Korešpondenčný seminár z programovania
XL. ročník, 2022/23
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

Filipko
(max. 12 b za popis, 8 b za program)

1. Ploský kamoši

Zo zadania

Ako zadanie hovorí, budeme spracovávať dvojice slov - jedno, ktoré vieme ako vyzerá (nazvime ho w) a druhé, ktoré si matne pamätáme z noci (nazvime ho q). Úlohou je zistiť, či sa slová môžu zhodovať.

Riešenie

Pre každé písmeno vo w sa budeme musieť pozrieť na písmeno alebo množinu písmen z q a zistiť, či medzi nimi existuje zhoda. Budeme si pamätať, na akej pozícii v slove w sa nachádzame a budeme prechádzať cez znaky slova q. Ak sa i-ty znak z q nerovná `(`, vieme, že na tejto pozícii bude len jedno písmeno. Teda porovnáme písmeno z q s písmenom z w. Ak sa nerovnajú, vypíšeme `NOT OK` a posunieme sa na ďalšiu dvojicu slov. Inak pokračujeme ďalším písmenom. Ak sa i-ty znak q rovná `(`, budeme prechádzať len cez q, až kým nepriídeme po `)` pričom si budeme ukladať do množiny písmená medzi zátvorkami. Po tom, ako sme našli v q znak `)`, pozrieme sa, či sa písmeno na pozícii, ktorú porovnáme, nachádza v tejto množine znakov. Ak sa v ňom nenachádza, vypíšeme `NOT OK` a posunieme sa na ďalšiu dvojicu slov. Takto prejdeme cez všetkých l znakov w a ak sme ich prešli všetky a nenašli nezhodu, môžeme vypísať `OK`.

Optimalnejsie riesenie

Optimálne riešenie je v podstate takmer totožné. V hore opísanom riešení prechádzame niektoré znaky viackrát. Práve vtedy, keď získavame znaky medzi zátvorkami a následne hľadáme, či sa v nich nachádza písmeno z w na porovnávanej pozícii. Riešenie vieme modifikovať tak, aby sme miesto pridávania do množiny rovno písmeno porovnávali s daným písmenom slova w.

Časová a pamäťová zložitosť

Časová zložitosť bude lineárna, keďže vstupom prechádzame len raz. Pamäťová zložitosť je konštantná – rovná l, keďže si musíme pamätať celé slovo w a jedno akutálne písmeno zo slova, ktoré si pamätáme z noci.

Listing programu (Python)

```
l = int(input())
d = int(input())

def createAlphabetArr(token: str) -> list:
    out = [False] * 26
    for i in token:
        out[ord(i) - ord('A')] = True
    return out


def correctWord(arr: list) -> int:
    for c in range(len(word)):
        ci = ord(word[c]) - ord('A')
        if not arr[c][ci]:
            return "NOT OK"
    return "OK"
```

```
for i in range(d):
    word = input()

    arr = []
    inp = input()
    j = 0
    while j < len(inp):
        c = inp[j]
        token = ""
        if c != '(':
            token = c
        else:
            while c != ")":
                j += 1
                c = inp[j]
                if c == ')': break
                token += c
            c = inp[j]
        arr.append(createAlphabetArr(token))
        j += 1
    print(correctWord(arr))
```

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>

void numberOfCorrectWords(const std::vector<std::vector<bool>> &arr, const std::
    ↪ string &word) {
    int out = 0;
    for (int i = 0; i < word.length(); ++i) {
        if (!arr[i][word[i] - 'A']) {
            std::cout<<"NOT OK"<<std::endl;
            return;
        }
    }
    std::cout<<"OK"<<std::endl;
    return;
}


int main(int argc, char const *argv[]) {
    int l = 0, w = 0, q = 0;
    std::cin >> l >> w;

    std::string word;
    for (int i = 0; i < w; ++i) {
        std::cin >> word;
        std::string query;
        std::vector<std::vector<bool>> arr(l, std::vector<bool>(26));
        std::cin >> query;
        int tokenInd = 0;
        for (int j = 0; j < query.length(); ++j) {
            char c = query[j];
            std::string token;
            if (c != '(') token += c;
            else {
```

```
                while (c != ')') {
                    c = query[++j];
                    if (c != ')') token += c;
                }
            }
            for (char ch: token) {
                arr[tokenInd][ch - 'A'] = true;
            }
            tokenInd++;
        }
        numberOfCorrectWords(arr, word);
    }
}
```

Janka

2. Láska kvitne v júni

(max. 12 b za popis, 8 b za program)

Zádrhel tejto úlohy spočíva v tom, že dve postele na izbe potrebujú obličky rôznych veľkostí. To znamená, že v niektorých prípadoch budeme musieť kúpiť dva kusy obličiek rovnakého typu (ale rôznych veľkostí). Keď sa rozhodujeme, z ktorých typov obličiek potrebujeme dva kusy, potrebujeme zvážiť niekoľko prípadov.

Každá ploštica preferuje iný typ obličiek

Tento prípad zodpovedá prvej sade, kde platí, že preferencia ploštice p_i sa rovná i.

V tomto prípade, keď n sa rovná 1, vieme, že ploštica pôjde na rande sama a je jedno, akú veľkosť obličiek kúpime, stačiť bude jeden kus.

Ak n sa rovná 2, tak vieme, že každá z ploštíc pôjde na rande presne raz, a to naraz s druhou plošticou z páru. Tým pádom pre každú plošticu potrebujeme presne jedny obličky (rôznej veľkosti) a výsledok, ktorý hľadáme je 2.

Komplikovanejšie to začína byť v prípade, že máme viac, ako dve ploštice. Bez ujmy na všeobecnosti si môžeme povedať, že prvá z n ploštíc sa bude vždy nachádzať na posteli číslo 1, a tak pre ňu zaobstaráme iba jednu veľkosť obličiek. Pri druhej ploštici si podobne môžeme povedať, že sa vždy bude nachádzať na posteli číslo 2, a tak pre ňu stačí tiež zaobstarať iba jednu veľkosť obličiek. Zatiaľ nám v tom nič nebráni, keďže tieto ploštice vedia spokojne ísť spolu na rande. No akonáhle by sme chceli povedať, že tretia ploštica bude vždy na posteli číslo 1, zistíme, že nebude vedieť ísť na rande s prvou plošticou. Takisto, ak povieme, že bude vždy na posteli číslo 2, nebude môcť ísť na rande s druhou plošticou.

Vieme teda povedať, že pre tretiu plošticu musíme kúpiť obe veľkosti obličiek, aby mohla ísť na rande s prvou aj s druhou plošticou. Toto platí aj pre všetky zvyšné ploštice. Tým pádom máme 2 ploštice (prvú a druhú), pre ktoré kúpime jeden kus obličiek a $n - 2$ ploštíc, pre ktoré kupime dve obličky. Celkový počet obličiek, ktorý kúpime je teda $2 * (n - 2) + 2$, po úprave $2n - 2$.

Listing programu (Python)

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

if n <= 2:
    print(n)
else:
    print(2*n-2)
```

Ploštice môžu preferovať rovnaký typ obličiek

V druhej sade sa stretneme so situáciou, kde rôzne ploštice môžu preferovať rovnaký typ obličiek. Tu je dôležité si uvedomiť, že ak existujú aspoň dve ploštice, ktoré majú rady rovnaký typ, potrebujeme presne dva kusy tohoto typu, aby dve ploštice s touto preferenciou mohli ísť spolu na rande. Taktiež nepotrebujeme viac ako 2 kusy, každý na jednu posteľ, keďže jeden kus obličiek môže byť použitý viackrát.

Poďme sa pozrieť len na tie typy obličiek, ktoré sú preferované len jednou plošticou. Zmenila sa nám situácia tým, že sme niektoré obličky rovno kúpili dva krát? Nie, keďže bezohľadu na to, ako pokúpime tie obličky,

ktoré preferuje len jedna ploštica, budú vedieť ísť randiť s plošticami, ktorým sme práve kúpili obličky na obe postele.

