競賽入門

陳克盈 Koying

2022-07-04

目錄

演算法競賽介紹

什麼是演算法？

- 在數學（算學）和電腦科學之中，一個被定義好的、計算機可施行之指示的有限步驟或次序
- 處理或計算資料的方法
- 有效、正確處理資料的方式

跟我們的生活有什麼關係？

- 搜尋引擎
- 導航
- AI
- 手邊手機裡所有的應用程式
- 登入系統

Pascal 設計者、1984 年圖靈獎得主維爾特 (Niklaus Emil Wirth) 曾經說過：

Algorithms + Data Structures = Programs

什麼是演算法競賽

- 又稱競賽程式（競程）

什麼是演算法競賽

- 又稱競賽程式（競程）
- 要在時間內寫出數道題目，並且要足夠快且足夠準確

什麼是演算法競賽

- 又稱競賽程式（競程）
- 要在時間內寫出數道題目，並且要足夠快且足夠準確
- 考驗參賽者的資料結構與演算法能力

什麼是演算法競賽

- 又稱競賽程式（競程）
- 要在時間內寫出數道題目，並且要足夠快且足夠準確
- 考驗參賽者的資料結構與演算法能力
- 過程非常的緊張刺激

競程題目的例子

硬幣問題

m 種硬幣，每種硬幣都有無限個，求湊到 x 元的最少硬幣數量
$(\mathrm{n} \leq 100, \mathrm{x} \leq 2 \times 10^5)$

最短路問題

給一張圖，兩點之間可能有一條長度為 $\mathrm{d_i}$ 的路徑相連，求 a 走到 b 的最短路徑
$(\mathrm{n} \leq 2 \times 10^5)$

競程題目的特色

- 可能有一些很像數學題的題目，但是數字都很大

競程題目的特色

- 可能有一些很像數學題的題目，但是數字都很大
- 用手算幾乎不可能

競程題目的特色

- 可能有一些很像數學題的題目，但是數字都很大
- 用手算幾乎不可能
- 只要精通演算法，就能夠輕鬆的解出看似很複雜的題目

競程題目的特色

- 可能有一些很像數學題的題目，但是數字都很大
- 用手算幾乎不可能
- 只要精通演算法，就能夠輕鬆的解出看似很複雜的題目
- 一道題目可能有非常多種不同的方式可以解出來

競程題目的特色

- 可能有一些很像數學題的題目，但是數字都很大
- 用手算幾乎不可能
- 只要精通演算法，就能夠輕鬆的解出看似很複雜的題目
- 一道題目可能有非常多種不同的方式可以解出來
- 每道題目都會有範例測試，但範例對了不代表最後的測資就會對

賽制

OI 賽制

- 資訊奧林匹亞所使用的賽制

OI 賽制

- 資訊奧林匹亞所使用的賽制
- 高中比賽中最常見的賽制

OI 賽制

- 資訊奧林匹亞所使用的賽制
- 高中比賽中最常見的賽制
- 每題 100 分，有部分分，沒有罰時

OI 賽制

- 資訊奧林匹亞所使用的賽制
- 高中比賽中最常見的賽制
- 每題 100 分，有部分分，沒有罰時
- 2017 年版本的 OI 賽制有子任務聯集

OI 賽制

- 資訊奧林匹亞所使用的賽制
- 高中比賽中最常見的賽制
- 每題 100 分，有部分分，沒有罰時
- 2017 年版本的 OI 賽制有子任務聯集
- 子題經常會引導參賽者想到正解

ICPC 賽制

- 大學比賽的賽制

ICPC 賽制

- 大學比賽的賽制
- 沒有部分分

ICPC 賽制

- 大學比賽的賽制
- 沒有部分分
- 優先比較 AC 數，再比罰時

ICPC 賽制

- 大學比賽的賽制
- 沒有部分分
- 優先比較 AC 數，再比罰時
- 罰時算法：每題 AC 時的時間 + 有 AC 的題目 WA 的次數 $\times$ 20

ICPC 賽制

- 大學比賽的賽制
- 沒有部分分
- 優先比較 AC 數，再比罰時
- 罰時算法：每題 AC 時的時間 + 有 AC 的題目 WA 的次數 $\times$ 20
- 每 AC 一題都會有氣球可以拿

升學制度

升學管道

- 特殊選才（10 ～ 12 月）
- 個人申請（1 ～ 6 月）
- 繁星推薦（3 月）
- 分科測驗（8 月）

特殊選才

- 大學獨立招生

特殊選才

- 大學獨立招生
- 提供給有特殊專長的學生

特殊選才

- 大學獨立招生
- 提供給有特殊專長的學生
- 不用考學測

特殊選才

- 大學獨立招生
- 提供給有特殊專長的學生
- 不用考學測
- 需要有很突出的經歷（比賽、專案等）

特殊選才

- 大學獨立招生
- 提供給有特殊專長的學生
- 不用考學測
- 需要有很突出的經歷（比賽、專案等）
- 以競賽程式為大宗

特殊選才

- 大學獨立招生
- 提供給有特殊專長的學生
- 不用考學測
- 需要有很突出的經歷（比賽、專案等）
- 以競賽程式為大宗
- 可以問 Colten

個人申請

- 大部分學生的升學方式

個人申請

- 大部分學生的升學方式
- 如果在高中有競賽或是專案的經驗，會很吃香

個人申請

- 大部分學生的升學方式
- 如果在高中有競賽或是專案的經驗，會很吃香
- APCS 組
  - 除了學測門檻之外，還有 APCS 成績門檻
  - 可以以較低的學測成績進入二階（以台大資工為例）
    - 一般組：英 + 自 27、數 15
    - APCS 組：APCS 4/4、數 13
  - (特選 + 學測)/2

