# PythAPCS45

APCS 實作題四五級的 Python 題解

## 前言

PythAPCS45 是一系列免費的教學影片。內容為 APCS 實作四五級的考古題 Python 題解，蒐集的題目包含目前(2021 年底)在 ZeroJudge 與 AP325 上歷屆考試的三四題共 28 題。 影片中講解的形式是針對資料結構與演算法的初學者，包含從看到題目後的思考方式到將程式建構出來的過程，以及搭配 Python 的資料結構。

作者：吳邦一

---

註

- 此為教學影片錄製腳本，非完整講義。
- Python 初學者請參考 PyAPCS123，該系列為針對無程式基礎者學習 Python APCS 從一級到三級的教學影片。
- AP325 講義為協助 APCS 實作題三級程度的學習者進步到五級所編撰的免費講義，採用 C++，講義取得位置為 https://drive.google.com/drive/folders/10hZCMHH0YgsfguVZCHU7EYiG8qJE5f-m?usp=sharing
  目前講義內的題目有放在台中一中的自動裁判機上提供解題練習。https://judge.tcirc.tw/
- 有問題或建議歡迎反映到臉書社團"APCS 實作題檢測"



## 目錄

標註 A325 開頭的題目為 AP325 的例習題編號，其餘為 ZeroJudge 題目編號。

# 0. b966. 線段覆蓋長度(AP201603_3)

(ZJ 看題目)

數線上給 N 個線段，求被任一個線段覆蓋到的總長度。也可以看成有 N 個區間，求這些區間聯集的長度。每個線段的長度是右端點座標減去左端點座標，本題的線段有可能左右端點相同(長度為 0)。

(輸入怎麼讀、要做什麼事、要輸出什麼)

對於第二子題的小座標範圍，我們可以用以下模擬的思維來設計流程。把[0,1000]這個區間看成 1000 個長度為 1 的區段，以一個陣列 seg[]來記錄每一個區段是否有被線段覆蓋。具體來說：

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

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

```
n = int(input())
seg = [0 for i in range(1000)]
for i in range(n):
    s,t = map(int,input().split())
    for x in range(s,t):
        seg[x]=1
print(sum(seg))
```

以上模擬法的時間複雜度是 O(MN)，無法適用於第三個子題。第三個子題的座標範圍 M 與線段數 N 都很大，所以需要一個更好的方法。思考的出發點是將互相重疊的線段聯集起來變成一根線段，那全部的線段就會變成若干不相交的線段，他們的長度總和就很容易計算了。問題在於要如何找出重疊的線段並將其聯集。

我們定義一個結構包含兩個欄位(left,right)來表達一個線段的左右端點，將所有線段存在此結構的陣列 seg[]中。假設 cur 是目前左端點最小的線段，只要有一個線段滿足 seg[i].left <= cur.right，

seg[i]就與 cur 有相交，兩者可以合併成一個新的線段；反之，剩下線段的左端都大於 cur.right，那麼就沒有線段會與 cur 相交。此時，我們可以將 cur 的長度加入最後的總和之中，並將其移除。我們只要再挑選剩餘線段中左端點最小的線段當作下一個 cur，然後重複以上的程序直到沒有剩餘線段為止。

執行以上程序的第一個關鍵在於每次都要找出左端最小的 cur，我們也可以發現，當合併發生時，cur 合併後的線段依舊會是下一回合左端最小的線段，也就是合併後的線段就是下一回合的 cur！此外，在尋找「是否有線段的左端小於等於 cur 的右端」時，我們其實也只要找剩下的線段中左端最小者就可以，因為如果有線段與其相交，左端最小者一定會相交，而左端最小者如果不與 cur 相交，那其他的也不會。因此，整個程序的關鍵都是**把左端最小的線段取出來做比較，所以，我們可以一開始將所有線段依照左端點排序**，這樣每一回合可以直接取到左端最小的線段。至於將 s[i]與 cur 合併的方式，因為已知兩者相交，cur 的左端又比較小，所以將右端換成兩者比較大的右端就可以了。

```
n = int(input())
seg = []
for i in range(n):
    s,t = map(int,input().split())
    seg.append((s,t)) # or [s,t]
seg.sort() #key = lambda x: x[0]
left = 0
right = 0
length = 0
for (s,t) in seg:
    if s <= right :
        right = max(right,t)
    else:
        length += right - left
        left,right = s,t
# end for
length += right - left # don't forget
print(length)
```

# 1. b967. 血緣關係(AP201603_4)

(ZJ 為多測資版)

輸入一個家族的親子關係，本題要計算的是最遠的血緣距離。親子關係是一個樹狀圖，所以這一題要計算的其實就是所謂**樹狀圖的直徑**。在圖形理論的術語中，兩點之間最短路徑的長度稱為兩點之間的距離，一個連通圖中，任兩點都有一個距離值，所有距離值中最大的就是該圖形的直徑。求圖上的距離有一些著名的演算法，本題的圖是樹狀圖，樹狀圖的距離相對簡單些，也有效率更好的演算法。

計算樹的直徑有多種方法，但都需要一些圖形與樹狀圖的概念，所以這個題目對於沒有接觸過的人會有些困難。以下先說明比較常用的 DP(動態規劃)的方法。

一條在樹上的路徑必然可以看成兩段(包含其中一段為空的退化情形)：從某點開始往上走，到達某點後再往下走，這個分段的點我們姑且稱它為這條路徑的最高點。令 L(v)為「以 v 點為最高點的最長路徑長度」，假設我們對每一點 v 都求出 L(v)，那麼直徑就是在所有 L(v)中取最大值，以下我們的目標專注於如何計算 L(v)。

一個點 r 的高度 H(r)定義為「從 r 點往下走的最長路徑」，也就是一端點在 r 而另一端點在 r 的子孫中的最長路徑。以遞迴的思考，H(r)可以用以下遞迴式計算(定義)：

當 r 點無孩子時，H(r) = 0；否則，

H(r) = max{ H(v)+1: for all children v of r}。

因為 L(r)是以 r 為最高點的路徑長度，如果 r 有兩個孩子以上，L(r)顯然就是：(最大的孩子高度 + 1) + (第二大的孩子高度 + 1)；而在只有一個孩子的時候 L(r) = H(r)；而沒有孩子的時候則 L(r) = 0。

recursion version suffers from stack overflow

```
import sys
sys.setrecursionlimit(100000)

def dfs(r):
    global child, dia, hei
    for v in child[r]:
        dfs(v)
        dia = max(dia, hei[r]+hei[v]+1)
        hei[r] = max(hei[r], hei[v]+1)
#main
while True:
    try: n = int(input())
    except: break
    child = [[] for i in range(n)]
    notr = [0 for i in range(n)]
```

```
    for i in range(n-1):
        s,t = map(int, input().split())
        child[s].append(t)
        notr[t] = 1
    root = 0
    while notr[root]: root += 1
    dia = 0
    hei = [0 for i in range(n)]
    dfs(root)
    print(dia)
```

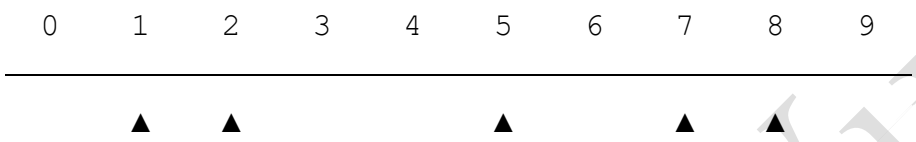
以下是 Bottom-up 的解法。但 ZJ 上這一題 Python 版本是無法通過全部測資的。

答案沒問題。

```
# 201603 q4, bottom-up
while True:
    try: n = int(input())
    except: break
    deg = [0]*n
    parent = [-1]*n
    height = [0]*n
    for i in range(n-1):
        s,t = map(int, input().split())
        parent[t] = s
        deg[s] += 1
    # bfs
    que = [v for v in range(n) if deg[v]==0]
    front = 0
    diameter = 0
    while front < len(que):
        v = que[front]; front += 1
        p = parent[v]
        if p<0: break
        diameter = max(diameter, height[p]+height[v]+1)
        height[p] = max(height[p], height[v]+1)
        deg[p] -= 1
        if deg[p]==0: que.append(p)
    # end while
    print(diameter)
```

## 2. c575. 基地台(AP201703_4)

輸入數線上 N 個點的座標以及一個正整數 K，想要用 K 個相同長度的線段蓋住此 N 個座標點，要計算線段長度最小為多少。以下是一個 N=5 的例子，五個服務點的座標分別是 1、2、5、7、8。




假設 K=1，最小的直徑是 7，基地台架設在座標 4.5 的位置，所有點與基地台的距離都在半徑 3.5 以內。假設 K=2，最小的直徑是 3，一個基地台服務座標 1 與 2 的點，另一個基地台服務另外三點。在 K=3 時，直徑只要 1 就足夠了。

先考慮以下問題：給數線上 N 個點座標，請問用 K 根長度 X 的線段是否能蓋住所有輸入點。我們以貪心法則來思考。考慮座標值最小的點 P，如果有一個解的最左端點小於 P，我們可以將此線段右移到左端點對齊 P，這樣的移動不會影響解的合法性，因此，可以得到以下結論：「如果此問題有解，一定有一個解是將一根線段的左端對準最小座標點。」

```
# K 根長度 X 的線段是否能蓋住所有輸入點
def enough(x):
    global p, k
    num = 1  # number of segments
    endline = p[0] + x  # already covered
    for point in p:
        if point <= endline: continue
        num += 1
        if num > k: return False
        endline = point + x
    return True
```

有了這個函數，我們可以逐步來測試 enough(1)、enough(2)、…，直到第一個回傳 True 的 enough(x)，答案就是 x。

接下來我們觀察一個很簡單的事實：**如果長度 X 的線段可以，則超過 X 長度的線段一定也可以**。我們只要找出最小可以通過驗證的長度，因此，根據這個單調性，我們可以使

用二分搜的技巧來搜尋，整個座標範圍是 10 億，使用二分搜每次可以把搜尋範圍減半，所以只需要最多 30 次的搜尋就可以找出答案。二分搜的完整程式。

```
# AP201703_4, using function
def enough(x):
    global p, k
    num = 1  # number of segments
    endline = p[0] + x
    for point in p:
        if point <= endline: continue
        num += 1
        if num > k: return False
        endline = point + x
    return True
#main
n, k = map(int, input().split())
p = [int(x) for x in input().split()]
p.sort()
imax = p[n-1] - p[0]
# binary search
jump = imax//2
d = 0 # max length that is not enough
while jump > 0:
    while d+jump<=imax and not enough(d+jump):
        d += jump
    jump >>= 1
# end while
print(d+1)
```

另外一種二分搜的寫法：

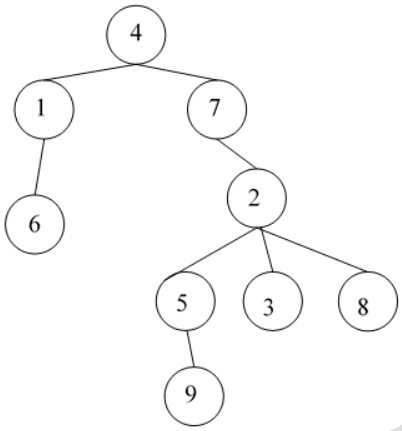
```
# AP201703_4
n, k = map(int, input().split())
p = [int(x) for x in input().split()]
p.sort()
low = 1;
up = p[n-1] - p[0]
# binary search
while low < up:
    diameter = (low + up) // 2
    # greedy to find the min number of segment
    num = 1  # number of segments
    endline = p[0] + diameter
    for point in p:
        if point <= endline: continue
        num += 1
        if num > k: break
        endline = point + diameter
    if num > k :
        low = diameter + 1
    else:
```

```
        up = diameter
# end while
print(low)
```

# 3. c463. apcs 樹狀圖分析(AP201710_3)

本題是要找出一個有根樹的根節點以及每一個節點的高度，題目中給了有根樹的一些定義以及遞迴求出節點高度的提示。

**提示**：輸入的資料是給每個節點的子節點有哪些或沒有子節點，因此，可以根據定義找出根節點。關於節點高度的計算，我們根據定義可以找出以下遞迴關係式：(1)葉節點的高度為 0；(2)如果 v 不是葉節點，則 v 的高度是它所有子節點的最大高度加一。也就是說，假設 v 的子節點有 a, b 與 c，則 h(v)=max{ h(a), h(b), h(c) }+1。以遞迴方式可以計算出所有節點的高度。




這一題中的第一個小問題是要找樹根，這只要輸入資料後找出哪一個節點沒有 parent 就行了。主要考的是求節點高度，這也只是基本的遞迴定義：(1)葉節點的高度為 0；(2)如果 v 不是葉節點，則 v 的高度是它所有子節點的最大高度加一。

- input and set parent[v], deg[p]
- Buttom-up traversal，when visiting v with parent p:
  h[p] = max(h[p], h[v]+1)
- sum up all heights。

```
# bottom-up
n = int(input())
parent = [-1]*n
deg = [0]*n
height = [0] * n
for i in range(n): # changed to 0-indexed
    a = [int(x)-1 for x in input().split()]
    deg[i] = len(a)-1 #first is num of children
    for j in range(1,len(a)):
```

```
        parent[a[j]] = i
que = [x for x in range(n) if deg[x]==0]
front = 0
root = parent.index(-1) # the node with parent -1
print(root+1)
while front < len(que): # queue is not empty
    v = que[front]; front += 1
    p = parent[v]
    if p<0: break
    height[p] = max(height[p],height[v]+1)
    deg[p] -= 1
    if deg[p]==0: que.append(p)
print(sum(height))
```

Top-down DFS cannot pass all test cases in ZJ。

```
def findh(r):
    global child, height
    for ch in child[r]:
        findh(ch)
        height[r] = max(height[r], height[ch]+1)
    return
#main
n = int(input())
child = [[]] * n
height = [0] * n
isroot = [True] * n
for i in range(n):
    child[i] = [int(x)-1 for x in input().split()]
    child[i].pop(0)
    for c in child[i]:
        isroot[c] = False
root = 0
while not isroot[root]: root += 1
print(root+1)
findh(root)
print(sum(height))
```

# 4. c471. apcs 物品堆疊(AP201710_4)

(ZJ 測資有誤 f(i)=0)

輸入 N 個物品的重量與取用次數，要將這些物品排成一個順序，每個物品的取用次數乘以在它之前物品的總重量是該物品所需消耗的能量，排出的順序要使得所有物品消耗的總能量最小。

直覺地看，重量大的物品應該排在後面，因為越後面需要被抬起來的次數越少，另一方面需求次數多的應該放前面，因為放後面的話它需要抬起的重量越大。在兩個因素的情形下該如何決定？又如何證明所排順序是最佳解？我們先看第 1 子題 N=2 且 f(i)=1 的情形，從題目中給的例子也可看出在此條件下，重的放後面就對了，事實上此狀況下的答案就是比較輕的重量。

接著思考如何推廣到第 3 子題的情形，也就是 N=1000 但 f(i)=1 的情形，是不是重量輕的放前面就會是答案呢？假設有一個最佳解的重量順序 w **並不是從輕的排到重的**，那麼一定有兩個相鄰位置 w[x]>w[x+1]，如果我們把這兩個交換會讓消耗能量發生什麼變化？因為只交換相鄰兩個物品，對於 x+1 之後的取用時所需的總重是沒改變，對於 x 之前也沒改變，所以改變只有 w[x+1]會多算一次，而 w[x]會少算一次，也就是變化量是 w[x+1] - w[x] < 0，我們可以得到結論：這樣的交換會降低總能量，所以它一定不是最佳解！也就是說，在最佳解重量順序 w 中，相鄰的兩個都一定是 w[x] ≤ w[x+1]。那麼，對於第 3 子題來說，將輸入的重量排序後依照題目定義計算能量就是答案了。

