序經過哪些節點。當遞迴深度為 8 時表示這個走法經過所有邊，因此輸出這組走法。

### 練習題

#### 1. 銀河帝國旅行社

TIOJ 1152

給你一顆至多有  $10^4$  個點的樹，請問樹上最遠的兩個點間的距離是多少。

(hint: 雖然值域很小可能可以枚舉所有點對的距離，但這題其實是樹直徑這樣的經典問題喔)

#### Destruction of a Tree

CF 963B

給你一顆至多  $2 \times 10^5$  個節點的樹，一個點可以被摧毀若且唯若它的度數為偶樹，且一個點被摧毀後，它所連接出去的邊也會跟著被摧毀，請問能不能摧毀掉整顆樹，如果可以，請輸出摧毀的順序。

(hint: 可以將節點數為偶數和奇數的情況分開考慮)

P.S 這題是 CF div2 的 D，稍微有點巧思，不妨嘗試思考看看吧

#### 1994. 冰塊線

TIOJ 1994

請輸出邊長為  $2^k$  的希爾伯特曲線。

$k \leq 11$

#### 數獨問題

TIOJ 1025

給你一個未完成的數獨，請輸出所有可能的答案。

(hint: 雖然乍看之下會 TLE，但其實只要好好檢查發生矛盾的情況就可以很快跑出結果來的)

給你一棵點數至多為  $10^4$  的樹，定義一條邊被佔領若且唯若那條邊的兩個端點中至少一個被占領，定義一棵樹被砍倒若且唯若所有邊都被占領，請問你至少要佔領多少點才能砍掉這棵樹？

### 8.4.2 廣度優先搜尋 BFS

BFS 的搜尋過程可以想像成是將水倒出來，這時水會以等速率向四周擴散，BFS 也是同個道理，BFS 先搜尋  $n$  步可以走到的點後，再搜尋  $n + 1$  步可以走到的點，以此類推。

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 bool visited[1000009];
5 vector<int> v[1000009];
6 queue<int> q;
7
8 int main() {
9     q.push(1);
10    while (q.size()) {
11        int u = q.front();
12        q.pop();
13        for (auto i:v[u]) if (!visited[i]) {
14            visited[i] = 1;
15            q.push(i);
16        }
17    }
18 }
```

程式碼 8.3: BFS 範例

和 DFS 不一樣的是，我們利用了 `queue` 這個資料結構來維護 BFS 的搜尋過程。每走到一個點後，就將其周圍還沒走到的點都放去 `queue` 的最後面，這樣就能保證離原點較近的點會先被走到。此外，已經放進 `queue` 裡的點要標記起來，以免重複放進去。

注意將節點標記為「已走過」時機，放錯地方的話可能導致無窮迴圈喔 ~~

## 例題演練

### 水桶問題

UVA 571

有兩個水桶 A,B，容量分別為  $C_a, C_b$ ，求如何操作可以得到 N 公升的水（位於 A 或 B 水桶皆可）。一開始兩個水桶都是空的。以下幾種操作：

1. 裝滿 A
2. 裝滿 B
3. 清空 A
4. 清空 B
5. A 倒滿 B ( 不溢出 )
6. B 倒滿 A ( 不溢出 )

以兩水桶水量當作狀態，每個狀態最多可以再延伸出 6 個狀態。不難發現這題要找的是此狀態圖中，從「A,B 都是 0」這個節點到「A,B 其中一個是 N」這個節點的路徑。我們可以用 BFS 的方式找出這條路徑。

### 8.4.3 練習題

#### Fire in the forest

TIOJ 1013

給你一張  $17 \times 10$  的棋盤圖，有些格子上著火了，若一個格子的上下左右有任何一個格子著火，那麼下一秒它也會跟著著火。你走路的速度為一秒一格，現在給你你的起點位置和目的地，還有目前棋盤的著火情況，請問你能夠成功到達目的地嗎？

#### 量杯問題

TIOJ 1008

你有最多五個有著整數容量的量杯，請問是否能用這些量杯量出 L 公升的水。  
(hint: 這題是經典的將狀態當作圖的節點的例子，但這題的狀態頗多，需要壓縮狀態的技巧)

## 8.5 最短路徑 Shortest Path

在一張圖上若兩點之間互不連通，則定義其距離為無限大；反之則定義為所有路徑之中，邊權和最小的那一個，也就是所謂的最短路徑。

### 8.5.1 BFS

首先考慮所有邊的邊權都一樣的情況，這樣一來很自然地就會想到使用 BFS，由於在 queue 中的節點都保證了先被放進去的肯定比後放進去的，離起點的距離更近，因此 BFS 就可以解決了；另外，有個可以增進效能的技巧稱為雙向 BFS，同時從起點與終點進行 BFS，但大多數比賽不會去卡這樣的常數。

#### 練習題

##### 三維迷宮問題

TIOJ 1085

給你一個三維棋盤狀的迷宮，長寬高皆小於 50，0 代表可以走、1 代表不能走的地方，請問從  $(0, 0, 0)$  走到對角線的那個最遠點的最短距離是多少。

##### H. 跑跑卡恩車

TIOJ 1022

在一張長寬皆小於 100 的棋盤上，相鄰兩格間能相通若且唯若兩格的數字差  $\leq 5$ ，請問從左上角走到右下角的最短距離為何。

### 8.5.2 鬆弛 Relax

作為圖論中的一個經典問題，求兩點間的最短路徑問題有個非常重要的核心概念，稱之為鬆弛 (Relax)：若目前  $a$  到  $b$  的距離大於  $a$  到  $c$  再從  $c$  到  $b$  的距離的話，則將其更新為較小值，即： $d[a][b] = \min(d[a][b], d[a][c] + d[c][b])$ 。

### 8.5.3 Dijkstra

Dijkstra 是單點源最短路徑演算法，其核心概念為貪心法：每次挑選一個離起點最近的點，將其標記為「確認了最短路徑的點」後，利用這個點去鬆弛起點與其他「還沒確認最短路徑的點」的距離，再從那些點中挑出一個離起點最近的點，重複這樣的動作。特別注意：上述性質成立的條件是圖上不存在負邊，因此 Dijkstra 不適用於有負邊的圖。

```

1 int n, m;
2 vector<pi> v[1000009]; //
3 priority_queue<pi, vector<pi>, greater<pi> > pq;
4 int d[1000009];
5
6 inline void dijkstra() { // 假設起點為1
7     for (int i = 2; i <= n; ++i) d[i] = 1e9;
8     pq.push(mk(1, 0));
9
10    while (pq.size()) {
11        auto u = pq.top();
12        pq.pop();
13        if (d[u.first] != u.second) continue;
14
15        for (auto i:v[u.first]) {
16            if (d[i.first] > u.second + i.second) {
17                d[i.first] = u.second + i.second;
18                pq.push(make_pair(i.first, d[i.first]));
19            }
20        }
21    }
22}

```

程式碼 8.4: Dijkstra 演算法

首先要記得將除了起點以外的點和起點的距離設定為無限大，`pq` 裡存的為每次選到的距離最近的點，但因為同個點可能會被鬆弛很多次，所以若是選到了一個「距離比較大」的點的時候，代表那個點已經被其他點鬆弛過了，因此就必須忽略這次。這個演算法的複雜度為  $O(E \log E)$ 。

### 練習題

#### F. 不定向邊

TIOJ 1209

給你一張點數少於  $10^3$ 、邊數少於點數的平方的帶權(非負整數)圖，在可以自由指定每條路的方向下，請問從起點到終點的最短路徑為多少。

(hint: 最短路徑在在邊權皆為非負的情況下是不會重複走到同樣的路的)

#### 地道問題

TIOJ 1509

給你一張有向帶權圖，點數邊數皆  $\leq 10^6$ ，對於每個點求從起點走到它再走回起點的最小距離，再把所有點的答案加總。

(hint: 可不是直接從起點做 *dijkstra* 加總後乘兩倍，因為回來的時候能走的路和去的時候不一樣)

給定一張點數少於  $10^4$ 、邊數少於  $2 \times 10^5$  的無向帶權(正整數)圖，並給定起點與終點，一開始在起點時你有 1 個東西，每經過一條邊權為  $P_i$  的點，你身上的東西就會變成(你現在有的東西數量)  $\times P_i$ ，請問到達終點時你身上最少有多少東西。

(hint: 看到連續乘法是不是有股衝動想要取 log 呢？)

#### 8.5.4 Bellman-Ford

Bellman-Ford 是另一種單點源最短路徑演算法，相對於 Dijkstra，Bellman-Ford 藉由每次檢查每條邊是否能夠鬆弛其端點，若沒有則代表找到了所有點的最短路；若檢查了  $V$ (點數) 次後仍能鬆弛，就代表此圖存在負環；時間複雜度為  $O(VE)$ 。

```

1 int n, m;
2 bitset<1000009> inque;
3 vector<pi> v[1000009];
4 queue<int> q;
5 int d[1000009], cnt[1000009];
6
7 inline bool SPFA() {
8     for (int i = 1; i <= n; ++i) d[i] = INF; // 初始化距離
9     q.push(1); // 假設起點為 1
10    inque[1] = 1;
11    d[1] = 0;
12    while (q.size()) {
13        int u = q.front();
14        q.pop();
15        inque[u] = 0;
16
17        for (auto i:v[u]) if (d[i.first] > d[u] + i.second) { // 只需要處理將
18            // 會被更新的點
19            if (++cnt[i.first] >= n) return false; // 發現負環
20            d[i.first] = d[u] + i.second;
21            if (!inque[i.first]) {
22                inque[i.first] = 1;
23                q.push(i.first);
24            }
25        }
26    }
27    return true; // 沒有負環
}

```

程式碼 8.5: SPFA 演算法

SPFA 演算法全名 Shortest Path Faster Algorithm，是 Bellman-Ford 的優化。SPFA 把更新過的節點放入一個 Queue，並且只檢查 Queue 中節點上的邊。在隨機生成的

圖上，平均每個節點被更新兩次，期望複雜度為  $O(2E)$ 。但很容易可以構造出會跑到  $O(VE)$  的測資。

SPFA 一樣可以用來檢測負環：如果一個點被更新  $V$  次，表示圖中含有負環。

### 8.5.5 Floyd-Warshall

重新看看鬆弛的算式： $d[a][b] = \min(d[a][b], d[a][k]+d[k][b])$ ，是不是看起來很像 DP 啊？若將 DP 的狀態設為：

$$dp(k, i, j)$$

代表從  $i$  走到  $j$  並且經由前  $k$  個節點鬆弛過後的結果，那麼轉移式就是：

$$dp(k+1, i, j) = \min \{ dp(k, i, j), dp(k, i, k+1) + dp(k, k+1, j) \}$$

觀察一下發現討論  $k+1$  時只會用到  $k$  的部份，因此我們可以直接重複使用 dp 陣列。Floyd-Warshall 的時間複雜度  $\mathcal{O}(V^3)$ ，建 dp 表後可以  $\mathcal{O}(1)$  查詢任一點對  $(i, j)$  的最短距離。

```
1 int dis[N][N];
2 for (int k = 0; k < N; ++k)
3     for (int i = 0; i < N; ++i)
4         for (int j = 0; j < N; ++j)
5             dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
```

程式碼 8.6: floyd warshall

這是一個需要用到鄰接矩陣的例子。

#### 練習題

##### 搶救雷恩大兵 (Saving Ryan)

TIOIJ 1034

給  $N \times N (\leq 20)$  的棋盤圖，每個棋盤上的點都有權值。 $Q (\leq N^4)$  筆詢問兩個點的路徑中，可以把一個點的值改成 0 的狀況下，最小的總和是多少。

(hint: 這題有點繁複，對於可以將其中一個點改為 0 這件事，可以嘗試修改 Floyd-Warshall 的 DP 式來達到)

圖論之最小圈測試

TIOJ 1212

給一張點數少於  $5 \times 10^2$  的無向圖，邊權皆為一，請問長度最小的環的長度為多少？

(hint: 一般的鄰接矩陣是將自己到自己設為零，這裡可以嘗試將其設為無限大試試看)

E. 漢米頓的麻煩

TIOJ 1096

給一張點數至多為 100 的圖，一個點的花費為從自己經過一些不重複的路徑後回到自己的邊權和，請問所有點中最少的花費為何？

(hint: 這題只是 TIOJ 1212 有邊權的版本唷)

## 8.6 樹 Tree

樹的定義為一張沒有環的連通圖。

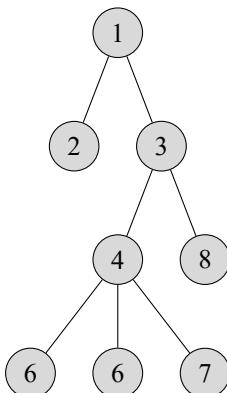


圖 8.3: 樹

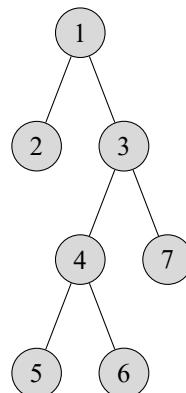


圖 8.4: 二元樹

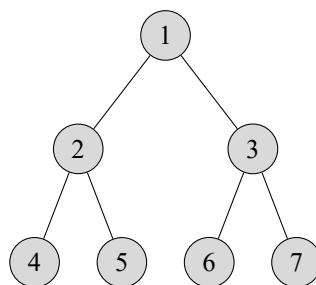


圖 8.5: 完整二元樹

### 8.6.1 專有名詞與性質 Glossary and Property

以下是樹當中的常見名詞：

1. 根 (root)：樹的起點，根據是否有指定起點可分為有根樹 (rooted tree) 和無根樹 (unrooted tree)。
2. 父節點 (parent)、子節點 (child)：遍歷樹的過程中，我們由父節點走向相鄰的子節點。
3. 葉節點 (leaf)：為位於樹的盡頭，延伸不出子節點。
4. 祖先 (ancestor)：由父節點到樹根路徑上的所有節點

5. 子代 (descendant)：向下所能造訪到的所有節點。
6. 子樹 (subtree)：分別以每個子節點為根的樹。
7. 層 (level)：一個節點與根的距離，根位在第 0 層。
8. 深度 (depth)：整棵樹的層數，即從樹根到最遠的葉節點的路徑上經過的節點總數。

根據樹的定義，我們可以推導出一些樹的性質：

1. 一顆樹剛好  $|V| - 1$  條邊。
2. 一顆樹沒有環，但加上任何一條邊都會出現環。
3. 一顆樹連通，且任何一條邊被拿掉都會導致圖不連通。
4. 一顆樹上任意兩點只存在一條簡單路徑。

一張圖上若沒出現環，則存在至少一棵樹（一個點也算樹）。如果圖上不只一棵樹，則稱為森林。

### 8.6.2 二元樹 Binary Tree

二元樹的特性為一個節點最多只有兩個子節點，例如圖8.4和8.5都屬於二元樹。

#### 樹的遍歷 Traversal

二元樹的遍歷基本上分為以下四種。

1. 前序：先走根，再走左子樹，最後走右子樹
2. 中序：先走左子樹，再走根，最後走右子樹
3. 後序：先走左子樹，再走右子樹，最後走根
4. 層次：由上到下、由左到右遍歷

如果你同時有著一棵二元樹的前序、中序，或者是中序、後序的話，那你就能夠確定這棵二元樹的結構。若你只有一顆二元樹的前序、後序的話，就可能會分辨不出一個葉節點是左節點還是右節點。

## 練習題

### Problem C 二元搜尋樹 (TRVBST)

TIOJ 1609

現在有一顆空的二元搜尋樹，並按照順序插入一些數字，最後請輸出這棵樹的中序表示法。

(hint: 這其實是梗題唷)

### 樹狀的堆積結構

TIOJ 1204

給你一顆點數至多為  $10^3$  的堆積 (heap，實作 priority\_queue 的資料結構，也就是所有父節點都大於其子節點) 的中序表示法，請輸出其前序表示。

### 遇見一株樹

TIOJ 1106

給你一棵樹的簡化表示法，請輸出這棵是幾元樹、深度為何、有多少葉子。

P.S 有點難以用口頭描述，建議直接去看原題

### 繁複的二元樹

TIOJ 1107

請問要將  $N$  個數字  $1, 2, \dots, N$  表示成一棵二元搜尋樹，有多少種表示法？由於答案可能很大，因此輸出四捨五入後的科學記號即可。

### 樹的三兄弟

TIOJ 1108

給定一棵樹的前序和中序表示法，請輸出後序表示法。

## 完整二元樹 Complete Binary Tree

完整二元樹為接滿的二元樹（除了葉節點外每個節點都有兩個子節點）。若父節點的編號為  $id$ ，則可以將左節點設為  $id * 2$ 、右節點為  $id * 2 + 1$ ，是不是似曾相似的感覺呢？線段樹的區間分割其實也是基於類似的想法。

根據上述性質，我們可以用一維陣列存放一顆完整二元樹，如下圖所示：

一棵完整二元樹的節點，其中大約有一半的節點都會是葉子。

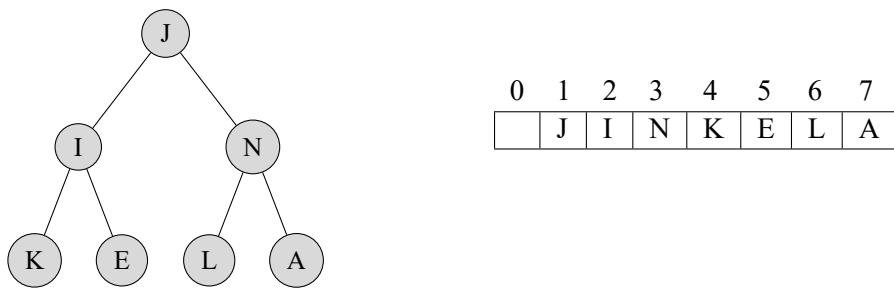


圖 8.6: 陣列存完整二元樹

練習題

樹論之好多星星 ver 1.3

TIOJ 1215

給你節點數  $N$ ，請畫出節點數為  $N$  的二元樹，其中  $N$  小於 64。

APCS1050305 血緣關係

zerojudge b967

給你一顆樹，求直徑（最遠的兩個節點距離）。

## 8.7 最小生成樹 Minimum Spanning Tree

生成樹 (spanning tree) 為一棵包含圖上所有點的樹。一張圖上可能存在多種生成樹。最小生成樹 (Minimum Spanning Tree, MST) 為權重和最小的生成樹。

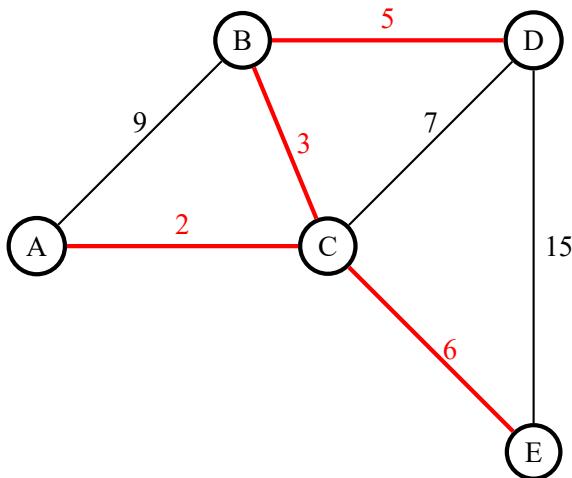


圖 8.7: 圖上最小生成樹

舉例來說，電信公司佈線到每戶人家，但希望所用的電線越少越好，如果對每戶人家建圖，權重為兩戶間的距離，此時佈線的最佳辦法為圖上的最小生成樹。常見建構 MST 的演算法有兩種：以邊出發的 Kruskal、以點出發的 Prim。兩種都是以貪心的方式生成 MST。

### 8.7.1 Kruskal

由權重最小的邊開始加入生成樹，若目前選到的邊的兩端點都已經在生成森林中的同一顆樹時則略過；具體作法為將邊排序後，以 disjoint set 維護點是否在同一顆樹中，因此時間複雜度為  $O(E \log E)$ 。

```
1 #include <bits/stdc++.h>
2 #define pi pair<int, int>
3 #define mk make_pair
4 #define F first
5 #define S second
6 using namespace std;
7
8 int n, m, ans;
9 vector<pair<int, pi>> G;
10
11 //省略Disjoint set
12
13 int main() {
14     cin >> n >> m;
15     for (int i = 1; i <= m; ++i) {
16         int a, b, w;
17         cin >> a >> b >> w;
18         G.pb(mk(w, mk(a, b)));
19     }
20
21     sort(G.begin(), G.end());
22     for (auto i:G) if (!same(i.S.F, i.S.S)) {
23         unite(i.S.F, i.S.S);
24         ans += i.F;
25     }
26 }
```

程式碼 8.7: kruskal

要注意的是，由於要對邊權排序，所以在這裡並不是使用鄰接串列來儲存圖，而是直接將每條邊存在 vector 裡面。

### 8.7.2 Prim

回想 Dijkstra 是每次尋找離起點最近且還沒標記過的點，Prim 實際就是每次尋找離目前的最小生成樹最近的點加進來。

```

1 #include <bits/stdc++.h>
2 #define pi pair<int, int>
3 #define mk make_pair
4 #define F first
5 #define S second
6 using namespace std;
7
8 int n, m;
9 bitset<1000009> visited;
10 vector<pi> v[1000009];
11 priority_queue<pi, greater<pi>, greater<pi> > pq;
12
13 inline int prim() {
14     int ans = 0;
15     visited[1] = 1;
16     pq.push(mk(0, 1));
17     while (pq.size()) {
18         pi u = pq.top();
19         pq.pop();
20         if (visited[u.second]) continue;
21         visited[u.second] = 1;
22         ans += u.first;
23         for (auto i:v[u.second]) {
24             pq.push(i.second, i.first); // second代表邊權、first代表連到的點
25         }
26     }
27     return ans;
28 }
```

**程式碼 8.8:** 優先佇列版 prim

在一般比賽裡，求生成樹的題目中點數和邊數基本上會是同個數量級，但 kruskal 的出場率遠遠比 prim 高，其原因就在於 kruskal 除了更容易理解外，實在是太好寫了。

### 練習題

#### 圖論之最小生成樹

TIOJ 1211

給一張點數至多  $10^5$ 、邊數至多  $10^6$  的無向帶權圖，請求出最小生成樹的權重和。

#### 最小格子生成樹

TIOJ 1326

平面上給至多  $10^3$  個點，現在要用一些水平或垂直的線將所有點連在一起，請問這些線段的總長度最小為多少？

黑色騎士團的番外野望 Dis-connect

TIOJ 1596

給你一棵點數至多為  $10^5$  的樹，拆掉一條邊的費用為其之上的邊權，其中有些點是軍事堡壘，請問至少要花多少錢才能使兩兩軍事堡壘不相通？

咕嚕咕嚕呱啦呱啦

TIOJ 1795

給你一棵點數至多為  $10^5$ 、邊數至多為  $3 \times 10^5$  的圖，每條邊的邊權是一或者是零，沒有其他可能性，請問是否存在一棵生成樹的邊權恰好是  $K$ ？

## 8.8 拓樸排序 Topological Sort



圖 8.8：拓樸排序

給定一張有向無環圖 (DAG)，給每個點一個序號，總存在至少一個方式，使得所有序號大(小)的點不存在路徑到達序號小(大)的點，我們稱以這樣的序號排列出的點序列為拓樸排序。要構造的方法也很簡單，每次將出度為零的點 push 進一個 stack(下方是以 vector 代替)中即可。

```
1 int n, m, deg[1000009];
2 vector<int> s, t, v[1000009];
3
4 inline vector<int> TopologicalSort() {
5     for (int i = 1; i <= n; ++i) {
6         if (deg[i] == 0) s.push_back(i);
7         deg[i] = v[i].size();
8     }
9
10    while (s.size()) {
11        t.push_back(s.back());
12        s.pop_back();
13
14        for (auto i:v[t.back()]) {
15            deg[i]--;
16            if (!deg[i]) s.push_back(i);
17        }
18    }
19    return t;
20 }
```

程式碼 8.9：拓樸排序

值得一提的是，就算以入度來思考拓撲排序也是可以的，同時也有許多其他方法可以求出拓撲排序的唷。

### 8.8.1 練習題

A. 跳格子遊戲

TIOJ 1092

給一張 DAG，但只有一個起點的入度為零、只有一個終點出度為零，現在有個玩偶放在起點，A、B 兩人每回合可以輪流移動玩偶一步，移動到終點的那個人獲勝，請問 A 和 B 誰有必勝策略？

Fox And Names

CF 510C

給你一些只由小寫字母組成的字符串，現在按一定順序給出這些字符串，問你怎樣重新定義字典序，使得這些字符串按字典序排序後的順序如題目所給的順序相同。

## 8.9 二分圖 Bipartite Graph

二分圖顧名思義就是指，能夠將點分成  $X$ 、 $Y$  兩類，使得同一類裡的點之間互相都沒有邊相連的圖。

### 8.9.1 判斷一張圖是否為二分圖

二分圖有個等價定義為：不包含奇環的圖。若這一張圖中只包含了長度為偶數的環的話，那間隔的將點歸到  $X$  與  $Y$  內，自然就能滿足二分圖的定義；反之若有奇環的話，在那環上一定會有相鄰的兩點被放在同一類中。

所以判斷一張圖是否為二分圖的方法就不言自明了，利用 DFS 或 BFS 即可完成。

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 vector<int> v[1000009];
5 bitset<1000009> visited;
6 int col[1000009];
7
8 inline bool dfs(int x, int color) {
9     bool isBip = 1;
10    col[x] = color;
11    visited[x] = 1;
12    for (auto i:v[x]) {
13        if (visited[i] && col[i] == col[x]) return false;
14        if (!visited[i]) isBip &= dfs(i, color^1);
15    }
16    return isBip;
17 }
18
19 int main() {
20     int isBipartite = dfs(1, 1);
21 }

```

**程式碼 8.10:** 判斷是否為二分圖範例

值得注意的是，上方的範例使用了 `color` 相關詞彙作為命名，這是因為在判斷二分圖的過程，有時候會習慣以將點染色來稱呼，相鄰兩點不能同色；另外，範例使用了 XOR 將要染的顏色在 1 與 0 間切換。

### 8.9.2 練習題

圖論之二分圖測試

TIOJ 1209

給你一張點數最多為  $4 \times 10^4$ 、邊數最多為  $5 \times 10^5$  的圖，請判斷這是不是一張二分圖。

## 8.10 佛洛伊德演算法 Floyd Cycle Detection Algorithm

考慮以下問題。

Floyd Algorithm

經典問題

給定一張有向圖，已知這張圖上僅包含一個 path 連接在一個環上。請求出環與 path 的交點與環的長度。

這一題很明顯有著時間複雜度為  $O(n)$  的解決方法，但同時空間複雜度也要  $O(n)$ 。雖然至今為止我們大多只著重於時間複雜度上，但有時候題目會故意卡空間；而這一個問題就是一個經典的範例。

首先，path 的起點只要檢查誰的入度為零就能輕易找到。接下來，在那裏放置一隻烏龜和一隻兔子，每次烏龜走一步、兔子走兩步。假設 path 的長度為  $N$ ，環的長度為  $M$ ，烏龜和兔子相遇時烏龜走了  $A$  圈又  $K$  步、而兔子走了  $B$  圈又  $K$  步。

此時烏龜走的步數為：

$$N + AM + K$$

此時兔子走的步數為：

$$N + BM + K$$

相減可知，此時兩者的步數差即為環的長度  $M$  的倍數。又由於兔子走的步數會是烏龜的兩倍，所以烏龜走的步數  $N+AM+K$  也會是  $M$  的倍數。

若在此時將兔子放回 path 的起點，再次讓烏龜每次走一步、兔子也改為每次向前走一步，烏龜和兔子再次相遇的地方就會是環與 path 的交會點。而證明也很簡單，已經知道烏龜在這之前走的步數為  $M$  的倍數了，而再次走了  $N$  步後，就會變成  $N + (M \text{ 的倍數})$  的情況，而這樣的話就會待在環與 path 的交會點了。

那麼要如何求環的長度就留給各位自行思考了。

可以注意到，這樣的演算法除了在一開始要儲存整張圖以外，額外的空間複雜度為  $O(1)$ 。

```

1 #include<bits/stdc++.h>
2 using namespace std;
3
4 int a, b;
5
6 inline void succ(int x) {
7     //
8 }
9
10 inline void floyd_algorithm(int x) {
11     a = succ(x);
12     b = succ((x));
13     while (a != b) {
14         a = succ(a);
15         b = succ(succ(b));
16     }
17
18     a = x;
19     while (a != b) {
20         a = succ(a);
21         b = succ(b);
22     }
23
24 // 此時a與b即為path與環的交點
25
26     b = succ(a);
27     int length = 1;
28     while (a != b) {
29         b = succ(b);
30         ++length;
31     }
32 }
```

**程式碼 8.11:** floyd algorithm 範例

值得注意的是，我這裡使用了 `succ` 這樣的一個函式來表示下一個節點，一般來說，這樣的圖上每個點的出度都是 1；但有時候會用這個演算法來尋找迭代函數的週期... 之類的，所以為了那個時候所以這裡的範例就使用了 `succ` 來表示。

本章節是在去年的講義內容之上，補充一些細節、修改了一些講法、並另外加入一些新內容而成的，特別感謝去年負責基礎圖論的講師，與網路上那些我可以輕易搜尋到的資料，和那些把 code 收藏區設為公開的朋友們。

# 進階圖論 Advanced Graph Theory

在本章中，我們主要分為兩大部份：在 9.2 節中，我們針對一般圖上的連通分量 (component) 及其相關問題和演算法進行討論；接著在 9.4 節中，我們討論一系列與樹 (trees) 相關的問題和演算法。

## 9.1 DFS Tree

圖論的很多演算法都基於 DFS 來時做，因此，先來探討一張圖做 DFS 時走訪的路徑。

```

1 void DFS(int v) {
2     used[v] = true;
3     for (int w:adj[v]) {
4         if (!used[w]) {
5             DFS(w,v); // v->w 是 tree edge
6         } else {
7             // 其他種類
8         }
9     }
10 }
```

**程式碼 9.1:** DFS

在 DFS 的過程中，所有直接走訪的邊會構成一個樹，稱之為 DFS Tree，而這一些邊稱之為 Tree edge，而沒使用的邊，根據相鄰兩點的關係，分別有 Back edge，Forward edge，以及 Cross edge 的分類。

Back edge 指的是會回到當前點祖先 (ancestor) 的邊，如果發現有 Back edge，就表示有一個環 (cycle) 出現了，如範例圖中的 (4, 2) 邊。

Forward Edge 指的是會通往當前點的後繼 (descendant)，相當於在 DFS Tree 中，走到目前點子樹中的一個點，如範例圖中的 (1, 8) 邊。

Cross edge 指的是非上面兩類，會到與當前點無關的其他子樹上的點，如範例圖中的 (6, 3) 邊。

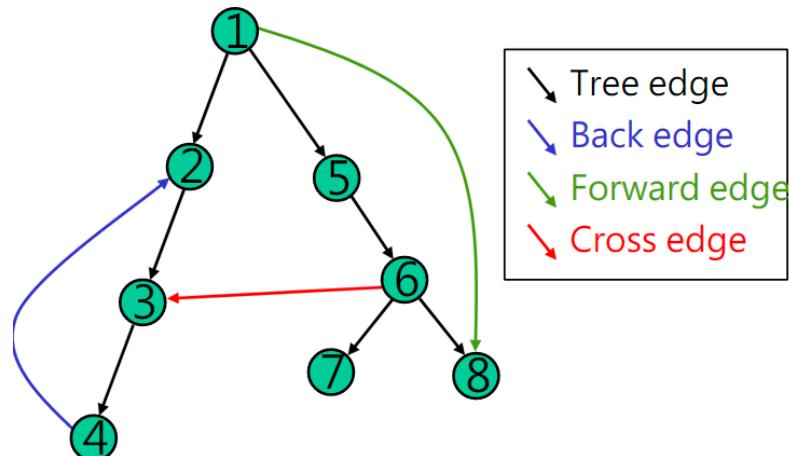


圖 9.1: DFS Tree (from wikipedia)

透過 DFS Tree，我們可以把某些圖演算法簡化成一種思考如何應對這四種不同類型的邊，在加以求得答案。特別注意的是，在無向圖中，只存在 Tree edge 與 Back edge，可以透過簡單反證法得到此結果，這個重要結論會直接應用在接下來的演算法上，使得演算法設計上更為簡單。

## 9.2 連通分量 Component

在本節中，我們探討在一般圖  $G = (V, E)$  上有關連通性的問題，此類問題包含：

- 圖上某些特殊點或邊的存在與否，會直接影響連通性
- 圖上由節點  $u$  出發到節點  $v$ ，是否存在一條（最短）路徑

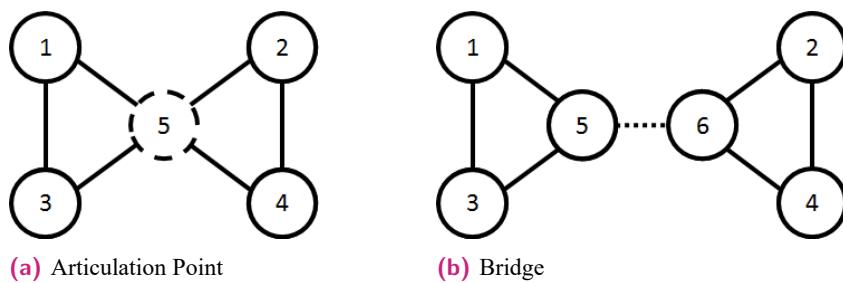


圖 9.2: Examples of articulation points and bridges

首先，我們介紹 articulation point(AP) 以及 bridge，中文翻譯分別為關節點與橋，無向圖中的 AP 和 Bridge 定義如下：

- (Articulation point) 圖  $G$  中，一個節點  $v \in V$  為 articulation point，若且唯若在  $G$  中移除  $v$  之後，會使原本為單一連通塊的  $G$  分離為兩個以上不連通的連通塊，如圖 9.2a 中的 5 號節點。
- (Bridge) 圖  $G$  中，一個邊  $e \in E$  為 bridge，若且為若在  $G$  中移除  $e$  之後，會使原本為單一連通塊的  $G$  分離為兩個以上不連通的連通塊，如圖 9.2b 中連接 5 號節點和 6 號節點的邊。

我們將在 9.2.1 小節中介紹以 DFS 方法來找出無向圖中所有 AP 與 bridge 的 Tarjan 演算法。

### 9.2.1 Tarjan's Algorithm for AP/Bridge

在 Robert Tarjan 於 1973 年提出用來尋找 AP 的線性時間演算法 (程式碼 9.2) 中，對於圖  $G = (V, E)$  上的所有節點  $v$ ，定義了兩個函數  $D(v)$  和  $L(v)$ ，如下：(假設圖  $G$  的 DFS Tree  $T$  以  $r$  點做為根)

- ( $D(v)$ ) Depth，在建立  $T$  時， $v$  點第一次被經過時的深度
- ( $L(v)$ ) Lowpoint，對於  $v$  點，其  $T$  中子樹內所有節點和這些節點在  $G$  上的鄰點，以及  $v$  本身的最低深度

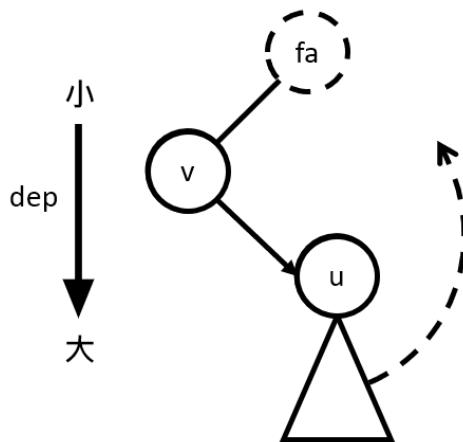


圖 9.3: Examples of Tarjan's algorithm for AP

而演算法的概念是，對每一個點，求從不通過父節點的路徑，能走回到的深度最小位置為何，參考圖 9.3，假設目前討論的點為  $v$ ， $fa$  是  $v$  在 DFS Tree 上的父節點。如果  $v$  能從某一個子樹找到一條通路到達  $fa$ ，讓虛線到達  $fa$ ，或其他深度更小的點，那麼這條路徑就會與  $fa, v$  形成一個環。而如果發現不論怎麼走，虛線只能到達  $v$ ，或是比  $v$  的 dep 更深的點，那麼很顯然的， $v$  就會是一個關節點。

藉由這樣的思路，我們可以得到以下結論：

- 根結點  $r$  是特例<sup>i</sup>，而  $r$  為 AP 僅當  $r$  在  $T$  中有兩個以上的子節點
- 對於除了  $r$  點以外的所有點  $v$ ， $v$  點在  $G$  上為 AP 的充要條件為其在  $T$  中至少有一個子節點  $w$  滿足  $D(v) \leq L(w)$
- 對於包含  $r$  在內的所有點  $v$  和  $v$  在  $T$  中的子節點  $w$ ，邊  $e(v, w)$  在圖  $G$  中為 bridge 的充要條件為  $D(v) < L(w)$ 。

其中上列的第三項資訊並不包含在 Tarjan 提出此演算法的論文中，我們另外在後面的演算法 9.1 紿出 Tarjan 於 1974 年首度提出用來尋找 bridge 的線性時間演算法。實作中，唯一要注意的地方大致上只有注意  $L[x]$  的取值，對於每一個非父親的點鄰居  $u$ ，都要取一次  $D[u]$ ，而如果  $u$  是一個未走過的點，表示  $(v, u)$  是 DFS Tree 的一個普通路徑，可以直接取用  $L[u]$  來得到  $u$  出發的所有路徑最淺的位置，有一點動態規劃的感覺。

```
1 void DFS(int v, int fa) { // 使用 DFS(v, v) 來呼叫函數
2     D[v] = L[v] = timestamp++;
3     int childCount = 0;
4     bool isAP = false;
5
6     for (int w:adj[v]) {
7         if( w==fa ) continue;
8         if ( !D[w] ) { // 用 D[w] = 0 當作沒走過
9             DFS(w,v);
10            childCount++;
11            if (D[v]<=L[w]) isAP = true; // 結論 2
12            if (D[v]< L[w]) edgeBridge.emplace_back(v, w); // 結論 3
13            L[v] = min(L[v], L[w]);
14        }
15        L[v] = min(L[v], D[w]);
16    }
17    if ( v == fa && childCount < 2 ) isAP = false; // 結論 1
18    if ( isAP ) nodeAP.emplace_back(v);
19 }
```

程式碼 9.2: Tarjan's Algorithm of finding AP and bridges

在有向圖上，AP 和 bridge 會根據定義的不同而有所區別，不過我們仍然可以透過 Dominator Tree 來解決此問題。

<sup>i</sup>因為他沒有父節點

### 演算法 9.1 Original Tarjan's Algorithm of finding bridge

- 1: 將圖  $G$  中的所有節點以 preorder traversal 的方式編號。
- 2: 以 postorder traversal 的順序處理每個節點  $v$ ：
  - 3: 計算  $v$  的子樹節點個數  $ND(v)$ ，即所有  $v$  的子節點  $w$  的  $ND(w)$  加總後再加 1
  - 4: 計算  $L(v)$ ：計算方式同程式碼 9.2 中的  $L(v)$
  - 5: 計算  $H(v)$ ：計算方式類似  $L(v)$ ，只是所有 min 的計算改成 max
  - 6: 檢查所有  $v$  的子節點  $w$ ，若  $L(w) = w$  且  $H(w) < w + ND(w)$ ，則  $v$  到  $w$  的邊是一座橋。

## 9.2.2 強連通分量 Strongly Connected Component

在一張有向圖中，如果兩個節點  $x$  和  $y$  間同時存在路徑  $\text{path}(x, y)$  和  $\text{path}(y, x)$ ，則我們說  $x$  和  $y$  位於同一個強連通分量 (Strongly Connected Component, SCC) 中，表示不論由  $x$  或  $y$  出發，都可以到達對方節點，也可以說  $x$  和  $y$  處在同一個有向環中。此外，如果  $x$  和  $y$  處在同一個強連通分量，且  $y$  和另一節點  $z$  也處在同一個強連通分量，則  $x$  和  $z$  必定也在同一個強連通分量內。

通常我們用強連通分量來歸納具有相同性質的節點，將一張有向圖  $G$  上，處在同一個強連通分量內的所有節點合併為一個點之後，產生的新圖  $G'$  必定是有向無環圖 (Directed Acyclic Graph, DAG)，我們稱此操作為縮點。

我們可以透過反證法得到這一個事實：如果  $G'$  中存在至少一個環，那麼同一個環上的所有節點肯定都處在同一個強連通分量中，這樣就違反了縮點時必須將同一個強連通分量內的所有節點合併為一個點這條規定，因此  $G'$  上肯定不會有環存在，也就是說， $G'$  肯定是一個 DAG。

比起原本的圖  $G$ ，簡化後的 DAG  $G'$  在許多演算法的規劃上會簡單許多，但是我們要如何計算強連通分量？以下我們介紹兩種常見的方法，時間複雜度皆為線性：Tarjan's SCC algorithm 和 Kosaraju's algorithm。

### Tarjan's SCC Algorithm

Tarjan 在圖論相關的領域發明很多演算法，包括這個 1972 年提出的演算法 (程式碼 9.3)。與前者找 AP 的演算法類似，如果一個節點  $v$  在 DFS 完成之後  $D(v)$  等於  $L(v)$ ，表示  $v$  與該 DFS 的子樹中的所有節點構成強連通分量，我們可以用一個堆疊 (stack) 來記錄這些節點。由於 Tarjan 的演算法是由 DFS Tree 的最末端開始找出強連通分量，因此會以縮點後 DAG 的拓撲排序 (topological sort) 逆序找出強連通分量。

要注意在有向圖中計算 Lowpoint 時，會出現無向圖中不存在的 Cross edge 與 Forward Edge，如圖 9.4。Forward Edge 對於計算 Lowpoint 沒有影響，而 Cross edge

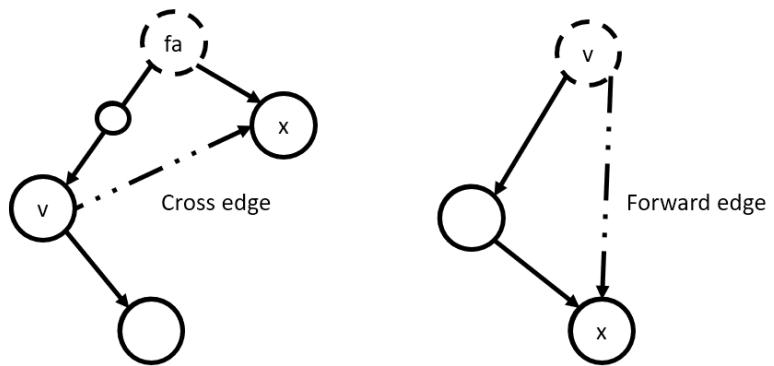


圖 9.4: Special case for Tarjan's algorithm

會導致原來的方法找到一個無法形成環的通路，因此在更新鄰點時，要加以判斷掉，而可以很簡單的紀錄該點是否在 Stack 內來處理即可。

```

1 void DFS(int v, int fa) { // 使用 DFS(v,v) 來呼叫函數
2     D[v] = L[v] = timestamp++;
3     st.push(v);
4     inSt[v] = true;
5
6     for (int w:adj[v]) {
7         if( w==fa ) continue;
8         if ( !D[w] ) { // 用 D[w] = 0 當作沒走過
9             DFS(w,v);
10            L[v] = min(L[v], L[w]);
11        }
12        else if (inSt[w]) { // 注意這個判斷
13            L[v] = min(L[v], D[w]);
14        }
15    }
16    if (D[v]==L[v]) {
17        int x;
18        do {
19            x = st.top();
20            st.pop();
21            inSt[x] = false;
22            contract[x] = sccID; /* 點x所在的SCC編號 */
23        } while (x!=v);
24        sccID++;
25    }
26 }
```

程式碼 9.3: Tarjan's Algorithm of SCC

Kosaraju 在 1978 年一篇未發表的論文中描述了一個用來計算 SCC 的線性時間演算法 (演算法 9.2) · 也就是 Kosaraju's algorithm 這個俗稱兩次 DFS 的方法 · 雖然它會因為比 Tarjan 的方法多遍歷了一次圖而導致較慢的執行速度 · 但它也會以完成縮點後 DAG 的拓樸排序找出強連通分量 · 可以在第二次查找的同時 · 直接把圖收縮完成 · 十分的方便 !

