B-II-1 Byrokracia

Ministerstvo zo zadania si vieme reprezentovať ako graf. Kancelárie budú *vrcholy* a prechody medzi nimi voláme *(orientované) hrany*, pričom vrchol a_i (alebo `a[i]`) je nasledovníkom vrchola i.
Pripomenieme, že cyklom voláme takú postupnosť $k \geq 1$ vrcholov $v_1, v_2 \dots v_k$, pre ktorú platí, že $a_{v_1} = v_2, a_{v_2} = v_3 \dots a_{v_k} = v_1$.

Ak návštevník ministerstva začne cestu vo vrchole, ktorý je na cykle, kým sa zopakuje nejaký vrchol, spraví presne toľko krokov, aká je dĺžka tohto cyklu.
Pre vrcholy, ktoré nie sú na cykle, ale ich nasledujúci vrchol na cykle je, vieme odpoveď tiež. Je to dĺžka cyklu plus jedna.
Vo všeobecnosti, ak x je vrchol mimo cyklu, $a[x]$ je nasledovník vrchola x, a $d[y]$ je riešenie úlohy pre vrchol y, tak platí:

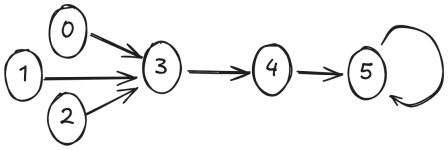
$$d[x] = d[a[x]] + 1$$

Teda, riešenie pre vrchol x je o 1 väčšie ako riešenie pre vrchol, do ktorého vedie hrana z vrchola x.

Tento typ myšlienok, *„najprv spočítam `d[a[x]]`, až potom na základe toho spočítam `d[x]`“*, sa obvykle dobre programuje pomocou rekurzie (rekurzívnej funkcie).

```
def calculate(x):
    if x je na cykle:
        return dlzka_cyklu[x]
    else:
        return calculate(a[x]) + 1
```

Typickým problémom rekurzií je, že sa nám môže stať, že rovnaké veci počítame zbytočne veľa krát. Napríklad, pre vstup `a[0] = a[1] = a[2] = 3`, a tiež `a[3] = 4`, `a[4] = 5`, `a[5] = 5`,



by sme postupne mohli volať:

- `calculate(0)` → `calculate(3)` → `calculate(4)` → `calculate(5)` → vrchol 5 je na cykle
- `calculate(1)` → `calculate(3)` → `calculate(4)` → `calculate(5)` → vrchol 5 je na cykle
- `calculate(2)` → `calculate(3)` → `calculate(4)` → `calculate(5)` → vrchol 5 je na cykle

Funkciu `calculate()` s argumentami 3, 4 a 5 voláme opakovane, jej hodnota je však zakaždým taká istá. Pri týchto volaniach preto robíme zbytočnú prácu navyše. Rozumnejšie by bolo zapamätať si odpoveď a volať funkciu pre každé x len raz.

- `calculate(0)` → `calculate(3)` → `calculate(4)` → `calculate(5)` → vrchol 5 je na cykle
- `calculate(1)` → odpoveď pre vrchol 3 už poznáme
- `calculate(2)` → odpoveď pre vrchol 3 už poznáme

Náš program by sme upravili nasledovne pridaním poľa `memo`.

```
memo = {}
def calculate(x):
    # ak uz pozname odpoved pre x, usetrime si cas
    if x in memo:
        return memo[x]
```

```
        # v opacnom pripade vysledok vypocitame a ulozime ho do memo[x]
        if x je na cykle:
            memo[x] = dlzka_cyklu[x]
        else:
            memo[x] = calculate(a[x]) + 1
        return memo[x]
```

Ostáva nám ešte zistiť, či je vrchol na cykle a akú dĺžku daný cyklus má.
Nemysliac na časovú zložitosť, ak chceme nájsť všetky cykly, mohli by sme začať postupne v každom vrchole, a pustiť z neho simuláciu pohybu po ministerstve. Akonáhle nájdeme vrchol, ktorý sme už navštívili v danej simulácii, vieme, že sme na cykle.
Následne spravíme ešte niekoľko krokov, kým znova nenavštívime daný vrchol, čím efektívne prejdeme celý cyklus ešte raz a zároveň odmeriame jeho dĺžku, pretože počas tejto prechádzky sme spravili toľko krokov, aká je dĺžka cyklu. Nakoniec prejdeme cyklus ešte posledný krát, aby sme si pre vrcholy na ňom zapísali zistenú dĺžku cyklu.
Pre efektívnu časovú zložitosť je dôležité púšťať spomínanú prechádzku po cykle len raz pre každý cyklus na ministerstve. To spravíme tak, že si budeme pamätať, pre ktoré vrcholy sme už zistili, že sú na cykle a nebudeme púšťať hľadanie zbytočne druhýkrát.
Celé sa to dá implementačne spojiť dokopy tak, že každý vrchol začne v stave „nenavštívený“, pri prvom volaní funkcie `calculate(x)` prejde do stavu „navštívený prvýkrát“, a potom buď zistíme, že sa vrchol nachádza na cykle (a zapamätáme si dĺžku cyklu), alebo zistíme, že sa na cykle nenachádza a spočítame odpoveď ako `calculate(a[x]) + 1`.
Časová zložitosť je lineárna od počtu kancelárií, teda $O(n)$, pretože pre každý vrchol najviac dvakrát zavoláme funkciu `calculate(x)` a funkcia `mark_loop(x)` prejde každým vrcholom tiež najviac dvakrát.