接下來思考第 2 與第 4 子題中 f(i)沒有限制是 1 的情形。直覺來說，取用次數多的應該放前面，我們延續前面的思考方式來觀察相鄰兩個物品：如果有個東西重量又輕且取用次數又多，也就是 w[x] < w[x+1]且 f[x] > f[x+1]，那 x 當然該放前面。那麼，如果 w[x] > w[x+1]但 f[x] > f[x+1]又該如何？考慮交換兩者，同樣地這個交換對於他們之前與之後的都沒改變，交換後 f[x+1]要抬起的重量少了 w[x]而 f[x]要抬起的重量多了 w[x+1]，能量差異為

f[x]*w[x+1] - f[x+1]*w[x]。

如果這個順序是最好的，那麼這個差異量必須不小於 0，也就是說

w[x]/f[x] ≤ w[x+1]/f[x+1]。

顯然在等號成立時，兩物品的順序對能量值沒差別，因此滿足這個順序的也就是最佳解。結論是我們的演算法只要將排序的依據 key 值改成 w[x]/f[x]就行了。

在計算答案時，我們由前往後掃過去，將目前的總重量存於一個變數 total_x，這樣就很容易在線性時間完成計算。當然在每一回合用 sum()求目前總重，這樣會跑到平方的複雜度。

```
n = int(input())
w = [int(x) for x in input().split()]
f = [int(x) for x in input().split()]
a = [(w[i],f[i]) for i in range(n)]
a.sort(key = lambda tup: tup[0]/tup[1])
ans = total_w = 0
for (x, y) in a:
    ans += total_w * y
    total_w += x
print(ans)
```

sort()的 key=lambda 不會用的話，可以學一下，也可以這樣寫：

```
n = int(input())
w = [int(x) for x in input().split()]
f = [int(x) for x in input().split()]
a = [(w[i],f[i]) for i in range(n)]
a.sort(key = lambda tup: tup[0]/tup[1])
ans = total_w = 0
for (x, y) in a:
    ans += total_w * y
    total_w += x
print(ans)
```

這題其實是 minimal waiting time (weighted version)的改包裝。

ZJ 的測資有誤，有些 f(i)=0，要通過的話，先過濾掉 f(i)=0 的部分

```
n = int(input())
w = [int(x) for x in input().split()]
f = [int(x) for x in input().split()]
w2 = []
f2 = []
for i in range(n):
    if f[i]!=0:
        f2.append(f[i])
        w2.append(w[i])
a = [[w2[i]/f2[i], w2[i], f2[i]] for i in range(len(f2))]
a.sort()
ans = total_w = 0
```

```
for [r, x, y] in a:
    ans += total_w * y
    total_w += x
print(ans)
```

# 5. f638. 支點切割(AP201802_3)

題目的意思是輸入一個大小為 N 陣列，要找其中一點將陣列切成左右兩塊，然後針對左右兩個子陣列繼續切割，切割的終止條件有兩個：格子數小於 3 或切到給定的層級 K 就不再切割。而切割的要求是讓左右各點人口數與到切點距離的乘積總和差異盡可能的小，也就是說，要找出切點 m，使得∑ p[i]×(i-m)的絕對值越小越好。注意切點不可在兩端點。

這個題目有兩個重點：

- 找切點之後對左右兩塊遞迴求解；
- 距離與人口數的乘積的總和，可以看成計算槓桿的力矩，所以切割點的找法是計算讓兩邊最平衡的支點。

我們來考慮如何有效率的找支點。提升計算效率的主要原則是減少重覆計算。對於一個支點算力矩需要包含每一個點的人數與距離，這似乎沒有什麼可以節省的了，但是每次移動支點都重新計算就有商榷的餘地了。假設支點在 m 的左右力矩分別是 l_tor[m]與 r_tor[m]，當支點往右移動一格時，根據定義，左邊的每一個點到支點的距離都會加一，而右邊的每一個點到支點的距離都會減一，也就是說：

- l_tor[m+1] = l_tor[m]+(p[left]+p[left+1]+…+p[m])；
- r_tor[m+1] = r_tor[m]-(p[m+1]+p[m+2]+…+p[right-1])；

運用前綴和(prefix sum)的觀念，假設 psum[i]為從左邊 p[left]到 p[i]的總和，而 ssum[i]為 p[i]到 p[right-1]的總和，我們可以簡化為：

- l_tor[m+1] = l_tor[m] + psum[m]；
- r_tor[m+1] = r_tor[m] - ssum[m+1]；而
- psum[m] = psum[m-1] + p[m];
- ssum[m+1] = ssum[m] - p[m];

也就是說，只要記住前綴和與後綴和，每次移動一格之後的左右力矩可以用一個加減法計算出來，而前綴和與後綴和本身也可以用一個加減法計算出來。所以算出每一個點的力矩只需要花線性的時間。

```
# [s, t-1]
def cut(s, t, k):
    global p
```

```
    if t-s < 3 or k < 1: return 0
    fulcrum = s+1 # 支點從 s+1~t-2
    psum = p[s] # prefix sum
    l_tor = p[s] # left torque 力矩
    ssum = sum(p[s+2:t]) # suffix sum
    r_tor = sum(p[i]*(i-s-1) for i in range(s+2,t))
    dif = abs(r_tor - l_tor)
    while fulcrum < t-1: # each posssible fulcrum
        psum += p[fulcrum]
        l_tor += psum
        fulcrum += 1
        r_tor -= ssum
        ssum -= p[fulcrum]
        res = abs(l_tor - r_tor)
        if res >= dif: break # decreasing and then increasing
        dif = res
    fulcrum -= 1
    return p[fulcrum]+cut(s, fulcrum, k-1)+cut(fulcrum+1, t, k-1)
# main start
n,k = map(int, input().split())
p = [int(x) for x in input().split()]
print(cut(0, n, k))
```

這題還有一個求切點的方式是運用重心的原理，所有點的力矩總和等於總重量於重心產生的力矩。注意求出來時通常並非整數，此時要決定左右兩點其中之一。

# 6. AP325 P-6-19.階梯數字(AP201802_4)

(這題在中一中的 TCFSH 的裁判機上，(TCFSH CIRC 編號 d080.)

題目中定義了階梯數字是一個正整數且從高位往低未看過去，每一位的數字只會相等或變大，不會變小。輸入一個數字 N，問不大於 N 有多少個階梯數字。要判斷一個數字是否是階梯數字並不難，只要把每一位數字拆解出來再檢查是否單調上升就可以了，題目的提示中也提到，數字範圍小的時候可以枚舉，但 N 很大時就必須想快速的方法了。

先看枚舉的方法，我們只要會檢查一個數字是否是階梯數字，然後由 1 開始一一檢查就可以了。檢查的方法可以用字串也可以用數字，以下是用字串處理的方式。

```
# enumerating
def isinc(k):
    ks=str(k)
    for i in range(1,len(ks)):
        if ks[i] < ks[i-1]:
            return 0
    return 1

n=int(input())
total=0
for i in range(1,n+1): total+=isinc(i)
print(total)
```

由 DP 的思考先找遞迴式：

令 step(i,j)是 i 位數字且以 j 開頭的階梯數字總數，其中 j = 0 就是所有小於 i 位的總數。 2XX, 22X, 23X, 24X, ..29X

Terminal case: step(1,0)=0; and step(1,j)=1 for j!=1.

Otherwise: step(i,j)=step(i-1,j)+step(i-1,j+1)+…+step(i-1,9).因為下一位必須 >= j。

在 step()算出來後，我們要求的是不大於 n 的階梯數字總數。若 n 的每一位數字為 n[0],n[1],…n[d-1]，我們分割成以下幾類來計算, n=2578

- 比 n[0]小開頭的 d 位數 1XXX, 0XXX; 2578. m=2549
- 以 n[0]開頭而第 2 位小於 n[1]的 d-1 位數 22XX, 23XX, 24XX
- n[0]n[1]開頭而第 3 位小於 n[2]的 d-2 位數 255X, 256X
- …
- n[0]n[1]…n[d-3]開頭而第 d-1 位小於 n[d-2]的 1 位數。
- n[0]n[1]…n[d-2]開頭而第 d 位**不小於** n[d-1]的 1 位數。2577, 2578

當中如果碰到 n[i]<n[i-1]則以後都是 0 個，因為 n[0]n[1]…n[i]並非階梯數字。注意最後一位是找不小於，前面都是找小於。

把上述思考直接寫成遞迴程式如下：

```
def step(i,j): # i digit starting with j
    if i==1: # terminal case
        if j==0: return 0
        return 1
    if j==9: return 1 # terminal case
    total = 0
    for p in range(j,10): # next digit must >=j
        total += step(i-1,p)
    return total

n = [int(x) for x in input()]
d = len(n)
ans = 0
for j in range(n[0]): # first digit <n[0]
    ans += step(d,j)
flag = True
for i in range(1,d): # with prefix n[0:i]
    if n[i] < n[i-1] :
        flag = False
        break
    for j in range(n[i-1],n[i]):
        ans += step(d-i,j)
if flag: ans += 1
print(ans)
```

遞迴版的程式跑的慢(雖然可能目前測資可以過)，寫成 top-down DP 是非常容易的事，標準的做法，建立一個表格儲存遞迴呼叫所可能的狀況，初值設為不可能的值(這裡可以用-1)，遞迴函數中先檢查表格，若已算過，直接回傳；否則遞迴呼叫後存入表格(以防下次重複計算)。

```
# top-down dp
def step(i,j): # i digit starting with j
    if table[i][j]>=0:
        return table[i][j]
    if i==1: # terminal case
        if j==0: return 0
        return 1
    if j==9: return 1 # terminal case
    total = 0
    for p in range(j,10): # next digit must >=j
        total += step(i-1,p)
    table[i][j] = total
    return total

n = [int(x) for x in input()]
d = len(n)
```

```
table = [[-1]*10 for i in range(d+1)]
ans = 0
for j in range(n[0]): # first digit <n[0]
    ans += step(d,j)
flag = True
for i in range(1,d): # with prefix n[0:i]
    if n[i] < n[i-1] :
        flag = False
        break
    for j in range(n[i-1],n[i]):
        ans += step(d-i,j)
if flag: ans += 1
print(ans)
```

非遞迴版的 DP 推論比較麻煩，請參考 AP325 中的說明或自行研究。

```
n = [int(x) for x in input()]
n_dig = len(n)
# initial a 2d array for steppiny table
step = [[0 for x in range(10)] for y in range(n_dig+1)]
for j in range(1,10): step[1][j] = 1
step[1][0] = 0
for i in range(2, n_dig+1):
    step[i][9] = step[i-1][9]
    for j in range(8, -1, -1):
        step[i][j] = step[i][j+1] + step[i-1][j]
# counting
total = sum(step[n_dig][:n[0]])
i = 1
while i < n_dig:
    if n[i] < n[i-1]: break
    total += sum(step[n_dig-i][n[i-1]:n[i]])
    i += 1
if i == n_dig: total += 1
print(total)
```

# 7. AP325 P-5-4.反序數量(AP201806_4)

(這題在中一中的 TCFSH 的裁判機上，TCFSH CIRC 編號 d064.)

一個數列的反序數量(inversion)是指有多少對(i,j)滿足 i<j 且 a[i]>a[j]，意即前大後小。計算反序數是一個教科書的分治(divide and conquer)經典題。直接對於每一個 a[j]去算一算前方有多少大於他的數字，可以得到一個很簡單的 O($n^2$)算法。如果從前往後掃，利用一個動態資料結構儲存已經看過的數字，如果可以在 O(log(n))查詢比 a[j]大的有幾個，顯然我們可以做到 O(nlogn)的時間複雜度，所謂動態是只要能夠加入資料。確實有些樹狀的資料結構可以做，但不是我們這裡的重點，以下我們要講分治的做法與思考方式。

一個典型的分治，是將問題的輸入資料均勻切割成兩半，對兩半資料各自遞迴求解，然後再將兩邊的解合併成最後的答案。以本題來說長度 n 的數列要計算反序數，將數列切成左右兩半各自(遞迴)計算反序數，然後我們要做的是如何計算跨在左右兩端的反序對，事實上，這個合併的計算不需要太精妙厲害，就可以設計出整體複雜度還不錯的程式。

在計算跨左右反序對有很多做法，先看比較簡單的寫法。下面的程式中，我們使用一個暫存的 list，每個元素除了原來的資料外，我們補了一個欄位，前(左)半部的元素這個欄位給 0，後(右)半部的元素補 1。然後做排序。掃描排序過的 list，一面紀錄有多少來自後半部的元素，這樣很容易算出一個在左一個在右的反序對有多少對。n 個資料做一回合是 O(nlogn)，整體的複雜度是 O($nlog^2n$)，已經可以通過這一題。

```
# worse method
def sol(ar):
    if len(ar) < 2: return 0
    mid = len(ar) // 2
    inv = sol(ar[:mid]) # left part will be sorted
    inv += sol(ar[mid:]) # also right part
    # find inversion for cross two parts
    t = [[x,1] for x in ar] # (x,1) for right part
    for i in range(mid): # left part
        t[i][1] = 0
    t.sort()
    right = 0 # num of element from right
    for [x, p] in t:
        if p==0:
            inv += right # num of < x in right part
        else:
            right += 1
    return inv
# main program
n = int(input())
a = [int(x) for x in input().split()]
```

```
print(sol(a))
```

換一種寫法。我們將右半部排序，對左半部的每個元素用二分搜去找右半部有多少小於他的元素，這樣一回合也是 O(nlogn)，整體的複雜度也是 $O(n\log^2 n)$，分治是不是很簡單？

```
import bisect
def sol(ar):
    if len(ar) < 2: return 0
    mid = len(ar) // 2
    left = ar[:mid]
    inv = sol(left) # left part
    right = ar[mid:]
    inv += sol(right) # right part
    # find inversion for cross two parts
    right.sort()
    for x in left: # num of < x in right part
        inv += bisect.bisect_left(right,x)
    return inv
# main program
n = int(input())
a = [int(x) for x in input().split()]
print(sol(a))
```

這題可以做到 O(nlogn)，關鍵因素是不要每次重新做排序，對於每次遞迴，在計算跨左右的反序對呼叫時，順便將數列排序完成，因此不需要在特別額外呼叫一次排序來做。將兩個排好序的數列在 O(n)時間完成合併，稱為 merge，其實是個很基礎的程序，而在合併兩序列時，就可以算出有多少跨左右的反序對。整合這個過程設計出來的程式其實非常類似於合併排序法。

```
# divide and conquer, return sorted list
def sol(ar):
    if len(ar) < 2: return 0
    mid = len(ar) // 2
    a1 = ar[:mid]
    a2 = ar[mid:]
    inv = sol(a1) # left part will be sorted
    inv += sol(a2) # also right part
    # merge a1 and a2 into ar
    i = i1 = 0
    for x in a2:
        while i1 < len(a1) and a1[i1] <= x:
            ar[i] = a1[i1]
            i += 1; i1 += 1
        inv += len(a1) - i1 # num of > x in a1
        ar[i] = x
        i += 1
```

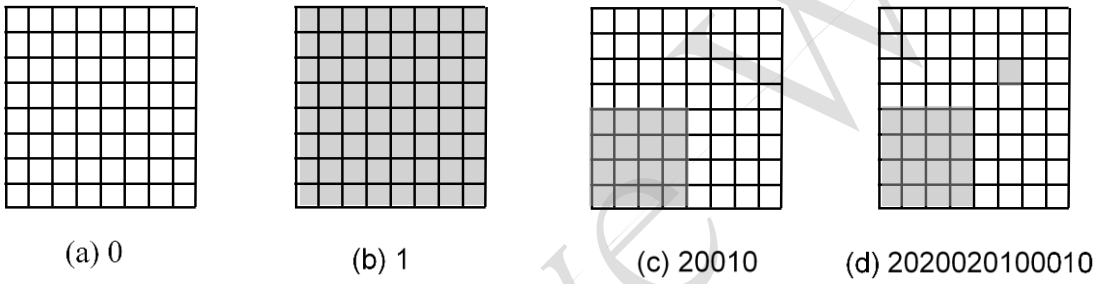
```
    while i1 < len(a1):
        ar[i] = a1[i1]
        i += 1; i1 += 1
    return inv
# main program
n = int(input())
a = [int(x) for x in input().split()]
print(sol(a))
```

有關分治，可以參考 AP325 中的說明，也可以再找其他教材了解。

# 8. f637. DF-expression(AP201810_3)

本題的題意是說，有一種編碼方式可以將一個 0/1 方陣編碼成一個由 0,1,2 組成的字串，輸入方陣大小以及編碼字串，要找出方陣中 1 的個數，就是黑色面積。基本想法是要把他解碼回去，所以重點就是這個編碼方式。根據題目，這個編碼是由遞迴的方式定義的，遞迴的規則是：

- 0 代表整塊都是 0；
- 1 代表整塊都是 1；
- 2 的話就要分成四塊，遞迴求出後再將字串接在一起。




當我們要解碼回去時，也就可以直接依照編碼的定義，因為我們只要算黑色面積，所以解碼回去時只要找出面積，不必產生對應的二維陣列。假設目前的大小是 n：

- 字串如果是 0，黑色面積是 0，結束；
- 字串如果是 1，黑色面積是 n*n，結束；
- 如果字串開頭是 2，就要將字串分成四段 $S_1S_2S_3S_4$，然後遞迴計算每一段的黑面積，然後加起來就是答案，其中這四小段的每一段的尺寸大小會變成 n/2。

前兩個情形沒問題，第三點乍看之下不容易處理，不知道要怎麼切割成四段！既沒有分隔符號，每一段的大小也不固定。但仔細一想，根本就不必事先知道如何切割，因為分成四段也是一段一段依序做，在做完第一段的時候，自然會知道下一段從哪裡開始。