Kosaraju 演算法利用的性質是 · 如果有向圖  $G$  上的兩點  $x$  和  $y$  處在同一個強連通分量內 · 那麼在所有的邊方向顛倒後得到的新圖  $G^T$  中 ·  $x$  和  $y$  仍然處在同一個強連通分量內 ; 若  $x$  和  $y$  在  $G$  中處在不同的強連通分量中 · 那麼在  $G^T$  中它們同樣處在不同的強連通分量。我們在本節最後面的程式碼 9.4 紿出 Kosaraju 演算法的範例程式。

---

### 演算法 9.2 Kosaraju's Algorithm

---

- 1: 對  $G$  做一次 DFS · 以 postorder 順序將節點放入堆疊中。
  - 2: 由堆疊逐一取出節點 · 在  $G^T$  上由該取出的節點開始做 DFS · 所有遇到尚未在  $G^T$  上被經過的節點 · 都處在原圖  $G$  上的同一個強連通分量內。
- 

我們簡單的證明 Kosaraju 的正確性 : 假設在  $G$  上 DFS 時 · 節點  $y$  比節點  $x$  先進入堆疊 · 那麼在  $G^T$  上 DFS 時會先由節點  $x$  開始

Case 1 若在  $G^T$  上 DFS 時 · 可以由  $x$  到達  $y$  · 則代表在  $G$  上存在  $\text{path}(y, x)$

若  $G$  上不存在  $\text{path}(x, y)$  · 則一開始在  $G$  上的 postorder traversal 中節點  $y$  必須比節點  $x$  晚進入堆疊 · 違反假設 · 因此在這樣的狀況下  $G$  上必存在  $\text{path}(x, y)$  · 也就是說  $x$  和  $y$  存在於同一個強連通分量中。

Case 2 若在  $G^T$  上 DFS 時 · 無法由  $x$  到達  $y$  · 則代表在  $G$  上不存在  $\text{path}(y, x)$  · 那麼不論在  $G$  上是否存在  $\text{path}(x, y)$  ·  $x$  和  $y$  都不會存在於同一個強連通分量中 · 也不會影響演算法的正確性。

歸納所有 Case 1 的狀況 · 就能求出所有的強連通分量 · Case 2 則構成了完成縮點後的邊。最後這個證明還缺了一個部份 : 這個演算法所找出的強連通分量是最大的 · 此部份留給讀者思考。

```

1 void DFS(vector<int> *dG, int v, int k=-1){
2     visited[v] = true;
3     contract[v] = k;
4     for (int w:dG[v]){
5         if (!visited[w]) DFS(dG,w,k);
6         else if (contract[v]!=contract[w]); //存在邊contract[w]>=contract[v]
7     }
8     if (dG==G) st.push(v);
9 }
10
11 int Kosaraju(int N){
12     visited.clear();
13     for(int i=0; i<N; ++i) if(!visited[i]) DFS(G,i);
14     visited.clear();
15     while(!st.empty()){
16         if (!visited[st.top()]) DFS(GT, st.top(), sccID++);
17         st.pop();
18     }
19     return sccID;
20 }
```

**程式碼 9.4:** Kosaraju's Algorithm

### 9.2.3 2-SAT 問題與強連通分量

SAT 是 Satisfiability 的縮寫，此類問題的主體是以 AND 運算 (conjunction) 將許多 OR 運算式 (disjunction) 合併成的算式，目標則是要找出算式中的變數 (variables) 以何種布林值 (boolean) 組合代入時會使得算式結果為 true。目前一般性的 SAT 問題屬於 NP-Complete，沒有多項式時間可解決的方法。

如果在最多 OR 的子運算式中，最多只有  $K$  個變數，那麼我們稱這一個問題為 K-SAT 問題。目前已知所有的 SAT 問題都能化簡為 3-SAT 問題。而如果問題能進一步化簡為 2-SAT 問題，就可以利用 Krom's Algorithm 在多項式時間 (polynomial time) 內求出答案！

因為是用 AND 運算串接所有的 OR 的子運算式，因此如果有解答的話，所有的 OR 的子運算式計算結果都必需是 true。因此我們發現如果有像  $a \vee b$  這樣的算式，且已知  $a$  的解答為 false，則可以推斷  $b$  的解答一定是 true，反之亦然。總之答案的選擇是受到限制的！

我們可以構造一個圖來表示這些變數解答的依賴關係：對於每一個變數  $v$  建立兩個節點： $v$  和  $\neg v$ ，分別代表選擇  $v$  為 true 以及選擇  $v$  為 false 的狀況。

我們用有向邊  $x \rightarrow y$  表示如果選擇  $x$  為 true，那麼  $y$  就必需也是 true。在表 9.1 中列了一些常見的 case 以供參考，其中  $\oplus$  代表 xor 運算， $a \oplus b = (a \vee b) \wedge (\neg a \vee \neg b)$

表 9.1: Boolean Expressions

| 子運算式                 | 連接邊 1                    | 連接邊 2                    |
|----------------------|--------------------------|--------------------------|
| $a \vee b$           | $\neg a \implies b$      | $\neg b \implies a$      |
| $\neg a \vee b$      | $a \implies b$           | $\neg b \implies \neg a$ |
| $a \vee \neg b$      | $\neg a \implies \neg b$ | $b \implies a$           |
| $\neg a \vee \neg b$ | $a \implies \neg b$      | $b \implies \neg a$      |
| $a \oplus b$         | $a \iff \neg b$          | $b \iff \neg a$          |

這樣構造出來的圖有一個性質：對於  $x$  和  $\neg x$  必需恰有一個為 true，另一個為 false。如果存在一個變數  $x$ ，滿足  $x$  和  $\neg x$  位在圖中同一個強連通分量，則此 2-SAT 問題無解，因為不論  $x$  為 true 或 false，位在同一個強連通分量的  $\neg x$  都會因得到相同的答案而矛盾。反之，只要對於所有的變數  $x$ ，其節點  $x$  和  $\neg x$  都位在不同的強連通分量，則我們必定可以構造出一個可行的解答！

#### 9.2.4 練習題 Exercise

| 無向圖上的 AP   | TOI 2013 一階選訓營 |
|--|----------------|
| 給定一張節點數 $N$ 不超過 10000 的無向圖 $G = (V, E)$ ，要求算出 $G$ 中 articulation points 的數量。 |                |
| 註：UVa 315 是 $N$ 不超過 100，但題目要求完全相同的題目。  |                |

| 無向圖上的 AP  | UVa 10765 |
|---|-----------|
| 給定一張節點數 $N$ 不超過 10000 的無向圖 $G = (V, E)$ ，每個節點 $v$ 的價值為移除該點之後，圖上的連通塊數量。要求輸出價值前 $m$ 大的節點。 |           |

| 無向圖上的 bridge                                | UVa 796 |
|---|---------|
| 給定一張無向圖 $G = (V, E)$ ，求 $G$ 中所有為 bridge 的邊。 |         |

| 強連通分量  | UVa 11504 |
|--|-----------|
| 給定一些骨牌，且已知推倒每張骨牌會連動哪一些骨牌，問至少需推倒幾張骨牌才能使所有骨牌都倒下。 |           |

| 強連通分量   | UVa 11770 |
|---|-----------|
| 給定一張節點數 $N$ 不超過 10000 的有向圖 $G = (V, E)$ ，每個節點上都有一個燈，已知某些節點上的燈打開後會同時連動哪些節點的燈，問至少需打開幾個燈才能使所有燈都亮起。 |           |

2-SAT

UVa 11294

最多有 62 人要坐在長桌兩側，其中有一些人不得坐在長桌的同一側，問安排座位的方法。

### 9.3 支配樹 dominator tree

在一張有向圖  $G = (V, E)$  中選擇一個節點  $R$ ，對於任意節點  $x$ ，如果任何一條  $R$  到  $x$  的路徑都必須經過節點  $y$ ，我們稱  $y$  是  $x$  的支配點（有些文章會寫必經點），設  $\text{idom}[x]$  表示距離  $x$  最近的支配點（**最近支配點 immediate dominator**）。

對於所有節點  $x$ ，連接  $\text{idom}[x]$  到  $x$  的有向邊後會形成一棵以  $R$  為根的樹，稱為 **支配樹 (dominator tree)**。

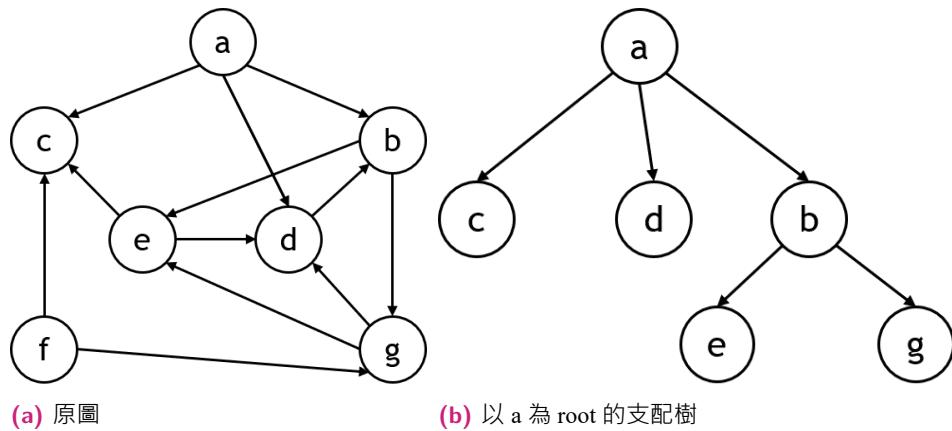


圖 9.5：支配樹範例

#### 9.3.1 支配樹性質

1. 支配樹是以  $R$  為根的樹。
2. 對任意節點  $x$ ，支配樹上  $x$  的所有祖先就是  $x$  的所有支配點（包含  $x$  自己）。
3. 支配樹的所有節點 = 原圖  $G$  中從  $R$  出發能走到的所有節點。

#### 9.3.2 簡化問題：有向無環圖 (DAG)

如果圖  $G$  是有向無環圖，我們可以按造拓樸排序的順序構建支配樹。假設當前我們構造到拓樸順序中第  $k$  個節點  $x$ ，那麼拓樸順序中第  $1 \sim k - 1$  的節點的  $\text{idom}$  都求出來了。現在考慮連向  $x$  的所有節點，對於這些節點我們求出它們在當前支配樹上的 **最近共同祖先**  $y$  就是  $\text{idom}[x]$ 。

這裡的最近共同祖先可以用未來會教到的倍增法在  $\mathcal{O}(\log|V|)$  的時間完成，因此總複雜度是  $\mathcal{O}((|V| + |E|) \log|V|)$ 。

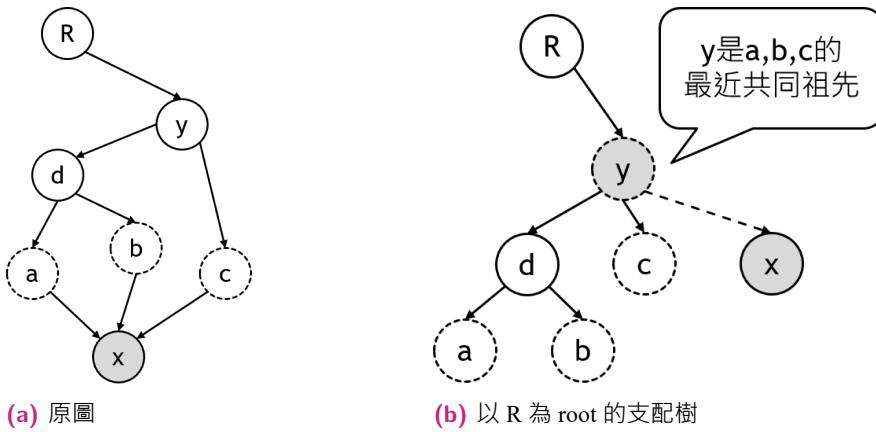


圖 9.6: 有向無環圖的支配樹

### 9.3.3 一般圖的支配樹 (Lengauer-Tarjan 演算法)

一般圖的情況相對複雜，我們希望能像前一小節（連通分量 Component）一樣，使用**深度優先搜索 (DFS)**的技巧來找出支配樹。但是要直接在 DFS 過程中直接求出每個節點的 idom 是困難的，因此在本節介紹的 **Lengauer-Tarjan 演算法**是先求出**半支配點 semi-dominator**後再透過半支配點計算出 idom。

半支配點 semi-dominator (sdom)

我們從節點  $R$  開始做 DFS，在 DFS 的過程中，經過的邊和節點會形成一棵樹，稱為**DFS 樹 (DFS Tree)**。對於每個節點  $x$ ，設  $\text{dfn}[x]$  表示節點  $x$  在 DFS 過程中的時間戳記，另外定義  $\text{id}[\text{dfn}[x]] = x$ 。

節點  $x$  的半支配點  $\text{sdom}[x]$  定義如下：

$$\begin{aligned}\text{sdom}[x] = \\ \text{id}[\min\{\text{dfn}[y] : \forall \text{path}(y \rightarrow x) \text{ 滿足除了 } y, x \text{ 外所有節點的 } \text{dfn} > \text{dfn}[x]\}]\end{aligned}$$

圖 9.7 中粗邊為 DFS 樹上的邊，節點旁邊的數字代表  $\text{dfn}$ ，本小節 9.3 中所有圖片皆如此表示。

經過觀察以及證明，可以確認  $\text{sdom}$  有以下性質：

1.  $\text{sdom}[x]$  一定是  $x$  在 DFS 樹上的祖先。

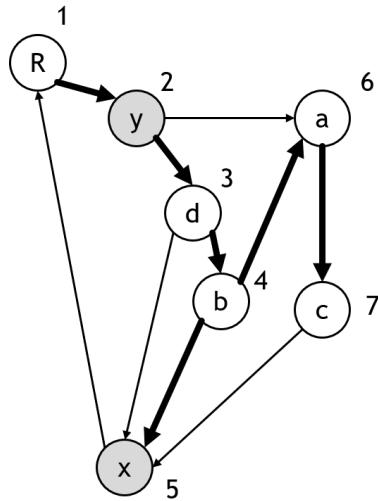


圖 9.7:  $x$  的半支配點  $y$  (路徑  $y \rightarrow a \rightarrow c \rightarrow x$  滿足條件)

*Proof.* 用反證法證明。設  $\text{sdom}[x]$  不是  $x$  的祖先，他們的最近共同祖先是  $y$ ，可以發現  $y \rightarrow \text{sdom}[x] \rightarrow x$  的路徑滿足半支配點的條件且  $\text{dfn}[y]$  更小，矛盾。

□

2.  $\text{sdom}[x]$  不一定是  $x$  的支配點。

*Proof.* 圖 9.8 提供了一個反例。

□

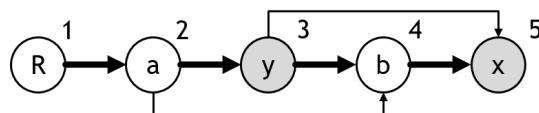


圖 9.8:  $\text{sdom}[x]$ (這裡  $\text{sdom}[x]=y$ ) 不一定是  $x$  的支配點

3. DFS 樹上  $\text{sdom}[x]$  到  $x$  路徑上的節點 (不包括  $x, \text{sdom}[x]$ ) 必然不是  $x$  的支配點。

*Proof.* 顯然這些節點是可以被其他路徑繞過去的。

□

4. 保留 DFS 樹上的所有邊，對於每個節點  $x$  增加邊  $(\text{sdom}[x], x)$ ，產生的新圖稱為  $G'$ 。可以發現  $G'$  是有向無環圖且每個節點的支配點和原圖  $G$  相同！

*Proof.* 可以透過  $\text{sdom}$  性質 1 知道  $G'$  是 DAG。

$G'$  的 DFS 樹以及每個節點的  $\text{sdom}$  會與原圖  $G$  相同，介紹  $\text{idom}$  性質的單

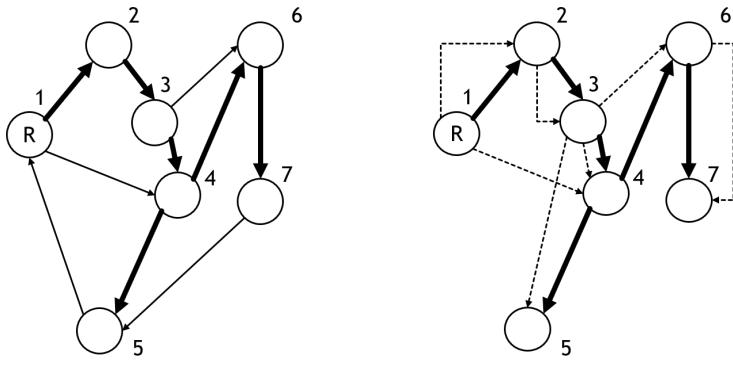


圖 9.9: 透過 DFS 樹以及 sdom 構造有向無環圖

元將會證明找出支配樹只需要這些資訊，因此  $G'$  的支配樹等於  $G$  的支配樹。 $\square$

顯然如果能快速計算出每個點的 sdom，就能透過 sdom 性質 4 快速的求出  $G$  的支配樹。

找出半支配點

對於圖  $G$  中每條連向節點  $x$  的邊  $(y, x)$ ，可以將其分成兩類找出  $\text{sdom}[x]$  的所有可能。

1.  $\text{dfn}[y] < \text{dfn}[x]$  (case 1)

這種情況根據定義存在  $\text{sdom}[x]=y$  的可能性。

2.  $\text{dfn}[y] > \text{dfn}[x]$  (case 2)

設  $z$  是 DFS 樹上  $y$  的祖先 (包含  $y$ ) 且  $\text{dfn}[z]>\text{dfn}[x]$ ，根據定義存在  $\text{sdom}[x] = \text{sdom}[z]$  的可能性。

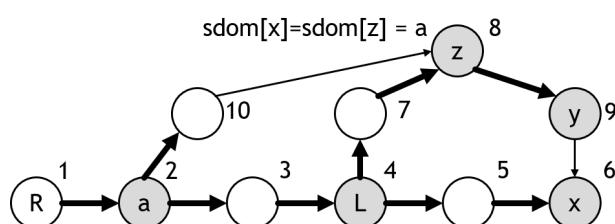


圖 9.10: case 2 的情況

找出所有可能的  $\text{sdom}[x]$  中  $\text{dfn}$  最小的那個就是答案了。

根據這些性質可以發現如果要計算 dfn 是  $i$  的節點，那 dfn 比  $i$  大的節點的 sdom 必須先算出來，按造 dfn 由大到小的順序分別計算就能找出每個點的 sdom。

設  $\text{pa}[x]$  表示節點  $x$  在 DFS 樹上的 **parent**。

對於 case 2 的邊，利用下面虛擬碼中的 `find_best_z` 沿著  $\text{pa}[y]$  遞迴往上爬直到  $\text{dfn}[y] < \text{dfn}[x]$  為止，經過的節點中 sdom 的 dfn 最小的節點就是我們要的  $z$ 。而對於 case 1，會發現直接使用 case 2 的計算方式找出來的  $z = y$ ，因此把 case 1 也當作 case 2 處理可以省下不少程式碼。

```
1 def calculate_sdom():
2     for( i : dfn 由大到小的順序 )
3         x = id[i];
4         for( y : (y,x) in E )
5             z = find_best_z(y, x);
6             if( dfn[sdom[x]] > dfn[sdom[z]] )
7                 sdom[x] = sdom[z];
8
9     def find_best_z(y, x):
10        if( dfn[y] <= dfn[x] )
11            return y;
12        var tmp = find_best_z(pa[y], x);
13        if( dfn[sdom[tmp]] < dfn[sdom[y]] )
14            return tmp;
15        return y;
```

程式碼 9.5: 樸素方法計算 sdom

整個演算法的複雜度是  $\mathcal{O}(N^2)$ ，效能上並不符合我們的要求。

帶權並查集 (“static tree” disjoint set data structure)

這裡的帶權並查集是中國的選手自己發明的名詞，括號中的英文名稱是筆者在 Tarjan 的論文中找到的 (Finding dominators via disjoint set union)，透過這個資料結構可以降低 `find_best_z` 的複雜度。

設  $\text{anc}[x]$  表示節點  $x$  在並查集樹上的 parent，初始化設  $\text{anc}[x] = x$ ，為了方便起見我們先不考慮路徑壓縮。

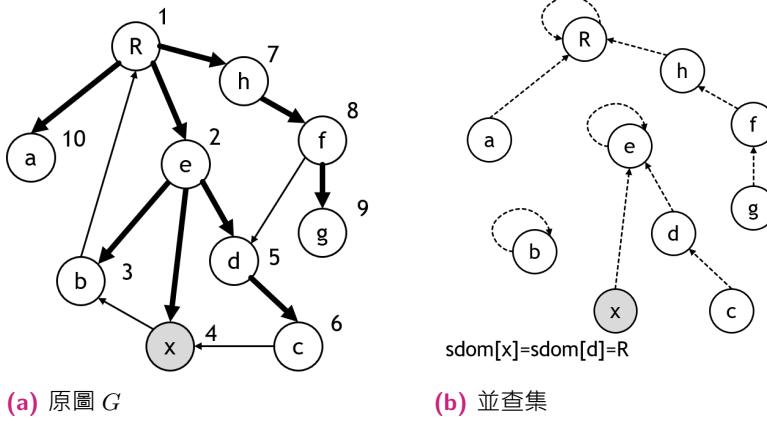


圖 9.11: 計算完  $\text{sdom}[x]$  後的並查集森林

每次計算完節點  $x$  的  $\text{sdom}$  後，把  $\text{anc}[x] = \text{pa}[x]$ 。這樣原本的  $\text{find\_best\_z}(y, x)$  遍迴會經過的節點等於當前並查集樹上  $y$  的所有祖先（包含  $y$ ），如圖 9.11 所示，因此可以寫出以下的虛擬碼：

```

1 def calculate_sdom():
2     for( x : 圖G中所有節點 )
3         anc[x] = x;
4     for( i : dfn由大到小的順序 )
5         x = id[i];
6         for( y : (y,x) in E )
7             z = find_best_z(y);
8             if( dfn[sdom[x]] > dfn[sdom[z]] )
9                 sdom[x] = sdom[z];
10            anc[x] = pa[x];
11
12 def find_best_z(y):
13     if( y = anc[y] )
14         return y;
15     var tmp = find_best_z(anc[y]);
16     if( dfn[sdom[tmp]] < dfn[sdom[y]] )
17         return tmp;
18     return y;

```

程式碼 9.6: 不考慮路徑壓縮

現在來考慮有路徑壓縮的情況，路徑壓縮後並查集樹的形狀會改變，我們需要一個額外的陣列來紀錄路徑壓縮前的資訊。設  $\text{best}[y] = \text{find\_best\_z}(y)$ ，我們在路徑壓縮的過程維護經過的節點的  $best$ ：

```

1 def findBest(y):
2     if( y == anc[y] )
3         return y;
4     var tmp = findBest(anc[y]);
5     if( dfn[sdom[best[y]]] > dfn[sdom[best[anc[y]]]] )
6         best[y] = best[anc[y]];
7     anc[y] = tmp;
8     return anc[y];

```

程式碼 9.7: 路徑壓縮的過程維護經過的節點的 *best*

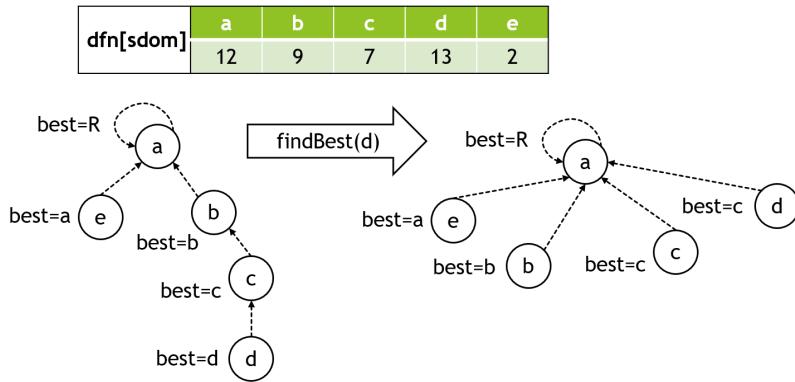


圖 9.12: 路徑壓縮

這樣 *findBest* 操作的均攤複雜度會是  $\mathcal{O}(\log N) \cdot \text{calculate\_sdom}$  就可以輕鬆做到  $\mathcal{O}(|E| + |V|) \log|V|$  :

```

1 def calculate_sdom():
2     for( x : 圖G中所有節點 )
3         anc[x] = best[x] = x;
4     for( i : dfn由大到小的順序 )
5         x = id[i];
6         for( y : (y,x) in E )
7             findBest(y);
8             z = best[y];
9             if( dfn[sdom[x]] > dfn[sdom[z]] )
10                sdom[x] = sdom[z];
11                anc[x] = pa[x];

```

程式碼 9.8:  $\mathcal{O}(|E| + |V|) \log|V|$  計算 sdom

### 最近支配點 immediate dominator (idom)

sdom 性質 4 提到，只要有 DFS 樹和每個節點的 sdom，就可以唯一確定每個節點的 idom，透過分析 idom 的性質可以證明這件事情：

1. DFS 樹上  $\text{idom}[x]$  的深度不大於  $\text{sdom}[x]$  的深度。

*Proof.* 用反證法證明。如果  $\text{idom}[x]$  的深度大於  $\text{sdom}[x]$  的深度，那麼存在路徑  $R \rightarrow \text{sdom}[x] \rightarrow x$  可以不經過  $\text{idom}[x]$ ，矛盾。  $\square$

2. DFS 樹上  $\text{idom}[x]$  到  $x$  路徑上的任何節點（不包括  $x, \text{idom}[x]$ ）必然不是  $x$  的支配點。

*Proof.* 根據  $\text{idom}$  的定義顯然成立。  $\square$

3. 設節點  $t$  是 DFS 樹上  $\text{sdom}[x]$  到  $x$  路徑上（不包括  $\text{sdom}[x]$ ）的節點中  $\text{sdom}$  的  $\text{dfn}$  最小的點，以下性質必然成立：

- a)  $\text{idom}[x] = \text{idom}[t]$ 。

*Proof.* 考慮將  $\text{idom}[t]$  從圖上刪除，這樣一來就無法從  $R$  走到  $\text{idom}[t]$  到  $t$  路徑上的所有點，對於  $t$  到  $x$  路徑上的所有點，其  $\text{sdom}$  的深度都不小於  $\text{idom}[t]$ ，故沒有任何路徑能到達  $x$ ，得證。  $\square$

*Proof.* 接著證明任意  $x$  的支配點其深度必不大於  $\text{idom}[t]$  的深度。

根據  $\text{idom}$  性質 2，DFS 樹上  $\text{idom}[t]$  到  $t$  路徑中的任意節點（不包含  $t, \text{idom}[t]$ ）必然不是  $x$  的支配點；又根據  $\text{idom}$  性質 1，可知  $\text{idom}[t]$  到  $\text{pa}[x]$  路徑上任何一點都不是  $x$  的支配點，得證。  $\square$

- b) 如果  $\text{sdom}[x] = \text{sdom}[t]$ ，則  $\text{idom}[x] = \text{idom}[t] = \text{sdom}[x]$ 。

*Proof.* 透過  $\text{sdom}$  性質 3，只需要證明  $\text{sdom}[x]$  是  $x$  的支配點就行了。

用反證法證明。若  $\text{sdom}[x]$  不是  $x$  的支配點，表示有一條路徑不經過  $\text{sdom}[x]$  也能到達  $x$ ，故必然存在節點  $t'$  位於  $\text{sdom}[x]$  到  $x$  路徑上且  $\text{dfn}[\text{sdom}[t']] < \text{dfn}[\text{sdom}[t]]$ ，矛盾。  $\square$

雖然已經找到一個夠快的方法來求支配樹，但將原圖轉成 DAG 再計算的方式過於複雜，這裡介紹直接利用  $\text{sdom}$  去計算  $\text{idom}$  的方法：

設集合  $\text{sdomSet}[y]$  存放那些 **sdom 是  $y$  且還沒算出 idom** 的節點。我們每次依照  $\text{dfn}$  由大到小的順序找出  $\text{sdom}[x]$  後就將節點  $x$  加入至集合  $\text{sdomSet}[\text{sdom}[x]]$  中，之後利用  $\text{idom}$  性質 3 計算  $\text{sdomSet}[\text{pa}[x]]$  中所有節點的  $\text{idom}$ ，這樣就能保證每個節點的  $\text{idom}$  都會被計算出來。

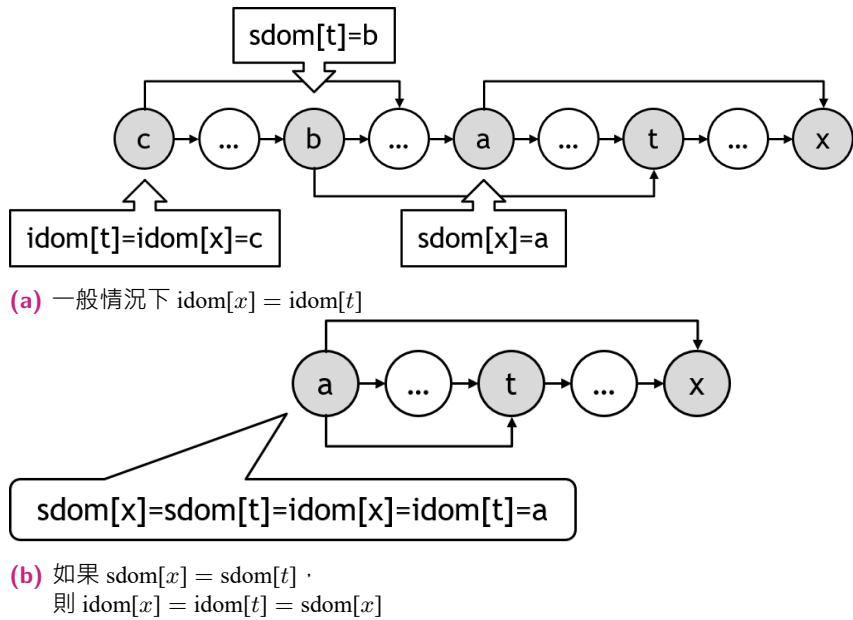


圖 9.13:  $\text{idom}[x]$  只有這兩種可能

計算過程中找出節點  $t$  的方式使用和  $\text{sdom}$  相同的帶權並查集。注意如果遇到節點  $w$  不滿足  $\text{idom}$  性質 3.a) 的情況，此時透過  $\text{idom}$  性質 3.a) 可知  $\text{idom}[w] = \text{idom}[t]$ ，但  $\text{idom}[t]$  可能還沒被計算出來，先暫時記錄  $\text{idom}[w] = t$ ，等到所有節點都處理過之後，依照  $\text{dfn}$  由小到大的順序就可以算出這些點的  $\text{idom}$ 。

找出  $\text{idom}$  花費的時間複雜度和找出  $\text{sdom}$ ，如此一來就能確認整個演算法的總時間複雜度是  $\mathcal{O}(|E| + |V|) \log |V|$ 。有論文提出  $\mathcal{O}(|E| + |V|)$  的方法，但較難實作故本單元並不會介紹。

這些步驟的詳情過程請參考完整程式碼。

#### Lengauer-Tarjan Algorithm 完整程式碼

```

1  class dominatorTree{
2      int n, dfnCnt;
3      vector<vector<int>> G, rG, sdomSet;
4      vector<int> dfn, id, pa, sdom, idom;
5      vector<int> anc, best;
6      int findBest(int x){
7          if( x==anc[x] ) return x;
8          int tmp = findBest(anc[x]);
9          if( dfn[sdom[best[x]]] > dfn[sdom[best[anc[x]]]] )
10             best[x] = best[anc[x]];
11             anc[x] = tmp;
12     }
13     void DFS(int x){

```

```

14     id[ dfn[x] = ++dfnCnt ] = x;
15     for(int y: G[x]){
16         if(dfn[y]) continue;
17         pa[y] = x, DFS(y);
18     }
19 }
20 public:
21     void init(int _n){
22         n = _n;
23         G.clear(), G.resize(n+1);
24         rG.clear(), rG.resize(n+1);
25     }
26     void addEdge(int x,int y){
27         G[x].emplace_back(y);
28         rG[y].emplace_back(x);
29     }
30     vector<vector<int>> buildTree(int root){
31         dfnCnt = 0;
32         sdomSet = vector<vector<int>>(n+1);
33         dfn = id = pa = sdom = idom = vector<int>(n+1);
34         iota(sdom.begin(), sdom.end(), 0);
35         anc = best = sdom;
36         DFS(root);
37         for(int i = dfnCnt; i>1; --i){
38             int x = id[i];
39             // 找出 sdom[x]
40             for(int y: rG[x]) if(dfn[y]){
41                 findBest(y);
42                 int z = best[y];
43                 if( dfn[sdom[x]] > dfn[sdom[z]] )
44                     sdom[x] = sdom[z];
45             }
46             anc[x] = pa[x];
47             sdomSet[sdom[x]].emplace_back(x);
48             for(int w: sdomSet[pa[x]]) {
49                 // 找出 idom[w]
50                 findBest(w);
51                 int t = best[w];
52                 if(sdom[t]==pa[x]) idom[w] = pa[x];
53                 else idom[w] = t; // idom性質3.1
54                 // 此時還不知道idom[t]是誰
55                 // 先記錄 idom[w] = t 之後在處理
56             }
57             sdomSet[pa[x]].clear(); // 算過的點從集合中移除(否則會TLE)
58         }
59         vector<vector<int>> tree(n+1);
60         for(int i = 2; i<=dfnCnt; ++i){
61             int w = id[i];
62             // 處理 idom性質3.1 idom[w] = t
63             if(idom[w]!=sdom[w]) idom[w] = idom[idom[w]];
64             tree[idom[w]].emplace_back(w); // 順便將樹建起來

```

```

65     }
66     return tree;
67 }
68 }

```

程式碼 9.9: Lengauer-Tarjan Algorithm

### 9.3.4 練習題 Exercise

小 P 的煩惱

bzoj 3281

給一張  $n(n \leq 100000)$  個點  $m \leq 200000$  條邊有向無環圖表示地圖 (邊的權重代表路徑長度) 以及起終點  $s, t$ ，我們稱  $s$  到  $t$  無論走任何路徑一定會經過的邊叫做橋，橋是非常不安全的，因此你帶了兩條長度為  $L$  的安全繩，安全繩可以覆蓋在多條連續的邊上，橋上被安全繩覆蓋的區域會是安全的。問你覆蓋完兩條安全繩後  $s$  到  $t$  的路徑還不安全區域的長度總和是多少？

災難

bzoj 2815

求出支配樹後計算每個點的子樹的大小。

L. Useful Roads

2014-2015 ACM-ICPC, NEERC, Southern Subregional Contest

給你  $n(n \leq 2 \times 10^5)$  個點  $m(m \leq 2 \times 10^5)$  條邊的有向圖，請你輸出所有邊滿足該邊在任何一條編號 1 的點做為起點的簡單路徑上。

Counting on a directed graph

Codechef GRAPHCNT

給你  $n(n \leq 10^5)$  個點  $m(m \leq 5 \times 10^5)$  條邊的有向圖，請你計算有多少個  $(x, y)$  pair 滿足編號 1 的點到編號  $x, y$  的點存在一條路徑，且這兩條路徑上除了編號 1 的點外沒有其他相同的點。

Graph Challenge

Codechef DAGCH

給你一張有向圖，每個點的編號就是他的 dfn，對於每個節點  $x$ ，你要找出有多少節點的 sdom 是  $x$ 。

Bytelandian Information Agency

SPOJ BIA

給你一張有向圖，和一個起點，保證起點能走到圖上所有點。問那些點從圖上移除後，會使得剩下的點中有起點無法走到的點。(模板題)

## 9.4 樹 Tree

在圖論的領域裡，我們將一張任意兩點間皆存在唯一一條路徑 (path) 的無向圖稱之為樹 (tree)，也可以說樹就是一張無環且連通的無向圖。在本節中，我們探討有關樹重心 (tree centroid)、樹分治 (divide & conquer on trees)、最近共同祖先 (lowest common ancestor, LCA)、樹序列化 (Euler Tour Technique) 和樹鏈剖分 (heavy-light decomposition) 的問題。

在這五個主題中，樹重心通常被作為輔助解樹分治的相關題目；樹序列化和樹鏈剖分的技巧雖然可以都用來解最近共同祖先的題目，但在最近共同祖先一節中程式碼 9.13 所使用的方法卻可以做到許多樹序列化所不能做到的事，使用上也比樹鏈剖分算法更有彈性。

### 9.4.1 樹重心 Tree Centroid

在一棵有  $n$  個節點的樹  $T$  上，樹重心的定義為滿足這個條件的節點：將此節點自  $T$  移除後， $T$  會被分為多棵樹，這些樹的節點數都不超過  $n$  的一半。其性質如下：

- 滿足樹重心定義的節點至少有一個，我們在定理 9.1 紿出證明。
- 滿足樹重心定義的節點不超過兩個，且當有兩個節點同時為樹重心時，它們必定相鄰，我們在定理 9.2 紿出證明。
- 若將兩棵樹通過一條邊相連得到一個新的樹，那麼新的樹的重心必定在連接原先兩棵樹各自重心的路徑上。
- 在一棵樹上添加或刪除一個葉節點，它的重心最多只移動一條邊的距離。

我們在本節最後面的程式碼 9.10 紿出尋找樹重心節點的範例程式。

**定理 9.1.** 在一棵有  $n$  個節點的樹  $T$  上，滿足樹重心定義的節點至少有一個。

*Proof.* 我們先選擇  $T$  上一節點  $r_0$  作為  $T$  的根，並假設所有的節點  $r_i$  其最大子樹的根為  $r_{i+1}$ ，且  $r_k$  為一葉節點。現在我們以反證法來證明：假設任意一節點  $r_i$  都不滿足樹重心定義。

我們首先以數學歸納法證明在這樣的情況下，以節點  $r_i$  作為  $T$  的根時，以節點  $r_{i+1}$  作為子樹根的子樹大小至少為  $\frac{n}{2} + 1$ 。

- 首先，由於節點  $r_0$  不滿足樹重心定義，因此以  $r_1$  作為子樹根的子樹，其大小至少為  $\frac{n}{2} + 1$ ，也就是說，當  $T$  以  $r_1$  作為根時，以  $r_0$  作為子樹根的子樹，其大小至多為  $\frac{n}{2} - 1$ 。
- 現在我們假設對所有的  $c \geq 1$  都有這樣的事實：當  $T$  以  $r_c$  作為根時，以  $r_{c-1}$  作為子樹根的子樹，其大小至多為  $\frac{n}{2} - 1$ 。當  $T$  以  $r_c$  作為根時，其最大子樹的根顯然只可能為  $r_{c-1}$  或  $r_{c+1}$ ，而由於  $r_c$  不滿足樹重心的定義，因此其最大子樹的大小至少為  $\frac{n}{2} + 1$ ，我們可以推得此時  $r_c$  的最大子樹的子樹根為  $r_{c+1}$ ，且其子樹大小至少為  $\frac{n}{2} + 1$ 。

這樣一來我們就得到：以節點  $r_{k-1}$  作為  $T$  的根時，其最大子樹的子樹根為  $r_k$ ，且其子樹大小至少為  $\frac{n}{2} + 1$ ，但我們知道  $r_k$  為一葉節點，這樣一來就有  $1 \geq \frac{n}{2} + 1$ ，而這是不可能的，因此原先的假設“任意一節點  $r_i$  都不滿足樹重心定義”是錯誤的，肯定有某個節點  $r_i$  滿足樹重心的定義。  $\square$

**定理 9.2.** 在一棵有  $n$  個節點的樹  $T$  上，滿足樹重心定義的節點至多有兩個，且它們必定相鄰。

*Proof.* 若  $n < 3$  則證明完成，我們只需要考慮  $n \geq 3$  的狀況。假設節點  $r$  為  $T$  的一個重心，以  $r$  作為  $T$  的根時，其  $k$  個子節點為  $r_i$  ( $1 \leq i \leq k$ )，且對應的子樹為  $T_i$ ，我們假設對於所有的  $1 \leq i < k$ ，有  $|T_i| \geq |T_{i+1}|$ ，其中  $|T|$  代表  $T$  的節點數。

由於  $r$  為  $T$  的一個重心，我們可以知道  $\frac{n}{2} \geq |T_1| \geq |T_2| \geq \dots \geq |T_k|$ ，且由於  $1 + i|T_i| \leq 1 + \sum_{c=1}^i |T_c| \leq 1 + \sum_{c=1}^k |T_c| = n$ ，我們可以得知若  $i \geq 2$ ， $|T_i| < \frac{n}{2}$ ，這樣一來，若以任意  $T_i$  ( $i \geq 2$ ) 中的任一節點作為  $T$  的根，其最大子樹的節點數都至少為  $n - |T_i| > \frac{n}{2}$ ，因此所有滿足  $i \geq 2$  的  $T_i$  中的任一節點都不可能為樹重心。

現在我們只需要考慮  $T_1$  中的節點，當  $T$  以  $r_1$  為根時，以  $r$  作為子樹根的子樹大小為  $n - |T_1| \geq \frac{n}{2}$ ，這使得  $r_1$  滿足樹重心定義的充要條件為  $|T_1| = \frac{n}{2}$ 。

我們進一步假設  $r_1$  的  $k'$  個子節點為  $x_i$  ( $1 \leq i \leq k'$ )，且對應的子樹為  $U_i$ 。顯然我們可以推得以任意  $U_i$  中的任一節點作為  $T$  的重心時，其最大子樹的大小至少為以  $r_1$  為子樹根的子樹大小  $n - |T_1| + 1 \geq \frac{n}{2} + 1$ ，因此在  $T_1$  中，除了  $r_1$  以外的節點都不可能是  $T$  的重心。

綜上所述， $T$  上除  $r$  以外的所有節點中，僅有與  $r$  相鄰的節點  $r_1$  可以是重心。  $\square$

```

1 pair<int,int> Tree_Centroid(int v, int pa){
2     // return (最大子樹節點數，節點ID)
3     sizeSubT[v] = 1;
4     pair<int,int> res(INT_MAX, -1);
5     int mx = 0;
6     for(size_t i=0; i<adj[v].size(); ++i){
7         int x = adj[v][i];
8         if (x==pa) continue;
9         res = min(res, Tree_Centroid(x, v));
10        subTsize[v] += subTsize[x];
11        mx = max(mx, subTsize[x]);
12    }
13    res = min(res, make_pair(max(mx, n-subTsize[v]), v));
14    return res;
15}

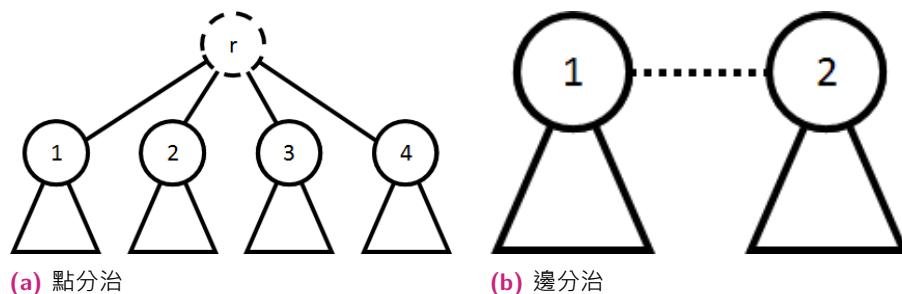
```

**程式碼 9.10:** Finding a tree centroid

#### 9.4.2 樹分治 Divide and Conquer on Trees

通常與樹上路徑相關的問題，我們可以用分治 (divide and conquer) 的方法來解決，目前常見的分治法有兩種：點分治與邊分治。

- (點分治) 首先選取樹  $T$  上的一個節點  $r$  作為根，接著以遞迴方式處理以  $r$  的子節點為根的子樹，如圖 9.14a。
- (邊分治) 首先選取樹  $T$  上的一個邊  $e$ ，接著以遞迴方式處理  $e$  兩邊互不相交的子樹，如圖 9.14b。



**圖 9.14:** 樹分治的兩種方法

然而目前與邊分治相關的研究及方法尚未成熟，仍舊存在許多問題，因此本節不討論其相關問題，僅討論點分治方法。

對於一棵樹  $T$  上的任一路徑  $(x_1, x_2)$  和任一節點  $v$  來說，其關係只有兩種：節點  $v$  是否處在路徑  $(x_1, x_2)$  上。若節點  $v$  不在路徑  $(x_1, x_2)$  上，那麼以節點  $v$  作為

$T$  的根時，路徑  $(x_1, x_2)$  必定只存在於某個以  $v$  的子節點為根的子樹內，與  $v$  本身和以其它  $v$  的子節點為根的子樹毫無交集。這樣的特性使得我們在處理相關問題時，得以將  $T$  遞迴分為 (divide) 許多子樹 (子問題) 來處理，而後再將這些結果合併 (conquer) 得到最終的答案。

在這樣的方法下，決定演算法效率的關鍵往往為遞迴深度所左右，而最糟的情況會出現在樹  $T$  實際上為一長鏈，且每次被選為樹根的節點都是此長鏈的一個端點，這樣的遞迴深度是  $O(n)$ ，並沒有任何的優勢。由於遞迴深度會被根節點的子樹大小影響，因此為了避免這樣的問題，被選擇為樹根的節點應該滿足其各個子樹中，各自節點數愈少愈好，而最好的節點選擇即為樹重心，尋找樹重心的相關算法可以在 9.4.1 節找到。

現在我們分析最壞的遞迴深度：由於每次都選擇樹重心作為根節點，因此每一步，我們都能將節點數減少至少一半，這樣的話遞迴深度必然不超過  $\log_2 n$ 。

以下我們在程式碼 9.11 紹出樹分治的範例程式。

```

1 int TreeDC(int v){
2     int c = 以節點v為根的子樹的樹重心;
3     int ans = Compute(c); // 計算與 c 有關的路徑數
4     visited[c] = true; // 將重心標記為不能走
5     for (size_t i=0; i<adj[c].size(); ++i){
6         int x = adj[c][i];
7         if(visited[x]) continue;
8         ans += TreeDC(x); //遞迴分治
9     }
10    return ans;
11 }
```

程式碼 9.11: 樹分治

### 9.4.3 最近共同祖先 Lowest Common Ancestor

在一棵有  $n$  個節點的樹  $T$  上，兩節點  $x_1$  和  $x_2$  之間最近共同祖先 (Lowest Common Ancestor, LCA) 節點  $a$  的定義是：假設  $T$  的根為節點  $r$ ，則在路徑  $(x_1, r)$  和路徑  $(x_2, r)$  都出現的所有節點中，最靠近  $x_1$  和  $x_2$  的節點。

這樣的節點  $a$  滿足一個性質：兩節點  $x_1$  和  $x_2$  之間的路徑總長為  $d(x_1, x_2) = d(r, x_1) + d(r, x_2) - 2d(r, a)$ 。而這也是競賽中一個常見、且與樹和樹上路徑相關的問題。

我們在程式碼 9.12 中給出最簡單的尋找兩節點間最近共同祖先的方法，其中  $D(v)$  代表在路徑  $(r, v)$  上邊的數量， $P(v)$  代表  $v$  在  $T$  上的父節點。

```

1 int LCA(int u, int v){
2     while(D[u]>D[v]) u = P[u];
3     while(D[u]<D[v]) v = P[v];
4     while(u!=v){
5         u = P[u];
6         v = P[v];
7     }
8     return u;
9 }
```

**程式碼 9.12:** Original LCA finding algorithm

程式碼 9.12 看似簡單易寫，但每次詢問時，最糟的時間複雜度可能是  $O(n)$ 。當  $n$  和詢問次數  $q$  很大時，這樣的算法效率非常糟糕。

我們考慮可能的加速：假設已知  $D(x_1)$  比  $D(x_2)$  大上許多，那麼是否能讓  $x_1$  這端在往所求的節點  $a$  前進時一次多走一些距離？

答案是可以的，考慮將所需的距離寫成二進位，例如  $7 = (111)_2$ ，我們可以將原本要往上走 7 次 1 步的距離切成 4 步 +2 步 +1 步，那麼要如何實作呢？我們在程式碼 9.13 紿出範例程式，其中  $D(v)$  代表在路徑  $(r, v)$  上邊的數量， $P(v, c)$  代表由  $v$  開始向  $r$  走  $2^c$  步的節點，也就是  $v$  在  $T$  上第  $2^c$  層的父節點。

```

1 int LCA(int u, int v){
2     if (D[u]>D[v]) swap(u, v);
3     int s = D[v]-D[u];
4     for (int i=0; i<=lgN; ++i) if (s&(1<<i)) v = P[v][i];
5     if (u==v) return v;
6     for (int i=lgN; i>=0; --i) if (P[u][i]!=P[v][i]){
7         u = P[u][i];
8         v = P[v][i];
9     }
10    return P[u][0];
11 }
12
13 void ComputeP(){
14     for (int i=0; i<lgN; ++i){
15         for (int x=0; x<n; ++x){
16             if (P[x][i]==-1) P[x][i+1] = -1;
17             else P[x][i+1] = P[P[x][i]][i];
18         }
19     }
20 }
```