Listing programu (Python)

```
import sys

sys.setrecursionlimit(4_000_047)
n = int(input())
a = [int(x) - 1 for x in input().split()]
# -2: nenavstiveny
# -1: navstiveny, ale nevieme ci je na cykle
# >0: odpoved pre dany vrchol
memo = [-2 for _ in range(n)]


def calculate(x):
    if memo[x] > 0:
        return memo[x]

    if memo[x] == -1:
        # ak uz sme v tomto vrchole boli, pomocou osobitnej funkcie
        # nastavime pre vsetky vrcholy na cykle: memo[x] = <dlzka_cyklu>
        mark_loop(x)
        return memo[x]

    # toto nam pomoze najst opakujuci sa vrchol
    memo[x] = -1

    # ak vrchol nie je na cykle, tak `res` je odpoved
    # mozno ale vnutri rekurzie zistime, ze vrchol je na cykle
    # preto aktualizujeme info, iba ak memo[x] < 0
    res = calculate(a[x]) + 1
    if memo[x] < 0:
        memo[x] = res
    return memo[x]

def mark_loop(start):
    length = 1
    current = a[start]
    while current != start:
        current = a[current]
        length += 1

    current = start
    for _ in range(length):
        current = a[current]
        memo[current] = length


for i in range(n):
    calculate(i)
print("␣".join(map(str, memo)))
```

B-II-2 Asfaltovanie

Riešenie hrubou silou

Ak všetky pokusy o nájdenie efektívneho riešenia úlohy zlyhajú, oplatí sa zamyslieť nad riešením hrubou silou. V tomto prípade je najpriamočiarejšia možnosť pre daný počet úsekov a firiem vygenerovať všetky možnosti asfaltovania vyhovujúce reguláciam. Ak ani jedna možnosť nevedie k asfaltovaniu na vstupe, máme istotu, že asfaltovanie neprebehlo podľa regulácií.
Aby sme overili asfaltovanie na vstupe, musíme v najhoršom prípade vyskúšať všetky možné poradia, v akých mohli firmy asfaltovať. Takýchto poradí je práve $k \cdot (k-1) \cdot (k-2) \ldots 2 \cdot 1$, teda $k!$ (k faktoriál). Pre jedno konkrétne poradie však stále existuje mnoho rôznych koncových výsledkov vyasfaltovania. Poradie v akom firmy asfaltovali totiž neurčuje výsledné vyasflatovanie. Pri každej firme záleží, na ktorom úseku asfaltovanie začalo a skončilo a týchto možností máme pre jednu firmu $\frac{n(n+1)}{2}$, teda $O(n^2)$. Našťastie, nie je potrebné skúšať všetky tieto asfaltovania.

Nech máme na vstupe nasledovné asfaltovanie:

```
1 1 2 3 2 2 4 4 1
```

Ktoré úseky mohla asfaltovať firma číslo 2? Zo vstupu vidíme, že firma 2 určite asfaltovala tretí, piaty a šiesty úsek. To ale znamená, že táto firma musela začať asfaltovať *na treťom úseku alebo skôr* a skončiť *na šiestom úseku alebo neskôr*. Môžeme si však uvedomiť, že skúšať asfaltovanie, ktoré začína pred úsekom 3 alebo končí po úseku 6 je zbytočné. Keďže na dosiahnutie asfaltovania na vstupe nemohla táto firma asfaltovať úseky pred 3 a po 6 ako posledná, aj keby ich vyasfaltovala, bol by jej asfalt prekrytý. Ak by teda firma 2 v nejakom správnom asfaltovala viac úsekov, prejavilo by sa to rovnako, ako keby asfaltovala len úseky od 3 po 6.

Pre zjednodušenie si vezmime jednu konkrétnu firmu a všetky úseky, na ktorých asfaltovala posledná. Prvý takýto úsek budeme nazývať *začiatočný úsek* firmy a posledný takýto úsek bude *koncový úsek* firmy. Týmto dvom a všetkým úsekom medzi nimi budeme hovoriť *oblasť* firmy.
Stačí teda, aby naše riešenie postupne prechádzalo všetkých $k!$ možných poradí v akých mohli firmy asfaltovať. Pre každé takéto poradie si odsimulujeme, k akému vyasfaltovaniu by viedlo ak by každá firma asfaltovala iba svoju oblasť, teda od svojho začiatočného po koncový úsek. Hodnoty začiatočných a koncových úsekov si predpočítame dopredu.
Pomerne jednoduchá implementácia tohto prístupu v jazyku Python:

Listing programu (Python)

```
import itertools

n, k = [int(x) for x in input().split()]
useky = [int(x) for x in input().split()]

zac = (k+1) * [None]
kon = (k+1) * [None]

for i in range(n):
    posledna_firma = useky[i]
    zac[posledna_firma] = zac[posledna_firma] if zac[posledna_firma] != None else i
    kon[posledna_firma] = i

for poradie in itertools.permutations(list(range(1, k+1))):
    oasfaltovanie = n * [None]
    for firma in poradie:
        if zac[firma] == None:
            continue
        for i in range(zac[firma], kon[firma]+1):
            oasfaltovanie[i] = firma
    if oasfaltovanie == useky:
        print("ANO")
        exit(0)
print("NIE")
```

Časová zložitosť takto implementovaného riešenia je $O(n \cdot k \cdot k!)$. Postupne prechádzame všetky permutácie poradia firiem. Pre každé poradie sa potom pokúsime zostrojiť asfaltovanie na vstupe. Pamäťová zložitosť riešenia je $O(n+k)$. Napriek relatívnej neefektivite takéto riešenie stačí na zisk 3 bodov.

Dávame ešte do pozornosti premennú k vystúpujúcu v odhadoch zložitostí. Napriek tomu, že túto premennú vieme zhora ohraničiť n, jej uvedenie nám umožňuje uviesť *tesnejší* horný odhad.