繁星推薦

- 校內、學測成績
- 可以問 Gino 講師

分科測驗

- 考試。

如何進步

一位電神可能會具備的特質

- 樂於探索新事物

一位電神可能會具備的特質

- 樂於探索新事物
- 參加各種活動、豐富經驗

一位電神可能會具備的特質

- 樂於探索新事物
- 參加各種活動、豐富經驗
- 勇於踏出舒適圈，探索更多自己不熟的領域

一位電神可能會具備的特質

- 樂於探索新事物
- 參加各種活動、豐富經驗
- 勇於踏出舒適圈，探索更多自己不熟的領域
- 不害怕失敗，永不放棄

一位電神可能會具備的特質

- 樂於探索新事物
- 參加各種活動、豐富經驗
- 勇於踏出舒適圈，探索更多自己不熟的領域
- 不害怕失敗，永不放棄
- 以上四點講師都沒有

練題

- 不要盲目刷題

練題

- 不要盲目刷題
- 可以針對某個單元練習
- 以不同的角度思考題目

練題

- 不要盲目刷題
- 可以針對某個單元練習
- 以不同的角度思考題目
- 網路上有很多題目
- OJ 們：
  - ZeroJudge
  - TIOJ
  - CSES
  - Codeforces
  - AtCoder

ZeroJudge

- 題數到達一定數量就可以自己出題
- 題目很多，尤其是各個地方的考古題
- 題目品質沒有保障
- 建議有題單再去練

TIOJ

- 建中的 OJ
- 很多高難度的題目
- TL、ML 通常都很緊

CSES

- 國外的 OJ，但題目的英文都不難
- 很多經典題
- 如果想要學一個新的演算法可能可以從那邊找到模板題
- 有些人會按照裡面的題目來學演算法

Codeforces

- 俄羅斯某大學維護的 OJ
- 每個禮拜幾乎都有超過一場比賽
  - 分為 Div1∼4，數字越小越難
  - 因為是俄羅斯平台，因此時間多在晚上十點
- 比較是針對 ICPC 打造的平台
- 題目跟 OI 賽制有一定落差
- Rating 跟實力成正相關，但是不代表 Rating 高就會在 OI 賽制打得很好

AtCoder

- 日本的 OJ
- 每個禮拜都有一到兩場比賽
  - ABC、ARC、AGC
  - 難度 ABC < ARC < AGC
  - 幾乎都在晚上八點舉行
- 比 Codeforces 的題目還要好讀
- ABC 很適合演算法初學者參加

練習賽

- NHDK 四校聯合初學者程式設計練習賽
- TOI 線上練習賽
- Codeforces
- AtCoder

NHDK 四校聯合初學者程式設計練習賽

- 由南部學生自發辦理的練習賽
- 每個月至少一場
- 週日下午 14:00 ～ 17:00
- 分為 Div.1～3
  - Div.1: 較進階的演算法
  - Div.2: 基礎演算法
  - Div.3: 語法

TOI 練習賽

- 師大舉辦的，OI 賽制
- 每年 4、5、6、10、11、12 月的最後一個禮拜舉辦
- 新手組、潛力組
  - 新手組：語法
  - 潛力組：算法
- 每次三題，共 300 分
- 參加之後有成績證明

Codeforces

- 某幾天的晚上有比賽
- 通常是 22:35 開始，但有時候會有其他時間
- 有 Rating 系統，如果有基礎的演算法知識可以去挑戰看看

- 注意這隻蜜蜂出的場次

AtCoder

- 周末的 20:00 有比賽
- 推薦 ABC 場
- 打起來比 Codeforces 舒服

非常規賽（按照時間排序）

- APCS 大學程式先修檢測
- YTP 少年圖靈競賽（7 月）
- ISSC 青年程式競賽
- 學科能力競賽（9 ∼ 12 月）
- NPSC 網際網路程式設計大賽
- HP CodeWars
- 資訊奧林匹亞

APCS 大學程式先修檢測

- 每年 1、6、10 月舉辦
- 觀念題、實作題
- 實作題共四題，每題滿分 100，每筆測資 5 分
- 可用 C/C++、Python、Java 作答
  1. 基本語法
  2. 進階語法運用（繁雜的實作題）
  3. 排序、STL 運用
  4. 分治、動態規劃、圖論
- https://apcs.csie.ntnu.edu.tw/

YTP 少年圖靈計畫

- 通常在暑假舉辦
- 分為四個階段：線上初賽、線下複賽、專題實作、海外學習
- 每隊三人，不限學校
- 線上初賽：每年的 7 月舉行，採線上進行
- 線下複賽：線上初賽的兩個禮拜後在台北舉行，每年都有不錯的食物／炸雞可以吃
- 專題實作：線下複賽最多取 15 組進入專題實作，在該年的 8 月到隔年 1 月會有指導教授指導進行專題實作以及其他提升軟體實作能力的活動，包括軟體開發/實作、社群交流以及線上座談，且每個月都有獎學金
- 海外學習：專題實作發表獲得前三名的隊伍將獲得海外學習的機會

ISSC 青年程式競賽

- 實體賽，辦在東海大學
- ICPC 制
- 題目品質不太穩定，但可以順便去台中玩

學科能力競賽

- OI 賽制，簡稱能競，與資奧被稱為一年中最重要的兩場賽事
- 校內賽：每年 6 ～ 9 月
- 區賽：10 ～ 11 月舉辦
- 全國賽：每年 12 月舉辦，大型膜拜現場，前十名會保送選訓營

NPSC 網際網路程式設計大賽

- 台大舉辦，ICPC 制，每隊三人

NPSC 網際網路程式設計大賽

- 台大舉辦，ICPC 制，每隊三人
- 高中組、國中組

NPSC 網際網路程式設計大賽

- 台大舉辦，ICPC 制，每隊三人
- 高中組、國中組
- 11 月舉辦線上初賽，12 月舉辦線下決賽（約取 20 ∼ 25 隊）

NPSC 網際網路程式設計大賽

- 台大舉辦，ICPC 制，每隊三人
- 高中組、國中組
- 11 月舉辦線上初賽，12 月舉辦線下決賽（約取 20 ∼ 25 隊）
- 難度頗高，但可以體驗到 ICPC 制的樂趣

HP CodeWars

- 惠普舉辦的線下賽

HP CodeWars

- 惠普舉辦的線下賽
- 有很多獎品可以拿

資訊奧林匹亞

- **OI 賽制**

資訊奧林匹亞

- OI 賽制
- 海選
  - APCS 考古題
  - 通過可以前進初選 （APCS 實作三級有同等效益）

資訊奧林匹亞

- OI 賽制
- 海選
  - APCS 考古題
  - 通過可以前進初選（APCS 實作三級有同等效益）
- 初選
  - 也稱作入營考
  - 每年三月舉辦
  - 每個年級的前四名 + 剩下的前 12 名會晉級選訓營

資訊奧林匹亞

- OI 賽制
- 海選
  - APCS 考古題
  - 通過可以前進初選（APCS 實作三級有同等效益）
- 初選
  - 也稱作入營考
  - 每年三月舉辦
  - 每個年級的前四名 + 剩下的前 12 名會晉級選訓營
- 1!
  - 入營考 20 名 + 能競 10 名
  - 根據模擬賽成績進入 2!