**程式碼 9.13:** LCA finding algorithm

程式碼 9.13 的寫法雖然比程式碼 9.12 要多出一開始計算  $P$  的時間  $O(n \log n)$ ，但是在每次詢問的時候，程式碼 9.13 都可以在  $O(\log n)$  的時間內回答，而程式

碼 9.12 所需的時間為  $O(n)$ ，在詢問次數多的時候顯然程式碼 9.13 的效率要高出許多。

#### 9.4.4 樹序列化 Euler Tour Technique

對於樹的單點修改問題，或是針對子樹所有節點進行同一操作，除了使用複雜的資料結構維護外，還可以嘗試把樹轉換成一般的序列問題來完成。

我們可以透過圖 9.15 的方法，以樹的走訪順序把樹壓平成一個序列：除了可以在走訪的同時計算代表節點  $v$  與根節點  $r$  的距離  $D(v)$  以外，還可以同時利用時間戳記記錄第一次經過節點  $v$  的時間  $t_{in}(v)$  和最後一次經過節點  $v$  的時間  $t_{out}(v)$ ，由於每個邊都被經過兩次，因此時間戳記的大小為  $2n$ ，也就代表必需用兩倍的記憶體來維護。之後就轉成 RMQ 問題。

我們在後面的程式碼 9.14 提供範例程式。

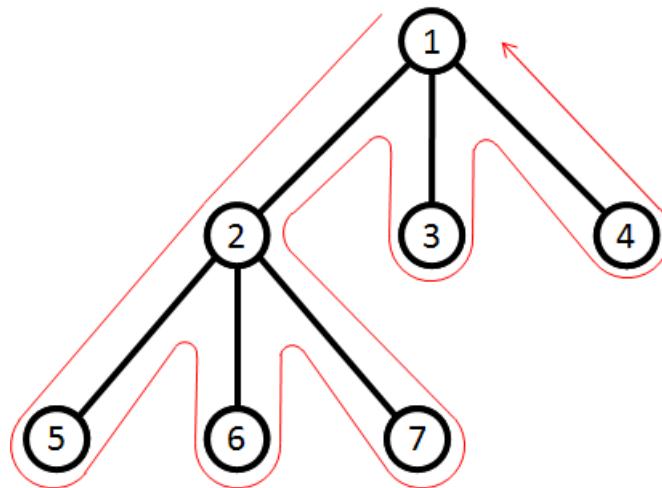


圖 9.15: Eulerian circuit of the tree

```

1 void DFS(int v, int pa){
2     timeI[v] = ts.size();
3     ts.emplace_back(D[v], v);
4     for (int x:adj[v]){
5         if(x==pa) continue;
6         D[x] = D[v]+1;
7         DFS(x, v);
8         ts.emplace_back(D[v], v);
9     }
10 }

```

程式碼 9.14: Euler Tour Technique

除了可以用 9.4.3 節中程式碼 9.13 來求樹上兩節點的 LCA 問題之外，樹序列化也可以用來解決相同的問題：先在由程式碼 9.14 得到的樹序列上建立一棵 segment tree，之後對節點  $x_1$  和  $x_2$  (假設  $x_1 < x_2$ ) 求  $\text{LCA}(x_1, x_2)$  時，就可以用 range minimum query (RMQ) 的方法來查詢  $\min\{t_{in}(i) | x_1 \leq i \leq x_2\}$ ，這樣得到的 pair 中，第一個資訊即為  $D(\text{LCA}(x_1, x_2))$ ，而第二個資訊即為  $\text{LCA}(x_1, x_2)$ 。

#### 9.4.5 樹鏈剖分 Heavy-Light Decomposition

在某些情況下，我們需要在一棵有  $n$  個節點的樹  $T$  上，對路徑  $(x_1, x_2)$  中所有的節點或邊進行權重修改和查詢的操作，當我們以樹序列化的 RMQ 來計算時，由於路徑  $(x_1, x_2)$  中相鄰的邊在樹序列化後所得到的陣列中並不一定相鄰，在修改時我們只能將每條邊的資訊各自更新，也就是說，若路徑  $(x_1, x_2)$  上共有  $k$  條邊，進行一次修改和查詢操作時均需要  $O(k \log n)$  的時間，在最壞的情況下會需要  $O(n \log n)$  的時間，效率顯然非常糟糕。

雖然以樹序列化的 RMQ 來維護資訊時，每次操作所需要的時間複雜度在一般樹圖上的效率並不好，但在某些情況下，以樹序列化的 RMQ 來維護資訊時，每次操作的時間複雜度卻可以達到  $O(\log n)$ 。這樣的情形出現在當樹的形狀為一長鏈時，此時在樹上相鄰的邊，在樹序列化後所得到的陣列中也必定相鄰，在這樣的情形下我們就可以使用樹序列化的 RMQ 來維護資訊。事實上，若我們將這樣的長鏈其中一端的節點作為這棵樹的樹根，在 RMQ 中我們甚至只需要維護由樹根往各節點這個方向的邊的資訊。

那麼，有沒有可能在將原本的樹  $T$  分成許多長鏈狀的子樹的同時，由  $T$  上任意一節點  $x_1$  到另一節點  $x_2$  又不會經過太多不同的子樹呢？

在 Robert Tarjan 於 1983 年提出的一篇論文中，提到在一棵有  $n$  個節點的樹  $T$  中，對於樹上所有的節點  $v$ ，在  $v$  與所有  $v$  的子節點當中，僅留下  $v$  與其最大子樹的子樹根相連的邊，這些被留下的邊會形成許多長鏈狀的樹，如圖 9.16c 所示。

圖 9.16d 代表的是將每一條鏈收縮為一個點，並且以最靠近  $T$  根節點的節點來代表該條鏈收縮後的節點。

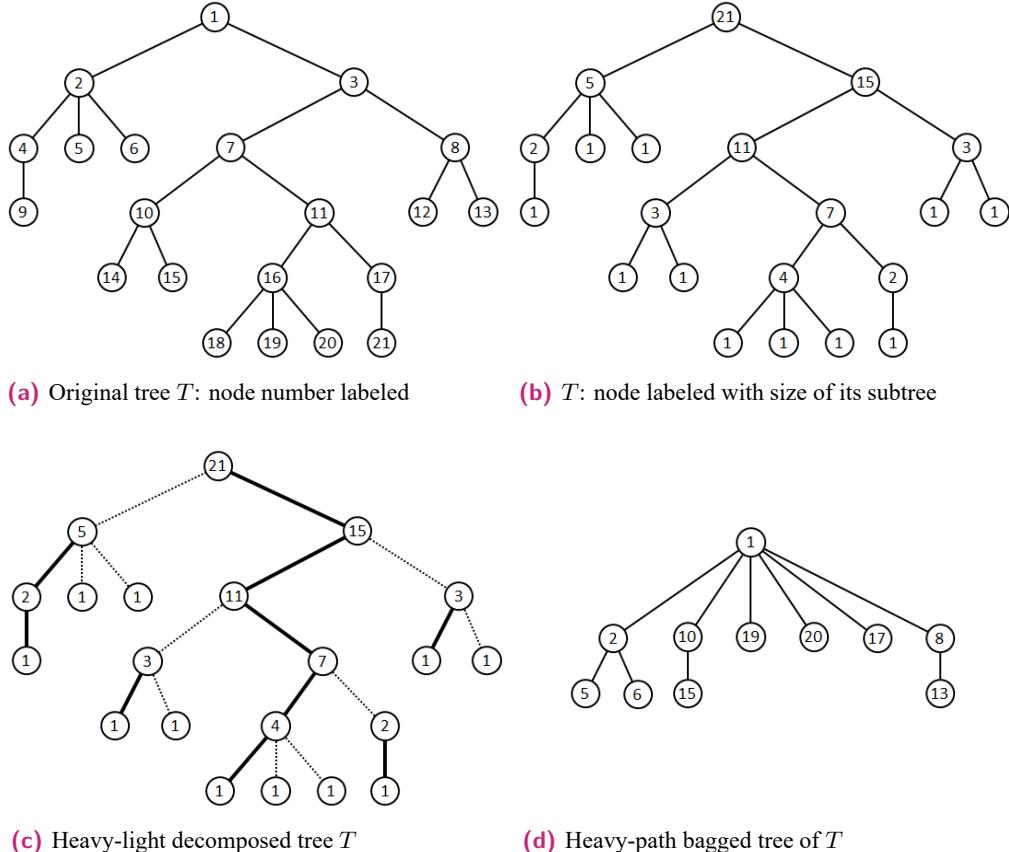


圖 9.16: Heavy-Light Decomposition

這樣的話，由原本  $T$  的根節點出發往任一葉節點前進時，假設在某個時候需由節點  $v_1$  走到處於不同長鏈上的節點  $v_2$ ，且與  $v_1$  所在同一長鏈上的  $v_1$  子節點為  $x$ ，那麼當我們假設由  $v_1, v_2$  和  $x$  作為根的子樹分別為  $T_1, T_2$  和  $T_x$  時，顯然有  $|T_2| \leq |T_x|$ 。

由於  $v_2$  與  $x$  均為  $v_1$  的子節點，我們可以得到事實  $|T_1| \geq |T_2| + |T_x| + 1$ ，由此可推得  $2|T_2| + 1 \leq |T_2| + |T_x| + 1 \leq |T_1|$ ，代表著在任何時候，若我們需要由一條鏈走到另一條鏈上時，所剩下可能走到的節點數必減少一半以上，這樣就保證了由  $T$  的根節點走向任一葉節點時，最多只能遇到  $\log n$  條不同的長鏈。

因為任意一條路徑  $(x_1, x_2)$  都可以藉由節點  $\text{LCA}(x_1, x_2)$  拆成兩段，因此由節點  $x_1$  走向  $x_2$  的沿途最多只會遇到  $2 \log n$  條鏈，所以在任意兩點間的路徑上，每次更新或查詢的操作最多只需要分為  $2 \log n$  次的區間操作，若是搭配線段樹的結構，時間複雜度可以達到  $O(\log^2 n)$ 。

以下我們在程式碼 9.15 紿出樹鏈部分的範例程式，其中  $r$  為樹  $T$  的根節點， $\text{sub}_T(v)$  代表以  $v$  作為根的子樹節點數， $\text{link}_R(v)$  代表  $v$  所在的長鏈中最靠近  $r$  的

節點， $link_P(v)$  代表  $v$  與  $link_R(v)$  的距離。我們假設對所有節點  $v$ ， $sub_T(v)$  為已知資訊， $link_P(v)$  預設為  $-1$ 。

```
1 void HLD(){
2     linkR[r] = r;
3     linkP[r] = 0;
4     Q.push(r); // Q為一佇列(queue)
5
6     while(!Q.empty()){
7         int v = Q.front();
8         Q.pop();
9         pair<int,int> res(-1, -1);
10        for (int x:adj[v]){
11            if (linkP[x]==-1) continue;
12            if (subTsize[x]>res.first) res = {subTsize[x], x};
13            Q.push(x);
14        }
15        if (res.second== -1) continue;
16        linkR[res.second] = linkR[v];
17        linkP[res.second] = linkP[v]+1;
18        for (int x:adj[v]){
19            if (linkP[x]==-1) continue;
20            linkR[x] = x;
21            linkP[x] = 0;
22        }
23    }
24 }
```

程式碼 9.15: Heavy-Light Decomposition

#### 9.4.6 練習題 Exercise

樹重心

POJ 1655

給定最多 20000 個節點的樹  $T$ ，求其重心節點。

點分治

POJ 1741

給定一節點數為  $n$  的邊帶權樹  $T$ ，求  $T$  中兩節點間距離不超過  $k$  的點對有多少個。

點分治

ACM-ICPC Live Archive 6900

給定一節點數為  $n$  的樹  $T$ ，其上每條邊都有所需花費和可獲收益，求  $T$  上總花費不超過  $C$  的路徑中，可獲的最大收益。

**最近共同祖先**

SPOJ 913

給定最多  $10000$  個節點的邊帶權樹  $T$ ，其上有兩種可能的查詢：一是詢問兩節點間最短距離，二是查詢由其中一節點沿最短路徑出發至另一節點所經過的第  $k$  個節點為何。

**最近共同祖先**

CodeForces 519E

給定最多  $10^5$  個節點的樹  $T$ ，以及最多  $10^5$  次詢問，每次詢問均給定  $T$  上兩節點，計算與此二節點距離相等的節點數量。

**樹序列化**

CodeForces 620E

給定一節點數為  $n$ ，根節點為節點 1 的樹  $T$ ，其上有兩種可能的操作：一是將某節點  $v$  及其子樹內的所有節點顏色都變為  $c$ ，二是詢問某節點  $v$  及其子樹內的所有節點共存在多少種不同顏色。

**樹鏈剖分**

UVa 1674

給定最多  $50000$  個節點的樹  $T$ ，每一次的操作都會將某兩點間路徑上的所有節點權重加上  $k$ ，問最後每一個節點的權重。

註：此題有不需要用到樹鏈剖分的方法，可單純以最近共同祖先的方法解決。

**樹鏈剖分**

SPOJ 375

給定一節點數為  $n$  的邊帶權樹  $T$ ，其上有兩種可能的操作：一是改變某個邊的權重，二是查詢兩點間路徑上邊的最大權重。

**挑戰題**

IPSC 2009 Problem L

給定最多  $10^6$  個節點的樹  $T$ ，以及最多  $2 \times 10^5$  次操作：對於給定兩點間的最短路徑上所有與指定顏色不同的邊都塗成指定顏色，對於每條需著色的邊都計算為使用該顏色一次。在所有操作結束後，輸出每種顏色被使用的次數。

註：SPOJ 上 Q-Tree 系列的題目共有七題，題號分別為 375、913、1487、2666、2939、16549 和 16580，均屬此章節範圍內可解的題目。



Number is the ruler of forms and ideas, and the cause of gods and daemons.

—— Πυθαγόρας o Σάμιος

『數學對於程式設計師來說到底重不重要？』在網路上已經不知道被討論多少次了。顯然想讓演算法能力變強的話一定數學基礎是必要的，透過數學推導可以讓某些演算法的複雜度大幅降低。本章首先介紹整數相關的數論，透過篩法快速找出質數、質因數，透過輾轉相除法找出最大公因數、最小公倍數；接著介紹模運算相關的操作，透過拓展歐基里德演算法、歐拉函數或是費馬小定理可以快速找出模逆元；最後介紹基礎的計算幾何。

## 10.1 數論 Number Theory

競賽中數論問題通常會與一些性質結合，本節會講解一些基本的數論定義、定理，以及利用藉由定理基礎寫出競賽中能夠使用的演算法。

### 10.1.1 整除性 Divisibility

**定義 10.1.** 對於兩個整數  $a, b$ ，我們說  $a$  整除  $b$ ，或  $b$  被  $a$  整除，若存在一個整數  $z$  使得  $az = b$ ，記作  $a|b$ ，此時我們稱  $a$  是  $b$  的因數， $b$  是  $a$  的倍數。

**定義 10.2.** 對於一個大於 1 的整數，如果其因數只有自己和 1，則稱這個數是質數，否則稱為合數。在此定義中 1 既不是質數也不是合數。

**定理 10.1.**  $n$  是一個合數。存在一個整數  $d$  使得  $d|n$  且  $d \leq \sqrt{n}$

*Proof.*  $n$  是一個合數代表可以寫為  $n = ab$ 。若  $a, b > \sqrt{n}$ ，則  $ab > n$ 。明顯矛盾！  $\square$

定理 10.1 提供了一種簡單的質數測試，給定一個大於等於 2 的正整數  $n$ ，可以測試所有小於等於  $\lfloor \sqrt{n} \rfloor$  的整數，是否整除  $n$ 。若存在  $a$  使得  $a|n$ ，則  $n$  是合數，否則為質數。時間複雜度為  $\mathcal{O}(\sqrt{n})$ 。

```

1 bool isPrime(int n) {
2     for (int i = 2; i * i <= n; ++i)
3         if (n % i == 0) return false;
4     return true;
5 }
```

**程式碼 10.1:** 檢素的質數測試

**定義 10.3.** 對於一個不知道是否是對的性質  $\mathcal{P}$ ，使用此符號表示：

$$[\mathcal{P}] = \begin{cases} 1 & \text{性質 } \mathcal{P} \text{ 是正確的} \\ 0 & \text{性質 } \mathcal{P} \text{ 是錯誤的} \end{cases}$$

**定義 10.4.**

$$\sum_{d|n} f(d) := \sum_{x=1}^n [x|n] f(x)$$

有一點值得注意的是

$$\sum_{d|n} H\left(\frac{n}{d}\right) = \sum_{d|n} H(d)$$

這裡  $f, H$  可以是任何的函數

### 10.1.2 埃氏篩法 Sieve of Eratosthenes

考慮以下這種題型：

$$\sum_{n=1}^N \sum_{p=1}^n f(p, n) [p \text{ 是質數}] [p|n]$$

限制是  $N \leq 10^5$

廣義而言，我們用 `doSomething(p, n)` 作為對一些  $p, n$  進行處理，注意執行順序不能影響答案。在這個例子中是在總和加進  $f(p, n)$ ，並假設 `isPrime(p)` 可以在常數時間內判斷  $p$  是不是質數，此時的程式碼會是：

```

1 for (int n = 1; n <= N; ++n) {
2     for (int p = 1; p <= n; ++p) {
3         if (isPrime(p) && n % p == 0) {
4             doSomething(p, n);
5         }
6     }
7 }
```

程式碼 10.2: 硬解

複雜度分析： $\mathcal{O}(n^2)$  · 吃了一個大大的 TLE。

注意到此處若將  $\sum$  交換可得到

$$\sum_{p=1}^N \sum_{n=p}^N f(p, n) [p \text{ 是質數}] [p|n] = \sum_{p=1}^N [p \text{ 是質數}] \left( \sum_{n=p}^N f(p, n) [p|n] \right)$$

括弧內可以運用每  $p$  個跳一次的方式加速，也就是

$$\sum_{p=1}^N [p \text{ 是質數}] \sum_{i=1}^{ip \leq N} f(p, ip)$$

```

1 for (int p = 2; p <= UPBD; ++p) {
2     if (isPrime(p)) {
3         for (int i = 1; i * p <= UPBD; ++i) {
4             doSomething(p, i * p);
5         }
6     }
7 }
```

程式碼 10.3: 交換迴圈的加速效果

注意到這兩次  $\text{doSomething}(p_i, n_i)$  中各別  $p_i, n_i$  的執行次數一樣，只差在執行順序不同，如果有特別的執行順序，可以先丟到某個陣列或是 `std::vector` 再排序，最後至多也只有  $\mathcal{O}(N \ln N)$  個元素。

| 質因數個數估計   | 證明題 |
|---|-----|
| $a$ 是大於一的正整數，令 $f(a)$ 代表 $a$ 的質因數個數，證明 $f(a) \leq \ln a$ · 這個性質可以幫估在 $N$ 個數字中 $\sum_{a=1}^N f(a)$ 的上界 $\mathcal{O}(N \ln M)$ , $a \leq M$ ) |     |

我們可以算算看 `doSomething` 這個函數被呼叫了幾次：

$$\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \dots + \frac{N}{p} = \mathcal{O}(N \ln \ln N)$$

測出這個上界是因為尤拉證出

$$\sum_{i=1}^N \frac{1}{p_i} = \Theta(\ln \ln N)$$

如果允許我們在某個範圍 (e.g. 範圍在  $10^6$  之內) 預處理的話，可以維護陣列 `divider[n]` 為  $n$  的最小質因數，這樣可以完成以下兩項任務：

1. 測試一個數字是不是質數，就測試 `divider[n] == n`
2. 在  $\mathcal{O}(\ln n)$  內分解所有質因數，只要一個一個把最小質因數分解出來就好了

注意到  $p$  是質數，若且唯若沒有任何質數  $q < p$  可以整除  $p$ ，所以我們將質數往前篩時，維護 **對於任何  $n$ ，`divider[n]` 表示目前最小的質數  $q < p$ ， $p$  是目前篩到最大質數的性質**，即埃式篩法。

```
1 vector<int> sieve(const int UPBD) {
2     const int UNSIEVED = 0;
3     vector<int> divider(UPBD, UNSIEVED);
4     for (int p = 2; p < UPBD; ++p) {
5         for (int n = p; n < UPBD; n += p) {
6             if (divider[n] == UNSIEVED) divider[n] = p;
7         }
8     }
9     return divider;
10}
11/*
12 2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19
13 2   3   2   5   2   7   2   3   2   11  2   13  2   3   2   17  2   19
14 */
```

程式碼 10.4：篩法

如果可以控制每個數字  $n$  都只能被篩一次的話，那麼篩法就可以達到線性：

```
1 vector<int> linear_sieve(const int UPBD) {
2     const int UNSIEVED = 0;
3     vector<int> primes, divider(UPBD, UNSIEVED);
4     for (int n = 2; n < UPBD; ++n) {
5         if (divider[n] == UNSIEVED) {
6             primes.push_back(n), divider[n] = n;
7         } // This is a prime.
8         for (int j = 0; primes[j] * n < UPBD; ++j) {
9             divider[primes[j] * n] = primes[j];
10            if (n % primes[j] == 0) break;
11        } // If n = 11 * 13 then 2, 3, 5, 7, 11 are selected
12    }
13    return divider;
```

### 程式碼 10.5: 線性篩

注意到 `divider[n]` 做完以後，對於很多的詢問，每個詢問  $n \leq N$  可以在  $\mathcal{O}(\log n)$  因數分解，或是  $\mathcal{O}(1)$  測試  $n$  是不是質數。

| 分解每個詢問                                  | 經典問題 |
|---|------|
| 給定十萬個小於 $10^6$ 的數字，每個詢問請在 20 步內求出質因數分解。 |      |

### 10.1.3 最大公因數與最小公倍數 GCD & LCM

**定義 10.5.** 對於整數  $a, b$  我們稱  $a, b$  的**最大公因數**，是最大的正整數  $d$  使得  $d|a$  且  $d|b$ ，記作  $d = \gcd(a, b)$ 。當  $\gcd(a, b) = 1$  時，我們稱  $a, b$  **互質**。

這是個直觀且好用的定理：

**定理 10.2.**  $\gcd(a, b) \neq 1 \Leftrightarrow \exists p$  是質數使得  $p|a$  且  $p|b$

| 最小字典序   | 經典問題 |
|---|------|
| 一個陣列有十萬個小於 $10^6$ 的數字。如果兩個相鄰的整數互質則可以交換，否則不行。請問這個陣列的最小的字典序是什麼？ |      |

### 10.1.4 輾轉相除法 Euclid Algorithm

**定理 10.3.**  $\gcd(a, b) = \gcd(b \pmod a, a)$

相信大多數人都在國小學過輾轉相除法了，因為用定理 10.3 遍迴寫就可以把問題慢慢變小，直到一邊整除另一邊馬上返回答案，算是十分基礎的演算法，複雜度  $\mathcal{O}(\log(a + b))$ 。

```

1 int gcd(int a, int b) {
2     if (a == 0) return b;
3     return gcd(b % a, a);
4 }
```

### 程式碼 10.6: 輾轉相除法

現在要來介紹介紹輾轉相除法的擴充算法。

**定理 10.4.**  $a, b, r \in \mathbb{Z}$  若  $d := \gcd(a, b)$  · 則找得到  $s, t \in \mathbb{Z}$  使得  $as + bt = r \Leftrightarrow d|r$

*Proof.* 可以透過輾轉相除法倒推構造  $s, t$  使得  $as + bt = d$  。  $\square$

關於如何倒推 · 我們注意到對於整數  $a, b$  可以線性寫成  $r = a - qb$  · 然後將問題慢慢變小

```
1 auto exd_gcd(int a, int b) {
2     if (b == 0) return pair{1, 0};
3     else if (a == 0) return pair{0, 1};
4     auto [s, t] = exd_gcd(b, a % b);
5     return pair{t, s - a / b * t};
6 } // return {s, t} s.t. s * a + t * b = gcd(a, b);
```

**程式碼 10.7:** 遞迴的擴充輾轉相除法

**引理 10.5** (歐幾里德引理). 已知  $\gcd(a, b) = 1$  · 那麼若  $a|bc$  · 則  $a|c$

*Proof.* 存在  $s, t$  使得  $as + bt = 1$  · 兩邊同乘以  $c$  得到  $acs + bct = c \Rightarrow a|c$   $\square$

## 10.2 模運算 Modular Arithmetics

### 10.2.1 同餘式與模數 Congruence & Modulo

**定義 10.6.** 若  $n|(a - b)$  · 則記作  $a \equiv b \pmod{n}$

**定理 10.6.** 以下是幾個模數運算常見的定理 :

1.  $a \equiv b \pmod{n}, c \equiv d \pmod{n} \Rightarrow a + c \equiv b + d \pmod{n}$
2.  $a \equiv b \pmod{n}, c \equiv d \pmod{n} \Rightarrow ac \equiv bd \pmod{n}$
3.  $a \equiv b \pmod{n} \Rightarrow a^k \equiv b^k \pmod{n} \forall k \in \mathbb{N}$

這裡挑第三點來證 :

*Proof.*

$$n|(a - b) \Rightarrow n|(a - b)(a^{k-1} + a^{k-2}b + a^{k-3}b^2 + \dots + ab^{k-2} + b^{k-1}) \Rightarrow n|(a^k - b^k)$$

□

利用模數運算，我們可以定義出一種代數結構：

**定義 10.7.** 對於  $n$  大於 1，定義  $\langle \mathbb{Z}_n, +_n, \times_n \rangle$ ，簡寫為  $\mathbb{Z}_n$ ，其中：

$$\mathbb{Z}_n = \{0, 1, 2, 3, 4, \dots, n-1\}$$

$$a +_n b := (a + b) \pmod{n}, a \times_n b := (a \times b) \pmod{n} \quad \forall a, b \in \mathbb{Z}_n$$

沒錯！這就是平常熟悉的% 運算，不過請注意正負值。

**定理 10.7.** 在  $\mathbb{Z}_n$  運算下同樣滿足

1. 加法乘法結合律：

$$(a \times_n b) \times_n c = a \times_n (b \times_n c), (a +_n b) +_n c = a +_n (b +_n c)$$

2. 加法乘法交換律：

$$a \times_n b = a \times_n b, a +_n b = b +_n a$$

3. 分配律：

$$(a +_n b) \times_n c = a \times_n c +_n b \times_n c$$

有時兩個不同模的數乘以常數後會是同模的，這時有消去定理，同時也給出在模運算之下「除法」的概念：

**定理 10.8** (消去定理).  $d = \gcd(c, n)$

$$ca \equiv cb \pmod{n} \Rightarrow a \equiv b \pmod{\frac{n}{d}}$$

*Proof.* 因為有定理 10.5，

$$n|c(a - b) \Rightarrow \frac{n}{d} \mid \frac{c}{d}(a - b)$$

而且

$$\gcd\left(\frac{n}{d}, \frac{c}{d}\right) = 1$$

我們可以推到

$$\frac{n}{d} \mid (a - b) \Rightarrow a \equiv b \pmod{\frac{n}{d}}$$

□

### 10.2.2 反元素 Inverse Element

**定義 10.8.**  $a, b \in \mathbb{Z}_n$  我們稱  $b$  是  $a$  的乘法反元素 (簡稱反元素)，有

$$a \times_n b = 1 \Leftrightarrow b := a^{-1}, a := b^{-1}$$

$\mathbb{Z}_n$  中，不是每個數都有反元素。例如，0 在任何  $\mathbb{Z}_n$  一定沒有反元素； $\mathbb{Z}_4$  中 2 沒有反元素。以  $\mathbb{Z}_9$  為例，看看它的反元素是什麼：

|          |   |   |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|---|---|
| $a$      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| $a^{-1}$ | 無 | 1 | 5 | 無 | 7 | 2 | 無 | 4 | 8 |

表 10.1:  $\mathbb{Z}_9$  乘法反元素表

由此可以觀察可發現以下定理：

**定理 10.9.**  $a$  在  $\mathbb{Z}_n$  中有乘法反元素若且唯若  $\gcd(a, n) = 1$ 。

*Proof.* 輾轉相除擴充演算法總是能把反元素造出來，可以利用擴充算法找模  $p$  下的反元素：

$$ax + py = 1 \Leftrightarrow ax = 1 - py \Leftrightarrow ax \equiv 1 \pmod{p} \Leftrightarrow x \equiv a^{-1} \pmod{p}$$

□

輾轉相除擴充演算法分解的步驟很少，寫成迭代更快！

```
1 int inv(int a, int p) {
2     auto [x, y] = exd_gcd(a, b);
3     if (x * a + y * b == 1) {
4         return (x + n) % n;
5     } else {
6         return -1; // gcd(a, b) != 1 時沒有反元素
7     }
8 }
```

**程式碼 10.8:** 輾轉相除擴充演算法求  $a^{-1} \pmod{p}$

**定理 10.10.** 反元素是唯一的，也就是說如果  $a \times_n b = 1, a \times_n c = 1$  則  $b \equiv c \pmod{n}$

*Proof.*  $a \times_n (b - c) = 0$ ，也就是  $n|a(b - c)$ 。

因為  $\gcd(n, a) = 1$ ，利用定理 10.5 可以推得  $n|(b - c)$

□

### 乘法反元素表

設  $\text{inv}(x)$  為  $x$  模  $P$  下的反元素，則  $\text{inv}(1) \sim \text{inv}(n)$  可以在  $\mathcal{O}(N)$  時間求出來（當  $\gcd(i, P) \neq 1$  時  $\text{inv}(i) = 0$ ）：

```
1 vector<int> linear_inv(int p) {
2     vector<int> inv(p);
3     inv[1] = 1;
4     for (int i = 2; i < p; ++i)
5         inv[i] = (p - p / i) * inv[p % i] % p;
6     return inv;
7 }
```

**程式碼 10.9:** 構造反元素表

*Proof.*

$$\begin{aligned} P - \lfloor P/i \rfloor \times i &= P \% i \\ \implies P \times i - \lfloor P/i \rfloor \times i &= P \% i \pmod{P} \\ \implies i \times (P - \lfloor P/i \rfloor) &= P \% i \pmod{P} \\ \implies i \times (P - \lfloor P/i \rfloor) \times \text{inv}(P \% i) &= 1 \pmod{P} \\ \implies \text{inv}(i) &= (P - \lfloor P/i \rfloor) \times \text{inv}(P \% i) \pmod{P} \end{aligned}$$

□

### 10.2.3 費馬小定理 Fermat's Little Theorem

**定理 10.11 (費馬小定理).** 對於任何整數  $a$ ，質數  $p$  皆滿足  $a^p - a \equiv 0 \pmod{p}$ 。若  $a$  不是  $p$  的倍數，則可以推得  $a^{p-1} \equiv 1 \pmod{p}$

*Proof.* 費馬有一個很關鍵的發現：對於任何  $a \in \mathbb{Z}_n$ ,  $a \neq 0$  在

$$1a, 2a, 3a, 4a, \dots, (p-1)a$$

中必須一一對應到

$$1, 2, 3, 4, \dots, (p-1)$$

否則，會造成矛盾： $i > j, ia \equiv ja \pmod{p} \Rightarrow p|(i-j)a$ 。

$$1a, 2a, 3a, 4a, \dots, (p-1)a, 1, 2, 3, 4, \dots, (p-1)$$

在兩邊連乘得到：

$$(p-1)!a^{p-1} = a \cdot 2a \cdot 3a \cdots (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdots (p-1) = (p-1)! \pmod{p}$$

根據定理 10.8，兩邊消去  $(p-1)!$  得到  $a^{p-1} \equiv 1 \pmod{p}$

這說明了本節的重點：任何一個非  $p$  倍數的正整數，都有乘法反元素  $a^{p-2}$ （還記得定理 10.10 說明反元素是唯一的嗎？） $\square$

現在我們知道  $\mathbb{Z}_p$  的元素支援加減，非零元素支援乘除。像這樣的性質的結構，數學家們稱作是一個體（Field）。因此可以使用快速幕來求  $a^{p-2}$ 。

我們可以用上述的性質把模數運算包裝起來

```
1 template <int mod>
2 struct ModInt {
3     int val;
4     int trim(int x) const { return x >= mod ? x - mod : x < 0 ? x + mod : x;
5             ; }
6     ModInt(int v = 0) : val(trim(v % mod)) {}
7     ModInt(long long v) : val(trim(v % mod)) {}
8     ModInt &operator=(int v) { return val = trim(v % mod), *this; }
9     ModInt &operator=(const ModInt &oth) { return val = oth.val, *this; }
10    ModInt operator+(const ModInt &oth) const { return trim(val + oth.val);
11            }
12    ModInt operator-(const ModInt &oth) const { return trim(val - oth.val);
13            }
14    ModInt operator*(const ModInt &oth) const { return 1LL * val * oth.val
15            % mod; }
16    ModInt operator/(const ModInt &oth) const {
17        function<int(int, int, int, int)> modinv = [&](int a, int b, int x,
18                int y) {
19            if (b == 0) return trim(x);
20            return modinv(b, a - a / b * b, y, x - a / b * y);
21        };
22        return *this * modinv(oth.val, mod, 1, 0);
23    }
24    bool operator==(const ModInt &oth) const { return val == oth.val; }
25    ModInt operator-() const { return trim(mod - val); }
26    template<typename T> ModInt pow(T pw) {
27        bool sgn = false;
28        if (pw < 0) pw = -pw, sgn = true;
29        ModInt ans = 1;
```

```

25     for (ModInt cur = val; pw; pw >= 1, cur = cur * cur) {
26         if (pw&1) ans = ans * cur;
27     }
28     return sgn ? ModInt{1} / ans : ans;
29 }
30 };

```

程式碼 10.10: 模數運算

把模數運算寫成物件的好處可以讓我們不用寫兩套快速傅立葉變換 ( 見11.2 )。

#### 10.2.4 歐拉函數 Euler Function

**定義 10.9.** 歐拉函數  $\Phi(n)$  的值表示在  $1, 2, \dots, n$  中與  $n$  互質的個數。

我們使用  $\mathbb{Z}_n^*$  代表從  $[0, n)$  中與  $n$  互質的數，也可以說  $\mathbb{Z}_n^*$  有  $\Phi(n)$  個元素。

**定理 10.12** (Euler). 如果  $\gcd(a, n) = 1$  · 則  $a^{\Phi(n)} \equiv 1 \pmod{n}$

*Proof.* 注意到費馬小定理的證明 · 如果  $\gcd(a, n) = 1$  · 也可以用來證明  $a^{\Phi(n)} \equiv 1 \pmod{n}$  □

**定理 10.13.** 若  $\gcd(m, n) = 1$  · 則  $\Phi(mn) = \Phi(m)\Phi(n)$

*Proof.* 我們可以注意下圖 · 可以發現只有  $\Phi(m)$  個列有跟  $m$  互質的數 ( 為什麼呢？) · 每個列剛好有  $\Phi(n)$  個項跟  $n$  互質 ( 因為列中的元素會跟  $0, 1, 2, \dots, n-1 \pmod{n}$  一一對應 ) □

|            |            |            |     |      |
|------------|------------|------------|-----|------|
| 1          | 2          | 3          | ... | $m$  |
| $m+1$      | $m+2$      | $m+3$      | ... | $2m$ |
| ⋮          | ⋮          | ⋮          | ⋮   | ⋮    |
| $(n-1)m+1$ | $(n-1)m+2$ | $(n-1)m+3$ | ... | $nm$ |

**定理 10.14.**

$$\sum_{d|n} \Phi(d) = n$$

*Proof.* 若  $d|n$  · 在  $\{1, 2, 3, 4, 5, \dots, n\}$  裡面總共有  $\Phi(d)$  個元素跟  $n$  的最大公因數是  $\frac{n}{d}$  則  $\sum_{d|n} |\{a| \gcd(a, n) = \frac{n}{d}\}| = \sum_{d|n} \Phi(d) = \sum_{d|n} \Phi(d) = n$   $\square$

請從上面性質自行推出歐拉函數公式：

| 歐拉函數公式  | 證明題 |
|---|-----|
| 給予一個 $n = p_1^{k_1} p_2^{k_2} p_3^{k_3} \cdots p_n^{k_n}$ · 請算出 $\Phi(n)$ · |     |

### 10.2.5 中國剩餘定理 Chinese Remainder Theorem

**定理 10.15 (中國剩餘定理).** 令  $\{n_k\}_{i=1}^k$  為兩兩互質的正整數 · 令  $a_1, a_2, \dots, a_k$  為任意的整數 · 則存在  $a \in \mathbb{Z}$  使得滿足： $a \equiv a_i \pmod{n_i}$ ,  $i = 1, \dots, k$  再者 · 令  $n = \prod_{i=1}^k n_i$  ·  $a' \in \mathbb{Z}$  也是一個解若且唯若  $a \equiv a' \pmod{n}$

*Proof.* 如果我們可以知道  $e_i \equiv \begin{cases} 1 & (\text{mod } n_i) \\ 0 & (\text{mod } n_j) \quad j \neq i \end{cases}$  · 則可以造出  $a \equiv \sum_{i=1}^k a_i e_i \pmod{n}$  要如何找出  $e_i$  呢？用歐幾里德擴充算法：

$$\Leftrightarrow n_i^* = \frac{n}{n_i} = \prod_{i=1, i \neq j}^k n_i \cdot \text{有 } s n_i + t n_i^* = 1 \cdot \text{ 則 } e_i := t n_i^* \text{ 就完成了}$$

更甚者 · 我們可以將中國剩餘定理看成一個對於兩個方程式的二元運算組：

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 template<typename T>
5 pair<T, T> exd_gcd(T a, T b) {
6     if (b == 0) return pair{1, 0};
7     else if (a == 0) return pair{0, 1};
8     auto [s, t] = exd_gcd(b, a % b);
9     return pair<T, T>{t, s - a / b * t};
10 } // return {s, t} s.t. s * a + t * b = gcd(a, b);
11
12 template<typename T>
13 pair<T, T> CRT(pair<T, T> eq1, pair<T, T> eq2) {
14     auto [x1, m1] = eq1;
15     auto [x2, m2] = eq2;
16     if (m1 == 0 or m2 == 0) return {0, 0};
17     T g = std::gcd(m1, m2);
18     if ((x1 - x2) % g) return pair<T, T>{0, 0}; // NO SOLUTION
19     m1 /= g, m2 /= g;
20     auto p = exd_gcd(m1, m2);
21     T lcm = m1 * m2 * g;
22     T res = (p.first * (x2 - x1) % lcm) * m1 % lcm + x1;

```

```

23     return {(res % lcm + lcm) % lcm, lcm};
24 }
25
26 int main() {
27 {
28     auto [res, lcm] = CRT<int>({25, 50}, {23, 26});
29     cout << res << ' ' << lcm << '\n';
30 }
31 {
32     vector<pair<int, int>> res_eqs = {{25, 50}, {23, 26}, {8, 9}};
33     auto [res, lcm] = accumulate(res_eqs.begin(), res_eqs.end(),
34                                 pair{0, 1}, CRT<int>);
35     cout << res << ' ' << lcm << '\n';
36 }
37 {
38     vector<pair<int, int>> res_eqs = {{25, 50}, {23, 26}, {24, 25}};
39     auto [res, lcm] = accumulate(res_eqs.begin(), res_eqs.end(),
40                                 pair{0, 1}, CRT<int>);
41     cout << res << ' ' << lcm << '\n';
42 }
43 }
44 /*
45 75 650
46 4625 5850
47 0 0
48 */

```

**程式碼 10.11:** 中國剩餘定理



本章的內容較為進階且幾乎不會出現在簡單的題目中，因此本次的營隊並不會討論其中內容。

## 11.1 進階模運算 Advanced Modular Arithmetics

### 11.1.1 數論函數 Number-Theoretic function

**定義 11.1.** 一個數論函數  $f(n)$  是一個定義在正整數的函數，值域是實數（或複數）。

簡單來說數論函數吃一個正整數，吐出一個實數或複數。

**定義 11.2.** 設數論函數  $f(n)$ ，若  $\gcd(m, n) = 1$ ，滿足  $f(1) = 1$  且  $f(mn) = f(m)f(n)$  的話我們稱  $f$  為積性函數 (Multiplicative function)

例子像是  $\Phi(x)$  是積性函數，考慮到任何一個正整數可以分解成質數的冪次，要算積性函數  $f(p_1^{k_1}p_2^{k_2}\dots p_n^{k_n})$ ，我們只要計算  $f(p_1^{k_1})f(p_2^{k_2})\dots f(p_n^{k_n})$  即可

| 構造積性函數   | 經典問題 |
|--|------|
| 給一堆兩兩互質的正整數 $\{a_i\}_{i=0}^{N-1}$ , $a_i \leq 10^6$ ，還有 $f(a_i)$ 們也給了，請造出任何一個符合條件積性函數 $f$ ，構造方法為輸出 $f(1), f(2), \dots, f(10^6)$ ，若無解請輸出 $-1$ |      |

**定理 11.1.** 若  $f(n)$  是積性函數，則  $F(n) = \sum_{d|n} f(d)$  也是積性函數

*Proof.* 若  $m, n$  互質，可以把任何  $mn$  的因數拆開寫成  $d_1|m, d_2|n \cdot d_1d_2|mn$

$$F(mn) = \sum_{d|nm} f(d) = \sum_{d_1|m, d_2|n} f(d_1)f(d_2) = \sum_{d_1|m} f(d_1) \sum_{d_2|n} f(d_2) = F(m)F(n)$$

□

對於數論函數，有所謂狄利克雷卷積 (Dirichlet convolution)：

**定義 11.3.** 我們定義兩個數論函數  $f, g$  的**狄利克雷卷積**為

$$(f * g)(n) := \sum_{d_1|n} f(d_1)g\left(\frac{n}{d_1}\right)$$

或者寫得更直觀一點：

$$(f * g)(n) := \sum_{d_1 d_2 = n} f(d_1)g(d_2)$$

三個函數的狄利克雷卷積長這樣，依此類推

$$((f * g) * h)(n) = \sum_{d_1 d_2 d_3 = n} f(d_1)g(d_2)h(d_3)$$

很明顯的可以看出狄利克雷卷積有結合率、交換率：

$$\begin{aligned} 1. \quad & (f * g) * h \\ &= \sum_{d_1 d_2 d_3 = n} f(d_1)g(d_2)h(d_3) = f * (g * h) \end{aligned}$$

$$\begin{aligned} 2. \quad & f * g \\ &= \sum_{d_1 d_2 = n} f(d_1)g(d_2) = g * f \end{aligned}$$

以下介紹幾個等等會用到的積性函數，可以自行驗證看看它是否是積性函數

$$1. \quad I(x) := [x = 1]$$

$$2. \quad \text{常數函數 } 1 : 1(x) := 1$$

3. 莫比烏斯函數：

$$\mu(x) = \begin{cases} 1 & x = 1 \\ 0 & p^2|x, p \text{ 是質數} \\ (-1)^k & x = p_1 p_2 \cdots p_k \end{cases}$$

注意一下

$$\sum_{d|x} \mu(d) = [x = 1] = I(x)$$

(可令  $x = p^k$  證證看)

### 11.1.2 莫比烏斯反演 Möbius Inversion Formula

**定理 11.2.** 對於數論函數  $f$  跟  $F$  關係如下：

$$F(x) = \sum_{d|x} f(d)$$

那麼我們有：

$$f(x) = \sum_{d|x} \mu(d) F\left(\frac{x}{d}\right)$$

*Proof.* 核心的證明想法就是交換  $\sum$  的技巧，這在比賽中很常見

$$\sum_{d|x} \mu(d) F\left(\frac{x}{d}\right) = \sum_{d|x} \mu(d) \sum_{c|\frac{x}{d}} f(c) = \sum_{d|x} \sum_{c|\frac{x}{d}} f(c) \mu(d)$$

改成這個寫法好看多了

$$\sum_{cd|x} f(c) \mu(d)$$

拜託拜託，一定要注意到

$$(d|x \text{ 且 } c|\frac{x}{d}) \Leftrightarrow cd|x \Leftrightarrow (c|x \text{ 且 } d|\frac{x}{c})$$

因此原式可以寫回

$$\sum_{c|x} \sum_{d|\frac{x}{c}} f(c) \mu(d) = \sum_{c|x} f(c) \sum_{d|\frac{x}{c}} \mu(d) = \sum_{c|x} f(c)[x=c] = f(x)$$

□

或者大可以把剛剛那一段醜醜的證明忘記，改成看這個簡潔有力的證明：

*Proof.*

$$\sum_{d|x} \mu(d) F\left(\frac{x}{d}\right) = (\mu * f * 1)(x) = ((\mu * 1) * f)(x) = ([x=1] * f)(x) = \sum_{d|x} [d=1] * f\left(\frac{n}{d}\right) = f(x)$$

□

$f$  是歐拉函數 ·  $g(n) = \sum_{d|n} f\left(\frac{n}{d}\right)$  · 試著計算

$$F_k(n) = \begin{cases} f(g(n)) & k = 1 \\ g(F_{k-1}(n)) & k > 1, k \text{ 是偶數} \\ f(F_{k-1}(n)) & k > 1, k \text{ 是奇數} \end{cases}$$

將答案模 1000000007 輸出

### 11.1.3 原根 Primitive Root

**定義 11.4.** 對於質數  $p$  · 我們說  $a \in \mathbb{Z}_p^*$  的序 (order) 是最小的正整數  $d$  · 使得  $a^d = 1$

這樣講可能有點抽象 · 舉個例子 :

12 在  $\mathbb{Z}_{29}^*$  的序是 4 · 因為  $12^4 \equiv 1 \pmod{29}$  · 而且比 4 小的正整數次方都不是 1

5 在  $\mathbb{Z}_{101}^*$  的序是 25 · 因為  $5^{25} \equiv 1 \pmod{101}$  · 而且比 25 小的正整數次方都不是 1

**定義 11.5.** 對於質數  $p$  · 我們說  $a \in \mathbb{Z}_p^*$  是  $\mathbb{Z}_p^*$  的原根 (Primitive Root · 群論用語會翻 generator) · 如果比  $p - 1$  小的  $a$  的正整數幕次都不是 1 · 也就是說 ·  $a$  的序是  $p - 1$  °

可以證明這樣的  $a$  一定存在 · 證明這裡先沒寫出來。

**定理 11.3.** 若  $a$  是  $\mathbb{Z}_p^*$  的原根 · 則  $a^0, a^1, a^2, \dots, a^{p-2}$  —— 對應到  $1, 2, 3, 4, \dots, p - 1$

*Proof.* 若  $0 \leq i < j < p - 2$ ,  $a^i = a^j$  · 則依據消去定理  $a^{j-i} \equiv 1 \pmod{p}$  不符合定義

□

例如 3 是  $\mathbb{Z}_7^*$  的原根 :

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$$

而 3 的幕次表如下

**定理 11.4.** 正整數  $d | p - 1$  ·  $\mathbb{Z}_p^*$  中序為  $d$  的元素剛好有  $\Phi(d)$  個

|       |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|
| $t$   | 1 | 2 | 3 | 4 | 5 | 6 |
| $3^t$ | 3 | 2 | 6 | 4 | 5 | 1 |

表 11.1: 3 的幂次表

*Proof.* 那些元素就是  $\{a^k : \gcd(k, p-1) = \frac{p-1}{d}\}$  中的元素 ·  $a$  是任一個原根 · 至於為什麼它們的序是  $d$  :

$(a^k)^x = 1 \Rightarrow p-1|x$  再利用引理 10.5 可以推到  $d|x$

□

給一個質數  $p$  · 我們有辦法找到任一個原根嗎？我們當然可以一個個試  $2, 3, 4, \dots, p-1$  · 然後將它們從  $1, \dots, p-1$  次方都算過一次 · 但若  $p \leq 10^9$  · 可能就沒有那麼容易了 · 關於這個問題 · 有一種機率解法是這樣：

### 演算法 11.1 找一個原根的演算法

- 
- 1: 先把  $p-1$  作因式分解 ·  $p-1 = \prod_{i=1}^r q_i^{e_i}$
  - 2: 對於每一種  $q_i$  隨便挑一個數  $\alpha_i$  使得  $\alpha_i^{\frac{p-1}{q_i}} \neq 1$  · 令  $\gamma_i = \alpha_i^{\frac{p-1}{q_i^{e_i}}}$  · 失敗了就再來一遍
  - 3: 輸出答案  $\gamma = \prod_{i=1}^r \gamma_i$
- 

在每一步驟中 · 簡單說明一下為什麼要這樣做：

1. 第二步 · 如果這樣挑可以令  $\gamma_i^{q_i^{e_i}-1} \neq 1$  · 且  $\gamma_i^{q_i^{e_i}} = 1$  。而挑中的機率其實挺大的 · 假設  $a$  是原根 · 只要挑中的不是  $a^{q_i}, a^{2q_i}, a^{3q_i}, \dots, a^{p-1} = 1$  即可 · 平均做兩次以內就找得到。在此 ·  $\gamma_i$  的序是  $q^{e_i}$  。
2. 第三步 · 這裡直接引用群論定理：如果  $a, b$  的序互質 · 則  $ab$  的序就是各自的序的乘積 · 因此  $\gamma$  的序是  $p-1$  · 講白一點 ·  $\gamma$  就是原根。
2. 3. 步驟複雜度的期望值是  $\mathcal{O}(r \log p)$  ·  $r$  是質因數個數 · 已經幾乎是常數了 · 因此最慢的部分還是在分解  $p-1$

現在問題反過來 · 給一個數  $\gamma$  · 它是不是  $\mathbb{Z}_p^*$  的原根？如果是的話 · 假設  $a$  是另一個原根 · 則  $\gamma = a^d$  ·  $\gcd(p-1, d) = 1$  · 否則一定有個質數  $q$  使得  $q|\gcd(p-1, d)$  · 則  $p-1|\frac{p-1}{q}d$  · 我們只要一一測試分解  $p-1$  的質數 · 看看  $\gamma^{\frac{p-1}{q}}$  是否等於一就好了。

其實利用以上的想法 · 可以概括到求一個數  $x$  在模  $m$  之下的序 (order) · 假設  $\phi(m) = \prod_{i=1}^r q_i^{e_i}$  且  $x$  的序是  $\prod_{i=1}^r q_i^{l_i}, l_i \leq e_i$  · 我們可以一一將  $e_i$  往下迭代 · 直到  $e_i = 0$  · 或是  $l_i > e_i$  ( 這個時候該數字不會整除  $x$  的序 i.e.  $x$  到這個次方不會等於一 · 就知道要退回去前一步 )。