Riešenie odzadu

Cesta k efektívnejšiemu riešeniu vedie cez zamyslenie sa nad tým, ako proces asfaltovania prebiehal. V každom momente (až na úplny začiatok) už nejaký asfalt položený bol a firma novým asfaltom prekryla nejakú súvislú časť. Vieme tento proces spätne odsimulovať? Vieme pre príklad zo zadania povedať, ktorá firma pokladala asfalt ako posledná?

```
9 5
1 1 4 4 2 2 5 4 1
```

V tomto prípade určite nemohla posledná asfaltovať firma 1. Tá totiž musela vyasfaltovať všetko od začiatočného úseku 1 po koncový úsek 9, úseky 3 až 8 sú však pokryté iným asfaltom, ktorý musel byť nanesený neskôr. Z rovnakého dôvodu nemôže byť poslednou firmou firma číslo 4, a keďže firma 3 na vstupe ani nie je, bola celá preasfaltovaná a určite nebola posledná v poradí. Jedinými možnosťami sú teda firmy 2 a 5, pretože oblasti týchto dvoch firiem nie sú prekryté asfaltom žiadnej inej firmy.
Táto úvaha nám však dáva návod na ďalšie riešenie. Ak vieme, ktorá firma pokladala asfalt ako posledná, môžeme ho odstrániť a tým sa akoby vrátiť v čase.
Situácia pred asfaltovaniami firiem 2 a 5 vyzerala nasledovne:

```
1 1 4 4 ? ? ? 4 1
```

Vieme v tejto novovzniknutej situácií určiť, ktorá firma asfaltovala posledná? Je zjavné, že poslednou asfaltujúcou firmou je firma 4. K určeniu tejto firmy sme použili rovnaké argumenty ako v predchádajúcej situácii.
Riešením je teda iteratívne odstraňovať asfalt poslednej asfaltujúcej firmy. Ak týmto procesom dospejeme k pôvodnej ceste, vieme, že vyasfaltovanie na vstupe prebehlo podľa regulácií. Proces asfaltovania podľa regulácií sme totiž práve odzadu odsimulovali.

Ako pre ľubovoľné vyasflatovanie určiť poslednú asfaltujúcu firmu? Vieme, že to je firma, ktorej asfalt bol položený posledný a tak oblasť tohto asfaltu nemôže obsahovať asfalt žiadnej inej firmy.
Trik, ktorým si zjednodušíme implementáciu a nezhoršíme časovú zložitosť, je si pred každým vyhľadávaním poslednej asfaltujúcej firmy *skomprimovať* súvislé úseky do jedného. To znamená, že opakované výskyty tej istej firmy na niekoľkých úsekoch po sebe budeme reprezentovať jedným úsekom.
Príjemnou vlastnosťou takejto reprezentácie je, že oblasť každej firmy, ktorá neobsahuje asfalt inej firmy bude pri takomto spracovaní reprezentovaná *práve jedným úsekom*. Medzi začiatočným a koncovým úsekom tejto firmy sa totiž mohli nachádzať iba úseky patriace tejto firme.
Keď nájdeme firmu, ktorá asfaltovala ako posledná, jej úsek z našej reprezentácie odstránime a zopakujeme komprimáciu.
Pri riešení si treba uvedomiť ešte dve veci. Čísla niektorých firiem sa na vstupe neobjavujú vôbec, lebo ich oblasti boli celé preasfaltované. Takéto firmy však môžeme ignorovať, keďže ľahko im vieme doplniť asfaltovanie spĺňajúce regulácie, napr. tieto firmy asfaltovali ako prvé ľubovoľné úseky.
Taktiež sa v niektorých momentoch môže stať, že viacero firiem je kandidátom na poslednú asfaltujúcu firmu, ako je tomu aj v príklade vyššie, keď to boli firmy 2 a 5. Tieto firmy sú však od seba nezávislé, keďže ich oblasti sa z definície nemôžu prekrývať. Je preto jedno v akom poradí ich spracujeme.

Listing programu (Python)

```
def skomprimuj(useky: list[int]) -> list[int]:
    bez_duplikatov = [useky[0]]
    for usek in useky:
     if usek != bez_duplikatov[-1]:
         bez_duplikatov.append(usek)
    return bez_duplikatov

def odstran_poslednu_firmu(useky: list[int], k: int) -> list[int]:
```

```
  if not useky:
     return useky
  bez_duplikatov = skomprimuj(useky)
  pocet_vyskytov = (k+1) * [0]
  for firma in bez_duplikatov:
     pocet_vyskytov[firma] += 1
  return [usek for usek in bez_duplikatov if pocet_vyskytov[usek] != 1]

n, k = [int(x) for x in input().split()]
useky = [int(x) for x in input().split()]

for _ in range(k):
   useky = odstran_poslednu_firmu(useky, k)
   if not useky:
     print("ANO")
     exit(0)

print("NIE")
```

Toto riešenie má časovú zložitosť $O(kn)$, kedže nanajvýš k-krát odstránime asfalt poslednej asfaltujúcej firmy. Jedno takéto odstránenie sa skladá zo:

- Skomprimovania všetkých súvislých úsekov asfaltovaných rovnakou firmou v čase $O(n)$.
- Identifikovania, ktoré súvislé oblasti rovnakého asfaltu neobsahujú žiadny iný asfalt (nachádzajú sa v skomprimovanej reprezentácii práve raz) taktiež v čase $O(n)$.

Pamäťová zložitosť je $O(n + k)$.

Práve dva úseky každej firmy

Poďme sa teraz pozrieť na riešenie podúlohy, v ktorej sa číslo každej firmy na vstupe vyskytuje práve dvakrát, čím sú presne definované oblasti každej firmy. Spôsob akým môžeme pristupovať k riešeniu je zamyslieť sa nad rôznymi možnosťami prekryvu oblastí dvoch rôznych firiem. Tieto možnosti sú:

- Dve oblasti sa neprekrývajú.
- Dve oblasti sa prekrývajú čiastočne.
- Dve oblasti sa prekrývajú úplne, teda jedna je kompletne obsiahnutá v druhej.

