A-III-1 Pán Buridan a kaviarne

Začnime triviálnym riešením: Pre každú z n^2 križovatiek overíme, či neexistujú dve kaviarne s rovnakou vzdialenosťou od tejto križovatky tak, že vyskúšame všetky dvojice kaviarní.

Vzdialenosť dvoch bodov $[x_1, y_1]$ a $[x_2, y_2]$ v štvorcovej mriežke vypočítame v konštantnom čase ako $|x_1 - x_2| + |y_1 - y_2|$. Označme počet kaviarní k, potom všetkých dvojíc kaviarní je $O(k^2)$. Naše prvé riešenie má teda časovú zložitosť $O(n^2k^2)$. Keďže kaviarní môže byť až toľko čo križovatiek, v najhoršom prípade dostávame zložitosť $O(n^6)$.

Skúšať všetky dvojice kaviarní je však zbytočné – zaujíma nás iba to, či sú nejaké dve rovnako vzdialené od vybranej križovatky. Stačilo by teda vygenerovať zoznam vzdialeností k jednotlivým kaviarniam, usporiadať ho a potom jedným prechodom overiť, či sa v zozname nenachádza nejaká vzdialenosť viackrát. Dostali by sme riešenie v čase $O(n^2k \log k)$, čo vieme zhora odhadnúť ako $O(n^4 \log n)$.

Pozrime sa teraz, aké vzdialenosti sa v tomto zozname môžu objaviť. Ak sa nejaká kaviareň nachádza priamo na vybranej križovatke, jej vzdialenosť je 0, čo je zrejme najmenšia možná hodnota. Naopak, najväčšiu vzdialenosť medzi sebou dosahujú dve križovatky v protiľahlých rohoch štvorcovej siete – sú vzdialené $2n - 2$. Všetky vzdialenosti v zozname sú teda z rozsahu $0, 1, \ldots, 2n - 2$, takže ich vieme usporiadať v lineárnom čase, napríklad CountSort-om, a tým zlepšiť časovú zložitosť na $O(n^2k)$.

V rovnakej zložitosti vieme úlohu vyriešiť aj jednoduchšie: Pre každú križovatku budeme postupne prechádzať všetky kaviarne, pre každú sa pozrieme na jej vzdialenosť a v poli `bool`-ov si zaznačíme, že sme dotyčnú vzdialenosť už videli. Akonáhle sa nám nejaká zopakuje, práve spracúvanú križovatku prehlásime za nevhodnú pre Dávidka.

Ako sme už spomínali, kaviarní môže byť až n^2, čiže toto riešenie má prinajhoršom zložitosť $O(n^4)$. Všimnime si ale, že ak je kaviarní priveľa (viac ako $2n-1$), žiadna križovatka v Manhattane nemôže Dávidkovi vyhovovať. Vyplýva to práve z toho, že v celom Manhattane existuje len $2n - 1$ rôznych vzdialeností. Ak teda máme $2n$ a viac kaviarní, nemôže existovať žiadna dobrá križovatka – vždy totiž máme viac kaviarní ako vzdialeností od nej, a teda musia niektoré dve kaviarne ležať v tej istej vzdialenosti.[1]

Toto pozorovanie nás vedie k jednoduchému vylepšeniu: Ak je k väčšie ako $2n - 1$, pre každú križovatku vypíšeme, že na nej Dávidko nemôže bývať. Iba v opačnom prípade spustíme naše riešenie s časovou zložitosťou $O(n^2k)$. Časovú zložitosť tohto vylepšeného riešenia teraz môžeme zhora odhadnúť ako $O(n^3)$.

Listing programu (Python)

```
n = int(input())
A = [[int(x) for x in input().split()] for i in range(n)]

kaviarne = []
for i in range(n):
    for j in range(n):
        if A[i][j]:
            kaviarne.append((i, j))
k = len(kaviarne)

R = [[False] * n for i in range(n)]
if k <= 2 * n - 1:
    for i in range(n):
        for j in range(n):
            bola = [False] * (2 * n - 1)
            R[i][j] = True
            for ki, kj in kaviarne:
                d = abs(i - ki) + abs(j - kj)
                if bola[d]:
                    R[i][j] = False
                    break
                else:
                    bola[d] = True

for x in R:
    print('␣'.join(('A' if y else 'N') for y in x))
```

[1] Hodnota $2n - 1$ je iba horné ohraničenie. Napríklad od stredu Manhattanu sa ostatné križovatky nachádzajú len v rádovo n rôznych vzdialenostiach. Teda napr. už pre $n + 47$ kaviarní vznikne v strede Manhattanu zóna križovatiek ktoré sú zaručene zlé. Toto ale nebudeme potrebovať, vystačíme si so slabším odhadom $2n - 1$ ktorý platí pre všetky políčka.

Aj vo vzorovom riešení samostatne ošetríme prípad s $2n$ a viac kaviarňami. Ďalej však budeme postupovať akoby z opačnej strany: Pre každú dvojicu kaviarní nájdeme tie križovatky, ktoré sú od oboch kaviarní vzdialené rovnako.

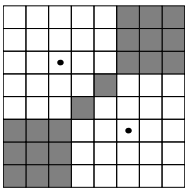
Zafarbime si križovatky ako šachovnicu. Keďže sa v Manhattane môžeme pohybovať len o jednu ulicu v štyroch základných smeroch, každým krokom sa zmení farba križovatky, na ktorej stojíme. To ale znamená, že ak sú dve kaviarne rovnako vzdialené od nejakej križovatky, potom tieto kaviarne musia ležať na križovatkách tej istej farby.

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

Obr. 1: Vzdialenosti od stredného políčka. Všimnite si, že políčka s rovnakou vzdialenosťou majú rovnakú farbu.