```

1 template<typename F>
2 F order(F x, F m) {
3     assert(std::gcd(x, m) == 1);

```

```

4   F ans = phi(m);
5   for (auto [q, e]: factor(ans)) { // {qi, ei}
6     for (int i = 0; i < e; ++i) {
7       if (powM(x, ans / q, m) == 1) ans /= q;
8       else break;
9     }
10  }
11  return ans;
12 }

```

## 11.2 快速傅立葉變換 Fast Fourier Transform

### 11.2.1 生成函數 Generating Function

生成函數是一種多項式的衍生函數  $F(x) = a_0 + a_1x + a_2x^2 + a_3x^3\dots$  . 函數行為意義不大，主要用途是對於用來查他的第  $N$  次項，可以把它想像成是有無限多項的陣列。像是  $G(x) = 1 + \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{8}x^3 + \frac{1}{16}x^4\dots$  可以想像成  $\langle 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}\dots \rangle$

**定義 11.6.** 我們定義一個序列  $\langle a_i \rangle_{i=0}^{\infty}$  的生成函數  $F(x)$  可被寫作  $F(x) = \sum_{i=0}^{\infty} a_i x^i$  對於一個生成函數，我們用  $[x^n]F(x)$  來代表  $F(x)$  的第  $n$  次項

相加（減）相乘就跟多項式一樣

**定義 11.7.** 令

$$A(x) = \sum_{i=0}^{\infty} a_i x^i, B(x) = \sum_{i=0}^{\infty} b_i x^i$$

兩個生成函數的相加定義為

$$A(x) + B(x) = \sum_{i=0}^{\infty} (a_i + b_i) x^i$$

兩個生成函數的相乘定義為

$$[x^n](AB)(x) = \sum_{i=0}^n \sum_{j=0}^n a_i b_j [i+j = n] = \sum_{i=0}^n a_i b_{n-i} = \sum_{i=0}^n a_{n-i} b_i$$

生成函數與 dp 還可以扯得上邊，甚至是拿來分析 dp 的好工具呢  
 如果遇到像是  $G(x) = 1 + kx + k^2x^2 + k^3x^3 + k^4x^4 \dots$  的生成函數，令  $A(x) = \sum_{i=0}^{\infty} a_i x^i$ ，計算  $[x^n](GA)(x)$  時，不必花  $\mathcal{O}(n^2)$  將他們相乘，留意到

$$[x^n](GA)(x) = \sum_{i=0}^n k^{n-i} a_i = a_n + k \sum_{i=0}^{n-1} k^{n-1-i} a_i = a_n + k[x^{n-1}](GA)(x)$$

變成了漂亮的 dp 式。

| 01 背包計數問題   | 經典問題 |
|---|------|
| 有一個小偷有容量 $V$ 的背包，考慮 $N$ 個物品，每個物品的體積不同，但我們今天不問小偷能不能塞滿這個背包，我們想知道它有多少種方法把這個背包塞滿，請在 $\mathcal{O}(NV)$ 時間內對於每種 $V$ 都輸出該答案，如果今天方法 A 所塞的物品，方法 B 都有了，而且方法 B 所塞的物品，方法 A 都有了，那我們就說方法 A 跟方法 B 是一樣的，不然這兩種方法就是不一樣的 |      |

令第  $i$  個物品重量為  $w_i$ ，那答案就是  $[x^V] \prod_{i=0}^{N-1} (1 + x^{w_i})$  這其實就是 dp 式的表現，令  $\text{dp}[i][v]$  為前  $i$  個選項中，容量為  $v$  的答案：則  $\text{dp}[i][v] = \text{dp}[i-1][v-w_i] + \text{dp}[i-1][v]$  對應到的生成函數是

$$\prod_{j=0}^{i-1} (1 + x^{w_j}) \times (1 + x^{w_i})$$

**定理 11.5.** 一個生成函數函數乘上  $(1 + x + x^2 + x^3 + x^4 + \dots)$ ，會變成它本身的前綴和

考慮到幾何級數，有時候我們會把生成函數  $(1 + x + x^2 + x^3 + x^4 + \dots)$  寫成  $\frac{1}{1-x}$ （因為在  $x$  絕對值小於一時兩邊是一樣的）想要把  $I$  個區間都加  $k_i$ ，而題目只有在最後做 query，利用生成函數的思想，不必用線段樹就可以把複雜度做到  $O(N + I)$ ，簡單又好寫：留意到

$$\begin{aligned}
 (x^n + x^{n+1} + x^{n+2} + \dots x^m) &= (1 + x + x^2 + x^3 + x^4 + \dots)(x^n - x^{m+1}) \\
 k_1(x^{n_1} + x^{n_1+1} + \dots x^{m_1}) + k_2(x^{n_2} + x^{n_2+1} + \dots x^{m_2}) + \dots \\
 &:= \sum_{i=1}^I k_i \sum_{j=n_i}^{m_i} x^j = \sum_{i=1}^I k_i \frac{(x^n - x^{m+1})}{1-x} = \frac{1}{1-x} \sum_{i=1}^I k_i (x^n - x^{m+1})
 \end{aligned}$$

告訴我們可以先在陣列  $n_i, m_{i+1}$  個別加上  $k_i, -k_i$ ，最後再做前綴和即得到答案

| 二項式恆等式   | 證明題 |
|--|-----|
| 利用生成函數證明：<br>$\sum_{i=0}^r \binom{n}{i} \binom{m}{r-i} = \binom{m+n}{r}$ |     |

那為什麼要提到生成函數呢，它的功用主要在於計數分析，不過有一部分計數的問題需要用到生成函數乘積，這時候一個好的多項式乘法 algo 會變得相當重要，因為要算  $n, m$  次多項式  $P(x), Q(x)$  相乘的時候，如果沒有特別好的條件的話，生成函數相乘硬解的複雜度會是  $\mathcal{O}(n^2)$ ，但假設現在我們知道了  $\{(x_i, P(x_i)Q(x_i))\}_{i=1}^{n+m+1}$ ，利用等會兒提到的多項式插值唯一定理，可以找回  $P(x)Q(x)$

### 11.2.2 離散傅立葉變換 Discrete Fourier Transform

在進入離散傅立葉變換 (DFT) 之前，先看個引理吧

**引理 11.6** (多項式插值唯一定理 Uniqueness of an interpolating polynomial). 一個平面上一堆點  $\{(x_i, y_i)\}_{i=0}^{n-1}$ ， $x_i$  們各不相同，存在唯一一個多項式  $P(x)$  使得  $P(x_i) = y_i$ ，使得多項式最高次項小於  $n$

*Proof.* 假設我們想要知道的多項式是  $P(z) = a_0 + a_1z + a_2z^2 + \dots + a_{n-1}z^{n-1}$  問題就變成了

1.  $a_0, a_1, \dots, a_{n-1}$  是什麼

2. 解為什麼是唯一的

我們有恆等式：

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ P(x_2) \\ \vdots \\ P(x_{n-1}) \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

左邊那個矩陣，我們叫它范德蒙矩陣 (Vandermonde matrix)。注意到上式中， $y_i, x_i$  我們已經有了，看來我們還需要擅長取反矩陣找  $a_i$  的朋友呢。

等等，這矩陣是可逆的嗎？

可以配合歸納法來證明：

$$\det \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} = \prod_{0 \leq i < j < n} (x_j - x_i)$$

因為  $x_i$  們是各不相同，這個矩陣的行列式 ( $\det$ ) 不為零，當然是可逆的，而且這同時也告訴我們  $a_0, a_1, \dots, a_{n-1}$  是唯一決定的！  $\square$

高斯消去法取反矩陣要花  $\mathcal{O}(n^3)$ ，拉格朗日插值法要花  $\mathcal{O}(n^2)$ ，然而，如果  $x_i$  選得很特別，有辦法做到  $\mathcal{O}(n \ln n)$  時間在  $a_i, y_i$  間快速轉換，這個我們叫它 FFT，在介紹 FFT 以前，先從 DFT 來下手。

假設有一種數字  $x$ ，對於  $n$  它滿足  $\sum_{i=0}^{n-1} x^i = 0$ ，除了零以外真的有這種鬼數字存在嗎？有的！

考慮對於  $nd + 1$  的質數  $p$ ，討論  $\mathbb{Z}_p^*$  時，這種鬼數字會存在，或者是不真實 (real) 而複雜 (complex) 的數，我們叫它複數  $\mathbb{C}$ 。

先說第一種情況：

**定理 11.7.** 若  $p$  是一個  $nd + 1$  的質數， $a$  是一個原根，令  $b = a^d$ ，則  $b$  的序為  $n$

*Proof.* 令  $x$  為  $b$  的序

若  $x$  比  $n$  小，則  $b^x = a^{xd} = 1$ ，則  $a$  不是原根，因為它的序比  $nd$  小，矛盾。

而  $b^n = a^{nd} = a^{p-1} = 1$  表示  $b$  的序比  $n$  小或等於  $n$ ，因此  $b$  的序是  $n$   $\square$

在此  $\sum_{i=0}^{n-1} x^i = 0$  的非零解是什麼呢？答案是在模  $p$  下的  $b^k, k = 1, 2, \dots, n-1$ 。為了標記方便，對於  $\mathbb{Z}_p^*$  的元素，我們在這章節統稱共軛為反元素，以  $\bar{b} = b^{-1}$  表示。

對於第二種情形，或許已經有人知道複數的概念，但這裡再提一下，複數是一個平面  $\mathbb{C} = \mathbb{R} + i\mathbb{R}$ ，其中  $i \times i = -1$ ，複數支援加減乘除，若  $A = a_1 + ia_2, B = b_1 + ib_2$ ，則

1. 我們稱  $A$  的共軛複數為  $\bar{A} = a_1 - ia_2$
2.  $A + B = (a_1 + b_1) + i(a_2 + b_2)$
3.  $A - B = (a_1 - b_1) + i(a_2 - b_2)$
4.  $A \times B = (a_1 b_1 - a_2 b_2) + i(a_1 b_2 + a_2 b_1)$
5.  $A \div B = \frac{a_1 + ia_2}{b_1 + ib_2} = \frac{(a_1 + ia_2)(b_1 - ib_2)}{(b_1 - ib_2)(b_1 + ib_2)} = \frac{(a_1 b_1 + a_2 b_2) + i(a_2 b_1 - a_1 b_2)}{b_1^2 + b_2^2}$

物件 `std::complex` 支援這些運算，包括三角函數的使用。

可以用 xy 座標畫出複數的點，一個複數的絕對值為它到原點  $0 + i0$  的歐幾里德距離（用尺量出來的那個距離）

這些數字都不是自然存在的數，你的午餐價格不可能是  $3 + 2i$ ，但就是因為複數有很好用的結構（複結構），才會被發展出來做解方程式的根、FFT 之類的事

在此  $\sum_{i=0}^{n-1} x^i = 0$  的非零解是什麼呢？答案是  $e^{\frac{2\pi ik}{n}}, k = 1, 2, \dots, n-1$ ，這些數的共軛跟反元素是一樣的

我們有歐拉公式 Euler Formula :  $e^{ix} = \cos x + i \sin x$

這是展開式  $e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$  在複平面上推廣所得出的結果。

如果是  $\mathbb{Z}_p^*$  我們令  $w_N^k = b^k$ ，如果是  $\mathbb{C}$  上我們令  $w_N^k = e^{2\pi i \frac{k}{N}}$ （說穿了就只是將 1 對著原點逆時針旋轉  $\frac{k}{N}$  個圓周的複數，如果  $k < 0$  就代表順時針旋轉  $\frac{|k|}{N}$  個圓周），為了直觀我們就用圓餅圖示範（下表），這裡我們叫它「派運算」，派運算表示時為了方便起見  $N$  都是 8，值得一提的是，派運算不管是對於  $\mathbb{Z}_p^*$  上的  $w_N = b$ ，還是  $\mathbb{C}$  上的  $w_N = e^{\frac{2\pi i}{N}}$ ，下面敘述都是滿足的，因此講者以下不再區分兩者的差別：

| $w_8^0$<br>或 $\odot$ | $w_8^1$<br>$\odot$ | $w_8^2$<br>$\odot\bullet$ | $w_8^3$<br>$\bullet\odot$ | $w_8^4$<br>$\bullet\bullet$ | $w_8^5$<br>$\bullet\odot$ | $w_8^6$<br>$\odot\bullet$ | $w_8^7$<br>$\odot\bullet\bullet$ |
|----------------------|--------------------|---------------------------|---------------------------|-----------------------------|---------------------------|---------------------------|----------------------------------|
|----------------------|--------------------|---------------------------|---------------------------|-----------------------------|---------------------------|---------------------------|----------------------------------|

表 11.2：派-複數根對照表

**定理 11.8.**  $w_N^i$  與派運算有以下規則

1.  $w_N^i \times w_N^j = w_N^{i+j}$ ，在派運算下就是將面積相加，例如  $\odot\odot = \odot$

2.  $(w_N^i)^j = w_N^{ij}$  · 在派運算下就是將面積相乘上一個常數 · 例如  $\bigodot^3 = \bigodot$
3.  $w_N^{-i}$  是  $w_N^i$  的共軛 · 在派運算下就是將一塊完整的派拿走原來派的面積 · 例如  $\overline{\bigodot} = \bullet$
4.  $w_N^i = w_N^{i+N}$  · 在派運算下就是將面積模一塊完整的派 · 例如  $\bigodot \bullet \bullet = \bigodot$
5.  $w_{Nk}^{ik} = w_N^i$  · 在派運算下就是指  $\bigodot$  切成四塊或是八塊面積都是一樣的：  
 $\bigodot = \bigodot \bigodot \bigodot \bigodot = \bigodot \bigodot$
6.  $\sum_{k=0}^{N-1} (w_N^i)^k = N[N|i]$  · 在派運算下就像是  $\sum_{k=0}^{N-1} (\bigodot)^k = 0$

*Proof.* 我們證最後一點的派式子 · 若  $i \neq 0 \pmod{N}$  · 可以發現：

$$\begin{aligned}
 & (\bullet - \bigodot)(\bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bigodot) \\
 &= (\bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bullet + \bullet) - (\bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bullet + \bullet + \bullet) \\
 &= \bigodot - \bullet = 0
 \end{aligned}$$

因此  $(\bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bigodot + \bullet + \bullet) = \frac{\bigodot - \bullet}{\bullet - \bigodot} = 0$   $\square$

注意  $3\bigodot \neq \bullet$  而是應該要像向量一樣直接寫成  $3\bigodot$

**定義 11.8.**  $w_N^0, w_N^1, \dots, w_N^{N-1}$  所構成的范德蒙矩陣 · 我們叫它 DFT 矩陣 · 以  $D_N$  簡寫

$$D_N = \begin{pmatrix} 1 & w_N^0 & w_N^{0*2} & \cdots & w_N^{0*(N-1)} \\ 1 & w_N^1 & w_N^{1*2} & \cdots & w_N^{1*(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_N^{N-1} & w_N^{(N-1)*2} & \cdots & w_N^{(N-1)*(N-1)} \end{pmatrix} = \begin{pmatrix} \bigodot & \bigodot & \bigodot & \cdots & \bigodot \\ \bigodot & \bullet & \bullet & \cdots & \bullet \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bigodot & \bullet & \bullet & \cdots & \bigodot \end{pmatrix}$$

是個漂亮的對稱矩陣呢！

它的反矩陣長這樣：

$$D_N^{-1} = \frac{1}{N} \begin{pmatrix} 1 & w_N^{-0} & w_N^{-0*2} & \dots & w_N^{-0*(N-1)} \\ 1 & w_N^{-1} & w_N^{-1*2} & \dots & w_N^{-1*(N-1)} \\ \vdots & \vdots & \ddots & & \vdots \\ 1 & w_N^{1-N} & w_N^{(1-N)*2} & \dots & w_N^{(1-N)*(N-1)} \end{pmatrix} = \frac{1}{N} \begin{pmatrix} \bullet & \bullet & \bullet & \dots & \bullet \\ \bullet & \circlearrowleft & \circlearrowleft & \dots & \circlearrowright \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bullet & \circlearrowright & \circlearrowleft & \dots & \bullet \end{pmatrix}$$

簡單來說  $D_n$  的反矩陣是  $D_n$  每個項取共轭再除以  $N$ ，如果學員學過（或未來會學到）線性代數，就可以知道這是一個 Unitary Matrix（忽略常數），自己乘乘看是不是單位矩陣吧。

從上述的定理我們可以知道一個小於  $n$  次的多項式的呈現，除了可以用  $P(z) = a_0 + a_1z + a_2z^2 + \dots + a_{n-1}z^{n-1}$  表示，還可以用  $n$  個點來代表一個唯一的多項式，這個表示法叫做點值表示法（Point Value Representation）

發現了嗎，如果有兩個多項式的點值表示法，而它們選用的那些  $x_i$  都一樣，要怎麼得到兩個多項式相乘的點值式呢？直接把每個  $y$  座標都相乘就好了對吧。

### 演算法 11.2 離散傅立葉變換

- 
- 1: 先用  $D_N$  矩陣乘積將  $P(\bullet)$ ,  $P(\circlearrowleft)$ ,  $P(\circlearrowright)$ , ...,  $P(\bullet)$  ·  $Q(\bullet)$ ,  $Q(\circlearrowleft)$ ,  $Q(\circlearrowright)$ , ...,  $Q(\bullet)$  算出來。
  - 2: 對於每個  $\circlearrowleft^i$  直接相乘兩個點值表示法的多項式  $P(\bullet)Q(\bullet)$  ·  $P(\circlearrowleft)Q(\circlearrowleft)$  · ... ·  $P(\bullet)Q(\bullet)$
  - 3: 用反矩陣  $D_N^{-1}$  將將  $P(x)Q(x)$  的  $x^i$  每一項算出
- 

這個算法是  $\mathcal{O}(n^2)$ ，門檻在 1.3. 步驟，但待會兒 FFT 我們會利用一點小技巧加速到  $\mathcal{O}(n \ln n)$ ，這裡注意  $P(x)Q(x)$  的次數最高只能是  $N - 1$ ，否則會得到唯一一個多項式  $H(x)$  使得  $H(\circlearrowleft^i) = P(\circlearrowleft^i)Q(\circlearrowleft^i)$  且次數小於  $N$  的多項式。

### 11.2.3 快速傅立葉變換 Fast Fourier Transformation

「快速傅立葉變換」聽起來真是嚇死人了，好像很難的樣子，可以叫他「利用奇怪矩陣的特性所做的加速來算特定幾個點的值之演算法」，講白了就是比較快的 DFT，利用 DFT 中， $D_N$  的特性所做的加速，它的輸入輸出跟 DFT 沒什麼差別，除了：

1.  $N$  一定要是小的質數的乘積，這裡我們指 2 的次方
2. 比較快，DFT 是  $\mathcal{O}(n^2)$ ，FFT 只要  $\mathcal{O}(n \ln n)$

為了方便說明，我們這裡一樣舉  $N = 8$  為例：令

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

要算出

$$\begin{pmatrix} P(\bigcirc) \\ P(\bigodot) \\ P(\bigoplus) \\ P(\bigodot) \\ P(\bigominus) \\ P(\bigotimes) \\ P(\bigoplus) \\ P(\bigotimes) \end{pmatrix} = \begin{pmatrix} \bigcirc & \bigcirc \\ \bigcirc & \bigodot & \bigoplus & \bigodot & \bigominus & \bigotimes & \bigoplus & \bigotimes \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigcirc & \bigoplus & \bigoplus \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigotimes & \bigoplus & \bigoplus \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigcirc & \bigominus & \bigoplus \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigotimes & \bigoplus & \bigoplus \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigotimes & \bigoplus & \bigoplus \\ \bigcirc & \bigodot & \bigoplus & \bigominus & \bigcirc & \bigotimes & \bigoplus & \bigoplus \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}$$

需要這些資訊：

$$P^{[0]}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$$

$$P^{[1]}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$$

則

$$P(x) = P^{[0]}(x^2) + x \cdot P^{[1]}(x^2)$$

看起來我們總共要算出

$$P^{[0]}(\bigcirc^2), P^{[0]}(\bigodot^2), P^{[0]}(\bigoplus^2), P^{[0]}(\bigodot^2), P^{[0]}(\bigominus^2), P^{[0]}(\bigotimes^2), P^{[0]}(\bigoplus^2), P^{[0]}(\bigotimes^2)$$

$$P^{[1]}(\bigcirc^2), P^{[1]}(\bigodot^2), P^{[1]}(\bigoplus^2), P^{[1]}(\bigodot^2), P^{[1]}(\bigominus^2), P^{[1]}(\bigotimes^2), P^{[1]}(\bigoplus^2), P^{[1]}(\bigotimes^2)$$

把平方乘進去發現**扣掉重複**的只有

$$P^{[0]}(\bigcirc), P^{[0]}(\bigodot), P^{[0]}(\bigominus), P^{[0]}(\bigotimes)$$

$$P^{[1]}(\bigcirc), P^{[1]}(\bigodot), P^{[1]}(\bigominus), P^{[1]}(\bigotimes)$$

意思是說我們只要算出

$$\begin{pmatrix} P^{[0]}(\bigcirc) \\ P^{[0]}(\bullet) \\ P^{[0]}(\ominus) \\ P^{[0]}(\ominus\bullet) \end{pmatrix} = \begin{pmatrix} \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bullet & \ominus & \bullet \\ \bigcirc & \ominus & \bigcirc & \ominus \\ \bigcirc & \bullet & \ominus & \bigcirc \end{pmatrix} \begin{pmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \end{pmatrix}$$

$$\begin{pmatrix} P^{[1]}(\bigcirc) \\ P^{[1]}(\bullet) \\ P^{[1]}(\ominus) \\ P^{[1]}(\ominus\bullet) \end{pmatrix} = \begin{pmatrix} \bigcirc & \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bullet & \ominus & \bullet \\ \bigcirc & \ominus & \bigcirc & \ominus \\ \bigcirc & \bullet & \ominus & \bigcirc \end{pmatrix} \begin{pmatrix} a_1 \\ a_3 \\ a_5 \\ a_7 \end{pmatrix}$$

便可以在  $\mathcal{O}(n)$  時間內求出原來的式子。

只要把一個 FFT 問題分解成兩個小的 FFT 問題，需要的運算量一瞬間少了一半

當問題變得最小時，要解的矩陣像這樣

$$(\bigcirc)(a_0) = (a_0)$$

寫成複雜度分析式：

$$T(N) = 2T\left(\frac{N}{2}\right) + \Theta(N)$$

FFT 的複雜度分析跟 MergeSort 相同，結果是  $\Theta(N \ln N)$ 。乘上  $D_N^{-1}$  作反 FFT 時，注意到  $D_N^{-1} = \frac{1}{N} \overline{D_N}$ ，因此改一個小小的根（改成反元素），然後結果再乘上  $\frac{1}{N}$  即可完成反 FFT。

下列程式碼比較使用 FFT 算跟剛剛三個步驟，並且利用迭代加速，這邊使用同一套模板就能夠同時實作 FFT 跟 NTT：

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 template <int mod>
5 struct ModInt {
6     int val;
7     int trim(int x) const { return x >= mod ? x - mod : x < 0 ? x + mod : x; }
8     ModInt(int v = 0) : val(trim(v % mod)) {}
9     ModInt(long long v) : val(trim(v % mod)) {}
10    ModInt &operator=(int v) { return val = trim(v % mod), *this; }
11    ModInt &operator=(const ModInt &oth) = default;
12    ModInt operator+(const ModInt &oth) const { return trim(val + oth.val); }
13    ModInt operator-(const ModInt &oth) const { return trim(val - oth.val); }
14    ModInt operator*(const ModInt &oth) const { return 1LL * val * oth.val % mod; }

```

```

15  ModInt operator/(const ModInt &oth) const {
16      function<int(int, int, int, int)> modinv = [&](int a, int b, int x,
17          int y) {
18          if (b == 0) return trim(x);
19          return modinv(b, a - a / b * b, y, x - a / b * y);
20      };
21      return *this * modinv(oth.val, mod, 1, 0);
22  }
23  bool operator==(const ModInt &oth) const { return val == oth.val; }
24  ModInt operator-() const { return trim(mod - val); }
25  template<typename T> ModInt pow(T pw) {
26      bool sgn = false;
27      if (pw < 0) pw = -pw, sgn = true;
28      ModInt ans = 1;
29      for (ModInt cur = val; pw; pw >>= 1, cur = cur * cur) {
30          if (pw&1) ans = ans * cur;
31      }
32      return sgn ? ModInt{1} / ans : ans;
33  };
34
35 /* p == (a << n) + 1
36    g = pow(root, (p - 1) / n)
37    n   1<<n      p           a       root
38    5     32        97         3       5
39    6     64        193        3       5
40    7     128       257        2       3
41    8     256       257        1       3
42    9     512       7681       15      17
43   10    1024      12289      12      11
44   11    2048      12289      6       11
45   12    4096      12289      3       11
46   13    8192      40961      5       3
47   14   16384      65537      4       3
48   15   32768      65537      2       3
49   16   65536      65537      1       3
50   17  131072      786433     6       10
51   18  262144      786433     3       10 (605028353, 2308, 3)
52   19  524288      5767169    11      3
53   20  1048576     7340033    7       3
54   20  1048576     998244353   952     3
55   21  2097152     23068673   11      3
56   22  4194304     104857601  25      3
57   23  8388608     167772161  20      3
58   24  16777216     167772161  10      3
59   25  33554432     167772161  5       3 (1107296257, 33, 10)
60   26  67108864     469762049  7       3
61 */
62
63 // w = root^a mod p for NTT
64 // w = exp(-complex<double>(0, 2) * PI / N) for FFT

```

```

65
66 template<typename F = complex<double>>
67 vector<F> FFT(vector<F> P, F w, bool inv = 0) {
68     int n = P.size();
69     int lg = __builtin_ctz(n);
70     assert(__builtin_popcount(n) == 1);
71
72     for (int j = 1, i = 0; j < n - 1; ++j) {
73         for (int k = n >> 1; k > (i ^= k); k >>= 1);
74         if (j < i) swap(P[i], P[j]);
75     } //bit reverse
76
77     vector<F> ws = {inv ? F{1} / w : w};
78     for (int i = 1; i < lg; ++i) ws.push_back(ws[i - 1] * ws[i - 1]);
79     reverse(ws.begin(), ws.end());
80
81     for (int i = 0; i < lg; ++i) {
82         for (int k = 0; k < n; k += 2<<i) {
83             F base = F{1};
84             for (int j = k; j < k + (1<<i); ++j, base = base * ws[i]) {
85                 auto t = base * P[j + (1<<i)];
86                 auto u = P[j];
87                 P[j] = u + t;
88                 P[j + (1<<i)] = u - t;
89             }
90         }
91     }
92
93     if (inv) for_each(P.begin(), P.end(), [&](F& a) { a = a / F(n); });
94     return P;
95 } //faster performance with calling by reference
96
97 int main() {
98     const int N = 1<<20;
99     {
100         const int MOD = 998244353;
101         using mint = ModInt<MOD>;
102         const auto w = mint(3).pow(952);
103         vector<mint> P(N), Q(N), R;
104         for (int i = 0; i < 10; ++i) Q[i] = 2, P[i] = 1;
105         P = FFT<mint>(P, w, false);
106         Q = FFT<mint>(Q, w, false);
107         transform(begin(P), end(P), begin(Q), back_inserter(R),
108                   std::multiplies<mint>());
109         R = FFT<mint>(R, w, true);
110         for (int i = 0; i < 20; ++i) cout << R[i].val << " \n"[i + 1 == 20];
111     }
112     {
113         const double PI = acos(-1);
114         const auto w = exp(-complex<double>(0, 2. * PI / N));
115         vector<complex<double>> P(N), Q(N), R;

```

```

116     for (int i = 0; i < 10; ++i) Q[i] = 2, P[i] = 1;
117     P = FFT(P, w, false);
118     Q = FFT(Q, w, false);
119     transform(begin(P), end(P), begin(Q), back_inserter(R),
120               std::multiplies<complex<double>>());
121     R = FFT(R, w, true);
122     for (int i = 0; i < 20; ++i) cout << R[i] << " \n"[i + 1 == 20];
123 }
124 }
```

**程式碼 11.1:** FFT: 快速乘 DFT 矩陣的實現

<https://gist.github.com/rareone/73e4f9cd395fd48e9d27617bfbf00193>

|  |           |
|--|-----------|
| Golf Bot   | UVa 12879 |
| 非負整數集 $G = \{0, a_1, a_2, a_3, \dots, a_n\}$ , $a_n \leq 2 \times 10^5$ · 可以從 $G$ 選兩個元素 (可重複挑選同一種元素) · 請問有沒有辦法選出兩個元素使得它們的和為 $m$ ? 對於各種指定的 $m$ 請輸出有沒有辦法達成 |           |

這題可以用 `std::bitset` 寫  $\mathcal{O}(n^2)$  的算法 · 但這題若改成 ·

|  |             |
|--|-------------|
| Golf Bot 改   | UVa 12879 改 |
| 非負整數集 $G = \{0, a_1, a_2, a_3, \dots, a_n\}$ , $a_n \leq 2 \times 10^5$ · 可以從 $G$ 選兩個元素 (可重複挑選同一種元素) · 請問有沒有辦法選出兩個元素使得它們的和為 $m$ ? 對於各種指定的 $m$ 請輸出有幾種辦法達成 · 答案模 998244353 |             |

就使用 FFT 吧 !

下面有一題可以使用 FFT 以及快速幕就可以算出來的題目

|  |         |
|--|---------|
| Thief in a Shop  | CF 632E |
| 一個小偷要從商店偷剛好 $k$ 個物品 · 有 $N$ 種物品 · 每種物品都有無限多個 · 第 $i$ 種物品價值 $a_i$ 元 · 問他離開商店時 · 肩包裡的物品價值可能是多少 ? |         |



# 基礎計算幾何 Basic Computational Geometry

計算幾何融合了解析幾何和演算法，一般的幾何問題到電腦中必須要用數值的方式表示並計算。除了點、線、面要如何表示之外還要解決浮點數的精度誤差，因此在計算上需要用各種方法避免誤差，演算法也會根據精度上限來做設計。

本單元中討論的是二維座標平面上的幾何，一方面是因為大部分高維度的問題都會遇到所謂維數災難 (Curse of Dimensionality)，一方面競賽比較常出。

本章內容部分靈感來自於 CF 文章，文章放在 Reference。首先介紹向量的概念，透過向量計算有向面積，進而判斷出線段是否相交以及求出交點；然後提到掃描線的概念，接著帶到多邊形，而多邊形我們會在凸包上做多點文章，此外會討論平面最遠點對為例子介紹旋轉卡尺的概念。

本章還提供了兩個跟幾何有關的優化方法，用來處理 DP 類別的問題，這邊會提到可拓展集合的方法，可以用來輔助如倒數第二節提到的最近點對問題。

最後會提到幾何的對偶性質。

因為是競賽導向，我們首先要求的是正確性，在此會選擇主要以整數做幾何。

## 12.1 線性代數

這邊介紹一些向量空間跟一些非歐幾何觀念供讀者當參考。

**定義 12.1.** 我們說一個向量空間  $\mathbb{V}$  對應到一個純量場  $\mathbb{F}$ ，滿足以下公理：

1. 向量加法滿足結合率： $u + (v + w) = (u + v) + w$
2. 向量加法滿足交換率： $u + v = v + u$
3. 向量加法零元素存在： $\exists 0, 0 + v = v, v \in \mathbb{V}$

4. 向量加法反元素存在 :  $\forall v \in \mathbb{V}, \exists -v : v + (-v) = 0$
5. 純量對向量乘法與純量的體乘法相容 :  $ab(v) = a(bv), a, b \in \mathbb{F}, v \in \mathbb{V}$
6. 體單位元素  $1 \in \mathbb{F}$  滿足 :  $1v = v, v \in \mathbb{V}$
7. 純量乘法對向量加法有分配律 :  $a(u + v) = au + av, a \in \mathbb{F}, u, v \in \mathbb{V}$
8. 純量乘法對體加法有分配律 :  $(a + b)u = au + bu, a, b \in \mathbb{F}, u \in \mathbb{V}$

向量加法是封閉性的 ( $u, v \in \mathbb{V} \implies u + v \in \mathbb{V}$ ) · 向量對純量的乘法是封閉性的 ( $u \in \mathbb{V}, a \in \mathbb{F} \implies au \in \mathbb{V}$ )。

這邊舉幾個例子：

考慮  $\mathbb{Z}_2^3$  對應到純量場  $\mathbb{Z}_2$ ，這是一個向量空間 · 向量加法定義為 xor · 檢查以上公理是否滿足：

1. e.g.  $(010) + (110) = (010) \oplus (110) = (101)$
2.  $\oplus$  滿足交換率
3. 零元素 :  $(000)$
4. 任何向量的加法反元素是自己
5. 純量對向量乘法與純量的體乘法相容 :  $ab(v) = a(bv) = \begin{cases} v & a = b = 1 \\ 0 & \text{otherwise} \end{cases}$
6.  $1 \in \mathbb{Z}_2 : 1v = v, v \in \mathbb{V}$
7. 略
8. 略

考慮區間  $[a, b]$  的實連續函數  $\mathcal{C}[a, b]$  對應到純量場  $\mathbb{R}$ ，向量加法定義為  $(f + g)(x) = f(x) + g(x)$  · 零元素定義為  $x \mapsto 0$  ·  $f$  的反元素定義為  $(-f)(x) = -f(x)$  · 這是一個向量空間。

考慮最高項不大於  $n$  的多項式  $P^n$  對應到實數  $\mathbb{R}$ ，是一個向量空間。

考慮  $\mathbb{C} = \{a + ib \mid a, b \in \mathbb{R}\}$  對應到實數  $\mathbb{R}$ ，是一個向量空間。

考慮  $\mathbb{Z}_6^n$  對應到  $\mathbb{Z}_6$  不是一個向量空間，因為  $\mathbb{Z}_6$  不是一個場 (field)。

| The crazy vector space  | 證明 |
|---|----|
| 令 $\mathbb{V} = \{(x, y) : x, y \in \mathbb{R}\}$ · 定義加法 $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2 + 1, y_1 + y_2 + 1)$ · 定義純量乘法 $a(x, y) = (ax + a - 1, ay + a - 1)$ 找出零向量跟反元素規則 · 並證明這是一個向量空間 |    |

我們常用向量空間  $\mathbb{R}^n$  來代表座標維度  $n$  為空間，其中的點可以寫作  $(x_0, x_1, \dots, x_{n-1})$ 。我們會用  $e_i$  來代表基礎座標，像是平面二維空間中  $(\mathbb{R}^2)$   $e_1$  表示 X 軸， $e_2$  表示 Y 軸。

點支援加法

$$(x_0, \dots, x_{n-1}) + (y_0, \dots, y_{n-1}) = (x_0 + y_0, \dots, x_{n-1} + y_{n-1})$$

點支援純量乘法

$$k(x_0, \dots, x_{n-1}) = (kx_0, \dots, kx_{n-1})$$

**定理 12.1.** 向量空間  $\mathbb{V}$  對應到  $\mathbb{F}$  滿足以下性質，證明省略：

1. 零向量跟反元素是唯一的
2. 任何純量乘以零元素跟純量零乘以任何向量都是零向量
3.  $-u = (-1)u, u \in \mathbb{V}$
4. 若  $au = 0, a \in \mathbb{F}, u \in \mathbb{V}$  則  $a = 0$  或  $u = 0$
5. 向量卡氏積 (Cartesian Product)  $\mathbb{V} \times \mathbb{V}$  是向量空間
6.  $\mathbb{F}$  是一個向量空間

### 12.1.1 線性變換 Linear Map

**定義 12.2.** 我們說一個向量空間打到另一個向量空間對應到純亮場  $\mathbb{F}$  的函數  $f : \mathbb{V} \rightarrow \mathbb{W}$  是線性變換，如果：

$$1. f(u) + f(v) = f(u + v), u, v \in \mathbb{V}$$

$$2. f(cu) = cf(u), c \in \mathbb{F}, u \in \mathbb{V}$$

**定理 12.2.** 線性變換滿足以下性質：

$$1. f(c_1u_1 + \cdots + c_nu_n) = c_1f(u_1) + \cdots + c_nf(u_n)$$

$$2. f(0_{\mathbb{V}}) = 0_{\mathbb{W}}$$

一般的計算幾何問題會將問題放在  $\mathbb{R}^n$ ，特別在競賽中大多的問題會放在  $\mathbb{R}^2$ ，因為越高維度的問題會越困難（見 Curse of Dimensionality），我們本篇文章主要以  $\mathbb{R}^2$  平面幾何為主。

我們討論  $\mathbb{R}^2 \rightarrow \mathbb{R}^2$  的線性變換，把一個向量

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

打到

$$\mathbf{x}' = \mathbf{Ax}$$

其中  $\mathbf{A}$  是一個  $2 \times 2$  的矩陣

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

這一類的線性變換有一個特徵是零點不會平移，考慮乘完  $\mathbf{A}$  後加上平移  $\mathbf{c}$ ，我們會把問題變成：

$$\mathbf{x}' = \mathbf{Ax} + \mathbf{c}$$

上述的公式可以改寫為：

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{A} & \mathbf{c} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}$$

這一類別的變換稱作仿射變換（Affine Transformation）。

### 12.1.2 平面點

這裡筆者會引用 AI.Cash 的一篇文章。我們要先以物件方式實做點集，因為不希望基礎操作會花太多空間在我們的主要大函數上（比如最大的 main），也會避免

重複的程式碼，因此把一些實作細節寫成物件，這些最基礎的操作定義好之後，對後面更大更抽象的程式碼的理解會好很多。不用害怕因為程式碼長就容易出錯，實作細節很簡單也很好除錯。

```
1 template<typename T>
2 struct Point {
3     T x, y;
4     Point() : x(0), y(0) {}
5     Point(const T x, const T y) : x(x), y(y) {}
6     template <class F> explicit operator Point<F> () const {
7         return Point<F>((F)x, (F)y);
8     }
9
10    Point operator+(const Point b) const {
11        return Point(x + b.x, y + b.y);
12    }
13    Point operator-(const Point b) const {
14        return Point(x - b.x, y - b.y);
15    }
16    template <class F> Point<F> operator*(const F fac) {
17        return Point<F>(x * fac, y * fac);
18    }
19    template <class F> Point<F> operator/(const F fac) {
20        return Point<F>(x / fac, y / fac);
21    }
22
23    T operator&(const Point b) const { return x * b.x + y * b.y; }
24    // 內積運算子
25    T operator^(const Point b) const { return x * b.y - y * b.x; }
26    // 外積運算子
27
28    bool operator==(const Point b) const {
29        return x == b.x and y == b.y;
30    }
31    bool operator<(const Point b) const {
32        return x == b.x? y < b.y: x < b.x;
33    } // 字典序
34
35    Point operator-() const { return Point(-x, -y); }
36    T norm() const { return *this & *this; } // 歐式長度平方
37    Point prep() const { return Point(-y, x); } // 左旋直角法向量
38};
```

程式碼 12.1: 平面向量與點類

事實上，根本不用寫註解，相信讀者也會發現這只是二維版本的加減乘除。關於 type 的使用，建議是：能避免浮點數誤差就盡量避免，包括避免利用線段交點、算平方根，特別是三角函數這種又慢又有誤差的函數。但是使用 double 可以用 eps 判誤差啊？別慌，我們一樣也可以把浮點數相等判斷包成一個 typename，在這之前我們只專心搞幾何。

為了方便 debug 這裡也提供可用的 IO 操作

```
1 template<class F> istream& operator>>(istream& is, Point<F> &pt) {  
2     return is >> pt.x >> pt.y;  
3 }  
4 template <class F> ostream& operator<<(ostream& os, const Point<F>& pt) {  
5     return os << pt.x << " " << pt.y;  
6 }
```

程式碼 12.2: 平面向量 IO 操作

### 12.1.3 內積與外積

#### 內積

**定義 12.3.** 數學上的內積是一個線性變換  $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}$ ，我們通常討論  $\mathbb{F} = \mathbb{C}$  或  $\mathbb{F} = \mathbb{R}$  滿足以下定義：

1. 共軛對稱  $\langle x, y \rangle = \overline{\langle y, x \rangle}$
2. 對第一個參數線性  $\langle ax + z, y \rangle = a\langle x, y \rangle + \langle z, y \rangle$
3. 正定  $\langle x, x \rangle > 0, x \neq 0$

**定理 12.3.** 內積空間函數  $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}$  滿足下面性質：

1. 自動對第二個參數線性  $\langle x, ay + z \rangle = a\langle x, y \rangle + \langle x, z \rangle$
2.  $\langle w + x, y + z \rangle = \langle w, y \rangle + \langle x, y \rangle + \langle w, z \rangle + \langle x, z \rangle$

例如：兩個  $f, g : [a, b] \rightarrow \mathbb{R}$  連續函數的內積可以定義成  $\int_a^b f(x)g(x)dx$

高中講的兩個平面座標內積是一個特例，把每一個分量的乘積相加。關於內積操作，一個直觀看法是說明了兩個向量在同一個方向的程度。

任何內積空間都可以定義範數 (Norm)  $\|x\| = \sqrt{\langle x, x \rangle}$ ，對高中課綱的平面座標，就是我們常見的歐式距離。

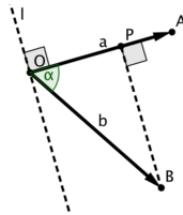


圖 12.1:  $(a, b) \cdot (c, d) = a * c + b * d$

### 外積

高中講的兩個座標外積如下圖所示，一個直觀看法是兩個向量的在逆時鐘方向的垂直程度。

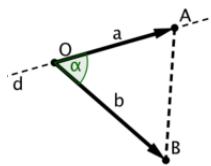


圖 12.2:  $(a, b) \times (c, d) = ad - cb$

外積在數學界有個專業名稱叫做 wedge product，可以發現外積是一個反對稱的操作，也就是  $\vec{a} \wedge \vec{b} = -\vec{b} \wedge \vec{a}$ 。

我們把這個性質延伸一下，定義函數：

$$f : \mathbb{V} \times \mathbb{V} \cdots \mathbb{V} \rightarrow \mathbb{W}$$

其中  $\mathbb{W}$  是一個由  $e_{k_1} \wedge e_{k_2} \cdots e_{k_n}$  為座標基底展開 (Span) 的向量空間， $k_i$  是一個子序列。

以三維空間  $\mathbb{R}^3$  的兩個向量外積為例，我們把像是  $u \wedge v$  寫成  $f(u, v)$ ，令  $f$  滿足以下性質：

1. 多線性  $f(ru, sv) = rs f(u, v)$
2. 反對稱  $f(u, v) = -f(v, u)$  因此  $f(u, u) = 0$

可以發現，在  $\mathbb{R}^n$  取  $k$  個向量做 wedge product 是  $\binom{n}{k}$  維度的，這就是為什麼二維空間做外積的會是一個一維的純量，而三維空間做外積會在三維上。

$\mathbb{R}^n$  空間中，如果將  $n$  個向量做 wedge product，他的維度是  $\binom{n}{n} = 1$ ，特別地，如果他的基礎座標按照  $1 \dots n$  的順序 wedge 當作單位向量，我們會幫他取一個熟悉的名字：行列式。

**定理 12.4.**  $\mathbb{R}^n$  空間中，如果將  $n$  個向量做 wedge product，那麼答案一定會跟他的行列式成線性。

## 12.2 線段

### 12.2.1 線段的表示方式

在二維座標上表示直線，可以用  $ax + by = c$  這樣的直線方程式表示，第二種表示法是兩點參數式  $A + t\overrightarrow{AB}, t \in [0, 1]$ 。通常在電腦中會使用**兩點參數式**來表示直線或是線段，更好的說法是用兩個**位置向量**來表示，這麼做有以下好處：