V nasledovnom vyasfaltovaní:

```
1 1 2 3 3 4 2 4
```

sa oblasti firmy 1 a 3 neprekrývajú, oblasť firmy 2 a 4 sa prekrývajú čiastočne a oblasť firmy 3 leží celá v oblasti asfaltovanej firmou 2.
Ak sa oblasti neprekrývajú, môžeme ich riešiť nezávisle od seba. Ak majú úplný prekryv, tak je zjavné, že najprv musela byť asfaltovaná väčšia oblasť, a až potom tá menšia, ktorá je v nej obsiahnutá.
Problematické je ale čiastočné prekrytie dvoch oblastí, kedy poradie firiem spĺňajúce regulácie *neexistuje*. Nech totiž ako prvú vyasfaltujeme ľubovoľnú z nich, pri asfaltovaní druhej prekryjeme jeden z krajných úsekov prvej oblasti. Ak by sme teda vedeli identifikovať takéto situácie, vedeli by sme, kedy pre vstup neexistuje žiadne asfaltovanie vyhovujúce reguláciám.
Pre upresnenie, čiastočný prekryv dvoch oblastí nastáva vtedy, keď sa v oblasti nejakej firmy nachádza začiatočný, ale nie koncový úsek nejakej inej firmy.
Naše riešenie bude prechádzať vstup zľava doprava (od prvého po posledný úsek). V každom momente si budeme uchovávať zoznam tzv. *aktívnych firiem*. To sú firmy, na ktorých začiatočný úsek sme už narazili, avšak sme zatiaľ nenarazili na ich koncový úsek.
Keď nájdeme začiatočný úsek nejakej firmy, pridáme si ju do zoznamu aktívnych firiem. Keď narazíme na koncový úsek nejakej firmy f_{kon}, prirodzene ju zo zoznamu odstránime. Predtým ale potrebujeme zistiť, či neexistuje firma f, ktorej začiatočný úsek prišiel niekedy po začiatočnom úseku firmy f_{kon}, ale je stále medzi aktívnymi firmami. To by znamenalo, že firmy f_{kon} a f sa prekrývajú. Nato nám stačí zistiť, či existuje aktívna firmy, ktoré sa stala aktívnou po firme f_{kon}.

Aby sme toto vedeli robiť efektívne, budeme si držať zoznam aktívnych firiem zoradený podľa poradia, v ktorom sa stali aktívnymi. Keď narazíme na začiatočný úsek nejakej firmy, firmu pridáme na koniec tohto zoznamu a naopak, pri spracovávaní koncového úseku nejakej firmy budeme túto firmu hľadať na konci zoznamu. Ak sa tam nenachádza, znamená to, že nejaká iná firma je aktívna a bola pridaná do zoznamu neskôr. Z toho vyplýva, že sme objavili dve čiastočne sa prekrývajúce firmy a riešenie neexistuje.

Pre úplnosť musíme ešte zdôvodniť, že ak sa žiadne čiastočné prekrytie na vstupe nenachádza, tak existuje asfaltovanie vyhovujúce reguláciám. To sa však dá dokázať využitím riešenia odzadu. Ak sa totiž oblasti firiem prekrývajú vždy buď úplne alebo vôbec, musí vždy existovať firma, ktorej oblasť neobsahuje oblasť žiadnej inej firmy. To ale znamená, že začiatočný a koncový úsek tejto firmy leží vedľa seba. Môžeme si teda povedať, že táto firma asfaltovala ako posledná a vymazať jej patriace úseky.

Časová aj pamäťová zložitosť takéhoto riešenia je $O(n + k)$.

Listing programu (Python)

```
n, k = [int(x) for x in input().split()]
useky = [int(x) for x in input().split()]

zac = (k+1) * [None]
kon = (k+1) * [None]

for i in range(n):
  posledna_firma = useky[i]
  zac[posledna_firma] = zac[posledna_firma] if zac[posledna_firma] != None else i
  kon[posledna_firma] = i

aktivne = []
for i, firma in enumerate(useky):
  if zac[firma] == i: # zaciatocny usek
    aktivne.append(firma)
  if aktivne[-1] != firma:
    print("NIE")
    exit(0)
  if kon[firma] == i: # koncovy usek
    aktivne.pop()
print("ANO")
```

Vzorové riešenie

Vo vzorovom riešení si treba predefinovať, čo znamená prekrývanie oblastí. Každá oblasť konkrétnej firmy obsahuje niekoľko úsekov, ktoré vo výslednom asfaltovaní patria práve tejto firme. Tieto úseky môžeme volať *dôležité úseky* firmy. Špecificky patrí medzi dôležité úseky aj začiatočný a koncový úsek oblasti.
Prekryv oblastí následne definujeme:

- Dve oblasti sa neprekrývajú, ak nemajú žiaden spoločný úsek.
- Oblasť firmy x úplne prekrýva oblasť firmy y vtedy, ak je oblasť y podmnožinou oblasti x **a zároveň** žiaden dôležitý úsek x nepatrí do oblasti y.
- Oblasť firmy x sa čiastočne prekrýva s oblasťou y vtedy, keď existuje dôležitý úsek x, ktorý *leží* v oblasti y **a zároveň** existuje taký dôležitý úsek x, ktorý v oblasti y *neleží*.

Môžete si rozmyslieť, že ak má každá oblasť práve dva dôležité úseky – začiatočný a koncový – sú tieto definície totožné s tými predchádzajúcimi.

Na riešenie použijeme ten istý prístup ako predtým. Udržiavame si zoznam aktívnych firiem usporiadaný podľa času pridania. Postupne prechádzame cez vstup zľava doprava. Ak narazíme na začiatok oblasti, pridáme firmu medzi aktívne firmy. Ak narazíme na koniec oblasti, skontrolujeme, že táto firma ja na vrchu poradia aktívnych firiem, a potom ju odtiaľ odstránime.
A ak narazíme na výskyt firmy x mimo začiatku a konca (dôležitý úsek x), musíme si uvedomiť, že aj v tomto prípade musí byť na vrchu poradia aktívnych firiem práve x. Ak by na vrchu poradia bola firma y, znamenalo by to, že oblasť y začala až po oblasti x, obsahuje však tento dôležitý úsek x, čo vytvorí čiastočný prekryv.

Predošlá implementácia bola napísaná všeobecne a v skutočnosti nerieši iba špecifický prípad dvoch výskytov pre každú firmu, ale aj tento všeobecnejší. Môžete sa k nej vrátiť a rozmyslieť si, ako bude pracovať s dôležitými úsekmi.