想清楚以後，這一題只要會遞迴就會寫了。遞迴是副程式直接或間接呼叫自己，大部分都是直接呼叫。遞迴很有用，是非常重要的計算思維方式，通常會覺得遞迴困難，只是因為遞迴跟我們從小到大的思維方式不太一樣，其實了解以後就會發現遞迴並不難，寫的好，程式碼還可以特別的精簡。

```
def rec(size):
    global index, instr, ans
    x = instr[index]
```

```
    index = index+1
    if x == '1':
        ans = ans+size*size
        return
    if x == '0': return
    size = size // 2
    for i in range(4): rec(size)
    return
# main start
index=0
ans = 0
instr = input()
n = int(input())
rec(n)
print(ans)
```

# 9. AP325 Q-6-10.置物櫃出租(AP201810_4)

(這題在中一中的 TCFSH 的裁判機上，TCFSH CIRC 編號 d075.)

n 個客戶，若 T = f(1)+f(2)+…+f(n)，現在剩餘量是 M - T，現在需求 S，若 S<=M-T，無須退租，損失 0

否則，要找出若干客戶 B，損失 $L(B)=\sum_{i\in B} f(i)$，滿足$L(B)+M-T\geq S$，問 L(B) 最小是多少。

這顯然是個 0/1 背包問題的退化版本，部分和的問題。給一群整數 Z 與一個數字 X，找出 Z 中某些整數使得總和至少 X，問最小的總和是多少。這個問題其實是 NP-complete，所給數字很大時，其實沒有好的算法可以做，在數字不大時，可以有個 O(nX)的做法，稱為 pseudo-polynomial。

我們用一個 ssum[]，對每一個 f(i)逐一考慮，ssum[j]=True 表示可以湊出 j 的部分和。當考慮到 f(i)時，原來為 True 的還是 True，若原來 ssum[j]=True，則 ssum[j+f(i)]也會是 True。計算時，從 ssum 算到 tem，算完後再交換回來，這樣做可以避免這回合新產生的值干擾到舊的值，是一個常用的技術，另外一個作法就是每回合都開一個 List，但那樣浪費記憶體且時間也會比較差。

```
n,amount,need = map(int, input().split())
item = [int(x) for x in input().split(' ')]
total = sum(item)
need -= amount - total
if need <= 0:
    print(0)
    exit()
ssum = [True] + [False for i in range(need)]
ans = total
for f in item:
    tem = ssum[:]
    for j in range(need-f):
        if ssum[j]: tem[j+f] = True
    for j in range(need-f,need+1): # find the first >=need
        if ssum[j]:
            ans = min(ans,j+f)
            break
    tem, ssum = ssum, tem

print(ans)
```

這一題是可以用一個 list 就完成的，竅門是在檢查 ssum 的時候由大往小走，新產生的都在 j 之後，而 j 由大到小，所以不會遇到這回合新放進來的值。下面這個程式另外

也做了一些小改變，我們將數字先排序，然後一邊做一邊計算總和，每次只從目前的總和往前做，這樣雖無助於複雜度，但有小的幫助。

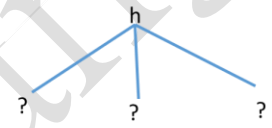
```
n,amount,need = map(int, input().split())
item = [int(x) for x in input().split(' ')]
item.sort()  # small-itenm-first runs faster
total = 0 # current total
ssum = [False for i in range(amount+1)]
ssum[0] = True
for x in item:
    for j in range(total,-1,-1): # from total to 0
        if ssum[j]: ssum[j+x] = True
    total += x
need -= amount - total
if need <= 0:
    print(0)
    exit()
for j in range(need,amount+1):
    if ssum[j]: break
print(j)
```

# 10. f640. 函數運算式求值(AP201902_3)

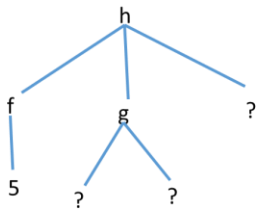
> Q-1-2. 合成函數(2) (APCS201902)
>
> 令 f(x)=2x-3；g(x,y)=2x+y-7；h(x,y,z)=3x-2y+z。本題要計算一個合成函數的值，例如 h(f(5),g(3,4),3)=h(7,3,3)=18。
>
> Time limit: 1 秒
>
> 輸入格式：輸入一行，長度不超過 1000，它是一個 f, g, 與 h 的合成函數，但所有的括弧與逗號都換成空白。輸入的整數絕對值皆不超過 1000。
>
> 輸出：輸出函數值。最後答案與運算過程不會超過正負 10 億的區間。
>
> 範例輸入：
>
> h f 5 g 3 4 3
>
> 範例輸出：
>
> 18

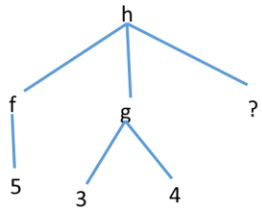
題目中定義了 f, g, h 三個函數，輸入一個三者的合成函數表示式，要求出函數值。這個表示式中所有的括弧與逗點都被去除了，但題目中提示可以使用遞迴的方式唯一決定原來的表示式與函數值。

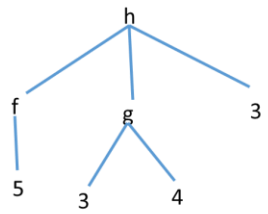
h f 5 g 3 4 3




- h
- h( , , )
- h(f() , , )
- h(f(5) , , )
- h(f(5) , g( , ) , )
- h(f(5) , g( 3 , ) , )
- h(f(5) , g( 3, 4 ) , )
- h(f(5) , g( 3, 4 ) , 3)



這一題如果有遞迴的觀念，其實就非常的簡單：只要根據函數定義直接寫出遞迴函式就可以了。整個輸入是一個表示式，所以我們定義一個函式 eval()，這個函式從輸入讀取字串，回傳函數值。其流程只有兩個步驟，第一步是讀取一個字串，根據題目定義，這個字串只有四種情形：f, g, h 或是一個數字。第二步是根據這個字串分別去呼叫 f(), g(), h()或是直接回傳字串代表的數字。而 f(), g(), h()三個函數的寫法是根據定義它有幾個參數，就呼叫 eval()幾次，然後直接照定義計算值。

```
def eval():
    global index,token
    func = token[index]
    index += 1
    if func == 'f': return f()
    if func == 'g': return g()
    if func == 'h': return h()
    return int(func)   # value

def f():
    x = eval()
    return 2*x - 3
def g():
    x = eval()
    y = eval()
    return 2*x + y - 7
def h():
    x = eval()
    y = eval()
    z = eval()
    return 3*x - 2*y + z

token = input().split()
index = 0
ans = eval()
print(ans)
```

# 11. AP325 Q-3-5.帶著板凳排雞排的高人(AP201902_4)

(TCFSH CIRC 編號 d029.)

這一題有個子題就是

d028. 例題 P-3-4. 最接近的高人(APCS201902, subtask)

差別是沒有帶板凳，我們先來看這一題。(以下說明取自 AP325 P.84)

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

要改善複雜度，要想想是否有沒有用的計算或資料。假設身高資料放在陣列 a[]，如果 a[i-1] ≤ a[i]，那麼 a[i-1]不可能是 i 之後的人的高人，因為由後往前找的時候會先碰到 a[i]，如果 a[i]不夠高，a[i-1]也一定不夠高。同樣的道理，當我們計算到 i 的時候，任何 j<i 且 a[j] ≤ a[i]的 a[j]都是沒有用的。如果我們丟掉那些沒有用的，會剩下甚麼呢？一個遞減的序列，只要維護好這個遞減序列，就可以提高計算效率。單調序列要用二分搜嗎？其實連二分搜都不需要，想想看，當處理 a[i]時，我們要先把單調序列後方不大於 a[i]的成員全部去除，然後最後一個就是 a[i]要找的高人，因為既然要做去除的動作，就一個一個比較就好了。那麼，要如何維護這個單調序列呢？第一個想法可能是將那些沒用的刪除，那可不行，陣列禁不起的折騰就是插入與刪除，因為必須搬動其後的所有資料。想想我們要做的動作：每次會從序列後方刪除一些東西，在加入一個東西。答案呼之欲出了，因為只從後方加入與刪除，堆疊是最適合的。請看以下的範例程式，這個程式非常簡單。

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

```
n = int(input())
oo = 100000001
b = [oo] + [int(x) for x in input().split()]
r_sum = 0
stack = [0] # index
for i in range(1,len(b)):
    while b[stack[-1]] <= b[i]:
        stack.pop()
    r_sum += i - stack[-1]
    stack.append(i)
```

```
print(r_sum)
```

接著看完全解，Q-3-5。要保持單調遞減序列的推論不變，差別是每次我們往前找的時候，要找到往前碰到第一個 > b[i]+p[i]的位置，所以我們不能一路往前走一路刪除，而必須先做一個搜尋，然後再做刪除的動作。此搜尋當然不能太差，因為是單調遞減序列，我們當然就可以使用二分搜。以下是自己寫二分搜的方式。

```
# binary search
n = int(input())
oo = 100000001
b = [oo]+[int(x) for x in input().split()]
p = [0]+[int(x) for x in input().split()]
r_sum = 0
dom = [[oo,0]] # (height,index)
for i in range(1,n+1):
    # binary search the last > b[i]+p[i]
    pos = 0
    jump = len(dom)//2
    while jump > 0:
        while pos+jump<len(dom) and dom[pos+jump][0]>b[i]+p[i]:
            pos += jump
        jump >>= 1
    r_sum += i - dom[pos][1] -1
    # pop useless
    while dom[-1][0]<= b[i]:
        dom.pop()
    dom.append([b[i],i])
print(r_sum)
```

呼叫庫存函數 bisect 來做二分搜，因為是遞減數列，我們將所有的值轉負號，這樣變成遞增了。注意這一題要的答案是每個人與高人之間的人數而非距離(差 1)。

```
import bisect
n = int(input())
oo = 100000001
b = [-oo]+[-int(x) for x in input().split()]
p = [0]+[-int(x) for x in input().split()]
r_sum = 0
dom = [-oo]
ndx = [0]
for i in range(1,n+1):
    pos = bisect.bisect_left(dom,b[i]+p[i])-1
    r_sum += i - ndx[pos] -1
    while dom[-1]>= b[i]:
        dom.pop()
        ndx.pop()
    dom.append(b[i])
    ndx.append(i)
print(r_sum)
```

## 12. AP325 Q-2-7.互補團隊(APCS201906_3)

(TCFSH CIRC 編號 d016.)

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
```

這是一個搜尋的題目，對於一個團隊，我們要搜尋他的互補團隊是否在這個清單中，根據題目的敘述，一個字串中字母出現的順序不一定而且可能會有重複的字母，所以第一件事是要設計一個集合的表示方法，否則表示法不惟一的狀況下，很難做搜尋。本題中集合可能的元素最多就是 26 個大寫英文字母，有多種可能的集合表示方式，常用的有下列三種：

(1) 以一個長度 26 的陣列 c[]表示一個集合，如果第 i 個字母在集合中，則字母 c[i]=1，否則 c[i]=0。

(2) 將集合中的元素字母依序排列在一個字串中。之所以需要排序是因為要使得一個集合的表示法唯一，在搜尋時比較方便。

(3) 類似(1)，但將此序列編碼成一個整數，也就是說以一個整數代表一個集合，整數的每一個位元是 1 或 0 或就代表一個元素在或不在集合中。

第(3)的表示法效率最高，這種方式也用在很多集合的問題上，使用時必須要用到位元運算。我們會示範(2)與(3)兩種方法。先說(2)。我們可以用一個長度 m 的表格來表示每一個可能的字元是否在此字串中，第 i 個為真的時候表示在，假則表示不在。對於一個輸入字串，我們可以用下面的程序建構出這個集合的標準字串以及他的互補字串。

```
tab = [False]*m
for c in input():
    tab[ord(c)-ord('A')] = True
team = ''
comp = ''
for j in range(m):
    c = chr(ord('A')+j)
    if tab[j]:
        team += c
    else:
        comp += c
```

至於搜尋也有多種方法，因為集合的數量多，使用一一比對的線性搜尋顯然並非適當。Python 有 set()函數可以做集合與元素的查詢，他是使用 hash function 所以效率非常高。另外一個方法是用排序與二分搜。

以下是利用 set()做查詢的範例，我們把所有的團隊代表字串建構在 team_s 中，他們的互補字串則放入 set() comp_s 中，最後對所有團隊查詢互補是否在 comp_s 中。

```
m, n = map(int, input().split())
team_s = []
comp_s = set() # set of complement
for i in range(n):
    tab = [False]*m
    for c in input():
        tab[ord(c)-ord('A')] = True
    team = ''
    comp = ''
    for j in range(m):
        c = chr(ord('A')+j)
        if tab[j]:
            team += c
        else:
            comp += c
    team_s.append(team)
    comp_s.add(comp)
#end of input
```

```
total = 0
for s in team_s:
    if s in comp_s :
        total +=1
print(total//2)
```

字串的處理也可以運用集合來做。將每一個可能的字母枚舉檢查是否在輸入字串中，做完了標準字串與互補字串後，分別放在兩個 list 中。剩下的工作是問 a 的每一個字串是否在 b 中，我們可以排序後以二分搜來做，但連續的二分搜不如一路爬過去，五門乾脆講兩個 list 都排序後，各用一個指標，以類似合併的方式一路爬過去。

```
m, n = map(int, input().split())
a = ['' for i in range(n)]
b = ['' for i in range(n)]
for i in range(n):
    s = set(input())
    team = ''
    comp = ''
    for j in range(m):
        c = chr(ord('A')+j)
        if c in s:
            team +=c
        else:
            comp += c
    a[i]=team
    b[i] = comp
#end of input
a.sort()
b.sort()
total = 0
j = 0
for i in range(n):
    while j< n and b[j]<a[i]:
        j += 1
    if (j>=n): break
    if (a[i]==b[j]) :
        total +=1
        j += 1
print(total//2)
```

下面是以一個整數來表示一個集合的方法，利用位元運算來做，第 i 個字元如果在，我們就讓第 i 個 bit 為 1，否則為 0。至於搜尋的方法，我們再用另外一種方法，將兩個 list 合併在一起排序，因為本題保證沒有相同集合，所以排序後相同的會在一起，這時候只要有相同的就知道找到一組互補了。這個程式跑起來最快。

```
m, n = map(int, input().split())
a = []
```

```
ff = (1<<m) -1 # 1111111111
for i in range(n):
    team = 0
    for x in input():
        team |= 1<< (ord(x)- ord('A'))
    a += [team, ff - team]
#end of input
a.sort()
total = 0
for i in range(len(a)-1):
    if a[i] == a[i+1]:
        total += 1
print(total//2)
```

註： (ZJ-e288.互補 CP。這程式可過 75%，因為原題只有大寫，這裡後 20%有小寫，原題沒有重複的集合，ZJ 這裡有重複的集合)

# 13. AP325 Q-3-12.完美彩帶(APCS201906_4)

TCFSH CIRC 編號 d036

Q-3-12. 完美彩帶 (APCS201906)

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

Time limit: 1 秒

輸入格式：第一行為整數 $m$ 和 $n$，滿足 $2 \le m \le n \le 2\times10^5$；第二行有 $n$ 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號是不超過 $10^9$ 的非負整數，每一筆測試資料的顏色數量必定恰好為 $m$。

輸出：有多少段完美彩帶。

範例輸入：

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

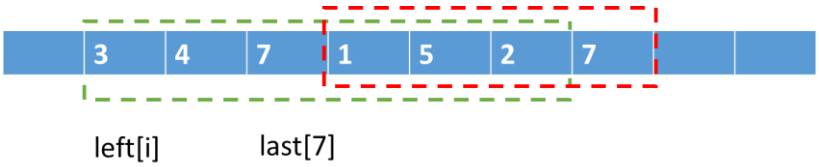
範例結果：