1. 在高維度這樣依舊不變，不用改太多程式碼
2. 只要參數改  $t$  的範圍就可以將它變成直線 ( $t \in (-\infty, \infty)$ )、射線 ( $t \in [0, \infty)$ )、線段 ( $t \in [0, 1]$ )
3. 寫成這樣比較有幾何上的直觀，而直線方程式偏向代數作法

在只有座標軸的時候，我們很容易的知道任何東西確切的位置，但是要比較兩個東西是在左還是右的時候就沒那麼容易了。想像一下在生活中不講前後左右，只講東西南北，大概是這種感覺。

我們定義：對於任何一個點  $P$ ，若滿足  $\overrightarrow{P_1 P_2} \wedge \overrightarrow{P_1 P} > 0$ ，則它所在的區域為  $\overrightarrow{P_1 P_2}$  的正方向（右手定則）。可以想成你現在站在  $P_1$ ，往  $P_2$  的方向看，左手邊的區域都屬於  $\overrightarrow{P_1 P_2}$  的正方向，如圖 12.3 所示。

有了線段的定向性的概念，我們可以定義線段的排序，排序方法為先按照弧度角  $\theta \in [0, 2\pi)$  排序，如果角度一樣我們將線段轉到箭頭朝上，左邊那個比較小，定義排序對之後要做半平面交以及進階一點的題目有幫助。首先，我們發現兩個線段的方向（終點減掉起始點）如果其中一個角度位於  $[0, \pi)$ ，另一個位於  $[\pi, 2\pi)$  那答案非常清楚，如果都在同一邊就用外積判斷，外積是零代表同向，那就看一條線的起始點在另一條左邊還是右邊。

對於點  $P_1, P_2, P$  來說， $|\overrightarrow{P_1 P_2} \wedge \overrightarrow{P_1 P}|/2$  實際就是這三個點當作頂點構成的三角形  $(P_1, P_2, P)$  面積，用海龍公式或是底乘高除以二之類的方法算都會有相同的結果。

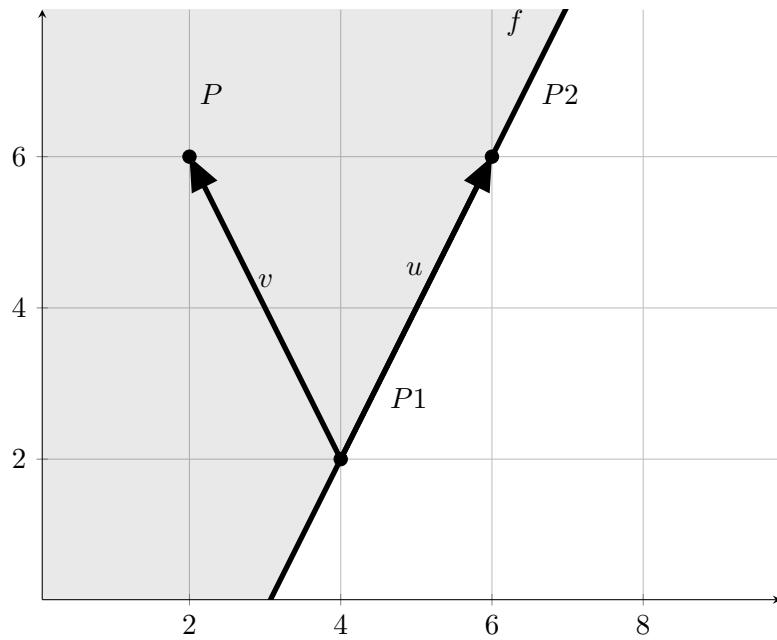


圖 12.3: 定義線段的方向後，沿著這個方向走，左手邊就稱為正方向 (塗色區域)

雖然一般來說面積一定是正的，但是外積運算的結果有可能是負的。拿掉絕對值後，如果  $P$  在直線  $\overrightarrow{P_1P_2}$  的正方向區域，也就是三角形  $(P_1, P_2, P)$  三個點的順序是逆時針順序的時會是正的，反之順時針順序時就會是負的。我們稱這種有正負號的面積為有向面積，為了方便起見，之後除非特別強調，「面積」一律指有向面積。

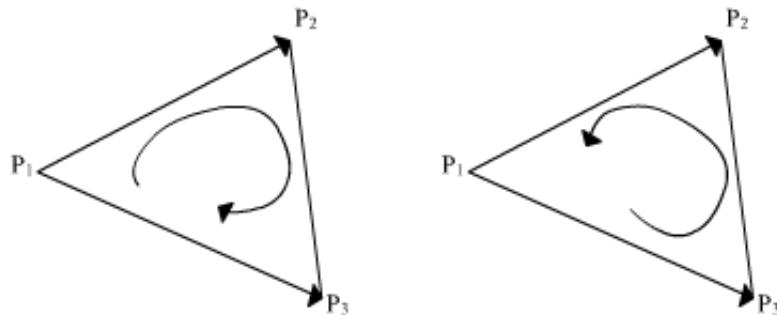


圖 12.4: 左邊的圖面積算為負，右邊的圖面積算為正

有向面積非常容易計算，可以不用考慮幾何圖形的形狀直接計算面積，考慮一個  $n$  個點簡單多邊形 (除了相鄰邊可以交於多邊形的頂點外，邊不相交的多邊形)，其點順時針或是逆時針順序為  $(P_0, P_1, \dots, P_{n-1})$ ，任選一個點  $P$ ，則這個簡單多邊形的面積為 (多項式是環狀的  $P_n = P_0$ )：

$$\frac{1}{2} \sum_{i=0}^{n-1} \overrightarrow{PP_i} \wedge \overrightarrow{PP_{i+1}}$$

以凹四邊形作為例子，用一般的方法計算面積的話可能將其拆成兩個三角形相加或是用減的，可能會花費額外的判斷，利用上面公式不論  $P$  選擇哪個點，凹陷處的部分都會因為正負號抵銷而變成 0，即使  $P$  使用外部的點也有一樣的效果，因此大部分的時候都會另  $P = \vec{0}$ ，如圖 12.5 所示。

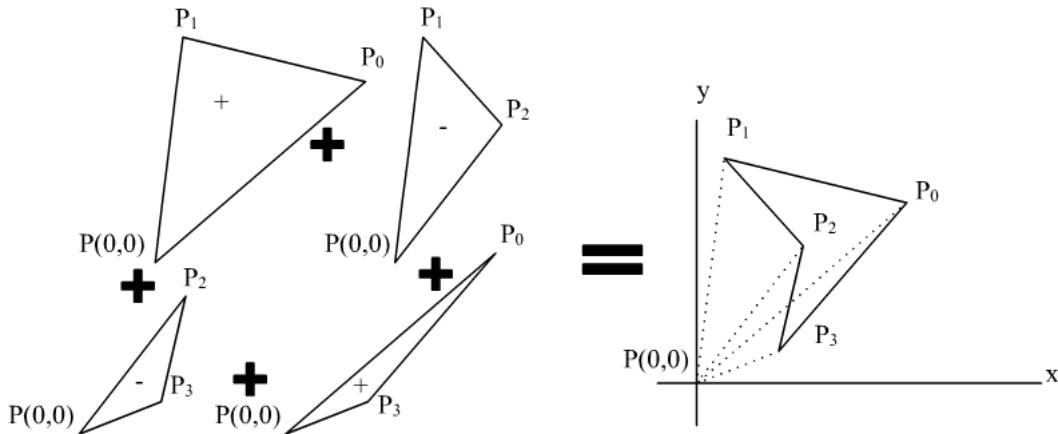


圖 12.5：右邊的凹四邊形面積可以由左邊四塊三角型面積加起來構成

$$\frac{1}{2} (\overrightarrow{P_0} \wedge \overrightarrow{P_1} + \overrightarrow{P_1} \wedge \overrightarrow{P_2} + \cdots + \overrightarrow{P_{n-1}} \wedge \overrightarrow{P_n})$$

注意我們可以讓函數吐出的是兩倍面積，因為我們不希望僅僅為了可能多的  $\frac{1}{2}$  分數把他轉型成浮點數，尤其像 CF 這種會專門構造測資讓浮點數答案爛掉的比賽。有向面積的正負判斷之後會經常遇到，我們可以看  $\overrightarrow{AB} \wedge \overrightarrow{AC}$  的正負來判斷  $\triangle ABC$  是不是逆時針的，正的話就是逆時針，負的話就是順時針，在此定義函數 ori (orientation)。

$$\text{ori}(a, b, c) = (b - a) \wedge (c - a)$$

### 12.2.2 判斷線段相交、找出直線交點

有了判斷有向面積正負的函數 ori，線段  $\overline{P_1P_2}, \overline{P_3P_4}$  的嚴格相交判定方式就可以變得很簡單：

|  |
|--|
| 如果 $\text{ori}(P_1, P_2, P_3) \times \text{ori}(P_1, P_2, P_4) < 0$ 且 $\text{ori}(P_3, P_4, P_1) \times \text{ori}(P_3, P_4, P_2) < 0$ ，則表示線段相交；反之條件不成立則是不相交 |
|--|

$\text{ori}(P_1, P_2, P_3) \times \text{ori}(P_1, P_2, P_4) < 0$  保證了  $P_3, P_4$  分別在直線  $\overline{P_1P_2}$  的兩側，同樣  $\text{ori}(P_3, P_4, P_1) \times \text{ori}(P_3, P_4, P_2) < 0$  會保證  $P_1, P_2$  分別在直線  $\overline{P_3P_4}$  的兩側，兩個條件同時成立時就能保證兩條線段會相交。

但是這只能用來判斷交點只有一個且不在任何一條線段的端點上的情況，如圖 12.6，如果是兩條線重疊、或是交點剛好在其中一個線段的端點上都會被判斷成不相交。想要包含端點的話，可以將上面兩個判斷改成小於等於 0 的時候視為相交，但是這樣又會引發另一個問題：共線時即使兩個線段沒有重疊也會被判斷成相交，因此需要將共線額外拿出來討論。

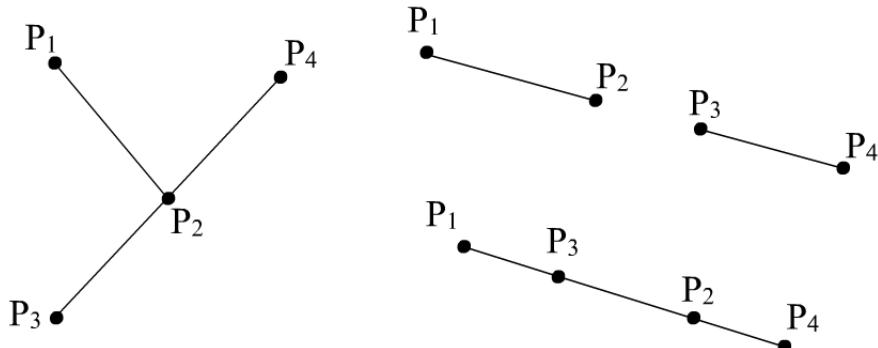


圖 12.6：類似這些情況要分別討論

共線時檢查一個線段的端點是否在另一條線段上。要檢查  $P_1$  是否在  $P_3, P_4$  上的方法是：檢查  $(P_3 - P_1) \cdot (P_4 - P_1) \leq 0$ ，若該式子成立則表示  $P_1$  往  $P_3, P_4$  的方向相反的，其他端點也可以用同樣的方法判斷。

有時候我們還是會無可避免地碰到計算交點（建議盡量能不用就不用），交點的座標可能會出現浮點數，計算方法可以用簡單代數方法推出，見圖 12.7：我們假設  $\vec{A} + i\vec{AB}$  會落在  $\overline{CD}$  之間，這時候可以推出：

$$\begin{aligned} (\vec{A} + i\vec{AB} - \vec{C}) \wedge \vec{CD} &= 0 \\ \vec{CA} \wedge \vec{CD} + i\vec{AB} \wedge \vec{CD} &= 0 \end{aligned}$$

解出

$$i = \frac{\vec{CA} \wedge \vec{CD}}{\vec{CD} \wedge \vec{AB}}$$

如果要計算一個點對直線的投影，我們考慮一個點  $\vec{P}$  與他對直線  $\overline{AB}$  的投影點  $\vec{A} + t_0\vec{AB}$ ，因為他們垂直，所以：

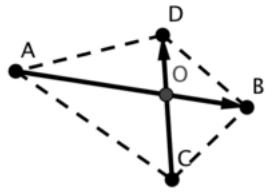


圖 12.7：用代數方程式計算交點

$$(\vec{P} - (\vec{A} + t_0 \vec{AB})) \cdot \vec{AB} = 0$$

$$\vec{AP} \cdot \vec{AB} - t_0 (\vec{AB} \cdot \vec{AB}) = 0$$

解出

$$t_0 = \frac{\vec{AP} \cdot \vec{AB}}{\vec{AB} \cdot \vec{AB}}$$

要算到直線的最短距離，可以把投影點算出來再算差距，如果要改成線段的話，因為參數  $t \in [0, 1]$ ，我們只要對 1 取 min，對 0 取 max 就好了。

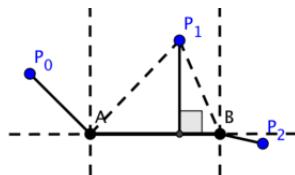


圖 12.8：距離線段最接近的點，分成三類區域

最後把我們的這些結論寫成線段類別：

```
1 template <typename T, typename Real = double>
2 struct Line {
3     Point<T> st, ed;
4     Point<T> vec() const { return ed - st; }
5     T ori(const Point<T> p) const { return (ed - st)^(p - st); }
6     Line(const Point<T> x, const Point<T> y) : st(x), ed(y) {}
7     template<class F> operator Line<F> () const {
8         return Line<F>((Point<F>)st, (Point<F>)ed);
9     }
10
11    // sort by arg, the left is smaller for parallel lines
12    bool operator<(Line B) const {
13        Point<T> a = vec(), b = B.vec();
14        auto sgn = [] (const Point<T> t) { return (t.y == 0? t.x: t.y) < 0; };
15        if (sgn(a) != sgn(b)) return sgn(a) < sgn(b);
16        if (abs(a^b) == 0) return B.ori(st) > 0;
17        return (a^b) > 0;
18    }
19
20    // Regard a line as a function
21    template<typename F> Point<F> operator()(const F x) const {
22        return Point<F>(st) + vec() * x;
23    }
24
25    bool isSegProperIntersection(const Line l) const {
26        return l.ori(st) * l.ori(ed) < 0 and ori(l.st) * ori(l.ed) < 0;
27    }
28
29    bool isPointOnSegProperly(const Point<T> p) const {
30        return ori(p) == 0 and ((st - p)&(ed - p)) < 0;
31    }
32
33    Point<Real> getIntersection(const Line<Real> l) {
34        Line<Real> h = *this;
35        return l((l.st^h.vec()) / (h.vec()^l.vec()));
36    }
37
38    Point<Real> projection(const Point<T> p) const {
39        return operator()(((p - st)&vec()) / (Real)(vec().norm()));
40    }
41};
```

程式碼 12.3: 線段類別實作

## 12.3 凸包演算法

### 12.3.1 凸包基礎

我們之前12.2提到過多邊形的有向面積算法：

```
1 template<typename T>
2 T twiceArea(vector<Point<T>> Ps) {
3     int n = Ps.size();
4     T ans = 0;
5     for (int i = 0; i < n; ++i)
6         ans += Ps[i] ^ Ps[i + 1 == n ? 0 : i + 1];
7     return ans;
8 }
```

程式碼 12.4：多邊形的兩倍有向面積算法

這一章節我們會著重在多邊形與凸包，為了方便起見，這裡不妨假設所有有像面積都是正的（如果是凸包就是逆時針轉回來）。我們會以 `vector<Pt<T>>` 實作多邊形與凸包。

首先我們來看一道題目：

Beauty Contest

POJ 2187

二維平面上有  $N$  個點，第  $i$  個點的座標是  $(x_i, y_i)$ ，同一個座標上不會有多個點。  
請找出這些點中最遠的兩個點的距離，並輸出此距離的平方。

限制： $2 \leq N \leq 50000, -10000 \leq x_i, y_i \leq 10000$

顯然若對每一個點對做檢查，時間複雜度是  $\mathcal{O}(N^2)$  會 TLE，所以需要省去沒必要的檢查。若某個點位於另外其他三個點構成的三角形內，這個點就絕對不是彼此最遠的兩個點之一。透過這樣的想法，沒有位於「**任意三個點構成的三角形**」內的點，也就是最外側的點，才需要枚舉點對判斷。這些點構成的集合，就是包含原始點集合的最小凸多邊形的頂點，這個凸多邊形就是原始點集合的**凸包**。可以想像凸包的邊是一個剛好包圍所有點的橡皮圈，如圖 12.9。

### 12.3.2 性質

凸包有很多好用的性質可以讓解題的時間變得更快速，在此介紹幾個常用的性質：

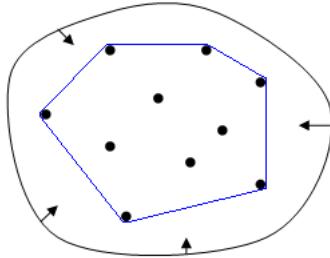


圖 12.9: 想像凸包的邊是一個剛好包圍所有點的橡皮圈 (圖片來自於維基百科)

**凸集合性質**: 兩凸包的交集還是凸包，用直線切割凸包還會是凸包，而且直線最多只會和凸包有兩個交點 (邊重合不算的話)，凸包中任兩點構成的線段一定會在凸包內。

不少題目求的極大極小值只會跟凸包的頂點或邊有關，內部的點可以不考慮，例如本題最遠點對問題、找出面積或周長最小且包含所有點的矩形等。

**整數點數限制**: 如果凸包上的所有點的座標範圍都限制在  $[0, W]$  的整數，那凸包上最多只會有  $\mathcal{O}(\sqrt[3]{W^2})$  個點，原因是凸包上相鄰邊不會共線。這個條件再  $W$  不大的時候很有用，可以使  $\mathcal{O}(N^2)$  枚舉點對的演算法變成  $\mathcal{O}(\sqrt{W^2}) = \mathcal{O}(W)$ 。像是本題的  $W = 20000$ ，先求出凸包後在枚舉凸包上的點對找距離最大的就可以 AC 此問題。

**邊斜率單調**: 凸包上每條邊的斜率 (角度) 是依序增加的，需要在邊上枚舉時可以用二分搜降低複雜度，例如計算一個點對凸包的切線可以做到  $\mathcal{O}(\log N)$ 。另外該性質可以推廣出等一下會介紹的**旋轉卡尺**的技巧。

### 12.3.3 找出凸包

目前已知的求凸包演算法有很多種，這邊介紹較為容易實作、複雜度為  $\mathcal{O}(N \log N)$  的 Andrew's monotone chain 演算法。

首先將所有點以  $x$  座標由小到大進行排序，若兩點  $x$  座標相同則根據  $y$  座標進行比較。排序後第一個和最後一個點一定在凸包的邊上，兩點之間的部分要把上側和下側分開來處理。建立上下側的方法類似，這裡以建立下側進行說明。建立下側時，將排序好的點由小到大依序看過去，逐漸建立凸包。建到一半的凸包，再尾端加上新的點後，可能就不滿足凸包的性質，此時要把凹陷的部分拿掉，凹陷部分的點可以用 cross 進行判斷。上側也是用相同的方式處理。

```

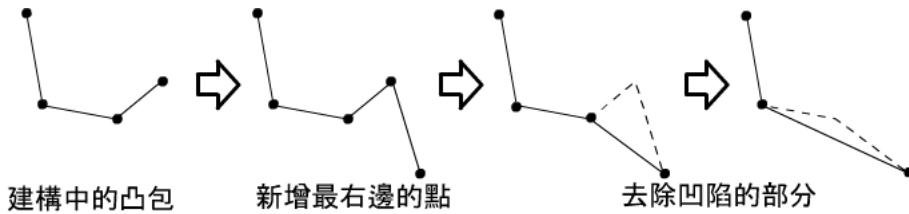
1 template <class F> using Polygon = vector<Point<F>>;
2
3 template <class F>

```

```

4  Polygon<F> getConvexHull(Polygon<F> points) {
5      sort(begin(points), end(points));
6      Polygon<F> hull;
7      hull.reserve(points.size() + 1);
8      for (int phase = 0; phase < 2; ++phase) {
9          auto start = hull.size();
10         for (auto& point : points) {
11             while (hull.size() >= start + 2 and
12                 Line<F>(hull.back(), hull[hull.size() - 2]).ori(point) <= 0)
13                 hull.pop_back();
14             hull.push_back(point);
15         }
16         hull.pop_back();
17         reverse(begin(points), end(points));
18     }
19     if (hull.size() == 2 and hull[0] == hull[1]) hull.pop_back();
20     return hull;
21 }
```

**程式碼 12.5:** Andrew's monotone chain 演算法全過程



**圖 12.10:** 去除造成凹陷部分的點

#### 12.3.4 包含測試

接下來我們要來看一個點能不能被放在多邊形裡面，我們分成三角形跟凸多邊形，最後再講任意多邊型：

三角形

我們先把結果分類起來，測試結果總共有三種可能：

- 點嚴格在三角形內部
- 點在三角形邊界上
- 點在三角形外部

我們先來定義 sign 函數：

$$\text{sign}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

接著做些觀察，可以發現不用管三角形是轉順逆時針的，我們如果觀察目標點在每條邊的方向，把 ori 求出在看他們的號，會有下面現象：

- 點在內部則全部同號
- 點在外部則一定存在兩條邊是異號
- 其他情況則是在邊界上

```
1 const int INSIDE = -1;
2 const int BOUNDARY = 0;
3 const int OUTSIDE = 1;
4 inline int prev(int i, int n) { return i == 0 ? n-1 : i-1; }
5 inline int next(int i, int n) { return i == n-1 ? 0 : i+1; }
6 template <class T> inline int sgn(const T& x) {
7     return (T(0) < x) - (x < T(0));
8 }
9
10 template <class F>
11 int pointVsTriangle(const Point<F> point, const Polygon<F> triangle) {
12     assert(triangle.size() == 3);
13     int signs[3];
14     for (int i = 0; i < 3; ++i)
15         signs[i] = sgn(Line<F>(triangle[next(i, 3)],
16                             triangle[i]).ori(point));
17     if (signs[0] == signs[1] and signs[1] == signs[2])
18         return INSIDE;
19     for (int i = 0; i < 3; ++i)
20         if (signs[i] * signs[next(i, 3)] == -1)
21             return OUTSIDE;
22     return BOUNDARY;
23 }
```

程式碼 12.6：測試點有沒有在三角形裡面

## 凸包

我們點類是按照字典序排序的，如果凸包是逆時針（正面積），不妨假設最小的點在最前面（Andrew's monotone chain 有保證這件事），用大小看的話，一個凸包會先遞增，然後遞減，所以如果抓出最大的點，這個凸包的一些性質基本上是可以掌握的，可以用三分搜或是用 `max_element` 預處理找到他，這裡叫他 `top`。

我們可以先看目標點是不是比凸包上最小點的小或是比凸包上最大的點大，如果有的話很明顯在外面。接著，畫一條線在最小的點跟最大的點之間，首先可以分辨目標點是在這條線上面還是下面，如果在這條線上只要判斷是不是端點就好了，要特判這條線是邊界的情況。我們假設目標點在下面（上面也是一樣做法），那我們就切開來，用二分搜抓出他的大小在下凸包的哪兩點之間，檢驗這兩點就好了。複雜度  $\mathcal{O}(\lg n)$ 。

```
1 template <class F>
2 int pointVsConvexPolygon(const Point<F>& point,
3                           const Polygon<F>& poly, int top) {
4     if (point < poly[0] or poly[top] < point) return OUTSIDE;
5     auto orientation = Line<F>(poly[0], poly[top]).ori(point);
6     if (orientation == 0) {
7         if (point == poly[0] or point == poly[top]) return BOUNDARY;
8         else return top == 1 or top + 1 == poly.size() ? BOUNDARY : INSIDE;
9     } else if (orientation < 0) {
10        auto itRight = lower_bound(begin(poly) + 1,
11                                begin(poly) + top, point);
12        return sgn(Line<F>(itRight[0], itRight[-1]).ori(point));
13    } else {
14        auto itLeft = upper_bound(poly.rbegin(), poly.rend() - top - 1, point);
15        return sgn(Line<F>(itLeft == poly.rbegin() ? poly[0] : itLeft[-1],
16                            itLeft[0]).ori(point));
17    }
18 }
```

程式碼 12.7：測試點有沒有在凸包裡面

## 多邊形

如果要判定一個點是否在一個（有可能有凹的部分）的多邊形裡面，有個方法是隨機打一條射線，然後祈禱他不會打到邊界點，看看他打到邊界是奇數次還是偶數次。不過我們既然都用整數點做幾何了，不用那麼委屈。

不妨假設目標點在原點，現在我們把平面切割成兩個半平面  $\mathbb{R} \times [0, \infty)$  和  $\mathbb{R} \times (-\infty, 0)$ 。我們定義目標點在平面的繞數（winding number）是這個點對於有向

面積的貢獻次數，等同在說有幾條邊穿過從他向右打出去切割出來的半平面，向上穿過則 +1，向下穿過則 -1，點在多邊形裡面若且唯若繞數 = 0。

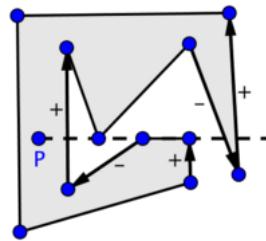


圖 12.11：直接向右打出射線

唯一要特判的是跟  $x$  軸方向一樣的邊，雖然他不會穿過半平面，但是目標點有可能在邊上。

```

1 template <class F>
2 int pointVsPolygon(const Point<F>& point, const Polygon<F>& poly) {
3     int n = static_cast<int>(poly.size()), windingNumber = 0;
4     for (int i = 0; i < n; ++i) {
5         if (point == poly[i]) return BOUNDARY;
6         int j = next(i, n);
7         if (poly[i].y == point.y and poly[j].y == point.y) {
8             if (min(poly[i].x, poly[j].x) <= point.x and
9                 point.x <= max(poly[i].x, poly[j].x)) return BOUNDARY;
10        } else {
11            bool below = poly[i].y < point.y;
12            if (below != (poly[j].y < point.y)) {
13                auto orientation = Line<F>(poly[i], poly[j]).ori(point);
14                if (orientation == 0) return 0;
15                if (below == (orientation > 0)) windingNumber += below ? 1 : -1;
16            }
17        }
18    }
19    return windingNumber == 0 ? OUTSIDE : INSIDE;
20 }
```

程式碼 12.8：計算繞數，同時也是 Timus 1599 的答案

`if (point == poly[i])` 這一行是必要的，不然把這一行拿掉可以構造會錯的測資：

```

1 Polygon<int> arb;
2 arb.emplace_back(0, -1);
3 arb.emplace_back(1, 0);
4 arb.emplace_back(0, 1);
5 arb.emplace_back(-1, 0);
6 assert(pointVsPolygon(Pt<int>(0, -1), arb) == BOUNDARY);
7 // Wrong! Returned OUTSIDE
```

---

**程式碼 12.9:** 測資構造

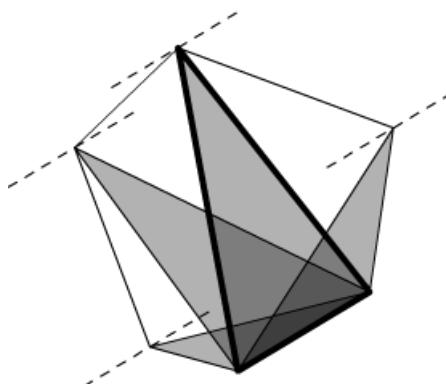
該算法複雜度為  $\mathcal{O}(n)$

### 12.3.5 旋轉卡尺

如果點的座標範圍  $W$  更大的話，構造凸包然後  $\mathcal{O}(W)$  枚舉所有點對的方法就會 TLE，這邊介紹旋轉卡尺（又稱旋轉卡殼, Rotating claper）的概念。

旋轉卡尺精神上是兩條平行線貼著凸包旋轉的線，顧名思義，就是通過一對平行線來卡住凸包上的兩個相對的點，然後通過旋轉這對平行線能夠得到凸包上距離最遠的一對點。

這樣直接去寫會發現不太容易，因此可以轉化一下，使這一對平行線在旋轉的過程中讓其中一條與凸包的一條邊重合，此時另一條線的頂點是距這條邊最遠的點。可以觀察到，當枚舉的邊逆時針旋轉時，最遠點也是跟著逆時針變化，這樣我們可以不用每次枚舉所有的頂點，直接從上次的最遠點開始繼續計算即可，此時複雜度為  $\mathcal{O}(n)$ 。



**圖 12.12:** 離直線最遠的點和該直線所在的邊構成的三角形面積會最大

計算點到直線的距離時沒必要直接計算，如圖 12.12，因為三角形面積示底乘高除以二，因此最遠的頂點和這條線組成的三角形面積一定是最大的，所以可以利用計算三角形面積求出離直線最遠的點。

本題解法如圖 12.13 所示，利用旋轉卡尺在過程中枚舉每個點的對踵點，其中距離最大的就是最遠點對。值得注意的是，並不是每個點的對踵點都是離該點最遠的點，練習題 UVA 12311 就是一個很好的例子。

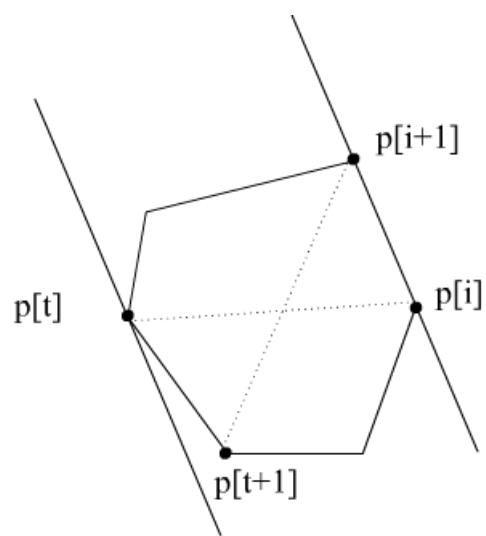


圖 12.13: 旋轉卡尺

```

1 template<typename F> //return square of maximum distance
2 F rotatingClaiper(vector<Point<F>> p){
3     int n = p.size();
4     F ans = 0;
5     p.push_back(p[0]);
6     for (int i = 0, j = 1; i < n; ++i) {
7         Line<F> L(p[i], p[i + 1]);
8         while (L.cross(p[j + 1]) >= L.cross(p[j])) j = next(j, n);
9         ans = max({ans, (p[i] - p[j]).norm(), (p[i + 1] - p[j]).norm()});
10    }
11    return ans;
12 }

```

**程式碼 12.10:** Rotating claper

### 12.3.6 練習題

Jack Straws

POJ 1127

給你  $n$  條線段，然後你需要回答接下來的數個詢問：第  $i$  條線段和第  $j$  條線段是否相交？( $i$  與  $j$  通過別的線段間接相交也算他們兩相交，且  $i$  線段與  $j$  線段交於端點也算相交)。

Cows

POJ 3348

給你  $n$  個點，請用這些點求出凸包後，計算凸包面積。

最小矩形覆蓋

bzoj 1185

給你  $n$  個點，請找出一個面積最寫且可以覆蓋所有點的矩形。

All-Pair Farthest Points(高難度題目)

UVA 12311

求一個凸多邊形上與每個點距離最遠的點，點的數量有 30000 個。

## 12.4 折線技巧

### 12.4.1 基礎問題

Sonya and Problem Wihtout a Legend

CF 713C

給定一個長度為  $n$  的陣列，可以做的操作有：

選定一個元素，將它的大小改變 +1 或是 -1

詢問最少步驟將該陣列變成遞增

題目預設複雜度為  $\mathcal{O}(n^2)$ ，但其實可以做到  $\mathcal{O}(n \lg n)$

### 12.4.2 折線結構

就跟許多的 dp/greedy 問題一樣，我們會維護某種性質，然後從左掃到右計算新元素對於原本維護的性質的改變，新進來的元素有可能會成為數字最大的元素，也有可能需要往下調，所以我們考慮維護以下函數：

令  $f(x)$  代表把所有數字壓到不大於  $x$  所需最小花費，顯然當數字  $M$  夠大， $f(M)$  就是答案， $f$  會根據新進來的元素而有所改變。

可以觀察  $f$  有以下性質，假設目前看到  $n$  個元素：

1. 最多可以切割成  $n + 1$  個斜率區段，最左側無界部分斜率一定是  $-n$ ，最右邊無界部分斜率一定是 0。
2. 初始條件  $f = 0$ 。
3. 假設新進的元素為  $a$ ，更新  $f'(x) = \min_{y \leq x} \{f(y) + |a - y|\}$ 。

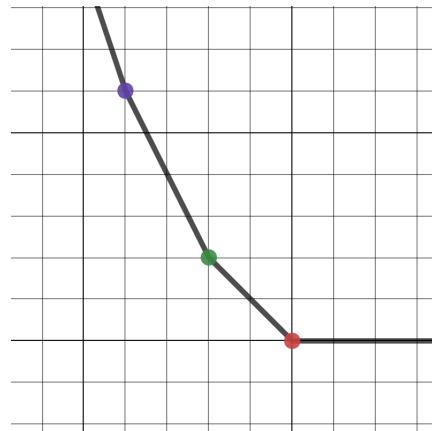


圖 12.14：折線範例

### 12.4.3 折線更新

所以我們現在考慮折線  $f'$  是如何更新的，在看到了元素  $a$  以及目前折線  $f$ 。

可以改寫函數的更新：

$$f'(x) = \min_{y \leq x} \{f(y) + |a - y|\}$$

分成兩個 Case 討論，假定  $f$  最右邊斜率轉折的點  $x$  座標為  $p$ ，如果  $p \leq a$  那情況非常簡單：

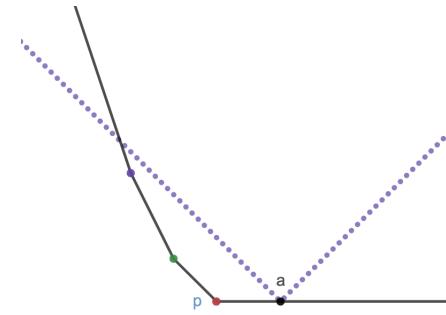


圖 12.15: Case 1

$$f'(x) = \begin{cases} f(x) + a - x & \text{if } x \leq a, (y = x) \\ f(a) & \text{if } x > a, (y = a) \end{cases}$$

$f'$  相較  $f$  在  $a$  以左斜率增加  $-1$ 。

第二種 Case，如果  $a < p$ ，我們一樣可以分三線段：

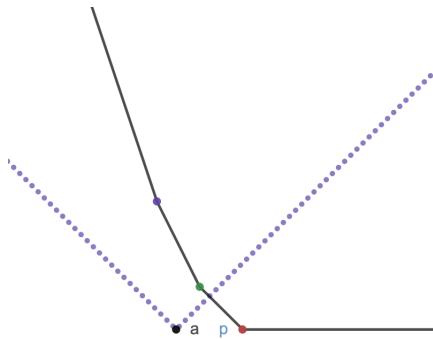


圖 12.16: Case 2

$$f'(x) = \begin{cases} f(x) + a - x & \text{if } x \leq a, (y = x) \\ f(x) + x - a & \text{if } a < x \leq p, (y = x) \\ f(p) + p - a & \text{if } x > p, (y = p) \end{cases}$$

相較  $f$ 、 $f'$  在  $a$  以左一樣斜率增加  $-1$ ，但是  $a < x \leq p$  斜率減  $-1$  i.e.  $+1$ ，注意到  $p$  不再是最右邊斜率轉折點的座標，取而代之的是倒數第二個轉折點。

#### 12.4.4 折線實作

我們用 priority queue 儲存這些轉折點，注意到用的是 priority queue 而不是 set 是因為可能會有重複的轉折點，也就是在同一個位置斜率增加  $> 1$ ，同時紀錄  $f(p)$  代表目前的水平線幅度，因此我們的資料結構就只有簡單的一個 priority queue 跟一個數字  $fp$ 。

Case 1：priority queue 推入一個元素  $a$ 。

Case 2: 三個步驟，沒有優先順序

1. priority queue 推入一個元素  $a$
2. pop 最大的轉折點  $p$ ，在  $a$  處 push 回來
3.  $fp$  更新為  $fp + p - a$

### 12.5 凸包優化技巧

#### 12.5.1 基礎問題

凸包優化技巧通常可以抽象成這個問題：

| Lena and Queries   | CF 678F 簡化 |
|--|------------|
| 一開始有一個空集合 $S$ 有以下兩種操作要進行，操作共 $N$ 個： <ul style="list-style-type: none"><li>• 把一條整數線性函數 <math>f_i(x) = a_i x + b_i</math> 加進這個集合</li><li>• 給一個整數 <math>x_j</math> 求 <math>\max_i f_i(x_j)</math></li></ul> |            |

這一類的問題反而比較常在動態規劃出現，我們今天主要討論這問題。

$\mathcal{O}(N^2)$  的操作很好想到，只要用 `vector` 實作就好了，現在我們要考慮快一點的作法。

## 12.5.2 凸包結構

我們把一堆直線丟進二維平面，如果我們對於每一個點只在乎最上方的那個函數點，那麼整體來看就會是一個下凸包，可以用一些繪圖軟體來畫看看：

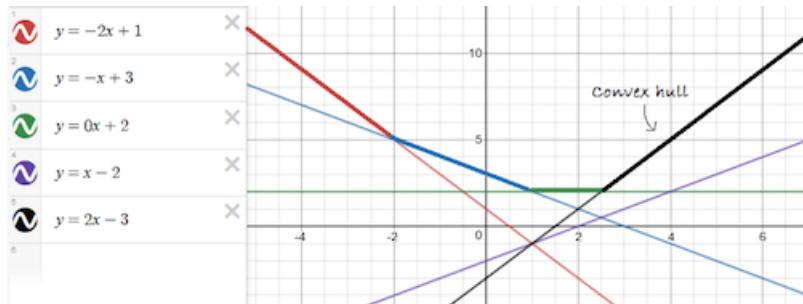


圖 12.17：由一些直線構成下凸包

首先，今天假設所有詢問都是靜態的，也就是可以預處理的情況，那麼我們可以把凸包構造好，因為只保留凸包得到的答案跟原本的答案是一樣的。接著我們可以三分搜答案，讓我們來看看構造凸包跟三分搜的實作細節。

## 12.5.3 構造凸包

因為如果有兩個直線函數有一樣的斜率（一樣的  $a$ ），那麼我們永遠可以選擇常數  $b$  較大的函數來構造凸包，比較小的那個永遠用不到，所以不妨假設所有直線斜率  $a$  相異。我們要把直線按照斜率  $a$  排序好，那麼因為凸包是這些斜率的子序列，可以透過從左到右掃一次將這些直線函數抽出來當作凸包。

先讓我們先來寫個直線函數類別，這樣我們就能把線當成是真的函數，也不用花太多時間在基礎操作上面。

```
1 template<typename F = long long>
2 struct Linear { // y = a * x + b;
3     F a, b;
4     Linear(F a = 0, F b = 0) : a(a), b(b) {}
5     bool operator<(Linear oth) const {
6         return a == oth.a ? b < oth.b : a < oth.a;
7     }
8     template<typename T> T operator()(T x) const { return a * x + b; }
9 };
```

程式碼 12.11：直線類別

在此強調他是線性函數，所以用 `Linear` 命名。談論我們要排常數  $b$  的原因，因為實作上我們也不用真的去把直線函數抽出來也能解決（看個人偏好），這樣實作的話能保證目前掃到的直線函數會在目前維護的凸包裡面。

那我們要把凸包這個子序列抽出來，實作上用 `vector` 就能做了。

每當有新的直線函數進來的時候，會有一些舊的（斜率較低或一樣但是常數較小）直線函數失效，所謂失效，就是指他很明顯不會待在凸包上面，比如說  $f(x) = -x + 1, g(x) = 0, h(x) = x + 1$ ，那麼  $g(x)$  當  $h(x)$  進來的時候就失效了。

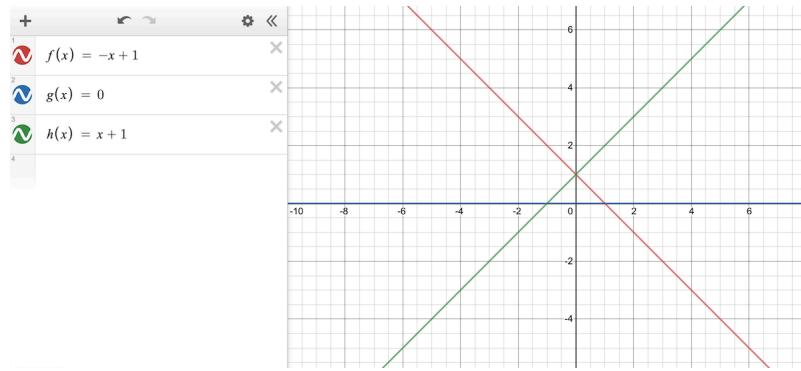


圖 12.18:  $g$  在  $h$  進來的時候失效

現在切換到數學角度來看待函數失效，假設在凸包上的兩個最右端的函數  $f(x) = a_1x + b_1, g(x) = a_2x + b_2$  而且  $a_1 < a_2$ ，可以導一下看什麼時候  $g(x) \geq f(x)$

$$\begin{aligned} g(x) &\geq f(x) \\ a_2x + b_2 &\geq a_1x + b_1 \\ (a_2 - a_1)x &\geq b_1 - b_2 \\ x &\geq \frac{b_1 - b_2}{a_2 - a_1} \end{aligned}$$

我們令  $x^* = \frac{b_1 - b_2}{a_2 - a_1}$ ，代表  $x^*$  右邊以後函數  $g$  代進去比較大，假設詢問只有整數點的話可以令  $x^* = \lceil \frac{b_1 - b_2}{a_2 - a_1} \rceil$  來避免浮點數誤差。

假設現在新進來一個函數  $h(x) = a_3x + b_3$ ，如果  $a_3 = a_2$  那  $g$  按照排序的方法一定會失效，如果  $a_2 < a_3$  而且  $f, g$  決策點在  $g, h$  右邊，那麼就代表  $g$  不會在任何一個地方成為最大值（如果用  $x^*$  帶入可能會導致 overflow），因此可以把  $g$  移出我們的凸包，我們會一直 pop 掉像  $g$  的直線函數直到這樣的條件不滿足或是只剩一條直線函數。

以上是我們解釋凸包的建構方式，接著我們會提到如何善用我們由直線函數構成的凸包。

## 12.5.4 尋找最大值

當有了由直線函數構成的凸包，我們假設凸包裡面的直線函數是  $f_0, f_1, \dots, f_{n-1}$ （當然，斜率遞增），對於一個固定值  $x$ ， $f_0(x), f_1(x), \dots, f_{n-1}(x)$  會先遞增再遞減，可以用三分搜找最大值，使用三分搜可以把當成是對斜率二分搜，看是什麼時候斜率會由非正轉正。

我們令  $df_i(x) = f_{i+1}(x) - f_i(x), i \in [0, n-1]$ ，我們可以知道  $df_i$  會遞減，二分搜第一個  $i$  使得  $df_i(x) < 0$  那麼答案就是  $f_i(x)$ ，要特判  $n=1$  或是  $df_0(x) \leq 0$  的情況，此時答案是  $f_0(x)$

```
1 template<typename F = long long>
2 class ConvexHull { // lower envelop
3     vector<Linear<F>> f;
4     const int QUERY_RANGE = INT_MAX;
5     bool cross(Linear<F> P, Linear<F> Q, Linear<F> T) const {
6         if (Q.a == T.a) return Q.b <= T.b;
7         assert(P.a < Q.a);
8         auto ceil2 = [=](F a, F b) { // return ceil(a/b) for b > 0
9             return a >= 0 ? (a + b - 1) / b : a / b;
10        };
11        const F x = ceil2(P.b - Q.b, Q.a - P.a);
12        return QUERY_RANGE < abs(x) ? Q.b <= T.b : Q(x) <= T(x);
13    }
14    void getHull() {
15        int m = 0;
16        for (int i = 0; i < f.size(); ++i) {
17            while (m >= 2 and cross(f[m - 2], f[m - 1], f[i])) --m;
18            if (m == 1 and f[0].a == f[i].a) --m;
19            f[m++] = f[i];
20        }
21        f.resize(m);
22    }
23 public:
24     int size() const { return f.size(); }
25     ConvexHull(vector<Linear<F>> A) : f(A) {
26         sort(f.begin(), f.end());
27         getHull();
28     }
29     F operator()(F x) const {
30         assert(abs(x) <= QUERY_RANGE);
31         int n = f.size();
32         auto df = [&](int i) { return f[i + 1](x) - f[i](x); };
33         if (n == 0) return numeric_limits<F>::min();
34         if (n == 1 or df(0) <= 0) return f[0](x);
35         int l = 0, r = n - 1;
36         while (r - l > 1) {
37             int m = l + r >> 1;
38             (df(m) < 0 ? r : l) = m;
39         }
40     }
```

```

40     return f[r](x);
41 }
42 };

```

程式碼 12.12: 凸包建構與三分搜

### 12.5.5 融合凸包

兩個凸包  $P, Q$  融合要花的時間是  $\mathcal{O}(|P| + |Q|)$ ，實作只要簡單地使用 STL 的 merge 函數就好了

```

1 class ConvexHull{
2 /*
3 basic functions
4 */
5 void mergeHull(const ConvexHull &Q) {
6     vector<Linear<F>> ls;
7     merge(begin(Q.f), end(Q.f), begin(f), end(f), back_inserter(ls));
8     f = ls;
9     getHull();
10 }
11 };

```

程式碼 12.13: 兩個凸包融合

### 12.5.6 可拓展集合 (Expandable set)

剛剛的解法中，我們的凸包是靜態的，那我們要怎麼把凸包變成可以動態插入點呢？

這個時候可以使用 Expandable set 的技巧，我們開一個 `vector<ConvexHull<F>>`，我們每次加入直線函數的方式是加入一個大小為一的凸包（這個凸包一點都不凸，看起來比較像是歪掉的地平線）這裡是拿來當 stack 使用，那我們維護凸包的方式是，**每當倒數第二個凸包大小比最後一個小，那就把最後兩個凸包融合起來**，令一個元素的位勢函數為自己在 stack 上的高度，平均下來每個元素的複雜度為  $\mathcal{O}(\lg N)$ 。

```

1 template<typename F>
2 struct ExpandableHull {
3     vector<ConvexHull<F>> hull;
4     void add(F a, F b) {
5         hull.push_back(vector<Linear<F>>{Linear<F>(a, b)});
6         int m = hull.size();
7         while (m >= 2 and hull[m - 1].size() >= hull[m - 2].size()) {
8             hull[m - 2].mergeHull(hull[m - 1]);
9             hull.pop_back();

