中學生學習程式解題之路

吳邦一

程設資結迷惘，算法遞迴如夢，
冷月豈可葬編程，柳絮靜待東風。

2024 年 11 月 16 日 (雄中線上)

講者(吳邦一)經歷：
(前)中正大學資工系教授
World’s Top 2% Scientists 2021 榜單
IOI出題作者，台灣代表隊帶隊老師
TOI培訓老師
NCPC/ICPC/全國高中程式賽出題裁判
AP325作者
Python-LeetCode 581系列筆記作者
多部APCS解題筆記與影片

Outlines

- 前言
  - 何謂程式解題(競技程式)
  - 確定短期目標 --- 學習的階段(基礎語法、基礎算法、選手初階、選手進階)
- 基礎語法階段
- 基礎算法階段 --- 裝備武器
  - APCS介紹與實作題五級所應具備的能力
  - 算法入門的ABCDE
- 解題的思路
  - 基本原則
  - 舉例說明
- 選手初階 --- 進階資料結構與演算法
- 其他
  - 關於APCS檢定與競賽的差異
  - 如何自我驗證程式

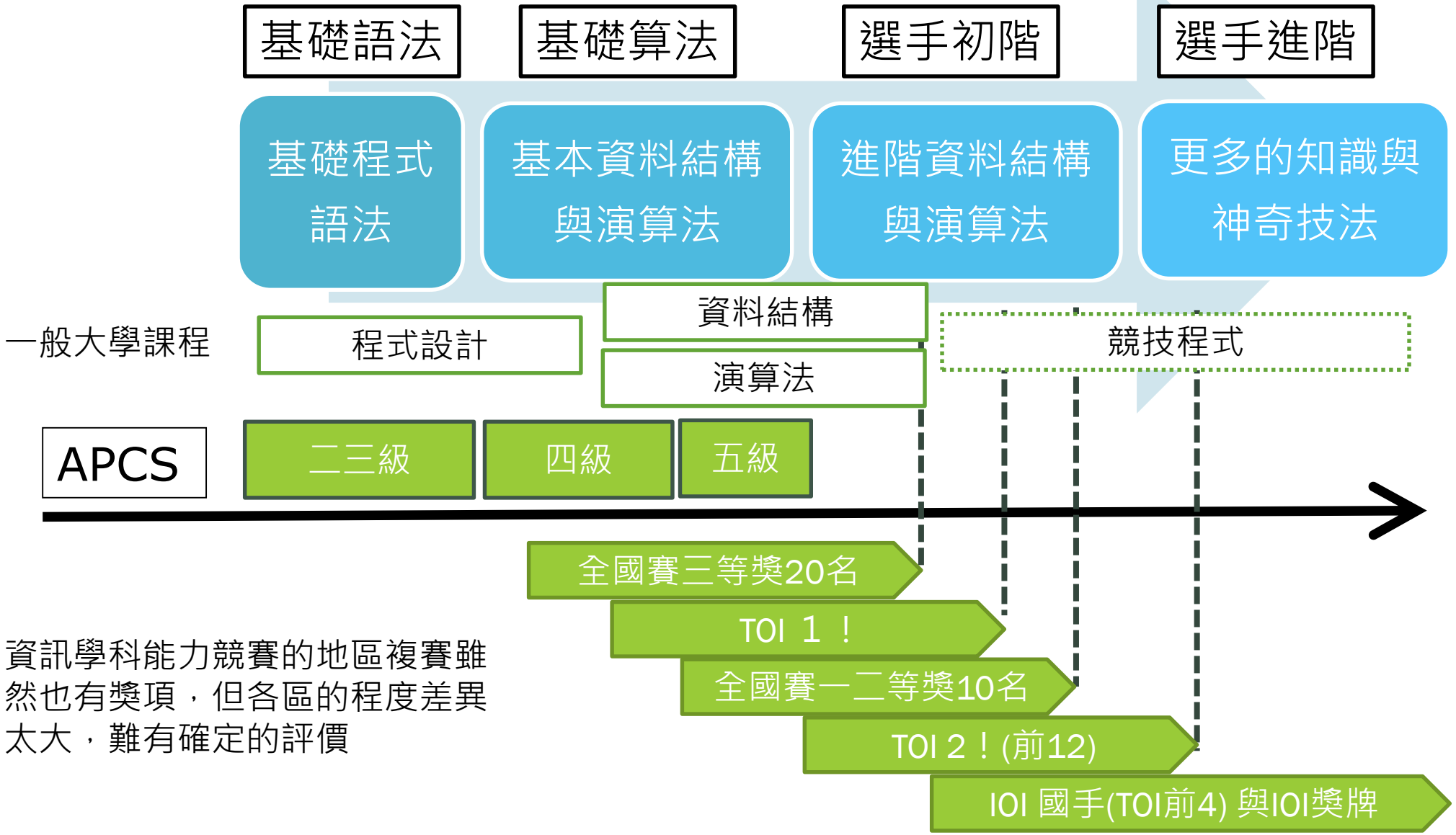
何謂程式解題(競技程式)

- 競技程式(competitive programming)是根據程式競賽而逐漸發展出來的一種程式設計領域，其內涵以程式設計、資料結構與演算法為主。有些人稱之為程式解題。
- 其解題的過程為：針對有明確輸入與答案的題目撰寫程式，裁判端將選手的程式進行編譯，餵入設計好的測試資料，以選手程式的輸出與執行時間來裁判其是否正確。
  - 在執行時間與記憶體使用限制下，使用合適的資料結構與演算法正確的計算出答案。
- 競賽與檢定的目的不同，各競賽/檢定的環境與難度也有差異，但大致上是競程的形式
  - 程式競賽：資訊學科能力競賽、TOI、IOI與ICPC(大學生的競賽)
  - 檢定： APCS實作題，LeetCode 科技公司面試模擬題的解題網站。

中學生重要競程比賽與檢定

- 資訊學科能力競賽
  - 校內初賽
  - 地區複賽(10~11月)
  - 全國決賽(12月)
- TOI(資奧培訓營)
  - 一階段(1!)
  - 二階選拔、國手選拔、IOI國際大賽
- NPSC (2024暫停，原因不明)
  - 台大主辦(11~12月)
  - 國高中可參賽
- 少年圖靈賽
  - 精誠公司舉辦
- APCS檢定
  - 1、6、10月，一年3次
- CPE(大學生程式檢定)
  - 每年四次
  - 非大學生繳費
  - UVA (ICPC)考古題(英文)
- 成功大學暑期高中生程式設計邀請賽

學習的階段與短期目標



資訊學科能力競賽的地區複賽雖然也有獎項，但各區的程度差異太大，難有確定的評價

基礎語法階段

- C++ or Python
  - 如果以競程為目標，還是選擇C++
  - 如果未必要走競程，可以考慮選擇Python，一開始會比較容易
- 選擇語言以及一開始的學習工具(IDE)時，其實不必太過糾結，因為即使將來更換，之前的學習經驗並不會白費
- 如果在學校，老師教哪一種就跟著學哪一種，是比較輕鬆的選擇。
- 以不涉及物件導向的C++是競程最方便的入門款，也就是幾乎是C的語法的C++。
- 在語法入門階段的教材都差不多，找一本基礎的書或網路教材，跟著學，但記得多做題目。例如，劉邦鋒教授所著「由片語學習C程式設計」

基礎語法階段

- 該學會的重點
  - 程式的基本原理
  - 變數的使用與基本資料型態(整數、浮點數、字元與字串)
  - 流程控制：循序式、條件分支if、迴圈、與副程式(函數)呼叫
  - 一維與二維陣列的使用
- 多做題目的重要性
  - 可以熟悉許多常用的形式，這些將來會像片語一樣內化為你的能力，而不是只有單字(指令)。例如，
    - 在陣列(或輸入)中找到滿足條件的最小值
    - 操作陣列資料，例如反轉或合併兩個sorted list (merge)
    - 在四四方方的格子裡面爬呀爬呀爬(方格圖走訪模擬)

基礎語法階段

- 應達成目標：看到簡單的題目(操作模擬類型的題目)會構思程式怎麼寫，遇到bug會知道如何找錯誤
  - debug工具是可以學，不過一開始程式的都很小，會不會用工具並非最重要的事。除錯的觀念與自己的習慣才是最要緊的。
  - 除錯需要測資，寫程式要先會在本機編譯除錯，不是都往online judge上丟。
- 排除故障的基本原則(不只是程式)：
  - 錯誤重現與故障隔離
  - print大法：重要的地方印出一些關鍵的變數或訊息就可以找到錯誤
- DeBIG。程式並非要追求簡短，而是邏輯清楚最重要。但是很多初學者會寫出許多無意義而冗長的程式碼。這些通常是在當下一面思考一面寫一面改而造成。這是沒有關係，但寫完之後，應該再看看是否可以有改善之處，將來才會進步

基礎語法階段 – 避免犯錯

- 程式風格：好的風格習慣減少犯錯的機會，例如
  - 初值設定放在迴圈入口處而不要放在很遠的地方。
  - 變數的命名不要用的很混亂，而是維持自己的習慣。
  - 良好的縮排(自己的習慣)

```
int sum=0,i,n,a[10],max=-1; // 第二筆的初值錯了，很難抓
while (1) {
cin >> n;
if (n==0) break;
for (i=0;i<n;i++) cin>>a[i];
for (i=0;i<n;i++) // 縮排誤導範圍
    if (a[i]>max) // 變數名稱衝內建函數
        max=a[i];
    sum=sum+a[i];
    cout << max << '\n' << sum<<'\n';
}
```

```
// Better style
int i,n,a[10];
while (1) {
    cin >> n;
    if (n==0) break; // 不換行可沒有括號
    for (i=0; i<n; i++) cin>>a[i];
    int isum=0, imax=-1; // 迴圈變數初值靠近起點，避免名字衝突
    for (i=0; i<n; i++) {
        if (a[i] > imax) { // 如果換行一律加括號
            imax = a[i];
        }
        isum += a[i];
    }
    cout << imax << '\n' << isum<<'\n';
}
```

清楚的邏輯與結構 --- 迴圈不變性

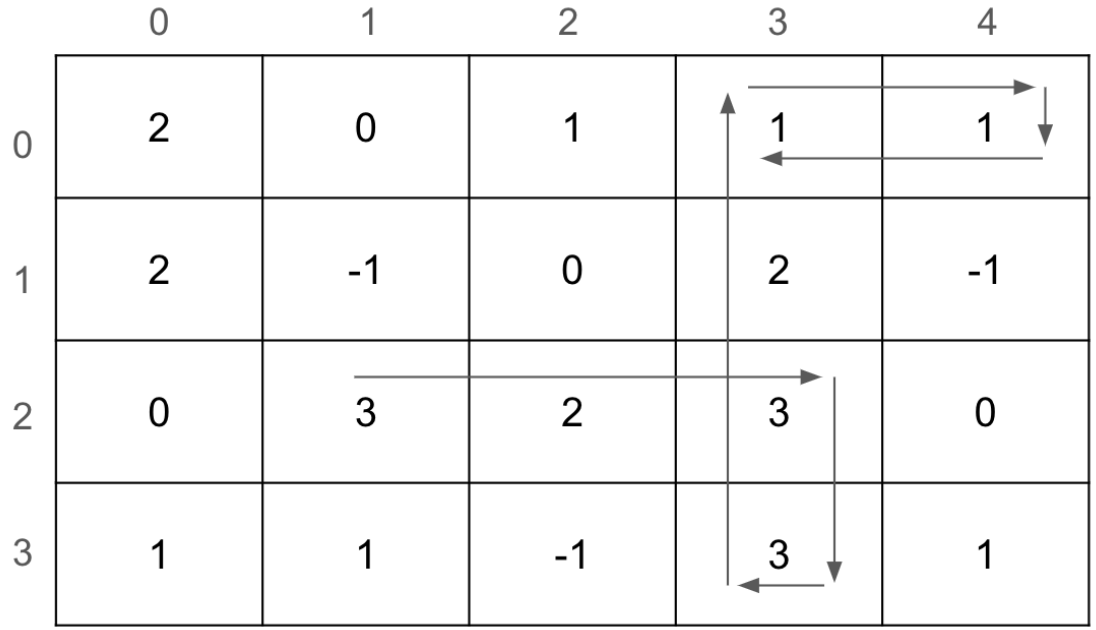
- 程式邏輯與結構紊亂，自己也不知道是否正確
  - ```
    int imin = 100; // assume a[i]<=100
    for (i=0;i<n;i++) imin = min(imin, a[i]);
    ```
- 為什麼這個程式是對的，imin可找出最小值？
  - 因為老師(書上)這樣寫 (X)
  - 因為大家都這樣寫 (X)
- 迴圈不變性：在一個迴圈中保持不變的特性
  - imin始終是前i個的最小值
  - 根據數學歸納法(1.初始成立，2.若第i個成立則第i+1個成立)，這個程式是對的。
- 其實每個程式的正確性都是這麼來的，使用到的歸納法有兩種：迴圈或是遞迴
    ```

清楚的邏輯與結構 --- 迴圈不變性

- 一個程式可以分成若干區塊，有一個主要流程(演算法)，然後就是區分成若干區塊。每一個區塊抓住那個我們需要的迴圈不變性，就很容易逐步完成，而且會是個邏輯清楚的程式。
- 也可以換一種說法。我們定義好每一個變數的意義，用程式維護好它的意義，這些意義就是迴圈不變性。
- 合併兩個sorted array a and b，假設長度是na, nb。
  - ```
    int c[100], nc=0; // 以c來放合併後的陣列，nc是c的目前長度
    int i=0, j=0; // 兩個陣列目前處理到的位置，之前的已經併入c
    while (i<na && j<nb) { // 兩個皆未走完
        if (a[i]<b[j]) c[nc++] = a[i++]; //兩者比較小的放進去
        else c[nc++] = b[j++];
    } // 某個陣列尚未處理完畢
    while (i<na) c[nc++] = a[i++];
    while (j<nb) c[nc++] = b[j++];
    ```
    ```