B-II-3 Pečieme makrónky

Ako sme v zadaní naznačili, v tejto úlohe, podobne ako v domácom kole, ideme skúšať možnosti hrubou silou (s rôznymi úrovňami šikovnosti).[1]

Skúšame všetkých permutácií

Začneme tým, ako vôbec nejaké možnosti vygenerovať. Ak ste vyriešili tretiu úlohu z domáceho kola, mohlo vám napadnúť zobrať si postupnosť $a_1, a_2, \ldots, a_{2n}$ v ktorej je každé číslo od 1 po $2n$ práve raz.
Následne vieme popárovať prvú makrónku v postupnosti s druhou, tretiu so štvrtou, atď., teda dostaneme páry $(a_1, a_2), (a_3, a_4), \ldots, (a_{2n-1}, a_{2n})$.
Vieme si jednoducho overiť, že ak prejdeme cez všetky *permutácie*[2] čísel od 1 po $2n$, tak naozaj prejdeme cez všetky možnosti popárovania polmakrónok. Ak si zoberieme popárovanie $(b_1, b_2), (b_3, b_4), \ldots, (b_{2n-1}, b_{2n})$, tak postupnosť $b_1, b_2, \ldots, b_{2n-1}, b_{2n}$ je permutácia čísel 1 až $2n$.
Stačí nám teda prejsť cez všetky permutácie čísel od 1 po $2n$, pre každú permutáciu spárovať prvé dve, druhé dve, atď., polmakrónky, overiť, či takto vzniknú akceptovateľné makrónky a ak áno, aká je ich výsledná krása.
Na prechod cez všetky permutácie vieme použiť knižničné funkcie z domáceho kola, napr. `next_permutation` v C++, alebo `permutations()` v pythone.
Takýto algoritus vyskúša $(2n)!$ možností a vie získať tri body.

Listing programu (C++)

```
#include<bits/stdc++.h>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<vector<int> > k(2 * n, vector<int>(2 * n));
    for (int i = 0; i < 2 * n; i ++) {
        for(int j = 0; j < 2 * n; j ++)
            cin >> k[i][j];
    }

    // zacneme s P = {0,1,..., 2n - 1}
    // polmakronky cislujeme od 0
    vector<int> P;
    for (int i = 0; i < 2 * n; i ++) P.push_back(i);

    vector<pair<int, int> > makronky;
    int najkrajsie = -1;

    do {
        int krasa = 0;
        // spocitame krasu sparovania
        for (int i = 0; i < n; i ++) {
            if (k[P[2 * i]][P[2 * i + 1]] == -1) {
                krasa = -1;
                break;
            }
            else krasa += k[P[2 * i]][P[2 * i + 1]];
        }

        if (krasa < 0) continue;

        // skontrolujeme, ci toto je doteraz najlepsia permutacia
        if (krasa > najkrajsie) {
            makronky.clear();
            for (int i = 0; i < n; i ++)
                makronky.push_back({P[i * 2], P[i * 2 + 1]});
```

[1] Ak by vás zaujímalo prečítať si viac o polynomiálnom riešení, vyhľadajte Edmonsov algoritmus https://en.wikipedia.org/wiki/Blossom_algorithm

[2] Permutácia dĺžky $2n$ je postupnosť, v ktorej je každé číslo od 1 po $2n$ práve raz

```
            najkrajsie = krasa;
        }

    } while(next_permutation(P.begin(), P.end()));

    if (najkrajsie < 0) {
        cout << "neda sa\n";
    }
    else {
        cout << najkrajsie << endl;
        for (int i = 0; i < n; i ++)
            cout << makronky[i].first + 1 << " "
                 << makronky[i].second + 1 << endl;
    }
}
```

Neskúšame možnosti viackrát?

Keď rozmýšľame, ako toto riešenie zlepšiť, mali by sme sa zamyslieť nad tým, či možnosti neskúšame viackrát. Zoberme si nejaké spárovanie polmakrónok, napríklad $(1,2),(3,4)$ pre $n=2$. Toto jedno spárovanie dostaneme až z 8 permutácií: $1,2,3,4$, $1,2,4,3$, $2,1,3,4$, $2,1,4,3$, $3,4,1,2$, $3,4,2,1$, $4,3,1,2$ a $4,3,2,1$.
Ako teda riešenie zlepšiť? Ak ste riešili podobnú úlohu z domáceho kola (úlohu B-I-3), mohlo vám napadnúť zamerať sa na prvú makrónku. Musí byť spárovaná s nejakou inou makrónkou, a na poradí v rámci páru, a ani na poradí páru medzi ostatnými pármi v permutácii nezáleží.
Tvrdíme teda, že stačí vyskúšať permutácie kde je 1 na prvom mieste. Prečo? Zoberme si nejaké spárovanie polmakrónok, $(1,a_2),(a_3,a_4),\ldots,(a_{2n-1},a_{2n})$ (prvá polmakrónka je spojená s a_2-tou). Potom postupnosť $1,a_2,a_3,\ldots,a_{2n}$ je *permutácia*, ktorá tvorí naše páry.
Postačuje teda prechod cez všetky permutácie čísel od 2 po $2n$ (resp. cez všetky premutácie od 1 po $2n$ kde je 1 na prvom mieste), čo vieme opäť spraviť pomocou knižničných funkcií. Dokopy vyskúšame $(2n-1)!$ možností, za čo vieme získať 5 bodov.

Listing programu (Python)

```
from itertools import permutations

n = int(input())
k = [list(map(int, input().split())) for _ in range(2 * n)]

postupnost = [i for i in range(1, 2 * n)]

najlepsie = None

for permutacia in permutations(postupnost):
    # spočítame, či makronky daju takto sparovat
    krasa = k[0][permutacia[0]]
    if krasa < 0:
        continue
    for i in range(1, 2 * n - 1, 2):
        if k[permutacia[i]][permutacia[i + 1]] < 0:
            krasa = -1
            break
        krasa += k[permutacia[i]][permutacia[i + 1]]

    if krasa >= 0 and (najlepsie == None or najlepsie[0] < krasa):
        najlepsie = (krasa, permutacia)

if najlepsie == None:
    print("neda sa")
else:
    print(najlepsie[0],"\n1 {}".format(najlepsie[1][0] + 1))
    for i in range(1, 2 * n - 1, 2):
        print("{} {}".format(najlepsie[1][i] + 1, najlepsie[1][i + 1] + 1))
```

Opúšťame knižničné funkcie: každá možnosť práve raz

Poďme späť k úvahe s prvou polmakrónkou. Zjavne musí byť spárovaná s nejakou zo zvyšných $2n-1$ polmakrónok. Ak jej vyberieme pár, ostane nám $2n-2$ makrónok ktoré treba popárovať.
Tam ale vieme použiť rovnakú úvahu. Nespárovaná makrónka s najmenším číslom (buď číslo 2 alebo 3, ak prvá makrónka je $(1,2)$) musí byť spárovaná s niektorou zo zostávajúcich polmakrónok. Máme $2n-3$ možností s ktorou. Čím nám ostane $2n-4$ nespárovaných polmakrónok, z ktorých ako ďalšiu môžeme spárovať tú s najmenším číslom.

