Biosalong

60 poäng

En lösning man skulle kunna tänka sig är att loopa igenom alla möjligheter för Axels plats, loopa igenom alla möjligheter för Beatrices plats, och ta det kortaste avstånder man hittar. Det skulle se ut ungefär såhär i python:

```
n = int(input())
s = input()

shortest = 100000000000
for i in range(n):
    for j in range(n):
        # Axel och Beatrice kan
        # inte sitta på samma plats
        if i==j:
            continue

        # Ifall båda platserna är lediga
        if s[i]=='.' and s[j]=='.':
            # Uppdatera hittils kortaste
            shortest = min(shortest, abs(i-j)-1)
print(shortest)
```

Skickar man in detta ser man dock att det bara ger 60 poäng, och att man får "Time Limit Exceeded" på den stora testfallsgruppen. Om vi funderar på hur många gånger koden inuti den inre for-loopen körs så förstår vi varför. För varje gång den yttre forloopen körs så körs den inre koden N gånger, och eftersom den yttre forloopen körs N gånger så kommer den inre koden köras $N \cdot N$ gånger. (man kan då säga att programmet har tidskomplexitet $O(N^2)$). Om $N = 1000000$ kommer den inre koden köras 10^{12} gånger, vilket är mycket mer än vad vi hinner på en sekund. I python hinner man ca 10^7 operationer på en sekund.

100 poäng

Så för 100 poäng måste vi få bort en av looparna. Det kan vi göra genom att inse att om vi går igenom alla platser vänster till höger så räcker det att kolla avståndet till den senaste lediga platsen man har sätt, eftersom den kommer alltid vara den närsmta åt vänster.

```
n = int(input())
s = input()

shortest = 100000000000
last_free_pos = -100000000000
for i in range(n):
```

```
    if s[i]=='.':
        shortest = min(shortest, i-last_free_pos-1)
        last_free_pos = i
print(shortest)
```

Robotdammsugaren

Detta är ett rent simuleringsproblem.

Vi vill hålla koll på vår nuvarande position, och sedan exekvera varje kommando ett i taget. När vi exekverar ett kommando stegar vi vår nuvarande position i riktningen kommandot specifierar, tills nästa position är en låda. Medans vi stegar markerar vi också i vår karta att vi städat den rutan. I slutet kan vi sedan räkna i kartan hur många rutor vi städat.

I c++ går det bra att läsa in och spara kartan som strängar, som man uppdaterar medans man rör sig. En detalj man måste tänka på i python dock är att strängar är "immutable", det vill säga man kan inte ändra på dem. Därför behöver vi omvandla strängarna till listor när vi läser in kartan.

```
r,c,n = list(map(int,input().split()))
kommandon = input()

karta = []
pos_c = -1
pos_r = -1

for i in range(r):
    rad = list(input())
    for j in range(c):
        if rad[j]=='0':
            pos_r = i
            pos_c = j

            #Städa startrutan
            rad[j] = ' '

    karta.append(rad)

for k in kommandon:
    riktning_c = 0
    riktning_r = 0
    if k=='v':
        riktning_r = 1
    if k=='^':
        riktning_r = -1
    if k=='>':
        riktning_c = 1
    if k=='<':
        riktning_c = -1

    while True:
```

```
        next_c = pos_c + riktning_c
        next_r = pos_r + riktning_r

        if karta[next_r][next_c] == '#':
            break

        pos_c = next_c
        pos_r = next_r

        #Städa rutan
        karta[pos_r][pos_c] = ' '

#Räkna städade rutor
ans = 0
for i in range(r):
    for j in range(c):
        if karta[i][j]==' ':
            ans+=1
print(ans)
```

Robotdammsugaren 2

Här är några olika strategier och ungefär hur mycket poäng de får:

Slumplösning: (20-30 poäng)

Antagligen det enklaste sättet att få hyfsat mycket poäng är att skriva ut en slumpmässig sträng av längd N (man behöver inte ens läsa in rutnätet). Utan några optimeringar ger det här ungefär 20 poäng. En optimering är att aldrig skriva ut samma tecken två gånger i rad, då får man ca 25 poäng. Sedan visar det sig vara en bra idé att växla mellan upp/ner och höger/vänster, och då får man ytterligare lite poäng. Det var ganska många som tog poäng med den här typen av lösningar.