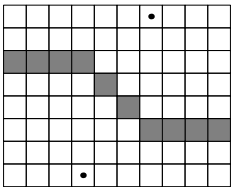
Stačí sa teda venovať len dvojiciam kaviarní s rovnakou farbou. Rozoberme dva prípady.

Prvý prípad: Kaviarne ležia na tej istej uhlopriečke, teda určujú štvorec. Ľahko sa presvedčíme, že hľadané križovatky (tie, ktoré sú rovnako vzdialené od oboch kaviarní) sú tie na opačnej uhlopriečke tohto štvorca a tiež v dvoch rohových oblastiach.



Obr. 2: Sivou farbou sú označené križovatky s rovnakou vzdialenosťou od obidvoch kaviarní (čierne krúžky).

Druhý prípad: Kaviarne neležia na tej istej uhlopriečke, teda určujú obdĺžnik. Tentoraz sa hľadané križovatky nachádzajú na jednej lomenej čiare, znázornenej na nasledujúcom ilustračnom obrázku:



Obr. 3: Križovatky rovnako vzdialené od oboch kaviarní v druhom prípade.

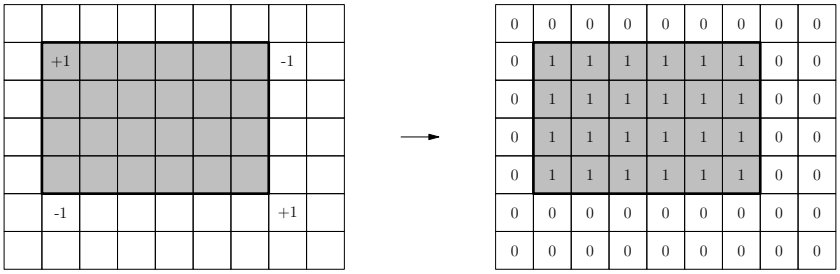
Zostáva nám nájsť zjednotenie týchto oblastí pre všetky dvojice kaviarní. Na to použijeme techniku z domáceho kola – prefixové súčty. Pre každú križovatku spočítame, kvôli koľkým dvojiciam kaviarní je táto križovatka pre Dávidka nevhodná.

Vezmime si najprv obdĺžnikové oblasti. Ak kvôli nejakej dvojici kaviarní nemôže Dávidko bývať v obdĺžniku s ľavým horným rohom na križovatke $[x_1, y_1]$ a pravým dolným rohom na križovatke $[x_2, y_2]$, potom si do pomocného poľa poznačíme nasledovné hodnoty:

- $+1$ na križovatku $[x_1, y_1]$
- -1 na križovatku $[x_1, y_2 + 1]$

- -1 na križovatku $[x_2 + 1, y_1]$
- $+1$ na križovatku $[x_2 + 1, y_2 + 1]$

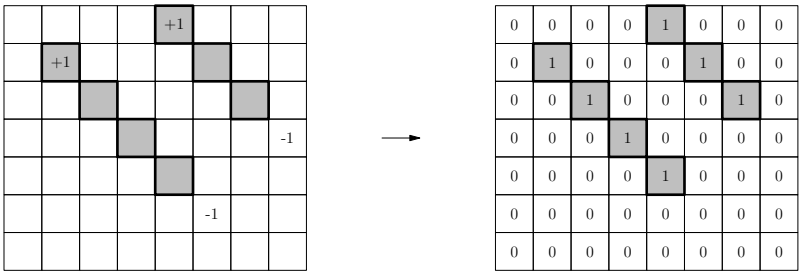
Ak potom na tomto poli spočítame dvojrozmerné prefixové súčty, dostaneme jednotky práve vnútri nášho obdĺžnika:



Obr. 4: Prefixové súčty pre obdĺžniky.

Rovnako to funguje aj s viacerými obdĺžnikmi – za každý najprv pričítame štyri hodnoty $+1/-1$ do pomocného poľa a potom prefixovými súčtami zistíme pre každú križovatku počet obdĺžnikov, ktoré ju prekrylo.

Okrem obdĺžnikových oblastí sa musíme vysporiadať aj s vodorovnými a zvislými čiarami – tie sú ale vlastne tiež obdĺžniky (s výškou/šírkou jeden), teda na nich funguje rovnaká technika. Zostali ešte šikmé čiary. Tie vyriešime podobne – jednorozmernými prefixovými súčtami v smere diagonál.



Obr. 5: Prefixové súčty pre uhlopriečky.

Na záver ešte zhrňme vzorové riešenie. Ak je kaviarní viac ako $2n - 1$, vypíšeme samé `N`. V opačnom prípade pre každú dvojicu kaviarní nájdeme križovatky, od ktorých sú tieto kaviarne vzdialené rovnako. Hranice útvarov, ktoré tieto križovatky tvoria, si poznačíme do pomocného poľa (pre každú dvojicu kaviarní nám to potrvá len $O(1)$). Nakoniec vypočítame prefixové súčty na pomocných poliach, na základe čoho budeme pre každú križovatku vedieť, či k nej existuje dvojica rovnako vzdialených kaviarní.

Celková časová zložitosť vzorového riešenia je teda $O(n^2)$.