```

```

10     --m;
11 }
12 }
13 F operator()(F x) { // get maximum
14     F ans = numeric_limits<F>::min();
15     for (auto& h : hull) ans = max(ans, h(x));
16     return ans;
17 }
18 };

```

**程式碼 12.14:** 從靜態凸包變成可拓展的靜態凸包集合

### 12.5.7 移除直線操作（離線）

我們來看原本的題目

| Lena and Queries | CF 678F |
|------------------|---------|
|------------------|---------|

一開始有一個空集合  $S$  有以下兩種操作要進行，操作共  $N$  個：

- 把一條整數線性函數  $f_i(x) = a_i x + b_i$  加進這個集合
- 把一條整數線性函數  $f_i(x) = a_i x + b_i$  移出這個集合
- 給一個整數  $x_j$  求  $\max_i f_i(x_j)$

多了移除操作之後看起來棘手許多，畢竟加入刪除無法達成均攤。改成在線操作，我們把問題移到離線處理，作法是：

開一條時間線段樹，讓每個節點代表一段時間（一個操作當一單位時間），紀錄每一條直線函數在哪一個時間  $[l, r)$  會存在在二維平面上，把那些直線放進去線段樹相對應節點，然後把詢問塞進時間線段樹的葉子，回答詢問時，只要去詢問一個葉子到根部所有節點中的靜態凸包就好了。

```

1 class Seg {
2     int l, m, r;
3     vector<Linear<long long>> ls;
4     int q = INT_MAX;
5     Seg* ch[2];
6 public:
7     Seg(int l, int r) : l(l), r(r), m(l + r >> 1) {
8         if (r - l > 1) {
9             ch[0] = new Seg(l, m);
10            ch[1] = new Seg(m, r);

```

```

11     }
12 }
13 void add(int ql, int qr, Linear<long long> L) {
14     if (ql <= l and r <= qr) ls.push_back(L);
15     else {
16         if (ql < m) ch[0]->add(ql, qr, L);
17         if (m < qr) ch[1]->add(ql, qr, L);
18     }
19 }
20 void setQuery(int p, int x) {
21     if (r - l == 1) q = x;
22     else if (p < m) ch[0]->setQuery(p, x);
23     else ch[1]->setQuery(p, x);
24 }
25 void compile(vector<long long>& ret, vector<ConvexHull<>> &hulls) {
26     hulls.emplace_back(ls);
27     if (r - l == 1) {
28         if (q != INT_MAX) {
29             long long ans = LLONG_MIN;
30             for (auto &h: hulls) ans = max(ans, h(q));
31             ret.push_back(ans);
32         }
33     }
34     else {
35         ch[0]->compile(ret, hulls);
36         ch[1]->compile(ret, hulls);
37     }
38     hulls.pop_back();
39 }
40 };

```

程式碼 12.15: 時間線段樹離線解決問題

### 12.5.8 set 維護凸包

另一種實作的方法是用 set 去實作凸包的插入，記得一個凸包的斜率是遞增的，所以我們可以用斜率去當凸包的 key，在插入的時候看看自己是否退化，如果沒有，往前後觀察附近的函數是否退化，有的話就刪除。

```

1 using F = long long;
2 struct Line {
3     static const F QUERY = numeric_limits<F>::max();
4     F m, b;
5     Line(F m, F b) : m(m), b(b) {}
6     mutable function<const Line*()> succ;
7     bool operator<(const Line& rhs) const {
8         if (rhs.b != QUERY) return m == rhs.m ? b < rhs.b : m < rhs.m;
9         const Line* s = succ();
10        return s and b - s->b < (s->m - m) * rhs.m;
11    }

```

```

12     F operator()(F x) const { return m * x + b; };
13 }
14
15 struct HullDynamic : public multiset<Line> {
16     bool isOnHull(iterator y) { //Mathematically, Strictly
17         auto z = next(y);
18         if (y == begin()) return z == end() or y->m != z->m or z->b < y->b;
19         auto x = prev(y);
20         if (z == end()) return x->m != y->m or x->b < y->b;
21         if (y->m == z->m) return y->b > z->b;
22         if (x->m == y->m) return x->b < y->b;
23         return (x->b - y->b) * (z->m - y->m) < (y->b - z->b) * (y->m - x->m);
24         // Beware long long overflow
25     }
26     void insertLine(F m, F b) {
27         auto y = insert(Line(m, b));
28         y->succ = [=] { return next(y) == end() ? nullptr : &*next(y); };
29         if (not isOnHull(y)) { erase(y); return; }
30         while (next(y) != end() and not isOnHull(next(y))) erase(next(y));
31         while (y != begin() and not isOnHull(prev(y))) erase(prev(y));
32     }
33     F operator()(F x) { return (*lower_bound(Line{x, Line::QUERY}))(x); }
34 };

```

程式碼 12.16: set 凸包

### 12.5.9 凸包優化小結

凸包優化的題目通常伴隨著一些 dp，比較少像這樣裸題直接出來的，比如像是

$$dp[i] = \max_{j < i} \{ dp[j] \times a[i] + c_j \}$$

這一類的題目，表面上看起來像是 dp 實際上 dp 式寫下來還要解凸包優化，那就要一邊把 dp 算好一邊維護我們剛剛的凸包資料結構。

## 12.6 最近點問題

本節會探討兩個問題，一是任兩點間最近點對問題，二是針對一個靜態點集回應多個最近點詢問的資料結構問題。

在這裡，我們一樣討論二維平面，因為 Curse of Dimensionality 的關係，下面演算法到高維度會退化到暴力解，因此我們只討論二維平面。

## 12.6.1 最近點對問題

先來看一個經典的計算幾何問題：

|                                |               |
|--------------------------------|---------------|
| 最近點對                           | SPOJ CLOPPAIR |
| 給定平面上 $N$ 個點，問任意兩點對之間的歐幾里德最短距離 |               |

一個細節是通常題目會要求輸出距離的平方避免使用浮點數。

大多數的做法是 Divide and Conquer，我們把平面切成兩堆，這樣最近點對只有三種可能，不是完全在左右就是橫跨左右，大致步驟如下：

### 演算法 12.1 最近點對演算法

- 1: 先按照字典序先 X 再 Y 排序
- 2: Divide: 以中位數當 pivot，依 X 軸分成左右兩堆，recursively 求出各自的最近點對
- 3: Combine: 令  $\delta$  為左右兩邊最近點對的最小值，將穿過 pivot 的軸左右開出寬為  $\delta$  的帶子，利用雙指標窮舉橫跨左右兩端的最近點對

```
1 template<typename T>
2 T ClosestPairSquareDistance(typename vector<Point<T>>::iterator l,
3                               typename vector<Point<T>>::iterator r) {
4     auto delta = numeric_limits<T>::max();
5     if (r - l > 1) {
6         auto m = l + (r - l >> 1);
7         nth_element(l, m, r); // Lexicographical order in default
8         auto x = m->x;
9         delta = min(ClosestPairSquareDistance<T>(l, m),
10                  ClosestPairSquareDistance<T>(m, r));
11         auto square = [&](T y) { return y * y; };
12         auto sgn = [=](T a, T b) {
13             return square(a - b) <= delta ? 0 : a < b ? -1 : 1;
14         };
15         vector<Point<T>> x_near[2];
16         copy_if(l, m, back_inserter(x_near[0]), [=](Point<T> a) {
17             return sgn(a.x, x) == 0;
18         });
19         copy_if(m, r, back_inserter(x_near[1]), [=](Point<T> a) {
20             return sgn(a.x, x) == 0;
21         });
22         for (int i = 0, j = 0; i < x_near[0].size(); ++i) {
23             while (j < x_near[1].size() and
24                   sgn(x_near[1][j].y, x_near[0][i].y) == -1) ++j;
25             for (int k = j; k < x_near[1].size() and
26                  sgn(x_near[1][k].y, x_near[0][i].y) == 0; ++k) {
27                 delta = min(delta, (x_near[0][i] - x_near[1][k]).norm());
28             }
29         }
30         inplace_merge(l, m, r, [](Point<T> a, Point<T> b) {
31             return a.y < b.y;
32         });
33     }
34 }
```

```

32     });
33 }
34 return delta;
35 }

```

程式碼 12.17: 最近點對

主要困難的部分在 Combine，實作稍微複雜，但是可以做到  $\mathcal{O}(n)$  這邊會做說明。

我們把左右距離 pivot 軸不到  $\delta$  的點搜集起來，然後各自開兩陣列，按照 Y 軸排序，用雙指標技巧，對於每個左邊的點去找對應右邊的點有哪些 Y 軸差距在  $\delta$  以內（根據  $\delta$  的定義，差距  $\delta$  之外的點不可能成為最近點對），滿足這些條件的點最多 6 個（在一個  $\delta \times 2\delta$  的矩形內部）。

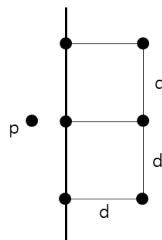


圖 12.19: 最多 6 個點可能成為最近點對

在 DFS 出來的時候可以做對 Y 軸的 Merge，這樣可以不用再對 Y 軸排序之後抽子序列，做到理論上的  $\mathcal{O}(n)$ 。

總體複雜度  $T(n) = 2T(n/2) + \mathcal{O}(n) = \mathcal{O}(n \lg n)$ 。

## 12.6.2 最近點詢問

這是另一個經典的計算幾何問題，我們會引述 k-d Tree 的概念：

|                                      |              |
|--------------------------------------|--------------|
| 最近點詢問                                | SPOJ FAILURE |
| 給定平面上 $N$ 個點，問每個點離他最近點的距離，答案輸出歐式距離平方 |              |

可以把這個問題簡化一下：每個點看成很多詢問，詢問這個點集除了該點以外最接近的點。顯然前面的最近點對問題可以用這個問題來解。

這裡我們引入 k-d tree 的概念，k-d tree 是一種用來處理多維的樹形資料結構，每個非葉節點會挑一個軸把點集分成兩區，然後在下一層挑另一個軸 recursively 造樹。

k-d tree 可以用來搜尋最近點詢問，在平均、點分布隨機狀況下運作  $\mathcal{O}(\lg n)$ ，最差狀況會到  $\mathcal{O}(\sqrt{n})$ ，分析較為困難，這裡不會提及。

進行最近點詢問的具體做法，假設我們稱詢問的點  $q$ ，要求目前在 k-d tree 裡面點集中最靠近  $q$  的距離，首先可以觀察到，k-d tree 的每一個點都代表一個軸（水平或垂直），根據這個軸我們可以決定往左樹還是右樹搜尋。

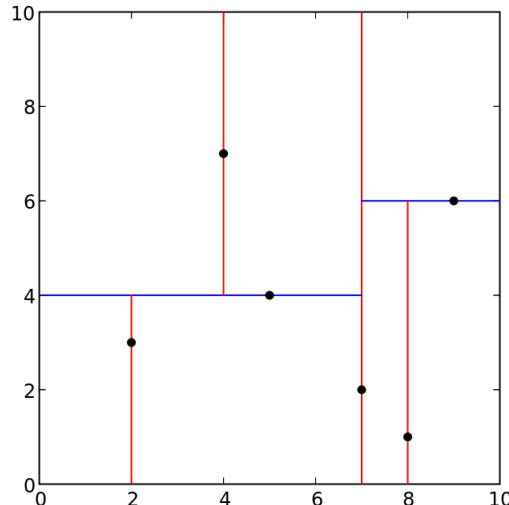


圖 12.20: k-d tree structure

也許讀者會問「可是最後找到的葉節點不一定是最近點」，就像點對問題一樣，我們會維護一個目前看到離  $q$  最短距離  $\delta$ 。當我們搜到最底端之後，從葉子上來可能會需要往其他子樹搜尋，但是這種情況只有在  $\delta > q$  到目前的軸的距離時才會需要，不然我們可以忽略另一邊子樹，對答案沒有影響，在點隨機分佈的情況下運作速度在  $\mathcal{O}(\lg n)$ 。

該樹不會自動維持平衡，如果要進行動態查找刪除，可以考慮替罪羊樹、離線時間線段樹或是之前提到的可拓展集合的方法。

附上講師寫的 k-d tree 模板：

```

1 template<typename T>
2 class KDTree {
3     struct KDNODE {
4         Point<T> v;
5         KDNODE* ch[2];
6         KDNODE(const Point<T>& v, KDNODE *l, KDNODE *r) : v(v), ch{l, r} {}
7         ~KDNODE() {
8             for (size_t i : {0, 1}) if (ch[i]) ch[i]~KDNODE();
9         }
10        T dfs(const Point<T> q,

```

```

11     T dis = numeric_limits<T>::max(),
12     bool parity = 0) {
13     dis = min(dis, (v - q).norm());
14     bool isRight = parity ? v.x < q.x : v.y < q.y;
15     if (ch[isRight])
16         dis = min(dis, ch[isRight]->dfs(q, dis, parity^1));
17     if (ch[isRight^1] and [](T x){
18         return x * x;
19     }(parity ? v.x - q.x : v.y - q.y) < dis)
20         dis = min(dis, ch[isRight^1]->dfs(q, dis, parity^1));
21     return dis;
22 }
23 } *root;
24 KDNode *buildKDTree(typename vector<Point<T>>::iterator l,
25                      typename vector<Point<T>>::iterator r,
26                      bool parity = 0) {
27     if (r == l) return nullptr;
28     auto m = l + (r - l >> 1);
29     nth_element(l, m, r, [=](Point<T> a, Point<T> b) {
30         return parity ? a.x < b.x : a.y < b.y;
31     });
32     return new KDNode(*m,
33                       buildKDTree(l, m, parity^1),
34                       buildKDTree(m + 1, r, parity^1));
35 }
36 public:
37     KDTree(vector<Point<T>> A) : root{buildKDTree(A.begin(), A.end())} {}
38     T nearestNeighborSquareDistance(Point<T> q) {
39         return root->dfs(q);
40     }
41 };

```

程式碼 12.18: k-d tree and dfs

## 12.7 對偶性質

所謂對偶是指有兩類數學物件在做了一些轉換以後性質會互換，例如向量空間的對偶變換。在這裡我們要介紹點跟線的對偶變換。至於要不要知道對偶性質就見仁見智了，他只是一個小技巧配上一個很數學的專有名詞，這裡簡單介紹一下他的動機

- 把問題丟到對偶空間可能會對問題有新的看法
- 幫助對題目有更進一步的洞察
- 可能給你比較好實作的作法切題目

你可以把原本的問題用對偶換過去變成更難的問題，即使他們基本上是同一個問題，舉例來說，凸包優化問題可以轉成：

| Ex: Lena and Queries   | CF 678F 改 |
|--|-----------|
| 一開始有一個空集合 $S$ 有以下兩種操作要進行，操作共 $N$ 個：  |           |
| <ul style="list-style-type: none"><li>• 把一個整數點加進這個集合</li><li>• 把一個整數點移出這個集合</li><li>• 紿一條線性函數，問是不是所有點都在這一條直線的同一側</li></ul> |           |

對偶轉換可以把它變成剛剛的題目，讓我們來看看他怎麼做的。

### 12.7.1 基本性質

**定義 12.4.** 一個點  $(a, b)$  的對偶是直線函數  $f(x) = ax - b$ 。反之，一個對偶直線函數  $f(x) = ax + b$  的對偶是一個點  $(a, -b)$ 。

**定理 12.5.** 假設點  $p = (p_x, p_y)$  在直線函數  $f = ax + b$  之上，則直線函數  $f$  的對偶點  $f^* = (a, -b)$  在點  $p$  的對偶直線函數  $p^* = p_x x - p_y$  之上。特別地，兩者直線到點的垂直距離（只走  $y$  軸方向，非歐氏距離）會一樣。

*Proof.* 翻譯成數學的語言就是： $p_y > ap_x + b \iff -b > p_x a - p_y$

也因此  $p_y - (ap_x + b) = -b - (p_x a - p_y)$  □

12.5 也告訴我們，點在線上對偶過去會變成線在點上，也因此，多點共線對偶過去會變成多線共點。如果  $\vec{r}$  在  $\vec{pq}$  上，那麼對偶的  $r^*$  斜率會在  $p^*$  和  $q^*$  之間

**定理 12.6.** 兩條平行線  $f_1(x) = ax + b, f_2(x) = ax + c$  的垂直距離是他們對偶轉換的垂直距離

上述定理改成小於或是等於都會對，可以發現線跟點的性質互換了。

*Proof.* 兩條線對偶轉換過去會在同一條線  $x = a$  上，所以他們的距離跟原本一樣都是  $|b - c|$ 。□

### 12.7.2 凸包與直線函數的關係

直接講結論：對於一個直線函數集合  $S$ ，我們把每一個點的最大值寫成函數  $\mathcal{U}(x) = \max_{f \in S} f(x)$ ，從12.5 可以知道那些直線會形成一個凹口向上的凸包，我們稱凸包上的直線函數集合為  $P \subseteq S$ 。

好那我們看看對偶空間  $S^*$  發生了什麼事情， $P^*$  上的點剛好換變成  $S^*$  的下半凸包 ( Lower Convex Chain/Lower Hull )，所謂下凸包是指說，把最左邊跟最右邊的點選出來（如果  $x$  一樣就選最下面那個點），然後下面那個路徑選出來所集合的點，包含邊界兩點。

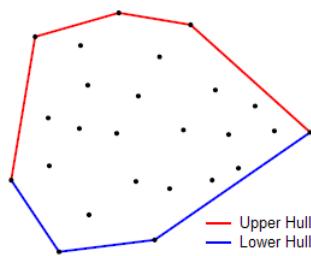


圖 12.21：上半凸包與下半凸包

與其給個繁複的證明，我們可以從直觀來看待這一件事情： $\mathcal{U}(x) = \max_{f \in S} f(x)$  之上的點  $\{(x, y) \in \mathbb{R}^2 | y \geq \mathcal{U}(x)\}$  到對偶空間就變成了這個對偶下半凸包  $P^*$  以下的直線，這些直線的上半平面的交集會是我們的凸包，把這些想法寫成證明筆者就不贅述。

## 12.8 感謝

本章的素材大多出自 CF 文章，在此感謝這些作者的耕耘：

- 感謝 Al.Cash 允許我使用他文章的程式碼跟圖片
- 感謝 meooow 允許我使用他的圖片

## 12.9 圖片來源

- 12.19 F52985 from Wikimedia Commons
- 12.20 KindDragon33 from Wikimedia Commons

# 其他解題技巧

## 13.1 暴力美學

暴力枚舉是程式競賽中的基礎技巧之一。本節會介紹某些暴力法的優化，以及看似暴力但實則複雜度良好的方法。

### 13.1.1 meet-in-the-middle

暴力枚舉方法通常涉及過多狀態，而在有些時候可以將狀態分兩部分枚舉，之後再用複雜度良好的方法（或資料結構）來將枚舉的結果「合併」。這個技巧被稱作 **meet-in-the-middle**。下面會透過例題來示範如何使用這個技巧。

| Maximum Subsequence  | Codeforces 888E |
|--|-----------------|
| 給定一個大小為 $n$ 的序列 $a$ 和一個整數 $m$ 。                              |                 |
| 要求從序列中選出一些元素，並最大化這些元素總和除 $m$ 的餘數。                            |                 |
| $1 \leq n \leq 35, 1 \leq a_i \leq 10^9, 1 \leq m \leq 10^9$ |                 |

先考慮暴力枚舉所有方案，再從中挑出最好的方案。

```

1 int n,m;
2 int a[40];
3 int ans = 0;      // 紀錄答案
4 // main function 裡呼叫 dfs(1,0) 來計算答案
5 void dfs(int idx,int sum) { // 當前處理到第 idx 個元素；總和為 sum
6     if(idx == n + 1) // 序列中所有元素都考慮完選/不選
7         ans = max(ans,sum);
8     else {
9         dfs(idx + 1, (sum + a[idx]) % m);    // 選第 idx 個元素
10        dfs(idx + 1, sum);                   // 不選第 idx 個元素
11    }
12 }
```

程式碼 13.1: Maximum Subsequences 暴力枚舉所有方案

接著分析時間複雜度： $a$  序列中每個元素各有選或不選兩種選擇，因此時間複雜度為  $\mathcal{O}(2^n)$ 。而最糟情況下，即  $n = 35$  時， $2^n \approx 3 \times 10^{10}$ ，必然是會超時的。

現在基於這暴力法使用 meet-in-the-middle。首先把整個序列從中間切下去，分成大小（幾乎）相等的兩個序列，接著對前半部分的序列使用暴力法枚舉所有方案，並記錄下來；再來一樣對後半部分序列用暴力法枚舉，而對於當前枚舉的方案（令當前的總和 mod  $m$  後為  $sum_{now}$ ），若要讓這個方案與前半部分枚舉的某個方案合併後（即相加）mod  $m$  要最大，那就貪心地從前半部分枚舉的方案中選擇「元素總和 mod  $m$  後小於  $m - sum_{now}$  中的最大的方案」來合併，如此一來合併後的元素總和 mod  $m$  後就會最接近  $m - 1$ （最佳解），然後再從後半部分枚舉並合併的方案中挑出總和 mod  $m$  後最大的即可。

而為了在後半部分枚舉時，快速查找前半部分最佳的合併方案，將前半部分枚舉出來的所有方案透過 `std::set` 紀錄下來，這樣便可在  $\mathcal{O}(\log 2^{\lfloor \frac{n}{2} \rfloor}) = \mathcal{O}(n)$  內完成單次的紀錄與查找。

```
1 int n, m;
2 int a[40];
3 set<int> pre_sum; // 維護前半部分枚舉的方案
4 void dfs1(int idx, int sum) {
5     if(idx == n / 2 + 1)
6         pre_sum.insert(sum);
7     else {
8         dfs1(idx + 1, (sum + a[idx]) % m);
9         dfs1(idx + 1, sum);
10    }
11 }
12 int ans = 0;
13 void dfs2(int idx, int sum) {
14     if(idx == n + 1) {
15         // 記得不要寫成 lower_bound(pre_sum.begin(), pre_sum.end(), m - sum)
16         auto it = pre_sum.lower_bound(m - sum);
17         if(it != pre_sum.begin()) { // 前半部分有總和(mod m)小於m - sum的方案
18             it--;
19             ans = max(ans, sum + (*it));
20         }
21     }
22     else {
23         dfs2(idx + 1, (sum + a[idx]) % m);
24         dfs2(idx + 1, sum);
25     }
26 }
27 // main function 裡呼叫 solve() 來計算答案
28 void solve() {
29     dfs1(1, 0);
30     dfs2(n / 2 + 1, 0);
31 }
```

程式碼 13.2: Maximum Subsequence 使用 meet-in-the-middle

最後來分析時間複雜度：前半部分枚舉並記錄需要  $\mathcal{O}(2^{\lfloor \frac{n}{2} \rfloor} \times n)$ ，後半部分枚舉並查找需要  $\mathcal{O}(2^{\lceil \frac{n}{2} \rceil} \times n)$ ，因此總時間複雜度為  $\mathcal{O}(2^{\lfloor \frac{n}{2} \rfloor} \times n + 2^{\lceil \frac{n}{2} \rceil} \times n) = \mathcal{O}(2^{\lceil \frac{n}{2} \rceil + 1} \times n)$ 。

| Xor-Paths  | Codeforces 1006F |
|--|------------------|
| <p>給定一個 <math>n \times m</math> 的網格圖，每格 <math>(i, j)</math> 都有一個權重 <math>a_{ij}</math>。<br/>         問有多少種從左上角 <math>(1, 1)</math> 走到右下角 <math>(n, m)</math> 的路徑數，滿足：</p> <ol style="list-style-type: none"> <li>若當前在格子 <math>(i, j)</math>，則下一步只能移動到格子 <math>(i + 1, j)</math> 或格子 <math>(i, j + 1)</math>。</li> <li>將路徑上經過的所有格子的權重 bitwise XOR 後必須與 <math>k</math> 相等。</li> </ol> <p><math>1 \leq n, m \leq 20, 0 \leq a_{ij} \leq 10^{18}, 0 \leq k \leq 10^{18}</math></p> |                  |

一樣先考慮樸素的暴力法，從左上角  $(1, 1)$  開始暴力 DFS ( BFS 也可以 ) 嘗試所有走到右下角  $(n, m)$  的路徑，再檢查路徑上所有格子的權重 xor 後是不是  $k$ 。

```

1 int n,m;
2 long long k,a[25][25];
3 long long ans = 0; // 紀錄答案
4 // main function 裡呼叫 dfs(1,1,a[1][1]) 來計算答案
5 void dfs(int pos_i,int pos_j,long long now_xor) {
6     if(pos_i == n && pos_j == m) { // 抵達右下角
7         if(now_xor == k)
8             ans++;
9     }
10    else {
11        if(pos_i + 1 <= n) // 往下走
12            dfs(pos_i + 1, pos_j, now_xor ^ a[pos_i + 1][pos_j]);
13        if(pos_j + 1 <= m) // 往右走
14            dfs(pos_i, pos_j + 1, now_xor ^ a[pos_i][pos_j + 1]);
15    }
16 }
```

程式碼 13.3: Xor-Paths 暴力 DFS

再來分析時間複雜度：每條路徑長度皆為  $n + m - 1$  個格子（移動到不同格子  $n + m - 2$  次），每次要往下個格子移動時，至多會有兩種選擇，因此時間複雜度為  $\mathcal{O}(2^{n+m-2})$ <sup>i</sup>。最糟情況下，即  $n = m = 20 \cdot 2^{n+m-2} \approx 2 \times 10^{11}$ ，顯然會超時。

現在觀察一下枚舉的過程，令任一合法路徑經過的格子依序為  $(i_1, j_1), (i_2, j_2), \dots, (i_{n+m-1}, j_{n+m-1})$ ，若滿足經過的所有格子權重 xor 後是  $k$ ，即  $a_{i_1,j_1} \oplus a_{i_2,j_2} \oplus \dots \oplus a_{i_{n+m-1},j_{n+m-1}} = k$ ，則  $k \oplus a_{i_{x+1},j_{x+1}} \oplus \dots \oplus a_{i_{n+m-1},j_{n+m-1}} = a_{i_1,j_1} \oplus a_{i_2,j_2} \oplus \dots \oplus a_{i_x,j_x}$ （還記得 xor 的性質嗎？ $a \oplus b = k \implies k \oplus b = a$ ），那麼就可以如同前一題 Maximum Subsequence 一樣，先枚舉所有走前  $x$  格的方案並紀錄下來，再枚舉走後面  $n + m - 1 - x$  格的方案後與前面紀錄的資訊合併即可！

<sup>i</sup>用組合數學可估出較緊的複雜度（也是會超時），這邊為了後續分析方便故不採用較緊的複雜度

為了加速後半部分枚舉時查找的速度，使用 `std::map` 來紀錄與查找資訊，如此便可在時間複雜度  $\mathcal{O}(\log 2^{x-1}) = \mathcal{O}(x)$  內完成紀錄與查找資訊。

實作上，令  $x = n$ ；後半部分的枚舉，從右下角  $(n, m)$  往回走。

```

1 int n,m;
2 long long k,a[25][25];
3 // cnt[i][j][v] : 從(1,1)走到(i,j)且路徑上所有格子權重 xor 後為 v 的路徑數
4 map<long long,int> cnt[25][25];
5 void dfs1(int pos_i,int pos_j,long long now_xor) {
6     if(pos_i + pos_j == n + 1) {
7         cnt[pos_i][pos_j][now_xor]++;
8     }
9     else {
10         if(pos_i + 1 <= n)
11             dfs1(pos_i + 1, pos_j, now_xor ^ a[pos_i + 1][pos_j]);
12         if(pos_j + 1 <= m)
13             dfs1(pos_i, pos_j + 1, now_xor ^ a[pos_i][pos_j + 1]);
14     }
15 }
16 long long ans = 0;
17 void dfs2(int pos_i,int pos_j,long long now_xor) {
18     if(pos_i + pos_j == n + 1) {
19         // a[pos_i][pos_j] 被重複 xor 到，故這邊要再 xor 一次
20         long long target = k ^ now_xor ^ a[pos_i][pos_j];
21         if(cnt[pos_i][pos_j].find(target) != cnt[pos_i][pos_j].end())
22             ans += cnt[pos_i][pos_j][target];
23     }
24     else {
25         if(pos_i - 1 >= 1) // 往上走
26             dfs2(pos_i - 1, pos_j, now_xor ^ a[pos_i - 1][pos_j]);
27         if(pos_j - 1 >= 1) // 往左走
28             dfs2(pos_i, pos_j - 1, now_xor ^ a[pos_i][pos_j - 1]);
29     }
30 }
31 // main function 裡呼叫 solve() 計算答案
32 void solve() {
33     dfs1(1,1,a[1][1]);
34     dfs2(n,m,a[n][m]);
35 }
```

#### 程式碼 13.4: Xor-Paths 使用 meet-in-the-middle

最後分析時間複雜度：前半部分的枚舉並記錄需要  $\mathcal{O}(2^{n-1} \times n)$ 。後半部分枚舉並查找需要  $\mathcal{O}(2^{m-1} \times n)$ 。因此總時間複雜度為  $\mathcal{O}((2^{n-1} + 2^{m-1}) \times n) = \mathcal{O}((2^n + 2^m) \times n)$ 。

### 13.1.2 啟發式合併

在介紹啟發式合併之前，先來看道例題。

| 集合維護問題  | 經典問題 |
|---|------|
| 現在有 $N$ 個集合，第 $i$ 個集合初始有 $i$ 一個元素。請 $Q$ 次支援兩種操作：<br>1. 將元素 $x$ 所在的集合與元素 $y$ 所在的集合合併<br>2. 詢問元素 $x$ 與元素 $y$ 是否在同一個集合裡<br>$1 \leq N, Q \leq 10^5, 1 \leq x, y \leq N$ |      |

看起來很眼熟？沒錯，這就是 Disjoint Set 的模板題。來看一下暴力法：

```
1 vector<int> Set[100005];      // Set[i] : 第 i 個集合
2 int where[100005];    // where[i] : 元素 i 所在的集合編號
3 void init(int n) { // 初始化
4     for(int i=1;i<=n;i++) {
5         Set[i].push_back(i);
6         where[i] = i;
7     }
8 }
9 void merge(int x,int y) { // 將 x 與 y 所在的集合合併
10    if(where[x] != where[y]) {
11        int X = where[x] , Y = where[y];
12        for(int val : Set[X]) { // 將 x 所在集合的全部元素移到 y 所在的集合
13            Set[Y].push_back(val);
14            where[val] = Y;
15        }
16        // Set[X].clear(); // 清空原本 x 所在的集合；不做不影響答案正確性
17    }
18 }
19 bool same(int x,int y) { // 詢問 x 與 y 是否在同一個集合
20     return where[x] == where[y];
21 }
```

程式碼 13.5：暴力法解集合維護問題

明顯地，這份 code 的時間複雜度是  $\mathcal{O}(QN)$

接著在這份 code 中插入兩行（下面 code 的 12、13 行）。

```

1 vector<int> Set[100005];
2 int where[100005];
3 void init(int n) {
4     for(int i=1;i<=n;i++) {
5         Set[i].push_back(i);
6         where[i] = i;
7     }
8 }
9 void merge(int x,int y) {
10    if(where[x] != where[y]) {
11        int X = where[x] , Y = where[y];
12        if(Set[X].size() > Set[Y].size()) // x 所在集合大小比 y 所在集合大
13            swap(X,Y); // 後面改成將 y 所在集合的全部元素移到 x 所在的集合
14        for(int val : Set[X]) {
15            Set[Y].push_back(val);
16            where[val] = Y;
17        }
18        // Set[X].clear();
19    }
20 }
21 bool same(int x,int y) {
22     return where[x] == where[y];
23 }

```

**程式碼 13.6:** 啟發式合併解集合維護問題

這份 code 的時間複雜度變為  $\mathcal{O}(N \log N + Q)$ 。下面來分析為什麼時間複雜度有這麼大的差異。

首先複雜度改變的原因出在集合合併上，現在讓我們把焦點放到個別的元素上；**單個元素什麼時候會對執行時間造成影響？**(什麼時候會「花費到時間」) 就在將它從它所在的集合移動到新的集合的時候。將每個元素移動的時間花費加總在一起，就是合併操作總花費的時間了。而新加入的兩行 code，保證了**每次合併時，都是將大小較小的集合併往大小較大的集合**，也就是說，**每個元素每次移動（向別的集合合併）時，它所在的集合大小都至少會變成原來的兩倍**，至此便可以得知，每個元素最多移動  $\log N$  次（移動  $\log N$  次後就不能再移動了，因為集合大小最多只能到  $N$ ），而共有  $N$  個元素，因此合併操作的時間複雜度就是  $\mathcal{O}(N \log N)$ 。

上面這種保證集合大小變化都會至少變為兩倍的技巧，就是**啟發式合併**了。

啟發式合併也時常出在一些跟樹有關的問題，一樣看道例題。

給定一棵大小為  $n$  的有根樹，樹根為 1。每個節點  $i$  都被塗上一種顏色  $c_i$ 。  
 對於任一棵子樹，如果某顏色  $c$  出現最多次，則稱顏色  $c$  支配這棵子樹（可能有不只一種顏色支配同一棵子樹）。

對於以這  $n$  個點為根的  $n$  棵子樹，分別輸出支配該棵子樹的顏色總和（把顏色視為數字加總）。

$1 \leq n \leq 10^5, 1 \leq c_i \leq n$

乍看之下不好下手，先考慮樸素的暴力法：遞迴枚舉所有子樹，並維護每棵子樹中各顏色的出現次數及各出現次數的顏色總和。

```

1 const int maxn = 100005;
2 int col[maxn]; // col[i] : 節點 i 的顏色
3 vector<int> G[maxn]; // G[u] : 與點 u 相鄰的點
4 int cnt[maxn]; // cnt[i] : 顏色 i 出現次數
5 int most = 0; // 為當前出現次數最多的次數
6 long long sum[maxn]; // sum[i] : 出現次數為 i 次的顏色總和
7 long long ans[maxn]; // ans[u] : 支配 u 為根的子樹的顏色總和
8 void add(int c) { // 顏色 c 出現次數加一次
9     sum[cnt[c]] -= c;
10    cnt[c]++;
11    sum[cnt[c]] += c;
12    most = max(most, cnt[c]);
13 }
14 void sub(int c) { // 顏色 c 出現次數減一次
15     sum[cnt[c]] -= c;
16     cnt[c]--;
17     sum[cnt[c]] += c;
18     if(sum[most] == 0) most--;
19 }
20 void update(int u, int pa, bool clear) { // 將 u 為根的子樹資訊加入或清除
21     if(!clear)
22         add(col[u]);
23     else
24         sub(col[u]);
25     for(int v : G[u])
26         if(v != pa)
27             update(v, u, clear);
28 }
29 void dfs(int u, int pa) { // 遞迴計算 u 為根的子樹的答案
30     for(int v : G[u]) {
31         if(v != pa) {
32             dfs(v, u);
33             update(v, u, true); // 消去資訊避免影響其他子樹的答案計算
34         }
35     }
36     update(u, pa, false); // 將 u 為根的子樹的資訊更新
37     ans[u] = sum[most];
38 }
```

```

39 // main function 裡呼叫 solve(n) 計算答案
40 void solve(int n) {
41     dfs(1, 1);
42     for(int i=1; i<=n; i++) {
43         if(i != 1) cout << " ";
44         cout << ans[i];
45     }
46     cout << '\n';
47 }

```

程式碼 13.7：暴力枚舉所有子樹解 Lomsat gelral

共有  $n$  棵子樹，計算一棵子樹答案（更新一棵子樹的資訊）需要  $\mathcal{O}(n)$ （單個點被 add/sub 需要  $\mathcal{O}(1)$ ），故時間複雜度為  $\mathcal{O}(n^2)$ 。明顯這個複雜度會超時。

現在使用啟發式合併來優化這個暴力法。一樣把焦點放在個別節點上，一個點在它的資訊被更新時才會對時間造成影響，也就是 code 裡 add 與 sub 兩個函式，而 sub 必是在 add 後才會呼叫，所以可以只考慮 add 的部分；與前面類似，關鍵在於如何讓一個點每次被 add 時，當前要計算答案的子樹大小會是前一棵大小的至少兩倍。而只要每次遞迴求子樹答案時，**先遞迴輕兒子**（重兒子以外的子節點）為根的子樹，並在遞迴結束時消去它們造成的影響，**最後再去遞迴重兒子**（子樹大小最大的子節點），**並將遞迴結果保留重複使用**，如此便可以保證任意點資訊每次被加入時（因該點或該點祖先呼叫上面 code 第 36 行），當前的子樹大小必至少變為前一次（更新該點的資訊）的子樹大小的兩倍。

以下圖來說（節點旁邊的數字為該點為根的子樹大小；塗色點表示其為父節點的重兒子）：點  $u$  某次因要計算其祖先為根的子樹答案時被遞迴遍歷到，而在下次又被遞迴到時，當前要計算答案的點為根的子樹大小變為前次的兩倍多。

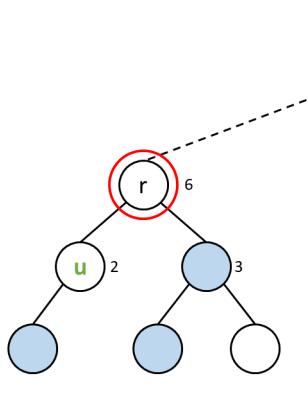


圖 13.1：點  $r$  計算答案遞迴輕兒子時，點  $u$  被 add；當前子樹大小為 6

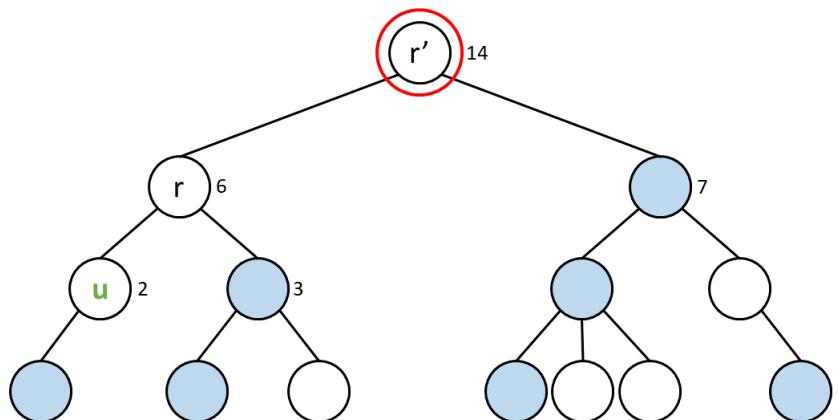


圖 13.2：點  $r'$  計算答案遞迴輕兒子時，點  $u$  被 add；子樹大小變為 14

```

1 // sz[u] : u 為根的子樹大小 ; max_son[u] : u 的重兒子
2 int sz[maxn],max_son[maxn];
3 void build(int u,int pa) { // 找出所有點的重兒子
4     sz[u] = 1;
5     max_son[u] = -1;
6     for(int v : G[u])
7         if(v != pa) {
8             build(v,u);
9             sz[u] += sz[v];
10            if(max_son[u] == -1 || sz[v] > sz[max_son[u]])
11                max_son[u] = v;
12        }
13    }
14 void dfs(int u,int pa) {
15     for(int v : G[u])
16         if(v != pa && v != max_son[u]) { // 遞迴計算輕兒子答案
17             dfs(v,u);
18             update(v,u,true); // 消去資訊避免影響其他子樹的答案計算
19         }
20         if(max_son[u] != -1) dfs(max_son[u],u); // 最後遞迴重兒子並保留資訊
21     for(int v : G[u]) // 將所有輕兒子子樹的資訊加入
22         if(v != pa && v != max_son[u])
23             update(v,u,false);
24     add(col[u]);
25
26     ans[u] = sum[most];
27 }
28 // main function 裡呼叫 solve(n) 計算答案
29 void solve(int n) {
30     build(1,1);
31     dfs(1,1);
32     for(int i=1;i<=n;i++) {
33         if(i != 1) cout << " ";
34         cout << ans[i];
35     }
36     cout << '\n';
37 }

```

**程式碼 13.8:** 樹上啟發式合併解 Lomsat gelral

最後分析時間複雜度：找出所有點的重兒子需要  $\mathcal{O}(n)$ ；每個點因 23 行而被 add 最多  $\log n$  次，因 24 行被 add 恰好一次，而共有  $n$  個點。因此時間複雜度為  $\mathcal{O}(n + n \log n + n) = \mathcal{O}(n \log n)$ 。

樹上啟發式合併的複雜度分析比較困難，讀者可以停下來多想一會。

### 13.1.3 均攤分析

均攤分析是一種分析時間複雜度的技巧。我們在分析算法的時間複雜度時，會估計最糟情況的時間花費作為複雜度，但有時這樣的粗略估計是不夠精確的，特別是在最糟情況發生次數不多或有限制時。下面透過一個例子來說明。

| stack 維護   | 經典問題 |
|--|------|
| <p>請維護一個類似 stack 的資料結構。試 <math>N</math> 次支援兩種操作：</p> <ol style="list-style-type: none"><li>1. 將一個元素 <math>v</math> 推入 ( push ) stack 裡</li><li>2. 將 stack 裡元素全部清空</li><li>3. 輸出當前 stack 頂端 ( top ) 的元素 ( 保證當下 stack 非空 )</li></ol> <p><math>1 \leq N \leq 10^5, 1 \leq v \leq N</math></p> |      |

可以很輕易地用 std::stack 來完成本題。

```
1 #include<bits/stdc++.h>
2 using namespace std;
3 int main() {
4     int N;
5     cin >> N;
6     stack<int> S;
7     while(N--) {
8         int type;
9         cin >> type;
10        if(type == 1) { // 操作一
11            int v;
12            cin >> v;
13            S.push(v);
14        }
15        else if(type == 2) { // 操作二
16            while(!S.empty())
17                S.pop();
18        }
19        else { // 操作三
20            cout << S.top() << '\n';
21        }
22    }
23    return 0;
24 }
```

程式碼 13.9: stack 操作

接著來分析時間複雜度：

操作一：stack 推入一個元素需要  $\mathcal{O}(1)$ ，因此操作一複雜度為  $\mathcal{O}(1)$ 。

操作二：stack 彈出一個元素需要  $\mathcal{O}(1)$ ，stack 裡最多有  $N$  個元素，因此操作二複雜度為  $\mathcal{O}(N)$ 。

操作三：查看 stack 頂端元素需要  $\mathcal{O}(1)$ ，因此操作三複雜度為  $\mathcal{O}(1)$ 。

每筆操作最多各做  $N$  次，故總時間複雜度為  $\mathcal{O}(N + N^2 + N) = \mathcal{O}(N^2)$ 。

有沒有覺得哪裡怪怪的？沒錯，一次操作二雖然最糟需要  $\mathcal{O}(N)$ ，但總的來看，**操作二的花費總和最多也是  $\mathcal{O}(N)$** （能彈出的元素數量最多就與推入的數量相等，既然最多只能推入  $N$  個元素，那能彈出的數量勢必最多就是  $N$  個）。因此總時間複雜度應為  $\mathcal{O}(N + N + N) = \mathcal{O}(N)$ 。我們可以稱呼操作二的複雜度最糟需要  $\mathcal{O}(N)$ ，但均攤是  $\mathcal{O}(1)$ （總複雜度  $\mathcal{O}(N)$ / 最多執行  $N$  次）。

| 樓層交換  | TNFSH OJ 512 |
|---|--------------|
| 給定大小為 $N$ 的序列 $arr$ ，序列初始為 $(1, 2, 3, \dots, N)$ 。試 $Q$ 次支援兩種操作：  |              |
| 1. 將 $arr_a$ 與 $arr_b$ 交換   |              |
| 2. 給定 $l, r$ 。將元素 $l$ 與 $arr_l$ 交換、將元素 $l + 1$ 與 $arr_{l+1}$ 交換……將元素 $r$ 與 $arr_r$ 交換。最後輸出有效交換次數。交換的兩元素相異稱這交換為有效交換。 |              |
| $1 \leq N, Q \leq 5 \times 10^5, 1 \leq l \leq r \leq N$  |              |

|   |
|---|
| 1      5      2      4      3      7      6   |
| 交換 $arr_2$ 與元素 $2(arr_3)$ 後 → 1 <u>2</u> <u>5</u> 4      3      7      6<br><small>(有效交換)</small> |
| 交換 $arr_3$ 與元素 $3(arr_5)$ 後 → 1      2 <u>3</u> 4 <u>5</u> 7      6<br><small>(有效交換)</small>      |
| 交換 $arr_4$ 與元素 $4(arr_4)$ 後 → 1      2      3 <u>4</u> 5      7      6<br><small>(不是有效交換)</small> |

圖 13.3：對序列  $(1, 5, 2, 4, 3, 7, 6)$  執行操作二 ( $l = 2, r = 4$ )；有效交換次數為 2。  
因此操作二簡單講就是透過交換將區間  $[l, r]$  復原成  $(l, l + 1, \dots, r)$ 。

操作的時間花費固然來自兩種操作內的交換，而操作二內非有效交換的交換必然是沒有執行的必要，因此可能會直覺地在操作二時僅執行有效交換。我們就先從如何「操作二時僅執行有效交換」下手。

要維護的資訊是「所有的元素  $i$  其滿足  $arr_i \neq i$ 」，即一個集合，而在操作二時只需對集合內屬於操作區間的元素進行交換就好。而實作可以採用 `std::set` 維護集合，在操作一交換元素後，如果元素符合條件則加入集合（如果當前不在集合內），否則從集合中剷除（如果當前在集合內）；操作二時，持續從集合中二分搜要進行有效交換的元素，並一樣更新集合，直到集合裡不存在元素屬於操作區間。

下面實作使用一個陣列紀錄各元素於序列中的位置（主要用於操作二）<sup>ii</sup>。

```
1 const int maxn = 500005;
2 int arr[maxn]; // arr[p]: 序列中位置 p 的元素
3 int pos[maxn]; // pos[p]: 元素 p 在序列中的位置
4 set<int> idxs; // 維護集合 {p | arr[p] != p}
5 void my_swap(int a, int b) { // 將序列位置 a 的元素與位置 b 的元素交換
6     if(a == b) return;
7     swap(pos[arr[a]], pos[arr[b]]); // pos 陣列也要記得交換
8     swap(arr[a], arr[b]);
9
10    // 更新 idxs
11    if(arr[a] != a)
12        idxs.insert(a);
13    else if(idxs.count(a) == 1)
14        idxs.erase(a);
15
16    if(arr[b] != b)
17        idxs.insert(b);
18    else if(idxs.count(b) == 1)
19        idxs.erase(b);
20 }
21 // main function 裡呼叫 solve(N,Q)
22 void solve(int N, int Q) {
23     for(int i=1; i<=N; i++)
24         arr[i] = i, pos[i] = i;
25     while(Q--) {
26         int type;
27         cin >> type;
28         if(type == 1) { // 交換 arr[a] 和 arr[b]
29             int a, b;
30             cin >> a >> b;
31             my_swap(a, b);
32         }
33         else { // 復原 [l,r]
34             int l, r;
35             cin >> l >> r;
36             int ans = 0;
37             // 每次對要進行有效交換的位置與對應元素的位置進行交換
38             while(true) {
39                 auto it = idxs.lower_bound(l);
40                 if(it == idxs.end() || *it > r) break;
41                 my_swap(*it, pos[*it]);
42                 ans++;
43             }
44             cout << ans << '\n';
45         }
46     }
47 }
```