```
3
```

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

給一個有 m 種數字的序列，找出有多少長度為 m 的區間，該區間中數字皆不重複。

我們用滑動視窗想法來做，保持一個視窗(區段)是以 i 為右端的區間，而此區間內皆無相同顏色。每次將右端往右滑動一格，如果沒有同色，則視窗範圍加一；若有同色，必定是右端與視窗中某一位置同色，因此將左端往右移，移到與右端同色的下一個位置即可。




如果目前的視窗[left(i), i] 是"以 i 為右端的最大無同色區段"。
left(i+1) = max(left(i), last[color[i+1]]+1)

問題是：如何知道撞同色呢？

我們要設法紀錄視窗內的顏色，並且能夠迅速查詢某種顏色是否在其中，並且新增與刪除顏色。

(這一題 APCS 考的時候，有一個子題是顏色編號為 1~m，完全解則須顏色編號在 1e9 範圍。)

第一個方法，若顏色一個小範圍的數字，例如 1~m。這時我們可以準備一個長度 m 的表格 last[]，用 last[i]紀錄顏色 i 上次出現的位置。下面這題是 AP325 的 Q-3-11，是極其類似的題目，要輸出的是最長的不同色長度，顏色編號是 0 ~ n。

Q-3-11. 最長的相異色彩帶

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

Time limit: 1 秒

輸入格式：第一行為整數 $n$，滿足 $n \le 2\times10^5$；第二行有 $n$ 個以空白間隔的數字，依序代表彩帶從左到右每一格的顏色編號，顏色編號是不超過 n 的非負整數。

輸出：最長的相異色彩帶的長度。

範例輸入：

```
10
6 4 1 6 0 4 5 0 7 4
```

範例結果：

```
5
```

說明：區間[3, 7]的顏色(1,6,0,4,5)皆不相同。

以下是這樣的程式。

```
n = int(input())
a = [int(x) for x in input().split()] # int
last = [-1]*(n+1) # last position of color i
longest = 0 #answer
left = 0
for i in range(n):
    if last[a[i]] >= left:
        left = last[a[i]]+1
    last[a[i]] = i
    if i-left+1 > longest :
        longest = i-left+1
# endfor
print(longest)
```

在一般情形呢？Python 可以用 set()來製做一個集合，他是用 hash function 來製做的函數，可以快速的做新增刪除與查詢。下面的程式中，在遇到同色時，我們的左端是逐步右移的方式，而未紀錄顏色的上一個出現位置。我們需要用到集合的：初始空集合、新增元素、查詢某元素是否存在、刪除某元素、集合的大小。

```
m, n = map(int, input().split())
a = [int(x) for x in input().split()] # int
color = set() # all color in windows, empty set
total = 0 #answer
left = 0
for i in range(n):
    if a[i] in color: # same color in window
        while a[left] != a[i]: # left go right
            color.remove(a[left])
            left += 1
        left += 1 # this color not removed
    else: #new color
        color.add(a[i])
    if i-left+1 == m : # find a m-color range [ ,i]
        total += 1
# endfor
print(total)
```

我們也可以用 Python 的字典 dict 來做。字典與 set 一樣也是用 hash table，但他可以將每一個元素對應到一個值，紀錄的方法如第三行，取用時的語法與 List 一樣是方括弧。運用字典可以記錄每個顏色的上次出現位置，這個程式很簡短。

```
m,n = map(int, input().split())
a = [int(x) for x in input().split()] # int
last = {x:-1 for x in a} # initial dictionary
total = 0 #answer
left = 0
```

```
for i in range(n):
    if last[a[i]] >= left:
        left = last[a[i]]+1
    last[a[i]] = i
    if i-left+1 == m :
        total += 1
# endfor
print(total)
```

把一群範圍大的數字對應到一個小範圍(0 ~ r)的整數稱為座標壓縮或離散化，這個技巧運用在很多場合。Python 做離散化可以用字典與集合，也可以只用排序與二分搜，以下是不使用集合也不用內建二分搜做離散化的方式。這題做完離散化後，就可以簡單用一個小表格來記錄顏色的位置了。

```
m,n = map(int, input().split())
a = [int(x) for x in input().split()] # int
b = sorted(a)
# remove duplicate
l = [b[0]] + [b[i] for i in range(1,n) if b[i]!=b[i-1]]
nl = len(l)
for i in range(n):
    # binary search a[i], always found
    le = 0; ri = nl-1
    while le <= ri:
        mid = (le+ri) >> 1
        if a[i] == l[mid]:
            a[i] = mid; break
        elif a[i] < l[mid]:
            ri = mid - 1
        else: le = mid + 1
# end
last = [-1]*nl
total = 0 #answer
left = 0
for i in range(n):
    if last[a[i]] >= left:
        left = last[a[i]]+1
    last[a[i]] = i
    if i-left+1 == m :
        total += 1
# endfor
print(total)
```

註：ZJ-e289. 美麗的彩帶(AP201906_4)題目修改了，要找 m 色而非全部顏色，另外後面數字很大，需用字串。

## 14. AP325 Q-7-5. 闖關路線 (AP201910_3)

TCFSH CIRC編號 d094

Q-7-5. 闖關路線 (APCS201910)

某個闖關遊戲上有一隻神奇寶貝與兩個可控制左右移動的按鍵。神奇寶貝被安置在僅可左右移動的滑軌上。滑軌分成 n 個位置，由左到右分別以 0 ~ n - 1 表示。當遊戲開始時，神奇寶貝從位置 0 開始，遊戲的資訊包含 P、L 與 R 三個數字，其中 P 表示所須移至的目標位置，L 與 R 則分別表示每按一次左鍵或右鍵後，會往左或往右移動的格子數。此外，每一個位置 x 都對應一個瞬間移動位置 S(x)；每一次按鍵後，神奇寶貝會先依據按鍵往左或右移動到某個位置 $x$，接著瞬間移動至 $S(x)$。某些點的瞬間移動位置等同原地點，也就是 $S(x) = x$，這些點稱為停留點。開始與目標位置都一定是停留點；此外，每個點的瞬間移動位置都一定是停留點(除非超出界外)，也就是不會發生連續瞬間移動的情形。

遊戲的目標是以最少的按鍵數操作神奇寶貝由開始位置到達目標位置，此外，在移動過程中不可以超過滑軌的範圍[0, $n$ - 1]，否則算闖關失敗；某些點的瞬間移動位置也可能會超出滑軌的範圍，移動到這些點也會導致闖關失敗。

Time limit: 1 秒

輸入格式：輸入有兩行，第一行有 4 個數字，第 1 個為 $n$，第 2 個為目標位置 $P$，第 3 個為 $L$，第 4 個為 $R$，後三個數字皆為小於 $n$ 之正整數，且 $2 \le n \le$ 1e6。第二行有 $n$ 個整數，依序是各點的瞬間移動位置 $S(0)$, $S(1)$, …, $S(n-1)$，這些數字是絕對值不超過 1e8 的整數。

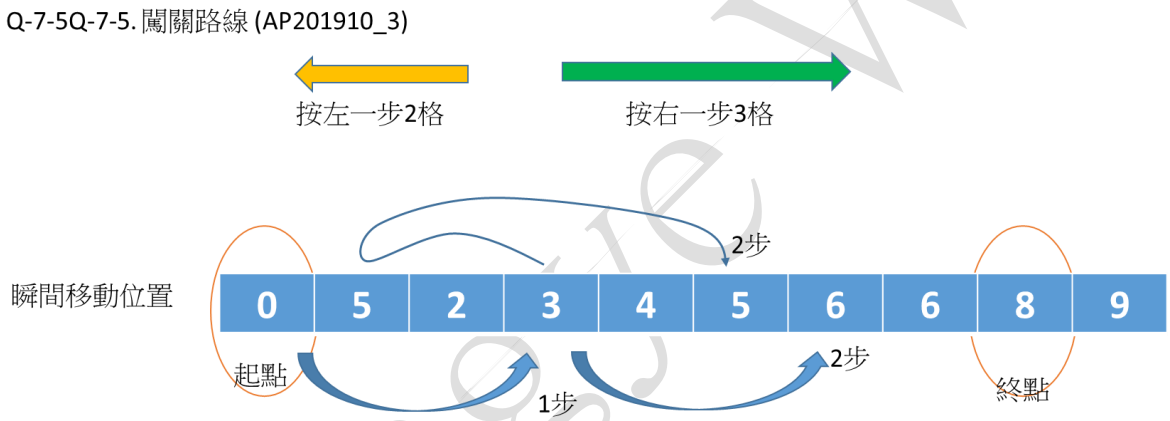
輸出：輸出到達目標位置所需的最少按鍵數，如果無法到達目標位置，則輸出 -1。。

| 範例一輸入：<br><br>5 3 1 2<br>0 3 2 3 5 | 範例一輸出：<br><br>2 |
|---|---|
| 範例二輸入：<br><br>10 8 2 3<br>0 5 2 3 4 5 6 6 8 9 | 範例二輸出：<br><br>3 |

範例一說明：位置區間為[0,4]，目標位置為 3，左鍵往左移動 1 格，右鍵往右移動 2 格。到目標位置最少的按鍵數是 2 次：從起點 0 第一次按下右鍵會到達位置 2，因為 S(2)=2，因此停留在 2，第二次按下左鍵，會往左移一格到達 1，因為 S(1)=3，所以瞬間移動到 3，由於 3 是目標位置，所以就闖關成功了。

範例二說明：位置區間為[0,9]，目標位置為 8，按一次左鍵往左移動 2 格，按一次右鍵往右移動 3 格。到達目標位置的最少按鍵數是 3 次，其經過的路徑是：0，按右鍵→ 3 (停留在 3)，按左鍵→ 1 (瞬間移動到 5)，按右鍵→ 8 (停留在 8，到達)。

(看圖示題目說明)




他顯然是 BFS 的退化在一維數線上的狀況，基本上我們算出一步(一次按鍵)可以到達的點，然後算兩步的點，…，直到走到目標或是無路可走為止，甚至不需要了解 BFS 也可以推想出作法，但是有 BFS 的知識讓我們可以很容易的作出標準作法。

先看一下 BFS 的基本演算法

```
# BFS
# using a list as first-in-first-out queue
que = 0 # starting point
front = 0 # head of que
while front < len(que): # que is not empty
    v = que[front]; front += 1 # pop
    # move one step from v
    # for each next_point
        # d(next_point)= distance(v) + 1
        # que.append(next_point)
# end while
```

在本題中，v 點的下一個拜訪點可能是往左移 L 位置，或者往右移 R 位置，但別忘了還有瞬間移動，在檢查瞬間移動前必須確認沒有出界，瞬間移動後也要檢查是否出界。

```
# Q-7-5, ap201910q3, BFS
n,p,l,r = map(int, input().split())
move = [int(x) for x in input().split()]
d = [n]*n # initial distance
d[0] = 0 # starting point
# using a list as first-in-first-out queue
que = [0] # starting point
front = 0 # head of que
while front < len(que) and d[p]==n: # que is not empty
    v = que[front]; front += 1 # pop
    # if v==p: break
    # go left
    left = v - l
    if left >= 0:
        left = move[left]
        if 0<=left<n and d[left] == n: # otherwise, already visited
            d[left] = d[v]+1
            que.append(left)
    # go right
    right = v + r
    if right < n:
        right = move[right]
        if 0<=right<n and d[right] == n: # otherwise, already visited
            d[right] = d[v]+1
            que.append(right)
# end while
if d[p]< n: print(d[p])
else: print(-1)
```

# 15. AP325 P-6-21.刪除邊界(AP201910_4)

TCFSH CIRC 編號 d082

P-6-21. 刪除邊界 (APCS201910)

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

Time limit: 1 秒

輸入格式：第一行為兩個不超過 25 的正整數 *m* 和 *n*，以下 *m* 列 *n* 行是矩陣內容，順序是由上而下，由左至右，矩陣內容為 0 或 1，同一行數字中間以一個空白間隔。

輸出：最少修改次數。

| 範例一輸入：<br><br>4 5<br>0 1 0 1 1<br>1 1 1 0 1<br>0 0 0 0 0<br>0 0 0 1 0 | 範例一輸出：<br><br>2 |
|---|---|
| 範例二輸入：<br><br>3 5<br>0 0 0 1 0<br>1 0 1 1 1<br>0 0 0 1 0 | 範例二輸出：<br><br>1 |

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

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

**本題有個子題 Q-1-11. 刪除矩形邊界 — 遞迴 (m+n <= 13)** (TCFSH d.009)

給一個矩陣，每次刪除一個邊界，刪除邊界的成本是該邊界線上 0 與 1 比較少的個數，要計算最小的刪除總成本，也就是刪除邊界的最好的順序。

(看圖例子)

遞迴的時間複雜度是 $O((m+n)4^{m+n-2})$，因為每次有 4 個選擇，每選擇一次，m+n 會減少 1。

先來寫遞迴版，之後再改成 top-down DP 版本。

一列或一行的刪除成本其實就是裡面 0 與 1 的個數中比較少的那個。為了清楚起見，我們把一個 list 的刪除成本寫成一個函數。

```
def icost(x):
    return min(x.count(0), x.count(1))

def dp(l,t,r,b):
    if l==r or t==b:
        return 0
    # top row
    mcost = icost(a[t][l:r+1]) + dp(l, t+1, r, b)
    # bottom row
    cost = icost(a[b][l:r+1]) + dp(l, t, r, b-1)
    mcost = min(mcost,cost)
    # left column
    v = [a[i][l] for i in range(t, b+1)]
    cost = icost(v) + dp(l+1, t, r, b)
    mcost = min(mcost,cost)
    # right column
    v = [a[i][r] for i in range(t, b+1)]
    cost = icost(v) + dp(l, t, r-1, b)
    mcost = min(mcost,cost)
    return mcost
#end of dp

# main prog
m, n = map(int, input().split())
a = [[] for x in range(m)]
for i in range(m):
    a[i] = [int(x) for x in input().split()]
if m+n<20:
    print(dp(0,0,n-1,m-1))
```

從 DP 的想法來看，因為刪除的是邊界，需要計算的不外乎所有可能的子矩陣，也就是某一塊矩形，那麼有多少矩形呢？有 $O(mn)$ 個可能的左上角與 $O(mn)$ 個可能的右下角，所以有 $O(m^2n^2)$ 個。以 top-down 的寫法，反正稀哩糊塗的拉個四層迴圈把表格建好，然後照遞迴的方式稍微

改一下就可以了，以下是範例程式，雖然程式不很短，但只是四個邊界每邊都要算一次，其實程式碼很簡單。這裡我們把計算一個 List 的 cost 直接寫在程式內。整體的複雜度是 O((m+n)$m^2n^2$)，事實上運用前綴和的預處理之後，可以在 O(1)求得一個 row 或 column 的某一段總和，而將複雜度改善到 O($m^2n^2$)，因為本題 m 與 n 都很小，這個就留給有興趣的人自己試試。

```
def cost(l, t, r, b):
    if l==r or t==b:
        return 0
    if tab[l][t][r][b] >= 0:
        return tab[l][t][r][b]
    # top row
    n1 = sum(a[t][l:r+1])
    mcost = min(n1, r+1-l-n1) + cost(l, t+1, r, b)
    # bottom row
    n1 = sum(a[b][l:r+1])
    c = min(n1, r+1-l-n1) + cost(l, t, r, b-1)
    mcost = min(mcost, c)
    # left column
    n1 = sum(a[i][l] for i in range(t, b+1))
    c = min(n1, b+1-t-n1) + cost(l+1, t, r, b)
    mcost = min(mcost, c)
    # right column
    n1 = sum(a[i][r] for i in range(t, b+1))
    c = min(n1, b+1-t-n1) + cost(l, t, r-1, b)
    mcost = min(mcost, c)
    tab[l][t][r][b] = mcost
    return mcost
#end of cost
# main prog
m, n = map(int, input().split())
a = []
for i in range(m):
    a.append([int(x) for x in input().split()])
tab = [[[[-1]*25 for i in range(25)] for j in range(25)] \
     for k in range(25)]
print(cost(0,0,n-1,m-1))
```

## 16. AP325 P-3-6 砍樹(AP202001_3)

(TCFSH CIRC 編號 d030)

P-3-6. 砍樹 (APCS202001)

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

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

Time limit: 1 秒

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

輸出：第一行輸出能被砍除之樹木數量，第二行輸出能被砍除之樹木中最高的高度。

範例輸入：

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

範例結果：

```
4
30
```

題目已經說：無論砍樹的順序為何，最後能砍除的樹木是相同的。所以只要不斷的找砍除條件的樹木，將它砍倒後移除，直到沒有樹木可以砍為止，就可以得到正確的答案，問題只是效率好不好。

寫程式最重要的是知道每次要做甚麼事情，資料中存放的是甚麼，有何特性。樹的資料包含中心位置與高度，如果我們用陣列來存這些樹，那麼立刻碰到一個問題：當樹被砍掉了之後怎麼辦？我們當然必須維護好陣列中是沒有被砍掉的樹，否則判斷就會出問題。

第一個直接想法是把砍倒的樹的資料從陣列中移除。這樣做有兩大問題：

- 如果使用

  ```
  for i in range(len(a)):
      if somecase: a.pop(i)
  ```

  這樣做是錯誤的，因為 pop(i)時會變動 a 在位置 i 以後的內容，下一個 a[i+1]並非原來的 a[i+1]而是 a[i+2]。

- 如果很聰明的從後往前走