資訊奧林匹亞

- OI 賽制
- 海選
  - APCS 考古題
  - 通過可以前進初選（APCS 實作三級有同等效益）
- 初選
  - 也稱作入營考
  - 每年三月舉辦
  - 每個年級的前四名 + 剩下的前 12 名會晉級選訓營
- 1!
  - 入營考 20 名 + 能競 10 名
  - 根據模擬賽成績進入 2!
- 2!
  - 可以代表臺灣參加 APIO（亞太資奧）
  - 1! + 2! 的模考加權之後前四名會代表臺灣參加 IOI（國際資奧）

營隊們

- APCS Camp
- ION Camp
- IOI Camp
- 本營隊

APCS Camp

- 台大資工系學生舉辦的
- 通常辦在暑假，線上
- 講師陣容有多位國手
- 有語法班以及算法班
- 算法班範圍是 APCS 5 級分的內容

ION Camp

- 清大資工舉辦，俗稱離子營
- 通常辦在暑假，實體，但今年因疫情轉為線上
- 難度比 APCS Camp 還要難，可以學到很多技巧
- 非常推薦大家在高中生活都能夠去一次

IOI Camp

- 由台大資工舉辦
- 舉辦於寒假，實體
- 堪稱是全台最難的營隊之一
- 含有許多超進階的演算法

三校聯合暑期培訓營

- 籌辦時程趕到我不知道怎麼生出來的營隊
- 免費，超香
- 不知道會不會辦下一屆

三校聯合暑期培訓營

- 籌辦時程趕到我不知道怎麼生出來的營隊
- 免費，超香
- 不知道會不會辦下一屆
- 還有機器狗可以玩

社群、研討會

- 參與社群活動也是進步的方法之一

社群、研討會

- 參與社群活動也是進步的方法之一
- 能夠認識更多電神、看到更多東西，可能對你的人生有很大的影響
- 也可以多多參加各種研討會，甚至是去當講者

社群、研討會

- 參與社群活動也是進步的方法之一
- 能夠認識更多電神、看到更多東西，可能對你的人生有很大的影響
- 也可以多多參加各種研討會，甚至是去當講者
- 推薦的社群、研討會或是課程
  - 中學資訊討論群 CISC
  - SITCON 學生計算機年會
  - SCIST 南部學生資訊社群
  - 資訊之芽

- 可以認識全台各地對資訊有興趣的學生的地方（不只演算法專長，也有很多開發專長的人）
- 雖然還不是很完美，但需要大家的參與才能夠繼續壯大

SITCON 學生計算機年會

- 為了學生所舉辦的資訊研討會（主要是大學生）
- 每年在中研院人社科學館舉辦，今年辦在 9/4
- 可以跟大家分享自己的成果
- 也有邀請業界的人士來舉辦論壇
- 可以認識到各種專業的人

SCIST 南部學生資訊社群

- 近年由南部的高中生所設立的社群
- 有演算法及資安的課程
- 某些周末會在成大上課，也有線上的旁聽名額
- 如果你是南部人，推推

資訊之芽

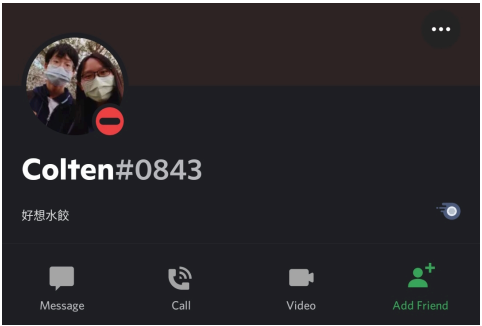
- 由台大、清大資訊系主辦的課程
- C 語法班、Python 語法、算法班
- 課程內容豐富，培養出了許多電神

練題練累了怎麼辦

- 寫點不一樣的東西

練題練累了怎麼辦

- 寫點不一樣的東西
- 找你的電神朋友聊天（如下圖），說不定可以激盪出什麼想法（雖然他可能會塑膠你）



競賽先備知識

題目中的雙標：時間複雜度與空間複雜度

什麼是時間複雜度

- 規劃考試作答時間嗎

什麼是時間複雜度

- 規劃考試作答時間嗎
- 運算數量隨著資料量上升而上升的指數（函數）關係

什麼是時間複雜度

- 規劃考試作答時間嗎
- 運算數量隨著資料量上升而上升的指數（函數）關係
- 用來估算這個程式在大資料量下的運算次數（通常考慮最差情況）

什麼是時間複雜度

- 規劃考試作答時間嗎
- 運算數量隨著資料量上升而上升的指數（函數）關係
- 用來估算這個程式在大資料量下的運算次數（通常考慮最差情況）
- 與執行時間成正相關，但執行時間不是只受複雜度影響

什麼是時間複雜度

- 規劃考試作答時間嗎
- 運算數量隨著資料量上升而上升的指數（函數）關係
- 用來估算這個程式在大資料量下的運算次數（通常考慮最差情況）
- 與執行時間成正相關，但執行時間不是只受複雜度影響
- 用來判斷一個程式是否能達到題目要求

什麼是時間複雜度

- 規劃考試作答時間嗎
- 運算數量隨著資料量上升而上升的指數（函數）關係
- 用來估算這個程式在大資料量下的運算次數（通常考慮最差情況）
- 與執行時間成正相關，但執行時間不是只受複雜度影響
- 用來判斷一個程式是否能達到題目要求
- 以大 O 符號（$\mathcal{O}(...)$）來表示

時間複雜度的算法

- 以一次基本運算（賦值、加法等等）作為 $\mathcal{O}(1)$

時間複雜度的算法

- 以一次基本運算（賦值、加法等等）作為 $\mathcal{O}(1)$
- 計算每個函式或是迴圈會運行幾次基本運算

時間複雜度的算法

- 以一次基本運算（賦值、加法等等）作為 $\mathcal{O}(1)$
- 計算每個函式或是迴圈會運行幾次基本運算
- 下圖是一個 $\mathcal{O}(\mathrm{n})$ 的程式

```
for (int i = 0; i < n; i++) {
    cout << i << endl;
}
```