- 這個程式為什麼是對的？
- 迴圈不變性：
  - a陣列中ai之前的都已經併入
  - b陣列中bi之前的都已經併入
  - 每回合放入c中的必定是剩下的最小值
    - 放入的是min(a[i], b[j])
    - 因為兩陣列是sorted，根據遞移性

模擬操作題的範例

蒐集寶石 (APCS 202410Q2 ZeroJudge 0712)

- 模擬在一個$m \times n$的方格圖中行走，碰到寶石就撿起一顆，碰到牆壁或是界外就右轉，同時要記錄一個分數，每次走到有寶石的格子，得分就增加該格目前寶石數量，如果目前總得分是k的倍數，要右轉。走到沒有寶石的格子就結束。
- 解題思考：
  - 題目有那些資料：m, n, g陣列內容
  - 要記錄那些變數(狀態)
    - 目前的位置
    - 目前的方向
    - 得到的寶石數量
    - 分數



- 方格圖走訪的常用小技巧
  - 用整數d當作目前的方向。以0,1,2,3分別當做東南西北，這樣有個好處，右轉就是+1再除以4取餘數，如果在其他題目，也可以用+3來當作左轉，+2就是後轉(都是除4的餘數)。
  - 可以定義 int dr[4]={0,1,0,-1}, dc[4]={1,0,-1,0}; 為四個方向的列差與行差。
    - 在d方向走一步的座標就是(r+dr[d], c+dc[d])，這樣可以省略用四個if去判斷
  - 某些題目可以在四周圍一圈牆壁(不可走的值)，這樣可以省略出界的判斷。
    - 否則通常要寫 if (r>=0 && r<m && c>=0 && c<n && g[r][c]!=XXX)
- 看本題的例子

```
6       int g[102][102],dr[4]={0,1,0,-1}, dc[4]={1,0,-1,0};
7       scanf("%d%d%d%d%d",&m,&n,&k,&r,&c);
8       for (i=1;i<=m;i++) for (j=1;j<=n;j++) scanf("%d",&g[i][j]);
9       m += 2; n += 2; r++; c++; // surrounding -1 as wall
10      for (i=0;i<m;i++) g[i][0] = g[i][n-1] = -1;
11      for (j=0;j<n;j++) g[0][j] = g[m-1][j] = -1;
12      int d=0, score=0, cnt = 0;
13      while (g[r][c]>0) { // until 0
14          cnt++;
15          score += g[r][c]; // update score
16          g[r][c]--;
17          if (score%k == 0) d=(d+1)%4; // turn right if multiple of k
18          while (1) { // turn right until not wall
19              int nr=r+dr[d],nc=c+dc[d]; // next row, next column
20              if (g[nr][nc]>=0) { // not wall
21                  r=nr; c = nc;
22                  break;
23              }
24              d = (d+1)%4; // next dir
25          }
26      }
27      printf("%d\n",cnt);
28      return 0;
```

網路上某個程式(有錯誤)

```
lead = [
        [-1] * (N + 2),
        [-1] * (N + 2)
]

for i in range(M):
    lis = (list(map(int,input().split())))
    lis = [-1] + lis + [-1]
    lead.insert(-1,lis)
'''
處理二維list
s
'''
f = 0
score = 0
r+=1
c+=1
score += lead[r][c]
lead[r][c] -= 1
g = 1
def turn(f):
    f = (f+1)%4
    if f == 0 and lead[r][c+1] == -1:
        f = (f+1)%4
    if f == 1 and lead[r+1][c] == -1:
        f = (f+1)%4
    if f == 2 and lead[r][c-1] == -1:
        f = (f+1)%4
    if f == 3 and lead[r-1][c] == -1:
        f = (f+1)%4
    return f
```

```
while True:
    if score % K ==0:
            f = turn(f)
    if f == 0 and lead[r][c+1] != -1:
        if lead[r][c+1] == 0:
            break
        c += 1
        score += lead[r][c]
        g+=1

    elif f == 1 and lead[r+1][c] != -1:
        if lead[r+1][c] == 0:
            break
        r += 1
        score += lead[r][c]
        g+=1


    elif f == 2 and lead[r][c-1] != -1:
        if lead[r][c-1] == 0:
            break
        c -= 1
        score += lead[r][c]
        g+=1

    elif f == 3 and lead[r-1][c] != -1:
        if lead[r-1][c] == 0:
            break
        r -= 1
        score += lead[r][c]
        g+=1
    else:
        f = turn(f)
    lead[r][c] -= 1
print(g)
```

```python
m,n,k,r,c = [int(x) for x in input().split()]
g = []
for i in range(m):
    g.append([int(x) for x in input().split()]+[-1])
g.append([-1]*n) # surrounding -1
d = 0 # current direction
score = 0 # current score
cnt = 0 # number of collected stone
dr = (0,1,0,-1) # east,south,west,north
dc = (1,0,-1,0) # row and col difference
while g[r][c]: # until 0
    cnt += 1 # num of collected
    score += g[r][c] # update score
    g[r][c] -= 1 # update g
    if score%k==0: d = (d+1)%4 # multiple, turn right
    while g[r+dr[d]][c+dc[d]] < 0: # until not wall
        d = (d+1)%4 # turn right
    r += dr[d]; c += dc[d]; # step forward
#
print(cnt)
```

基礎語法階段

- 初學者找不到的錯誤：
  - overflow（資結算法一場夢，不開long long見祖宗）
  - 陣列超界
  - 主要原因就是他在小資料時可能並不發生
- 不變的心法
  - 學而不思則惘，思而不學則殆
  - 要去思考，也要學習別人的程式寫法與經驗

如何達到神乎其技(神之一招)

少了佐為就沒有棋靈王，沒有塔矢亮也成就不了進藤光

對手跟隊友一樣重要

成功更重要的要素 -- 動機與發願

- 教練！我想打籃球

你寫的程式是茅還是矛？你以為作對的題目，可能不符合執行時間的限制

基礎算法：裝備武器 (基礎資料結構與演算法)

基礎算法

- 從語法階段進入算法階段，最重要的是寫出效率高的程式
  - 所謂效率通常指時間與空間，也就是程式的執行時間與記憶體用量
- 學習基礎算法的知識與技能
  - 了解時間複雜度：
    - 你寫的程式是茅還是矛？你以為作對的題目，可能不符合時間限制
    - 清楚認知題目所要求的時間複雜度，不符合複雜度要求的解法，沒有僥倖
  - 學習基本資料結構與演算法策略
- 其它可以提升能力的學習活動
  - 找適合的練功場，多做題目
  - 參加好的營隊(台清交的學生有辦一些營隊，收費不多，有些甚至幾乎免費)，可以學到東西也找到同好。

基礎算法 ---時間複雜度

- 時間複雜度其實不好算，好在大部分的狀況做個粗估並不難
  - 只算最高項次，忽略常數
  - 迴圈就相乘，想加就取大的
- 對中學生比較難的是遞迴的複雜度。
  - 常見的就了解一下，其他的就暫時忘了它
  - 分治的複雜度(參見AP325)
  - 子集合與基本排列組合
- 複雜度跟一些數學有關
  - 排列組合
  - 算術級數、幾何級數(等比)、調和級數
  - 1/2+1/3+1/4+…+1/n = O(logn)，有些中學生不知? f(x)=1/x的積分

基礎算法---時間複雜度

- 均攤分析(amortized analysis)
  - 雙迴圈時算總帳而非單純相乘
  - 在堆疊與滑動視窗類的題目中經常出現
- 複雜度不是只有事後分析而是會影響設計算法
  - 被外表蒙蔽，不知道它的好，就不會追求它
- 例如這個質數篩法找出1e5以內的質數
- ```
  n = 10**5
  prime = [True]*n # ignore [0,1]
  for i in range(2,n):
      if not prime[i]: continue
      for j in range(i+i,n,i): # mark multiple of i
          prime[j] = False
  ```
- 時間複雜度其實不壞於O(nlogn)，因為調和級數的和。
  (實際更小，並不是每個i都會跑內迴圈)
  ```

基礎算法：先來看看APCS實作題考什麼

APCS實作題檢測內容

- 實作題為競程形式，但為賽後裁判而非online judge
- 4 題400分，有子題，共計 150 分鐘，通常每題有20筆測資，每筆測資

程式設計觀念題		程式設計實作題		
級別	原始總分範圍	級別	原始總分範圍	說明
五	90 ~ 100	五	350 ~ 400	具備常見資料結構與基礎演算程序運用能力
四	70 ~ 89	四	250 ~ 349	具備程式設計與基礎資料結構運用能力
三	50 ~ 69	三	150 ~ 249	具備基礎程式設計與基礎資料結構運用能力
二	30 ~ 49	二	50 ~ 149	具備基礎程式設計能力
一	0 ~ 29	一	0 ~ 49	尚未具備基礎程式設計能力

APCS實作題的內容

- 雖然每次的題目都不一樣(AP沒有出過一模一樣的考古題)，但內涵幾乎固定
  - 第一題：基本的輸入輸出、運算、判別式與迴圈。第1子題60分(二級分)不須迴圈，第2子題要迴圈
  - 第二題：簡單的陣列運用，通常一維50分二維50分。
  - 第三題：遞迴、排序與二分搜、stack/queue以及BFS/DFS
  - 第四題：貪心、DP、Sliding window、分治、tree以及topological sort。
- 第三第四題或許混合，但內容不脫此範圍。
- APCS的範圍僅為常見資料結構，所以不使用進階資料結構應該都可以解，但有些題目用進階資料結構可能比較好寫。
- Python都可以通過，但是Python要達到四五級還是有其不利的因素(雖然可能有較寬鬆的時間限制)，例如少了編譯器優化
- APCS與程式競賽的題型是一樣的，但目的與檢測方式不同，所以略有差異：APCS的題目通常比較直接，不會有競賽中那些要思考很久的題目。

APCS學習的書籍與資源

- 網路上有很多教材資源，但範圍多半以競程為目標
  - 因此對APCS來說，內容太多
  - 這是我寫AP325的動機，以APCS為範圍，也當作競技初步
- 初學者推薦兩本免費的書
  - AP325（APCS範圍）
  - Competitive Programmer’s Handbook（競程範圍）
    - 一位芬蘭的IOI教練 Antti Laaksonen 所著
    - 上網找CSES，有題目以及一本免費的書
- 網站資源：APCS官網與競程的網路資源
  - 考古題，黃惟(前新竹女中學生)整理的學習筆記 YUI HUANG 演算法學習筆記
  - 最近每次考完後都有人將題目蒐集在ZeroJudge上

AP325講義

- 325意為3-to-5，這是一份為了三級分的人所撰寫的免費講義，依照程式技巧區分為0~8章，除了文字講解外，內含例題與習題共121題，目前剛好325頁，另外含題目與測資。
  - 包含多次APCS過去考題的第三第四題。
  - 每題附測資與答案，可以自己練習。有一些高中有將題目放在online judge，方便練習。包括台中一中、惠文、彰化精誠、景美、花蓮、彰化。
- 相關資訊可以到FB「APCS實作題檢測」社團查詢。
- 搜尋「吳邦一的APCS題解目錄」，可以找到我寫過的解題筆記與解題影片

AP325講義內容

- 預備知識
- 遞迴
- 排序與搜尋
- 佇列與堆疊、滑動視窗
- 貪心與掃描線演算法
- 分治
- 動態規劃
- 基本圖論
- 樹

基礎資結與算法

- 資結與算法提升時間效率的根本原理
  - 避免重複的計算
    - 安排計算的流程與紀錄(空間換取時間)
  - 避免不必要的計算
    - 靠性質、規律、數學
- 擁有不同資料結構會採取不同的算法，擁有不同的交通工具會採取不同的路線。
  - 配備的更強大的資料結構可能採取更直接的計算思考流程，逢山開路，遇水搭橋，當然題目就變得更簡單。

算法入門的ABCDE

- 五個包含算法基本原理的知識與技巧，如果能夠理解，大致表示適合學習資結與算法。
  - 複雜度分析(complexity Analysis)
  - 二分搜(Binary search)
  - 離散化(coordinate Compression)
  - 差分與前綴和(Difference and prefix sum)
  - 快速冪(binary Exponentiation)(倍增法)

二分搜

- 最簡單的形式，終極密碼：在某個整數範圍內最快的猜到某個數字。
- 在一個排好序的數列中，找到等於(或最接近)某個數字的位置。
- 其實二分搜還有很多形式。
  - 數值方法解函數的根
  - 對答案二分搜
  - 二分檢測
- 二分搜的基本運用原則：在一個序列中搜尋第一個滿足某命題的位置，而此序列對該命題滿足單調性。
  - 所謂單調性就是在序列某個點之前都是False，而其後則都是True。
- 效率來源則是：一個檢測之後就可以知道有一半的資料不必計算，而能夠節省這些不必要的計算，原因來自於單調性(性質)。

離散化(座標壓縮)

- 將一序列的數字轉換對應到[0,k]，保持數字的大小關係。也就是找出每個數字的rank(第幾小)
- 有很多做法，用字典(C++ map, hash)會容易些
- 不使用字典時
  - 排序
  - 移除相同(找出rank)
  - 將序列中每個數字用他的rank替換(二分搜)
- 離散化的重要性
  - 座標壓縮是很多方法需要搭配的前處理(如BIT、某些DP、某些線段樹)
  - 他的做法是處理陣列資料很好的習題

差分與前綴和

- 差分與前綴和就是離散版的微分與積分
  - 數序p[0], p[1], p[2]…
  - 差分d[0]=p[0], d[i] = d[i]-d[i-1] for i>0
  - d做前綴和就是p，sum(d[0]+d[1]+…d[i]) = p[i]
- 前綴和最直接的應用就是計算區間和，可以用一個減法計算出任意區間的總和p[i]-p[j-1] = d[j]+d[j+1]…+d[i]
  - 也有二維前綴和
  - 套用在很多問題上(DP, greedy)
- 很多問題做了差分轉換後就變得簡單許多

- max subarray(陣列中找總和最大的subarray)，如果把數字做前綴和，等價於先買後賣的一次買賣的最大獲利。後者的解法顯而易見：對每一個點找他之前的最小值，歷遍時O(n)維護好prefix minimum
  - ```
    pmin = p[0]; best = 0;
    for (i=1; i<n; i++) {
        best = max(best, p[i]-pmin);
        pmin = min(pmin, p[i]);
    }
    ```
- 區間修改(將一段連續區間每個就加上x)在差分轉換後變成兩個單點修改
  - 若[i,j]要+x => 在起點d[i] += x，在終點下一個扣回來 d[j+1] -= x
  - 經典題：數線上有很多線段，求線段聯集長度。或者各有厚度，求最厚的地方(一根針插某個位置，最多可以插到幾個線段(或多厚的線段)。
    - 差分轉換後，紀錄差分，最後由前往後求前綴和即可。(可搭配離散化)
    ```