Listing programu (Python)

```
n = int(input())
A = [[int(x) for x in input().split()] for i in range(n)]

kaviarne = []
for i in range(n):
    for j in range(n):
        if A[i][j]:
            kaviarne.append((i, j))
k = len(kaviarne)

def pridaj_obdlznik(S, x1, y1, x2, y2):
    if 0 <= x1 <= x2 < n and 0 <= y1 <= y2 < n:
        S[y1][x1] += 1
```

```
        S[y1][x2 + 1] -= 1
        S[y2 + 1][x1] -= 1
        S[y2 + 1][x2 + 1] += 1

def pridaj_kus_uhlopriecky(U, cislo, od, do):
    if 0 <= od <= do < len(U[cislo]):
        U[cislo][od] += 1
        U[cislo][do + 1] -= 1

R = [[False] * n for i in range(n)]
if k <= 2 * n - 1:
    # pomocné pole pre obdĺžniky
    S = [[0] * (n + 1) for i in range(n + 1)]
    # pomocné pole pre uhlopriečky zľava hore doprava dole
    UH = [[0] * (n + 1) for i in range(2 * n - 1)]
    # pomocné pole pre uhlopriečky zľava hore doprava dole
    UV = [[0] * (n + 1) for i in range(2 * n - 1)]
    for k1 in range(len(kaviarne)):
        for k2 in range(k1 + 1, len(kaviarne)):
            y1, x1 = kaviarne[k1]
            y2, x2 = kaviarne[k2]
            if x1 > x2:
                x1, y1, x2, y2 = x2, y2, x1, y1
            hlavna = (y1 <= y2)
            # ležia na rovnakej farbe?
            if (x1 + y1) % 2 != (x2 + y2) % 2:
                continue
            if abs(x1 - x2) == abs(y1 - y2):
                # kaviarne ležia na spoločnej uhlopriečke
                if hlavna:
                    pridaj_obdlznik(S, 0, y2, x1, n - 1)
                    pridaj_obdlznik(S, x2, 0, n - 1, y1)
                    pridaj_kus_uhlopriecky(UV, x1 + y2, y1, y2)
                else:
                    pridaj_obdlznik(S, 0, 0, x1, y2)
                    pridaj_obdlznik(S, x2, y1, n - 1, n - 1)
                    pridaj_kus_uhlopriecky(UH, x1 - y2 + n - 1, x1, x2)
            elif abs(x1 - x2) > abs(y1 - y2):
                # ... neležia a bounding box je širší ako vyšší
                posun = (abs(x1 - x2) - abs(y1 - y2)) // 2
                if hlavna:
                    pridaj_obdlznik(S, x1 + posun, y2 + 1, x1 + posun, n - 1)
                    pridaj_obdlznik(S, x2 - posun, 0, x2 - posun, y1 - 1)
                    pridaj_kus_uhlopriecky(UV, x1 + y1 + posun + abs(y1 - y2), y1, y2)
                else:
                    pridaj_obdlznik(S, x1 + posun, 0, x1 + posun, y2 - 1)
                    pridaj_obdlznik(S, x2 - posun, y1 + 1, x2 - posun, n - 1)
                    pridaj_kus_uhlopriecky(UH, x1 - y1 + posun + abs(y1 - y2) + n - 1, x1 + posun, x2 - posun)
            else:
                # ... neležia a bounding box je vyšší ako širší
                posun = (abs(y1 - y2) - abs(x1 - x2)) // 2
                if hlavna:
                    pridaj_obdlznik(S, 0, y2 - posun, x1 - 1, y2 - posun)
                    pridaj_obdlznik(S, x2 + 1, y1 + posun, n - 1, y1 + posun)
                    pridaj_kus_uhlopriecky(UV, x1 + y1 + posun + abs(x1 - x2), y1 + posun, y2 - posun)
                else:
                    pridaj_obdlznik(S, 0, y2 + posun, x1 - 1, y2 + posun)
                    pridaj_obdlznik(S, x2 + 1, y1 - posun, n - 1, y1 - posun)
                    pridaj_kus_uhlopriecky(UH, x1 - y1 + posun + abs(x1 - x2) + n - 1, x1, x2)
    for i in range(n):
        for j in range(n):
            if i > 0:
                S[i][j] += S[i - 1][j]
                UV[j + i][i] += UV[j + i][i - 1]
            if j > 0:
                S[i][j] += S[i][j - 1]
                UH[j - i + n - 1][j] += UH[j - i + n - 1][j - 1]
            if i > 0 and j > 0:
                S[i][j] -= S[i - 1][j - 1]
            R[i][j] = (S[i][j] == 0 and UH[j - i + n - 1][j] == 0 and UV[j + i][i] == 0)

for x in R:
    print('␣'.join(('A' if y else 'N') for y in x))
```

A-III-2 Preťahovanie lanom

Stručný popis riešenia: Na polynomiálnu časovú zložitosť nám stačí použiť dynamické programovanie, pri ktorom si pamätáme, koľko mladých a koľko starých už odišlo. Lepšie riešenia sú založené na dodatočnom pozorovaní že vždy existuje optimálne riešenie, v ktorom najskôr odchádzajú starí a až potom mladí hráči. Pre konkrétny počet starých hráčov ktorí odídu vieme maximálny počet mladých nájsť binárnym vyhľadávaním. Existuje aj ešte šikovnejšie riešenie, ale pri ňom je treba dať dobrý pozor na detaily.

Predpočítanie

Ľahko nahliadneme, že v ľubovoľnom okamihu tvorí každý z tímov súvislý úsek hráčov zo vstupu. Aby sme vedeli v konštantnom čase povedať súčet síl hráčov v ľubovoľnom takomto úseku, stačí, keď si predpočítame prefixové súčty síl hráčov. Formálne, nech $s_1, \ldots, s_n$ sú sily hráčov, ktoré sme dostali na vstupe (v poradí od najmladšieho po najstaršieho). Definujme $p_0 = 0$ a $\forall i : p_{i+1} = p_i + s_{i+1}$. Tieto hodnoty vieme vypočítať v čase $O(n)$ a zjavne platí, že p_i je súčet síl i najmladších hráčov.