Obličky, ktoré preferuje práve jedna ploštica, teda môžeme ponakupovať úplne nezávisle a spravíme to tak, ako sme si vysvetlili v predošlej sekcií. Čiže z týchto obličiek môžeme zobrať maximálne dva rôzne typy, z ktorých nám stačí kúpiť jeden kus a pre zvyšné typy platí, že musíme kúpiť dva kusy.

Dokázali sme si, že opäť môžeme mať maximálne dva typy obličiek, z ktorých kúpime jeden kus. A podmienkou je, že tieto typy musia byť preferované len jendou plošticou. Otázkou je, ako zistíme, či takéto typy existujú a koľko ich je.

Potrebujeme použiť dátovú štruktúru, ktorá nám pre každý typ obličiek povie, koľko ploštíc danú obličku preferuje.

Jednou takouto dátovou štruktúrou je vector (v pythone list) veľkosti k, v ktorom každá pozícia prezentuje typ obličky a hodnota na danej pozícii reprezentuje počet ploštíc, ktoré danú obličku preferujú. Pri načítavaní sa pozrieme na preferenciu konkrétnej ploštice a navýšime zodpovedajúcu hodnotu vo vectore o 1. Na konci celý vector prejdeme a v osobitnej premennej raz si zapamätáme, koľko hodnôt vo vectore sa rovnalo 1 a v premennej $viac$ koľko ich bolo aspoň 2.

Každú z $viac$ typov obličky budeme musieť kúpiť dva krát a obličiek z raz typov kúpime podľa predošlého vzorca: raz kusov, ak sú najviac 2, inak $2raz - 2$.

Časová zložitosť načítania vstupu a prejdenia listu je $O(n +)$ a pamäťová je $O(k)$, keďže si vytvárame pole veľkosti k.

Listing programu (Python)

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

pocty = [0 for _ in range(k)]

preferencie = input().split()
for x in preferencie:
    pocty[int(x)-1] += 1

raz, viac = 0, 0
for pocet in pocty:
    if pocet == 1:
        raz += 1
    elif pocet >= 2:
        viac += 1

pre_jednotlivcov = raz if raz <= 2 else 2*raz-2

print(viac*2 + pre_jednotlivcov)
```

Použitie mapy

Riešenie, ktoré sme si predstavili, môže byť ešte efektívnejšie, ak použijeme dátovú štruktúru map (dictionary v pythone). Táto štruktúra nám dovoľuje držať v pamäti a prechádzať hodnoty zodpovedajúce len tým typom obličiek, čo sú preferované aspoň jednou plošticou. Ak napríklad máme veľké k (10^9), ale všetky plošticе preferujú ten istý typ obličiek, v mape si budeme pamätať údaje práve o tomto jednom type.

Časová a pamäťová zložitosť sa nám teda zhodí na $O(n)$. Nebolo nám to však pri daných obmedzeniach treba.

Listing programu (Python)

```
from collections import Counter

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

cnt = Counter(input().split())

one, more_than_one = 0, 0
```

```
for c in cnt.values():
    if c == 1:
        one += 1
    else:
        more_than_one += 1

for_ones = one if one<=2 else 2*one-2

print(more_than_one*2 + for_ones)
```

David Krchňavý

3. Opravte matfyz

(max. 12 b za popis, 8 b za program)

Úlohou bolo nájsť najrýchlejší možný čas, za ktorý sa dalo splniť n úloh, pričom splnenie každej trvá t hodín. Plnenie úloh bolo možné urýchliť (ale aj spomaliť) vyškolením a najatím najviac r robotníkov, pričom každý má samostatne určenú zručnosť.

Bruteforce

Platí, že jediné spôsoby, akými môžeme ovplyvniť čas splnenia úloh sú:

- počet robotníkov, ktorých najmeme, označme si ho i, pričom $0 \leq i \leq r$ - to je r možností
- ktorých konkrétnych i robotníkov z r si vyberieme - to je $\binom{r}{i}$ možností
- po splnení ktorej úlohy každého z nich vyškolíme - to je r^n možností.

Čas splnenia n úloh, pričom každá trvá t hodín s i robotníkmi vieme vypočítať v konštantnom čase vzorcom $\lceil \frac{n}{i} \rceil \cdot t + u$, kde u je doba školenia daných robotníkov. Pomocou bruteforce teda vyskúšame nájsť čas splnenia úloh pre každú kombináciu týchto faktorov. Pamäťová zložitosť takéhoto riešenia je $O(r)$, keďže si pamätáme len zručnosť každého robotníka. Časová zložitosť takéhoto riešenia je $O(r^n)$.

Vlastnosti optimálneho riešenia

Poďme za pozrieť na spôsoby, akými vieme eliminovať niektoré z vyššie opísaných faktorov.

Kedy vyškoliť robotníkov

Zamyslime sa nad tým, kedy sa najviac oplatí školiť robotníkov. Predpokladajme, že už sme našli optimálny počet aj výber konkrétnych robotníkov, ktorých zamestnáme a stačí nájsť už len optimálne úlohy, po ktorých splnení jednotlivých robotníkov vyškolíme. Zrejme platí, že to bude vždy ešte pred začatím stavby, keď nie je splnená ani jedna úloha. Keďže školenie robotníka trvá rovnako dlho bez ohľadu na to, kedy sa začne, nikdy sa neoplatí školenie nejakých robotníkov odkladať, lebo sa tým zbytočne stratí čas, počas ktorého mohli títo robotníci pracovať, ale nepracujú.

Ktorých robotníkov si vybrať

Teraz predpokladajme, že už máme optimálny počet robotníkov a aj sme určili správny moment, kedy ich vyškoliť (už vieme, že to je na začiatku). Môže nastať prípad, kedy je optimálny počet robotníkov menší ako r, čiže si môžeme vyberať, ktorých konkrétnych najmeme. V tomto prípade bude zrejme najoptimálnejšie najať tých najzručnejších, lebo ich školenie zaberie najkratší čas.

Koľko robotníkov vyškoliť

Keďže sme už určili, kedy sa najviac oplatí vyškoliť robotníkov aj ktorých konkrétnych si vybrať, stačí nám už len nájsť optimálny počet robotníkov, ktorý vyškolíme. Nakoľko čas splnenia n úloh s i robotníkmi ak každá trvá t dlho vieme vypočítať v konštantnom čase vzorcom z predošlého riešenia, môžeme si dovoliť vyskúšať všetky možné počty robotníkov, ktoré zaškoliť. Uvedomme si, že ak by sme aj mali možnosť vyškolit viac robotníkov ako je úloh, zrejme sa to nikdy neoplatí, a teda počet robotníkov ktorý sa oplatí vyškolit je zhora ohraničený počtom úloh.

Počet možností bude preto vždy najviac $\min(r, n)$.

Riešenie

Zhrňme si, ako bude vyzerať nájdenie optimálneho riešenia. Nemusíme skúšať všetky možné úlohy, po ktorých vyškoliť robotníkov, ale ich všetkých vyškoliť hneď zo začiatku. Zároveň nemusíme skúšať všetky

možné kombinácie jednotlivých robotníkov, ale vybrať vždy len tých najzručnejších. Zoradenie robotníkov od najzručnejších zaberie $O(r \log r)$. Tým pádom stačí nájsť len optimálny počet robotníkov. Tento počet nájdeme vyskúšaním všetkých možností. Teda, pre každý počet robotníkov od 0 po r vypočítame, ako dlho bude trvať splnenie úloh a vypíšeme najmenšiu nájdenú hodnotu, čo bude trvať O(r). Pamäťová zložitosť takéhoto riešenia je $O(r)$, keďže si pamätáme len zručnosť každého robotníka. Časová zložitosť takéhoto riešenia je $O(r \log r)$.

Listing programu (Python)