Girig lösning: (~50 poäng)

I den här lösningen kollar vi åt alla fyra håll och går åt det håll där det finns flest obesökta rutor. Man kan få en hel del poäng med en sån här lösning, men det finns en del fällor. En sak som kan hända är att det finns 0 rutor åt alla håll. Då måste lösningen tie-breaka på något bra sätt så att den inte fastnar och åker in i väggen resten av tiden. Det enklaste sättet är att välja ett slumpmässigt håll om flera var lika bra. Glöm heller inte att se till så att roboten aldrig åker direkt in i en närliggande vägg.

Trädsökning: (75-90 poäng)

Det här är en generalisering av den giriga lösningen. I varje steg kollar vi alla möjliga sekvenser av D steg, och går åt det håll där den bästa lösningen fanns. Om vi undviker att åka direkt in i väggar finns det i varje position 3 (eller färre) olika håll vi kan åka åt, så antalet sekvenser vi kollar igenom blir ca 3^D. Domarlösningen kör en sån här trädsökning för $T \leq 5$ och $T = 9$, med $D = 10$ för de små fallen och $D = 7 - 9$ för de stora. Under tävlingen var det den här typen av lösningar som lyckades bäst, och det finns många fler optimeringar man kan slänga på. Både Olle Lapidus och Victor Vatn skrev t.ex. trädsökningslösningar som är bättre än domarnas.

BFS + DP: (100+ poäng tillsammans med trädsökningen)

På testfall 7 och 10 är nedre halvan av rutnätet en guldgruva som vi vill ta oss till så fort som möjligt och sedan åka fram och tillbaka i. Trädsökningslösningen kollar inte tillräckligt djupt för att hitta ner dit (i alla fall i testfall 10), och kommer därför få väldigt lite poäng. För att hitta kortaste vägen till nedre halvan är det en bra idé att först göra en bredden-först-sökning (BFS) för att komma dit. Men BFS visar sig vara ett bra verktyg även för att hitta bra allmänna lösningar. Vi vill ju hitta en sekvens drag som går långa sträckor på få drag, och som inte korsar sig själv så mycket. Och kortaste vägar som BFSen

hittar är just såna. Om man tar alla kanter i grafen som finns i kortaste vägar så bildar de en riktad acyklisk graf, och vi kan hitta den kortaste väg med flest obesökta noder med dynamisk programmering. Så domarlösningen kör flera såna här BFSer och hittar de bästa kortaste vägarna från där vi befinner oss. Det visar sig funka bra framförallt i $T = 7$ och 10, men även i de stora glesa rutnäten som $T = 8$.

Kontringsattack

50p

För 50 poäng testar vi alla k mellan 0 och den största skillnaden mellan Friberg och Skog på en match (vi kan sätta in 1000 då poängen är mindre än 1000). För varje k räknar vi differensen mellan antalet matcher då Friberg är bättre och antalet matcher då Skog är bättre och skriver ut det k som ger störst differens.

Tidskomplexitet: $O(n \cdot \max(F, S)) \approx 10^6$ operationer

```
#!/usr/bin/env python3
n = int(input())
matcher = []
for i in range(n):
    match = input().split()
    f = int(match[0])
    s = int(match[1])
    matcher.append([f, s])

def lika(f, s, k):
    skillnad = abs(f-s)
    if skillnad <= k:
        return True
    else:
        return False


bästa_diff = -1
bästa_k = -1
for k in range(1001):
    diff = 0
    for match in matcher:
        if not lika(match[0], match[1], k):
            if match[0] > match[1]:
                diff += 1
            else:
                diff -= 1
    if diff > bästa_diff:
        bästa_diff = diff
        bästa_k = k
print(bästa_k)
```

100p

För 100 poäng kan vi inte gå igenom alla matcher för varje k. Istället vill vi gå igenom matcherna på så sätt att det räcker att göra det en gång. Detta kan vi

åstakomma genom att gå igenom matcherna i sorterad ordning så att vi börjar på matchen med störst skillnaden mellan Friberg och Skog och slutar på den med minst. Vi använder en variabel w där vi sparar antalet fler matcher som Friberg har vunnit, samtidigt vet vi att tillhörande k:et är skillnad på nästa match. På så sätt behöver vi bara sparar k värdet då w är som störst.