Uvažujme teraz úsek hráčov, ktorý začína i-tym a končí j-tym najmladším. Toho súčet síl môžeme zjavne pomocou predpočítaných prefixových súčtov vyjadriť ako $p_j - p_{i-1}$.

(Poznámka: Existujú aj riešenia, ktoré si vedia poradiť aj bez tohto predpočítania – ak vieme, aké boli sily oboch tímov, vieme v konštantnom čase prepočítať, ako sa zmenili odchodom jedného z hráčov. Keďže však toto predpočítanie vieme spraviť v lineárnom čase, teda v podstate zadarmo, budeme ho pre jednoduchosť používať aj v riešeniach, ktoré by sa bez neho zaobišli.)

Skúšame všetky možnosti

Začneme jednoduchým rekurzívnym riešením, ktoré vyskúša všetky možné priebehy turnaja (teda všetky možné poradia odchádzajúcich hráčov) a vyberie najlepší z nich.

V každom kroku toto riešenie skontroluje, či ešte turnaj neskončil, a ak nie, tak postupne vyskúša obe možnosti pre nasledujúcu zmenu (t.j. raz pošle preč najmladšieho a raz najstaršieho z ešte hrajúcich), pre každú z nich rekurzívne nájde ako najdlhšie ešte mohol turnaj trvať, a následne vráti lepšiu z oboch možností (plus jedna za aktuálny zápas).

Akú má toto riešenie časovú zložitosť? V najhoršom možnom prípade môže turnaj mať až $n-k+1$ zápasov. Ak by toto nastalo pre každé možné poradie odchodu hráčov, vyskúšali by sme až 2^{n-k} rôznych poradí, a aj naša časová zložitosť by teda bola $\Theta(2^{n-k})$. A tento najhorší možný prípad skutočne nastáva – napr. v situácii kedy je kopec taký strmý, že kým má dolný tím aspoň jedného hráča, vždy vyhrá.

Listing programu (Python)

```
data = input().split()
N, K, Q = int( data[0] ), int( data[1] ), float( data[2] )
S = [ int(x) for x in input().split() ]

P = [0]
for s in S: P.append( P[-1]+s )

def solve(lo,hi):
    # vráti maximálny počet zápasov pre turnaj,
    # v ktorom ešte hrajú hráči s číslami lo..hi-1 (číslované od 0)
    sucet_hore = P[hi] - P[hi-K]
    sucet_dole = P[hi-K] - P[lo]
    if sucet_hore > Q*sucet_dole + 1e-9: # týmto zápasom turnaj končí
        answer = 1
    else: # turnaj bude pokračovať, vyberieme lepšiu možnosť koho poslať preč
        a1 = solve(lo+1,hi)
        a2 = solve(lo,hi-1)
        answer = 1 + max(a1,a2)
    return answer

print( solve(0,N) )
```

Memoizácia

V predchádzajúcom riešení je ľahko vidieť, prečo je neefektívne – zbytočne zas a znova počítame odpoveď na tie isté otázky. Rekurzívna funkcia `solve` počas výpočtu volá samú seba exponenciálne veľa ráz. My si však môžeme všimnúť, že v skutočnosti je *rôznych* volaní funkcie `solve` veľmi málo. Parametre `lo` a `hi` majú vždy hodnoty z rozsahu 0 až n. A navyše dokonca vždy platí, že `hi` je aspoň o k väčšie ako `lo`, inak by už turnaj dávno skončil. Počas celého behu predchádzajúceho riešenia sa teda dokola počíta hodnota funkcie `solve` pre iba $O((n-k)^2)$ rôznych vstupov.

Štandardnou technikou, ako takýto algoritmus zefektívniť, je *memoizácia*: vždy, keď prvýkrát vypočítame nejakú návratovú hodnotu funkcie `solve`, zapamätáme si ju. A vždy, keď v budúcnosti náš program zavolá funkciu

`solve` s tými istými parametrami, namiesto toho, aby sme ju znova vyhodnotili (pod čím si treba predstaviť celý strom rekurzívnych volaní), jednoducho rovno v konštantnom čase dáme na výstup zapamätanú hodnotu.

Takto vylepšený algoritmus bude až prekvapivo efektívny. Jeho časovú zložitosť môžeme odhadnúť nasledovne: pre každú platnú kombináciu parametrov `lo` a `hi` sa telo funkcie `solve` (teda tá jej časť, ktorú sme mali už v predchádzajúcom riešení) vykoná najviac jedenkrát. A samotné vykonanie tela funkcie `solve` prebehne v konštantnom čase.[2] Preto je celková časová zložitosť nanajvýš priamo úmerná počtu rôznych vstupov, pre ktoré potrebujeme funkciu `solve` vyhodnotiť – a teda je časová zložitosť nášho algoritmu $O((n-k)^2)$.

Naša implementácia uvedená nižšie má o chlp horšiu časovú zložitosť $O(n^2)$ kvôli inicializácii tabuľky, v ktorej si pamätáme vypočítané hodnoty. Toto by sa pochopiteľne dalo spraviť aj lepšie, ale bolo by to na úkor čitateľnosti programu.

