Lösningsförslag Lägertävling 2018

Företagsrykte

Delpoäng 1: För denna gruppen räcker det att skriva en bruteforce, där vi vid varje dag testar att nollställe ryktet eller inte. Detta implementeras enklast rekursivt och leder till en $O(2^n)$ lösning.

Delpoäng 2: Detta problemet löses med DP. Vi vill ogärna ha ryktet som en state, då denna kan vara uppemot 10^{11}. För att slippa hantera detta använder vi ett vanligt trick för DP:s av dessa slag. Istället för att hålla koll på tillstånd i formen (dag, rykte), så kan vi istället hålla koll på tillstånd i formen (dag där ryktet är 0). Detta innebär att vi får $O(n)$ states, varav varje representerar minsta möjliga totala förlusten denna dag om vi har noll rykte. Detta leder till följande transition mellan states: $dp[i] = k + min(dp[j] + cost(j, i)), j < i$. Alltså, för varje dag kollar vi hur mycket pengar vi hade förlorat om vi inte gör något mellan varje tidigare dag och tar den minsta av dessa. Sedan betalar vi k för att nollställa ryktet. Denna DP-relationen kan enkelt implementeras i $O(n^2)$. När vi skriver ut vårt svar måste vi subtrahera k, då vi inte bryr oss om ryktet den sista dagen.

Full poäng

För fullt poäng kan vi kolla noggrannare på DP-relationen. Detta är en ganska vanlig relation, som kan optimiseras om $cost(j, i)$ är konvex, konkav eller linjär. Vi kan då söka efter ett uttryck för att beräkna cost(j,i) i $O(1)$ tid. För att göra detta kan vi studera uttrycket för totala förlusten om vi inte gör något. Denna blir: $(r_1) + (r_1 + r_2) + (r_1 + r_2 + r_3) + ...$ Om vi börjar från låt säga dag 3 vill vi snabbt omvandla ovanstående uttryck till $(r_3) + (r_3 + r_4) + ...$ För att göra detta gör vi två prefixsummor, $f[i]$ och $r[i]$, där $f[0] = r_1$, $f[1] = (r_1) + (r_1 + r_2)$ och $r[0] = r_1$, $r[1] = r_1 + r_2$. Vi visar nu ett exempel: låt säga att vi vill räkna förlusten från dag 3 till dag 5. Vi utgår ifrån $f[4] = (r_1) + (r_1 + r_2) + .. + (r_1 + r_2 + r_3 + r_4 + r_5)$. Vi kan först subtrahera bort $f[2]$ för att få bort $(r_1) + (r_1 + r_2)$, som är helt utanför vårt intervall. Det som är fel nu är att r_1 och r_2 summeras i varje term. Det finns 5-3+1=3 termer, därmed kan vi bli av med dessa genom att subtrahera bort $p[1] * 3$, och vi får därmed kvar $(r_3) + (r_3 + r_4) + (r_3 + r_4 + r_5)$. I allmänhet blir $cost(i, j) = f[i] - f[j-1] - (r[j-1] * (i - j + 1)) = f[i] - f[j-1] - r[j-1] * i + r[j-1] * (j+1)$. Detta låter oss skriva om DP-relationen till följande:

$DP[i] = k + f[i] + min(dp[j] - f[j-1] + r[j-1] * (j+1) + r[j-1] * i), j < i)$. Detta verkar inte simplare, men om vi låter $a(j) = dp[j] - f[j-1] + r[j-1] * (j+1)$ och $b(j) = r[j-1]$ får vi $DP[i] = k + f[i] + min(a(j) + b(j) * i), j < i)$, alltså är den långsamma delen av dp-relationen att evaluera ett flertal linjära funktioner vid x=i och hitta den minsta av dessa! Detta visar sig vara ett standardproblem som kan lösas i log(n) tid genom så kallade convex hull trick. Genom att låta tidigare DP-states förvaras som linjer i datastrukturen som låter oss query:a i log(n) tid får vi en komplexitet av $O(n \log(n))$. Det finns en viss risk att om du implementerar convex hull trick själv kan du få 77 poäng, då du beräknar skärningspunkten av linjer, vilket kan ligga utanför long longs i detta problemet. En bra implementation av convex hull trick är kactl's linecontainer: `https://github.com/kth-competitive-programming/kactl/blob/main/content/data-structures/LineContainer.h` Mer läsning: `https://codeforces.com/blog/entry/63823`. (Det visar sig att eftersom vi bara evaluerar linjerna vid i=0,1,2.. och lutningarna på linjerna är ökande kan problemet lösas i $O(n)$)