快速冪(binary exponentiation)

- 問題很簡單，對於兩個非負整數x, y，要計算x^y。(有時要mod m)。
  - 直接的方法是從1開始，跑一個迴圈把x乘y次，但這樣要花y次乘法。
  - 也用在矩陣，矩陣的快速冪是某種DP的優化方式。
- 從英文的名字可以了解作法。我們把y以二進位表示。例如
  $y = 37 = (100101)_2 = 1*32+0*16+0*8+1*4+0*2+1*1$
  - 所以$x^{37} = x^{32} * x^4 * x^1$
  - 如果我們把x^1, x^2, x^4, x^8, x^{16}, x^{32},…這些都先準備好，那麼只要log(y)次的乘法就可以算出。而且，算出那些預備的冪次，也只需要log(y)次乘法(倍增法)。
- 可以從preprocessing的觀點看。很多可以先算出一些中間結果，在拿這些暫存結果去產生我們要的答案。

- 快速冪的遞迴版本也是理解遞迴的好例子
  - ```
    int exp(int x,int y) {
        if (y==0) return 1; // 遞迴終點
        if (y&1) return exp(x,y-1)*x; // 奇數，求y-1次再乘一次
        int t = exp(x,y/2); // 偶數，先求一半再自乘
        return t*t;
        // return exp(x,y/2)*exp(x,y/2) is BAD
    }
    ```
- 這速度是log(y)
- 註解中那個偶數時的寫法是壞的，但可以拿來說明memoization的例子(top-down DP)
- 迴圈版這裡就不演示了
    ```

先看一些基本原則，再來舉例說明

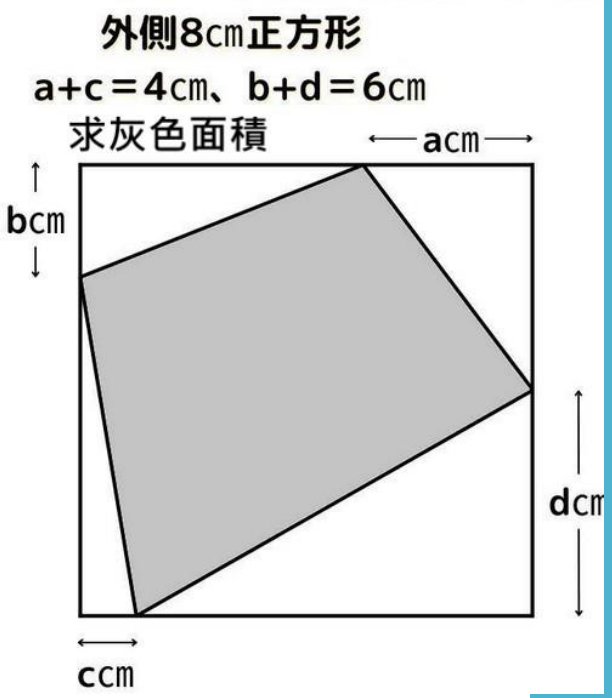
解題的思路

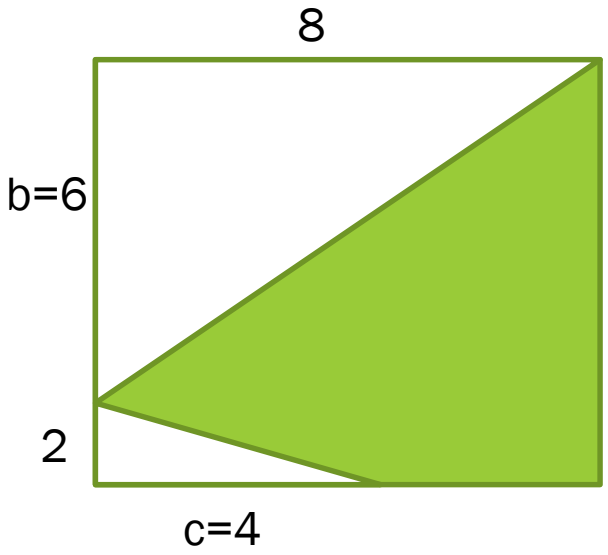
解題的思路

- 解題者面對題目的最大優勢：
  「題目必然存在一個合理的解。」
- 這看似廢話，其實是很重要的。想想看很多選擇題可以用代入答案或嘗試答案的方法得到正解。
- 不只有在程式解題用得到，看一個數學的例子

題目必然存在一個合理的解

- 假設答案是存在的，那麼可以在題目條件下做任何假設。
- 看看這個有趣的題目，這題或許有很多做法。
- 如果這題的答案是存在的，也就是在題目給兩個的條件下，無論a, b, c, d是多少，算出來的答案都應該是一樣的
- 我們可以找一個最簡單的答案。例如，設a=d=0，那答案就是 8×8−6×8÷2−2×4÷2=64−24−4=36。





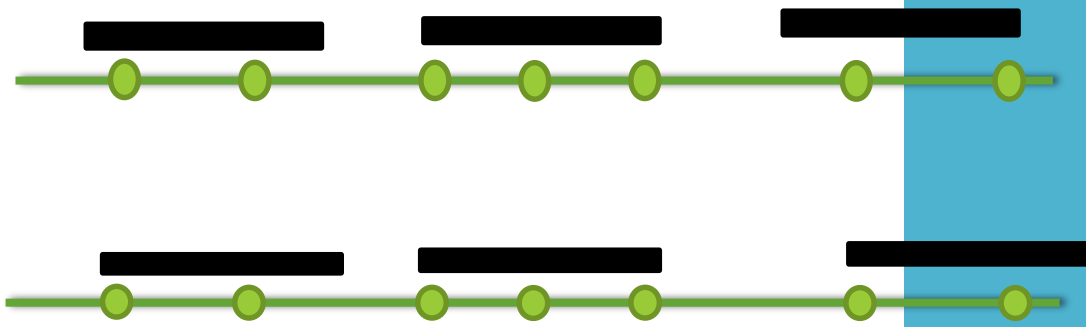
解題的思路：答案必然存在

- 在分析題目的找解法的過程中
  - 每一個題目必然存在合理的解法
  - 面對一個想不出來的問題時，要嘛它本來就超過了你的知識範圍，否則就是你忽略了什麼或者看錯了什麼。
    - 題外話，把設計給人考的數學考卷拿給電腦考，有的時候是沒意義的，電腦很多時候不是用人的推理方法解數學題。
  - 把時間與精力放在找到一個自己可能會的解法
- 所謂的合理，包含滿足執行時間與記憶體限制，以及使用到的技術、難度與程式碼的長度
  - IOI是有syllabus的，有些東西不能考。雖然國內高中比賽偶而未遵循。
  - 合理的難度是比較難定義的，在一些大的比賽中，偶有超難被認為不可做的題目(尤其在大專的比賽)，有些比賽希望選手不要破台。

解題的思路：答案必然存在

- 在面對最佳化的題目，我們常常要去分析最佳解的長相，以便找到性質來設計算法。
- 一筆輸入測資，往往也是存在很多個最佳解，我們就去找最佳解中最簡單的那個。
- 舉一個簡單的例子：
  給數線上N個點的座標，以長度R的線段要蓋住所有的點，最小需要幾根相同長度的線段？
- 假設最佳解的答案是是K根線段。以K根長度R的線段蓋住所有點的方法可能有很多，他們的答案都是一樣(K根)，我們去找其中比較簡單的，有什麼性質

- 從貪心的觀點出發，盡量不要浪費
- 必然存在一個最佳解，第一根的左端對其齊最左方的點
- 我們可以用一個線段，左端對其目前尚未蓋住的最左邊的點，把它蓋住的點刪除。重複這個步驟就可以算出最少的線段數
  - 剩下的是實現這個算法的問題



```
num_seg = 0;
last = -oo; // last covered position
sort(p, p+n); // check points left to right
for (int i=0; i<n; i++) {
    if (p[i] <= last) continue; // covered
    num_seg++; // using a new segment
    last = p[i]+r; // left on p[i]
}
```

解題的思路

- 看到一個題時，當然是先理解題目的要求，包含
  - 題意(要算什麼)
  - 輸入與輸出(輸入輸出格式與範圍)
  - 時間複雜度(根據參數範圍與時間限制判斷)
- 輸入輸出與時間複雜度要當成題目的一部份。即使完全一樣的題意，輸入與時間的要求不同，有時解法完全不同。
- 如果曾經看過或似曾相似
  - 當然先想看看這個解(這個誰都知道呀)
- 假設(對你來說)是一個比較需要思考的題目
  - 每一個題目背後必然存在一個合理的解法。
  - 從你擁有的武器著手

解題的思路

- 題目雖然有可能有很多不同樣貌，先不要管很難的題目，大多數題目是容易歸類的
  - 資料區分：一群數字(順序無關)，數列(順序有關)、字串、圖(二維陣列或方格圖)、樹
  - 採用策略：遞迴與暴搜、排序與搜尋、堆疊佇列(滑動視窗)、貪心或掃描、DP、圖形走訪、以及樹的問題。
- 大部分(遞迴與暴搜、圖形走訪、樹)的問題都很明顯。
- 排序與搜尋常是搭配在其他問題中使用，單獨出現的問題也容易辨識。
- 剩下的(堆疊佇列(滑動視窗) 、貪心或掃描、DP)是可能比較混雜。

解題的思路 -- 遞迴暴搜

- 遞迴通常用在兩個地方
  - 為了計算答案，以遞迴來進行窮舉暴搜。
  - 根據題目定義的遞迴結構來實作。
- 暴搜通常是很簡單或是很難的問題（蛤？），APCS似乎還沒有考過，只有某些子題是遞迴解。競賽有可能出現，通常不是難題。
- 從複雜度很容易看出
  - $O(n), O(n^2), O(n^3)$的枚舉通常是最直接的方法
    - APCS的第一二題通常根本不需要考慮複雜度的問題
  - 其他的都是指數複雜度的$O(2^n)$、$O(3^n)$、$O(n!)$。
- 簡單的很看容易出來，例如子集合、單純的排列或組合
- 很難是指有些遞迴的複雜度是很難估算的，或者超過高中的範圍(它是個差分方程的解)，或者超過人類目前的範圍(一些帶有剪枝的方法)

- 另一方面是不少人對遞迴不熟悉不了解
  - 很多學生不會做用遞迴暴搜所有子集合
- 比較難的是折半枚舉(meet at middle)，但這一類的題目比較固定，也容易判斷出來
- n=10大概是排列、n=20~25就是暴子集合、n=40~50是折半暴子集合

解題的思路－排序與搜尋

- 排序通常有幾個運用的時機：
  - 需要把相同資料排在一起
  - 便於快速搜尋
  - 做為其他演算法的執行順序，例如Sliding window, Sweep-line, Greedy, DP。
- 搜尋的方法
  - 線性：一個一個找
  - 二分搜：有序資料的快速搜尋
  - hash(字典)：只能用於exact match，APCS與高中通常沒有非用不可的題目。但會用的話很多題目確實變得比較簡單。

解題的思路 –堆疊佇列(滑動視窗)

- 堆疊與佇列有些題目不易辨識，例如單調隊列(monotonic queue/stack)，但這一類題目有限，樣貌類似
- 滑動視窗的題目容易辨識
  - 在一個序列中，找某些區間，滿足某些條件，條件具備單調性，例如區間愈大其值愈大
  - 解題的重點在每次變動一端時，調整而不要重新計算

解題的思路 – 貪心或掃描、DP

- 這兩個大概是最難區分的
- 貪心需要證明，但考試與比賽時未必有此時間，online judge時，有很多人其實是用猜的
- 貪心算法由一連串決定組成，每次考慮當前最好決定，選擇後就不再更改。但所謂的貪心要看你所訂的決策函數。
- 有個小訣竅可以判斷貪心不成立。
  - 如果多個可以挑選且其值相同，但又不能隨意挑，這個貪心策略不成立。
  - 舉例來說，數線上挑選最多不重疊線段(區間)。如果每次挑最短的，很容易發現是錯的，[1,2],[2,3],[3,4]，三個長度一樣，先挑中間就錯了

- DP是最困難的，因為樣子多，變化也多，可以合併很多技巧。而且有很多都是DP的優化的技術。
- APCS不至於考到很難的DP或複雜的優化。
- 基本形式的DP並不是特別難，其樣貌也可以大致歸類。
- 競賽的難題就沒有甚麼特別的訣竅，只能多學多看。

逆向思考與整體方向思考

- 大部分前面提到的技巧，大多是直線的思考(要找什麼就算什麼)。事實上這也是大多數人會採取也比較簡單的思考模式。
- DP很多時候從逆向思考比較簡單，遞迴的思考方式，實作時有順向(bottom-up)與逆向(top-down)兩種
- 分治與DP類似都是遞迴思維出發，也就是逆向思考。分治除了在樹狀圖之外，序列的分治題目大多時候都存在DP的解，但通常需要搭配較複雜的資料結構。
- 其他非直線思考的是
  - 「對答案二分搜」
  - 改變計算的順序，例如題目是一一切斷，計算時逆轉順序變成一一接合。
  - 非在線查詢的題目可以改變計算順序

解題的思路 --- 舉例說明

Q-2-7. 互補團隊 (APCS201906)

- 前 *m* 個英文大寫字母每個代表一個人物，以一個字串表示一個團隊，字串由前 *m* 個英文大寫字母組成，不計順序也不管是否重複出現，有出現的字母表示該人物出現在團隊中。兩個團隊**沒有相同的成員而且聯集起來是所有*m* 個人物**，則這兩個團隊稱為「互補團隊」。輸入 *m* 以及*n* 個團隊，請計算有幾對是互補團隊。我們假設沒有兩個相同的團隊。
- Time limit: 1秒
- 輸入格式：第一行是兩個整數 m 與 n，2 ≤ m ≤ 26，1 ≤ n ≤ 50000。第二行開始有 n 行，每行一個字串代表一個團隊，每個字串的長度不超過100。
- 輸出格式：輸出有多少對互補團隊。

- 範例輸入：

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

- 範例輸出：