Listing programu (Python)

```
# načítanie a predspracovanie vyzerá rovnako ako v predchádzajúcom riešení

memo = [ [ None for hi in range(N+1) ] for lo in range(N+1) ]

def solve(lo,hi):
    # vráti maximálny počet zápasov pre turnaj,
    # v ktorom ešte hrajú hráči s číslami lo..hi-1 (číslované od 0)

    # ak už poznáme odpoveď pre tieto vstupy, rovno ju vrátime
    if memo[lo][hi] is not None:
        return memo[lo][hi]

    # ak túto kombináciu (lo,hi) vidíme prvýkrát, vyriešime ju
    sucet_hore = P[hi] - P[hi-K]
    sucet_dole = P[hi-K] - P[lo]
    if sucet_hore > Q*sucet_dole + 1e-9: # týmto zápasom turnaj končí
        answer = 1
    else: # turnaj bude pokračovať, vyberieme lepšiu možnosť koho poslať preč
        a1 = solve(lo+1,hi)
        a2 = solve(lo,hi-1)
        answer = 1 + max(a1,a2)

    # a pred tým ako vrátime odpoveď si ju zapamätáme
    memo[lo][hi] = answer
    return answer

print( solve(0,N) )
```

Dynamické programovanie

To isté riešenie ako v predchádzajúcej časti vieme implementovať aj v iteratívnej podobe: pôjdeme napríklad postupne od menších počtov hráčov k väčším. V okamihu, kedy potrebujeme zistiť, ako dlho môže trvať turnaj, ak ešte hrajú hráči s číslom 4 až 17, už vieme aj najdlhšie trvanie pre turnaj hraný hráčmi s číslom 5 až 17, aj trvanie pre turnaj s hráčmi 4 až 16. Vieme teda v konštantnom čase vypočítať optimálnu dĺžku trvania pre práve spracúvaný turnaj.

Všimnite si, že implementácia je v princípe totožná s implementáciou predchádzajúceho riešenia – len rekurzívne volania funkcie `solve` sú v tomto prípade nahradené pohľadom do už vyplnenej časti tabuľky.

Listing programu (Python)

```
# načítanie a predspracovanie vyzerá rovnako ako v predchádzajúcom riešení

for dlzka in range(K,N+1):
    for lo in range(N+1-dlzka):
        hi = lo+dlzka
        sucet_hore = P[hi] - P[hi-K]
        sucet_dole = P[hi-K] - P[lo]
        if sucet_hore > Q*sucet_dole + 1e-9: # týmto zápasom turnaj končí
            answer = 1
        else: # turnaj bude pokračovať, vyberieme lepšiu možnosť koho poslať preč
            a1 = memo[lo+1][hi]
            a2 = memo[lo][hi-1]
            answer = 1 + max(a1,a2)
        memo[lo][hi] = answer

print( memo[0][N] )
```

[2] Všimnite si, že do tohto konštantného času nerátame prípadné rekurzívne volania. Tie totiž buď tiež vrátia výstup okamžite, alebo ich započítame inokedy.

Zjednodušenie úlohy

Skôr, než sa pustíme do lepších riešení, si trochu zjednodušíme problém, ktorý ideme riešiť.

Najskôr ošetríme špeciálny prípad, keď turnaj skončí hneď prvým zápasom. Odteraz teda budeme predpokladať, že existuje riešenie tvorené aspoň dvoma zápasmi.

Všimnime si teraz *predposledný* zápas v *optimálnom* riešení. Po tomto zápase musí byť jedno, ktorý hráč odíde – obe možnosti musia viesť k zápasu v ktorom horný tím vyhrá. Každé optimálne riešenie teda končí postupnosťou „predposledný zápas – hocikto odíde – posledný zápas – koniec“. Túto časť riešenia odteraz budeme ignorovať.

Formálne si náš problém upravíme nasledovne: Pre konkrétne i a j budeme hovoriť, že stav turnaja, v ktorom majú aktívni hráči čísla i až j, je *živý*, ak by nasledujúcim zápasom ešte turnaj neskončil. Inými slovami, v živom stave je ešte horný tím prislabý v porovnaní s dolným.

Namiesto pôvodnej úlohy budeme teraz riešiť ekvivalentnú: nájsť najdlhšiu postupnosť odobratí hráča (vždy najmladšieho alebo najstaršieho) takú, že všetky stavy turnaja, ktoré počas odoberania nastanú, sú živé – a to vrátane stavu po odobratí posledného hráča.

Skoro optimálne riešenie

Ak chceme ešte lepšie riešenie ako to, ktoré sme vyššie dosiahli použitím memoizácie, nemôžeme si dovoliť ani len sa pozrieť na všetky možné dosiahnuteľné stavy počas turnaja. Potrebujeme teda nájsť nejaké kritérium, ktoré nám umožní sústrediť sa len na niektoré z nich. Prípadne sa namiesto konkrétnych stavov môžeme zamerať na hľadanie konkrétnych *priebehov* turnaja. Naším cieľom je teda objavenie nejakého tvrdenia tvaru „Vždy bude existovať optimálny priebeh turnaja, ktorý spĺňa [túto dodatočnú vlastnosť].“

Poďme sa teda pozrieť na to, v akom poradí sa najviac oplatí hráčov odoberať. Predpokladajme, že sa niekedy počas riešenia našej novej úlohy odohrala nasledovná postupnosť udalostí:

- Sme v živom stave S_0 tvorenom hráčmi i až j.
- Odišiel najmladší hráč (číslo i).
- Sme v živom stave S_1 tvorenom hráčmi $i+1$ až j.
- Odišiel najstarší hráč (číslo j).
- Sme v živom stave S_2 tvorenom hráčmi $i+1$ až $j-1$.

Pozrime sa na stav S_2. Keďže je tento stav živý, hráči na vrchu kopca v nasledujúcom zápase ešte nevyhrajú. V stave S_2 sú na vrchu kopca hráči s číslami $j-k$ až $j-1$ a na jeho spodku hráči s číslami $i+1$ až $j-k-1$.

Predstavme si teraz, že by sme *najskôr* poslali preč hráča číslo j a až potom hráča číslo i. Čo by sa tým zmenilo?