Guitar hero

Det viktigaste med den här uppgiften är att inte missförstå statementen: noten får vara på en arbiträr sträng som är högre/lägre.

Delpoäng

Delpoäng 1: Det är ganska uppenbart att det enda fallet där det inte går att placera ut noterna är när det finns ett segment av noter som ökar/minskar $\geq m$ gånger på raken. Med denna insikten går det att komma på en $O(nq)$ lösning, där vi för varje query naivt beräknar de längsta sekvenserna.

Delpoäng ??: med lite förberäkning genom exempelvis sqrt-decomposition går det troligtvis att lösa problemet i $O(n\sqrt{n} + q\sqrt{n})$, vilket antingen ger 50 eller 100 poäng, beroende på hur effektivt det är implementerat och om det krävs log-faktorer eller inte.

Full poäng

För full poäng kan vi inse att uppgiften frågar om det finns en sekvens av längd större än m i intervallet, inte längsta sekvensen. Därmed kan vi formulera om frågan till "finns det ett intervall som ökar/minskar exakt m gånger i intervallet". För att hitta alla dessa intervall kan vi skapa en ny array av inputsiffrorna, där det inte finns några element som är exakt lika med det föregående elementet. Det är mycket lättare att svara på queries i denna. För att kunna svara på queries borde man samtidigt generera en karta på index i originalarrayen till index i den nya. Sedan kan man i linjär tid hitta alla intervall som ökar/minskar $\geq m$ gånger i linjär tid. Sedan kan varje sådant intervall delas upp i ett flertal intervall av exakt längd m, startpositionen på dessa sparas i en array. För

att svara på en query l, r behöver vi endast hitta första siffran s i listan av startpositioner, sådant att $s \geq l$ (kan göras med binary search eller genom att precomputa svaret i $O(n)$ för alla möjliga queries med en tvåpekar lösning). Då blir svaret på queryn $s + m < r$.

Tvåpekarmetoden för att precomputa alla queries kan se ut följande:

```
intervals = [/*alla intervall av exakt längd m*/]
ans = []
ptr = 0

for i in range(n):
    while intervals[ptr] < i:
        ptr += 1
    ans.append(ptr)
```

Det är värt att notera att om problemet inkluderade querys där vi ändrar arrayen eller bad om längsta ökande/minskande sekvensen i ett intervall räcker inte metoden ovan. Då kan man använda ett segmentträd istället. Abdullah Zaghmout hade en bra presentation om det under lägret 2022

Kryssring

6 poäng

För 6 poäng kan man slumpmässigt placera ut kryss ut och ringar (detta kommer fördela de ungefär jämnt, vilket leder till mindre fel över lag) för att få ett accepterbart svar.