  ```
  for i in range(len(a)-1, -1, -1):
      if somecase: a.pop(i)
  ```

  這樣做雖然可以，但是時間複雜度是糟糕的，因為陣列最經不起插入與刪除的折騰，a.pop(i)所花的時間是 O(len(a)-i)，當 a 的長度是很大的時候，這樣做耗時過多。

下面是這樣的寫法，答案會算正確但會超時(TLE)。這裡我們加了一個小技巧，在兩個邊界加上兩個端點座標，這是常用的技巧，目的是省卻檢查邊界條件的麻煩。

```
# n^2 method, remove in list
n, l = map(int, input().split())
c = [0] + [int(x) for x in input().split()]+[l]
h = [0]+[int(x) for x in input().split()]+[0]
total = high =0 #answer
while True: # until no change
    nochange = True
    for i in range(len(c)-2,0,-1) :
        if c[i]-h[i] >= c[i-1] or c[i]+h[i] <= c[i+1] :
        # removable
            total += 1; high = max(high, h[i])
            c.pop(i)
            h.pop(i)
            nochange = False
    if nochange: break
#end for
print(total); print(high)
```

另外一個可能的想法是將砍掉的樹的高度與位置改掉，但這樣一來每次要決定是否可以砍的時候，就必須往前往後搜尋到下一棵存活的樹或者是找到已經可以砍的條件，類似的想法可以用一個 List 存放存活的樹的編號，下面的範例程式是這樣的想法寫出來的。

每一回合把存活的樹全部檢查一遍，可以砍的就砍了，不能砍的放進一個暫存的 temp 中，一輪砍完再將兩者交換。

```
# n^2 method
n, l = map(int, input().split())
c = [0] + [int(x) for x in input().split()]+[l]
h = [0]+[int(x) for x in input().split()]+[0]
alive = [i for i in range(n+2)] # index of alive tree
total = high =0 #answer
while True: # until no change
    temp = [0]
    for i in range(1,len(alive)-1) :
        if c[alive[i]]-h[alive[i]] >= c[alive[i-1]] \
           or c[alive[i]]+h[alive[i]] <= c[alive[i+1]] :
        # removable
            total += 1; high = max(high, h[alive[i]])
        else:
            temp.append(alive[i])
    # end if
    temp.append(n+1)
    if len(temp) == len(alive): break
    temp, alive = alive, temp
#end for
print(total); print(high)
```

複雜度如何？很不幸，複雜度還是 O(n²)，因為有可能需要 O(n) 回合才會結束。

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

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

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

index，當然也可以寫成在堆疊中存放樹的位置與高度。另外在端點的樹高度給無限大，以免他在堆疊中被 pop 掉。

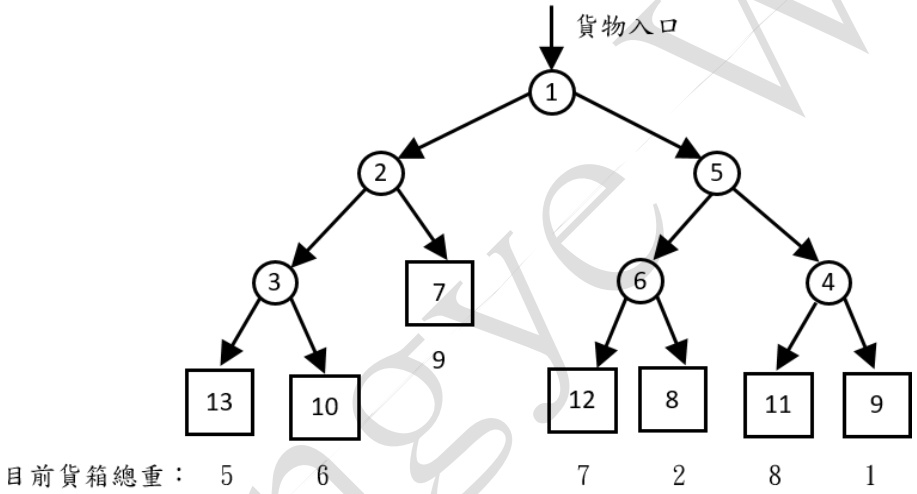
```
n, l = map(int, input().split())
c = [0] + [int(x) for x in input().split()]+[l]
oo = 1000000001
h = [oo]+[int(x) for x in input().split()]+[oo]
stack = [0] # index of alive tree
total = high =0 #answer
for i in range(1, n+1):
    if c[i]-h[i]>=c[stack[-1]] or c[i]+h[i]<=c[i+1] :
        # removable
        total += 1; high = max(high, h[i])
        while c[stack[-1]]+h[stack[-1]] <= c[i+1] :
            total +=1; high = max(high, h[stack[-1]])
            stack.pop()
        #end while
    else:
        stack.append(i)
    # end if
#end for
print(total); print(high)
```

# 17. AP325 P-8-5.自動分裝(AP202001_4)

(TCFSH CIRC 編號 d105)

P-8-5. 自動分裝 (APCS202002)

某工程公司設計了一個自動分裝的系統。貨物會一個一個的被送進此系統，經過一些切換器的轉送後，會被輸送到 n 個貨箱。系統有 n－1 個切換器與 n 個貨箱。每個切換器有一個入口以及左右兩個出口，切換器的出口會接到其他切換器或是連接到貨箱，貨箱則只有入口。下圖是一個 n=7 的例子，圓代表切換器，方形代表貨箱，請注意編號 1 ~ n－1 的裝置是切換器，貨箱編號是 n ~ 2n－1，且貨物入口一定是 1 號切換器的入口。




每一個切換器會分別記錄左右兩個出口所通往貨箱的總重量，當貨物進入此切換器時，切換器會將貨物轉送到「貨箱總重量比較輕的那個出口」，如果兩邊一樣重，則送往左邊。以上圖的例子來說，假設每一個貨箱目前的重量如各矩形下方的標示，下一個到達的貨物的運送過程如下：一號切換器左邊出口是連到 {13, 10, 7} 三個貨箱，目前總重 5+6+9=20；右邊出口連到的是 {12, 8, 11, 9} 四個貨箱，目前總重 7+2+8+1=18，因此貨物會由右邊出口送到 5 號切換器。5 號切換器的左邊與右邊的貨箱總重是一樣的（7+2=8+1），因此貨物由左出口送至 6 號切換器。最後，6 號切換器將貨物送到 8 號貨箱（7>2）。貨物進入貨箱後就存放在該貨箱，該貨物的重量就會加入該貨箱的總重，因此可能影響下一個貨物的運送路徑。

輸入此系統的連接架構與貨箱目前的重量，以及接下來依序進入的 *m* 個貨物的重量，請計算這 *m* 個貨物分別會被送到哪一個貨箱。

Time limit: 1 秒

輸入格式：第一行為兩個正整數 $n$ 和 $m$，其中 $n \leq 1e5$ 且 $m \leq 100$。第二行有 $n$ 個非負整數，依序是編號為 $n \sim 2n-1$ 各個貨箱初始的重量。第三行是 $m$ 個正整數，代表接下來依序進入的貨物重量。全部貨箱初始的重量與貨物重量總和不會超過 $10^9$。第四行開始有 $n-1$ 行，這些是系統架構的資訊：每一行有三個整數 $p$、$s$ 與 $t$，代表裝置 $p$ 的左右出口分別接到裝置 $s$ 與 $t$，其中 $p$ 一定是一個切換器的編號($1 \leq p < n$)。同一行數字之間以空白間隔。

輸出：輸出一行有 m 個整數，依序代表接下來進入系統的 m 個貨物所進入到的貨箱編號，數字之間以一個空白間隔。

| 範例一：輸入 | 範例一：正確輸出 |
|---|---|
| 4 5<br>0 0 0 0<br>5 3 4 2 1<br>1 2 3<br>2 4 5<br>3 6 7 | 4 6 7 5 5 |
| 範例二：輸入 | 範例二：正確輸出 |
| 7 2<br>9 2 1 6 8 7 5<br>2 3<br>1 2 5<br>2 3 7<br>3 13 10<br>4 11 9<br>6 12 8<br>5 6 4 | 8 7 |

範例二的架構即是題目中的圖。

(ZJ f163. 貨物分配, only 90% on ZJ，memory error, data size 比考試時大了很多)

本題所描述的系統其實是一個所謂的二元樹(binary tree)的結構，每一個切換器是一個中間節點(internal node)，每一個貨箱是一個葉節點(leaf node)。題目的要求是模擬每一個貨物進入後的流程，要知道貨物在每個切換器會流向左方或右方，我們需要知道左邊與右邊貨箱**總重量**。假設我們知道初始狀態時，每個切換器左右的總重

量之後，我們就可以依照題意模擬貨物流向，而且根據流向來更新左右總重，所以在下一個貨物進入時，就可以繼續模擬的動作。

所以解這個題目需要會做兩件事：1.如何儲存此樹狀結構；2.會計算初始每個中間節點左右的總重量。

對於會處理樹狀圖的人來說，此題很簡單，對於不了解樹狀圖的人來說，雖然也可以依照題意想出解法，但陌生的題目類型可能會有一些難度。

對每個節點，我們紀錄 3 個結構資料: parent, left child, right child。如何計算每個節點其下的貨物總重呢？對每一個節點 i，其下的重量紀錄在 w[i]中，對於每一個貨箱(葉節點)，因為他屬於他的每一個祖先(中間節點)之下的貨箱，所以我們需要將它的重量加到他的所有祖先。一個直覺的想法是：

> 對每個貨箱 v，一路沿著 parent 的連結往上走，直到 root，這樣可以經過他的所有祖先，將 w[v]加入到沿路經過的祖先。

以下是這樣寫出來的程式，很簡單，但是跑得不夠快。

```
# non-recursive
n, m = map(int, input().split())
w = [0 for x in range(n)] + [int(x) for x in input().split()]
seq = [int(x) for x in input().split()] # incoming aequence
lc = [-1 for x in range(2*n)]  #left child
rc = [-1 for x in range(2*n)]  # right child
p = [-1 for x in range(2*n)]  # parent
for i in range(n-1):  # input tree
    s, l, r = map(int, input().split())
    lc[s] = l; rc[s] = r
    p[l] = p[r] = s
# find weights, O(n^2)
for v in range(n,2*n):
    u = p[v]
    while u >= 0:
        w[u] += w[v]
        u = p[u]
#start simulation
out = []
for i in range(m):
    v = 1
    while v < n:
        if w[lc[v]] <= w[rc[v]]:
            v = lc[v]
        else: v = rc[v]
        w[v] += seq[i]
    out.append(v)
#end for
print(*out)
```

這個方法的時間複雜度是每個葉節點的祖先人數總和，在 worst case 時，這個二元樹的每一個中間節點形成一根鏈(一根棍子)，幾乎所有的中間節點都有一個孩子是 leaf，這個樹的高度會達到 n/2，所以每個葉子的祖先總數會達到 $O(n^2)$。

一如很多 tree 的問題，可以用動態規劃(DP)的觀點來思考。對於每個中間節點 v，我們很容易發現：

w[v] = w[left_child[v]] + w[right_child[v]]。

要實現這個遞迴式，我們可以用 top-down 的遞迴或 bottom-up 兩種方式。這裡採用 Bottom-up 的方式。

所謂 post-order 的順序是將所有樹節點排一個順序，滿足以下條件：每個點都出現在他的所有孩子之後。只要沿著一個 post-order 的順序，我們將每個點的重量加到他的 parent 上，那麼，走到每個 parent 時，該 parent 以下的所有的 w[]值都是計算好的。也就是說可以在 O(n)的時間就計算出所有 n 個點的 w[]值。

得到 post-order sequence 的方法，前面很多樹的題目都講過了，簡單再說明一次，如果不清楚就回去看或看 AP325 的講義(8.2.4 第 279 頁)。

計算完畢 w[]後，第二部分是模擬進來貨物的流向，在結構存好之後，這個只要根據左右孩子的 w[]來判斷並且更新就好了。

```
# non-recursive
n, m = map(int, input().split())
w = [0 for x in range(n)] + [int(x) for x in input().split()]
seq = [int(x) for x in input().split()] # incoming aequence
lc = [-1 for x in range(2*n)]  #left child
rc = [-1 for x in range(2*n)]  # right child
p = [-1 for x in range(2*n)]  # parent
for i in range(n-1):  # input tree
    s, l, r = map(int, input().split())
    lc[s] = l; rc[s] = r
    p[l] = p[r] = s
# find weights
deg = [2 for x in range(n)]  # num of children
que = [x for x in range(n, 2*n)] # FIFO queue
qi = 0 # head of queue
while qi < len(que): # while queue is not empty
    v = que[qi]; qi += 1  # pop from que
    if v==1: break  # root
    u = p[v]
    w[u] += w[v]
    deg[u] -= 1
    if deg[u] == 0:
        que.append(u)
#start simulation
out = []
for i in range(m):
```

```
    v = 1
    while v < n:
        if w[lc[v]] <= w[rc[v]]:
            v = lc[v]
        else: v = rc[v]
        w[v] += seq[i]
    out.append(v)
#end for
print(*out)
```

# 18. f581. 圓環出口(AP202007_3)

(AP325 P-2-15, TCFSH CIRC 編號 d024)

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。

Time limit: 1 秒

輸入格式：第一行有兩個正整數 n 與 m，第二行有 n 個正整數，依序是各房間的點數 p(i)，第三行有 m 個正整數依序是各鑰匙需要的點數 Q(i)，。同一行連續二數字間以空白隔開。n 不超過 2e5，m 不超過 2e4，p(i)總和不超過 1e9，Q(i)不超過所有 p(i)總和。

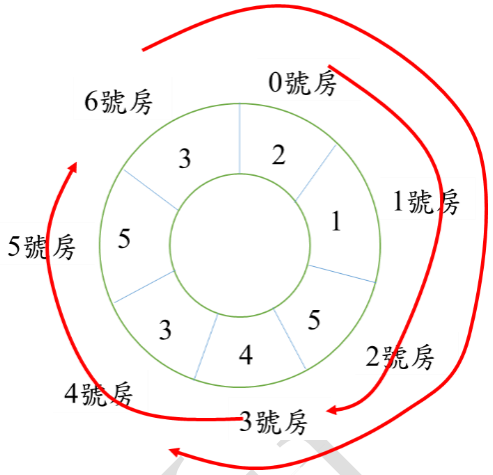
輸出格式：輸出拿到最後一把鑰匙時所在的房間編號。

範例輸入：

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

範例輸出：

```
4
```




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

把一個長度為 n 的一維整數陣列看成圓環，每次從當前位置開始往前，累加陣列內的數字直到達到或超過某個要求的數字 $Q_i$，然後將累加數字歸零。重複這個步驟 m 次，計算出最後一次到達的位置。

這個題目類似操作題，依照題意來模擬，但是如果直接做，時間效率會有問題，所以必須找到一個方法很快地計算出每次停留的位置。

只看每一次的需求，這是一種在陣列中搜尋的問題，想到有效率的搜尋，腦海中浮現的自然是二分搜，但是二分搜必須是在單調遞增序列上找第一個超過某數的位置，我們現在要的是以目前為起點第一個超過某數的區間總和。假設我們一開始把所有陣列的數字轉換成前綴和(Prefix-sum)，也就是把陣列的第 i 個元素(假設是 p[i])轉換成 $\sum_{j=0}^{i} p[j]$，我們知道要算[s,t]的區間和其實就等於 p[t] - p[s-1]。反過來說，假設目前的位置是 s，我們要找到第一個位置 t 使得[s,t]區間和至少 Q，那就是等於找第一個不小於 Q + p[s-1]的前綴和 p[t]。而且因為正數的前綴和必然是嚴格遞增的，所以我們可以用二分搜來快速的搜尋到所要的前綴和。

題目還有一些細節要處理。第一個小問題是若目前位置是 0，則前述的 p[s-1]要設為 0；第二個要處理的問題是陣列是圓環的問題，所以如果找到陣列的尾端，必須從頭開始。處理這樣的狀況有兩個常用的做法：(A)搜尋時先檢查一下是否會超過尾端，如果會，扣除剩下的總和，再從陣列的起始位置搜尋；(B)把陣列重複一次，也就是把長度延長一倍，每次搜完之後再將位置取除以 n 的餘數。

```
n, m = map(int, input().split())
a = [int(x) for x in input().split()]
q = [int(x) for x in input().split()]
for i in range(1,n):
    a[i] += a[i-1]
room = 0
```