```
2
```

思考

- 一個字串的字母順序與是否字元出現次數無關，所以是一個集合
- 要找互補的集合有幾對
- 複雜度要求，5萬個中找配對，O(n^2)一定不行，要O(nlogn)以內的方法
- 如何表示一個集合
  - 利用位元運算把每一個字串集合轉換成一個整數。這是一個常用方式
    - bit 0表示A在不在，bit 1表示B，…
- 把每一個集合都化成整數後，如何找互補?
  - 兩集合互補就是代表的數字位元0/1互換
  - 也就是$2^m - 1 - x$
- 所以只要在一群數字中搜尋另外一個數字就可以了，用二分搜或是字典都可以
- 時間複雜度：字串處理O(nm), m<=26，搜尋與排序O(nlogn)
- 實作細節：需要位元運算

轉數字 + set (也可以用sort)

```
 5  int main() {
 6      int m=0, n=0;
 7      int teams[50000]={0};
 8      char s[110];
 9      set<int> D;
10      scanf("%d%d", &m, &n);
11      int ff = (1<<m)-1;
12      for (int i=0; i<n; i++) {
13          scanf("%s",s);
14          int len=strlen(s);
15          for (int j=0;j<len;j++) // 1 for existing
16              teams[i] |= 1<<(s[j] - 'A');
17          D.insert(ff-teams[i]); // insert the complement into set
18      }
19      int ans=0;
20      for (int i=0; i<n; i++) { // O(nlogn) check if complement is in teams
21          if (D.find(teams[i])!=D.end())
22              ans++;
23      }
24      printf("%d\n",ans/2);
```

m個1

- 可以直接用字串做，但順序與出現次數不計一個集合可能有多個不同的字串可以表示，所以我們要把字串標準化
  - 最簡單的方法是字元排序好且不重複。
  - 將這些字串排序
  - 對每一個字串，算出互補字串，然後進行二分搜(或字典搜)
- 如何將一個字串中的字元改成不重複且依照順序排列？
  - 例如 SISCAA => ACIS
  - 有多種方法。
  - 因為字元範圍有限，先抄到表示是否存在的表格中，再依序抄下來

字串+binary_search

```
 8  int main() {
 9      int m=0, n=0;
10      cin >> m >> n;
11      for (int i=0; i<n; i+=1) {
12          string s;
13          int alpha[26]={0};
14          cin >> s;
15          for (int i=0; i<s.size(); i+=1) // 1 for existing
16              alpha[s[i]-'A'] = 1;
17          // build set-string and complement-string
18          for (int k=0; k<m; k+=1)
19              if (alpha[k]) teams[i] += ('A'+k); // append a char
20              else comp[i] += ('A'+k);
21      }
22      // sort for binary search
23      sort(teams, teams+n);
24      int ans=0;
25      for (int i=0; i<n; i+=1) { // O(nlogn) check if complement is in teams
26          if (binary_search(teams,teams+n,comp[i]))
27              ans += 1;
28      }
29      cout <<ans/2 << endl;
```

alpha[0]表示A在不在

每一對會算到兩次

P-2-15. 圓環出口 (APCS202007)

- 有n 個房間排列成一個圓環，以順時針方向由0到n − 1編號。玩家只能順時針方向依序通過這些房間。每當離開第i號房間進入下一個房間時，即可獲得p(i)點。玩家必須依序取得m把鑰匙，鑰匙編號由0至m-1，兌換編號i的鑰匙所需的點數為Q(i)。一旦玩家手中的點數達到Q(i)就會自動獲得編號i的鑰匙，而且手中所有的點數就會被「全數收回」，接著要再從當下所在的房間出發，重新收集點數兌換下一把鑰匙。遊戲開始時，玩家位於0號房。請計算玩家拿到最後一把鑰匙時所在的房間編號。
- n不超過2e5，m不超過2e4，p(i)總和不超過1e9，Q(i)不超過所有p(i)總和。

圓環出口

- 在一個陣列當成圓環繞，陣列每一個有正整數，每次要往前走一個最短的連續區間，蒐集數字總和大於給定的限制。問最後停在哪裡
- 所以是相同的事要做m次，每次在目前的位置找最短區間，其區間和>=Q(i)
- 時間複雜度的要求：O(m*n)太大，所以不能一步一步走。可否很快地找到每一次任務的停留點？
- 要找區間和，可用前綴和
- 把原序列看成差分，把它做前綴和轉換，就會變成一個遞增序列
- 遞增序列就可以二分搜：下一個停留點就是要找(目前所在的值+任務需求的值)的位置

202109第3題：幸運號碼

- 有n個人排成一列，從左而右由1到n編號，每個人有一個幸運號碼，編號i的幸運號碼為k(i)，每個人的幸運號碼都是正整數且互不相同。對於兩個正整數L與R，我們以[L,R]來表示編號從L到R的區間，也就是L到R的所有整數(包含L與R)，若L ＞ R則此區間為空集合。我們要依照以下程序在 [1,n]區間中找出一個幸運號碼。
  在[L,R]中找出一個幸運號碼的程序：
  - 如果L ＝ R，也就是此區間中只有一個人，則幸運號碼就是k(L)；
  - 否則，先找出[L,R]區間中最小的幸運號碼，假設為k(m)，m將此區間分割成左邊的[L,m-1]與右邊的[m+1,R] 兩個區間，幸運號碼總和比較大的區間的幸運號碼即為所求。如果兩邊的總和相等，則所求為右邊區間的幸運號碼。空區間的幸運號碼總和為零。
- n不超過3×10^5，所有幸運號碼皆為不超過10^7的正整數且互不相同。

舉例來說

- (9, 3, 5, 4, 1, 6, 2, 8)。
- min=1，左邊(9,3,5,4)，右邊 (6,2,8)；
  - 左邊總和為21而右邊的總和為16
  - 剩下(9,3,5,4)
  - min = 3; left=(9), right=(5,4)
  - (5,4)
  - min=4, left=(5), right=()
  - (5) => found

解題思路

- 這是一個遞迴過程，每次在序列中找最小值，切成兩段，依照兩段的總和，丟棄其中一段，在另一段中遞迴此程序，直到剩下一個元素。
- 很像二分搜的過程，只是切點與決策方式不同。我們可以寫下流程
- 輸入序列S；

```
left = 1, right = n;
while (left < right) { // 當區間[left, right]超過一個元素
    找出[left, right]區間中最小值的位置mid;
    sum1 = [left, mid-1]區間的和;
    sum2 = [mid+1, right]區間的和;
    if (sum1 > sum2)
        right = mid -1; // 剩下左區間
    else
        left = mid +1; // 剩下右區間
    // end if
// end while
輸出S[left];
```

Key points

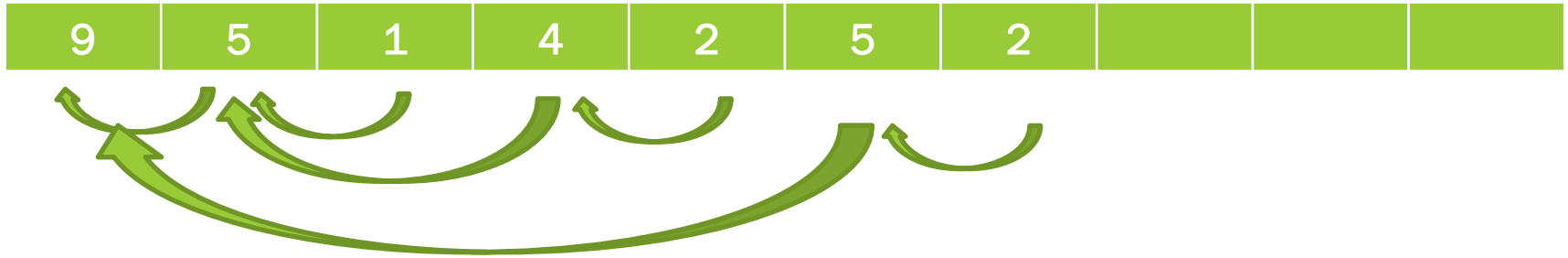
- 有兩個問題需要克服
  - 找區間最小
  - 找區間和
- 複雜度需求如何？n=300000，大概必須是O(n)或O(nlogn)
- 如果用直接的方法求最小值，可以嗎?
- 這問題涉及這個while會執行多少次，每次長度可能只減少1(最小值在最左或最右)，所以可能執行O(n)次，這樣會導致O(n^2)，不行
- 區間和很容易用前綴和來做到每次O(1)
- 但區間最小值怎麼辦?
- 查詢區間的最小值稱為RMQ(range minimum/maximum query)。有一些複雜的資料結構可以做到online query。但這題有無簡單的方法?

- 我們每次要查詢的區間是任意的嗎? 如果是，那就沒有辦法了。如果不是，在此查詢區間的規律中找方法
- 我們需要的查詢並不是任意區間都會發生，而是每次的區間都會是前一次區間的一部分
- 所以可以這樣做
  - 將所有元素先由小排到大，紀錄每一個數原來的位置
  - 如果當前的最小值在目前的區間，它就是我們要找的。
  - 否則，當前的最小值不在目前的區間時，它也不會在未來的區間中，因此我們可以將它直接丟棄。

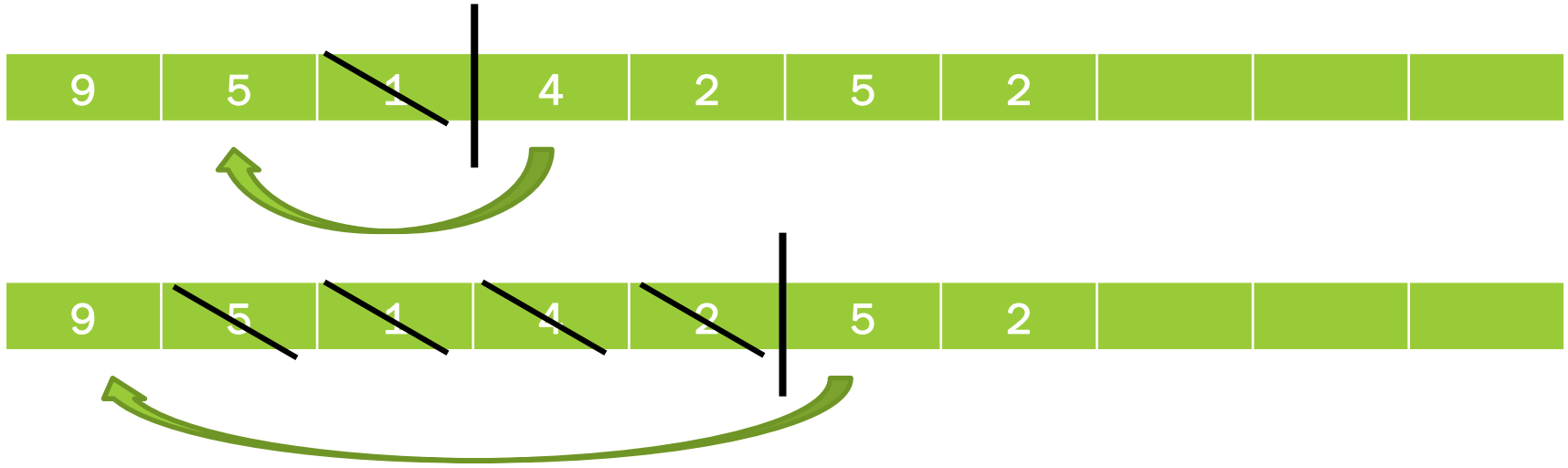
```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int i,n;
    scanf("%d", &n);
    vector<int> lucky(n+1);
    vector<long long> prefix(n+1);
    vector<pair<int,int>> min_q;
    prefix[0]=0;
    for (i=1; i<=n; i++) {
        scanf("%d", &lucky[i]);
        prefix[i] = prefix[i-1]+lucky[i];
        min_q.push_back({lucky[i],i});
    }
    sort(min_q.begin(), min_q.end());