Stav S_2 by sa tým nezmenil vôbec – stále by sme v ňom mali hráčov s číslami $i+1$ až $j-1$. Zmenil by sa stav S_1. V tom by teraz dole stáli hráči s číslami i až $j-k-1$ a hore hráči s číslami $j-k$ až $j-1$. No a teraz príde dôležité pozorovanie: *aj tento stav musí byť živý.* Prečo? Lebo je to ten istý stav ako S_2, len *navyše* máme dole aj hráča s číslom i. Tým skôr je teda dolný tím dostatočne silný.

Túto úvahu môžeme ľubovoľne veľakrát zopakovať. Ak teda začneme s ľubovoľným riešením, vieme postupne vymieňať kroky, kedy posielame preč mladých hráčov, s krokmi, kedy posielame preč starých. Na konci takto dostaneme *rovnako dobré* riešenie, v ktorom *najskôr* pošleme preč niekoľko starých hráčov a *až následne* niekoľko mladých. Platí teda nasledovné tvrdenie: **Vždy existuje optimálne riešenie** (našej upravenej úlohy), **v ktorom najskôr posielame preč najstarších hráčov a až potom najmladších.**

(Ten istý argument inými slovami: predstavme si, že už vieme, ktorých starých a ktorých mladých hráčov pošle preč optimálne riešenie. Potom určite môžeme poslať preč najskôr tých starých – zatiaľ totiž všetci mladí, ktorých časom chceme poslať preč, ešte stoja na spodku kopca a pomáhajú tak dolnému tímu. Ak by sme mladých poslali preč priskoro, len tým dolnému tímu uškodíme.)

Na základe práve dokázaného tvrdenia ľahko navrhneme riešenie s časovou zložitosťou $O((n-k)\log(n-k))$. V tomto riešení postupne vyskúšame všetky možnosti pre to, koľko najstarších hráčov postupne odoberieme. (Počas tohto skúšania nezabudneme kontrolovať, či sú všetky stavy, cez ktoré ideme počas odoberania najstarších hráčov, živé.)

Keď už máme pevne zvolený počet p odobratých najstarších hráčov, máme vlastne pevne zvolenú k-ticu hráčov, ktorí budú stáť na vrchu kopca počas toho ako my postupne odoberáme najmladších hráčov. Najmladších hráčov však nebudeme odoberať po jednom. Namiesto toho použijeme efektívnejšiu metódu: binárnym vyhľadávaním na intervale od 0 po $n-k-p$ nájdeme najväčšie x také, že po odobratí p najstarších a následne x najmladších hráčov ešte stále budeme mať živý stav.

Listing programu (Python)

```
# načítanie a predspracovanie vyzerá rovnako ako v predchádzajúcom riešení

def je_zivy(mladych,starych):
    sucet_hore = P[N-starych] - P[N-starych-K]
    sucet_dole = P[N-starych-K] - P[mladych]
    return sucet_hore <= Q*sucet_dole + 1e-9

# ošetríme špeciálny prípad keď turnaj končí prvým zápasom
if not je_zivy(0,0):
    print(1)
    from sys import exit
    exit()

odpoved = 0
for starych in range(0,N-K+1):
    if not je_zivy(0,starych): break # toľkoto ani viac starých hráčov už nemôžeme odobrať
    lo, hi = 0, N-K-starych
    # binárne vyhľadávame -- invariant: lo mladých ešte môžeme odobrať, hi už nie
    while hi-lo > 1:
        med = (lo+hi)//2
        if je_zivy(med,starych): lo = med
        else:                    hi = med
    odpoved = max( odpoved, lo+starych+2 )

print(odpoved)
```

Nesprávna úvaha

V tomto okamihu je veľmi ľahké nechať sa zlákať nasledujúcou nesprávnou úvahou: Začneme tým, že nájdeme optimálny počet najmladších pre 0 najstarších. Teraz budeme postupne, rovnako ako v predchádzajúcom riešení, počet najstarších postupne zväčšovať, a vždy, keď je to nutné, počet najmladších postupne zmenšovať. Takto dostaneme riešenie s lineárnou časovou zložitosťou.

Vyššie popísané riešenie síce má lineárnu časovú zložitosť, ale úvaha, ktorá k nemu vedie, je nesprávna. V nej totiž implicitne predpokladáme, že väčšiemu počtu odstránených starých hráčov musí nutne zodpovedať menší alebo rovný počet odstránených mladých hráčov. No a toto vôbec nie je pravda.

Ukážeme si to na konkrétnom príklade. Majme $q = 1.01$ (teda skoro rovinu) a zoberme napr. $n = 10$ ľudí, pričom $k = 3$ najstarší sú vždy na vrchu kopca. Sily (od najmladšieho) nech sú $(47, 1, 1, 1, 1, 1, 1, 1, 3, 1)$.

- Ak odíde 0 najstarších, vedia potom odísť 2 najmladší. Posledný živý stav má $1+1+1+1+1$ na spodku a $1+3+1$ na vrchu kopca.
- Ak odíde 1 najstarší, vie potom odísť **len 1** najmladší. Posledný živý stav má $1+1+1+1+1$ na spodku a $1+1+3$ na vrchu kopca.
- Ak ale odídu 2 najstarší, vedia potom odísť **znova až 2** najmladší. Posledný živý stav má $1+1+1$ na spodku a tiež $1+1+1$ na vrchu kopca.

Optimálne riešenie

Ako opraviť úvahu z predchádzajúcej časti? Pôjdeme na to od konca. Začneme tým, že si zistíme, koľko najviac najstarších hráčov môžeme prípustným spôsobom odobrať. Následne sa pokúsime spraviť v podstate to isté ako v predchádzajúcom riešení: budeme postupne *zmenšovať* počet odobratých starých hráčov a zakaždým sa budeme snažiť *zväčšovať* počet odobratých mladých.