時間複雜度的例子

請問這隻程式的時間複雜度是多少呢？

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        cout << i << ' ' << j << endl;
    }
}

for (int i = 0; i < n; i++) {
    cout << i << endl;
}
```

時間複雜度的例子

- 可能有些人覺得是 $\mathcal{O}(\mathrm{n}^2 + \mathrm{n})$

時間複雜度的例子

- 可能有些人覺得是 $\mathcal{O}(\mathrm{n}^2 + \mathrm{n})$
- 但實際上，當 n 很大時，$\mathcal{O}(\mathrm{n})$ 對 $\mathcal{O}(\mathrm{n}^2)$ 的影響微乎其微

時間複雜度的例子

- 可能有些人覺得是 $\mathcal{O}(\mathrm{n}^2 + \mathrm{n})$
- 但實際上，當 n 很大時，$\mathcal{O}(\mathrm{n})$ 對 $\mathcal{O}(\mathrm{n}^2)$ 的影響微乎其微
- 量級差太多的項次我們會省略，並稱他為常數

時間複雜度的例子

- 可能有些人覺得是 $\mathcal{O}(\mathrm{n}^2 + \mathrm{n})$
- 但實際上，當 n 很大時，$\mathcal{O}(\mathrm{n})$ 對 $\mathcal{O}(\mathrm{n}^2)$ 的影響微乎其微
- 量級差太多的項次我們會省略，並稱他為常數
- 因此這支程式的時間複雜度為 $\mathcal{O}(\mathrm{n}^2)$

時間複雜度的例子

再來一個