```
from math import ceil


def time(n: int, t: int, i: int) -> int:
    """č
    Výpoet času do splnenia výstavby.
    :param n: čpoet úloh
    :param t: čpoet hodín, ktorý trvá jedna úloha
    :param i: čpoet robotníkov (bez stavbyvedúceho), ktorí budú ťrobi úlohy
    :return: čPoet hodín, ktorý trvá splnenie n úloh, čpriom žkadá trvá t hodín,
        ↪ \
     plus čpoet hodín strávený školením i robotníkov
    """
    return suma[i] + ceil(n / (i + 1)) * t


n, t, r = map(int, input().split())
robotnici = list(map(int, input().split()))
robotnici.sort()  # robotníkov zoradíme od čšnajzrunejích (takých, ktorých
    ↪ školenie bude ťtrva šnajkratie) vzostupne

suma = [0] * (r + 1)
# pred for cyklom: suma[i] = 0, 0 <= i <= r
# po for cykle: suma[i] = čpoet hodín, ktorý trvá šzakolenie i čšnajzrunejích
    ↪ robotníkov
for i in range(r):
    suma[i + 1] = suma[i] + robotnici[i]
print(min(time(n, t, i) for i in range(0, min(r + 1, n))))
```

Listing programu (C++)

```
#include<iostream>
#include<cmath>
#include<algorithm>
#include<vector>
#include<numeric>

using namespace std;

/**
 * čVýpoet času do splnenia výstavby.
 * @param n čpoet úloh
 * @param t čpoet hodín, ktorý trvá jedna úloha
 * @param i čpoet robotníkov (bez stavbyvedúceho), ktorí budú ťrobi úlohy
 * @param suma časy školení robotníkov
 * @return čPoet hodín, ktorý trvá splnenie n úloh, čpriom žkadá trvá t hodín,
     plus čpoet hodín strávený školením i robotníkov.
 */
long long cas(long long n, long long t, long long i, vector<long long> &suma) {
    return suma[i] + (long long) (ceil((double) n / (double) (i + 1))) * t;
```

```
}

int main() {
    long long i, n, t, r;
    cin >> n >> t >> r;

    vector<long long> robotnici(r + 1);
    vector<long long> suma(r + 1);

    for (i = 0; i < r; i++)
        cin >> robotnici[i + 1];

    // robotníkov zoradíme od čšnajzrunejích (takých, ktorých školenie bude ťtrva
    ↪  šnajkratie) vzostupne
    sort(robotnici.begin(), robotnici.end());
    // suma[i] = čpoet hodín, ktorý trvá školenie i čšnajzrunejích robotníkov
    partial_sum(robotnici.begin(), robotnici.end(), suma.begin());

    long long res = n * t;
    for (i = 0; i < min(r + 1, n + 1); i++) {
        res = min(res, cas(n, t, i, suma));
    }

    cout << res << endl;
}
```

Janči

4. Štrádovanie si

(max. 12 b za popis, 8 b za program)

Našou úlohou je nájsť **najmenší** možný strom (čiže súvislý acyklický graf), ktorý zodpovedá popisu. V takomto strome, pre každú zadanú výšku, existuje vrchol danej výšky, ktorý susedí s vrcholmi výšok zadaných tesne pred a po jeho výške.

Ako vyzerá najmenší strom

Môžeme si všimnúť, že nemá zmysel, aby jeden vrchol V mal dvoch susedov rovnakej výšky. Keby takí dvaja susedia existovali, nazvime ich S a T, mohli by sme zobrať všetkých susedov vrcholu S (vrátane V) a napojiť ich do T namiesto do S. Potom by sme mohli vrchol S úplne zmazať, pretože by nemal žiadnych susedov. Postupnosť výšok by bola stále validná, pretože medzi každými dvoma susedmi pôvodného vrcholu S by viedla rovnaká cesta cez T (čo sa týka poradia výšok), ako viedla cez S. Zároveň by náš nový strom mal o jeden vrchol (S) menej, čo by bol spor s tvrdením, že pôvodný strom bol najmenší možný. Preto vieme, že žiadny vrchol nemá dvoch rovnako vysokých susedov.

To už nám v podstate hovorí, ako vytvoriť najmenší strom – budeme si vždy pamätať posledný navštívený vrchol a pre každú výšku sa pozrieme, či tento vrchol už má suseda danej výšky. Ak má, iba sa do tohoto suseda presunieme. Ak nemá, vytvoríme si nový vrchol danej výšky a presunieme sa do neho.

Ideálny strom nemôže byť menší, pretože sa nevieme presúvať medzi vrcholmi, ktoré nesusedia (a keďže ide o strom, vrcholy, ktoré nie sú zadané tesne po sebe v nejakej časti postupnosti výšok, spolu susediť nemôžu, inak by existoval cyklus).

Ako to implementovať

Vrcholy si budeme pamätať v poradí, v akom sme ich prvý krát videli. Na to vieme použiť obyčajné pole, kam vždy pri vytvorení nového vrcholu pridáme jeho výšku. Vrcholy budeme inde reprezentovať indexami tohoto poľa.

Zároveň si pre každý vrchol chceme pamätať výšky všetkých jeho susedov, a to tak, aby sme sa vedeli čo najrýchlejšie spýtať, či má aktuálny vrchol suseda danej výšky. A ak áno, ktorý to je. Pomalá implementácia by mohla vyzerať takto:

Pre každý vrchol máme pole, v ktorom si ukladáme všetkých jeho susedov. Keď chceme zistiť, či má vrchol suseda s danou výškou, prejdeme všetkých jeho susedov a skúsime nájsť jedného so správnou výškou. Takáto

implementácia je kvadratická, čo sa najlepšie predstavuje na grafe tvaru hviezdy – každý druhý raz navštívime stredový vrchol a potom prejdeme až lineárne mnoho vrcholov, aby sme zistili, či treba pridať nový.

Aby sme zlepšili časovú zložitosť, môžeme namiesto poľa susedov použiť rýchlejšiu dátovú štruktúru - napríklad slovník alebo vyhľadávací strom. Pri použití slovníku a hashovania sa nám lineárny čas na otázku zlepší na konštantný a taký bude aj čas pridávania vrcholu. Pri použití vyhľadávacieho stromu budú oba časy logaritmické.

Časová a pamäťová zložitosť

Prechádzame celý vstup a pre každú výšku robíme konštantne mnoho operácií, preto je celková časová zložitosť $O(n)$ (respektíve $O(nlogn)$ pri použití vyhľadávacieho stromu namiesto slovníka). Pamäťová zložitosť je rovná počtu vrcholov plus počtu hrán, ale keďže graf je strom, je oboch lineárne mnoho.

Pri praktickej implementácií môžeme tiež použiť jeden slovník pre všetky vrcholy tak, ako v autorskom riešení.

Listing programu (Python)