Ako sme ale videli vo vyššie uvedenom protipríklade, môžu nastať situácie, v ktorých by sme mali (na udržanie prípustnosti riešenia) počet odobratých mladých hráčov zmenšiť. Čo s takýmito situáciami? Jednoducho ich odignorujeme a počet mladých hráčov nezmenšíme. Takáto situácia totiž zjavne nemôže predstavovať optimálne riešenie – aj starých aj mladých hráčov by sme odobrali menej ako v inom riešení ktoré už poznáme.

Takto dostávame korektné riešenie s časovou zložitosťou $O(n)$.

Listing programu (Python)

```
# načítanie, predspracovanie, je_zivy() a ošetrenie špeciálneho prípadu sú rovnaké ako doteraz

mladych, starych = 0, 0
while je_zivy(mladych,starych+1): starych += 1
while je_zivy(mladych+1,starych): mladych += 1
odpoved = mladych+starych+2

while starych > 0:
    starych -= 1
    if not je_zivy(mladych,starych): continue # preskakujeme zjavne neoptimálnu možnosť
    while je_zivy(mladych+1,starych): mladych += 1
    odpoved = max( odpoved, mladych+starych+2 )

print(odpoved)
```

A-III-3 Mimozemské počítače

V prvej podúlohe stačila drobná úprava zadaného grafu. V druhej sa dalo postupne po jednej odstraňovať hrany a overovať, či sme tak neodstránili hľadanú cestu. Efektívnejšie riešenie však vie vhodne odstrániť viac hrán naraz. V poslednej, tretej podúlohe bolo treba každý vrchol pôvodného grafu nahradiť viacerými vrcholmi v novom grafe.

Podúloha A (1 bod)

V tejto podúlohe sme smeli raz zavolať funkciu `cesta_s_hranou(n,E,u,v)` a pomocou tohto volania sme chceli o našom grafe G zistiť, či obsahuje Hamiltonovskú cestu.

Ak by sme funkciu `cesta_s_hranou` mohli volať viackrát, bolo by riešenie jednoduché: stačilo by túto funkciu postupne zavolať toľkokrát, koľko hrán má náš graf G, pričom ako u a v mu vždy dáme vrcholy spojené jednou z hrán. Alebo by stačilo zvoliť $u = 0$ a ako v postupne vyskúšať všetky vrcholy, ktoré sú s 0 spojené. My však smieme funkciu `cesta_s_hranou` volať len raz. Ako na to?

Graf G má vrcholy s číslami 0 až $n-1$. My doň pridáme ešte dva vrcholy: n a $n+1$. Tieto spojíme medzi sebou, a navyše vrchol n spojíme s každým z vrcholov 0 až $n-1$.

Nech `E` je zoznam hrán takto upraveného grafu. Potom my zavoláme funkciu `cesta_s_hranou(n+2,E,n,n+1)`. Teda zistíme, či náš upravený graf obsahuje Hamiltonovskú cestu, na ktorej je hrana medzi vrcholmi n a $n+1$.

Správnosť algoritmu je zjavná, stačí si všimnúť, že v novom grafe *každá* Hamiltonovská cesta musí obsahovať hranu medzi n a $n+1$, a že nový graf má nejakú Hamiltonovskú cestu práve vtedy, ak mal nejakú Hamiltonovskú cestu pôvodný graf.

Podúloha B (5 bodov)

Pomalšie riešenie. Postupne prejdeme všetky hrany grafu G. Pre každú hranu zopakujeme nasledovný proces: Odoberieme ju z aktuálneho grafu, a následne volaním funkcie `cesta` overíme, či v grafe ešte ostala nejaká Hamiltonovská cesta. Ak ostala, práve spracúvanú hranu necháme odstránenú. Ak neostala, hranu vrátime späť.

Na konci tohto algoritmu nám zjavne musí ostať práve jediná Hamiltonovská cesta. (Je zjavné, že nám nejaká ostane, lebo po každom kroku máme graf, v ktorom aspoň jedna cesta existuje. Tiež je zjavné, že tam okrem nej už nemôžu byť žiadne iné hrany – každú takú hranu sme niekedy spracúvali, a keďže v danom okamihu existovala Hamiltonovská cesta na ktorej daná hrana neležala, nutne sme ju vyhodili.)

Tento algoritmus potrebuje presne m volaní funkcie `cesta`. Pre husté grafy je m rádovo rovné n^2.

Lepšie riešenie. Ukážeme si teraz riešenie, ktoré si vystačí s $O(n \log n)$ volaniami funkcie `cesta`. Existuje viacero podobne efektívnych riešení, my sme si vybrali jedno, ktoré sa ľahko implementuje. Hľadanú Hamiltonovskú cestu budeme zostrojovať postupne, vrchol za vrcholom.

Na začiatok by sme potrebovali vedieť, kde naša cesta začína. Toto vyriešime napríklad tak, že si graf upravíme rovnako ako v riešení podúlohy A. Teraz teda vieme, že prvé dva vrcholy na hľadanej Hamiltonovskej ceste sú vrcholy $n+1$ a n.

Nech x je posledný vrchol na tej časti cesty, ktorú sme už zostrojili. Ako nájsť ďalší jej vrchol? Z vrcholu x vedú v našom grafe G nejaké hrany. Jedna z nich je tá, ktorou do x príde nami zostrojovaná cesta. Ostatné vedú do nových, ešte nenavštívených vrcholov. (Toto platí na začiatku, keď $x = n$. Následne budeme priebežne odstraňovať nepotrebné hrany z G, takže to bude platiť aj v ďalších iteráciách.) No a my sa teraz potrebujeme rozhodnúť, ktorou z týchto hrán bude naša cesta pokračovať.