```
total = a[n-1]
#print(n,m, len(q))
for qi in q:
    if room>0: qi += a[room-1]
    if qi>total:
        room = 0
        qi -= total
    # binary search the first >=, [left, right]
    # always found
    left = room; right = n-1
    while left < right:
        mid = (left+right)//2
        if qi <= a[mid]:
            right = mid
        else:
            left = mid+1
    room = (left+1)%n
#end for
print(room)
```

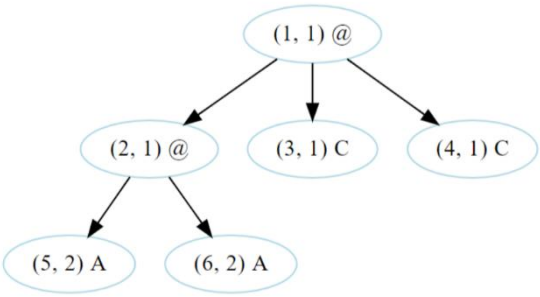
二分搜可以用 bisect.bisect_left 來做，速度快很多而且方便。

```
import bisect
n, m = map(int, input().split())
a = [int(x) for x in input().split()]
q = [int(x) for x in input().split()]
for i in range(1,n):
    a[i] += a[i-1]
room = 0
total = a[n-1]
#print(n,m, len(q))
for qi in q:
    if room>0: qi += a[room-1]
    if qi>total:
        room = 0
        qi -= total
    # binary search the first >=qi
    room = bisect.bisect_left(a,qi)
    room = (room+1)%n
    #print(q[i], left, a[left], right, a[right], room)
#end for
print(room)
```

# 19. f582. 病毒演化(AP202007_4)

(root definition： i==j)(need bottom-up)

給一棵病毒演化樹，每一點的
(編號，親代，RNA 字串)，
RNA 字串由 AUCG@組成，其中@表示未知。要決定每一點的@代入 AUCG 之一，使得每一點與親代的變異量總和最小。




兩字串的變異量是 Hamming distance，也就是每個字串為置各自比對，看看不同的有幾個位置，例如 AUCGG 與 ACGGU 的變異量是 3。

| | | | | |
|---|---|---|---|---|
| A | U | C | G | G |
| A | C | G | G | U |

這是個樹狀結構的問題，另外一個很重要的事情是：雖然每個點是個字串，但是字串之間的比對只比相同位置，也就是每個位置是獨立的。所有字串的第 1 個字元，要算一個最小變異量；所有字串的第 2 個字元，要算一個最小變異量；對每一個字串位置 i 都要算一個最小變異量，最後的答案是把他們加起來。所以，我們只要會解字串長度 m=1 的狀況，最後套一個迴圈就好了。

如果沒有@字元，所有的變異都是確定的，只要把每個點與它的 parent 比較是否相等就好了，所以問題在如何決定@變異量。如果葉節點是@，我們可以將他設為與它 parent 相同的字元，這顯然不會有變異量，也就是說是@的葉子可以直接忽略。

為了計算方便，我們先將 AUCG 轉換為數字 0~3，以 DP(動態規劃)的觀點，對於每個節點 v，我們算四個成本 cost[v][i], i = 0 ~ 3，分別代表 parent 的字元是 i 的時候，v 點以下子樹的總成本，然後去建構出遞迴式。

第 1 子題沒有@，所以根本連樹狀結構也不必管，每個節點字串跟 parent 的字串做比對就可以了。子題 1 與 2 在節點編號上有特殊的方式，parent 的編號必然小於子節點的編號，這是用來讓找 bottom-up 順序時可以很簡單的由大到小。對於第 3 子題，我們必須自己找出 DP 的順序。

動態規劃最重要的是找出遞迴式，為了計算方便，我們先將 AUCG 轉換為數字 0~3，字元@就轉換為 4，節點 v 的字元轉換後就放在 rna[v]。我們把要計算的最小變異量區分成已經確定的變異量以及尚未確定的變異量，對於已經確定的，可以先加總到總和之中。對於一個節點 v，i = 0~3，我們定義

cost[v][i]：若rna[parent(v)] = i 時，v點以下所有節點(含v點)尚未確定的最小總和變異量。

先來看看 rna[v] < 4，也就是字元不是@的狀態。此時 v 點以下的所有變異量都可以決定了，根據定義，這些變異量就是$\sum_{u\in C(v)} cost[u][rna[v]]$，其中 C(v)是 v 的所有孩子。而尚未確定的變異量就是 v 與 parent(v)是否相同，如果相同則變異量為 0；否則為 1，因此，

cost[v][i] = (rna[v]==i) ? 0 : 1;

接下來看 rna[v]=4，也就是字元是@的情形。

對 i = 0~3，令$\text{sum}[i] = \sum_{u\in C(v)} cost[u][i]$，而 mcost = min(sum[i])是四種成本中的最小值。那麼我們知道，不管將來 parent 是什麼，mcost 的量是少不掉的，所以我們可以把 mcost 當成確定變異量先去加總起來，剩下的部份呢？如果 parent 就是讓我達到 mcost 的字元，那麼最小變異量就是 mcost。否則有兩種情形

- 我設成達到 mcost 的字元，parent 與我不同，則變異量是 mcost+1；
- 我設成與 parent 相同，則變異量至少是 mcost+1。

所以結論是：parent 若與我達成 mcost 的字元相同，則最小變異量就是 mcost，否則是 mcost+1。

用舉例來說比較容易懂，假設 sum[0]=sum[2]=2, sum[1]=3, sum[3]=5，現在的 mcost=2，i=0 或 2 時是最小，將來如果 parent 的 rna 是 0 或 2 時，我們就讓 v 點與 parent 是一樣的，所以變異量就是 mcost，如果 parent 的 rna 不是 0 或 2(是 1 或 3)，我們把 v 點設成 0 或 2，則變異量除了 mcost 還要多增加 1，這個 1 是 v 與 parent 不同造成的；如果我們讓 v 設成與 parent 相同，那 v 與 parent 沒有變異，但 v 點以下的變異量會大於 mcost。

所以結論是 mcost 先拿出來加總之後，

cost[v][i] = (sum[i] == mcost) ? 0 : 1;

這個遞迴式的推導過程有些複雜，但結論是簡單的。

在樹狀圖中以非遞迴的方式找一個 bottom-up 順序是一個經典教科書中的程序，我們需要的是找出一個節點的順序，這個順序只要保證子節點一定在 parent 之前就可以。以下我們介紹一個非遞迴的方法來得到一棵樹的 Bottom-up sequence，這個方法也用於找 DAG 的 topological sort。在這個系列的題解中，我們以經碰到好幾次了。

此法的規則就是葉節點可離開(leaves can leave)。我們準備一個先進先出的佇列(queue)，用來存放已經可以輸出但尚未輸出的節點。初始時，沒有孩子的節點就是所有的葉節點，所以初始時將葉節點放入 queue 中。然後，每一回合從 queue 中拿出一個點 v，將 v 輸出；然後模擬把他從樹上移除，如果他的 parent 只剩 v 這個孩子，那麼，移除他之後，他的 parent 就是可移除的葉節點，所以就可以將他的 parent 放入 queue 中。重複執行模擬移除的動作，直到 queue 中沒有節點為止就可以得到整棵樹的 bottom-up sequence。

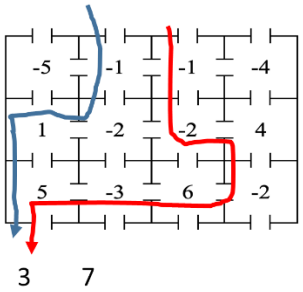
以下是範例程式。其中將字元 AUCG@轉換成 0 ~ 4 可以用 dict 做，也可以用一個字串然後找位置 index，因為本題只有 5 種字元，轉換時間上不是問題。

```
n, m = map(int, input().split())
deg = [0]*(n+1)
par = [0]*(n+1)
rna = [[0]*m for i in range(n+1)]
#sym = {'A':0, 'U':1, 'C':2, 'G':3, '@':4}
char = 'AUCG@'
for i in range(n):
    x,y,s = input().split()
    x=int(x); y=int(y)
    if x != y:
        par[x] = y
        deg[y] += 1
    rna[x] = [ char.index(c) for c in s]
    # for j in range(len(s)):
        #rna[x][j]=sym[s[j]]
# find a bottom-up sequence
seq = [i for i in range(1,n+1) if deg[i]==0]
head = 0
while len(seq) < n:
    v = seq[head]; head += 1
    p = par[v]
    if p == 0: break
    deg[p] -= 1
    if deg[p] == 0: seq.append(p)
# end while
if len(seq)!=n or par[seq[n-1]]!=0 :
    print('wrong tree'); print(seq)
# starting computing
total = 0
for i in range(m): # for each position
    cost = [[0]*4 for i in range(n+1)]
    for v in seq:
        ch = rna[v][i]
        if ch < 4:
            total += cost[v][ch]
            for j in range(4):
                if j!=ch:
                    cost[par[v]][j] += 1
        else: # case @
            mcost = min(cost[v])
```

```
            total += mcost
            for j in range(4):
                if cost[v][j] != mcost:
                    cost[par[v]][j] += 1
        #endif
    #end for-v
#end for-position
print(total)
```

# 20. f314. 勇者修煉(AP202010_3)

輸入為 m x n 大小的陣列，每一格是一個介於 -100 與 100 之間的整數，表示經過這格可以累積的經驗值。你可以從最上面一排任何一個位置開始，在最下面一排任何一個位置結束。過程中每一步可以選擇往左、往右或往下走，但不能走回已經經過的位置。




雖然給的是二維矩陣，但因為只能往下走，其實可以看成是一個由上到下的過關遊戲，每一關我們要決定往左或往右走到某個位置再進入下一關。目標是希望累積的分數越大越好。

也就是說，所謂的一個解就是決定每一關(層)的進入位置，或者說離開位置。

因為不可以往上走，所以我們只要一層層往下算，重點是相鄰兩層的關係以及每一層如何計算。

假設目前這一層的格子是存在 a[]陣列，每一層的得分是從進入點到離開點(包含兩端)的區間和，所以這一層的得分與進入點以及離開點有關。由於這一層的離開點就是下一層的進入點，以動態規劃的思考方式，我們可以由上往下計算(記錄)每一層每一個離開位置的最大可能得分。因為在一層裡面可能是往右或往左，所以我們要計算兩種能走法中較好的。

```
m,n = map(int, input().split())
a = [[0]*n]
for i in range(m):
    a.append([int(x) for x in input().split()])
# 初始化一個陣列 dp[M][N]，以 dp[i][j]紀錄第 i 層從位置 j 離開的最大可能得分
dp = [[0]*n] + [[0]*n for i in range(m)]
for i in range(1,m+1): # 第 i 層的結果
    for j in range(n): # 走到 j 的最大得分
        dp[i][j] = dp[i-1][j] + a[i][j]  # 上方
        for k in range(j): # 從左邊 k 走到 j 的最大得分
```

```
            t = dp[i-1][k] + sum(a[i][k:j+1])
            if t>dp[i][j]: dp[i][j] = t
        for k in range(j+1,n): # 從右邊 k 走到 j 的最大得分
            t = dp[i-1][k] + sum(a[i][j:k+1])
            if t>dp[i][j]: dp[i][j] = t
    #end for j
#end for i
print(max(dp[m]))
```

這樣寫的時間複雜度是 $O(mn^2)$，因為在每一層中我們計算了從每一個位置 k 到每一個位置 j 的狀態，所以一層要花 $O(n^2)$ 的時間。

這一題提升效率的解法其實很簡單，不需要什麼高深的技巧。假設我們目前在第 i 層，要計算從左邊走到 j 的最佳 left[j]而且 j>0，因為只能從左方或上方走到 j，從上方走過來的得分是 dp[i-1][j]，而左方走過來的得分就是剛算好的 left[j-1]，因此：

left[j]=max(left[j-1],dp[i-1][j])+a[j]；

初始邊界 left[0]=dp[i-1][0]+a[0]。相同地，從右往左算 right[j]時，

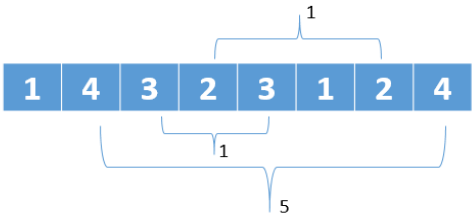
right[j]=max(right[j+1],dp[i-1][j])+a[j]。

我們每一層從 dp[i-1][]算出 left[]與 right[]，然後算出 dp[i][j]=max(left[j],right[j])。而上一層的資料與計算結果此後都不會再用到，因此這一題根本就不需要使用二維陣列。以下是第二種方法，時間複雜度是 O(mn)。要注意的是，因為 left[]與 right[]必須在兩個迴圈中分別計算，因為計算的順序不同。

```
m,n = map(int, input().split())
dp = [0]*n
left = [0]*n
right = [0]*n
for i in range(m):
    a = [int(x) for x in input().split()]
    left[0] = dp[0] + a[0]
    for j in range(1,n): # from left or top to j
        left[j] = max(left[j-1], dp[j]) + a[j]
    right[n-1] = dp[n-1] + a[n-1]
    for j in range(n-2,-1,-1): # from right or top
        right[j] = max(right[j+1], dp[j]) + a[j]
    for j in range(n):
        dp[j] = max(left[j], right[j])
# end for
print(max(dp))
```

## 21. f315. 低地距離(AP202010_4)

輸入一個長度為 2n 的序列，其中 1~n 每個數字恰好出現兩次，對於每一對相同的數字，要計算兩者之間有多少小於它們的數字。




最簡單的想法對每一對相同的數字 x，從前面的 x 往後面掃描過去，數數看在碰到後面的 x 之前有多少小於 x 的數字。但是這個方法顯然耗時太久，對於每一種數字要掃描一次，總共是 $O(n^2)$ 的時間，在 n 達到 10 萬的情形下，無法在一兩秒跑完。題目中有以下提示。

根據提示，本題有兩種方法，第一種是依照數字大小分治；第二種算出每個位置前有多少小於它的數。以下將先介紹大小分治的方法。

分治(Divide and Conquer)就是把要處理的資料分割成兩半，個別遞迴求解，然後再合併，問題在於如何分割與設計合併的步驟。以本題來說，我們要計算的是每一對數字之間小於他們的數字個數(低地距離)，分割的方式採取依數字大小分成兩個子序列(子序列的意思是一個序列中刪除其中一部份，原序列的先後順序不可改變)。假設拆分出了的序列是 small[]與 large[]，而我們也求出了兩個序列個別的答案，那麼與我們所要的答案差別在哪裡呢？對於 small[]中的數字，有多少 large[]的成員在其中並無影響，但是對於 large[]中的數字，在 large[]中求到的低地距離還必須加上有多少 small[]介於其間才是真正的答案。

例如題目中的範例[1,4,3,2,3,1,2,4]，拆成兩序列是
small = [1,2,1,2]，
large = [4,3,3,4]，
假設 d[i]是數字 i 的低地距離，在兩子序列中求到的答案是 d[1]=0, d[2]=1, d[3]=0, d[4]=2。對於 1 與 2 而言，答案不會再改變，因為 3 與 4 插入其間並不會改變答案。但是對於 3 與 4 而言，我們還要計算有多少 1 與 2 在每一對大數字之間。因此, d[3]=0+1，因為有一個 2 在一對 3 之間；d[4]=2+3，因為有 3 個 1 或 2 在一對 4 之間。

剩下的問題是如何計算小數字介於大數對的總數。對於一個小數字 s 而言，我們需要知道的就是 s 介於多少大數對之間。若 s 在大數對 b 之間，則 s 的左右方各有一個 b，所以我們可以這樣計算：當我們由左到右掃描序列時，時時記錄目前的位置在多少大數對之間，也就是左方恰有一個的大數對。以下的程式中，我們以 Python 的集合 set() 來做，當一個大數 x 出現時，我們把他加入 between 中，當第二個 x 出現時，我們把他從 between 中移除。當看到一個小數字時，len(between) 就是目前在幾對大數字之間，這些 set 的運算都可以視為是 O(1)。結合遞迴的架構就可以得到完整的分治程式。

```
# divide and conquer, arr of element in [low,up]
def sol(arr, low, up):
    if len(arr) <= 2: return 0 # terminal case
    mid = (low+up)//2  #median
    d=0
    small = [] #[x for x in arr if x <= mid]
    large = [] #[x for x in arr if x >= mid]
    between = set() # large which we are between
    for e in arr:
        if e <= mid:
            # number of large pairs which e is between
            d += len(between)
            small.append(e)
        else:
            large.append(e)
            if e in between:
                between.remove(e)
            else:
                between.add(e)
    # end for
    d += sol(small,low,mid)
    d += sol(large,mid+1,up)
    return d
#end sol()
#main
n = int(input())
a = [int(x) for x in input().split()]
d = sol(a,1,n)
print(d)
```

如果不會用 set 怎麼辦？學呀 XD。其實這個部分並不難，有很多的方法都可以做，例如以下的程式中，用一個 list 來表示目前是否位於某數之間。也可以用一個 global 的表格來做，要注意的是在一次遞迴中，要把時間壓在 O(len(arr))，所以不可在每次遞迴中都處理一個大小為 n 的 list。