```
def solve():
    vysky = [] # Zoznam vsetkych vrcholov podla poradia prveho navstivenia
    susedia = dict() # Index suseda (3) vrcholu cislo (1) s vyskou (2),
                     # indexujeme susedia[(1, 2)] = 3
    prvy_vrchol = True
    n = int(input())

    for vyska in map(int, input().split()):
        # Prejdeme postupne cely zoznam vysok:
        if prvy_vrchol:
            # Ak je to uplne prvy vrchol, len ho pridame
            prvy_vrchol = False
            vysky.append(vyska)
            index = 0 # Index oznacuje aktualny vrchol v zozname vysok
            continue

        if (index, vyska) in susedia.keys():
            # Ak ma uz aktualny vrchol suseda hladanej vysky, len sa presunieme
            index = susedia[(index, vyska)]
        else:
            vysky.append(vyska) # Pridame novy vrchol
            susedia[(len(vysky) - 1, vysky[index])] = index
            # Pridame prepojenie z noveho vrcholu na aktualny
            susedia[(index, vyska)] = len(vysky) - 1
            # Pridame prepojenie z aktualneho vrcholu na novy
            index = len(vysky) - 1 # Presunieme sa do noveho vrcholu

    return len(vysky)
    # Nakoniec vratime pocet rozdielnych navstivenych vrcholov

print(solve())
```

Paulinka

5. Taktická Záchrana

(max. 12 b za popis, 8 b za program)

Prvé technické detaily

Pred tým, ako sa pustíme do riešenia úlohy, pozrime sa na to, ako si reprezentujeme druhy ploštíc v posteliach a ako s touto reprezentáciou pracovať.

V zadaní sú ploštice v nejakej posteli reprezentované cez celé čísla - presnejšie cez ich binárny zápis. Ak čísla a a b reprezentujú ploštice v dvoch posteliach. Farby, ktoré sa nachádzajú na niektorej z nich, sa dajú zistiť pomocou *binárneho ORu* – a to ako `a | b` (aj v pythone aj v C++). Táto operácia vráti číslo ktoré má na i-tom bite jednotku práve vtedy, ak má aspoň jedno z čísel a a b na i-tom bite jednotku.

Jednoduché riešenie

Vybavený potrebnou mechanikou, poďme sa pustiť do riešenia úlohy. Ak vieme stav každej postele, vieme zistiť pre každú posteľ, aké farby ploštíc sa zachránia, ak práve táto bude napadnutá – jednoducho si vypočítame OR ostatných postelí. Takto vieme odpovedať na jednu testovaciu sadu pomocou n otázok v $O(n^2)$ čase pomocou a $O(n)$ pamäti. A tak vieme získať 1 bod.

Toto riešenie vieme zrýchliť pomocou OR-ekvivalentu prefixových a suffixových súčtov: prefixových a suffixových OR-ov. Vieme si v $O(n)$ čase spočítať pre každé $1 \leq i \leq n$, aké farby ploštíc sú na posteliach $1, 2, \ldots, i-1$ a aké farby ploštíc sú na posteliach $n, n-1, \ldots, i+1$ a následne vypísať OR týchto dvoch hodnôt (všimnite si, že na vypočítanie prvej hodnoty pre pozíciu i z hodnoty pre pozíciu $i-1$ stačí konštantný čas a nápodobne pre druhú hodnotu z hodnoty pre $i+1$).

Menej otázok: pokrývame postele

Na viac bodov sa musíme vedieť nespýtať na každú posteľ separátne. Ako na to?

Položme si otázku inak. Pre aké otázky vieme vyskladať odpoveď?

Predstavme si, že sa spýtame na k množín postelí. Označme ich ako $S_1, \ldots, S_k$ a dostaneme odpovede $a_1, \ldots, a_k$. Kedy vieme zistiť z týchto otázok informáciu "aké farby ploštíc ostanú, ak bude prvá zasiahnutá posteľ i"? Nemôžeme použiť informácie zo žiadnej množiny ktorá obsahuje i - inak nevieme, či farba ploštíc je na posteli i alebo aj na inej posteli z množiny. Pozrime sa teda na všetky množiny neobsahujúce i. Ak je každá iná posteľ (okrem i) obsiahnutá v niektorej z týchto množín, všetky jej farby ploštíc budú obsiahnuté v OR-e odpovedí pre tieto postele.

Takže dostávame podmienku: na to aby sme vedeli odpovedať s pomocou otázok na množiny $S_1, \ldots, S_k$, musí pre každý pár rôznych postelí $i \neq j$ existovať množina, v ktorej nie je posteľ číslo i, ale nachádza sa v nej posteľ j.

Ako ale takúto množinu skonštruovať?

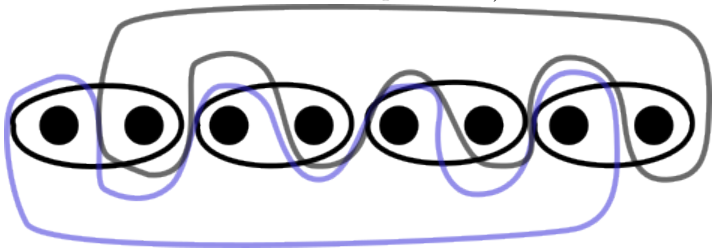
$n/2 + 2$ otázok

Hint, ako vyriešiť úlohu na dva body je v počte otázok: $n/2$ naznačuje, že by sme si mohli rozdeliť postele na dvojice – postele $1, 2$, postele $3, 4$, až $n-1, n$[1] (na obrázku môžme vidieť rozdelenie pre $n = 8$). Takéto rozdeliene nám ale ešte úplne úlohu nevyrieši.

Pre každú z postelí, zjednotenie množín, ktoré ju neobsahujú, obsahujú všetky postele okrem jej dvojice. Chceli by sme teda pridať nejaké (najviac dve) množiny, pomôcou ktorých by sme vedeli "rozlíšiť" aj postele v jeden dvojici. Na to nám stačí si uvedomiť dve veci:

1. Je v poriadku, ak sú informácie o nejakej posteli "pridané dvakrát" (keďže nepočítame počet výskytov rôznych farieb, len či sú alebo nie sú)