    int left=1, right=n, qi=0;
    // range [left, right]
    while (left < right) {
        // find the smallest in [left, right]
        while (min_q[qi].second<left || min_q[qi].second
>right)
            qi++; // out of range, discard
        int m = min_q[qi].second; // min in [left, right]
        // sum of the two range
        long long sum1 = prefix[m-1] - prefix[left-1];
        long long sum2 = prefix[right] - prefix[m];
        if (sum1 > sum2) right=m-1;
        else left=m+1;
    }
    printf("%d\n",lucky[left]);
    return 0;
}
```

P-3-4 前方的高人(APCS201902)

- 給一個整數序列a，對每一個a[i]，找到他前方離他最近比他大的(高人)



- 最天真又直接的方法就是：對每一個i，從i-1開始往前一一尋找
- 改善：假設由前往後掃過去的時候，哪些沒有用？
  - a[i-1] ≤ a[i]，那麼a[i-1]不可能是i之後的人的高人，因為由後往前找的時候會先碰到a[i]，如果a[i]不夠高，a[i-1]也一定不夠高。
  - 如果我們丟掉那些沒有用的，會剩下甚麼呢？
  - 一個遞減序列。維護好這個遞減序列，就可以有效率的解此題



- 以一個stack存此遞減序列，在a[i]時，從後往前看 <=a[i]的都沒有用了；碰到第一個 >a[i]的就是他的高人，並且把a[i]放入stack。
- 延伸：Q-3-5. 帶著板凳排雞排的高人 (APCS201902)
  - 在此遞減序列上做二分搜(因此不要用stack)
- 這個題型稱為monotonic stack/queue 或單調隊列，基本形是對於數列的每一個點，找出大於它且距離它最近的點(nearest greater)

- for 迴圈內有個while，複雜度會不會壞掉？
- 事實上不會，整個程式的複雜度是O(n)
- 雖然while可能某次做很多次pop()，但總共只會做O(n)次
  - amortized analysis
  - 均攤分析

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

單調隊列 Deque版本

- 與monotonic stack不同之處在於，在deque的題型中，前方的元素可能因為過期而失效，所以需要把它移除。這個狀況是不出現在前面stack題型中的。
- 例題：LeetCode 239. Sliding Window Maximum (hard)
- 題目說有一個長度為k的sliding window在一序列中一步一步移動，請計算出在每一個位置時的區間最大值。題目名稱有sliding window，但並非用該技巧。
- 其實要算的是$[max(nums[0:k]), max(nums[1:k+1]), \cdots]$，也就是所有$n-k+1$個長度為$k$的區間的每一個區間最大值。當然，直接這麼寫的時間複雜度是$O(kn)$，太慢了。

單調隊列的小故事

- 「學生一屆一屆的進入學校，小明很喜歡程式比賽，當他進入學校時，如果有學長姐厲害到小明難以超越，那沒關係，小明在學校還是有出頭之日，因為學長姐會比他先畢業。但如果在小明之後新進來一位他無法超越的學弟妹，那完蛋了，小明在該校沒有什麼機會獨領風騷了，因為他會先畢業。」
- 我們的window在序列一步一步右移，可以看成一屆有一位學生進入，視窗長度固定為k，也就是學生進入後k屆就會畢業，同時在校的也恰有k位，我們要找出每一年最厲害的校內冠軍。

- 上面的例子告訴我們，對於某個i(小明)，如果同時在視窗內有個j，$nums[j]>nums[i]$，如果$j<i$，則 i 尚未絕望，將來還有可能有機會當冠軍；但如果$j>i$，則 i 從此刻起永遠不會是校內冠軍了。
- 如果我們把視窗內這些不可能成為冠軍的去除不看，那麼視窗內的將是個單調下降的子序列(monotonically decreasing subsequene)。這就是單調隊列。
- 我們用一個容器存放目前的隊列，維護的迴圈不變性是：
  - 隊列是遞減的
  - 隊列中均為尚在範圍內的元素(還沒畢業)
- 請注意，根據迴圈不變性，隊列最前方的就是區間最大值。

- 每次新元素進入時，我們做兩件事：
  - 新生刪除所有不比自己強的學長姐。根據迴圈不變性，隊列為遞減的，我們只要從後往前刪除比新元素小的成員，當碰到第一個大於他的元素，刪除動作即已完成，因為前面的更大。刪除完成後再將自己加入隊列尾端(新生永遠有希望)。
  - 檢查最前方，也就是最大的元素是否已經出界，若是，將其移除。
- 要使用什麼容器來存放隊列呢？因為要支援前方與後方的移除，典型的做法是用deque（double ended queue雙向佇列）。
  - C++與Python都有提供deque，我個人比較喜歡用list加上一個head變數記錄頭端的位置即可。因為list尾端的刪除與新增都是$O(1)$，所以很方便。

P-3-6. 砍樹 (APCS202001)

- N棵樹種在一排，每棵樹有它的位置與高度，現階段砍樹必須符合以下的條件：「讓它向左或向右倒下，倒下時不會超過林場的左右範圍之外，也不會壓到其它尚未砍除的樹木。」。計算所有能砍除的樹木。
- 我們可以不斷找到滿足砍除條件的樹木，將它砍倒後移除，然後再去找下一棵可以砍除的樹木，直到沒有樹木可以砍為止。無論砍樹的順序為何，最後能砍除的樹木是相同的。

解題思路

- 最直接的想法：每次檢查所有剩下的樹，砍掉那寫可以砍的。重複這個步驟直到無樹可砍。
- 時間複雜度如何？每次可能只發現一棵可以砍，這個方法可能要O(n)個回合，總時間也就是平方了。
- 是不是有不必要算的？
- 假設由前往後一一檢視，能砍就砍了，不能砍的暫時放著。想一想，暫時不能砍的樹何時會變成可以砍？
  - 除非他後面那棵樹被砍，否則它的狀態沒有改變，所以不可能可以砍。
- 用一個stack把暫時不能砍的樹放起來，每次下一棵樹被砍時，往前檢查堆疊內的就可以了。
- 時間？O(n)

Q-3-12. 完美彩帶 (APCS201906)

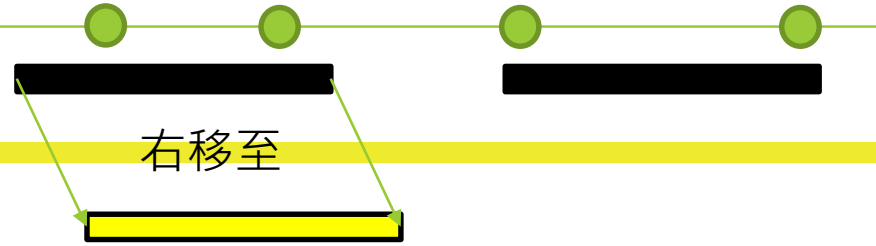
- 有一條細長的彩帶，總共有m種不同的顏色，彩帶區分成n格，每一格的長度都是1，每一格都有一個顏色，相鄰可能同色。長度為m的連續區段且各種顏色都各出現一次，則稱為「完美彩帶」。請找出總共有多少段可能的完美彩帶。請注意，兩段完美彩帶之間可能重疊。
- 在一個數列中找有多少長度m的區段，它裡面的數字都不重複。
- 數列找區段，滑動視窗
- 維護一個長度m的區間，每次往右端一格，維護好區間內顏色的數量。
- 如何維護：紀錄每個顏色出現的次數。
  - 右移時，新加入的+1，左端移除的-1。
  - 顏色數量怎麼算?
    - 不能每次都算算有幾個顏色
    - 加入變成1時，數量加一；減少變成0時，數量減1。
- 顏色的數字很大時怎麼做計數器？ 離散化或者用字典

P-4-9. 基地台 (APCS201703)

- 直線上有N個要服務的點，每架設一座基地台可以涵蓋直徑R範圍以內的服務點。輸入服務點的座標位置以及一個正整數K，請問：在架設K座基地台以及每個基地台的直徑皆相同的條件下，基地台最小直徑R為多少？座標範圍不超過1e9，1≤ K < N ≤ 5e4
- 服務點可以看成數線上的點，基地台可以看成數線上的線段
  - 以K根長度相同的線段蓋住所有的點，最小的線段長度。
- K與N都不小，嘗試找DP關係式成功機會不大
  - 例如以f(k,n)表示前n點用k根的最小直徑，最後一根蓋[p[i],p[n]]
    f(k,n) = min(max(f(k-1,i-1),p[n]-p[i]) for i<n)
  - 除非找到優化的方法，此法不少於O(KN)

解題思路

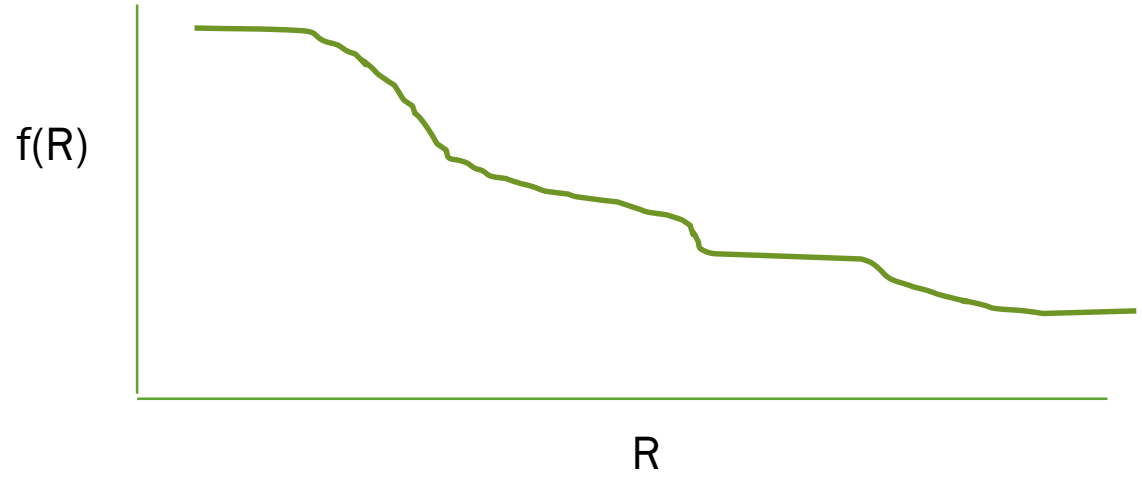
- 看不出有什麼可以貪心的?
- 我們看看解的樣子，有沒有什麼可以把解簡化(標準化)的方式
  - 問題可能存在多個解，我們在其中找一個滿足某些條件的解或許比較容易
- 第一根的左端一定可以對齊最左邊的點，最後一根的右端一定對齊最右的點，一定至少有一根左右端都對著某個點(否則答案可以減少)
- 找到一點性質但不知道有用否? 也就是答案必然是某兩點的距離
  - ans = p[j]-p[i] for some i<j
  - 枚舉任兩點的距離需要O(N^2*驗證時間)
  - 枚舉時顯然有很多浪費白做的事，當找到R=p[j]-p[i]是可以蓋得住時，不再需要找p[j+1]-p[i]，因為>R的必然也可以
- 是了！這裡有單調性，若某個長度是可以的，更長的必然可以。只要能夠快速的驗證某個長度是否可以，就可以二分搜來找答案



- 驗證的方法：
  - 對於給定的N個點與某個長度R，請問是否可以用K根長度R的線段蓋住所有的點
  - 「一定有一個最佳解是將第一根線段的左端放在最小座標點上。」
  - 所以座標排序後，可以用貪心的原則掃過去就知道至少要幾根
  - 每次新進來一點時，
    - 蓋得住就不要理它
    - 蓋不住時，拿一根新的，左端對齊它
  - O(N)可以完成驗證
- 在那些範圍搜? 不能把N^2個可能的長度找出再去搜
  - 我們可以不要管那N^2個可能，放到大所有可能的整數，因為即使是座標範圍是2^64，取log也只有64
- 我們可以得到一個O(Nlog(P))的解，P是座標範圍

對答案二分搜(外掛二分搜)

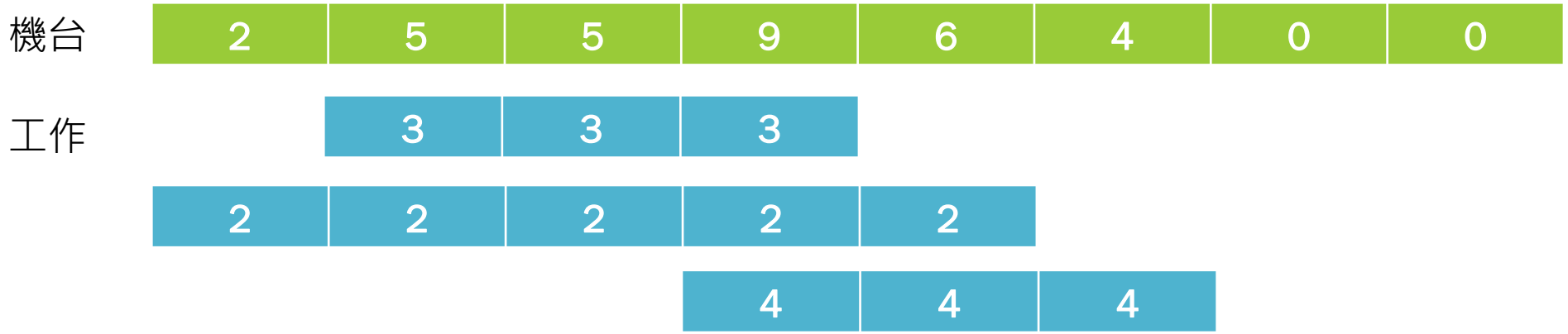
- 用長度R的K根線段蓋住所有點
- 假設f(R)是給定長度R的最少線段數，那麼上述驗證的方法可以求得f(R)。
- 當R增加時，f(R)必然只會相同或減少
  - 因為用更長的線段去蓋相同的點，不會需要更多線段。
- 所以我們可以二分搜來找出最小的R，滿足f(R)≤ K。



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

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

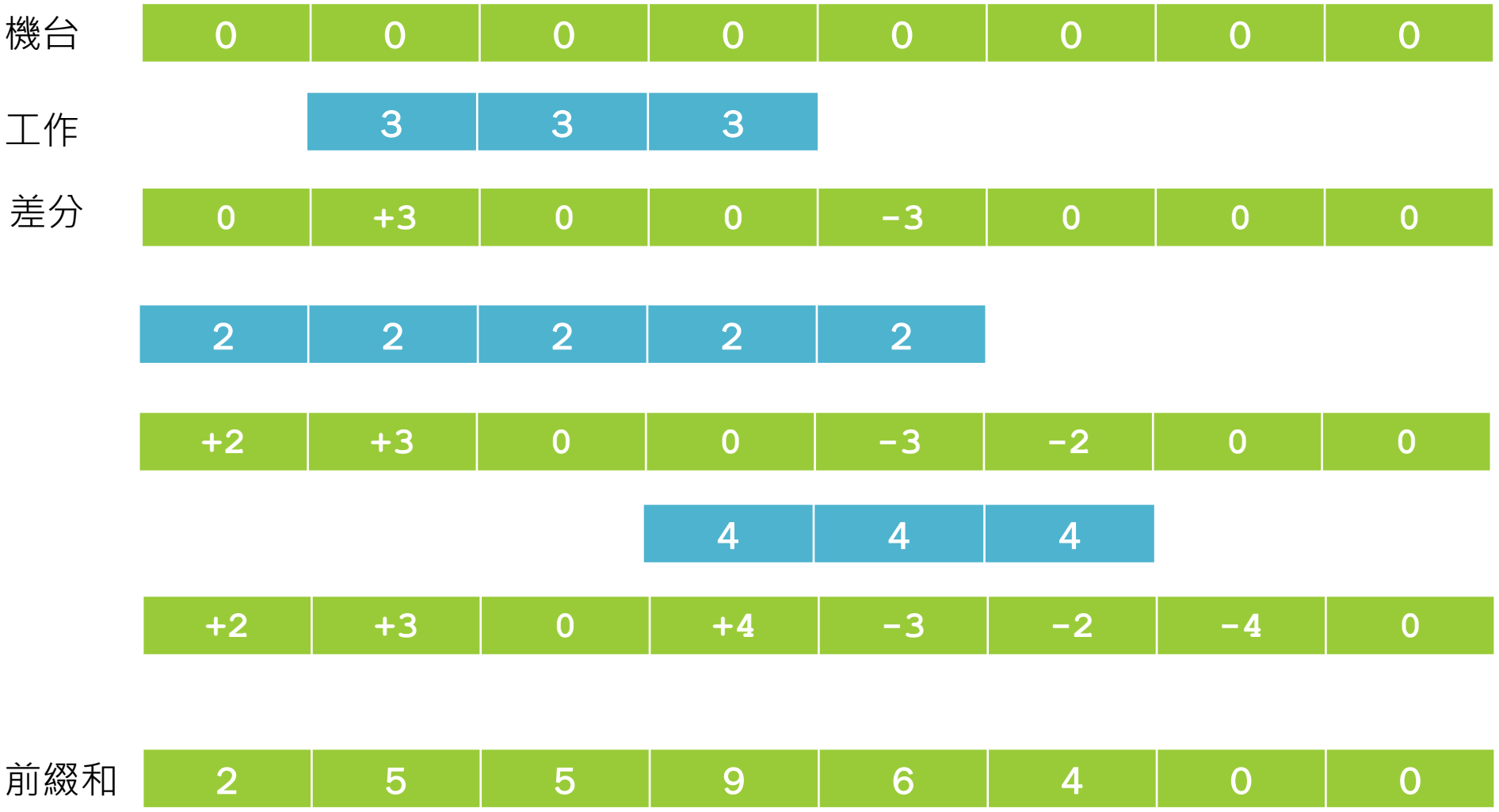
202111第三題：生產線

- 有n個機台由左而右排成一列，編號為1至n。有m個工作，編號為1至m，每個工作會使用到某些連續編號的機台，其中第i個工作使用到的機台編號[s(i),f(i)]。處理工作i的每個機台都需要透過傳輸器傳輸w(i)的資料到總機，第j個機台預設的傳輸器每傳輸1單位的資料所需要的時間是t(j)。傳輸器與機台是可以任意搭配的，現在希望調整傳輸器的位置，以便資料傳輸的總時間能夠越短越好。
  m,n <= 2e5
- 算出每個機台的資料量data(j)，再分配傳輸器給機台，最小化總傳輸時間sum(data(j)*t(j))



解題思路

- 兩個向量(數值皆非負)的內積要最小，大的乘(配)小的
  - 證明？否則交換總和更小，a<b, x<y, ay+bx<ax+by
- 兩段式的題目
  - 算出每個機台的傳輸總量
  - 將最大的傳輸量配最快的傳輸器 --- greedy
- 如何計算傳輸總量（線段厚度）
  - 不只一種方法
  - 可以看成每一個工作做一個區間修改
  - 用差分轉換最簡單


機台
0 0 0 0 0 0 0 0
工作
3 3 3
差分
0 +3 0 0 −3 0 0 0
2 2 2 2 2
+2 +3 0 0 −3 −2 0 0
4 4 4
+2 +3 0 +4 −3 −2 −4 0
前綴和
2 5 5 9 6 4 0 0

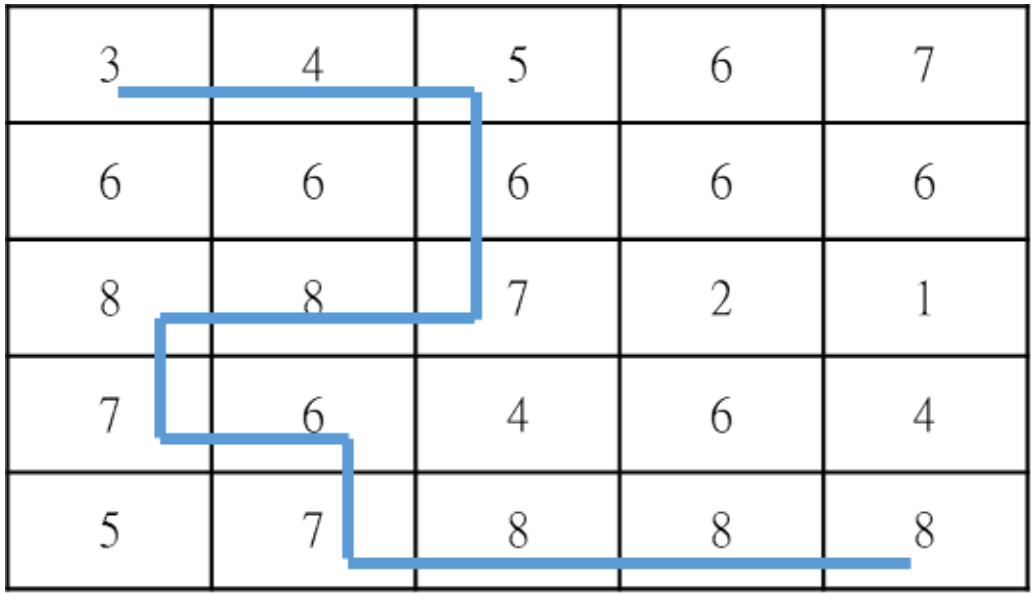
線段厚度(座標為1~n)

- 退化到不需要排序
- 也不須離散化