```
# arr of element in [low,up]
def sol(arr, low, up):
    if len(arr) <= 2: return 0 #// terminal case
    mid = (low+up)//2  #median
    d=0
```

```
    small = [] #[x for x in arr if x <= mid]
    large = [] #[x for x in arr if x >= mid]
    between = [False]*(up-mid) # large which we are between
    num = 0 # num of true in between
    for e in arr:
        if e <= mid:
            d += num
            small.append(e)
        else:
            large.append(e)
            e2 = e-mid-1
            if between[e2]:
                between[e2] = False; num -= 1
            else:
                between[e2] = True; num += 1
    # end for
    d += sol(small,low,mid)
    d += sol(large,mid+1,up)
    return d
#end sol()
#main
n = int(input())
a = [int(x) for x in input().split()]
d = sol(a,1,n)
print(d)
```

接下來我們再看第二種思考方式：對於每一個位置 i，計算出在 i 之前有多少小於 a[i] 的元素。令 f(i) 是在 i 之前小於 a[i] 的數量。對於介於 1~n 的每一個數字 x，我們知道 x 恰好出現兩次，假設出現在 i1 與 i2，a[i1] = a[i2] = x，那麼 f(i2)-f(i1) 就是 x 的低地距離，而只要將所有的 x 的低地距離加總就是答案。

事實上計算有多少 j<i 且 a[j]<a[i] 跟之前另外一題反序數量 (inversion) 是相同類型的問題。直接的做法是利用某些資料結構，以下我們介紹與解 inversion 類似的分治演算法。

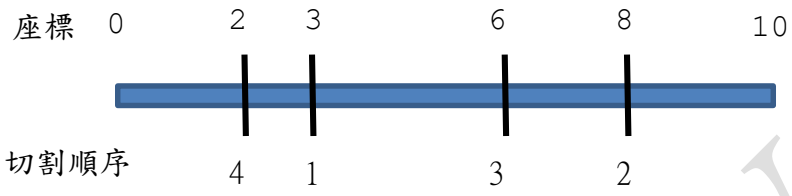
我們打算將序列切成左右兩半分治，合併的時候只要對右邊的每一個 x 補上左邊有多少小於他的數字即可，這裡我們有點小困難，就是每個數字 x 出現兩次，當我們在遞迴過程中碰到 x 時，不知道他是第一個還是第二個 x，對於第一個我們要減去 f() 值，對於第二個則要加上 f() 值。解決的方法有多種，以下我們把原來的第一個 x 轉換成 2x 而第二個 x 轉換成 2x-1，這樣所有出現的數字變成 1~(2n-1) 各出現一次，然後只要把奇數的 f() 值減去偶數的 f() 值就是答案了。

遞迴函式的部份與 merge sort 有 87%相像，均勻切成左右兩端各自遞迴解之後，我們合併兩個 sorted list，合併過程對右邊的 x 記錄左邊小於他的數量，然後 x 如果是奇數就加入答案，偶數就從答案中扣除。

```
# return sorted list
def sol(ar):
    global total
    if len(ar) < 2: return # terminal case
    mid = len(ar)//2
    left = ar[:mid]
    right = ar[mid:]
    sol(left) # recursively solve left and right
    sol(right)
    # merge left and right into ar
    i = 0; li = 0 #index of ar and left
    for x in right:
        while li<len(left) and left[li]<=x:
            ar[i] = left[li]
            i += 1; li += 1
        if x%2: total += li # odd x
        else: total -= li
        ar[i] = x
        i += 1
    ar[i:] = left[li:] #remaining left
#end sol()
#main
n = int(input())
a = [int(x) for x in input().split()]
# change i to 2i and 2i-1
t = [0]*(n+1)
for i in range(len(a)):
    y= a[i]
    a[i] = y*2 - t[y]
    t[y] = 1
total = 0
sol(a)
print(total)
```

## 22. f607. 切割費用(AP202101_3)

輸入為數線上 n 個介於[0,L]之間的座標位置以及他們的切割順序，要找出每個點被切割時的左右端點。把每個點的右端點座標減去左端點座標就是切割時的長度，長度的加總就是所要的答案。




輸入的點給的是座標與切割順序，這一題有三個不同的做法：

1. 直接模擬題目的描述。依照切割順序，對於每一點，在已切割點中查詢當時座標的左右端點，計算出該切割點切割時的長度。

2. 對於每個切割點 P，他被切割時的左端點其實就是 P 的左方切割順序早於 P 而位置最接近 P 的點。同理，右端點就是 P 的右方切割順序早於 P 而位置最接近 P 的點，這裡我們把座標 0 與 L 的切割順序設為 0，也就是最早被切割的點。因此，將點依照座標排序後，剩下的問題就是找出每個點的左右切割順序早於他而最接近的點。

3. 將點依照座標排序後，依照切割的相反順序，每次計算一個切割點的左右端然後「合併」該點左右的棍子(相當於刪除該點)。

第一子題的方法：

```
n,len = map(int, input().split())
if n>1000: exit()
temp=[[0,0]]+[[int(x) for x in input().split()] for i in range(n)]
temp.append([len,0])
cost=0
for [pi,si] in temp:
    if si == 0: continue # dummy left and right point
    cost -= max(p for [p,s] in temp if p<pi and s<si)
    cost += min(p for [p,s] in temp if p>pi and s<si)
# end for
print(cost)
```

我們把每個切割點的[位置,順序]放在 cut[]中，假設切割點已經依照座標排好序了，想像我們從左往右掃描 cut[]。對於每一個點，我們想要找到右邊第一個切割順序早於

它的點，也就是它的右端。我們使用一個資料結構 S 存放那些右端尚未找到的點(依照由左到右的順序)，既然 S 中都是尚未找到右端的點，我們可以知道

S 中的切割順序必然是遞增的。

在掃描到 cut[i]時，對於 S 中那些切割順序大於 cut[i]的點，他們的右端點就是 cut[i]的位置；因為 S 中的切割順序是遞增的，所以我們可以從後往前找到這些點，並切將它們從 S 中刪除(因為它們已經找到右端)。刪除這些點之後，而 S 中的最後一個就是 S 中最大一個切割順序比 cut[i]小的點，也就是 cut[i]的左端！

依照這個方式掃描到最後，S 中剩下的點(尚未找到右端)就是棍子的右端，我們可以在最後加上棍子的右端點 L 當作最後一個切割點，這樣就可以找到所有點的左端與右端。

S 應該使用哪一種資料結構呢？因為只需要拿 S 的最後一個元素來比較，因此堆疊(Stack)就是最合適的。這一題其實類似於前面講解過的 AP325 的 P-3-4 找前方高人(APCS201902Q4 的子題)

```
# using stack for those whose right endpoint are not found
# their orders must be monotonic increasing
n,len = map(int, input().split())
cut = []
for i in range(n):
    cut.append([int(x) for x in input().split()]) # [pos,ord]
cut.sort() # sort by position
cut.append([len,0])
stack = [[0,0]] # [pos,ord]
cost = 0
for p in cut:
    while stack[-1][1] > p[1]: # stack[-1] is top
        cost += p[0] # p is right endpoint of top
        stack.pop()
    cost -= stack[-1][0] # top is left endpoint of p
    stack.append(p)
# end for
print(cost)
```

接下來介紹也很有趣第三種方法：依照切割的相反順序，每次計算一個切割點的左右端然後「合併」該點左右的棍子(相當於刪除該點)。

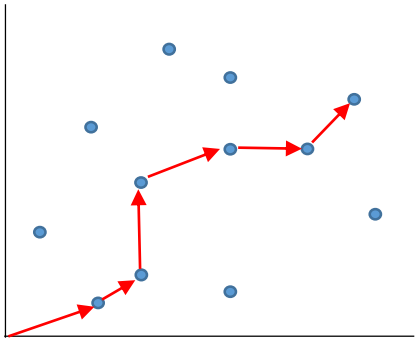
先將點依照位置排序，並在頭尾加上虛擬的點當端點。我們必須記住每個點的左邊與右邊是哪一個點，用 linked list 的觀念，以 left[i]與 right[i]紀錄 i 點的左右點編號。接著將切割順序放入 seq[]中，因為切割順序必然是 1~n 的排列，我們可以直接算排列的反運算：若 y = cut[i][1], 則 seq[y] = i，這裡我們把 1~n 換成 0~n-1。

接著依照 seq[]的相反順序做(最後切割的先做)：找出左右點座標，相減加入 cost，然後消除此點。消除的方法就是將左邊的右鄰居改成右鄰居，右邊的左鄰居改成左鄰居。這樣 p 點就在串列中消失了。

```
# bottom-up merge, linked list
n,len = map(int, input().split())
cut = [[0,0]]
for i in range(n):
    cut.append([int(x) for x in input().split()]) # [pos,ord]
cut.sort() # sort by position
cut.append([len,0])
left = [-1]*(n+2) # left and right points
for i in range(1,n+1): left[i] = i-1
right = [-1]*(n+2)
for i in range(1,n+1): right[i] = i+1
seq = [0]*n # cutting sequence
for i in range(1,n+1):
    seq[cut[i][1]-1] = i
cost = 0
for p in seq[::-1]: # in reverse order
    l,r = left[p],right[p]
    cost += cut[r][0] - cut[l][0]
    # remove p
    left[r] = l; right[l] = r
# end for
print(cost)
```

# 23. f608. 飛黃騰達(AP202101_4)

平面第一象限有 n 個點，從原點出發，只能向右與向上走，要計算出最多能走到幾個點。



這個問題有一個簡化的版本(也就是第一子題)：在一個 0/1 矩陣中，從左下走到右上方，每次只能向右或向上走一步，最多可以走到幾個 1。這個簡化的版本是個簡單的 DP(Dynamic Programming 動態規劃)問題。

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | | | 1 | |
| 0 | | | 1 | | |
| 0 | 1 | | | | 1 |
| 0 | 0 | 1 | 1,2 | 2 | |
| 0 | 0 | 1 | 1 | 1,2 | 2 |
| 0 | 0 | 0 | 0 | 0 | 0 |

對於每個格子(x,y)，計算左邊(x-1,y)與下邊(x,y-1)的較大值，再加上自己的值。計算時，x 由小到大，y 由小到大。

```
n = int(input())
if n>100: exit()
tab = [[0]*101 for i in range(101)]
for i in range(n):
    x,y = map(int, input().split())
    tab[x][y] = 1
for x in range(1,101):
    for y in range(101):
        tab[x][y] += max(tab[x-1][y], tab[x][y-1])

print(tab[100][100])
```

對於第二子題 $O(n^2)$ 的解法。本題沒有相同座標的點，我們將這 n 個點看成一個 dag(directed acyclic graph)，其中若 p 點座標可以走到 q 點座標，則 p 點有一條邊指向 q。這一題要的答案就是在此 dag 上的最長路徑的點數，要計算 dag 的最

長路徑，應該要沿著 topological sort 來計算。我們可以將 dag 建出來，再求 topological sort。此題中兩點有邊的充要條件是 x 與 y 座標值皆非遞減，因此將點依照 x 座標排序(相同 x 值依照 y 值)後就是一個 topological sort，我們可以一面檢查是否有邊，一面檢查最長路徑。

```
n = int(input())
p = []
for i in range(n):
    p.append([int(x) for x in input().split()])
p.sort() # x then y
d = [1]*n
for i in range(n):
    for j in range(i+1,n):
        if p[i][1] <= p[j][1]: # edge from i to j
            d[j] = max(d[i]+1, d[j])
print(max(d))
```

對於本題一般化的情形，如果我們將所有的點依照 X 座標排序後，將 Y 座標看成一個整數序列(y[1],y[2],…, y[n])，對任意 i<j，若 y[i]≤y[j]即表示 i 點可以走到 j 點，所以每一個走法就對應到一個非遞減子序列(Non-decreasing subsequence)，而答案就是要找一個最長的非遞減子序列，也就是一個經典問題 LIS(Longest Increasing Subsequence)的簡單變形(將嚴格遞增改成非遞減)。

第三子題要將原題經過 X 座標排序後求 LIS，所以以下說明 LIS 的解法。

一個序列的子序列是指可以任意挑選某些成員，但是不可以交換他們的前後順序，也可以說是刪除某些元素。在一個數列中找到一個最長的遞增子序列，這個問題被稱為 LIS(longest Increasing subsequence)。假設序列是 y[1:n]，從 DP 的思維開始想，我們定義 lis(i)是以第 i 點結束的最長 LIS 長度，因為前一點一定要比 y[i]小，所以很容易可以寫出遞迴式，為了方便起見，我們在序列前方加一個-1，這樣便不需要初始條件了：

lis(i) = max{lis(j)+1: for j<i and y[j]<=y[i]}。

```
n = int(input())
p = []
for i in range(n):
    p.append([int(x) for x in input().split()])
p.sort() # x then y
y = [-1] + [c[1] for c in p] # a dummy -1
lis = [1]*(n+1)
for i in range(1,n+1):
    lis[i] = max(lis[j]+1 for j in range(i) if y[j]<=y[i])
    # max RE if no dummy -1
```

```
print(max(lis)-1)
```

上面的程式是實作這個遞迴式，顯然需要 O(n²)，因為每個 i 要往前搜尋所有它前面的點。考慮有那些沒有用的計算，假設我們已經做到第 i 個點。對於 j<i，若 y[i]≤y[j] 而 lis(i)≥lis(j)，也就是 y[i]是比較小的結尾但是有比較長的 LIS，那麼 j 顯然是沒有用的，因為可以接在 y[j]後面的一定也可以接在 y[i]後面，而接在 y[i]後面可以得比較好的結果。如果我們刪除那些沒有用的子問題結果，會剩下甚麼呢？每一種可能的長度只會有一個點被留下來，lis 長度為 L 而被留下來的就是「長度 L 的最小可能結尾」。如果我們將這個資料存放在一個陣列 last[]，這個陣列的元素是單調非遞減的，在第 i 個回合計算 lis(i)時，我們搜尋找到 L=max(j: last[j] ≤ y[i])，於是我們得到 lis(i)=L+1，最後我們要將 lis(i)的資料納入，也就是說 L+1 的長度多一個可能結尾是 y[i]，因此
last[L+1]=min(last[L+1],y[i])。
但是還記得 L 是怎麼來的嗎？L 是≤y[i]的最大的那一個 last[]，也就是說除非 L 是 last[]的最後一個元素，否則 last[L+1]>y[i]，因此根本不必多做一次比較。

以下流程是直接根據這樣的思考寫的，作法很直接，序列也不必先讀進來存，讀一個做一個就可以。雖然比 O(n²)好，但是還是不太好，因為時間複雜度其實是 O(nL)，其中 L 是 LIS 的長度，也就是 last[]的長度，在最壞的狀況還是 O(n²)。

```
n = int(input())
p = []
for i in range(n):
    p.append([int(x) for x in input().split()])
p.sort()
y = [e[1] for e in p]
last = [-1] # dummy -1
for yi in y:
    # find the first > yi
    for i in range(len(last)):
        if yi < last[i]: break
    if yi >= last[i]:
        last.append(yi)
    else:
        last[i] = yi
print(len(last)-1) # minus 1 for dummy -1
```

因為 last[]必定為非遞減序列，所以 for 迴圈內的搜尋可以使用二分搜，因此時間複雜度為 O(nlog(n))。以下是使用 bisect 的內建函數做二分搜，因為是要找到第一個>的數，所以是用 bisect_right()。

```
import bisect
n = int(input())
```

```
a = [[int(x) for x in input().split()] for i in range(n)]
a.sort()
last = [-1]
for e in a:
    p = bisect.bisect_right(last, e[1])
    if p < len(last):
        last[p] = e[1]
    else:
        last.append(e[1])
#end for
print(len(last)-1)
```

以下是自己寫二分搜。