Robíme vlastne stále to isté, iba so zmenšujúcim sa počtom polmakrónok. To znie ako rekurzia!
Ako z toho teda spraviť algoritmus? Majme napríklad funkciu, `sparuj_polmakronky()` ktorá pre daných $2k$ polmakrónok vráti ich všetky možné spárovania.
Ak zavoláme `sparuj_polmakronky()` na presne dve polmakrónky, máme jediný možný pár. Ak funkciu zavoláme pre $2k \geq 4$ polmakrónok, vyberieme jednu z nich (napr. tú s najmenším číslom) a zvolíme, s ktorou zo zvyšných $2k-1$ polmakrónok ju spárovať.
Pre každý takýto pár (nazvyme ho (a, b)) si zoberieme pole polmakrónok bez a, b (ostane nám $2k-2$ polmakrónok), a *rekurzívne* zavoláme funkciu `sparuj_polmakronky()` na tieto ostávajúce polmakrónky. Dostaneme tak ich možné spárovania, ku ktorým stačí pridať pár (a, b), čím dostaneme všetky možné sprárovania daných $2k$ polmakrónok obsahujúcich pár (a, b).
Ak toto spravíme pre každú dvojicu (a, b) tak po zavolaní funkcie `sparuj_polmakronky([1,2,...,2n])`, dostaneme všetky možné spárovania polmakrónok. Potom nám už len stačí prejsť si cez každú možnosť a vybrať si tú najkrajšiu (resp. vyhodnotiť, že Kubík makrónky vyrobiť nevie).

Koľko možností skúšame?

Pozrime sa teraz na počet možností, ktorý takáto rekurzia prejde. Pri prvej polmakrónke máme na výber $2n-1$ polmakrónok. Pri ďalšej párovanej polmakrónke máme na výber z $2n-3$ polmakrónok. Pri ďalšej $2n-5$. Toto pokračuje, až kým napokon máme na výber iba 1. Formálne to vieme zapísať ako $\prod_{k=1}^{n}(2k-1) = (2n-1)\cdot(2n-3)\cdot\cdots\cdot 3\cdot 1$. Toto číslo je zároveň počet všetkých možných spárovaní, takže algoritmus prejde každým práve raz.

Tento odhad je síce veľmi presný, ale trochu nešikovný na zapisovanie. Aby sme si lepšie predstavili, koľko možností vlastne skúšame, môžeme spraviť hrubý horný odhad. Ak každý činiteľ v súčine $\prod_{k=1}^{n}(2k-1)$ zväčšíme o 1, dostaneme oveľa krajší súčin $\prod_{k=1}^{n} 2k$, čo vieme ľahko zapísať ako $2^n \cdot n!$. Počet možností teda môžeme v O-notácii zapísať ako $O(2^n \cdot n!)$, čo je výrazné zlepšenie oproti $O((2n)!)$.

Ide to ešte zlepšiť?

Ukázali sme si, že tento algoritmus skúša toľko možností, koľko je spárovaní, takže algoritmus, ktorý potrebuje vygenerovať a prezrieť všetky možné spárovania už *v najhoršom prípade* byť efektívnejší nevie.
Podobne ako sme ale videli aj na domácom kole, v niektorých vvstupoch vieme vylúčiť pomerne veľa možností, ak sa v nich nachádza pár, ktorý Kubík odmieta spojiť.
Funciu `sparuj_makronky()` vieme jednoducho upraviť, aby tieto možnosti neskúšala: keď vyberá s čím spárovať najmenšiu zostávajúcu polmakrónku, pozrie sa pred zavolaním rekurzie, či ide daný pár vôbec spojiť. Ak nie, nevolá už rekurziu na zostávajúce makrónky, iba pokračuje s ďalším možným párom.
Takto nám funkcia vráti iba spárovania, ktoré je Kubík ochotný urobiť – ak nevráti žiadne, znamená to, že pre danú množinu polmakrónok žiadne akceptovateľné spárovanie neexistuje.

Listing programu (Python)

```
n = int(input())
k = [list(map(int, input().split())) for _ in range(2 * n)]

def sparuj_polmakronky(polmakronky):
    if len(polmakronky) == 2:
        if k[polmakronky[0]][polmakronky[1]] < 0:
            return []
        return [[(polmakronky[0], polmakronky[1])]]
    moznosti = []
    for i in range(1, len(polmakronky)):
        if k[polmakronky[0]][polmakronky[i]] >= 0:
            bez_paru = polmakronky.copy()
            bez_paru.remove(polmakronky[0])
            bez_paru.remove(polmakronky[i])
            vysledok = sparuj_polmakronky(bez_paru)
            for moznost in vysledok:
                moznost.append((polmakronky[0], polmakronky[i]))
                moznosti.append(moznost)
    return moznosti

najlepsie = (-1, None)
moznosti = sparuj_polmakronky(list(range(0, 2 * n)))
```

```
for moznost in moznosti:
    krasa = 0
    for a, b in moznost:
        # sparuj polmakronky generuje iba sparovatelne pary
        krasa += k[a][b]
    if krasa > najlepsie[0]:
        najlepsie = (krasa, moznost)

if najlepsie[0] == -1:
    print ("neda sa")
else:
    print(najlepsie[0])
    for a, b in najlepsie[1]:
        print("{} {}".format(a + 1, b + 1))
```

B-II-4 Opravujúce sa svetielkové kódovanie

Vašou úlohou je vymyslieť také kódovanie, ktoré umožní Alici a Hanke poslať informáciu o ľubovoľnom čase medzi 14:00 a 22:31 použitím štvorcovej mriežky svetielok. Navyše, toto kódovanie musí byť odolné voči jednej chybe, teda Hanka musí vedieť zistiť správny čas aj v prípade, že sa práve jedno svetielko rozsvietilo naopak ako chcela Alica.