```
int main() {
    int i, n, m, w;
    int le, ri, w;
    scanf("%d%d", &n, &m);
    vector<int> t(n), load(n+1,0);
    for (i=0;i<m;i++) {
        scanf("%d%d%d", &le, &ri, &w);
        load[le-1] += w; // 0-index
        load[ri] -= w; // delete at right+1
    }
    // prefix sum of difference sequence
    for (i=1; i<=n; i++) load[i] += load[i-1];
    load.pop_back();
    sort(load.begin(),load.end());
    for (i=0; i<n; i++) scanf("%d", &t[i]);
    sort(t.begin(), t.end(), greater<int>());
    long long total=0;
    for (i=0; i<n; i++) total += (long long)t[i]*load[i];
    printf("%lld\n",total);
    return 0;
}
```

APCS2022.10 平緩步道

- 一個n*n的方格區域，每個小方格可以與它上下左右四個相鄰的方格之間架設連通步道，相鄰兩格高度差定義為這段步道的坡度。
- 找出一條由左上角到右下角的最平緩的步道，也就是這條步道所經過相鄰兩格的最大坡度要越小越好。滿足最小坡度的步道找出最短的步道。
- 在此例中可以找到一條坡度不超過1的步道。此外，坡度為1的步道不只有一條，但圖示中的步道是最短的，其長度為12，也就是包含入口與出口一共經過13個方格。
- n<=300，高度<=1e6



解題思路

- 方格圖(grid)其實是graph的一種，高中特別喜歡出，因為可以考圖的很多問題而輸入與描述很簡單。
- graph有很多演算法，在APCS階段不外乎BFS/DFS走訪與topological sort。計算(無權重)最短距離可以用BFS，但本題
  最小高度差要怎麼求呢？
- 方格圖的鄰居往往定義為上下左右，本題也是。BFS走訪時，我們是可以根據需要定義「鄰居」，也就是滿足某個條件的才能走過去。在本題中，我們可以給定高度差的條件來定義鄰居，也就是，我們可以對一個給定的h，定義一個點相鄰且滿足高度差h的才能走。
- 如此用BFS就可以計算在高度差h的限制下的最短路徑。
- 如何求最小高度差呢？如果高度差限制太小就會走不到，所以可以到的最小高度差限制就是要求的最小高度差。
- 由小到大逐一嘗試顯然複雜度不夠好，n*n*H = 9e10。
- 有單調性嗎？顯然有，高度限制放更寬可走的路更多。
  二分搜，O(n^2 log(H))

方格圖的走訪小技巧

- 用dr, dc與迴圈取代4個if
- 距離以不可能的值，減少使用另外一個visit陣列
- 邊界外圍一圈不可能的值減少檢查出界