```
for (int t = 0; t < 2; t++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cout << i << ' ' << j << endl;
```

時間複雜度的例子

- $\mathcal{O}(2 \times \mathrm{n}^2)$ **還是** $\mathcal{O}(\mathrm{n}^2)$**？**

時間複雜度的例子

- $\mathcal{O}(2 \times \mathrm{n}^2)$ **還是** $\mathcal{O}(\mathrm{n}^2)$**？**
- **我們仍然會將那個兩倍視為常數**

時間複雜度的例子

- $\mathcal{O}(2 \times \mathrm{n}^2)$ 還是 $\mathcal{O}(\mathrm{n}^2)$？
- 我們仍然會將那個兩倍視為常數
- 為甚麼呢？

時間複雜度的例子

- $\mathcal{O}(2 \times \mathrm{n}^2)$ 還是 $\mathcal{O}(\mathrm{n}^2)$？
- 我們仍然會將那個兩倍視為常數
- 為甚麼呢？
- 其實並不是所有的基本運算都是一樣的執行時間，像是 `mod` 就會比一般的加法還要慢，因此去計較是 $\mathcal{O}(2 \times \mathrm{n}^2)$ 還是 $\mathcal{O}(\mathrm{n}^2)$ 是沒有意義的，能夠看出成長的趨勢即可

時間複雜度的例子

- $\mathcal{O}(2 \times \mathrm{n}^2)$ 還是 $\mathcal{O}(\mathrm{n}^2)$？
- 我們仍然會將那個兩倍視為常數
- 為甚麼呢？
- 其實並不是所有的基本運算都是一樣的執行時間，像是 `mod` 就會比一般的加法還要慢，因此去計較是 $\mathcal{O}(2 \times \mathrm{n}^2)$ 還是 $\mathcal{O}(\mathrm{n}^2)$ 是沒有意義的，能夠看出成長的趨勢即可
- 時間複雜度的計算，除了會忽略低次項之外，也會忽略係數

均攤時間複雜度

- 我們在估算時間複雜度的時候，我們會以最壞的可能去估算，通常是考慮範圍最大的情況

均攤時間複雜度

- 我們在估算時間複雜度的時候，我們會以最壞的可能去估算，通常是考慮範圍最大的情況
- 但有些操作，在不同的數值所需的操作數可能大可能小，例如某個操作在 i 為二的次方時的複雜度為 $\mathcal{O}(\mathrm{n})$，但在其他情況下的複雜度卻只有 $\mathcal{O}(1)$

均攤時間複雜度

- 我們在估算時間複雜度的時候，我們會以最壞的可能去估算，通常是考慮範圍最大的情況
- 但有些操作，在不同的數值所需的操作數可能大可能小，例如某個操作在 i 為二的次方時的複雜度為 $\mathcal{O}(\mathrm{n})$，但在其他情況下的複雜度卻只有 $\mathcal{O}(1)$
- 那我們就可以將所有操作的複雜度加起來，再取平均，就會發現 $\mathcal{O}(\mathrm{n})$ 的情況被 $\mathcal{O}(1)$ 平攤掉了

均攤時間複雜度

- 我們在估算時間複雜度的時候，我們會以最壞的可能去估算，通常是考慮範圍最大的情況
- 但有些操作，在不同的數值所需的操作數可能大可能小，例如某個操作在 i 為二的次方時的複雜度為 $\mathcal{O}(\mathrm{n})$，但在其他情況下的複雜度卻只有 $\mathcal{O}(1)$
- 那我們就可以將所有操作的複雜度加起來，再取平均，就會發現 $\mathcal{O}(\mathrm{n})$ 的情況被 $\mathcal{O}(1)$ 平攤掉了
- 這種有好有壞，對他做平均的分析方法就叫做均攤分析

一些常見的時間複雜度

- $\mathcal{O}(1)$：基本運算
- $\mathcal{O}(n)$：線性搜、遍歷陣列
- $\mathcal{O}(n^2)$：氣泡排序、某些情況下的枚舉
- $\mathcal{O}(\log n)$：二分搜、快速冪
- $\mathcal{O}(n \log n)$:merge sort、某些分治演算法
  - $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$ 也是 $\mathcal{O}(n \log n)$
- $\mathcal{O}(n!)$：排列枚舉
- $\mathcal{O}(2^n)$：位元枚舉

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？
- 現在電腦處理器的運行時脈是以 GHz 的單位來表示的，代表了 CPU 一秒可以做 $\mathrm{n} \times 10^9$ 次的基本操作

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？
- 現在電腦處理器的運行時脈是以 GHz 的單位來表示的，代表了 CPU 一秒可以做 $\mathrm{n} \times 10^9$ 次的基本操作
- 但是每一次的運算不會只需要一次的基本操作，再加上系統的效能分配，所以大概把他減去一個數量集，就可以得到 10^8 這個數字

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？
- 現在電腦處理器的運行時脈是以 GHz 的單位來表示的，代表了 CPU 一秒可以做 $n \times 10^9$ 次的基本操作
- 但是每一次的運算不會只需要一次的基本操作，再加上系統的效能分配，所以大概把他減去一個數量集，就可以得到 10^8 這個數字
- 我們會以 10^8 為標準，若時間複雜度算出來超過這個數字不少，那基本上就可以確定會得到 TLE

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？
- 現在電腦處理器的運行時脈是以 GHz 的單位來表示的，代表了 CPU 一秒可以做 $n \times 10^9$ 次的基本操作
- 但是每一次的運算不會只需要一次的基本操作，再加上系統的效能分配，所以大概把他減去一個數量集，就可以得到 10^8 這個數字
- 我們會以 10^8 為標準，若時間複雜度算出來超過這個數字不少，那基本上就可以確定會得到 TLE
- Judge 的速度都不一樣，像是 Codeforces 就有機會跑到一秒接近 10^9，甚至有時候編譯器會幫我們做一些黑魔法加速，所以 10^8 算是一個比較保守的估計

時間複雜度的標準

- 學會了如何估算時間複雜度之後，該怎麼判斷這個複雜度會不會過呢？
- 現在電腦處理器的運行時脈是以 GHz 的單位來表示的，代表了 CPU 一秒可以做 $n \times 10^9$ 次的基本操作
- 但是每一次的運算不會只需要一次的基本操作，再加上系統的效能分配，所以大概把他減去一個數量集，就可以得到 10^8 這個數字
- 我們會以 10^8 為標準，若時間複雜度算出來超過這個數字不少，那基本上就可以確定會得到 TLE
- Judge 的速度都不一樣，像是 Codeforces 就有機會跑到一秒接近 10^9，甚至有時候編譯器會幫我們做一些黑魔法加速，所以 10^8 算是一個比較保守的估計
- 學會估算時間複雜度之後，有時候看到題目的範圍就可以大概猜出是什麼複雜度的解法

時間複雜度的優化

- 接下來的課程，除了會教你如何成功地將答案算出來之外，也會教你如何優化這隻程式

時間複雜度的優化

- 接下來的課程，除了會教你如何成功地將答案算出來之外，也會教你如何優化這隻程式
- 所做的事情基本上就是在優化時間複雜度

時間複雜度的優化

- 接下來的課程，除了會教你如何成功地將答案算出來之外，也會教你如何優化這隻程式
- 所做的事情基本上就是在優化時間複雜度
- 這邊先舉一個最簡單的例子

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$

$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度
- 每次查詢最多會計算 N 個元素，共查詢 Q 次，所以時間複雜度是 $\mathcal{O}(\mathrm{NQ})$，顯然會 TLE

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度
- 每次查詢最多會計算 N 個元素，共查詢 Q 次，所以時間複雜度是 $\mathcal{O}(\mathrm{NQ})$，顯然會 TLE
- 我們會發現到說，在這 Q 次的操作中，我們可能會重複計算到很多重疊的區間，例如我們如果已經算過 $[1, 10]$，那再算 $[2, 5]$ 時就重複計算到了區間

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度
- 每次查詢最多會計算 N 個元素，共查詢 Q 次，所以時間複雜度是 $\mathcal{O}(\mathrm{NQ})$，顯然會 TLE