Tidskomplexitet: $O(n\log(n))$

```
#!/usr/bin/env python3
n = int(input())
differences = []
for i in range(n):
    f, m = [int(i) for i in input().split()]
    if f == m:
        continue
    # differences[i] = (abs(diff), -1 if Friberg == winner else 1)
    differences.append((abs(m-f), -1 if f > m else 1))
# protect against out of bounds error and makes sure that we check k=0
differences.append((0, 0))
differences.append((1e9, 0))
# sorts in decreasing order with the matches that Filip won last
# when they have the same difference this will result in only
# looking at the last game of every difference because total_wins
# will be the highest otherwise you might think that the current k is better than it is.
# ex:        (5, Friberg), (5, Skog), (4, Friberg)...
# total_wins     1     ,    0     ,    -1
# looks like k = 5 will give you 1 win but really it should be:
#            (5, Skog), (5, Friberg), (4, Skog)...
# total_wins     -1    ,      0    ,     -1
differences.sort(reverse=True)
best = (0, 0)
won_matches = 0
for i in range(len(differences)-1):
    # adds the result of the game to the total
    won_matches -= differences[i][1]
    # less or equal because you want the lowest k
    if won_matches >= best[0]:
        # (won_matches, 1+(the differens of the next game))
        # because you will still have the same matches won but with a lower k
        best = (won_matches, differences[i+1][0])
print(best[1])  # best k
```

Lite kortare kod för dem som vill:

Python2 159 bytes

```
b=(0,1e9);w=0
for c,d in sorted([b]+[(lambda f,m:(-abs(m-f),cmp(f,m)))(*map(int,raw_input().split()))for _
```

```
 b=min(b,(w,-c));w-=d
print b[1]
```

Python3 169 bytes

```
b=(0,1e9);w=0
for c,d in sorted([b]+[(lambda f,m:(-abs(m-f),1 if f>m else -1))(*map(int,input().split()))f
 b=min(b,(w,-c));w-=d
print(b[1])
```

Köpa tavlor

Det första man bör inse för att lösa den här uppgiften är att vi kan anta att Mona bara går vänster till höger eller höger till vänster. Om den vänstraste tavlan Mona köper är tavla L och den högraste är tavla R så måste Mona spendera minst $R - L$ sekunder bara på att gå mellan L och R. Men om hon ända ska gå så mycket kan vi lika gärna säga att hon börjar sin tavelhandel på tavla L, går bara åt höger, och slutar på tavla R.

Lösning 1: Memoisering/dynamisk programmering

När vi står vid en tavla har vi två val: antingen kan vi köpa tavlan och gå åt höger till nästa tavla, eller går vi ett steg åt höger utan att köpa tavlan. Om vi definerar en funktion som tar in vilken position vi är på just nu och hur många tavlor vi har kvar att köpa kan vi rekursivt räkna ut vad den kortaste tiden är genom att testa dessa två möjligheter och välja den bästa:

```
#t[i] är hur lång tid det tar köpa i:te tavlan

def shortest_time(nuvarande_position, antal_tavlor_kvar):
    if antal_tavlor_kvar == 0:
        return 0
    if nuvarande_position == n:
        #Vi har kommit till slutet men inte köpt alla tavlor,
        #returnera stort tal för att indikera "omöjligt".
        return 1e9

    #Köp inte tavlan, gå ett steg åt höger
    tid1 =  1 + shortest_time(antal_tavlor_kvar+1, antal_tavlor_kvar)

    #Köp tavlan, gå ett steg åt höger
    tid2 =  1 + t[nuvarande_position] + \
        shortest_time(antal_tavlor_kvar+1, antal_tavlor_kvar-1)

    return min(tid1,tid2)
```

Den här lösningen fungerar men stöter på ett problem: varje gång man anropar `shortest_time` anropas `shortest_time` på nytt två gånger. Dessa två nya anrop kommer vardera anropa två nya, o.s.v tills man kommer till ett av basfallen `antal_tavlor_kvar == 0` eller `nuvarande_position == n`. Det kommer göras exponentiellt många anrop till `shortest_time`, och man får Time Limit Exceeded för allt utom de allra minsta testfallen.