```
06 struct POS {
07     int r,c;
08 };
09 int grid[N][N], dr[4]={0,1,0,-1}, dc[4]={1,0,-1,0};
10 // BFS, source=(1,1), dest=(n,n), return distance or -1 if not reachable
11 int bfs(int n, int slope) {
12     int i,j, d[N][N];
13     for (i=1;i<=n;i++) for (j=1;j<=n;j++)
14         d[i][j] = oo;
15     d[1][1] = 0;
16     queue<POS> Q;
17     Q.push({1,1});
18     while (!Q.empty() && d[n][n]==oo) {
19         auto e=Q.front();
20         Q.pop();
21         for (int i=0; i<4; i++) {
22             int r=e.r+dr[i], c=e.c+dc[i];
23             if (d[r][c]==oo && abs(grid[r][c]-grid[e.r][e.c])<=slope) {
24                 Q.push({r,c});
25                 d[r][c] = d[e.r][e.c]+1;
26             }
27         }
28     }
29     return d[n][n];
30 }
```

方格圖的走訪有些小技巧，多熟悉可以寫出比較簡潔清爽的程式，否則可能寫的又臭又長，這類的題目很多，平時可多練習。

選手初階 --- 競技程式

要學的很多，但其實到這階段應該已經具備自學能力了
可以不需要老師，因為網路上都是老師

IOI and ICPC

- IOI 有範圍限制(IOI syllabus)而ICPC沒有
  - 簡單來說，競程常見的範圍不在IOI的範圍內的包括
    - max flow (linear programming)
    - 進階數論
    - 複數、高維幾何與三角函數
    - 解遞迴
  - 但有些高中競賽也沒規定必須在IOI範圍

起碼的配備

- 經驗與程度已經提升後，學習這些東西並不會太困難
  - 理解一下背後的原理，看看使用的方法，做些練習就可以了
  - 大部分的風格/使用習慣/命名都很類似，少部分比較複雜
  - 但是裝備越來越多的時候，也有混淆的可能。(只能靠多使用來增強)
- STL中的裝備
  - 基本的 vector, stack, queue, deque，也有array與list
  - priority queue, (multi)set/map, unordered_(multi)set/map
  - lower_bound之類的binary search
  - bitset
- 其他重要資料結構
  - disjoint set (union and find)
  - binary index tree (BIT, Fenwick tree)
  - 線段樹
  - Range Minimum/Maximum Query (RMQ)
  - Lowest Common Ancestor (LCA)

- 進階DP與常用優化
  - 斜率優化
  - totally monotonic (Monge)(四角不等式)
- Graph algorithm
  - 算距離的：Dijkstra, Bellman-Ford, Floyd-Warshall
  - minimum spanning tree: Kruskal, Prim
  - DFS for finding bridge and cut vertex, DFS for offline LCA
  - Bipartite matching
- String algorithm
  - KMP

進階資料結構簡介

- priority queue
  - 以陣列實作完整二元樹
  - 任意加入，查詢或刪除最小值(或最大)
  - logn
  - 可以用set代替，但pq速度較快
- (multi)set/map
  - 平衡的binary search tree
  - 有序資料，單一鍵值，multiset可以相同key
  - 任意加入與刪除，二分搜尋lower_bound/upper_bound
- unordered_(multi)set/map
  - hash table，也稱之為字典。以雜湊函數來很快的(存與取)資料
  - 鍵值必須可以做hash，常用在整數，字串，數對
  - 資料量不大時，平均存取時間可以視作O(1)，資料量太大時難以估算。

- bitset
  - 用第i個bit 1 or 0表示第i個元素在或不在的集合表示法，元素不多時<=64，常常自己用unsigned long long(或 int)來做，但集合很大時，需要多個long long來做，比較麻煩。可以用bitset
  - 有bit parallel的效果，一次操作32(64)個元素，某些題目可以得到32(64)倍的加速
- lower_bound之類的binary search
  - 幫妳寫好的二分搜，減少自己寫的精力與犯錯可能
  - C++有binary_search, lower_bound, upper_bound, equal_range

需要自製的

- disjoint set (union and find)
  - 用陣列完成樹狀結構的一種資料結構，處理不相交的子集合。
  - 程式碼簡潔，容易寫，速度快，用途多
  - 基本原理：
    - 每個集合以一個代表元素來表示，每個元素有一個parent，代表元素的parent為不可能的值(或自己)
    - find的時候沿著parent找到代表元素
    - union時將其中一個集合代表元素的parent設為另一個
  - 它內部有兩個技巧
    - 合併時小的併給大的
    - find的時候將找過的路徑上的點，全部帶給root，下次就很快了（路徑壓縮）
  - 多半的時候只使用路徑壓縮就夠好了

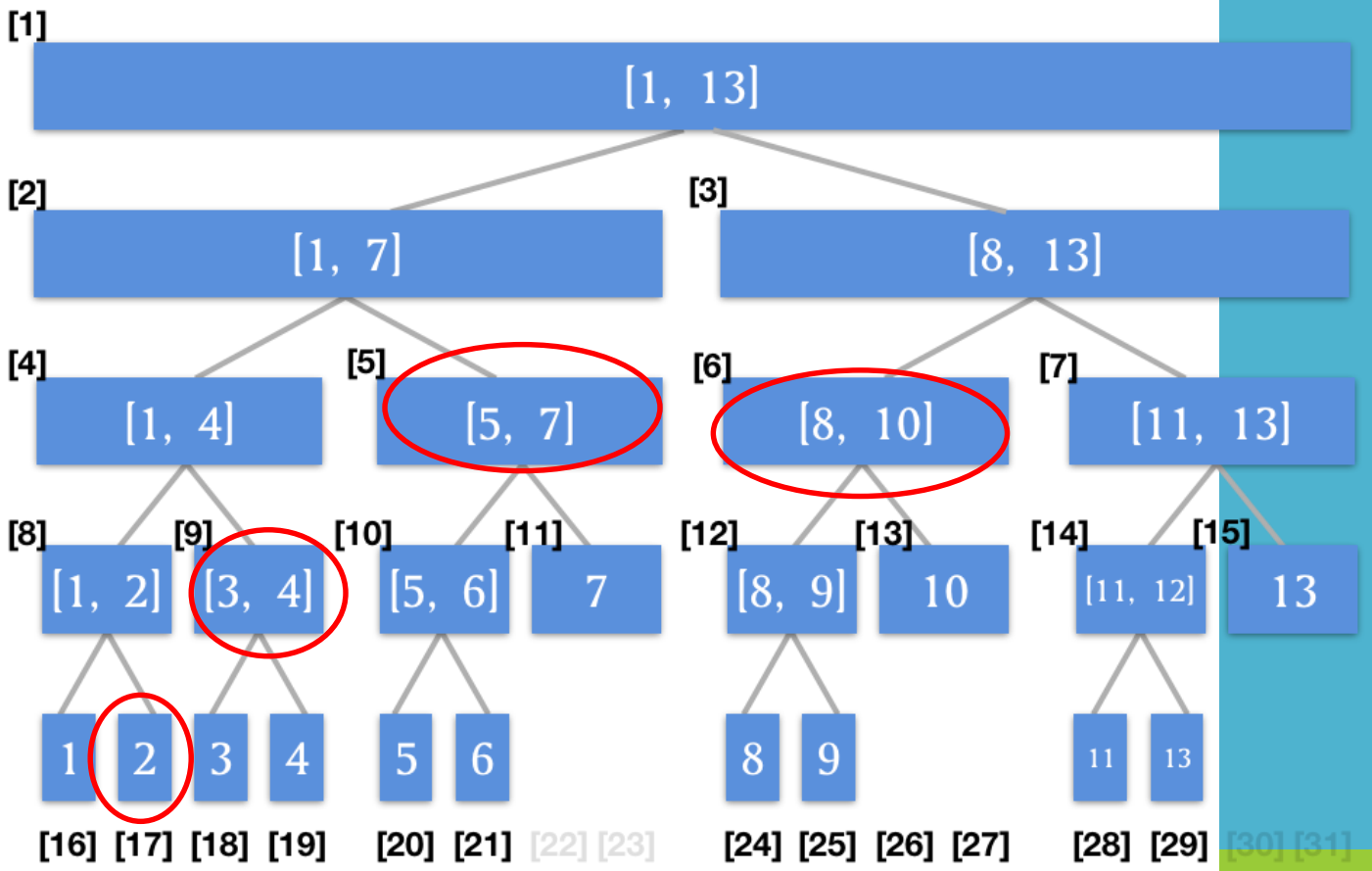
Binary index tree (BIT, Fenwick tree)

- 簡化版的線段樹，可以用線段樹取代
- 程式碼簡潔，容易寫，速度快
- 資料可修改，logn算前綴和與區間和，也可以做前綴和的二分搜尋
- 也可以做prefix max/min
- 範圍大時需要離散化
- 基本原理
  - 把每個點的資料放在若干個位置，計算前綴和時，取用某些位置可以取到所有且不重複的位置
  - 這件事其實沒那麼神奇，分組的概念而已。把所有學生分年級分班，各班班長紀錄該班總和，要找10年1班1號到10年5班27號的總和就不必每一個學生都拿來加。
  - 不過這個方法要更有效率，它巧妙了利用位元運算與位置安排。
  - 有些程式學過一次就不會忘也不容易寫錯，這是一個。

線段樹

- 用途非常強大的資料結構，可以做單點修改與區間修改，查詢區間最大、區間總和等等，也可以有很變化，基本上只要可以分治的大概都可以做。
  - 一個區間切成兩半後，你要的東西可以從兩邊各自的解中求得，例如 max(S) = max(max(S1),max(S2)); sum(S)=sum(S1)+sum(S2)
  - 線段樹有多個種寫法，有些時候只需要其中某些功能或是簡化版就足夠。
  - 線段樹的觀念並不難但程式碼比較複雜一些，有些細節也比較容易寫錯。
- 基本原理：
  - 將區間逐層二分
  - 每一個節點表示一個區間
  - 每一個操作或查詢的區間，可以分解為樹上一些不相交的區間，每一層最多兩個

- 例如：[2,10] =>
- 每個區間儲存該區間的值
- 修改與查詢時，針對分解出來的區間就可以了，所以大致是0(logn)
- 適合遞迴的方式來做，有時需要先離散化
- 還有很多細節以及延伸的技巧



Range Minimum Query (RMQ)

- Minimum/Maximum是類似做法
- 可以用線段樹做
- 靜態的(immutable，資料不變動)有更簡單的方法
- 基本原理：倍增法
  - 前處理：對於每個點，以其為左端區間長度為1,2,4,8,…的區間最小值都求出來，有O(nlogn)個，利用倍增法每個只要O(1)可算出
    min([0,3]) = min(min([0,1]),min([2,3])) 長度2^{i+1}的可以由兩個長度2^i的求出
  - 查詢：對於區間[L,R]，根據區間長度(R-L+1)，我們找到兩個長度2^i的區間使得這兩個區間可以恰好涵蓋[L,R]，那麼
    min([L,R]) = min(min([L,L+2^i-1], min([R-2^i+1,R]))
    其中 L+2^i-1 >= R-2^i+1-1

Lowest Common Ancestor (LCA)

- 在一個rooted tree上做一些前處理，以便可以很快的查詢任兩點的做低共同祖先
- 有多種方法，
  - 其中之一是利用RMQ，用DFS在樹上走一個Euler cycle，把拜訪順序記下，lca(u,v)就是u,v兩點在序列中第一次出現這個區間的RMQ
  - 另外一種是倍增法
    - 先跑一次DFS記錄每一個點的深度在第幾層
    - 倍增法紀錄每個點距離為1,2,4,8,…的祖先
    - 查詢時，先將較深的點往上跳到與另外一點相同深度的祖先，跳的時候用二進位表示法(類似快速冪)。然後兩點用二分搜往上跳，找最大不同祖先，往上一個就是LCS
- 常見題型：題目中要求多次樹上任兩點距離。

幾個例子

- Inversion number（反序數）
  - APCS考題
- Longest Increasing Subsequence (LIS)
  - APCS考題
- 直升機抓寶
  - 105高中全國賽（約5人答對）
- 山花聚頂
  - 113北市賽(5人答對)
- 樹上不回家的推銷員
  - 107高中全國賽（2人答對）

反序數(Inversion number)

- 給一個數列a[]，對每一個位置i，計算出有多少個j<i且a[j]>a[i]。
- 需求O(nlogn)的解，只少也要$O(n\log^2 n)$
- 直接的思考，對於i=0,1,2…，計算前方有幾個>a[i]的數。
  - 用什麼資料結構可以把前方的數字裝起來，很快的查詢a[i]在其中第幾大?
  - 很抱歉，沒有簡單的資料結構可以做。
    - multiset底層的資料結構BST可以做，但它被封裝起來沒有提供此功能。
    - 線段樹可以做，但有些不好寫
- 從算法上來看，不是greedy不是DP，分治是否可用?
  - 一切兩半，各自去解(遞迴不必計算時間)
  - 左邊算出來的就是答案，因為前方並無別人
  - 對於右邊的a[i]，還要加上左半部>a[i]的個數
    - 排序後二分搜就好
  - $T(n) = 2T(n/2)+O(n\log n) \Rightarrow T(n) = O(n\log^2 n)$ ←需要了解複雜度才會想到
  - 再做細緻一點，避免重複排序就可以做到O(nlogn)

反序數

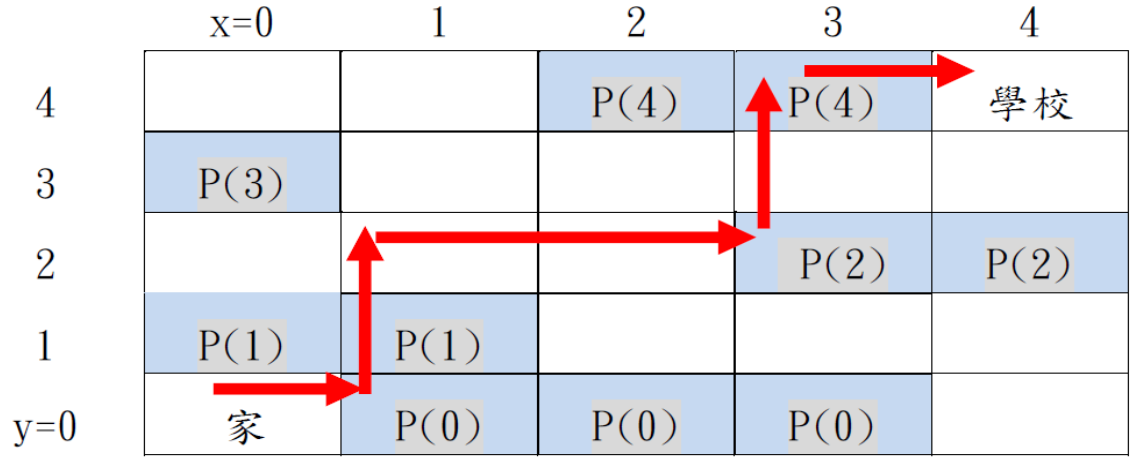
- 運用資料結構的直線解法
- 回到剛才直接的思考，對於i=0,1,2…，計算前方有幾個>a[i]的數。
  - 用什麼資料結構可以把前方的數字裝起來，很快的查詢a[i]在其中第幾大?
- 如果這些數字是1,2,3,…,n，有什麼辦法
  - 假設有個陣列b[2]=3表示2有3個，b[5]=0表示5有0個，那麼sum(b[0,a[i]])就是<=a[i]的有幾個，i-sum(b[0,a[i]])就是我們要的>a[i]的個數。
  - 只要能快速對b做單點修改(加入)以及查詢前綴和就可以辦到
  - Binary Indexed Tree是個合適的好東西
- 那數字範圍很大怎麼辦? 先做離散化就行了
- Total time complexity O(nlogn)

LIS (Longest Increasing Subsequence)

- 給一個數列，計算最長的遞增子序列
  - 無加權版(求長度)與加權版(每個點有權重，求最大權總和)
- 這是一個經典一維DP題，無加權的版本請參考AP325，有些推導過程，但結論是個簡單的方法，用到二分搜。
- 加權的版本可以用相似的推導過程，但需要用到map。
- 有沒有更直接的方法？假設數值是[1,K]的範圍(離散化後就做得到)
- a[i]一個一個進來，我們要算以a[i]結尾的最佳解，就是要知道在目前結尾介於[1,a[i]-1]的最佳解，因為a[i]可以接在它們的後面
- 這不就是prefix max嗎？所以用BIT就可以做到
- 每回合計算prefix max，加上1(加權版加上w(i))，做單點修改

直升機抓寶 (105高中全國賽)

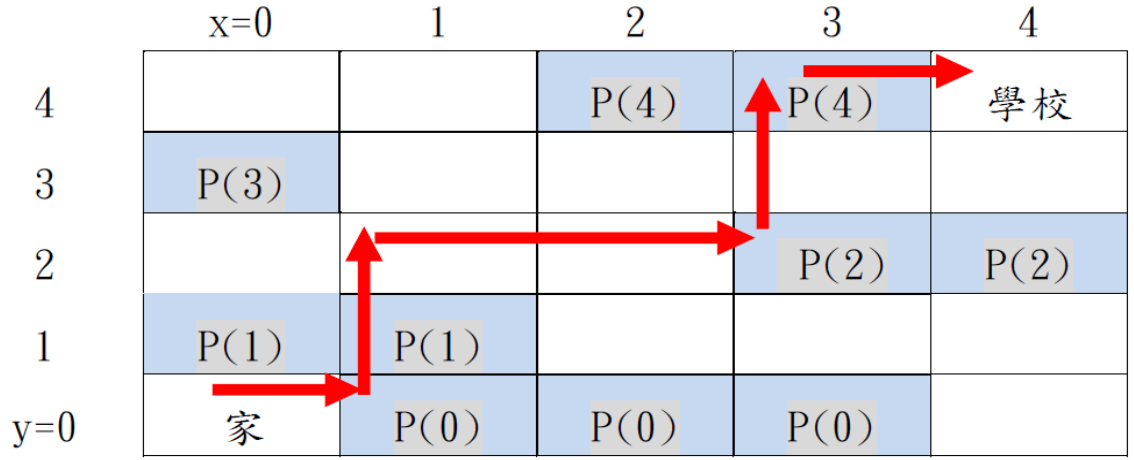
- N*N的方格，每一列有一個區間可以+1分
- 由左下走到右上每次只能向右或向上，最多可以得幾分
- N<=2.5e5，也就是要O(nlogn)，不能把格子建出來
- 有一個很簡單DP的例子跟此題很像
  - 每個格子有一個數字
- 解法是每一格記錄到達它時可以得到的最大分數
- 由下往上，由左而右，每次取下面與左邊較大值



1	3	3	2	0
0	3	0	5	1
4	1	2	0	7
0	2	1	1	0

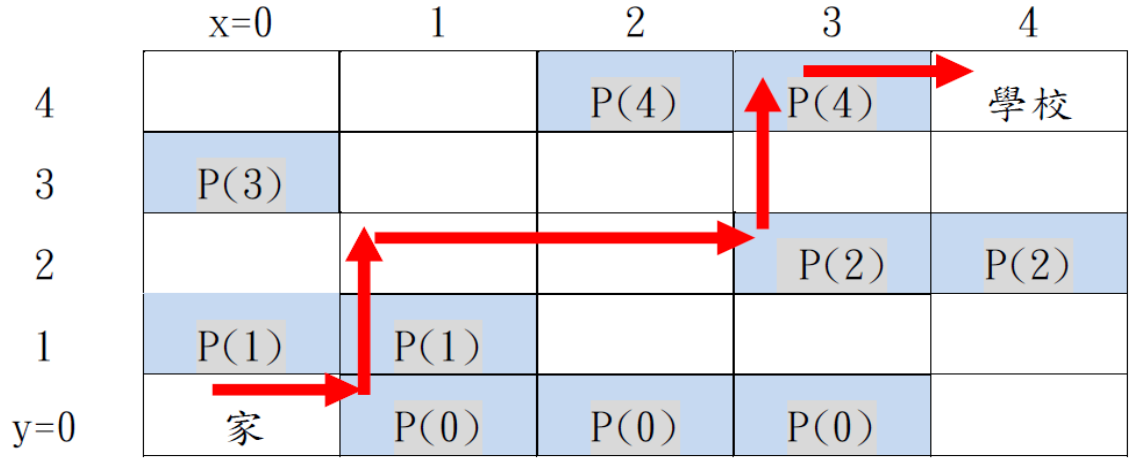
		max(8,7)+5 = 13		
4	8	8	13	15
4	5	7	7	14
0	2	3	4	4

max(4,2)+1 = 5



- 使用相同的定義，可以略做修改計算式
  - 一樣由下而上，由左而右，每一列計算時
  - 每個點的得分 = max(左方+(如果是區間頭),下方+(如果在區間內))
- 但O(N*N)太大
- 維護好dp[i]=到達第i直欄可以得到的最大分數，一列一列往上更新
- dp[]必然是非遞減序列。
- 到達每一列，針對此區間範圍做+1
- 區間以後的，維持非遞減就對了

2	2	3	4	4
2	2	2	3	3
1	2	2	3	3
1	2	2	2	2
0	1	1	1	1



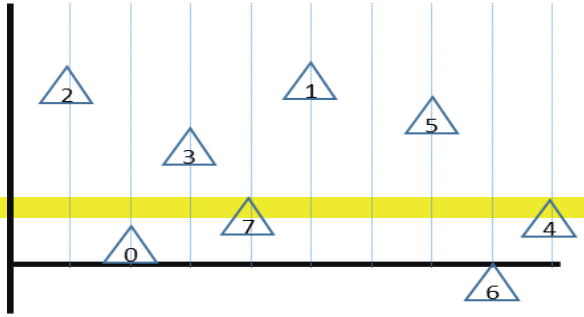
- 因為每次值只在該區間上升1，所
- 所以可以用線段樹來做區間+1
- 有沒有更簡單的方法？
- 如果做差分轉換，區間+1可以改成在區間頭+1，在區間尾-1，而要維持非遞減，只要把-1改成將其後的+1處(如果存在)抵銷即可
- 最後做prefix sum還原
- 為了要找後面的+1，我們用multiset維護

		+1		(無-1)
+1	-1			
			+1	
+1		(無-1)		
	+1			(無-1)

- 難嗎？當年大概是全國賽二等獎的程度(大約五人答對)
- 現在大概率變成水題，因為選手大部分都看過了