2. V každej dvojici je jedna párna a jedna nepárna posteľ

Teda nám stačí sa navyše spýtať na množinu všetkých párnych a na množinu všetkých nepárnych postelí (ako môžme vidieť na obrázku dole pre $n = 8$).



Vieme ľahko skontrolovať, že táto voľba množín spĺňa horeuvedenú podmienku: pre každé dve rôzne postele - buď nenasledujú priamo za sebou (a teda nie sú v rovnakej dvojici) alebo majú rozličnú paritu.

Skonštruovať odpoveď pre každú posteľ vieme pomocou $n/2$ otázok, a tak dostaneme časovú zložitosť $O(n^2)$ a pamäťovú $O(n)$.

[1] Ak je n nepárne, môžeme sa spýtať na n ako množinu s jediným prvkom, nezhorší nám to počet otázok

Vieme zlepšiť toto riešenie?

V predchádzajúcom riešení sme rozdelili postele na po sebe idúce dvojice, a následne podľa zvyšku po delení dvoma. Ale číslo dva nie je v tomto prípade špeciálne! Mohli by sme nápodovne použiť trojice, štvorice, resp. akékoľvek k-tice. Rovnakou stratégiou ako pre dvojice, vieme získať pre k-tice odpoveď pomocou $n/k+k$ otázok.

Ak sa s týmto výrazom pohráte, zistíte, že najmenej otázok $(2\lceil\sqrt(n)\rceil)$ nám výjde pre k rovné $\sqrt(n)$ (zaokrúhlené na najbližšie celé číslo). Takéto zlepšenie nám dá 4 body a zlepší časovú zložitosť na $O(n\sqrt{n}))$ a pamäťovú na $O(n)$.

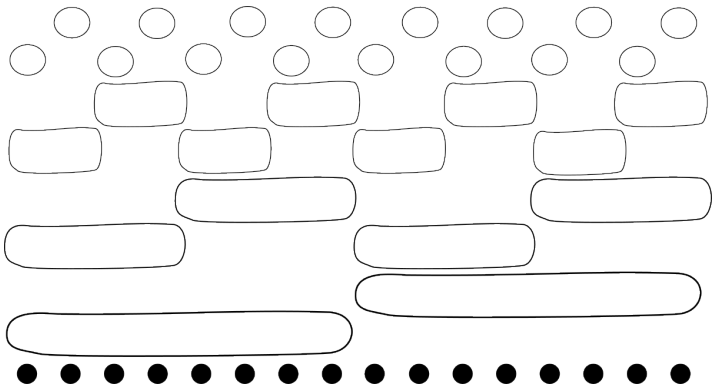
Ešte menej otázok

Skupinkovaciu metódu sme už očividne zlepšili ako sa dalo, ale ešte stále nemám plný počet bodov. Mohli však by sme sa inšpirovať na lepšie riešenie?

Vráťme sa späť ku rozdeleniu postelí na dvojice. Toto rozdeľovanie je zjavne neefektívne, keďže potrebujeme lineárne množstvo množín, aby sme pokryli všetky postele. Čo keby sme sa v jednej množine spýtali na polovicu dvojíc a v druhej na druhú polovicu dvojíc. Takto stále očividne nevieme rozoznávať všetky postele. Odpoveďou je: rozdelenie dvojíc na polovice viacerými spôsobmi.

Limit $2\log n$ v zadaní nám vie napríklad napovedať nasledovné množiny:

- všetky párne postele, všetky nepárne postele
- každá párna dvojica, každá nepárna dvojica
- každá párna štvorice, každá nepárna štvorica
- …
- prvá polovica postelí, druhá polovica postelí[2]



Na obrázku môžeme vidieť množiny pre $n = 16$. Ako prvé si všimnime, že takto dostaneme $2\lceil\log n\rceil$ otázok. Po druhé, tieto množiny nám vedia vyskladať všetky odpovede – predstavme si, že chceme odpoveď na i-tú posteľ (teda, ktoré farby sú na ostatných posteliach). Pozrime sa na binárny zápis čísla i. Ten nám presne povie ktoré query máme skombinovať. Ak má i[3] na j-tom bite 1, poto chceme použiť odpoveď pre *párne* 2^i-tice a v opačnom prípade odpoveď pre *nepárne* 2^i-tice. Keďže rôzne postele majú rôzne binárne zápisy a v každom "leveli" otázok sa každá posteľ nachádza v presne jednej z množín, vieme tak rozlišovať každú dvojicu postelí.

Takto dostaneme teda riešenie v časovej zložitosti $O(n\log n)$ s pamäťou $O(n)$, ktoré vie získať 6 bodov (pre $n = 1000$ použije 20 otázok).

Vzorové riešenie

Narozdiel od predchádzajúcich riešení, vo vzorovom si množiny necháme vygenerovať programom, ale nie na základe jednoduchého vzoru.

Pozrime sa na množiny z pohľadu postelí: v ktorých množinách z k opýtanych množín sa bude nahádzať posteľ i?

Zjavne dáva zmysel, aby každá posteľ bola v približne rovnakom počte množín. V predchádzajúcom riešení bola každá posteľ v polovici množín, čo keby sme sa týmto princípom inšpirovali?

Vzorové riešenie potrebuje použiť najviac 13 množín, mohli by sme zariadiť aby každá posteľ bola v 6 z nich?

Existuje $\binom{13}{6} = 1716$ rôznych spôsobov, akým je možné vybrať 6 z 13 množín, do ktorých nejaká posteľ bude patriť. Keďže to je menej, ako všetkých postelí, ktoré vieme na vstupe dostať, vieme pre každú vybrať unikátny výber množín, v ktorých bude zaradená. Takto teda vieme nájsť pre každú dvojicu postelí množinu, v ktorej prvá posteľ je a druhá nie je (keďže všetky postele sú v rovnakom počte množín).

[2] rozmyslite si, ak n nie je mocnina dvojky

[3] číslujme tu postele od nuly

Teda na dokončnie riešenia nám treba len vymyslieť ako ho implementovať – odporúčam použiť `next_permutation` v C++, alebo knižnicu `itertools` v pythone. Keď si množiny generujeme, vieme si rovno zapísať, ktorú množiny treba OR-ovať, aby sme získali výsledok pre tú-ktorú posteľ. Toto riešenie použije 13 otázok a vieme ho implementovať v čase a pamäti $O(n)$.

V skutočnosti, ak by n nebolo horne ohraničená, potrebovali by sme rádovo $\log n$ otázok – vieme ukázať, že menej než $2 \log n$ postačí, ale potrebujeme aspoň $\log n$[4], takže časová zložitosť je v skutočnosti $O(n \log n)$.

Listing programu (Python)