Hur löser vi detta? Jo, vi inser att `shortest_time` bara tar in två argument, och om vi anropar `shortest_time` flera gånger med samma argument så ska svaret bli det samma. Därför kan vi spara svaret efter att vi kört funktionen, och sedan om vi stöter på exakt samma argument igen så kan vi kolla upp vårt sparade svar utan

att behöva köra funktionen igen. Detta kallas för memoisering, och gör att vi bara göra ett riktigt anrop till `shortest_time` för varje möjligt värde på argumenten. `nuvarande_position` kan max ha 2000 värden, och `antal_tavlor_kvar` kan max ha 2000 värden. Alltså behöver vi köra funktionen max $2000 \cdot 2000$ gånger, vilket vi gott och väl hinner inom tidsgränsen. Tidskomplexiteten är $O(Nk)$.

```
#!/usr/bin/env python3
import sys
sys.setrecursionlimit(150000)
mem = [[-1 for _ in range(2001)] for _ in range(2001)]

n,k = [int(i) for i in input().split()]
l = [int(i) for i in input().split()]

def shortest_time(nuvarande_position, antal_tavlor_kvar):
    if antal_tavlor_kvar == 0:
        return 0
    global mem, k, l, n
    if nuvarande_position == n:
        return 1e9

    #Ifall vi redan har kört med de här argumenten
    if mem[nuvarande_position][antal_tavlor_kvar] != -1:
        return mem[nuvarande_position][antal_tavlor_kvar]

    tid1 = 1 + shortest_time(antal_tavlor_kvar+1, antal_tavlor_kvar)
    tid2 = 1 + t[nuvarande_position] + \
        shortest_time(antal_tavlor_kvar+1, antal_tavlor_kvar-1)

    mem[nuvarande_position][antal_tavlor_kvar] = min(tid1,tid2)
    return mem[nuvarande_position][antal_tavlor_kvar]

#Testa alla ställen att starta på och ta det bästa
print(min([dp(i, k)-1 for i in range(n)]))
```

Lösning 2: Iterera + prioritetskö

Det går också att lösa den här uppgiften utan rekursion eller dynamisk programmering. Ifall vi vet både L och R, dvs vilken tavla Mona börjar och slutar på, så kan vi relativt enkelt räkna ut hur lång tid det tar. Insikten är att Mona alltid vill köpa de k tavlorna mellan L och R som går snabbast att köpa. Tiden det tar är alltså $R-L+(summan av de k minsta talen bland t_L, t_{L+1}, ..., t_R). Denna summa kan vi räkna ut på $O((R-L)\log(R-L))$ tid genom att lägga talen i en lista, sortera listan och summera de k första talen. Detta ger en $O(N^3\log(N))$-lösning.

Vi kan också vara lite smartare. Om vi fixerar L och sedan itererar över alla

möjligheter för R, så ändras den här listan av tavlor mellan L och R så mycket i varje steg. Närmare bestämt vill vi kunna lägga till en ny tavla, och sedan snabbt kunna hitta summan av de k snabbaste tavlorna. Detta går att åstakomma med en prioritetskö, vilket är en datastruktur där man snabbt kan lägga in tal, snabbt få reda på vad det största talet är och snabbt plocka bort det största talet. När vi ökar R kan vi lägga in t[R] i vår prioritetskö, och sedan ta bort de största talen från prioritetskön tills den innehåller exakt k element. Om vi håller koll på nuvarande summan av alla element i prioritetskön genom att addera när vi lägger in saker och subtrahera när vi tar bort, så kan vi snabbt få den summan vi vill ha, summan av de k snabbaste tavlorna.