20 poäng

För 20 poäng kan man slumpmässigt testa att byta plats på bokstäver i en slumpmässig rad och endast behålla förändringen om det minskar antalet fel. Här kan det vara trevligt att kolla hur man mäter tid i ditt favoritspråk, så att du kan köra fram till 11.5 sekunder. För att räkna antalet fel kan vi skapa en funktion counterrors. Dess korrekthet kan verifieras med hjälp av att antalet fel i sample-fallet är givet.

```
def counterrors(grid,n,m):
    errors = 0
    for i in range(n):
        for j in range(m):
            c = grid[i][j]
            if j >= 2 and grid[i][j - 1] == c and grid[i][j - 2] == c: errors+=1
            if i >= 2 and grid[i - 1][j] == c and grid[i - 2][j] == c) errors+=1
            if i >= 2 and j >= 2 and grid[i-1][j-1] == c and grid[i-2][j-2] == c errors+=1
            if i + 2 < n and j >= 2 and grid[i+1][j-1] == c and grid[i+2][j-2] == c errors+=1
    return errors
```

80 poäng

20 poängs-lösningen ger väldigt bra poäng på delgrupp 1 och 2, men i princip 0 på de större grupperna. Detta beror på att funktionen för att räkna fel kör i $O(n*m)$ tid. Om $n = m = 500$ skulle det ta $500^4 \approx 6*10^{10}$ operationer att byta varje ruta en gång, alltså tar det alldeles för lång tid att räkna ut felen. För att snabba upp räkningen av fel kan vi inse att ett givet byte endast kan påverka fel lokalt. Om man istället endast räknar ut de fel som kan ha ändrats kan vi göra varje byte i $O(1)$ tid. Detta kan se ut ungefär följande. (Nedanstående kod har nån bug i hörnen, men författaren orkar inte fixa de, när det max påverkar ca 1 poäng).

```
def counterrorslocal(x,y):
    directions = [ [0,1],[1,1],[1,0],[1,-1],[0,-1],[-1,-1],[-1,0],[-1,1] ]
    errors = 0
    for dir in directions:
        pos1 = [x+dir[0],y+dir[1]]
        pos2 = [x+dir[0]*2,y+dir[1]*2]

        #check we aren't outside the grid, too lazy to include
        c = grid[y][x]
        if c == grid[pos1[1]][pos1[0]] and c == grid[pos2[1]][pos2[0]]:
            errors += 1

    return errors
def hillclimb():
    while time()<11500:
        row = random(0,n_rows)
        a,b=random(0,cols),random(0,cols)
        errors_before = counterrorslocal(row,a)+counterrorslocal(row,b)
        swap(grid[row][a],grid[row][b])
        errors_after = counterrorslocal(row,a)+counterrorslocal(row,b)
        if errors_after >= errors_before:
            swap(grid[row][a],grid[row][b])
```

96 poäng

Den förra lösningen får absolut sämst poäng på grupp 10, det tomma rutnätet, alltså räcker det inte längre att placera ut kryss och cirklar slumpmässigt (förbättringsalgoritmen på rutnätet är trots allt typ optimalt). För att hitta vad som är ett bra mönster kan man ge en tom grid till sitt program och se vad den hittar. Det visar sig att följande mönster har 0 fel:
XOXO
XOXO
OXOX
OXOX
XOXO

Alltså kan vi girigt försöka placera ut ovanstående mönster så gott som det går och sedan använda förbättringsalgoritmen. Det kan se ut ungefär som följande:

```
def place_initial(grid,x_per_row):
    for i in range(n):
        # number x placed
        x = 0
        symbols = ['x','o']
        if i % 4 > 1: swap(symbols[0],symbols[1])

        for j in range(m):
            # place what we want if possible
            if x < x_per_row[i]:
                grid[i][j] = symbols[j%2]
                x+=symbols[j%2]=='x'
            else:
                grid[i][j] = 'o'

        # make sure to place enough x's even if it worsens the score
        while x < x_per_row[i]:
            j = rand() % m
            if grid[i][j] == 'o':
                grid[i][j] = 'x'
                x+=1
```

110 poäng

Det visar sig att det även funkar bra att göra en girig placering på de andra testgrupperna. Koden för dessa blir nästintill identisk, med enda skillnaden att vi inte skriver över redan placerade kryss/punkter. Detta kan bli lite lurigt i andra loopen, där det inte räcker att kolla grid[i][j] == '.', istället får vi skapa en 2d-array som kommer ihåg om varje ruta originellt var tom eller placerad.