```
#!/usr/bin/env python3
import itertools, sys

def has_seven_ones(a):
    cnt = 0
    for i in range(13):
        if ((1 << i) & a) > 0:
            cnt += 1
    return (cnt == 7)

def solve():
    n = int(input())
    k = 0
    a = (1 << 7) - 1
    S = [[] for _ in range(13)]
    ktore = [[] for _ in range(n)]
    res = [0 for _ in range(13)]
    while k < n:
        if has_seven_ones(a):
            for i in range(13):
                if ((1 << i) & a) > 0:
                    S[i].append(k)
                else:
                    ktore[k].append(i)
            k += 1
        a += 1

    for i in range(13):
        print("? {} {}".format(len(S[i]), " ".join(map(lambda x: str(x + 1), S[i
            ↪ ]))))
        sys.stdout.flush()
        res[i] = int(input())
    final = [0 for _ in range(n)]
    for i in range(n):
        for j in ktore[i]:
            final[i] |= res[j]
    print("! {}".format(" ".join(map(str, final))))

t = int(input())
for _ in range(t):
    solve()
```

[4] žiaľ presný počet nie je pekná funkcia

Viktor

6. Idiomatická rekonštrukcia

(max. 12 b za popis, 8 b za program)

Základná myšlienka riešenia

Najprv si predstavme, že jednoducho skladáme reťazec zo zadaných podreťazcov (nazvime ich "kúsky"). Ako to môžeme robiť? Aké máme kedy možnosti, ako v ktorom momente pokračovať? Na začiatku si vieme zvoliť ľubovoľný znak, ktorým nejaký kúsok začína. Potom ľubovoľný znak, ktorým niektorý z takto začínajúcich kúskov pokračuje. Až dosiahneme bod, keď sme napísali nejaký celý kúsok, vieme určite, že ďalšie znaky do toho istého kúsku patriť nemôžu, keďže by sme vyskladali nejaký kúsok, ktorého prefixom je ten kúsok, ktorý sme teraz dokončili…

Ukážka: skladajme od začiatku reťazec z kúskov "ahoj" "aha" a "hanoj". Najprv môžeme začať znakom "a" alebo "h". Zvoľme si "a". Pokračujeme určite znakom "h", teda teraz máme zatiaľ reťazec "ah". Tretím písmenom môže byť "o" alebo "a". Ak si zvolíme "a", vyskladáme hotový reťazec "aha". Vieme, že toto je koniec tohto kúsku, pretože žiaden iný kúsok nemôže začínať prefixom "aha". Teda pokračujeme od začiatku: znova si môžeme vybrať "a" alebo "h". A tak ďalej…

Reprezentácia slovníka

Štruktúra, ktorá nám môže pomôcť pamätať si aké písmenká sme zatiaľ v budovanom kúsku použili, je napríklad trie: strom, ktorého koreň označuje prázdny kúsok (buď keď sme na začiatku skladania alebo keď sme práve dokončili kúsok pred tým). Každý ďalší vrchol v trie označuje reťazec v jeho rodičovi, ku ktorému je na koniec pridaný jeden znak. V trie sú všetky reťazce, ktoré sú prefixom nejakého povoleného kúsku. Teda, keď chceme budovať nejaký reťazec z kúskov, vieme sa jednoducho pohybovať po trie vždy do dieťaťa aktuálneho vrcholu a z listov (dokončených kúskov) sa vrátiť do koreňa.

Skladanie palindrómu

Dobre, čo teraz? Chceme, aby náš reťazec bol nielen vyskladaný z kúskov, ale aj aby bol palindrómom. Prípadne zistiť, že neexistuje. Skladajme reťazec nielen od začiatku, ale aj od konca zároveň. Spravíme dve trie - jedno pre všetky povolené kúsky odpredu a jedno pre povolené kúsky odzadu.

Ako vyzerá skladanie palindrómu? V oboch trie začneme od koreňa - na začiatku ani na konci ešte žiaden znak nie je. Následne vieme pridať na začiatok aj koniec ten istý znak. Teda musí to byť nejaký taký, ktorý vieme v oboch trie pridať - teda oba momentálne vrcholy v nich majú dieťa, kde je pridaný ten istý znak. Keď v nejakom trie narazíme na koniec, vrátime sa hneď do koreňa.

Kedy skončiť

Kedy sme palindróm doskladali? Ako spoznáme, že môžeme začiatok aj koniec palindrómu, ktorý skladáme, spojiť do celku, ktorý bude tiež pozostávať z povolených kúskov? Spravíme trik: budeme hľadať iba palindrómy párnej dĺžky, v ktorých žiadne kúsky neprechádzajú stredom. Teda prvá aj druhá polovica sú samé osebe vyskladané z kúskov. Určite, ak existuje akékoľvek riešenie, existuje aj takéto - jednoducho, akékoľvek riešenie napíšeme dvakrát za seba. Stále bude palindrómom a prvá aj druhá polovica budú vyskladané z kúskov samé osebe. Takéto riešenia sa budú hľadať omnoho jednoduchšie - ak sa vrátime do situácie, že v oboch trie sme v koreni, našli sme ho. Totiž náš budovaný prefix aj suffix sú v tomto momente uzatvorené, a teda ich vieme proste nalepiť za seba a dostať riešenie.

Vyhodnotenie

Už nám teda ostáva len premyslieť si, ako zistiť, či sa takto dá vyskladať riešenie alebo nie. Teda máme niekoľko stavov, každý je popísaný dvojicou vrcholov v oboch trie a chceme zistiť, či sa zo stavu koreň-koreň dá do seba vrátiť. V podstate nám stačí DFS algoritmus zbehnutý na dvojiciach týchto vrcholov. V dvojrozmernom poli si zapamätáme pre každú dvojicu, či sme ju už navštívili. Ak nájdeme spôsob, ako sa tam vrátiť, teda riešenie existuje, tak konkrétne riešenie nájdeme, ak si pre každú dvojicu zapamätáme, z ktorej sme sa na ňu dostali a takto odzadu celú cestu zrekonštruujeme.

Zložitosť

Aké sú zložitosti nášeho programu? Jeho časová zložitosť je $O(n^2)$, pretože na začiatku spravíme obe trie - na to musíme prejsť n slov konštantnej dĺžky a každé z nich vložiť do trie odpredu a odzadu. To trvá len $O(n)$. Potom zbehneme DFS na dvojiciach vrcholov týchto trie. Každé trie má najviac $O(n)$ vrcholov, teda dvojíc je $O(n^2)$ a to je aj zložitosť tohoto DFS. Jeho pamäťová zložitosť je $O(n^2)$, pretože si pamätáme okrem dvoch trie veľkosti $O(n)$ ešte dvojrozmerné pole veľkosti $O(n^2)$.

Listing programu (Python)

```
from string import ascii_lowercase

size = len(ascii_lowercase)


class Node:
    def __init__(self):
        self.children = {}

    def existing_children(self):
        return set(self.children.keys())


def add_word(word: str, root: Node):
    v = root
    for c in word:
        if c not in v.children:
            v.children[c] = Node()
        v = v.children[c]


n = int(input())
words = [input() for _ in range(n)]

prefix = Node()
suffix = Node()

for word in words:
    add_word(word, prefix)
    add_word(word[::-1], suffix)

q = [(prefix, suffix, "")]

while q:
    p, s, w = q.pop(0)

    p_child = p.existing_children()
    if len(p_child) == 0:
        p = prefix
        p_child = p.existing_children()

    s_child = s.existing_children()
    if len(s_child) == 0:
        s = suffix
        s_child = s.existing_children()

    if p == prefix and s == suffix and w != "":
        print(w+w[::-1])
        exit(0)

    child = p_child.intersection(s_child)
    for i in child:
        q.append((p.children[i], s.children[i], w+i))

print(-1)
```

Listing programu (C++)

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct trie_node{
    trie_node* parent;
    trie_node* children[26];
    int depth;
    bool end;
    int id;
};

trie_node* new_node(trie_node* parent, int letter, int id){
    trie_node* o = new trie_node;
    if(parent != NULL){
        parent->children[letter] = o;
        o->depth = parent->depth + 1;
    }else{
        o->depth = 0;
    }
    o->id = id;
    o->parent = parent;
    o->end = false;
    for(int i = 0; i < 26; i++){
        o->children[i] = NULL;
    }
    return o;
}

int save_word(trie_node* where, string s, bool reverse, int offset, int ids){
    if(offset == s.length()){
        where->end = true;
        return ids-1;
    }
    int l;
    if(reverse){
        l = s[s.length()-1-offset]-'a';
    }else{
        l = s[offset]-'a';
    }
    if(where->children[l] == NULL){
        trie_node* o = new_node(where,l,ids);
        ids++;
        ids = save_word(o,s,reverse,offset+1,ids);
    }else{
        ids = save_word(where->children[l],s,reverse,offset+1,ids);
    }
    return ids;
}

trie_node* find_node(trie_node* where,string s,int offset){
    if(offset == s.length()){
        return where;
    }
```

```
    return find_node(where->children[s[offset]-'a'],s,offset+1);

}

char vyries(trie_node* pre,trie_node* post,vector<char>& result,bool** visited,
    ↪ char** goal,trie_node* posl,trie_node* posr,bool ignore = false){

    if(posl->end){
        posl = pre;
    }
    if(posr->end){
        posr = post;
    }

    if(!ignore){
        if(visited[posl->id][posr->id]){
            return 'X';
        }
        visited[posl->id][posr->id] = true;

        if(goal[posl->id][posr->id] != 'X'){
            return goal[posl->id][posr->id];
        }
    }



    for(char i = 'a'; i <= 'z'; i++){
        if(posl->children[i-'a']==NULL or posr->children[i-'a']==NULL){
            continue;
        }
        result.push_back(i);
        char o = vyries(pre,post,result,visited,goal,posl->children[i-'a'],posr
            ↪ ->children[i-'a']);
        if(o != 'X'){
            return o;
        }
        result.pop_back();
    }

    return 'X';

}

int main(){
    int n;
    cin >> n;
    string words[n];
    for(int i = 0; i < n; i++){
        cin >> words[i];
    }

    trie_node *preroot,*postroot;
    preroot = new_node(NULL,0,0);
    postroot = new_node(NULL,0,0);
    preroot->id = 0;
    postroot->id = 0;
    int n1 = 1;
```

```
    int n2 = 1;

    for(int i = 0; i < n; i++){
        string word = words[i];

        n1 = save_word(preroot,word,false,0,n1);
        n2 = save_word(postroot,word,true,0,n2);
    }

    bool* visited[n1];
    char* goal[n1];
    for(int i = 0; i < n1; i++){
        visited[i] = new bool[n2];
        goal[i] = new char[n2];
        for(int j = 0; j < n2; j++){
            visited[i][j] = false;
            goal[i][j] = 'X';
        }
    }
    goal[0][0] = 'O';

    for(int i = 0; i < n; i++){
        string word = words[i];
        trie_node *left,*right;
        left = find_node(preroot,word,0);
        right = postroot;
        for(int j = word.length()-1; j >= 0; j--){
            left = left->parent;
            goal[left->id][right->id] = word[j];
            right = right->children[word[j]-'a'];
            if(j == 0){
                break;
            }
            goal[left->id][right->id] = 'O';
        }
    }


    vector<char> result;
    char o = vyries(preroot,postroot,result,visited,goal,preroot,postroot,true);

    if(o == 'X'){
        cout << -1;
    }else{
        for(int i = 0; i < result.size(); i++){
            cout << result[i];
        }
        if(o != 'O'){
            cout << o;
        }
        for(int i = result.size()-1; i >= 0; i--){
            cout << result[i];
        }
    }
    cout << endl;

    for(int i = 0; i < n1; i++){
        delete visited[i];
```

```
        delete goal[i];
    }

}
```

DávidB

7. Cypriánove záhyby

(max. 12 b za popis, 8 b za program)

Na vyriešenie tejto úlohy bolo potrebné zistiť, kolko štvorčekov sa nachádza "v záhyboch". Existuje na to viacero spôsobov. Jeden z nich je zistiť obsah útvaru na vstupe, potom ho akosi obaliť z vonka, zistiť obsah vrátane záhybov a vypísať ich rozdiel. My si ale ukážeme implementačne jednoduchšie riešenie.