Najprv si ukážme, že mriežka 3×3 im stačiť nebude. Počet minút medzi 14:00 a 22:31 je presne $8 \cdot 60 + 32 = 512$. Mriežka 3×3 obsahuje 9 svetielok, pričom každé z nich môže byť buď zapnuté alebo vypnuté. Dokopy je preto $2^9 = 512$ rôznych rozsvietení tejto mriežky.
Navrhnuté kódovanie musí byť *jednoznačné*. To znamená, že Hanka vie pomocou svetielok jednoznačne určiť, o ktorý čas sa jedná. Z toho vyplýva, že každé rozsvietenie mriežky môže kódovať *nanajvýš* jeden čas. Keďže však počet minút a počet konfigurácií mriežky je rovnako veľký, znamená to, že každé rozsvietenie musí prislúchať niektorému z možných časov. Nevieme, na akom priradení týchto časov sa Alica s Hankou dohodnú, ale kvôli jednoznačnosti to musia spraviť.
Tu ale nastáve problém. Predstavme si, že Alica chce Hanke poslať čas x, ktorému zodpovedá mriežka, na ktorej sú zhasnuté všetky svetielka. Mriežka sa však pokazí a keď ju Hanka zapne, vidí na nej svietiť jedno svetielko. Aj takéto rozsvietenie musí zodpovedať nejakému možnému času y. Hanka preto nevie rozlíšiť medzi dvoma možnosťami, pre ktoré má dať rôznu odpoveď – nevie povedať, či sa mriežka nepokazila a Alica chcela poslať čas y alebo či sa svetielko rozsvietilo kvôli chybe a Alica chcela poslať čas x. (A dokonca určite existujú aj mnohé ďalšie časy, pre ktoré tiež mohla dostať to isté rozsvietenie, len v dôsledku inej chyby ako pri čase x.) Tým pádom ale toto kódovanie nie je odolné voči chybe, mriežka 3×3 je preto nedostatočná.

Zálohovanie

Pre zvyšok vzorového riešenia budeme predpokladať, že dievčatá si časy reprezentujú ako počet minút od 14:00. Chcú si preto medzi sebou poslať jedno číslo medzi 0 až 511 (vrátane). Na reprezentáciu tohto čísla budú používať binárnu sústavu, pomocou 9 svetielok teda vedia zakódovať ľubovoľné z týchto čísel a úspešne ho prečítať.

Našou úlohou je navrhnúť kódovanie, ktoré dokáže nielen rozpoznať chybu, ale ju aj opraviť. Môžeme sa preto zamyslieť, ako pristupujeme k oprave chýb (najmä tých digitálnych) v bežnom živote. Obvyklý spôsob je vytváranie záloh. Ak sa naše dáta môžu pokaziť, vytvoríme si ich kópiu, na ktorú sa v prípade problémov obrátime. Alica s Hankou tiež môžu využiť „zálohovanie“ keď správu, ktorú by chceli poslať nakopírujú. Napríklad ak potrebujú 9 svetielok na zakódovanie času, pri použití 18-tich svetielok vedia tento čas poslať v jednej správe dvakrát. Nanešťastie, to im stačiť nebude. Ak sa totiž dve polovice správy nerovnajú, Hanka vie, že nastala chyba, nevie však, v ktorej polovici.
Riešenie hrubou silou je pridanie ďalšej kópie do správy. Chybu totiž Hanka rozozná tak, že jedna z kópií bude iná ako *zvyšné dve* časti. A tie dve, ktoré sa rovnajú musia byť správne. Na poslanie dvoch dodatočných kópií budú potrebovať 27 svetielok a teda mriežku veľkosti 6×6.

Hankiným problém pri posielaní jednej kópie je však iba to, že nevie, ktorá z častí je tá správna. Overenie správnosti sme však riešili na domácom kole, kde k správe stačilo pridať jedno dodatočné svetielko. Konkrétne,

Alica vie využiť pravidlo, že 9 svetielok využije na zakódovanie času a 10-te svetielko rozsvieti iba vtedy, ak je počet rozsvietených svetielok nepárny. To znamená, že správy, ktoré posiela Alica majú vždy rozsvietené párne množstvo svetielok a nepárny počet svetielok môže nastať iba pri chybe svetielka.
Ak teda Alica pridá k správe toto *paritné svetielko* a správu raz nakopíruje, Hanka bude vedieť zistiť, ktorá polovica správy je korektná a z nej odčíta posielaný čas. Na to budú potrebovať $10 + 10 = 20$ svetielok, teda mriežku 5×5.

Pre zaujímavosť si môžete rozmyslieť, že bude postačovať ak sa paritné svetielko pridá iba k jednej zo správ, takémuto riešeniu teda bude postačovať 19 svetielok.

Viac paritných svetielok

Poslať kópiu správy zaberá príliš veľa svetielok. Ak si chceme vystačiť s mriežkou 4×4, budeme musieť vymyslieť lepší spôsob. Uvedomme si, že chyba, ktorá nastáva je jednoduchá – svetielko sa rozsvietilo keď nemalo, alebo naopak. Ak by sme teda vedeli, na ktorom mieste v správe sa chyba vyskytla, vedeli by sme ju opraviť. Stačilo by zmeniť stav príslušného svetielka.
Jedno paritné svetielko bolo dobré na to, aby nám povedalo, či chyba nastala. O jej pozícii sme sa však nedozvedeli nič, nebolo to teda postačujúce. Zaujímavé je, že aj na posielanie kópie správy sa vieme pozerať ako na posielanie paritných svetielok. Konkrétne, ak by sme ku každému svetielku správy pridali jedno paritné svetielko. Následne to, ktoré paritné svetielko nekorešpondovalo s hodnotou správy určovalo, kde nastala chyba. Toto riešenie bolo funkčné, ale použitých svetielok bolo príliš veľa.
Hľadáme teda niečo medzi. Chceli by sme pridať dosť paritných svetielok na to, aby sme vedeli jednoznačne určiť, kde nastala chyba, a zároveň aby ich nebolo príliš veľa a zmestili sme sa do mriežky 4×4.