- 我們會發現到說，在這 Q 次的操作中，我們可能會重複計算到很多重疊的區間，例如我們如果已經算過 $[1, 10]$，那再算 $[2, 5]$ 時就重複計算到了區間
- 那我們還不如預先將一些區間先算出來，後面要用就直接用

前綴和優化

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度
- 每次查詢最多會計算 N 個元素，共查詢 Q 次，所以時間複雜度是 $\mathcal{O}(\mathrm{NQ})$，顯然會 TLE
- 我們會發現到說，在這 Q 次的操作中，我們可能會重複計算到很多重疊的區間，例如我們如果已經算過 $[1, 10]$，那再算 $[2, 5]$ 時就重複計算到了區間
- 那我們還不如預先將一些區間先算出來，後面要用就直接用
- 我們可以定義 $\mathrm{pre_i}$ 是 $1 \sim \mathrm{i}$ 的前綴和，也就是 $\sum_{\mathrm{i}=1}^{\mathrm{i}} \mathrm{a_i}$，那如果要取得 $[\mathrm{l}, \mathrm{r}]$ 的和，就只需要算 $\mathrm{pre_r} - \mathrm{pre_{l-1}}$ 就可以了

TPR #23 PH1 區間求和問題（一維版本）

給定一個長度為 N 的數列 a，接著有 Q 筆查詢，對於每次查詢會輸入兩個變數 L, R 代表要求 $[\mathrm{L}, \mathrm{R}]$ 之間的元素和，也就是 $\sum_{\mathrm{i}=\mathrm{L}}^{\mathrm{R}} \mathrm{a_i}$
$(\mathrm{n} \leq 2 \times 10^5, \mathrm{Q} \leq 10^4)$

- 先簡單計算一下暴力解的時間複雜度
- 每次查詢最多會計算 N 個元素，共查詢 Q 次，所以時間複雜度是 $\mathcal{O}(\mathrm{NQ})$，顯然會 TLE
- 我們會發現到說，在這 Q 次的操作中，我們可能會重複計算到很多重疊的區間，例如我們如果已經算過 $[1, 10]$，那再算 $[2, 5]$ 時就重複計算到了區間
- 那我們還不如預先將一些區間先算出來，後面要用就直接用
- 我們可以定義 $\mathrm{pre_i}$ 是 $1 \sim \mathrm{i}$ 的前綴和，也就是 $\sum_{\mathrm{i}=1}^{\mathrm{i}} \mathrm{a_i}$，那如果要取得 $[\mathrm{l}, \mathrm{r}]$ 的和，就只需要算 $\mathrm{pre_r} - \mathrm{pre_{l-1}}$ 就可以了
- 這個就是最常見的前綴和優化，而利用區間相減得到另外一個區間的動作就叫做差分

先備知識與實用技巧

常見的數學符號

- $\sum$ 代表連加，如 $\sum_{i=1}^{n} a_i$ 代表將 $a_1 \sim a_n$ 加起來
- $\prod$ 則是將 $\sum$ 的連加變為連乘
- $a \mid b$ 代表 a 整除 b 或是 b 能被 a 整除例如 $6 \mid 24$
- $\mathbf{gcd}(a, b)$ 代表 a, b 的最大公因數
- $\mathrm{lcm}(a, b)$ 代表 a, b 的最小公倍數
- $\log_a b$ 代表 $a^{\log_a b} = b$，而電腦科學中的 **log** 是以 2 為底的 $(\mathbf{log}_2\, b)$
  - $\mathbf{log}_2\, 8 = 3$，$\mathbf{log}_2\, \frac{1}{2} = -1$
- a **mod** b 代表 a 對 b 取餘數的結果

位元運算

- AND: 在 C++ 中以 & 表示，當兩個 bit 都為 1 時為 1，否則為 0

位元運算

- **AND: 在 C++ 中以 & 表示，當兩個 bit 都為 1 時為 1，否則為 0**
- **OR: 在 C++ 中以 | 表示，當兩個 bit 有至少一個為 1 時為 1，否則為 0**

位元運算

- AND: 在 C++ 中以 & 表示，當兩個 bit 都為 1 時為 1，否則為 0
- OR: 在 C++ 中以 | 表示，當兩個 bit 有至少一個為 1 時為 1，否則為 0
- XOR: 在 C++ 中以 ^表示，當兩個 bit 洽有一個為 1 時為 1，否則為 0

位元運算

- AND: 在 C++ 中以 & 表示，當兩個 bit 都為 1 時為 1，否則為 0
- OR: 在 C++ 中以 | 表示，當兩個 bit 有至少一個為 1 時為 1，否則為 0
- XOR: 在 C++ 中以 ^表示，當兩個 bit 洽有一個為 1 時為 1，否則為 0
- NOT: 在 C++ 中以 ~表示，代表將 0 轉為 1，1 轉為 0

萬用標頭檔

- 在 C++ 中，有這麼一個標頭檔叫做 bits/stdc++.h

萬用標頭檔

- 在 C++ 中，有這麼一個標頭檔叫做 bits/stdc++.h
- 他會函入幾乎所有常用的標頭檔

萬用標頭檔

- 在 C++ 中，有這麼一個標頭檔叫做 bits/stdc++.h
- 他會函入幾乎所有常用的標頭檔
- 這個標頭檔是方便用，但還是盡量要知道每個容器或是函式需要含入的標頭檔是哪個
- 請別在刷題以外的地方使用

萬用標頭檔

- 在 C++ 中，有這麼一個標頭檔叫做 bits/stdc++.h
- 他會函入幾乎所有常用的標頭檔
- 這個標頭檔是方便用，但還是盡量要知道每個容器或是函式需要含入的標頭檔是哪個
- 請別在刷題以外的地方使用
- 接下來兩個禮拜的內容，講師的範例程式碼應該超過九成都是使用萬用標頭檔

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少

IO 加速

- 若是使用 cin, cout 來輸入輸出，會發現他的速度比 scanf, printf 還要慢上不少
- 這是因為 cin 會自動將緩衝區清除，而 scanf 不會

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少
- 這是因為 `cin` 會自動將緩衝區清除，而 `scanf` 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少
- 這是因為 `cin` 會自動將緩衝區清除，而 `scanf` 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字
- 但這對於演算法題目並不是非常必要

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少
- 這是因為 `cin` 會自動將緩衝區清除，而 `scanf` 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字
- 但這對於演算法題目並不是非常必要
- 因此我們可以使用 `cin.tie(0)` 來解除自動清除緩衝區

IO 加速

- 若是使用 cin, cout 來輸入輸出，會發現他的速度比 scanf, printf 還要慢上不少
- 這是因為 cin 會自動將緩衝區清除，而 scanf 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字
- 但這對於演算法題目並不是非常必要
- 因此我們可以使用 cin.tie(0) 來解除自動清除緩衝區
- 需要注意的是，endl 也會清除緩衝區，需要將 endl 換成'\n' 才能夠有效的加速

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少
- 這是因為 `cin` 會自動將緩衝區清除，而 `scanf` 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字
- 但這對於演算法題目並不是非常必要
- 因此我們可以使用 `cin.tie(0)` 來解除自動清除緩衝區
- 需要注意的是，`endl` 也會清除緩衝區，需要將 `endl` 換成`'\n'` 才能夠有效的加速
- 另外，為了避免使用者與 `scanf, printf` 混用，C++ 會自動將兩種輸入方式同步，這也會造成延遲，但我們如果不會混用的話，就可以將其解除

IO 加速

- 若是使用 `cin, cout` 來輸入輸出，會發現他的速度比 `scanf, printf` 還要慢上不少
- 這是因為 `cin` 會自動將緩衝區清除，而 `scanf` 不會
- 緩衝區清除的用意是讓使用者可以馬上看到輸入的文字
- 但這對於演算法題目並不是非常必要
- 因此我們可以使用 `cin.tie(0)` 來解除自動清除緩衝區
- 需要注意的是，`endl` 也會清除緩衝區，需要將 `endl` 換成`'\n'` 才能夠有效的加速
- 另外，為了避免使用者與 `scanf, printf` 混用，C++ 會自動將兩種輸入方式同步，這也會造成延遲，但我們如果不會混用的話，就可以將其解除
- 解除方法: `ios::sync_with_stdio(0);`

define

你可能會在一些選手的程式碼看到這種東西：