### 程式碼 13.10: 解樓層交換

<sup>ii</sup>也可以將其位置一起紀錄在 set 裡（自定義結構或使用 std::pair），看同學的實作習慣

接著分析時間複雜度：執行 `my_swap` 函式時，會對 `std::set` 進行操作（加入與刪除），因此複雜度為  $\mathcal{O}(\log N)$ ；操作一時會執行一次 `my_swap`，而最多有  $Q$  次操作一；操作二時最多執行  $N$  次 `my_swap` 且每次都會先二分搜一次，而最多有  $Q$  次操作二。故總時間複雜度為  $\mathcal{O}(Q \log N + QN(\log N + \log N)) = \mathcal{O}(QN \log N)$ 。

如果你有想過暴力法（同時維護各元素出現位置）的話，那暴力法是可以做到  $\mathcal{O}(QN)$  的。當然上面作法和之前的複雜度分析一樣，複雜度的分析不夠緊。與前面 `stack` 例題類似，**操作二的有效交換次數其實與之前操作一「打亂」的次數有關**（`stack` 清空的元素數量與推入的數量有關）。也就是如果操作二時復原（有效交換）了位置  $p$ （將  $arr_p$  與元素  $p$  交換），那麼在這之前必有一次操作一將位置  $p$  的元素透過交換將其換至位置  $p$ 。因此只要操作二時僅去執行有效交換，便能確保總有效交換次數  $\leq 2 \times$  總操作一次數）。

所以操作二的總複雜度應為有效交換次數乘上有效交換的複雜度，最多有效交換  $Q$  次，故複雜度為  $\mathcal{O}(Q \log N)$ 。總時間複雜度為  $\mathcal{O}(Q \log N + Q \log N) = \mathcal{O}(Q \log N)$ 。

而在其他課程裡有時也有用到均攤分析，像基礎資料結構提到的 Disjoint Set 與進階資料結構中的線段樹區間取餘皆是使用了均攤分析來分析時間複雜度。

### 13.1.4 bitset 優化常數

在基本語法知識的章節有提到，`std::bitset` 對  $N$  位 bits 進行操作，時間複雜度是  $\mathcal{O}(\frac{N}{32})$ （複雜度與一般陣列一樣，但常數較小）。有些題目若利用 `bitset` 優化批量布林操作，能讓某些作法（通常看起來很暴力）的時間壓到時限內。下面一樣透過題目來介紹如何應用。

| 冰塊塔  | TIOJ 1993 |
|--|-----------|
| $t$ 次回答這個問題：給定 $n$ 套物品組合，每套物品組合 $i$ 裡恰有三個物品，重量分別為 $x_i, y_i, z_i$ ，要求選出一些物品放入負重上限為 $h$ 的背包裡，並最大化物品重量總和；同套物品組合裡的物品須有恰好一件被放入背包裡。 |           |
| $1 \leq t \leq 20, 1 \leq n \leq 10^3, 1 \leq h \leq 10^5, 1 \leq x_i, y_i, z_i \leq 10^4$                                     |           |

題目是 0/1 背包問題的小變形，直接使用動態規劃  $\mathcal{O}(n \times h)$  求解。

```

1  bool dp[100005];
2  void solve(int n,int h) {
3      memset(dp,false,sizeof dp);
4      dp[0] = true;
5      for(int i=0,x,y,z;i<n;i++) { // 題目給定的 n 套物品組合
6          cin >> x >> y >> z;
7          for(int j=h;j>=0;j--) { // 更新 DP 狀態
8              bool new_state = false;
9              if(j - x >= 0) new_state |= dp[j - x];
10             if(j - y >= 0) new_state |= dp[j - y];
11             if(j - z >= 0) new_state |= dp[j - z];
12             dp[j] = new_state;
13         }
14     }
15     for(int i=h;i>0;i--)
16     if(dp[i]) {
17         cout << i << '\n';
18         return;
19     }
20     cout << "no solution" << '\n';
21 }

```

**程式碼 13.11:** 動態規劃解冰塊塔

但在最糟情況下，這做法是會超時的。而第 7 行迴圈進行的多次狀態轉移，其實便是將 `dp` 這個 `bool` 陣列平移後做 bitwise OR，因此我們改用 `bitset` 維護 DP 表格，來優化狀態轉移時花費時間的常數（透過 `bitset` 對狀態做 bitwise OR 時間複雜度是  $\mathcal{O}(\frac{h}{32}) = \mathcal{O}(h)$ ，即複雜度是一樣，僅是優化了常數）。

```

1  void solve(int n,int h) {
2      bitset<100005> dp;
3      dp[0] = 1;
4      for(int i=0,x,y,z;i<n;i++) {
5          cin >> x >> y >> z;
6          dp = (dp << x) | (dp << y) | (dp << z);
7      }
8      for(int i=h;i>0;i--)
9      if(dp[i] == 1) {
10         cout << i << '\n';
11         return;
12     }
13     cout << "no solution" << '\n';
14 }

```

**程式碼 13.12:** 動態規劃搭配 `bitset` 解冰塊塔

給定二維平面上  $n$  個相異點，點  $i$  的座標是  $(x_i, y_i)$ 。

要求找到最大的  $R$  滿足：存在三個整數  $a, b, c (1 \leq a < b < c \leq n)$ ，使得以  $(x_a, y_a), (x_b, y_b), (x_c, y_c)$  各為圓心且半徑皆為  $R$  的三個圓兩兩不相交（可相切）。

$1 \leq n \leq 3000, -10^4 \leq x_i, y_i \leq 10^4$

時限：九秒。

首先可以對  $n$  個點兩兩建邊，而邊權為兩點的距離，這樣就變成一道圖論題了：給定一張完全圖，要求找三個點，最大化這三個點兩兩邊權中的最小值除以二（像這種平面多點轉成圖論模型也是比賽中常見的一種套路）。

接著將所有邊依照邊權由大到小排序，然後將所有邊依序加入到圖裡，加入的同時對每個點維護與它距離為二的點集合，這樣當加入了一條邊  $(u, v)$ （假設邊權為  $w$ ），且  $u$  與  $v$  當前的距離為二，那麼問題的答案就會是  $\frac{w}{2}$ ；簡化來講就是依照邊權由大到小把邊加入圖裡，一旦加入某條邊（令邊權為  $w$ ）後會使得圖上有三角形（三個點兩兩相鄰），那答案就是  $\frac{w}{2}$ ，概念上與 Kruskal's algorithm 求最小生成樹有點類似（不斷加邊直到圖中所有點都連通了）。

```

1 const int maxn = 3005;
2 struct point {
3     int x,y;
4 }arr[maxn]; // 紀錄給定的 n 個點
5 int cal_dis(point a,point b) { // 計算兩點距離的平方
6     int A = a.x - b.x , B = a.y - b.y;
7     return A*A + B*B;
8 }
9 struct edge { // 轉成圖論模型的邊
10     int u,v,cost;
11 };
12 bool dis_1[maxn][maxn]; // dis_1[u][v] : u 與 v 距離是否為 1
13 bool dis_2[maxn][maxn]; // dis_2[u][v] : u 與 v 距離是否為 2
14 void solve(int n) {
15     vector<edge> E; // 儲存所有邊
16     for(int i=0;i<n;i++) // 兩兩建邊，邊權為兩點的距離平方
17         for(int j=i+1;j<n;j++)
18             E.push_back(edge{i,j,cal_dis(arr[i],arr[j])});
19     // 依照邊權由大到小排序
20     sort(E.begin(),E.end(),[=](edge a,edge b){return a.cost > b.cost;});
21
22     for(edge e : E) { // 由大到小將邊加入
23         int u = e.u , v = e.v; // 當前的邊為 (u,v)
24         if(dis_2[u][v] || dis_2[v][u]) { // 若 u 與 v 距離為二，則得到答案
25             cout << fixed << setprecision(7) << sqrt(e.cost) / 2.0 << '\n';
26             return;
27     }
28 }
```

```

30  /*更新 u 與 v 其距離為二的點集合；只更新單向，故 25 行須對兩個都 OR*/
31  for(int k=0;k<n;k++) {
32      // 與 v 距離為一的點，在加入邊 (u,v) 後，u 與它們的距離為二
33      dis_2[u][k] |= dis_1[v][k];
34      // 與 u 距離為一的點，在加入邊 (u,v) 後，v 與它們的距離為二
35      dis_2[v][k] |= dis_1[u][k];
36  }
37  dis_1[u][v] = dis_1[v][u] = 1; // u 與 v 兩點相鄰(距離為一)
38 }
39 }
```

**程式碼 13.13:** 轉圖論後解 Summer Earnings

接著分析時間複雜度： $n$  個點兩兩建邊的複雜度為  $\mathcal{O}(n^2)$ ；將所有邊排序需要  $\mathcal{O}(n^2 \log n^2)$ ；再來枚舉每條邊並加入圖裡需要  $\mathcal{O}(n^2)$ 。同時更新與  $u, v$  距離為二的點需要  $\mathcal{O}(n)$ 。故總時間複雜度為  $\mathcal{O}(n^2 + n^2 \log n^2 + n^3) = \mathcal{O}(n^3)$ 。

明顯是會超時的。現在使用 `bitset` 優化加入邊時做的大量布林操作（更新與  $u, v$  距離為二的點集合）。

```

1 void solve(int n) {
2     bitset<maxn> dis_1[maxn];
3     bitset<maxn> dis_2[maxn];
4     vector<edge> E;
5     for(int i=0;i<n;i++)
6         for(int j=i+1;j<n;j++)
7             E.push_back(edge{i,j,cal_dis(arr[i],arr[j])});
8     sort(E.begin(),E.end(),[=](edge a,edge b)->bool
9         {return a.cost > b.cost;});
10
11    for(edge e : E) {
12        int u = e.u, v = e.v;
13        if(dis_2[u][v] || dis_2[v][u]) {
14            cout << fixed << setprecision(7) << sqrt(e.cost) / 2.0 << '\n';
15            return;
16        }
17        dis_2[u] |= dis_1[v];
18        dis_2[v] |= dis_1[u];
19        dis_1[u][v] = dis_1[v][u] = 1;
20    }
21 }
```

**程式碼 13.14:** bitset 優化解 Summer Earnings

時間複雜度一樣是  $\mathcal{O}(n^3)$ ，而  $\frac{n^3}{32} \approx 8 \times 10^8$ ，但實際上不會真的執行那麼多次基本操作運算（邊實際數量不為  $n^2$ 、加入一定數量的邊後必會存在三角形等）。

有興趣的同學可以思考一下不用 `bitset` 的幾何解法。另外 `bitset` 優化常數屬於較少見且非常規的技巧，學習的同時切記不要走火入魔。

### 13.1.5 練習題 Exercise

題目名稱後標註「\*」的題目為筆者認為較為困難的題目。

|  |                 |
|--|-----------------|
| Lizard Era: Beginning  | Codeforces 585D |
| 有三個整數 $L, M, W$ ，初始皆為 0。<br>接著有 $n$ 個事件，每個事件 $i$ 給定三個整數 $l_i, m_i, w_i$ ，分別對應 $L, M, W$ ，每個事件都必須從給定的三個整數中挑出兩個，並讓對應的整數加上其值。<br>求一個方案使得最終 $L = M = W$ （可能無解）。若有多個方案，則最大化 $L$ 。<br>$1 \leq n \leq 25, -10^7 \leq l_i, m_i, w_i \leq 10^7$ |                 |

|   |                  |
|---|------------------|
| Make Them Similar*  | Codeforces 1257F |
| 給定一個大小為 $n$ 的序列 $a$ 。<br>試找到任意一個 $x$ 使得 $a$ 中所有元素分別對 $x$ 做 bitwise XOR 後的值在二進制下 1 出現的次數相同（可能無解）。若存在 $x$ 須滿足 $0 \leq x \leq 2^{30} - 1$ 。<br>$1 \leq n \leq 100, 1 \leq a_i \leq 2^{30} - 1$ |                  |

|   |                 |
|---|-----------------|
| VBlood Cousins  | Codeforces 208E |
| 給定一個 $n$ 個點的有根樹。<br>試 $m$ 次回答：有多少點往根走 $p_i$ 步後停下的點與點 $v_i$ 往根走 $p_i$ 步停下的一樣。<br>$1 \leq n, m \leq 10^5$ |                 |

|   |                  |
|---|------------------|
| Vicky's Delivery Service*   | Codeforces 1166F |
| 給定一張 $n$ 個點 $m$ 條邊的無向圖，第 $i$ 條邊有個顏色 $c_i$ 。<br>令一條路徑（walk）依序經過的邊顏色分別為 $x_1, x_2, \dots, x_k$ ，若對於所有的 $1 \leq i \leq \lfloor \frac{k}{2} \rfloor$ ，滿足 $x_{2i-1} = x_{2i}$ ，則這條路徑為雙彩虹路徑。<br>現在有 $q$ 個事件，每個事件 $i$ 為兩種類型的其中一種：<br>1. 在點 $x_i$ 與點 $y_i$ 之間新增一條顏色為 $z_i$ 的邊<br>2. 詢問是否存在一條從點 $x_i$ 到點 $y_i$ 的雙彩虹路徑<br>保證圖上任意時刻皆不會有重邊或自環。<br>$2 \leq n \leq 10^5, 1 \leq m, q \leq 10^5$ |                  |

### Galactic Collegiate Programming Contest

Codeforces 101572G

有  $n$  個隊伍在比賽，根據每個隊伍解出的題數與罰時來排名。每個隊伍初始解開的題數與罰時皆為 0。

假設隊伍  $x$  解出的題數與罰時分別為  $a_x, b_x$ ，隊伍  $y$  解出的題數與罰時分別為  $a_y, b_y$ ；若  $a_x > a_y$  或  $a_x = a_y$  同時  $b_x < b_y$ ，則稱隊伍  $x$  的表現比隊伍  $y$  好。

一個的隊伍排名若為第  $k + 1$  名，則表示有恰好  $k$  個隊伍的表現比該隊伍好。

現在有  $m$  筆事件，每筆事件  $i$  代表某隊伍  $t_i$  多解出一題，其罰時增加  $p_i$ 。請在每個事件後，輸出當前隊伍 1 的排名。

$1 \leq n \leq 10^5, 1 \leq m \leq 10^5$

( 試試看只用 STL 解本題 ! )

### Connected Components?

Codeforces 920E

有一張  $n$  個點的無向圖。

給定  $m$  個點對  $(u_i, v_i)$ ，表示圖上點  $u_i$  與  $v_i$  間沒有邊，沒被給定的點對則表示兩點間有邊。

問圖上連通塊數量與各塊大小。

$1 \leq n \leq 2 \times 10^5, 0 \leq m \leq \min(\frac{n \times (n-1)}{2}, 2 \times 10^5)$

### Magic Matrix

Codeforces 632F

給定一個  $n \times n$  個二維陣列  $a$ 。

問  $a$  是否滿足這三個條件：

1. 對於所有的  $1 \leq i, j \leq n$ ， $a_{ij} = a_{ji}$
2. 對於所有的  $1 \leq i \leq n$ ， $a_{ii} = 0$
3. 對於所有的  $1 \leq i, j, k \leq n$ ， $a_{ij} \leq \max(a_{ik}, a_{jk})$

$1 \leq n \leq 2500$

### Axel and Marston in Bitland\*

Codeforces 781D

給定一張  $n$  個點  $m$  條邊的有向圖，圖上的邊權非 0 即 1。

現有一個字串  $s$ ，初始為”0”，接著（持續）進行以下流程：

- (1) 令  $s'$  為將  $s$  中所有字元取反形成的字串（'0' 轉為'1'；'1' 轉為'0'）
- (2) 將  $s'$  接到  $s$  後
- (3) 回到 (1)

即  $s$  的變化依序為”0”→”01”→”0110”→”01101001”→.....（最終  $s$  變為無限長）

現從點 1 出發任意走，須滿足第  $i$  次經過的邊其邊權為  $s_i$ ，問最長經過幾條邊。

若經過的邊數超過  $10^{18}$ ，輸出”-1”。

$1 \leq n \leq 500, 0 \leq m \leq 2n^2$

## 13.2 其他補充技巧

本章會補充營隊其他課程沒提到但同樣重要的內容。

### 13.2.1 hash

Hash 可以將一筆資料（或字串）轉化成一個整數（同個資料轉化出來的整數是一樣的），這樣就可以快速比較兩筆資料是否相等了。而因轉化後的整數通常是有範圍限制的，但轉換前的資料的可能性往往非常多，因此**有可能發生兩筆相異的資料 hash 出來的數字是相等的**，這個情況被稱作碰撞，所以透過良好的 hash 方法盡可能避免碰撞就會顯得很重要。本節重心會放在 hash 的教學與應用，並不會提及碰撞機率等的數學證明。

首先介紹賽場上最常見的 hash 方法：rolling hash。

使用 rolling hash 時需要先決定好兩個相異質數  $p, q(p < q)$ ，可以隨機挑選或程式直接寫死，接著便會都使用這兩個質數來 hash 資料。若現在要對一筆大小為  $n$  的有序資料  $a_0, a_1 \dots a_{n-1}$  做 hash，其 hash 值

$$h(a, p, q) = (\sum_{i=0}^{n-1} a_i \times p^{n-1-i}) \pmod{q} = (a_0 \times p^{n-1} + a_1 \times p^{n-2} + \dots + a_{n-1} \times p^0) \pmod{q}$$

寫成 code 就是：

```
1 int h(vector<int> a, int p, int q) { // 將 a 整個資料 hash
2     int ret = 0;
3     for(int val : a)
4         ret = ((long long)ret * p + val) % q;
5     return ret;
6 }
```

**程式碼 13.15:** 將一筆資料 hash

於是在求出 hash 值後，便可以快速判斷兩筆資料是否相等了，將一個大小為  $n$  的資料 hash 的時間複雜度為  $\mathcal{O}(n)$ 。

| 字符串比較  | 經典問題 |
|--|------|
| 給定 $N$ 個字串 $s_1, s_2 \dots s_N$ ，請 $Q$ 次比較其中兩個字串是否相同。<br>$1 \leq N \leq 2000, 1 \leq  s_i  \leq N, 1 \leq Q \leq 10^6$ |      |

很輕易就可以寫下樸素的暴力比較法。

```

1 const int maxn = 2005;
2 string s[maxn]; // 紀錄給定的字串
3 void solve(int N,int Q) {
4     while(Q--) {
5         int a,b;
6         cin >> a >> b;
7         if(s[a] == s[b])
8             cout << "Yes" << '\n';
9         else
10            cout << "No" << '\n';
11    }
12 }

```

**程式碼 13.16:** 樸素暴力比較兩個字串是否相等

時間複雜度為  $\mathcal{O}(N^2 + QN)$  ( 輸入要  $\mathcal{O}(N^2)$  ) · 必然超時。下面使用 hash 加速。

```

1 const int maxn = 2005;
2 string s[maxn];
3 int H[maxn]; // 紀錄每個字串的 hash 值
4 int h(int idx,int p,int q) {
5     int ret = 0;
6     for(char c : s[idx])
7         ret = ((long long)ret * p + (int)c) % q;
8     return ret;
9 }
10 void solve(int N,int Q) {
11     const int p = 107 , q = 1000000007;
12     for(int i=1;i<=N;i++) // 先對每個字串 hash 並紀錄下來
13         H[i] = h(i,p,q);
14     while(Q--) {
15         int a,b;
16         cin >> a >> b;
17         if(H[a] == H[b])
18             cout << "Yes" << '\n';
19         else
20             cout << "No" << '\n';
21    }
22 }

```

**程式碼 13.17:** hash 比較兩個字串是否相等

對一個字串 hash 需要  $\mathcal{O}(N)$  · 共有  $N$  個字串；每次比對兩個字串的 hash 值需要  $\mathcal{O}(1)$  · 共有  $Q$  次比對，因此總時間複雜度為  $\mathcal{O}(N^2 + Q)$  。

可見 rolling hash 在多次比對時很有優勢（當然若只需比較一次直接樸素比較即可）。而 rolling hash 還可以透過前綴和的方法快速求得一個區間 hash 值。

現對於大小為  $n$  的有序資料  $a_0, a_1, \dots, a_{n-1}$ ，對其每個前綴  $[0, i]$  紀錄其 hash 值，即  $H[i] = (\sum_{j=0}^i a_j \times p^{i-j}) \pmod{q} = (H[i-1] \times p + a_i) \pmod{q}$ ；於是便可透過這求出任意區間  $[l, r]$  的 hash 值

$$h(a, p, q, l, r) = \begin{cases} H[r] & l = 0 \\ (H[r] - H[l-1] \times p^{r-l+1}) \pmod{q} & l \neq 0 \end{cases}$$

例如求  $[2, 3]$  的 hash 值。首先  $[0, 3]$  的 hash 值  $H[3] = (a_0p^3 + a_1p^2 + a_2p + a_3) \pmod{q}$ ，而  $[0, 1]$  的 hash 值  $H[1] = (a_0p + a_1) \pmod{q}$ 。因此  $[2, 3]$  的 hash 值  $= (H[3] - H[2-1] \times p^{3-2+1}) \pmod{q} = (a_2p + a_3) \pmod{q}$ 。

實作上除了紀錄各個前綴的 hash，也預先計算並紀錄  $p$  的各個冪次。

```

1 const int p = 107, q = 1000000007;
2 int p_table[maxn]; // p_table[i] : p^i
3 int arr[maxn];
4 int H[maxn]; // H[i] : [0, i] 的 hash 值
5 void build(int n) { // 對所有前綴計算 hash 值、維護 p 的冪次表
6     p_table[0] = 1;
7     H[0] = arr[0];
8     for(int i=1;i<n;i++) {
9         p_table[i] = ((long long)p_table[i - 1] * p) % q;
10        H[i] = ((long long)H[i - 1] * p + arr[i]) % q;
11    }
12 }
13 int query(int l, int r) { // 查詢 [l, r] 的 hash 值
14     if(l == 0) return H[r];
15     int ret = (H[r] - (long long)H[l - 1] * p_table[r - l + 1]) % q;
16     ret = (ret + q) % q; // 上一行運算完可能有負數
17     return ret;
18 }
```

**程式碼 13.18:** 對所有前綴紀錄其 hash 值、查詢區間 hash 值

計算各前綴的 hash 值與  $p$  的冪次表需要  $\mathcal{O}(n)$ 。查詢區間 hash 值則為  $\mathcal{O}(1)$ 。

給定一個長度為  $N$  的字串  $S$ 。要求從前綴、後綴以外的位置找到一個最長的子字串滿足，其同時為  $S$  的前綴與後綴。無解輸出”Just a legend”。

$1 \leq N \leq 10^6$

首先能用 hash 求出有哪些前綴與後綴是相同的，並紀錄下來列為候選可能。

而候選可能的字串是否有在前後綴以外的位置出現，在字串長度上有單調性，意即若長度  $L$  的候選可能有在前後綴以外的位置出現，則長度小於  $L$  的候選可能也會有出現；而若長度  $L$  的候選可能不在任何前後綴以外的位置出現，那長度大於  $L$  的候選可能必然不會出現。因此可以對這些候選可能二分搜來求答案。同樣也利用 hash 去檢查有沒有在前後綴以外的位置出現當前二分搜的候選可能字串。

```

1 const int maxn = 1000005;
2 const int p = 107, q = 1000000007;
3
4 int p_table[maxn];
5 string s; // 題目給定的字串
6 int H[maxn];
7
8 void build(int n) {
9     p_table[0] = 1;
10    H[0] = (int)s[0];
11    for(int i=1;i<n;i++) {
12        p_table[i] = ((long long)p_table[i - 1] * p) % q;
13        H[i] = ((long long)H[i - 1] * p + (int)s[i]) % q;
14    }
15 }
16
17 int query(int l,int r) {
18     if(l == 0) return H[r];
19     int ret = (H[r] - (long long)H[l - 1] * p_table[r - l + 1]) % q;
20     ret = (ret + q) % q;
21     return ret;
22 }
23
24 bool check(int mid,int n) { // 檢查是否存在前綴後綴以外的目標字串
25     int target = query(0,mid - 1);
26     for(int i=1;i + mid - 1 < n - 1;i++)
27         if(query(i,i + mid - 1) == target)
28             return true;
29     return false;
30 }
31
32 void solve() {
33     cin >> s
34
35     int n = (int)s.size();

```

```

36     build(n);
37
38     vector<int> candidate; // 候選答案
39     for(int i=1;i<=n;i++)
40         if(query(0, i - 1) == query(n - i, n - 1)) //長度 i 的前綴與後綴相等
41             candidate.push_back(i);
42
43     string ans = "Just a legend";
44     if(!candidate.empty()) {
45         int l = -1 , r = (int)candidate.size();
46         while(l + 1 < r) { // 二分搜答案
47             int mid = (l + r) >> 1;
48             if(check(candidate[mid],n))
49                 l = mid;
50             else
51                 r = mid;
52         }
53         if(l >= 0) ans = s.substr(0,candidate[l]);
54     }
55     cout << ans << '\n';
56 }
```

**程式碼 13.19:** hash 解 Password

計算各前綴 hash 值一樣是  $\mathcal{O}(N)$ ；比對兩個區間 hash 值也一樣是  $\mathcal{O}(1)$ ；枚舉候選解需要  $\mathcal{O}(N)$ ；二分搜複雜度是  $\mathcal{O}(\log N)$ ，檢查是否存在前綴後綴以外的目標子字串為  $\mathcal{O}(N)$ ，因此總時間複雜度為  $\mathcal{O}(N + N + N \log N) = \mathcal{O}(N \log N)$ 。

而除了對一個序列或字串 hash，有序集合也是可以透過 hash 來比較的，例如給定大數的質因數分解，可以將其每個質數與冪次 hash 來比對兩個大數是否相等（注意 hash 的順序，記得保證相同數 hash 出來結果一樣）。

另外有時為了降低碰撞機率，可以使用兩個不同的 hash 函數（甚至更多！但要注意常數必然會變大），只有當兩筆資料兩種 hash 出來的值都一樣才將它們視為是相等的資料（注意：不是先透過一個函數 hash 後，再將 hash 出來的值丟入另一個 hash 函數，這樣碰撞機率反而會提高！）。

至於 rolling hash 裡  $p, q$  的挑選同時也是一門學問，除了影響碰撞機率外，有些出題者還會卡掉比較常見的組合，而範例 code 裡的 107 與  $10^9 + 7$  可能就不是個很好的  $p, q$  組合（即便大部分情況下筆者這個組合便夠用了）。這篇 Codeforces 的文章（<https://codeforces.com/blog/entry/60442>）有詳盡地提到如何盡可能避免碰撞，有興趣的同學可以研究看看。

### 13.2.2 分塊

在前面的章節中，已經學到了能用來處理序列操作的線段樹與 Treap，本節會再介紹一個同樣能處理序列操作但較為暴力的方法：分塊。它能夠解決一些線段樹與 Treap 難以處理的題目。

分塊的核心想法很簡單，便是將一個長度為  $N$  的序列切成連續（幾乎）均等的數個塊，將每個塊的大小都盡量設為一個我們設定的  $K$ 。接著會在每個塊裡維護需要的資訊，在修改操作時直接更新單個塊的資訊，詢問時則將多個塊維護的資訊合併來獲得答案。一樣看道例題。



圖 13.4: 將大小 11 的序列，每 3 個元素分成一塊，共 4 塊

| 區間最值問題                            | 經典問題 |
|-----------------------------------|------|
| 給定一個長度為 $N$ 的正整數序列，試 $Q$ 次支援兩種操作： |      |
| 1. 修改序列某個元素的值                     |      |
| 2. 查詢序列某個區間的最大值                   |      |
| $1 \leq N, Q \leq 10^5$           |      |

經典的 RMQ 問題，使用線段樹可以在  $\mathcal{O}(N + Q \log N)$  內解決。我們來看如何用分塊解這題。

將序列每  $K$  個元素分成一塊，共有  $\frac{N}{K}$  個塊，對於每個塊維護這個塊裡所有元素的最大值。修改（操作一）時更新該元素所在塊的資訊；詢問（操作二）時，將區間完全涵蓋的所有塊的資訊合併（取最大值），並對於那些在區間內但其所在的塊沒被完全涵蓋的元素，逐個合併。



圖 13.5: 合併塊時，有些元素其所在塊未被詢問區間完全涵蓋，因此要逐個合併

實作上，塊的分界與逐個合併時的邊界要謹慎實作。

```
1 int N,Q;      // 題目給定的 N, Q
2 int arr[100005];    // 題目給定的序列
3 int block_size;    // 每個塊的大小
4 int max_value[330]; // 維護每個塊的最大值
5 void build(int pos) { // 更新第 pos 個元素所在塊的資訊
6     int idx = pos / block_size;
7     int l = max(1, idx * block_size);           // 該塊對應的左邊界
8     int r = min(N, (idx + 1) * block_size - 1); // 該塊對應的右邊界
9     max_value[idx] = 0;
10    for(int i=l;i<=r;i++)
11        max_value[idx] = max(max_value[idx], arr[i]);
12 }
13 void modify(int pos,int new_val) { // 將第 pos 個元素修改成 new_val
14     arr[pos] = new_val;
15     build(pos);
16 }
17 int query(int l,int r) { // 查詢 [l,r] 的最大值
18     int ret = 0;
19     int l_idx = l / block_size; // 左邊界所在塊的編號
20     int r_idx = r / block_size; // 右邊界所在塊的編號
21     for(int i=l_idx + 1;i < r_idx;i++) // 合併完全涵蓋的塊的各最大值
22         ret = max(ret, max_value[i]);
23
24     // 左邊界所在塊往右的邊界
25     int l_bound = min((l_idx + 1) * block_size - 1, r);
26     for(int i=l;i<=l_bound;i++) // 逐個合併
27         ret = max(ret, arr[i]);
28     // 右邊界所在塊往左的邊界
29     int r_bound = max(r_idx * block_size, l);
30     for(int i=r;i>=r_bound;i--) // 逐個合併
31         ret = max(ret, arr[i]);
32     return ret;
33 }
34 void init(int N,int K) { // 設定塊大小與對每個塊初始化
35     block_size = K; // 令每塊大小為 K
36     for(int i=1;i<=N;) {
37         build(i);
38         i = (i / block_size + 1) * block_size;
39     }
40 }
```

程式碼 13.20：分塊解區間最值問題

再來分析時間複雜度：對每個塊計算該塊內元素的最大值需要  $\mathcal{O}(K)$ ；一開始會對每個塊維護其最大值，而共有  $\frac{N}{K}$  個塊；每次修改時會重新計算某個塊的最大值，最多  $Q$  次修改；每次詢問時最多將  $\frac{N}{K}$  個塊合併，而最多逐項合併  $2 \times K$  個元素，最多有  $Q$  次詢問。總時間複雜度為  $\mathcal{O}(\frac{N}{K} \times K + QK + Q \times (\frac{N}{K} + 2 \times K)) = \mathcal{O}(N + QK + Q \times \frac{N}{K} + QK)$ 。

若令  $K = \sqrt{N}$ ，則總時間複雜度為  $\mathcal{O}(N + Q\sqrt{N} + Q \times \frac{N}{\sqrt{N}} + Q\sqrt{N}) = \mathcal{O}(N + Q\sqrt{N})$ 。

雖然時間複雜度比線段樹作法還差，但分塊的強處在於塊在維護資訊上更加彈性，特別是修改時難以（快速）更新的類型，這部分是線段樹所不及的。

接著來看道較難處理的題目。

| Holes  | Codeforces 13E |
|--|----------------|
| 現有 $N$ 個傳送門，分別依序在座標 $1, 2, \dots, N$ 上。  |                |
| 如果在座標 $i$ 上放下一顆球，球會傳送至座標 $i + a_i$ 上，然後會再傳送至座標 $(i + a_i) + a_{(i+a_i)}$ ，接著一直持續傳送直到所在座標沒有傳送門。 |                |
| 試 $M$ 次支援兩種操作：   |                |
| 1. 修改 $a_i$  |                |
| 2. 在某個座標放下一顆球，輸出最後一次傳送前的座標及傳送次數  |                |
| $1 \leq N, M \leq 10^5, 1 \leq a_i \leq N$   |                |

看起來不論使用線段樹或 Treap 都無法輕易解決，現在考慮使用分塊。

將傳送門每  $\sqrt{N}$  個分成一塊，共有  $\sqrt{N}$  個塊。對每個傳送門維護如果在其座標上放下一顆球，球在塊內最後停留的座標與及傳送次數。操作一時，更新塊內各傳送門最後停留座標。操作二時，透過塊內維護的資訊加速球的傳送。

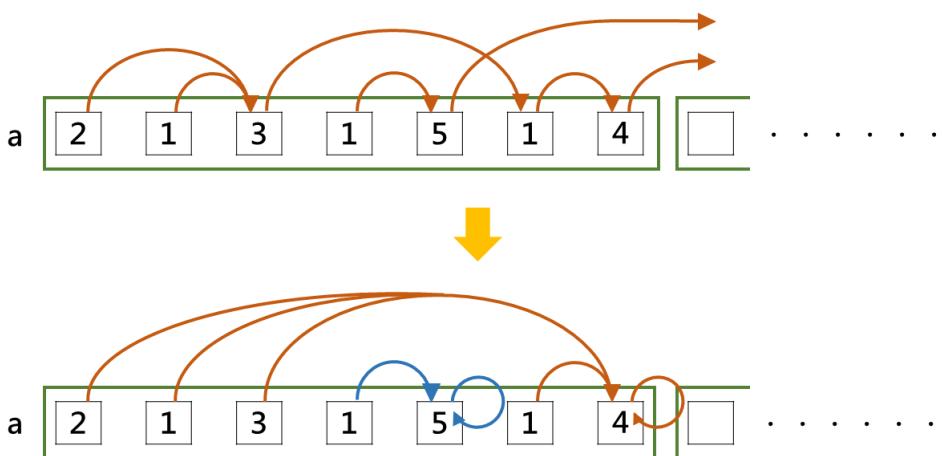


圖 13.6：一個大小為 7 的塊；維護在各傳送門上放球時，球最後在該塊內停留的座標

```

1  const int maxn = 100005;
2  int N,M;
3  int arr[maxn];
4  struct state { // 每個傳送門傳送球最後停留在塊內的座標與傳送次數
5      int ed,step;
6  }table[maxn];
7  void build(int l,int r) {
8      for(int i=r;i>=l;i--) {
9          int nxt = i + arr[i]; // 球在座標 i 放下後經過一次傳送的下個座標
10         if(nxt > r) // 傳送一次就會離開塊，故 i 就是最後停留的座標
11             table[i] = {i, 0};
12         else // 從下個座標的狀態轉移
13             table[i] = {table[nxt].ed, table[nxt].step + 1};
14     }
15 }
16
17 int block_size;
18 void init() { // 初始化
19     block_size = sqrt(N);
20     for(int i=1;i<=N;) { // 初始對每個傳送門維護最後停留的座標與傳送次數
21         int idx = i / block_size; // 現在維護第 idx 個塊
22         int l = max(1, idx * block_size); // 該塊對應的左邊界
23         int r = min(N, (idx + 1) * block_size - 1); // 該塊對應的右邊界
24         build(l,r);
25
26         i = r + 1;
27     }
28 }
29
30 void modify(int a,int b) { // 操作一，修改某個位置的值
31     arr[a] = b;
32     int idx = a / block_size;
33     int l = max(1, idx * block_size);
34     int r = min(N, (idx + 1) * block_size - 1);
35     build(l,r); // 更新整塊的資訊
36 }
37
38 void query(int a) { // 操作二，問在某座標投入球最後停留的座標與次數
39     int step = 0;
40     while(true) { // 持續傳送直到座標超過 N
41         // 跳至塊內最後停留的座標
42         step += table[a].step;
43         a = table[a].ed;
44         if(a + arr[a] > N) break;
45         // 跳至下一個塊內
46         step++;
47         a += arr[a];
48     }
49     cout << a << " " << step + 1 << '\n';
50 }
51

```

```

52 void solve() {
53     init();
54     while(M--) {
55         int type;
56         cin >> type;
57         if(type == 0) {
58             int a,b;
59             cin >> a >> b;
60             modify(a,b);
61         }
62         else {
63             int a;
64             cin >> a;
65             query(a);
66         }
67     }
68 }
```

**程式碼 13.21:** 分塊解 Holes

最後分析時間複雜度：初始會對每個傳送門維護需要  $\mathcal{O}(N)$ ；修改時會更新整個塊的資訊，花費  $\mathcal{O}(\sqrt{N})$ ，最多進行  $M$  次修改；詢問時傳送最多經過  $\sqrt{N}$  個塊，最多有  $M$  次詢問。故總時間複雜度為  $\mathcal{O}(N + M\sqrt{N} + M\sqrt{N}) = \mathcal{O}(N + M\sqrt{N})$ 。

另外有時候塊的大小不一定設為  $\sqrt{N}$  時間複雜度就會最好，某些情況設為別的大小可能會比較好（例如  $\sqrt{N \log N}$ ），同學要根據題目與範圍隨機應變。

### 13.2.3 莫隊

莫隊算法是個離線演算法，它基於分塊概念，**依照特定順序去處理區間操作**，使得即便解決問題的方式乍看有些暴力，但其仍保有優美的時間複雜度。

區間問題使用莫隊必須滿足一個核心條件：若現已處理好區間  $[l, r]$  的答案，則必須能夠快速求解  $[l - 1, r], [l + 1, r], [l, r - 1], [l, r + 1]$  的答案。

在滿足上述條件情況下，對序列分塊後，莫隊會依照操作**左邊界所在的塊編號**來將操作排序，若**塊編號相等**則依照操作**右邊界排序**。然後依照排好的順序逐個處理操作，不同操作間重疊部分的資訊會被保留，沒重疊的部分則暴力添加/刪除。

下面透過例題具體說明如何使用莫隊算法。

給定一個大小為  $N$  的序列  $a$ ，試  $Q$  次回答：給定一個區間  $[l, r]$ ，將區間內每種數字與其出現次數平方的乘積加總後輸出。

$1 \leq N, Q \leq 2 \times 10^5, 1 \leq a_i \leq 10^6$

因為不會依照題目給定的順序去處理詢問，故需要對每筆詢問紀錄其是第幾筆詢問。同時對每個詢問紀錄其左邊界位於哪個塊內。另依照莫隊規則定義詢問間的排序規則：先依照詢問左邊界所在的塊編號排序，否則依照詢問的右邊界排序。

```

1 struct que {
2     int l,r,idx,block_idx; // idx : 詢問的編號 ; block_idx : 左邊界所在的塊
3     bool operator < (const que &a) const{ // 詢問間的排序規則
4         return block_idx != a.block_idx ? block_idx < a.block_idx : r < a.r;
5     }
6 }ques[maxn];

```

程式碼 13.22: Powerful array 詢問的結構定義

將序列每  $\sqrt{N}$  個分成一塊。若  $N = 11$  時，依序給定三個詢問： $[5, 11], [2, 8], [6, 10]$ ，則序列元素每  $\sqrt{11} \approx 3$  個分成一塊，詢問在排序後會像圖13.7 (順序由上往下)：

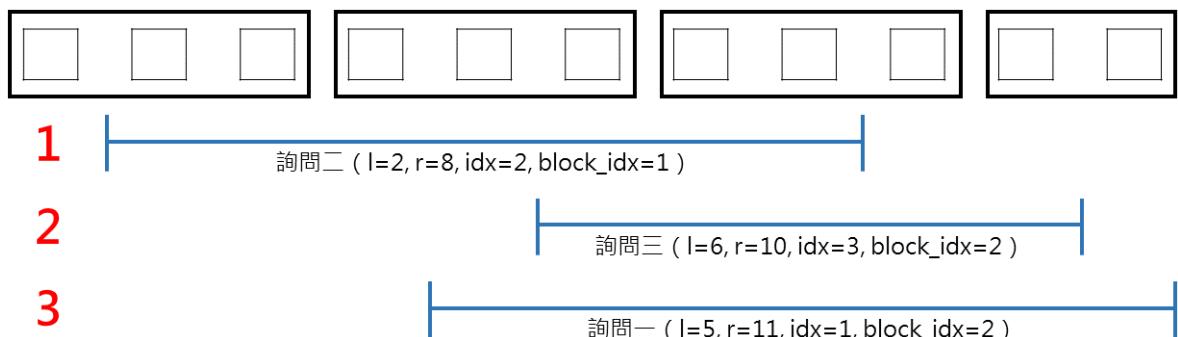


圖 13.7: 詢問二因為左邊界所在的塊的編號最小故排最前面；  
詢問三與詢問一的左邊界皆在同塊，但詢問三右邊界較小故排詢問一前面

將詢問排序好後，就只剩下計算答案了。使用兩個指針 (two pointer) 來維護當前的區間：如果處理完了詢問  $a$ ，則兩指針就會對應詢問  $a$  的左右邊界，若接著要處理詢問  $b$ ，則移動這兩個指針使其改為其對應詢問  $b$  的左右邊界 (移動指針的同時更新資訊)。

舉例來說，若兩個指針現在對應的區間為  $[5, 10]$ ，而接下來要處理的詢問其區間為  $[8, 13]$ ，則在移動兩指針的同時，將區間  $[11, 13]$  的資訊加入，並移除區間  $[5, 7]$  的資訊。

```

1 int arr[maxn]; // 題目給定的原序列
2 int cnt[1000005]; // cnt[i] : i 當前的出現次數
3 long long sum = 0; // 目前各種數字與其出現次數平方乘積的總和
4 void update(int val, int delta) { // 處理左右邊界移動 (更新某值出現次數)
5     sum -= (long long)val * cnt[val] * cnt[val];
6     cnt[val] += delta;
7     sum += (long long)val * cnt[val] * cnt[val];
8 }
9 long long ans[maxn]; // 對每筆詢問紀錄其答案
10 void solve(int n, int q) {
11     sort(ques+1, ques+1+q); // 將詢問排序 (先依左邊界所在塊，再依右邊界)
12     int l = 1, r = 0; // 兩個指針維護當前的區間
13     for(int i=1; i<=q; i++) {
14         que now = ques[i];
15         // 移動兩指針，更新與前次詢問不重疊的部分
16         while(r < now.r) update(arr[r + 1], 1), r++;
17         while(r > now.r) update(arr[r], -1), r--;
18         while(l < now.l) update(arr[l], -1), l++;
19         while(l > now.l) update(arr[l - 1], 1), l--;
20         ans[now.idx] = sum;
21     }
22     for(int i=1; i<=q; i++)
23         cout << ans[i] << '\n';
24 }

```

**程式碼 13.23:** 莫隊解 Powerful array

最後分析時間複雜度：將詢問排序的部分是  $\mathcal{O}(Q \log Q)$ ；兩個指針移動一次（code 裡的 update 函式）的複雜度為  $\mathcal{O}(1)$ ，接著對左右指針分開考慮複雜度。

對於左指針，如果前次詢問是同塊的，則其最多移動  $\sqrt{N}$  步，總共有  $Q$  筆詢問；而如果是不同塊的，則最多移動  $N$  步，但橫跨不同塊的情況只會發生  $\frac{N}{\sqrt{N}}$  次<sup>i</sup>。因此左指針移動的時間複雜度為  $\mathcal{O}(Q\sqrt{N} + \frac{N}{\sqrt{N}}N) = \mathcal{O}((N + Q)\sqrt{N})$ 。

對於右指針，因為同塊內詢問照右邊界排序，因此同塊內右指針最多移動  $N$  次，而共有  $\frac{N}{\sqrt{N}}$  個塊；橫跨兩相異塊部分則與左指針一樣。因此右指針移動的時間複雜度為  $\mathcal{O}(\frac{N}{\sqrt{N}}N + \frac{N}{\sqrt{N}}N) = \mathcal{O}(N\sqrt{N})$ 。

總時間複雜度為  $\mathcal{O}(Q \log Q + Q\sqrt{N} + N\sqrt{N} + N\sqrt{N}) = \mathcal{O}(Q \log Q + (N + Q)\sqrt{N})$ 。

在實作上，建議先移動右指針再移動左指針，否則可能會出現右指針在左指針左邊的情況，這也許會導致答案的計算出現問題。

<sup>i</sup>事實上，可以證明相異塊移動的總複雜度為  $\mathcal{O}(N)$ ，但這不影響整體總複雜度

另外莫隊與分塊一樣，塊的大小不一定要設為  $\sqrt{N}$ 。而在莫隊的進階變化上，還有帶修改莫隊與樹上莫隊等，有興趣的同學可以自行研究。

### 13.2.4 根號想法

除了分塊與莫隊這種（通常）基於以根號個元素為單位切割序列的算法，有的題目解法同樣也圍繞著根號。這種技巧稱為平方分解（square root decomposition），中文有時也會有人稱之為分塊。

| Modulo Data Structures   | Kattis |
|--|--------|
| 給定大小為 $N$ 的序列 $a$ ，序列各元素初始值皆為 0。請 $Q$ 次支援兩種操作：<br>1. 對所有的 $i$ 滿足 $i \equiv A \pmod{B}$ ，將 $a_i$ 加上 $C$<br>2. 詢問 $a_D$<br>$1 \leq N, Q \leq 2 \times 10^5, 0 \leq A < B \leq N$ |        |

操作一因為難以預測要修改的元素，所以不易快速修改需要修改的元素。

而操作一給定的  $B$  對於修改的元素數量影響最為關鍵，那麼現在將其以  $\sqrt{N}$  為界分成兩種情況分析它們所有的性質：

- $B \leq \sqrt{N}$ ： $B$  只會有  $\sqrt{N}$  種可能，而每個  $B$  對應的  $A$  最多也只有  $\sqrt{N}$  種
- $B > \sqrt{N}$ ：修改的元素數量最多為  $\frac{N}{B}$  個，即修改的元素數量最多為  $\sqrt{N}$  個