117 poäng

För att få lite mer poäng går det att använda metoden som kallas "simulated annealing". Denna fungerar precis som hill climbing, vilket är vad som gjort hittills, med skillnaden att den med en viss sannolikhet accepterar förändringar som kan försämra vår score. Kriteriet för att den accepterar en förändring är följande: $e^{-D/T} > R(0,1)$, där D är förändringen i score (negativt representerar bra förändring, positiv sämre), T en parameter och $R(0,1)$ ett slumpat tal mellan 0 och 1. Idén är att till en början är T stort, vilket tillåter stora förändringar, men sedan sänks T gradvis tills algoritmen degenererar till vanlig hillclimbing. Poängen med detta är att undvika att fastna i lokala minima, som hillclimbing ofta gör.

Rymdpromenad

Subtask 1

För denna gruppen räcker det att rekursivt testa att gå höger/vänster vid varje fönster. Detta kan relativt lätt göras med en rekursiv funktion.

Subtask 2

Vi kan optimera den rekursiva funktionen subtask 1 med dp.

$$DP[i][r] = \text{min möjligt totalt avstånd}$$

där i är fönstret vi är på just nu, och r är vår rotation. Eftersom rotationen inte kan bli mer mer än $O(NM)$ blir det sammanlagt $O(NM^2)$.

Subtask 3

Vi kan inse att förra subtasken hade överflödig state: det räcker att bara hålla koll på fönstret vi är på just nu och och hur många fler fulla varv höger vi tagit än vänster (kan vara negativ om vi taget fler åt vänster). Men hur hittar vi vår exakta rotation? Varv räcker inte? Jo, fönstret vi är på just nu bestämmer ju vår exakta rotation (om man inte räknar fulla varv. Men vi håller ju koll på antalet fulla varv vi roterat). Detta ger $O(M^2)$ state, nog för subtasken.

Full poäng

Den fulla lösningen är en girig lösning. Den viktiga insikten är följande: om du är vid position k och ska till position a genom höger kostar det $a-k$ steg. Om du väljer vänster kostar det istället $n-(a-k)$ steg. Vad är skillnaden i rotation mellan höger och vänster? Mer exakt, om vi har valt att rotera höger för något fönster men byter från till vänster, hur förändras rotation? Den kommer minska med $a-k$ och också minska med $n-(a-k)$ igen p.ga. annan riktning. Detta blir $-(a-k)-(n-(a-k)) = -n$, alltså ändras rotationen endast med en konstant! Då kan vi starta med en lösning som bara svänger höger. Eftersom rotationen påverkas exakt samma av varje höger till vänster-byte, så kan vi lika gärna göra de bytena som maximerar avståndet vi sparar. Vi gör då detta tills det inte är värt det längre. Vi bryr oss inte heller om att få exakt noll rotation: vi betalar bara i slutet för att rotera tillbaka till start. Här är en fungerande python-lösning:

```
n,m=[int(i) for i in input().split()]
windows = [int(i) for i in input().split()]

# Compute a solution only going clockwise
curr_window = 1
clockwise_costs = []
tot_dist = 0
```

```
rotation = 0
for i in range(m):
    next_window = windows[i]
    if curr_window<=next_window:
        distance = next_window - curr_window
    else:
        distance = n - curr_window + next_window

    tot_dist += distance
    rotation += distance
    clockwise_costs.append(distance)
    curr_window = next_window

# Consider which moves to switch to counter-clockwise
# Do the biggest ones first
clockwise_costs.sort(reverse=True)
for distance in clockwise_costs:
    old_cost = tot_dist + abs(rotation)
    new_cost = tot_dist + n - 2 * distance + abs(rotation-n)
    if new_cost < old_cost:
        tot_dist += n - 2 * distance
        rotation -= n

print(tot_dist + abs(rotation))
```