```
#define fast ios::sync_with_stdio(0); cin.tie(0);
#define int long long
#define pii pair<int,int>
#define x first
#define y second
#define N 200015
```

- 這是 C++ 的好用功能
- 可以將 A define 成 B

define

你可能會在一些選手的程式碼看到這種東西：

```
#define fast ios::sync_with_stdio(0); cin.tie(0);
#define int long long
#define pii pair<int,int>
#define x first
#define y second
#define N 200015
```

- 這是 C++ 的好用功能
- 可以將 A define 成 B
- 舉例：#define int long long

define

你可能會在一些選手的程式碼看到這種東西：

```
#define fast ios::sync_with_stdio(0); cin.tie(0);
#define int long long
#define pii pair<int,int>
#define x first
#define y second
#define N 200015
```

- 這是 C++ 的好用功能
- 可以將 A define 成 B
- 舉例：#define int long long
- 可以有效的加快寫題的過程

define

你可能會在一些選手的程式碼看到這種東西：

```
#define fast ios::sync_with_stdio(0); cin.tie(0);
#define int long long
#define pii pair<int,int>
#define x first
#define y second
#define N 200015
```

- 這是 C++ 的好用功能
- 可以將 A define 成 B
- 舉例：#define int long long
- 可以有效的加快寫題的過程
- 講師們的範例程式碼許多都有將 int define 成 long long，自己在練習時要注意溢位的問題

全域變數

- 其實區域變數是有大小限制的

全域變數

- 其實區域變數是有大小限制的
- 因此若是建了一個比較大的表，就有可能會造成 RE (Runtime Error)

全域變數

- 其實區域變數是有大小限制的
- 因此若是建了一個比較大的表，就有可能會造成 RE (Runtime Error)
- 將一些系統輸入的變數、陣列或是額外建的表、資料結構等宣告在全域，就能夠避免空間太大而產生 RE 的情況

全域變數

- 其實區域變數是有大小限制的
- 因此若是建了一個比較大的表，就有可能會造成 RE (Runtime Error)
- 將一些系統輸入的變數、陣列或是額外建的表、資料結構等宣告在全域，就能夠避免空間太大而產生 RE 的情況
- 這在開發上可能是個不好的習慣，因此要用對地方

pass by reference

- 在一般的函數傳遞上，例如 f(a)，我們只是將一個值傳過去而已，這稱為 pass by value，若是在 f() 內更改 a，呼叫地的 a 是不會有變化的

pass by reference

- 在一般的函數傳遞上，例如 `f(a)`，我們只是將一個值傳過去而已，這稱為 `pass by value`，若是在 `f()` 內更改 `a`，呼叫地的 `a` 是不會有變化的
- 但如果我們想要能夠直接改到原本的值，就可以使用這個語法，可以想像成是他會將原始的東西傳過去，所以在更改的時候就是直接改呼叫地的值

pass by reference

- 在一般的函數傳遞上，例如 `f(a)`，我們只是將一個值傳過去而已，這稱為 `pass by value`，若是在 `f()` 內更改 `a`，呼叫地的 `a` 是不會有變化的
- 但如果我們想要能夠直接改到原本的值，就可以使用這個語法，可以想像成是他會將原始的東西傳過去，所以在更改的時候就是直接改呼叫地的值
- 在使用上，我們只需在函式宣告傳入值的變數前方加上一個 `&`，就可以成功達成 `pass by reference`

pass by reference

```cpp
void swap(int &x, int &y) {
    int z = x;
    x = y;
    y = z;
}

int main() {
    int a = 10, b = 20;
    swap(a, b);
    cout << a << " " << b << endl;
}
```

這個程式碼會輸出 20 10

range based for loop

- 寫過一些高階語言的話，可能會看到一些 `forEach` 或是 `for...in...` 之類的迴圈

range based for loop

- 寫過一些高階語言的話，可能會看到一些 forEach 或是 for...in... 之類的迴圈
- C++11 也加入了類似這種功能的語法，稱為 range based for loop

range based for loop

- 寫過一些高階語言的話，可能會看到一些 forEach 或是 for...in... 之類的迴圈
- C++11 也加入了類似這種功能的語法，稱為 range based for loop
- 如果我有一個整數陣列 arr，那麼我只要呼叫 for (int it: arr)，那麼他就會自動遍歷 arr 中的每個元素，並以 it 這個變數呈現

```
int arr[5] = {1, 2, 3, 4, 5}
for (int it: arr)
    cout << arr << " ";
```

output: 1 2 3 4 5

range based for loop with auto

- 如果我今天是一個 `struct` 陣列，而這個 `struct` 的名字很長，不想打那麼長怎麼辦？

range based for loop with auto

- 如果我今天是一個 struct 陣列，而這個 struct 的名字很長，不想打那麼長怎麼辦？
- 在 C++14 中，新增了 auto 這個語法，當有明確的初始值，auto 就會自動判斷它的型態並轉換成他

range based for loop with auto

- 如果我今天是一個 struct 陣列，而這個 struct 的名字很長，不想打那麼長怎麼辦？
- 在 C++14 中，新增了 auto 這個語法，當有明確的初始值，auto 就會自動判斷它的型態並轉換成他
- 所以我們可以直接使用 for (auto it: arr) 就可以完成整個陣列的遍歷了

```
int arr[5] = {1, 2, 3, 4, 5}
for (auto it: arr)
    cout << arr << " ";
```

output: 1 2 3 4 5

range based for loop with reference

- 然後你會發現，剛剛的寫法是 pass by value 的，沒有辦法更改到陣列中的內容
- 那我們就把他變成 pass by reference 就好啦

```
int arr[] = {1, 2, 3, 4, 5};

for (auto &it: arr)
    it++;
for (auto &it: arr)
    cout << it << " ";
```

output: 2 3 4 5 6

sort

- 當大家在寫 sort 的 compare 函式時，大概會寫一個 return a < b 之類的東西

sort

- 當大家在寫 sort 的 compare 函式時，大概會寫一個 return a < b 之類的東西
- 那如果寫的是 a $\leq$ b 呢？

sort

- 當大家在寫 sort 的 compare 函式時，大概會寫一個 return a < b 之類的東西
- 那如果寫的是 $a \leq b$ 呢？
- 如果你有一個長度足夠大、且每個元素都相同的陣列，並且在 compare 函式裡嘗試輸出每次傳入的值，你會發現他進入了無窮迴圈

sort

- 當大家在寫 sort 的 compare 函式時，大概會寫一個 return a < b 之類的東西
- 那如果寫的是 $a \leq b$ 呢？
- 如果你有一個長度足夠大、且每個元素都相同的陣列，並且在 compare 函式裡嘗試輸出每次傳入的值，你會發現他進入了無窮迴圈
- 為甚麼呢？

sort

- 當大家在寫 sort 的 compare 函式時，大概會寫一個 return a < b 之類的東西
- 那如果寫的是 $a \leq b$ 呢？
- 如果你有一個長度足夠大、且每個元素都相同的陣列，並且在 compare 函式裡嘗試輸出每次傳入的值，你會發現他進入了無窮迴圈
- 為甚麼呢？
- 嘗試在本機運行以下的程式