Vstup

Prvá vec, ktorú musíme spraviť asi pri ľubovolnom riešení tejto úlohy, je rozumne načítať vstup. Potrebujeme nejakým spôsobom preložiť reťazec na čísla, teda súradnice. Dobrý spôsob je začať na súradniciach `5000, 5000`, aby bolo všetko kladné, a potom si len pamätať aktuálnu pozíciu a smer, a podla vstupu updatovať a ukladať.

Hladanie záhybov

Aby sme si úlohu zjednodušili, predstavme si, že políčko je v záhybe, iba ak je ohraničené z hora a z dola. S takouto zmenou môžeme potom vstup spracovať po stĺpcoch, nezávysle na sebe. V každom stĺpci si zoradíme horizontálne pohyby od hora dole a následne iba prejdeme tento zoznam. Prvý pohyb znamená vonkajšiu hranicu, teda začína ohraničenú oblasť. Druhý prechod musí teda ukončovať ohraničenú oblasť, čiže začína záhyb. Tretí zase ukončuje záhyb a tak ďalej až po posledný. Prechodov je určite párny počet, lebo rovnako veľa smeruje doprava, ako do ľava. Každá ohraničená oblasť je teda aj z dola uzavretá.

Takýmto spôsobom vieme identifikovať políčka, ktoré sú vertikálnym smerom v záhyboch. Ak rovnaký postup zopakujeme po riadkoch, nájdeme všetky políčka v záhyboch.

Opakujúce sa políčka

Jediný problém tohoto riešenia je, že niektoré políčka zarátame dvojmo - aj pre riadok, aj pre stĺpec. Vzhľadom na veľkosť vstupu toto môžeme vyriešiť jednoducho tak, že si jednotlivé políčka (ich súradníce), vložíme do setu a už len vypíšeme veľkosť tohoto setu. V prípade jazykov, ktoré rýchlo pracujú s poľami, stačí tieto súradnice vyznačovať v dvojrozmernom poli a na konci spočítať.

Časová a pamäťová zložitosť

Dĺžka vstupného reťazca n nám v tomto prípade až tak veľa nehovorí, no pri najmenšom ho musíme načítať. Zadanie nám však dáva oveľa menšie ohraničenie, a to na veľkosť súradníc m. Do celkovej časovej zložitosti teda započítame $O(n)$ za načítanie vstupu a pri najhoršom $O(m^2)$ za prechádzanie mriežky po riadkoch a stĺpcoch. Spolu je to teda $O(n + m^2)$ za predpokladu, že ak použijeme set, tak je to hash-set a predpokladáme, že má konštantné operácie.

Pamäťovú zložitosť môžeme odhadnúť rovnako, pretože si ukladáme pretransformovaný vstup, ktorý je v princípe rovnako veľký ako pôvodný a tiež si ukladáme všetky políčka, ktoré sú v záhyboch, čo je najviac $O(m^2)$.

Listing programu (Python)

```
from collections import defaultdict

dir = [[1,0], [0,1], [-1,0], [0,-1]]

ver = defaultdict(list)
hor = defaultdict(list)

x = 5000
y = 5000
nd = 0

s = input()
```

```
for j in s:
    if j == 'P': nd = (nd + 1) & 3
    elif j == 'L' : nd = (nd + 3) & 3
    elif j == 'R' :
        nx = x + dir[nd][0]
        ny = y + dir[nd][1]
        if (y != ny):
            hor[min(y, ny)].append(x)
        else:
            ver[min(x, nx)].append(y)
        x = nx
        y = ny

policka = set()

for i in ver.keys():
    ver[i].sort()
for i in hor.keys():
    hor[i].sort()


for x in ver.keys():
    j = 1
    while j + 1 < len(ver[x]):
        y1 = ver[x][j]
        y2 = ver[x][j+1]
        for y in range(y1, y2):
            policka.add((x,y))
        j+=2

for y in hor.keys():
    j = 1
    while j + 1 < len(hor[y]):
        x1 = hor[y][j]
        x2 = hor[y][j+1]

        for x in range(x1, x2):
            policka.add((x,y))
        j+=2

print(len(policka))
```

Listing programu (C++)

```
#include <iostream>
#include <set>
#include <vector>
#include <algorithm>
#include <cstring>

using namespace std;

const int dir[4][2] = {{1,0}, {0,1}, {-1,0}, {0,-1}};

vector<int> ver[10000], hor[10000];


int main(){
```

```
int x = 5000, y = 5000, nd = 0;

for (int i = 0; i < 10000; i++)
    ver[i].clear(), hor[i].clear();

char s[70000];
int t;
scanf("%s", s);
for (int j = 0; j < strlen(s); j++)
{

    int nx, ny;
    switch (s[j])
    {
        case 'P' : nd = (nd + 1) & 3; break;
        case 'L' : nd = (nd + 3) & 3; break;
        case 'R' :
                    nx = x + dir[nd][0], ny = y + dir[nd][1];
                    if (y != ny)
                        hor[min(y, ny)].push_back(x);
                    else
                        ver[min(x, nx)].push_back(y);
                        x = nx, y = ny;
                    break;
    }

}

set<int> policka[10000];

for (int i = 0; i < 10000; i++)
    sort(ver[i].begin(), ver[i].end()), sort(hor[i].begin(), hor[i].end());


for (int x = 0; x < 10000; x++)
    for (int j = 1; j + 1 < ver[x].size(); j+=2)
    {
        int y1 = ver[x][j], y2 = ver[x][j+1];
        for (int y = y1; y < y2; y++)
            policka[x].insert(y);
    }



for (int y = 0; y < 10000; y++)
    for (int j = 1; j + 1 < hor[y].size(); j+=2)
    {
        int x1 = hor[y][j], x2 = hor[y][j+1];

        for (int x = x1; x < x2; x++)
            policka[x].insert(y);
    }

    int ret = 0;
    for (int i = 0; i < 10000; i++)
        ret += policka[i].size();

    printf("%d\n", ret);
```

```
    return 0;
}
```

Paulinka

8. Ešte nás vidíš?

(max. 12 b za popis, 8 b za program)

Táto osmička výnimočne je vlastne len o hrubeh sile. A ako ju robiť šikovne. Tak poďme na to.

Skúšame všetky permutácie

Prvé riešenie, ktoré vieme skúsiť je skúšanie všetkých možností. Čo sú v tomto prípade všetky možnosti? Permutácie k-tic. Pre každú k-ticu si vieme zistiť počet rôznofarebných susediacich dvojíc v linárnom čase. Teda vieme všetky možnosti poskúšať v čase $O(nk!)$ a pamäti $O(n+k)$, čo nám stačí na štyri body.

Hľadáme cestu

Na vyriešenie úlohy si úlohu preformulujeme ako úlohu na hľadanie najkratšie hamiltonovského cyklu[5] vo vhodnom grafe.