```
n = int(input())
a = [[int(x) for x in input().split()] for i in range(n)] #(x,y)
a.sort() # sort by x, breaking tie by y
last = [-1]
for e in a:
    # binary search the first > e[1]
    jump = len(last)//2 # jump distance
    p = 0
    while jump>0:
        while p+jump<len(last) and last[p+jump]<=e[1]:
            p += jump # jump as long as <=e[1]
        jump = jump//2 # half the jump distance
    p += 1 # next is >e[1]
    if p < len(last):
        last[p] = e[1]
    else:
        last.append(e[1])
#end for
print(len(last)-1)
```

## 24. g277. 幸運數字(AP202109_3)

(ZJ上 Python sorting 不一定得100，時間在邊緣)

舉例來說，若 $n = 8$，幸運號碼的序列為 (9, 3, 5, 4, 1, 6, 2, 8)。一個可能的找尋步驟如下：

A. 首先找出序列中最小的幸運號碼是 1，左邊序列是 (9, 3, 5, 4)，而右邊序列是 (6, 2, 8)；左邊幸運號碼總和為 21 而右邊的總和為 16，所以接下來要在左邊找。

B. 對於 (9, 3, 5, 4)，先找出最小的號碼為 3，所分割出的左邊序列為 (9)，右邊序列為 (5, 4)；兩邊總和相同，所以要在右邊找。

C. 對於 (5, 4)，先找出最小的號碼為 4，所分割出的左邊序列為 (5)，右邊為空；所以要在左邊找。

D. 對於 (5)，只有一個幸運號碼，所以要找的幸運號碼就是 5。

題目定義了一個在一個數字序列上抽出一個號碼的過程，在這個過程中，序列的區間會不斷的縮小，直到剩下一個數字為止。這個過程是一個遞迴的定義，所以基本上是一個模擬的題目，重點在於每次如何求出區間的最小值以及切割後兩子區間的總和。

如果用最直白的算法，程式很好寫的，但是跑不快。

```
n = int(input())
lucky = [int(x) for x in input().split()]
while len(lucky) > 1:
    #m = min([(v,i) for i,v in enumerate(lucky)])[1]
    imin = min(lucky)
    m = lucky.index(imin)
    t1 = lucky[:m]
    t2 = lucky[m+1:]
    if sum(t1) > sum(t2):
        lucky = t1
    else:
        lucky = t2
#end while
print(lucky[0])
```

求任一區間總和是一個經典而常見的資料結構問題，在本題中，原始序列並不會變動，因此有個很簡單的做法就是利用前綴和。對於一個序列 S，所謂前綴和是指對於所有 1 ≤ i ≤ n，前 i 個元素的和，這 n 個和可以在前置處理時很簡單的計算出來：

```
// all prefix sums of S
```

```
prefix[0] = 0;

for i = 1 to n do

     prefix[i] = prefix[i-1] + S[i];
```

在需要計算一個區間[x, y]的總和時，只要使用一個減法就可以做到：

prefix[y] – prefix[x-1]

第二個問題是要找區間最小值，這個比較複雜，找區間最小或最大稱為 range minimum/maximum query (RMQ)，有一些(複雜的)資料結構可以提供有效率的查詢。同樣的，這一題的序列並不會改變而且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內部的子區間，所以我們可以很簡單的以下面的方式來查詢：

```
// 區間最小值位置的查詢
將序列S的每一個元素與它的位置(S[i],i)依照第一欄位S[i]進行排序後放入一個list Q;
left = 1, right = n;
while left < right do
    // 找出[left, right]區間中最小值的位置mid;
    while Q中最小值(S[i],i)的i不在[left, right]內 do
        將(S[i],i)丟棄;
    Q中最小值(S[i],i)即是目前區間的最小值;
end while
```

因為區間是逐步縮減，所以當目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。這樣做的時間複雜度如何呢？一開始進行一次排序是O(nlog(n))，在 while 迴圈中每次雖然可能要看多個資料後才找到最小值，但總共也就只會看 n 個資料，所以，除了一開始的排序外，所有的最小值查詢只會花O(n)。

```
n = int(input())
lucky = [0]+[int(x) for x in input().split()]
prefix = [0]*(n+1) # prefix sum
for i in range(1,n+1):
   prefix[i] = prefix[i-1]+lucky[i]
minq = [(lucky[i],i) for i in range(1,n+1)]
minq.sort(reverse=True) # list as a queue of minimum
left = 1; right = n # range[left, right]
```

```
while left < right:
    # find minimum in [left,right]
    while True:
        x = minq.pop()
        if left <= x[1] <= right:
            break
    m = x[1]
    sum1 = prefix[m-1]-prefix[left-1]
    sum2 = prefix[right]-prefix[m]
    if sum1 > sum2:
        right = m-1
    else:
        left = m+1
#end while
print(lucky[left])
```

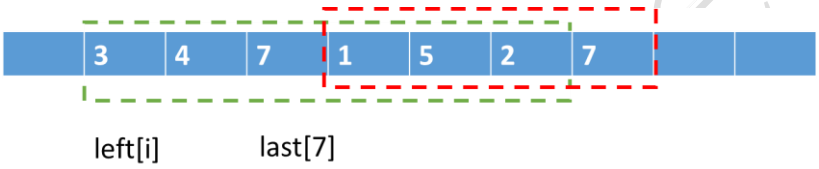
# 25. g278. 美食博覽會(AP202109_4)

(a[i] from 1, 50% TLE)

給一個整數序列，把其中的整數想像成顏色的編號。我們要從中取出 k 段不重疊的區段，每個區段不可以有同色的格子，希望 k 段的總長度越長越好。

當 k=1 時，本題就是找最長的一段，就是 AP325 的 Q-3-11(請見 APCS201906_4 完美彩帶)。對於每一點 i 都計算出最遠的左端 left[i]，滿足[left[i],i]之間沒有同色。在 k>1 時，需要使用動態規劃的技巧來計算了。

先討論 k=1 的情形。考慮滑動視窗的做法，始終維護一個視窗區間[left[i],i]使得視窗內沒有同色。我們用一個表格 last[x]紀錄顏色 x 上一次出現的位置，每次將區間右端 i 往右移動一格到 i+1 時，如果 last[color[i+1]]<left[i]，則 color[i+1]並未出現在視窗內，所以將視窗右端移到 i+1 仍然是無同色的狀態，也就是說 left[i+1]=left[i]。否則，last[color[i+1]]>=left[i]，也就是撞同色了，這時，視窗的左端需要移動到 left[i+1]=last[color[i+1]]+1。



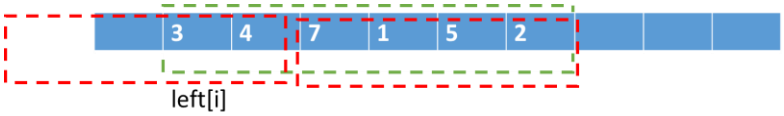

如果目前的視窗[left(i), i] 是"以 i 為右端的最大無同色區段"。
left(i+1) = max(left(i), last[color[i+1]]+1)

接著要說明的第二子題是 k>1 時的做法。

定義 f(j,i)是在 i 點之前選 j 個區間的最佳解(最長總和)，我們很容易寫出以下遞迴式：

- 若 j=0 或 i=0，初始值 f(j,i)=0;
- 如果第 i 點不在解裡面：f(j,i)=f(j,i-1);
- 如果第 i 點在解裡面，最後一個區間是以 i 為右端越長越好的區間，也就是[left[i],i]]，在 left[i]左邊要取 j-1 個區間，所以：
  f(j,i)=f(j-1,left[i]-1)+(i-left[i]+1);

關於第三點，為何以 i 為右端時，最後一段得區間越長越好呢？我們可以證明如下：




若有一解最後一個區間是[x,i]而 x>left[i]，也就是沒有取到最長。那麼我們將這個解的最後一個區間置換成[left[i],i]，區段的長度會增加 x-left[i]，而這個延長雖然可能會讓前面的區段減損，但減少的不會超過 x-left[i]，所以這個置換不會減少解的總長度。由此可知，一定有個最佳解最後一個區間是取到最長的。

```
n,k = map(int,input().split())
a = [0]+[int(x) for x in input().split()]
prev =[0]*100001 # last
left = [0]*(n+1)
# for each i, [left[i],i] is a maximal range
for i in range(1,n+1):
    left[i] = max(left[i-1],prev[a[i]]+1)
    prev[a[i]] = i
#
p = [0]*(n+1)
d = [0]*(n+1)
for j in range(k): # from d to p
    for i in range(1,n+1):
        p[i]= max(p[i-1],d[left[i]-1]+i-left[i]+1)
    p,d = d,p
print(d[n])
```

本題無法通過 ZJ 100%，原因是考試時的 python 時限是 4 秒，ZJ 是 3 秒，judge 機器也不同

# 26. g597. 生產線(AP202111_3)

有 n 個機台以及 n 個傳輸器，要將傳輸器搭配給機台，讓總傳輸時間最小。機台的傳輸量來自 m 個工作，每個工作或用到編號在[s(i),f(i)]區間的機台，而且在所用到的每個機台需要傳輸 w(i)的資料量。

因為工作使用到的機台是不會改變，因此每個機台的資料傳輸需求是根據題目的輸入確定的，假設 n 個機台的傳輸量先算出來是 f(1),f(2),…,f(n)，而 n 個傳輸器的每單位傳輸時間是輸入的 t(1),t(2),…,t(n)，那麼傳輸器與機台該如何搭配呢？

顯然傳輸量大的機台應該配給速度快的傳輸器。

這是一個貪心法則的思考，可以很容易證明如下：不失一般性，假設我們將 t 先排序過使得 t(i) ≤ t(i+1)。假設 f(i)<f(i+1)，交換兩個工作的總時間增加量是

f(i+1)t(i)+f(i)t(i+1) − (f(i)t(i)+f(i+1)t(i+1))
=(f(i+1)-f(i))(t(i)-t(i+1)) ≤ 0

也就是說，交換兩者可以讓總時間減少或相等。因此可知，傳輸器依照單位傳輸時間由小排到大時，機台工作量應該由大排到小。剩下的問題是如何計算機台的傳輸量。

每個工作是使用到一個區間編號的機台，把一個工作想像成數線上在區間[s(i),f(i)]的一根線段，而厚度為 w(i)，我們把 n 根線段疊在數線的對應區間上，每個點的線段厚度就是該點機台的資料傳輸量，現在要有效率的求出這些點的厚度。

這是一個經典的題目。假設線段的數量不多，長度也很短的時候，可以用下面的流程來做：

```
n,m = map(int, input().split())
load = [0]*(n+1)
for i in range(m):
    le,ri,w = map(int, input().split())
    for j in range(le,ri+1):
        load[j] += w
t = [int(x) for x in input().split()]
load = sorted(load[1:])
t.sort()
total = 0
for i in range(n):
    total += t[i]*load[n-1-i]
print(total)
```

這樣做的時間複雜度是線段的長度總和，在本題是 O(mn)，只能通過第一子題的測資範圍。要能夠更有效率的做，我們想像由左而右掃描這些線段，維護目前的厚度，碰到一

根線段的左端點 s(i)，就將厚度增加 w(i)；碰到一根線段的右端點 f(i)，就在 f(i)+1 處減少厚度 w(i)。用差分的觀念來看會更清楚。

**差分序列**

若 p[1~n]是一個序列，我們可以定義 p 的差分序列 d[]：
d[1] = p[1];
d[i] = p[i] - p[i-1] for i>1。

也就是說 d 是 p 的每一項與前一項的差。如果我們對 d 計算前綴和(prefix sum)，
d[i]=d[1]+d[2]+…+d[i]
=p[1]+(p[2]-p[1])+(p[3]-[p2])+…+(p[i]-p[i-1])
=p[i]

也就是說，如果 d 是 p 的差分序列，則 d 的前綴和就是 p，差分與前綴和是反運算，這個道理跟微分與積分是反運算是一樣的，只是這裡是離散函數而非連續函數。

在線段求各點總厚度的問題中，p[]是各點厚度的序列，對於一個線段(s,f):w，其實是對 p[]的[s,f]區間中每一個點都做 w 的修改(稱為區間修改)，而一個區間修改在差分序列上就等於在左端 s 做一個(+w)，而在右端 f+1 做一個(-w)的動作，這樣這一題就清楚了。我們對於一個區間修改，可以在差分序列上做 2 個動作，等到所有區間修改做完畢後，利用 prefix sum 將元序列還原。這樣做的時間複雜度顯然是 O(m+n)而已，非常的快。

在計算完各機台傳輸量總量之後，只要把他與傳輸機各自排序後，讓量最大的機台配號時最小的傳輸機即可，或是將其中一個反轉後做內積。

```
n,m = map(int, input().split())
load = [0]*(n+1)
for i in range(m):
    le,ri,w = map(int, input().split())
    load[le-1] += w
    load[ri] -= w
for i in range(1,n,1):
    load[i] += load[i-1]
t = [int(x) for x in input().split()]
load.pop(-1)
load.sort()
t.sort()
total = 0
for i in range(n):
    total += t[i]*load[n-1-i]
print(total)
```

將一個序列做差分轉換在很多問題上都有運用，尤其在要一個區間做相同修改的問題時，就可以考慮差分轉換。

# 27. g598. 真假子圖(AP202111_4)

本題的題意是說，有很多個邊的集合 E(0), E(1), E(2), …, E(p)，其中 E(0)是個大圖，其他是小圖，小圖有最多三個是假圖，其他是真圖，所有真圖與 E(0)聯集起來是個二分圖，而任何一張假圖與 E(0)的聯集不是二分圖。根據這個線索要找出那些是假圖。

我們先來了解二分圖(bipartite graph)。

關於圖的簡單介紹與基本演算法，可以參見 AP325。二分圖是指一個圖的點可以分成兩個獨立集，所謂獨立集是一群互不相聯的點，所以，二分圖也就是可以將點分成 AB 兩群，使得所有的邊都是連接 1 個 A 中的點與 1 個 B 中的點。

先說明二分圖檢測的方法，只要會做二分圖檢測就可以通過前兩個子題。二分圖以就是可以將點以兩種顏色著色，使得相同顏色的點之間沒有邊。所以我們只要以某一種歷遍(traversal)的方法走訪全部的點，一面走訪一面著色，對於 1 號色的鄰居給與 2 號色，2 號色的鄰居則著 1 號色。因為所有顏色的賦與都是必然的，所以只要碰到無法著色的情形就是非二分圖，否則全部點順利完成著色的話，就是二分圖。

AP325 的習題 Q-7-8(中一中編號 d100)是個單純二分圖檢測的練習題。

輸入擺好；寫個 BFS；寫個迴圈處理每一個連通區塊

對每個小圖 gi 測試 gi+g0。

```
# bipartite test
# for a component
def BFS(G,col,r): # return true if bipartite
    que = [r]
    col[r] = 1
    head = 0
    while head < len(que):
        v = que[head]; head +=1
        for u in G[v]:
            if col[u] == col[v]:
                return False
            if col[u] == 0:
                col[u] = -col[v]
                que.append(u)
            # otherwise col[u] == -col[v]
        # end for
    return True

def bipar(G,n):
    col = [0]*n
    for i in range(n):
        if col[i]==0:
```

```
            if not BFS(G,col,i): return False
    return True
# main start
n, m = map(int, input().split())
g0 = [[] for i in range(n)]
edge = [int(x) for x in input().split()]
for i in range(0,m*2,2):
    g0[edge[i]].append(edge[i+1])
    g0[edge[i+1]].append(edge[i])

p,k = map(int, input().split())
if p>200: exit()
gi=[]
for i in range(p):
    gi.append([int(x) for x in input().split()])
# test one by one
for i in range(p):
    G = [e[:] for e in g0]
    for j in range(0,2*k,2):
        G[gi[i][j]].append(gi[i][j+1])
        G[gi[i][j+1]].append(gi[i][j])
    if not bipar(G,n): print(i+1)
```

對於完全解，因為小圖很多，不能一一檢測，想想看下面這個題目。


我們將一群小圖一起下去做檢測，如果沒事，就通通都是真圖，否則將這一群分成兩群繼續做檢測。因為保證假圖個數不超過 3，所以 3log(p) 次就能找出來。

```
# bipartite test
# for a component
def BFS(G,col,r): # return true if bipartite
    que = [r]
```

```
    col[r] = 1
    head = 0
    while head < len(que):
        v = que[head]; head +=1
        for u in G[v]:
            if col[u] == col[v]:
                return False
            if col[u] == 0:
                col[u] = -col[v]
                que.append(u)
            # otherwise col[u] == -col[v]
        # end for
    return True

def bipar(G,n):
    col = [0]*n
    for i in range(n):
        if col[i]==0:
            if not BFS(G,col,i): return False
    return True
# main start
n, m = map(int, input().split())
g0 = [[] for i in range(n)]
edge = [int(x) for x in input().split()]
for i in range(0,m*2,2):
    g0[edge[i]].append(edge[i+1])
    g0[edge[i+1]].append(edge[i])

p,k = map(int, input().split())
gi=[]
for i in range(p):
    gi.append([int(x) for x in input().split()])
# binary test
stack = [[0,p-1]] # [left,right]
ans = []
while len(stack):
    [le,ri] = stack.pop()
    G = [e[:] for e in g0]
    for i in range(le,ri+1):
        for j in range(0,2*k,2):
          G[gi[i][j]].append(gi[i][j+1])
          G[gi[i][j+1]].append(gi[i][j])
    if not bipar(G,n):
        if le == ri: ans.append(i+1)
        else:
            mid = (le+ri)//2
            stack += [[le,mid],[mid+1,ri]]
#end stack
ans.sort()
for x in ans: print(x)
```