Dôležitým pozorovaním je, že paritné svetielko sa dá použiť na kontrolu ľubovoľnej sady svetielok. Napríklad si môžeme povedať, že vytvoríme paritné svetielko x, ktoré bude kontrolovať rozsvietenie svetielok na pozíciách 2, 5 a 6. Ak bude počet svetielok medzi svetielkami 2, 5, 6 a x párny, vieme, že všetky tieto svetielka sú v poriadku. Naopak, ak bude nepárny, budeme vedieť, že chyba nastala buď na svetielku 2, 5, 6 alebo x (pozor, aj samotné paritné svetielko sa môže pokaziť).

Spomeňme si, že na kódovanie čísel od 0 po 511 potrebujú Alica s Hankou využiť mriežku 3×3. Ak v tejto správe nastane chyba, čo nám postačuje na jej jednoznačné určenie? Napríklad číslo riadka a číslo stĺpca, kde chyba nastala. Ku každému riadku a každému stĺpcu preto pridáme paritné svetielko, dokopy teda 6 svetielok navyše.
Mriežka, ktorá vznikne, potom vyzerá ako na nasledovnom obrázku. Čísla označujú svetielka použité na kódovanie správy (tmavomodré políčka) a hodnota $p(x, y, z)$ je paritné svetielko k svetielkam číslo x, y a z (svetlomodré svetielka).

0	1	2	p(0, 1, 2)
3	4	5	p(3, 4, 5)
6	7	8	p(6, 7, 8)
p(0, 3, 6)	p(1, 4, 7)	p(2, 5, 8)	

Ak sa pokazí svetielko v rámci časti, ktorá kóduje čas (mriežka 3×3), v danom riadku a stĺpci zrazu nebude rozsvietený párny počet svetielok. Čo Hanke jednoznačne určí, na ktorom mieste nastala chyba a stačí, aby správu rozkódovala s tým, že na tejto pozícii je svetielko rozsvietené opačne. Ak sa pokazí niektoré z paritných svetielok, ovplyvní to paritu *iba* v danom riadku alebo stĺpci, ktoré toto paritné svetielko dopĺňalo. Ak teda

vidí, že je pokazená iba jedna parita, vie, že mriežka 3×3 je v poriadku. No a ak sa žiadne svetielko nepokazí, vo všetkých riadkoch aj stĺpcoch bude rozsvietený párny počet svetielok a Hanka vie, že všetko je tak, ako to poslala Alica.

Všimnite si, že pravé dolné svetielko v našom kódovaní vôbec nepoužívame. Ak by sme Hanke chceli zjednodušiť robotu, vieme toto políčko využiť na kontrolu paritných svetielok – bude to paritné svetielko pre paritné svetielka. Alica toto svetielko rozsvieti tak, aby celkový počet rozsvietených paritných svetielok bol párny.
Hanka potom ako prvé skontroluje, či sedí celková parita paritných svetielok. Ak nie, chyba nastala v nich, ale tým pádom je pôvodná správa v poriadku a stačí ju prečítať a paritné svetielka ignorovať. V opačnom prípade sú paritné svetielka v poriadku a vieme ich využiť na nájdenie chyby (ak nastala).

Hammingovo kódovanie

V tejto úlohe ste tvorili takzvaný „samoopravný kód“. Teda spôsob prenosu dát, ktorý vie detegovať a tiež opravovať chybu. Vo vašom prípade ste potrebovali korekciu jednej chyby, existujú však aj všeobecnejšie riešenia pre väčší počet možných chýb.
Nami navrhnuté riešenie nie je v skutočnosti najlepšie možné a počet svetielok potrebných na kódovanie by sa dal ešte zmenšiť. Stále by sme však potrebovali aspoň mriežku 4×4, akurát viac svetielok by bolo nevyužitých, preto toto riešenie nebolo potrebné na získanie plného počtu bodov.
Týmto riešením je Hammingovo kódovanie, ktoré využíva podobnú myšlienku ako v predchádzajúcej časti, akurát efektívnejšie. Paritné svetielka sú nastavené tak, aby poskytovali čo najviac možnej informácie. V našom riešení každé paritné svetielko kontroluje niektoré iné tri svetielka. Ak sa táto parita poruší, vieme, že chyba je v niektorom z týchto troch svetielok. Ak je však správna, chyba môže byť vo zvyšných *až šiestich svetielkach*.
Hammingovo kódovanie nastavuje paritné svetielka tak, aby dávali informáciu o *polovici* všetkých svetielok. Vďaka tomu, bez ohľadu na to, či je parita porušená alebo nie, vieme vylúčiť polovicu svetielok, ktoré sú k dizposícii – ak sa parita neporuší, vieme vylúčiť tie svetielka, ktoré sú kontrolované (ktorých je polovica); ak sa poruší, vylúčime tú polovicu, ktoré nie sú kontrolované.
Presný spôsob ako nastaviť pokrytie paritnými svetielkami tak, aby sme vždy jednoznačne určili kde nastala chyba si viete nájsť alebo skúsiť rozmyslieť. Princípom je klásť nasledovné otázky: Je správne rozsvietená prvá polovica svetielok? Je správne rozsvietená prvá a tretia štvrtina svetielok? Je správne rozsvietená prvá, tretia, piata a siedma osmina svetielok? Atď. (Ako súvisí takéto delenie do skupín s poradovými číslami svetielok, keď ich číslujeme od nuly a zapíšeme si ich v dvojkovej sústave?)

ŠTYRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Ján Hozza, Andrej Korman, Paulína Smolárová
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024