Detta ger en $O(N^2\log(N))$-lösning.

```
#!/usr/bin/env python3
import heapq

class Pqueue:
    def __init__(self, k):
        self.q = []
        self.tot = k-1
        self.k = k
        self.mn = 1e9
    def add(self, val):
        if len(self.q) < k:
            heapq.heappush(self.q, val*-1)
            self.tot+=val
            if len(self.q) == k:
                self.mn = min(self.mn, self.tot)
        else:
            self.tot+=1
            top = heapq.heappop(self.q)*-1
            if self.tot + val - top <= self.tot:
                self.tot += val - top
                self.mn = min(self.mn, self.tot)
                heapq.heappush(self.q, val*-1)
            else:
                heapq.heappush(self.q, top*-1)


n, k = [int(i) for i in input().split()]
l = [int(i) for i in input().split()]
mn = 1e9
n = len(l)
for start in range(n-k+1):
    curl = Pqueue(k)
    for j in range(start, n):
        curl.add(l[j])
```

```
    mn = min(mn, curl.mn)
print(mn)
```

Stökiga känguruungar

Det finns många olika sätt att lösa det här problemet på (bl.a. med dynamisk programmering); det här lösningsförslaget tar upp två olika giriga (greedy) lösningar.

Vi kan börja med att fundera över hur vi löser delgrupper 2 och 4, där alla ungar garanterat är stökiga. Här behöver vi alltså bara kunna beräkna snabbt huruvida ett ord T är en unge till S eller inte. En enkel metod för att göra det är att gå igenom T tecken för tecken, och hitta nästa förekomsten av det tecknet i S. (Det är aldrig meningsfullt att hoppa över tecken i S, därav att algoritmen kan beskrivas som girig.) Om vi lyckas hitta alla tecken i S är T en unge, annars inte. Om vi hittar nästa förekomst i S genom att loopa får vi en tidskomplexitet på $O(NM + K)$ – för vardera av de N orden loopar vi potentiellt igenom hela S, som är M tecken lång, och vi går också igenom alla K tecken hos orden T. Detta klarar testgrupp 2.

För att snabba upp algoritmen måste vi göra steget "hitta positionen för nästa förekomst av tecknet X i S, med start på position I" snabbare. Ett sätt att göra detta på är att ha en förberäknad tabell över "nästa förekomst" av storlek $26 \cdot \mathrm{M}$, som kan beräknas baklänges:

```
vector<int> next(26, -1);
vector<vector<int>> nexts(M);
for (int i = M-1; i >= 0; i--) {
    next[S[i] - 'a'] = i;
    nexts[i] = next;
}
```

Genom att använda `nexts[i][T[j] - 'a'] + 1` för att hoppa framåt i S får vi ned tidskomplexiteten till $O(AM + K)$ där $A = 26$ är alfabetsstorleken, vilket är tillräckligt för att klara testgrupp 4.

Ett annat sätt är att för varje tecken skapa en lista över positionerna där tecknet förekommer, och sen binärsöka bland dessa listor – detta ger $O(A + M + K \log M)$ vilket också är tillräckligt snabbt.

För full poäng i problemet måste vi dock även kunna säga huruvida en unge är stökig, d.v.s. förekommer som subsekvens på minst två olika sätt. Det finns minst två olika sätt att göra detta på. Ett sätt baserar sig på observationen att den giriga algoritmen som beskrivits ovan kommer hitta den vänstraste möjliga placeringen av subsekvensen. Genom att reversera alla strängar och göra samma sak (och sen reversera tillbaka positionerna vi får ut) kan vi även hitta den högraste möjliga placeringen. Om dessa skiljer sig åt är ungen stökig.

Ett annat sätt är att utgå från subsekvenspositionerna vi hittar i den giriga algoritmen, och försöka byta ut varje position mot nästa förekomst av samma tecken. Om det finns en till förekomst, och den förekomsten är innan positionen för tecknet direkt efter, så är ungen stökig.

Se sl.cpp och sl2.cpp i repot med domarlösningar/testdata för implementationer av lösningsförslagen. (Det går att förbättra tidskomplexiteterna något och lösa problemet i $O(A + M + K)$ om man gör det offline, d.v.s. för alla synonymer samtidigt, efter att man läst in hela indatan. Vi lämnar detta som övning åt läsaren!)