現在得知了在兩種情況下  $B$  的可能數與修改元素數量都被限制在  $\sqrt{N}$  以內，接著我們使用對兩種情況分別採用不同的方式來修改元素：

- $B \leq \sqrt{N}$ ：因為  $B$  的可能數不多，且  $A < B$ ，因此使用一個二維陣列  $add[i][j]$  去維護當前所有  $A = i, B = j$  的操作一增加值的總和；在操作二詢問  $a_D$  的值時，再去遍歷這個二維陣列來計算正確的  $a_D$
- $B > \sqrt{N}$ ：因為修改的元素數量不多，所以直接更新序列元素即可

```

1 const int maxn = 200005;
2 int arr[maxn], add[450][450];
3 void solve(int N,int Q) {
4     int sqrt_N = sqrt(N);
5     while(Q--) {
6         int type;
7         cin >> type;
8         if(type == 1) {
9             int A,B,C;
10            cin >> A >> B >> C;
11            if(B <= sqrt_N)
12                add[A][B] += C;
13            else {
14                for(int i=A;i<=N;i+=B)
15                    arr[i] += C;
16            }
17        }
18        else {
19            int D;
20            cin >> D;
21            int ans = arr[D];
22            for(int i=1;i<=sqrt_N;i++)
23                ans += add[D % i][i];
24            cout << ans << '\n';
25        }
26    }
27 }

```

**程式碼 13.24:** 平方分解解 Modulo Data Structures

最後分析時間複雜度：操作一時，若  $B \leq \sqrt{N}$ ，則修改只需要  $\mathcal{O}(1)$ ，而  $B > \sqrt{N}$  時，最多修改  $\sqrt{N}$  個元素，最多有  $Q$  筆操作一；操作二時，遍歷  $add$  陣列計算答案需要  $\mathcal{O}(\sqrt{N})$ ，最多有  $Q$  筆操作二。因此總時間複雜度為  $\mathcal{O}(Q\sqrt{N} + Q\sqrt{N}) = \mathcal{O}(Q\sqrt{N})$ 。

### 13.2.5 練習題 Exercise

題目名稱後標註「\*」的題目為筆者認為較為困難的題目。

| Watto and Mechanism   | Codeforces 514C |
|---|-----------------|
| <p>給定 <math>n</math> 字串 <math>t_i</math>。</p> <p>試支援 <math>m</math> 次詢問：給定一個字串 <math>s_i</math>，問是否存在一個 <math>j</math> 使得 <math> s_i  =  t_j </math> 且 <math>s_i</math> 與 <math>t_j</math> 有恰好一個字元（位置）相異。</p> <p>所有的字串都只會由'a', 'b', 'c' 所組成。</p> <p><math>0 \leq n, m \leq 3 \times 10^5, (\sum_{i=1}^n  t_i ) + (\sum_{i=1}^m  s_i ) \leq 6 \times 10^5</math></p> |                 |

|   |                 |
|---|-----------------|
| Permutation*  | Codeforces 452F |
| 給定一個 1 到 $n$ 的排列。問排列中是否存在兩相異元素 $a, b$ ，使得 $\frac{(a+b)}{2}$ 的位置出現在它們之間。注意 $\frac{(a+b)}{2}$ 不用取整，即它可能為小數。 |                 |
| $1 \leq n \leq 3 \times 10^5$   |                 |

|  |                 |
|--|-----------------|
| Anton and Permutation                                      | Codeforces 785E |
| 給定一個大小為 $n$ 的序列。   |                 |
| 請 $q$ 次支援操作：將序列中兩個元素交換，並於交換後輸出當前的逆序數對數。                    |                 |
| $1 \leq n \leq 2 \times 10^5, 1 \leq q \leq 5 \times 10^4$ |                 |

|   |                 |
|---|-----------------|
| DZY Loves Fibonacci Numbers*  | Codeforces 446C |
| 給定一個大小為 $n$ 的序列 $a$ 。   |                 |
| 試 $m$ 次支援兩種操作：  |                 |
| <ol style="list-style-type: none"> <li>1. 對於所有的 <math>j</math> 在區間 <math>[l_i, r_i]</math>，將 <math>a_j</math> 加上 <math>F_{j-l_i+1}</math>。<math>F</math> 為費式數列</li> <li>2. 輸出 <math>(\sum_{j=l_i}^{r_i} a_j) \bmod (10^9 + 9)</math></li> </ol> |                 |
| $1 \leq n, m \leq 3 \times 10^5$  |                 |

|  |                 |
|--|-----------------|
| XOR and Favorite Number  | Codeforces 617E |
| 給定一個大小為 $n$ 個序列 $a$ ，及一個整數 $k$ 。   |                 |
| 回答 $m$ 次詢問，每次詢問給定 $l_i$ 與 $r_i$ ，輸出區間 $[l_i, r_i]$ 中有多少子區間滿足將 $a$ 中所有在該子區間的元素 bitwise XOR 後與 $k$ 相等。 |                 |
| $1 \leq n, m \leq 10^5, 0 \leq a_i, k \leq 10^6$   |                 |

|  |                  |
|--|------------------|
| Ehab's REAL Number Theory Problem*         | Codeforces 1325E |
| 給定 $n$ 個元素，保證每個元素 $a_i$ 最多都只有 7 個因數。       |                  |
| 試從中挑出盡量少的元素，使得挑出的元素乘積為完全平方數（可能無解）。         |                  |
| $1 \leq n \leq 10^5, 1 \leq a_i \leq 10^6$ |                  |

## 13.3 經典問題選講

本章節會分享比賽中經典或有趣的題目，這之中有些解法特別所以經典，有些則是解法的核心想法很重要，它們可能是前面課程所學的經典應用或變形。

### 13.3.1 約瑟夫問題

| 約瑟夫問題  | 經典問題 |
|--|------|
| 有 $N$ 個人，編號分別為 0 到 $N - 1$ ，還有一個正整數 $K$ 。現在 $N$ 個人圍成一圈，編號差 1 的人位置相鄰（編號 0 與編號 $N - 1$ 相鄰），接著從編號 0 的人開始輪流報數（編號 0 報完數換編號 1，以此類推），報到 $k$ 的那個人淘汰，然後從淘汰的人之後的下一個人重新開始報數，並在沒被淘汰的人裡持續繼續進行報數與淘汰；直到剩下一個人時，那個人獲勝。問獲勝的人編號為多少。 |      |
| $1 \leq N \leq 10^5, 1 \leq K \leq N$  |      |

同學應該不難想到時間複雜度  $\mathcal{O}(N^2K)$ 、 $\mathcal{O}(N^2)$  或  $\mathcal{O}(NK)$  的解法，但這三種解法的複雜度顯然在本題的最糟情況下都會超時。

而解題關鍵明顯在於要能快速找出這輪要淘汰的人並將他淘汰，然而陣列（或是 std::vector）和 linked-list（或是 std::list）這兩種資料結構分別無法快速淘汰人與快速找出要淘汰的人，有沒有其他資料結構能夠快速地完成這兩件事嗎？（同學可以回顧資料結構章節學到的技巧，看看有沒有哪種資料結構能夠符合所需）

下面分享一個使用資料結構的解法：要維護還沒淘汰的人，同時快速找出某人開始數  $K$  個人後的人是誰，以及將其刪去，那不妨考慮使用 Treap（忘記的同學或是不會的同學可以翻回進階資料結構的章節看看）。用 Treap 維護現在還沒淘汰的人，至於求出淘汰的人，只要維護好每輪要開始報數的人是當前沒被淘汰的人之中編號第幾小的人，那就可以輕易得知要淘汰的人是當前這輪編號第幾小的人；而淘汰就只要把被淘汰的人從 Treap 裡移除即可。

```
1 /*  
2 部分與進階資料結構章節的 Treap code 幾乎雷同的 code 省略，僅文字說明  
3  
4 Treap 裡 idx 紀錄各節點對應的人的編號  
5  
6 Treap* merge(Treap *a, Treap *b) : 回傳兩棵 Treap a 與 b 合併後的樹根  
7  
8 void split(Treap *root, Treap *&a, Treap *&b, int k) : 將 a 指向 root 前 k 個  
   節點形成的 Treap 的樹根；將 b 指向剩下的節點形成的 Treap 的樹根  
9  
10 int sz(Treap *p) : 回傳 p 指向的點為根的 Treap 大小  
11 */  
12
```

```

13 void remove(Treap *&root, int pos) { // 刪除 Treap 裡第 pos 個節點
14     Treap *l,*r;
15     split(root,root,r,pos);
16     split(root,l,root,pos - 1);
17     root = merge(l,r);
18 }
19 int solve(int N,int K) {
20     Treap *root = nullptr;
21     for(int i=0;i<N;i++) {
22         root = merge(root, new Treap(i));
23         int pre_pos = 0; // 每輪從當前 Treap 裡第 pre_pos + 1 個節點開始報數
24         for(int i=1;i<N;i++) {
25             int nxt_pos = (pre_pos + (K - 1)) % sz(root);
26             remove(root, nxt_pos + 1);
27             pre_pos = (nxt_pos >= sz(root) ? 0 : nxt_pos);
28         }
29         return root->idx;
30     }

```

程式碼 13.25: Treap 解約瑟夫問題

最後分析時間複雜度：初始建 Treap 的複雜度為  $\mathcal{O}(N \log N)$ ；每輪知道要淘汰的人是當前沒被淘汰的人中第幾小的人需要  $\mathcal{O}(1)$ ，而將該人對應的節點從 Treap 裡刪去需要  $\mathcal{O}(\log N)$ ，共會進行  $N - 1$  輪淘汰。因此總時間複雜度為  $\mathcal{O}(N \log N + (N - 1) + (N - 1) \log N) = \mathcal{O}(N \log N)$ 。

有興趣的同學可以想想怎麼用線段樹解。下面分享線性時間的遞迴解法。

定義  $F(N)$  為  $N$  個人情況下，最後獲勝的人的編號，而  $F(1) = 0$ 。遞迴式：

$$F(N) = (F(N - 1) + K) \bmod N$$

遞迴式的解釋：在  $N$  個人情況下，經過一輪後，就變為求  $N - 1$  個人情況下獲勝的人編號。但  $F(N - 1)$  求得的贏家編號，是基於開始報數的人編號為 0 的條件下的，也就是「把  $N$  個人情況下經過一輪後，開始報數的那個人編號當成 0」，然而他實際編號應為  $K \bmod N$ ，因此需要將  $F(N - 1)$  求得的贏家編號平移後才會是正確的  $F(N)$ 。

假設  $N = 6, K = 5$ ，已知  $F(5) = 1$

|      |                  |                  |                  |                  |   |                  |
|------|------------------|------------------|------------------|------------------|---|------------------|
|      | 0                | 1                | 2                | 3                | 4 | 5                |
| 第一輪  | 0                | 1                | 2                | 3                | 4 | 5                |
| 重新編號 | 0 <sub>(1)</sub> | 1 <sub>(2)</sub> | 2 <sub>(3)</sub> | 3 <sub>(4)</sub> |   | 5 <sub>(0)</sub> |

圖 13.8: 第一輪後重新編號；因  $F(5) = 1$ ，故贏家是新編號為 1 的人，即原本編號 0 的人

```

1 int solve(int N,int K) {
2     int ans = 0;
3     for(int i=2;i<=N;i++)
4         ans = (ans + K) % i;
5     return ans;
6 }
```

**程式碼 13.26:** 數學解約瑟夫問題

總時間複雜度為  $\mathcal{O}(N)$ 。

### 13.3.2 關燈遊戲

| 關燈遊戲  | 經典問題 |
|---|------|
| <p>給定一個 <math>N \times M</math> 的網格圖，其中每格都放有一個燈泡，燈泡可能是開著的也可能是關著的。可以切換任意格內燈泡的狀態（開著的燈泡變關著的；關著的燈泡變開著的），但當切換了某格燈泡的狀態時，其相鄰格子（上下左右）內燈泡的狀態也會跟著被切換。</p> <p>現在要求切換最少次，使得所有燈泡都是關著的（可能無解）。</p> <p><math>1 \leq N, M \leq 15</math></p> |      |

看起來很難有個好策略直接解，如果暴力枚舉每顆燈泡要不要切換，再去判斷是否所有燈泡都是關著的，那複雜度也需要  $\mathcal{O}(2^{NM}NM)$ 。

現在考慮問題的弱化版本（這邊比較像簡化版），將原問題降一個維度變為  $1 \times M$  的網格圖（有的時候如果能解決弱化版的問題，也許解法可以推廣回原問題或是對於解決原問題獲得一些啟發<sup>i</sup>，而且通常弱化版本會比較方便思考），也就是變成在一個大小為  $M$  的序列上做關燈遊戲。熟悉 DP 的同學，應該可以推出時間複雜度  $\mathcal{O}(M)$  的 DP 方法，但因其（筆者的 DP 解法）推廣回原問題的複雜度不甚理想，且礙於篇幅就不提及了。

觀察可以發現，一個燈泡與其相鄰燈泡的切換次數總和的奇偶性是固定的（例如一個開著的燈泡，那麼它與相鄰燈泡的切換次數總和必須是奇數，這樣最後該燈泡才會是關著的），而第一顆燈泡最多只有一個相鄰燈泡（最後一顆燈泡也是），**那一旦決定了第一顆燈泡要不要切換，第二個燈泡要不要切換也會確定下來**（為了讓第一顆從開著變關著或維持關著）。類似地，第二顆確定要不要切換時，第三顆也會確定要不要切換……一直持續到最後一顆。至此已得到一個解法：對第一顆燈泡嘗試切換與不切換，再去判斷最後能不能將所有燈泡關起來並統計切換次數。

<sup>i</sup>甚至「這題我應該不會」都是種啟發（可以考慮跳題），特別是無法解原問題的明顯弱化版本時

```

1  bool state[20]; // state[i] : 第 i 顆燈泡當前狀態
2  void update(int pos) { // 切換第 pos 顆燈泡的狀態
3      for(int i=-1;i<=1;i++) // 相鄰的燈泡狀態都要切換
4          state[pos + i] = !state[pos + i];
5  }
6  int solve(int m) { // 計算最小切換次數使得所有燈關掉；無解回傳 -1
7      int ret = -1;
8      for(bool s : {false,true}) { // 枚舉第 1 顆燈泡是否要切換狀態
9          if(s) update(1); // 要切換狀態時則切換狀態
10
11         int sum = (s ? 1 : 0);
12         queue<int> undo; // 紀錄將哪些位置的燈泡狀態切換（不含第 1 顆）
13
14         // 判斷第 2 顆到第 m 顆燈泡哪些需要切換狀態
15         for(int i=2;i<=m;i++)
16             if(state[i - 1]) { // i - 1 顆燈泡開著，則第 i 顆必得切換狀態
17                 update(i);
18                 sum++;
19                 undo.push(i);
20             }
21
22         // 檢查是否所有燈都關了
23         for(int i=1;i<=m;i++)
24             if(state[i])
25                 sum = -1;
26
27         // 更新答案
28         if(sum != -1 && (ret == -1 || sum < ret)) ret = sum;
29
30         // 復原所做的操作
31         while(!undo.empty()) {
32             int now = undo.front();
33             undo.pop();
34             update(now);
35         }
36         if(s) update(1);
37     }
38     return ret;
39 }
```

程式碼 13.27：解一維關燈遊戲

枚舉第一顆方案有兩種，判斷是否有解與計算切換次數需要  $\mathcal{O}(M)$ ，因此時間複雜度為  $\mathcal{O}(M)$ 。

現在嘗試將解法推廣回原問題，與原本類似，第一列 (row) 的方案確定時，第二列為了要讓第一列開著的變為關著/關著的繼續關著，所以第二列的方案也是確定的，類似地後面幾列方案也會確定下來，那麼就只需要枚舉第一列哪些燈泡狀態要切換，再去判斷是否有解及找切換次數最小的方案即可。

```

1 int n,m;
2 bool state[maxn][maxn];
3 void update(int i,int j) { // 切換位於 (i,j) 的燈泡狀態
4     static int delta_i[] = {0,-1,0,1,0};
5     static int delta_j[] = {0,0,-1,0,1};
6     for(int t=0;t<5;t++) {
7         int now_i = i + delta_i[t];
8         int now_j = j + delta_j[t];
9         state[now_i][now_j] = !state[now_i][now_j];
10    }
11 }
12 // 遞迴枚舉第一列各個燈泡要不要切換後，嘗試將其他燈泡關上
13 int solve(int pos,int sum) { // pos：現枚舉到(1,pos)的燈泡；sum：切換次數
14     int ret = -1;
15     if(pos == m + 1) { // 第一列狀態枚舉完
16         queue<pair<int,int>> undo;
17         // 判斷第 2 列到第 n 列的燈泡哪些需要切換狀態
18         for(int i=2;i<=n;i++)
19             for(int j=1;j<=m;j++)
20                 if(state[i - 1][j]) { // (i,j) 上面的燈泡開著，故切換
21                     update(i,j);
22                     sum++;
23                     undo.push({i,j});
24                 }
25
26         // 檢查是否所有燈泡都關了
27         for(int i=1;i<=n;i++)
28             for(int j=1;j<=m;j++)
29                 if(state[i][j])
30                     sum = -1;
31     ret = sum;
32
33     // 復原所做的操作
34     while(!undo.empty()) {
35         auto now = undo.front();
36         undo.pop();
37         update(now.first, now.second);
38     }
39 }
40 else {
41     for(auto s : {false,true}) {
42         if(s) update(1,pos);
43         int now = solve(pos + 1, sum + (s ? 1 : 0));
44         if(now != -1 && (ret == -1 || now < ret)) ret = now;
45         if(s) update(1,pos); // 復原所做的操作
46     }
47 }
48 return ret;
49 }

```

**程式碼 13.28:** 解關燈遊戲

最後分析時間複雜度：枚舉第一列哪些燈泡要切換需要  $\mathcal{O}(2^M)$ ，嘗試將剩下的燈泡關掉並檢查與計算答案需要  $\mathcal{O}(NM)$ ，因此總時間複雜度為  $\mathcal{O}(2^M NM)$ 。

比賽中有些題目乍看狀態或選擇很多，但也許經過某些情況或操作後就會變少，甚至唯一。而嘗試解決弱化版本來找原問題解法的方式也不見得適合每位同學（甚至有時解決弱化版本可能沒有任何幫助），實際上還是要看個人習慣與題目。

### 13.3.3 樹上最大獨立集

| 樹上最大獨立集  | 經典問題 |
|--|------|
| 給定一棵大小為 $N$ 的樹，且根節點為 1。要求從中找到一個最大的點集合，滿足集合中任兩點在樹上沒連邊。輸出集合大小即可。 |      |
| $1 \leq N \leq 10^6$   |      |

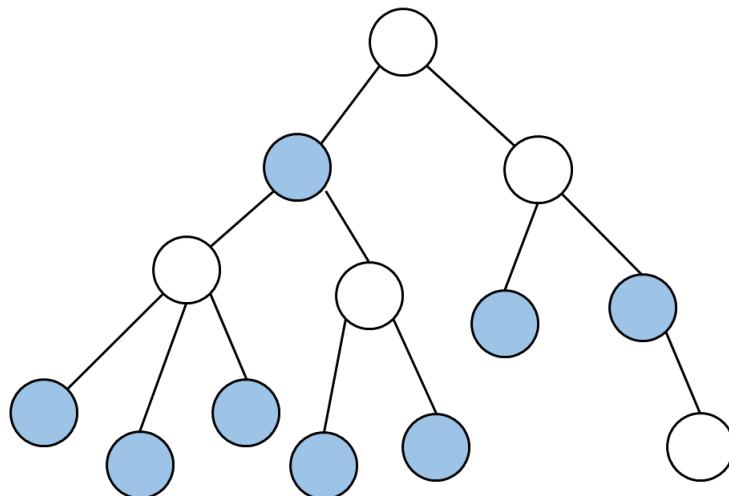


圖 13.9：一個樹上最大獨立集的例子，塗色點為選中的點集合

如果你會解二分圖上的最大獨立集，那麼同樣的解法也可以在這題使用，因為樹也是一張二分圖，只要將樹上的點依照深度的奇偶性分成兩群，就變成一張二分圖了<sup>ii</sup>。不過，二分圖最大獨立集解法的複雜度顯然是不符合題目要求的。下面來分享這道問題的一個經典解法：**樹 DP**。

樹 DP 的想法就是在樹上做 DP，那麼關鍵就在於如何去定義狀態以及狀態轉移。現在考慮求解一棵以  $u$  為根的子樹的最大獨立集。對於  $u$  這個節點有選與不選兩種可能，我們將兩種可能都來設想一下： $u$  如果被選了，那麼  $u$  的所有子節點必然都不能選（否則就存在兩個選中點在樹上有建邊）；而  $u$  如果不選，那麼任意子節點選與不選都可以（因為選與不選都不會使得有兩個選中的點之間有建邊）。

<sup>ii</sup>順道補充，若將網格圖上的每個格子對其上下左右的格子建邊，那這張圖也會是一張二分圖

對於兩種可能的答案，勢必都要對  $u$  的所有子（孫）節點遞迴考慮選與不選，而且在每次遞迴，顯然我們只在意其最佳解的情況（以選  $u$  的情況來說，固然只需考慮各個子節點為根的子樹，在不選根節點情況下的最大獨立集）。至此，已經可以清楚地定義 DP 狀態與狀態轉移了。

定義 DP 的狀態： $dp_1(u)$  為以  $u$  為根的子樹，在選  $u$  時的最大獨立集大小。 $dp_2(u)$  為以  $u$  為根的子樹，在不選  $u$  時的最大獨立集大小。

狀態轉移式：

$$dp_1(u) = 1 + \sum_{pa(v)=u} dp_2(v)$$

$$dp_2(u) = \sum_{pa(v)=u} \max \{dp_1(v), dp_2(v)\}$$

一棵樹根為  $u$  的樹，其最大獨立集大小即為  $\max \{dp_1(u), dp_2(u)\}$ 。

```

1 const int maxn = 1000005;
2 vector<int> G[maxn];      // G[u] : 與點 u 相鄰的點集合
3 int dp1[maxn], dp2[maxn];
4 void dfs(int u, int pa) {
5     dp1[u] = 1; dp2[u] = 0;
6     for(int v : G[u])
7         if(v != pa) {
8             dfs(v, u);
9             dp1[u] += dp2[v];
10            dp2[u] += max(dp1[v], dp2[v]);
11        }
12    }
13 int solve() {
14     dfs(1, 1);
15     return max(dp1[1], dp2[1]);
16 }
```

**程式碼 13.29:** 樹 DP 解樹上最大獨立集

狀態的總轉移次數為  $2N$ ，因此總時間複雜度為  $\mathcal{O}(N)$ 。

### 13.3.4 次小生成樹

在基礎圖論的章節中，已經學了在一張無向圖中求最小生成樹，現在考慮這個問題的進階版本：次小生成樹。次小顧名思義就是第二小的意思，次小生成樹故為一棵邊權總和第二小的生成樹。我們先從非嚴格次小生成樹開始。

給定一張  $V$  個點  $E$  條邊的無向圖。並給定哪些邊會形成某棵最小生成樹  $T$  ( $T$  是一個邊集合)。要求找到一棵  $T$  以外邊權和最小的生成樹。假設邊權皆為正。

$$1 \leq V \leq E \leq 10^5$$

首先，本題要求的非嚴格次小生成樹的邊權和是可能與最小生成樹（即  $T$ ）的邊權和相同，可想而知，嚴格次小生成樹就是一棵邊權和最小且小於最小生成樹（的邊權和）的生成樹，也就是說非嚴格與嚴格的差別就是邊權和小於等於  $T$  或小於  $T$ （的邊權和）。我們會在後面討論嚴格次小生成樹。

而關於求非嚴格次小生成樹，可以暴力枚舉所有生成樹，再找出與  $T$  相異且邊權和最小的生成樹，當然時間複雜度不會是我們想要的。

一個顯而易見的性質是，**非嚴格次小生成樹必有一條不在  $T$  裡的邊**。那麼就可以枚舉每一條不在  $T$  的邊（令當前枚舉到的邊為  $e$ ），並求出含有  $e$  且邊權和最小的生成樹，再從中挑出邊權和最小的那棵生成樹即為非嚴格次小生成樹了。共會求  $E - (V - 1)$  次生成樹（不在  $T$  裡的邊有  $E - (V - 1)$  條），如果使用 Kruskal's algorithm，維護連通性使用啟發式合併 Disjoint Set 搭配路徑壓縮，那時間複雜度便為  $\mathcal{O}(E \log E + (E - (V - 1)) \times E\alpha(V)) = \mathcal{O}(E^2\alpha(V))$ 。複雜度明顯會比前面的暴力好，但看來還是會超時。

枚舉不在  $T$  裡的邊雖可行，但每次求生成樹太花時間了，而且雖然每次都限制必選某條邊，但求出的生成樹的樣子（選中的邊）難道每次都差很多嗎？

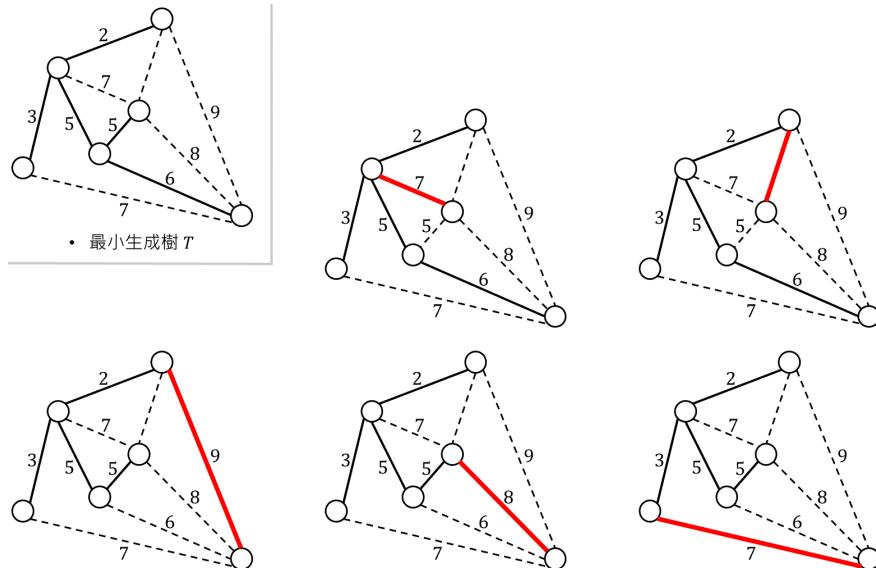


圖 13.10：限制必選各條不在  $T$  上的邊後，形成的各個生成樹；虛線表示邊沒被選中

觀察可以發現，每次枚舉求出的生成樹都跟  $T$  長很像。其實每次要找含  $e$  且邊權和最小的生成樹時，除了必選的  $e$  外，剩下的  $V - 2$  條邊總是可以從  $T$  裡挑出，相當於將  $e$  替換  $T$  裡的某條邊（可能存在含  $e$  且不同邊集但邊權和相同的方案）。宏觀來看，被  $e$  替換掉的邊在  $T$  中負責的是將  $e$  連接的兩點分別所在的連通塊連接在一起，而  $e$  也是一樣；固然不會因為換成了  $e$  而影響其他部分，所以這樣求出的生成樹即為含  $e$  且邊權和最小的。那麼關鍵就在要於從  $T$  裡拔掉哪一條邊。

令當前枚舉的邊  $e$ （也就是限制必選的邊）連接的兩個點為  $u$  與  $v$ ，若在  $T$  裡加入這條邊，則必會產生一個環。

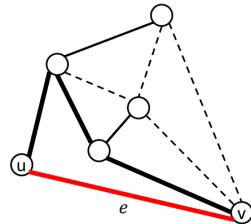


圖 13.11:  $T$  上加入  $e$  後必會產生一個環

因此不選的邊必然就是環上除了  $e$  之外的某條邊，而為了最小化邊權和，當然貪心地拔掉（不選）邊權最大的邊，也就是  $T$  中  $u$  到  $v$  路徑上邊權最大的邊。

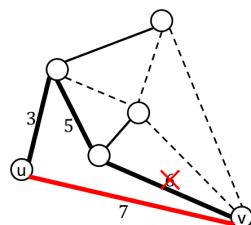


圖 13.12: 拔掉環上（除了  $e$  之外）邊權最大的邊

值得注意的是，拔掉的邊權重必不大於  $e$  的權重，否則會得到一棵比  $T$  邊權和更小的生成樹，這與  $T$  是最小生成樹矛盾。

再來的問題就是，如何快速地求出  $T$  中任兩點路徑上邊權最大的邊了。首先可以稍微更改 Floyd-Warshall algorithm，並透過它來求出任兩點路徑上邊權最大的邊，總時間複雜度為  $\mathcal{O}(V^3 + E)$ （現在每次求邊權和最小的生成樹不需要使用 Kruskal's algorithm 了）。

但別忘記  $T$  是一棵樹！樹上兩點路徑唯一，求兩點路徑上邊權最大的邊也比較簡單。可每次枚舉起點，再透過 DFS 維護好起點到各點路徑上邊權最大的邊。

```

1 const int maxv = 100005, maxe = 100005;
2 struct edge {
3     int u,v,weight;
4 }edges[maxe]; // 儲存每一條邊
5 bool in_MST[maxe]; // in_MST[i] : 第 i 條邊是否在給定的最小生成樹裡
6 vector<edge> G[maxv]; // G[u] : T 裡與點 u 相鄰的點集合
7 int max_edge[maxv][maxv]; // max_edge[u][v] : 樹上 u 到 v 路徑上的最大邊權
8 void dfs(int st,int u,int max_weight) { // 維護 st 到各點路徑上的最大邊權
9     max_edge[st][u] = max_weight;
10    for(edge e : G[u]) {
11        int v = e.v, weight = e.weight;
12        if(max_edge[st][v] == -1) // 這輪 DFS 還沒走過點 v
13            dfs(st,v,max(max_weight, weight));
14    }
15 }
16 int solve(int V,int E) {
17     int MST_sum = 0; // 累積的最小生成樹的邊權和
18     for(int i=1;i<=E;i++) {
19         if(in_MST[i]) {
20             int u = edges[i].u, v = edges[i].v, weight = edges[i].weight;
21             G[u].push_back(edge{u,v,weight});
22             G[v].push_back(edge{v,u,weight});
23             MST_sum += weight;
24         }
25     }
26     memset(max_edge, -1, sizeof max_edge); // 初始化 -1 表示沒走過
27     for(int i=1;i<=V;i++)
28         dfs(i,i,0);
29     int ret = -1; // 非嚴格次小生成樹的邊權和；-1 表示還沒有答案
30     for(int i=1;i<=E;i++)
31         if(!in_MST[i]) {
32             int u = edges[i].u, v = edges[i].v, weight = edges[i].weight;
33             int now_sum = MST_sum - max_edge[u][v] + weight;
34             if(ret == -1 || (now_sum < ret))
35                 ret = now_sum;
36         }
37     return ret;
38 }
```

**程式碼 13.30:** DFS 求兩點路徑最大權邊後求非嚴格次小生成樹

總時間複雜度為  $\mathcal{O}(V^2 + E)$ 。複雜度雖然更好了，但看來較適合用在稠密圖，本題在最糟情況下明顯是會超時的。另外記憶體是不夠開  $V \times V$  的二維陣列的，所以上面那份 code 在編譯時可能就編譯錯誤了。

還記得倍增法求 LCA（最近共同祖先）嗎？（忘記的同學或不會的同學不妨翻回進階圖論的章節看看）倍增法維護了每個點往上走 2 的幕次步後的點是哪個點，而我們其實可以同時維護這 2 的幕次步裡邊權最大的邊！

```

1 int dep[maxv]; // dep[i] : 點 i 在給定的最小生成樹上的深度
2 struct state {
3     int pa,max_weight; // 往上 2 的幕次步的點與最大邊權
4 }table[20][maxv]; // table[i][j] : j 往上走  $2^i$  步
5 void dfs(int u) { // 在給定的最小生成樹上 DFS 獲得倍增法所需資訊
6     for(edge e : G[u]) {
7         int v = e.v, weight = e.weight;
8         if(table[0][u].pa != v) {
9             dep[v] = dep[u] + 1;
10            table[0][v] = {u,weight};
11            dfs(v);
12        }
13    }
14 }
15 void build(int V) { // 建倍增法的表
16     dep[1] = 1;
17     table[0][1] = {1,0};
18     dfs(1);
19     for(int i=1;i<20;i++) {
20         for(int j=1;j<=V;j++) {
21             table[i][j].pa = table[i - 1][table[i - 1][j].pa].pa;
22             table[i][j].max_weight = max(table[i - 1][j].max_weight, table[i - 1][table[i - 1][j].pa].max_weight);
23         }
24     }
}

```

**程式碼 13.31:** 倍增法維護祖先與最大邊權

那麼若要求樹上某兩點  $u$  到  $v$  路徑上邊權最大的邊，就看  $u$  與  $v$  到它們的 LCA 的路徑上邊權最大的邊哪條比較大即可。

```

1 int query(int u,int v) { // 查詢給定的最小生成樹上 u 到 v 的最大邊權
2     int ret = 0;
3     if(dep[u] < dep[v]) swap(u,v); // 令 u 的深度總是大於 v
4     for(int i=19;i>=0;i--) {
5         if(dep[table[i][u].pa] >= dep[v]) {
6             ret = max(ret, table[i][u].max_weight);
7             u = table[i][u].pa;
8         }
9         if(u == v) return ret;
10        for(int i=19;i>=0;i--) {
11            if(table[i][u].pa != table[i][v].pa) {
12                ret = max({ret, table[i][u].max_weight, table[i][v].max_weight});
13                u = table[i][u].pa;
14                v = table[i][v].pa;
15            }
16            ret = max({ret, table[0][u].max_weight, table[0][v].max_weight});
17        }
18    }
19

```

```

20 int solve(int V, int E) {
21     int MST_sum = 0;
22     for(int i=1; i<=E; i++) {
23         if(in_MST[i]) {
24             int u = edges[i].u, v = edges[i].v, weight = edges[i].weight;
25             G[u].push_back(edge{u,v,weight});
26             G[v].push_back(edge{v,u,weight});
27             MST_sum += weight;
28         }
29     build(V);
30     int ret = -1;
31     for(int i=1; i<=E; i++) {
32         if(!in_MST[i]) {
33             int u = edges[i].u, v = edges[i].v, weight = edges[i].weight;
34             int now_sum = MST_sum - query(u,v) + weight;
35             if(ret == -1 || now_sum < ret) ret = now_sum;
36         }
37     return ret;
38 }

```

**程式碼 13.32:** 查詢兩點路徑上最大邊權；求非嚴格次小生成樹

最後分析時間複雜度：倍增法建表時，DFS 需要  $\mathcal{O}(V)$ ，維護各點往上走 2 的  
幕次步的資訊需要  $\mathcal{O}(V \log V)$ ；後面求非嚴格次小生成樹時，查詢  $T$  上兩點最  
大邊權需要  $\mathcal{O}(\log V)$ ，而會求邊權和最小的生成樹  $E - (V - 1)$  次。因此總時間  
複雜度  $\mathcal{O}(V + V \log V + (E - (V - 1)) \log V) = \mathcal{O}(E \log V)$ 。注意，本題有先給  
哪些邊是最小生成樹上的邊，若題目沒給，則需要先求最小生成樹，因此若要先  
求最小生成樹，則複雜度變為  $\mathcal{O}(E \log E + E \log V) = \mathcal{O}(E \log E)$  ( 使用 Kruskal's  
algorithm )。

這個技巧不單可以維護樹上兩點路徑上的最大邊權，同樣也可以維護其他不同的  
資訊！像是兩點路徑上邊權的最大連續和、兩點路徑上所有邊權的最大公因數等，  
有沒有覺得很像用線段樹解序列問題呢？畢竟將兩點路徑抽出來攤平其實也就  
跟一個序列差不多了！

最後來看如何求嚴格次小生成樹。

| 嚴格次小生成樹   | 經典問題 |
|---|------|
| 給定一張 $V$ 個點 $E$ 條邊的無向圖。並給定哪些邊會形成某棵最小生成樹 $T$ ，現<br>要求找到一棵邊權和小於 $T$ ( 的邊權和 ) 且邊權和最小的生成樹。<br>$1 \leq V \leq E \leq 10^5$ |      |

首先，嚴格次小生成樹可能存在（例如一張邊權皆相等的圖必然不存在嚴格  
次小生成樹）。下面假設嚴格次小生成樹存在。

嚴格次小生成樹同樣包含一條不在  $T$  上的邊。這時可能有個直觀的想法：使用前面求非嚴格次小生成樹的方法來求出嚴格次小生成樹，也就是看枚舉求出的那  $E - (V - 1)$  棵生成樹，哪棵邊權和小於  $T$  ( 的邊權和 ) 且邊權和最小。不過這方法是錯的，看個反例：

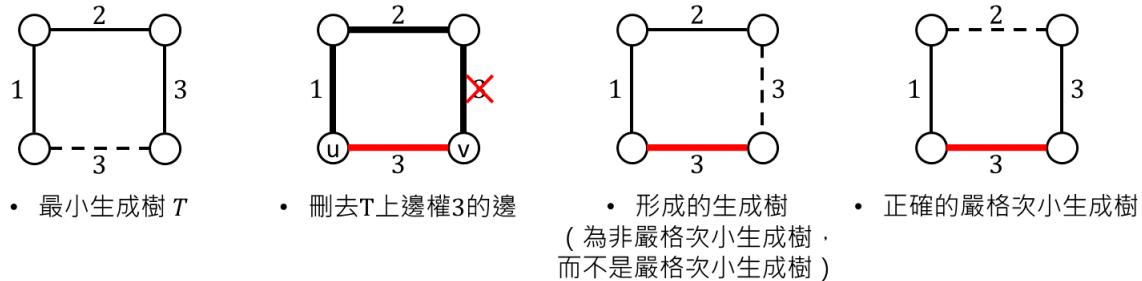


圖 13.13: 反例圖片

其實只要稍微改變一下上面方法即可：在每次枚舉必選的  $e$  時，再多嘗試替換 ( $T$  裡加入  $e$  會產生的) 環上邊權嚴格第二大的邊。而求環上的邊權第二大的邊，相當於求  $T$  上兩點路徑的權重嚴格第二大的邊，同樣可以使用倍增法維護。

```

1 struct state_weight {
2     int max, sec; // 多維護次大邊權；-1 表示不存在
3 };
4 struct state {
5     int pa;
6     state_weight w; // 更改原本倍增法狀態裡定義的邊權型別
7 }table[20][maxv];
8 /*
9 使用 merge 函式在 build 函式裡維護兩狀態合併後的最大邊權與次大邊權
10 query 函式裡一樣透過類似方法求次大邊權
11 */
12 state_weight merge(state_weight a, state_weight b) {
13     state_weight ret;
14     ret.max = max(a.max, b.max);
15     ret.sec = max({a.sec, b.sec, min(a.max, b.max)});
16     if(ret.sec == ret.max) ret.sec = -1;
17     return ret;
18 }
```

程式碼 13.33: 倍增法維護嚴格次大邊權

總時間複雜度一樣是  $\mathcal{O}(E \log V)$  ( 在不用先求最小生成樹的情況下 )。

而如果嚴格次小生成樹不存在，則表示替換最大權邊求出的生成樹邊權和都與最小生成樹一樣，且每次要替換次大權邊時，樹上兩點路徑的所有邊邊權都相等 ( 即不存在次大權邊 )。如果非本題，則可能還有種情況是圖根本不連通。

### 13.3.5 僅有刪邊的連通性維護

| 集合維護問題-改                                       | 經典問題 |
|--|------|
| 給定一張 $V$ 個點 $E$ 條邊的無向圖。請 $Q$ 次支援兩種操作：          |      |
| 1. 將第 $i$ 條邊刪除                                 |      |
| 2. 詢問點 $u$ 與點 $v$ 是否在相同的連通塊裡                   |      |
| $1 \leq V, E, Q \leq 10^5, 1 \leq u, v \leq N$ |      |

題目長得很像 Disjoint Set 的模板題（將某兩個元素所在的集合合併相當於圖上在兩點之間建邊；詢問兩點是否同集合相當於問兩點是否在相同的連通塊裡），只是建邊操作變成了刪邊操作。

也許可以修改 Disjoint Set 使它可以支援刪邊操作或是想辦法將題目轉化成 Disjoint Set 可以處理的樣子（即只有建邊操作）。下面針對後者進行討論。

而雖然無法處理刪邊操作，但我們可以處理其反著的操作：建邊。

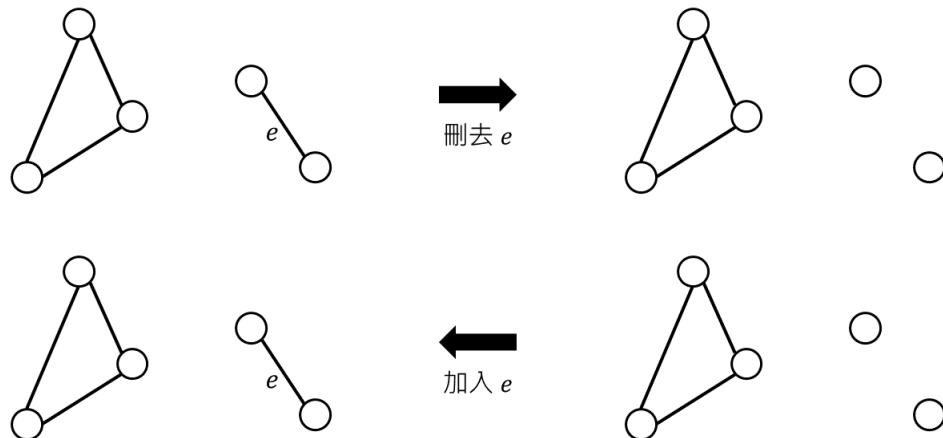


圖 13.14: 刪去  $e$  後，加入  $e$  即變回  $e$  刪去前的樣子

那要思考的點就是如何將刪邊反過來變成建邊。而其實只要把題目倒著做回來，那刪邊操作就變成了建邊操作了。具體來說，考慮這樣的一個離線作法：將操作全部讀入並儲存後，自然就知道了最後哪些邊被刪除，也就是圖在經過最後一次操作一後的樣子。那初始就建一張圖其為原本圖最終的樣子，再把讀入的操作反著處理回去，操作一時就把刪除的邊加回去，這樣就可以保證處理到任何操作時，圖的樣子會確實是原本正著一直處理到該筆操作後的樣子。操作二時直接查詢兩點是否在同個連通塊即可。至此問題已被轉化為 Disjoint Set 的模板題了。而這個技巧被稱做時光倒流。

```

1  /*
2  void Init(int V) : 初始化 Disjoint Set 所需資訊
3  int Find(int x) : 查詢 x 所在的集合的代表點編號
4  void Union(int x,int y) : 將 x 與 y 所在的兩個集合合併
5  bool Same(int x,int y) : 回傳 x 與 y 是否在相同的集合裡
6  */
7  const int maxe = 100005, maxq = 100005;
8  struct edge { // 紀錄圖上的邊
9      int u,v;
10 }edges[maxe];
11 bool del[maxe];
12 struct que { // 紀錄給定的詢問
13     // 分別為操作類型、(若為操作一)刪邊編號、(若為操作二)詢問的兩個點
14     int type,erase_idx,u,v;
15 }ques[maxq];
16 bool ans[maxq];
17 void solve(int V,int E,int Q) {
18     Init(V);
19     memset(del, false, sizeof del);
20     for(int i=1;i<=Q;i++) // 標記最終時刻 · 哪些邊被刪除
21         if(ques[i].type == 1)
22             del[ques[i].erase_idx] = true;
23     for(int i=1;i<=E;i++) // 將最終時刻還在的邊加入圖裡維護
24         if(!del[i]) {
25             int u = edges[i].u, v = edges[i].v;
26             if(Find(u) != Find(v))
27                 Union(u,v);
28         }
29     for(int i=Q;i>=1;i--) // 倒著處理各項操作
30         if(ques[i].type == 1) {
31             int u = edges[ques[i].erase_idx].u, v = edges[ques[i].erase_idx].v;
32             if(Find(u) != Find(v))
33                 Union(u,v);
34         }
35         else {
36             int u = ques[i].u, v = ques[i].v;
37             ans[i] = Same(u,v);
38         }
39     for(int i=1;i<=Q;i++)
40         if(ques[i].type == 2) // 是操作二才輸出
41             cout << (ans[i] ? "Yes" : "No") << '\n';
42 }

```

**程式碼 13.34:** 時光倒流

時間複雜度與 Disjoint Set 解一般的集合維護差不多 · 為  $\mathcal{O}(V+E+Q\alpha(V))$ 。

### 13.3.6 練習題 Exercise

|  |                    |
|--|--------------------|
| A Criminal   | Codeforces 101864A |
| <p><math>T</math> 次詢問：第 <math>i</math> 次詢問會隨機從 <math>[L_i, N_i]</math> 中挑選一個整數 <math>Y_i</math>。若約瑟夫問題中的 <math>N = Y_i</math> 且 <math>K = 2</math>，問獲勝的人編號是 <math>X_i - 1</math> 的機率為多少。</p> |                    |
| $1 \leq T \leq 10500, 1 \leq X_i \leq N_i \leq 10^{15}, 1 \leq L_i \leq N_i$   |                    |

|  |                  |
|--|------------------|
| Maximum Weight Subset  | Codeforces 1249F |
| <p>給定一棵 <math>n</math> 個點的樹，其中點有點權。再給定一個整數 <math>k</math>。</p> |                  |
| <p>求一個點集合滿足集合中任兩點在樹上的距離大於 <math>k</math>，並最大化集合的點權和。</p>       |                  |
| $1 \leq n, k \leq 200$<br>( 試試看 $\mathcal{O}(n^2)$ 的解法吧！)      |                  |

|   |                  |
|---|------------------|
| MST Unification   | Codeforces 1108F |
| <p>給定一張 <math>n</math> 個點 <math>m</math> 條邊的無向圖，其中邊有邊權。</p> |                  |
| <p>每次操作可以將一條邊的權重增加一（同一條邊可以被重複執行操作）。</p>                     |                  |
| <p>問最少執行幾次操作後，可以使得圖上的最小生成樹唯一。</p>                           |                  |
| $1 \leq n, m \leq 2 \times 10^5$                            |                  |

|   |                 |
|---|-----------------|
| Greg and Graph  | Codeforces 295B |
| <p>給定一張 <math>n</math> 個點的有向圖，任意點對間都有邊（即有恰好 <math>n^2</math> 條邊），邊有邊權。</p>  |                 |
| <p>接下來有 <math>n</math> 次操作，每次操作 <math>i</math> 須先輸出當前圖上所有點對的最短路的總和，接著將點 <math>x_i</math> 從圖上刪去（保證點 <math>x_i</math> 必還在圖裡；點 <math>x_i</math> 相鄰的邊也都會被刪去）。</p> |                 |
| $1 \leq n \leq 500$   |                 |



## 感謝

特別感謝 2020 清大程式競賽集訓營全體工作人員、講師群組、清大資訊工程學系承辦人員以及場館管理人，以及懷抱著熱誠希望的學員們。



# 版權宣告

本書籍僅限 2020 清大程式競賽集訓營成員瀏覽使用。

版權所有，翻印必究。

台灣新竹，國立清華大學, *August 3, 2020*

---

ION Camp