Toto by sme vedeli spraviť sekvenčne, podobne ako v predchádzajúcom riešení. Existuje ale aj efektívnejší spôsob, podobný binárnemu vyhľadávaniu. Kým budeme mať na výber viac ako jednu hranu, budeme opakovať nasledovný postup: Z G odstránime polovicu spomedzi kandidátov na nasledujúcu hranu a zavoláme funkciu `cesta`. Ak graf stále obsahuje nejakú Hamiltonovskú cestu, pokračujeme priamo ďalej. Ak nám ale volanie funkcie `cesta` oznámi, že nový graf už Hamiltonovskú cestu nemá, znamená to, že (každá možná) hľadaná hrana z x ďalej je medzi tými, ktoré sme práve odstránili. Vrátime ich teda späť do G – a *namiesto nich* odstránime tú polovicu hrán z x, ktorú sme pôvodne v G nechali!

Takto pre konkrétny vrchol x jedným volaním funkcie `cesta` zmenšíme počet hrán vedúcich z x približne na polovicu, pričom naďalej dostaneme graf obsahujúci aspoň jednu Hamiltonovskú cestu. Keď tento postup opakujeme, po $O(\log n)$ volaniach funkcie `cesta` nám ostane vo vrchole x (okrem hrany, ktorou sme doň prišli) už len jediná hrana – a teda sme práve našli nasledujúcu hranu na zostrojovanej Hamiltonovskej ceste.

Listing programu (Python)

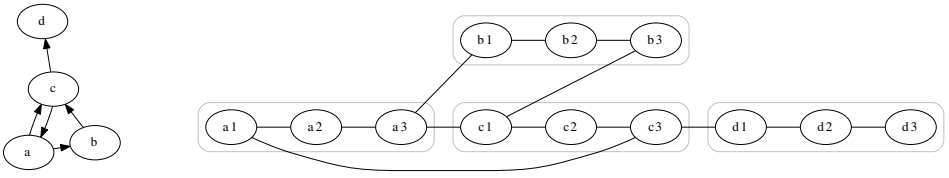
```
def susedia(v,E):
    s1 = set( y for x,y in E if x==v )
    s2 = set( x for x,y in E if y==v )
    return [ z for z in s1+s2 ]

def najdi_cestu(n,E):
    E += [ (n,x) for x in range(n) ] + [ (n,n+1) ]
    cesta = [ n+1, n ]
    for kolo in range(n):
        kde = cesta[-1]
        kam_mozem = susedia( kde, E )
        kam_mozem.remove( cesta[-2] )
        ostatne_hrany = [ x,y for x,y in E if x!=kde and y!=kde ] + [ (kde,cesta[-2]) ]
        while len(kam_mozem) > 1:
            cnt = len(kam_mozem)
            k1, k2 = kam_mozem[:cnt//2], kam_mozem[cnt//2:]
            if cesta( n+2, ostatne_hrany + [ (kde,x) for x in k1 ] ):
                kam_mozem = k1
            else:
                kam_mozem = k2
        vitaz = kam_mozem[0]
        cesta.append( vitaz )
        E = ostatne_hrany + [ (kde,vitaz) ]
    return cesta[2:]
```

Podúloha C (4 body)

Z pôvodného orientovaného grafu G s n vrcholmi spravíme nový neorientovaný graf s $3n$ vrcholmi – a to tak, že každý pôvodný vrchol nahradíme tromi novými.

Namiesto každého vrcholu v teda budeme mať tri vrcholy: v_1 (tzv. vstupný), v_2 (tzv. stredný) a v_3 (tzv. výstupný). Dvojice v_1v_2 a v_2v_3 budú spojené hranou.

Orientované hrany z pôvodného grafu zmeníme v novom grafe na neorienované hrany z príslušného výstupného do príslušného vstupného vrcholu. Ak sme teda napr. v pôvodnom grafe mali hranu $u \to v$, budeme v novom grafe mať neorientovanú hranu u_3v_1.



Vľavo: orientovaný graf. Vpravo: k nemu zostrojený neorientovaný graf.

Zjavne ak v pôvodnom grafe existovala orientovaná Hamiltonovská cesta, existuje Hamiltonovská cesta aj v našom novom neorientovanom grafe – vždy, keď sme v pôvodnom grafe vošli do vrcholu v, vôjdeme v novom grafe postupne do vrcholov v_1, v_2 a v_3.

Ako je to s opačnou implikáciou? Predpokladajme, že v našom novom grafe existuje Hamiltonovská cesta. Čo o nej vieme povedať? V prvom rade to, že musí ísť cez všetkých n stredných vrcholov. A keďže každý v_2 je spojený len s dvomi inými vrcholmi, vieme, že sa táto Hamiltonovská cesta musí skladať z n 3-vrcholových úsekov tvaru $v_1v_2v_3$. Ďalej, v našom grafe nemáme žiadnu hranu tvaru u_1v_1 ani u_3v_3, preto sa na našej Hamiltonovskej ceste musí dokola opakovať vstupný, stredný a výstupný vrchol. No a takejto Hamiltonovskej ceste zjavne zodpovedá orientovaná Hamiltonovská cesta v pôvodnom grafe.

(Rozmyslite si, kde by dôkaz tejto implikácie zlyhal, ak by sme spravili konštrukciu s $2n$ vrcholmi, pri ktorej vynecháme stredné vrcholy a namiesto toho spojíme hranou priamo každý vstupný vrchol so zodpovedajúcim výstupným.)

DVADSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Vladimír Boža, Michal Forišek, Peter Fulla
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2014