```cpp
int main() {
    int n;
    scanf("%d",&n);
    multiset<int> S;
    int low, up;
    for (int i=0;i<n;i++) {
        scanf("%d%d",&low,&up);
        S.insert(low);
        auto it=S.upper_bound(up); // next to up
        if (it!=S.end()) S.erase(it);
    }
    printf("%d\n",(int)S.size());
    return 0;
}
```

山花聚頂(113北市賽)



- 20萬座山峰排一列，每個山峰有水平位置與高度，有5萬次查詢，每次給k_i山峰(可重複)，要在m座山峰中找出一座高度不小於本次查詢的所有山峰，最小化到這些點的水平距離總和。sum(k_i) <= 200萬。
- 數線上與一群點距離總和最小的是median。
- 對每次查詢，若他們的高度最高的是max_h，找出median左右個找出一點最接近median且高度不小於max_h的點，答案是兩者之一。
- 有兩種解法
  - online query：線段樹(本題有簡易形式)
  - 這題是假在線，可以更改順序以offline的方式計算
    - 將查詢依照它們的max_h由大排到小，用一個set維護高度>=max_h的所有山峰位置。
    - 每次查詢前將高度>=max_h山峰的水平位置放入set中，以lower_bound查詢左右即可。

樹上不回家的推銷員

- 給一棵樹上n個點，邊上有長度。有m個詢問，每次給你k_i個點，請問拜訪這k_i個點的最短距離是多少?可以從任一點出發，不必回到起點。
- sum(k_i)與n大約是同一個量級，需要O(nlogn)的時間以內。
- 這是一個難題，需要一些知識以及競程的技巧
- 把一棵樹的所有點走一遍回到起點，是樹上所有邊長總和的2倍
  - doubling tree edges forms a Eulerian graph(可以一筆畫)
  - 如果不要回到起點，最短路徑 = 2*total_tree_edge - diameter
    - diameter就是最遠的兩點
- 這個性質在tree的點的子集合依然成立，等價於spanning這些點的subtree上的相同問題。
- 所以對每個query我們要什麼?
  - 走訪這些點的Euler tour長度與subtree diameter

- Euler tour的順序就是在原tree上的順序
  - 所以先走遍一次，記錄每個點的順序，查詢時先將這些點依此順序排序$(v_1, v_2, \cdots, v_k)$, Euler tour = $d(v_1, v_2)+d(v_2, v_3)+\cdots+d(v_k, v_1)$
- 任兩點距離怎麼求?
  - LCA，前處理後O(1)
- 剩下這k點的直徑怎麼算?
  - tree的直徑有個性質：從任一點出發，最遠的點必是某直徑的一端點
  - 找直徑演算法：farthest of farthest，從任一點找最遠，再從這個最遠點找最遠就是直徑，
  - 所以算2k個距離就可以找出這k個點的直徑
- O(klogk)可以完成一次查詢，所有查詢可以在O(nlogn)完成

- 這題頗難，code也沒有太簡單
- 當年只有兩人答對(2018全國賽)

其他 --- APCS與競賽的差異

- APCS與程式比賽，除了範圍比較有限之外，最大的差異是後測
- 目前台灣大多數比賽已經採取online judge，選手在賽中可以得知該題是否已經通過。但APCS是後測，在考試期間只會以範測來測試繳交的程式，正式的測資要在考完之後才進行測試評分。
- 從過往看起來，APCS對於後測採取了一些措施，但對考生還是有比較困難的地方
  - 沒有現場提問：所以APCS的題目會寫得比較清楚，減少提議混淆的可能
  - 範例比較友善，很少有小陷阱，輸出較單純：避免因為一些小錯誤，考生難以在考試中即時發現。
  - 單測資計分，每通過一筆測資得5分：而不是像競賽必須整組通過才能得到分數。
  - 執行時間限制給的比較寬鬆，因為對於大測資的題目，考生無法得知精確的執行時間。(以經驗看，師大後測的機器跑得很快，比考生用的電腦快)
- APCS實施9年了，雖然大家不太喜歡後測，但大致沒有太大的問題。

初學者參加APCS應該注意的事

- 常常在網路上聽到一些人參加APCS後，自怨自艾題目看錯或是忘了什麼之類的。其實很容易避免的
  - 這些人大多是初學者，去考之前也沒有把考場的規則與環境弄清楚
    - 可能是反正可以考很多次，又不需要報名費，就沒有對考試很慎重
- 初學者去考，最多就做兩題或是加上後兩題的第一子題。
  - 其實重點是好好做前兩題，後面那個並不會提升你的級分，頂多是拔保險分。主辦單位應該都算好了的。
  - 所以時間是非常充分的(如果是很認真的去參加)
- 看清楚題目，注意範例測資的測試結果。
- 範測很重要的：
  - 釐清題意
  - 輸出入格式
  - 避免stupid error
- APCS的第一與第二題的題意都寫得很直接明瞭，而且幾乎都給了很友善的例子(與競賽不同)，只要好好的看題目並注意範測出來的結果，幾乎沒疏忽的可能。例如這次第一題，題目中範例的說明已經就告訴你怎麼算了

如何自我驗證程式

- 從根本上來說，驗證程式是一件很困難的事情，甚至從理論上來說沒有100%保證的方法。
- 從出題者來說，很多題目出測資要比寫解答程式困難N倍。
- 就實際面來說，面對APCS後測的狀況
  - 執行時間方面，一部分只能相信敵人(主辦單位)。相信自己的時間複雜度夠好就可以通過。
    - 這一點在C++比較沒有問題，在Python可能會有一些問題。因為Python沒有編譯器優化，相同的算法，程式寫得好與不好，可能會差到數倍的時間。
- 在賽中要驗證自己的程式的話，首先時間必須足夠。2.5小時對一般考生來講，不是很多，做題目的時間都不見得夠，如何能有時間自己生測資做驗證？
- 如果程度比較好，時間夠，是可以做一些簡單的驗證。

- 複雜的測資通常很難在短時間產生
  - 需要思考你的敵人，也就是想要擋掉的寫法
- 純粹隨機生成的測資則不難
  - 熟悉幾個random函數就可以根據題目的輸入格式產生
- 正解輸出怎麼辦
  - 通常資料結構與演算法的題目，都有其它的寫法，尤其是複雜度比較差的Brute Force (暴搜枚舉)的方法
  - 對於要求O(nlogn)的題目，我們可以寫一個簡單的$O(n^2)$甚至$O(n^3)$的方法來產生正確解答，比對我們的那個效率高的程式的正確性。這個動作稱為對拍。
- 對拍在我們思考難題時也很有用。例如我們猜測某題的解法(例如greedy)時，要用理論推導出正確的性質有時很難，這時先產生一些小的隨機測資，用對拍來驗證我們的猜想。雖然通過對拍未必一定正確，但發現反例就一定不對。

謝謝