```cpp
int x[] = {1, 2, 3, 4, 5};

bool cmp(int a, int b) {
    cout << a << " " << b << endl;
    return a < b;
}
```

- 你會發現到，a, b 的前後順序竟然是相反的

sort

- 你會發現到，a, b 的前後順序竟然是相反的
- 這是因為，`sort` 他會先預設你要交換，然後再問你這樣對不對，如果不對（`false`）就換回原本的順序

sort

- 你會發現到，a, b 的前後順序竟然是相反的
- 這是因為，`sort` 他會先預設你要交換，然後再問你這樣對不對，如果不對 (`false`) 就換回原本的順序
- 如果你是用 $\leq$ 的話，遇到有相同的元素，系統每次詢問時你都跟他說這樣對，他就會一直換一直換，最後就會得到 RE

sort

- 你會發現到，a, b 的前後順序竟然是相反的
- 這是因為，sort 他會先預設你要交換，然後再問你這樣對不對，如果不對 (false) 就換回原本的順序
- 如果你是用 $\leq$ 的話，遇到有相同的元素，系統每次詢問時你都跟他說這樣對，他就會一直換一直換，最後就會得到 RE
- 因此 compare 函式是不能使用 $\leq$ 或是 $\geq$ 的

lambda

- 覺得每次 sort 都要寫 compare 函式很麻煩嗎？

lambda

- 覺得每次 sort 都要寫 compare 函式很麻煩嗎？
- C++14 提供了匿名函式 lambda 可以直接內嵌在 sort 呼叫裡

lambda

- 覺得每次 sort 都要寫 compare 函式很麻煩嗎？
- C++14 提供了匿名函式 lambda 可以直接內嵌在 sort 呼叫裡
- lambda 的用法非常多元，以下只展示最簡易的用法

```cpp
int arr[] = {1, 2, 3, 4, 5};

sort(arr, arr + 5, [](int a, int b) {
    return a < b;
});
```

自訂 operator

- 有時候我們自定義了一個 `struct`，並且想要讓他可以做一些相加的操作，我們可能會寫一個函式

自訂 operator

- 有時候我們自定義了一個 `struct`，並且想要讓他可以做一些相加的操作，我們可能會寫一個函式
- 但是要呼叫一個函式看起來就不漂亮

自訂 operator

- 有時候我們自定義了一個 `struct`，並且想要讓他可以做一些相加的操作，我們可能會寫一個函式
- 但是要呼叫一個函式看起來就不漂亮
- 所以我們可以自定義一個運算子，如此一來只要直接呼叫 `A+B` 就行了

自訂 operator

- 有時候我們自定義了一個 `struct`，並且想要讓他可以做一些相加的操作，我們可能會寫一個函式
- 但是要呼叫一個函式看起來就不漂亮
- 所以我們可以自定義一個運算子，如此一來只要直接呼叫 `A+B` 就行了
- 用法一樣超多，舉個簡單的例子

```
struct test {
    int a, b, c;
} A, B;

test operator +(test _a, test _b) {
    test res;
    res.a = _a.a + _b.a;
    res.b = _a.b + _b.b;
    return res;
}
```

stringstream

- 如果今天有未知的數字，該怎麼輸入呢？

stringstream

- 如果今天有未知的數字，該怎麼輸入呢？
- 我們可能會用個 `getline` 將整行讀進來，然後再呼叫函數將字串轉換為多個數字

stringstream

- 如果今天有未知的數字，該怎麼輸入呢？
- 我們可能會用個 getline 將整行讀進來，然後再呼叫函數將字串轉換為多個數字
- 但其實 C++ 有內建工具：字串串流 (stringstream)，使用需 include

sstream

```
string s
stringstream ss;
getline(cin, s);
ss << s;
int a;
while (ss >> a)
    cout << a << endl;
```

stringstream

- 如果今天有未知的數字，該怎麼輸入呢？
- 我們可能會用個 `getline` 將整行讀進來，然後再呼叫函數將字串轉換為多個數字
- 但其實 C++ 有內建工具：字串串流 (`stringstream`)，使用需 `include` `sstream`

```
string s
stringstream ss;
getline(cin, s);
ss << s;
int a;
while (ss >> a)
    cout << a << endl;
```

- 如此一來就可以將字串中的所有數字取出了

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化
  - `C++20 > C++17 > C++14`

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化
  - C++20 > C++17 > C++14
- 裁判機是以 32-bit 運行或是 64-bit 運行也是有影響的，位元數越高代表電腦可以同時處理越多事情

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化
  - C++20 > C++17 > C++14
- 裁判機是以 32-bit 運行或是 64-bit 運行也是有影響的，位元數越高代表電腦可以同時處理越多事情
  - 以 Codeforces 為例：GNU G++20 11.2.0 (64 bit, winlibs) > GNU G++17 7.3.0

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化
  - C++20 > C++17 > C++14
- 裁判機是以 32-bit 運行或是 64-bit 運行也是有影響的，位元數越高代表電腦可以同時處理越多事情
  - 以 Codeforces 為例：GNU G++20 11.2.0 (64 bit, winlibs) > GNU G++17 7.3.0
- 編譯器優化也有影響

編譯器對執行時間的影響

- 相信大家在一些 OJ 提交程式碼時都會看到一串編譯的指令，這對執行結果有什麼關係呢？
- 其實編譯器的參數會大大影響到執行的時間
- 例如更新的版本可能會將你的程式碼更加優化
  - C++20 > C++17 > C++14
- 裁判機是以 32-bit 運行或是 64-bit 運行也是有影響的，位元數越高代表電腦可以同時處理越多事情
  - 以 Codeforces 為例：GNU G++20 11.2.0 (64 bit, winlibs) > GNU G++17 7.3.0
- 編譯器優化也有影響
  - 以 O2 優化為例，他就會自動把你的程式碼做優化，讓他省去一些不必要的動作
  - 可以上 https://godbolt.org/ 這個網站嘗試，他會將你的 Code 轉成較接近機器可讀的組合語言 (assembly)

總結

總結

- 這堂課教到了許多演算法競賽的入門知識，有些是必要的，有些是非必要的，希望學員們能夠學到一些小技巧
- 希望學員們能夠對演算法競賽有更多的認識
- 演算法競賽中的某些習慣或是技巧並不是在每個地方都適合使用的，接下來的課程會教到更多高深技巧，這些都只是工具，希望大家都可以用在適合的地方，不要在不適合的地方使用