Bikupor

Vi börjar med att lösa ett enklare fall, nämligen när vi inte tar någon av de K sista noderna (låt oss kalla dem "tunga" noder och de $N-K$ första för "lätta" noder). Då kommer Ljungström att ta de K tunga noderna och vi vill helt enkelt hitta en delmängd lätta noder som inte har några tunga noder som grannar. Här finns det ingen mening med att ta mer än en nod: om vi har en lösning som innehåller fler noder så är det också en giltig lösning att ta bara en av dem. Så vi loopar igenom alla lätta noder, och för var och en loopar vi igenom dess grannar och kollar om någon av dessa är tung. Om ingen är det så har vi hittat en lösning.

Om man bara implementerar fallet ovan och skriver ut "-1" ifall ingen lösning hittades, så får man 12 poäng. Det beror på att när $K = 1$ så räcker det med att kolla lätta noder. Men i de andra fallen så kan lösningen innehålla tunga noder. Här är en strategi som ger ganska mycket delpoäng: Testa att lägga en tung nod A till vår lösning. En sak som händer då är att Ljungström inte kan ta noden A och kommer istället ta nod $N-K$ (låt oss säga att nod $N-K$ blir en tung nod då). Vi kan nu kolla alla A:s grannar och ifall ingen av dem är tunga så har vi en lösning. Annars så måste vi ta med de tunga grannarna i vår lösning. Därefter blir flera noder tunga, och vi fortsätter kolla på grannarnas grannar osv. Notera att vi lägger till noden A, och därefter lägger vi bara till noder som vi måste lägga till ifall A finns i en lösning. Så om vi kör den här proceduren för var och en av de K sista noderna så kommer den garanterat hitta en lösning om det finns någon. Det enklaste sättet att implementera det på är nog att göra något som liknar Prims algoritm och använda prioritetsköer. Då blir det $O(K(N+M)\log M)$ men det går även att få bort log-faktorn. Den här typen av lösning ger upp till 65 poäng.

Fullpoängslösningen ser lite annorlunda ut. Låt oss säga att vi letar efter en lösning där vi tar X av de $K+X$ sista noderna, och Ljungström tar resterande K. Vi kan anta att vår delmängd är sammanhängade, för om den bestod av flera sammanhängade komponenter skulle vi kunna ta en av dem och få en giltig lösning. Dessutom vill vi ju inte att Ljungströms noder ska vara grannar med någon av våra. Det vi vill göra är alltså att kolla på grafen som bildas genom att bara ta de $K+X$ sista noderna, och hitta en sammanhängande komponent av storlek exakt X ($< X$ funkar också). För att få lösningen tillräckligt snabb behöver vi göra det i nästan konstant tid för varje X. Om vi låter X börja från 1 och öka så får vi att en nod ($N-K-X+1$) läggs till i varje steg. Vissa komponenter kommer då att mergas ihop, och de växer då i storlek. Vi behöver något effektivt sätt att slå ihop komponenter, kolla om två noder är i samma komponent, och hålla koll på komponenternas storlek. Det här är ett jobb för den fantastiska datastrukturen Union-Find, som kan göra dessa operationer i nästan konstant tid! Det enda som återstår är då att snabbt kunna kolla om minsta komponentstorleken är $\leq X$, vilket kan göras genom att ha komponentstorlekarna i någon lämplig datastruktur (t.ex. set i C++). Komplexiteten blir $O(N\log N + M)$.

Värt att notera är att det antagligen går att klara sig utan union-find med fullpoängslösningen, tack vare problemets struktur. Eftersom vi bryter lösningen om den minsta komponenten är $\leq X$ får vi begränsningar på hur små komponenterna kan vara efter ett visst antal steg. Efter $X = \sqrt{N}$ kommer alla komponenter ha minst $\sqrt{N}$ noder, och det finns därför högst $\sqrt{N}$ komponenter då. Det här innebär att varje nod kommer delta i högst $2\sqrt{N}$ merge-operationer (en för varje $X < \sqrt{N}$ och en för varje annan komponent efter $X \geq \sqrt{N}$). Så om vi mergar ihop komponenter genom att söka igenom noder och markera dem (med DFS eller BFS), går det att få ihop en $O(N\sqrt{N})$-lösning som antagligen är tillräckligt snabb.