Prvé pozorovanie je, že si počet odlišných susedov napísať ako súčet hodnôt pre ze sebou idúce pozície, teda ako súčet

- počtu úsekov kde má prvá a druhá ploštica rôznu farbu
- počtu úsekov kde má druhá a tretia ploštica rôznu farbu
- ...
- počtu úsekov kde má predposledná a posledná ploštica rôznu farbu
- počtu úsekov kde má posledná ploštica rôznu farbu od prvej ploštice nasledovného úseku

Druhé pozorovanie, že ak sú pozície i a j spermutované vedľa seba, nezáležiac na tom, na ktorom mieste sa relatívne v k-tici nachádzajú. Túto hodnotu, si vieme spočítať, pre každú dvojicu jednoduchým prechodom cez všetky k-tice, a spočítaním v koľkých z nich majú pôvodne i-ta a j-ta ploštica rôznu farbu.

Rovnako vieme, pre každý pár pozícií i, j, spočítať pre koľko k-tic je i-tá ploštica inej farby ako j-ta ploštica v nasledovnej k-tici.

Takto vlastne dostaneme *kompletný ováhovaný graf* – vrcholy sú pozície, a hrana medzi vrcholny i, j má váhu "počet úsekov kde majú i-ta a j-ta ploštica rôznu farbu".

Permutácia zodpovedá nejakej *hamiltonovskej ceste*^[ceste obsahuj[cej v3etky vrcholy] v tomto grafe, a hodnotu permutácie (výsledný počet rôznofarebných párov) vieme spočítať ako súčet váh použitých hrán a rôznofarebných dvojíc ktoré dostaneme použitím začiatku a konca cesty.

Takto vieme získať riešenie použitím algoritmu na hľadanie najkratšej hamiltonovskej cesty, raz pre každý možný začiatok, teda k-krát.

Vedeli by sme to ešte zrýchliť? Áno – vieme úlohu vyriešiť len s pomocou jediného vyhľadania hamiltonovskej kružnice (tú vieme nájsť v rovnakom čase ako cestu). A to tak, že si do grafu pridáme špeciálne orientované hrany – hrana z i do j indikujúca koľko rôznofarebných dvojíc na "rozmedzí k-tic" dostaneme ak i-ta pozícia bola na konci permutácie a j-ta na začiatku. Následne musíme implementovať len podmienku, že takúto hranu nevieme použiť dvakrát.

Ako na hamiltonovskú kružnicu

Ostáva už len nájsť dostatočne rýchly algoritmus na hľadanie najkratšieho cyklu obsajúceho všetky vrcholy (tak že špeciálne hrany nebudú použité viackrát).

Potrebujeme niečo rýchlejšie ako $k!$. Síce nie polynomiálna[6], ale zložitosť $O(k^2 2^k)$ bude postačovať.

S technika ktorú použijeme sa môžte stretnúť pod menom *bitmasková dynamika*. Ide o dynamické programovanie, v ktorom sú stavy

- v ktorom vrchole sa nachádzame
- ktoré vrcholy sme už videli (reprezentované ako číslo s tým binárnym zápisom)
- použili sme už špeciálnu hranu?

[5] cyklu ktorý obsahuje všetky vrcholy

[6] polynomiálny algorithmus zrejme neexistuje

Všimnite si, že týchto stavov je $2n \cdot 2^n$, a v každom stave sa jednoducho vieme pozrieť na každý vrchol kde sme ešte neboli (normálnou aj špeciálnou hranou ak sme ju ešte nepoužili). Na zistenie finálnej hodnoty, vždy začíname dynamiku iba z jedného konkrétneho vrcholu (permutácia ho nemusí nutne dať na prvú pozíciu), a potom ku každému možnému koncu pripočítať váhu hrany do poťiatočného vrchola (používajúc špeciálnu hranu, ak v ceste nebola použitá).

Takto vieme v našom kompletnom grafe nájsť najkratšiu hamiltonovskú kružnicu s presne jednou použitou špeciálnou hranou v čase $O(k^2 2^k)$.

Graf vieme vybudovať v čase $O(k^2(n/k)) = O(nk)$, a teda celková časová zložitosť algoritmu je $O(nk+k^2 2^k)$. Pamätať si potrebujeme vstup, graf a hodnoty dynamiky, teda je pamäťová zložitosť $O(n + k \cdot 2^k)$.

Listing programu (Python)

```
#!/usr/bin/python3

def constr_graph(k, s):
    m = len(s) // k
    E = [[len(s) for _ in range(k)] for _ in range(k)]
    se = [[len(s) for _ in range(k)] for _ in range(k)]

    for i in range(k):
        for j in range(k):
            if i == j:
                continue;
            E[i][j] = 0
            se[i][j] = 0
            for l in range(m):
                if s[l * k + i] != s[l * k + j]:
                    E[i][j] += 1
                if l > 0 and s[l * k + i] != s[(l - 1) * k + j]:
                    se[i][j] += 1
    return (E, se)

def solve(k, s):
    E, se = constr_graph(k, s)

    # print(E, se)

    dp = [[[len(s) for _ in range(2)] for _ in range(k)] for _ in range(1 << k)]

    dp[1][0][0] = 0

    for a in range(1 << k):
        for v in range(k):
            if ((1 << v) & a) == 0 or (a % 2 == 0):
                continue
            # print(a, v, dp[a][v])
            for w in range(k):
                if ((1 << w) & a) == 0:
                    # print(w)
                    dp[a | (1 << w)][w][0] = min(dp[a | (1 << w)][w][0], dp[a][v
                        ↪ ][0] + E[v][w])
                    dp[a | (1 << w)][w][1] = min(dp[a | (1 << w)][w][1], dp[a][v
                        ↪ ][1] + E[v][w], dp[a][v][0] + se[w][v])
    res = len(s)
    en = (1 << k) - 1

    for v in range(k):
        res = min(res, dp[en][v][1] + E[v][0], dp[en][v][0] + se[0][v])
```

```
    print(res)

k = int(input())

solve(k, input())
```

Listing programu (C++)

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

using namespace std;

#define FOR(i,n)  for(int i=0;i<(int)n;i++)

typedef complex<long double> point;

template<class T>
T get() {
    T a;
    cin >> a;
    return a;
}

template <class T, class U>
ostream& operator<<(ostream& out, const pair<T, U> &par) {
    out << "[" << par.first << ";" << par.second << "]";
    return out;
}

template <class T>
ostream& operator<<(ostream& out, const set<T> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";
    out << "}";
    return out;
}

template <class T, class U>
ostream& operator<<(ostream& out, const map<T,U> &cont) {
    out << "{";
    for (const auto &x:cont) out << x << ", ";
    out << "}"; return out;
}

template <class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
    FOR(i, v.size()){
        if(i) out << " ";
        out << v[i];
    }
    out << endl;
    return out;
}

bool ccw(point p, point a, point b) {
    if((conj(a - p) * (b - p)).imag() <= 0) return false;
    else return true;
}
```

```
pair<vector<vector<int> >, vector<vector<int> > > construct_graph(int k, string
    ↪ &s) {
    vector<vector<int> > E(k, vector<int>(k, 0)), start(k, vector<int>(k, 0));
    int m = s.size() / k;
    FOR (i, k) {
        E[i][i] = s.size();
        start[i][i] = s.size();
        FOR (j, k) {
            if (i == j) continue;
            FOR (l, m) {
                if (s[l * k + i] != s[l * k + j]) E[i][j] ++;
                if (l && s[l * k + i] != s[(l - 1) * k + j]) {
                    start[i][j] ++;
                }
            }
        }
    }
    return {E, start};
}

void solve() {
    int k = get<int>();
    string s = get<string>();
    int n = s.size();

    auto blah = construct_graph(k, s);
    auto E = blah.first, start = blah.second;

    int res = n;

    vector<vector<vector<int> > > dp(k, vector<vector<int> >(1 << k, vector<int
        ↪ >(2, n)));

    dp[0][1][0] = 0;
    for (int a = 1; a < (1 << k); a ++) {
        FOR(j, k) {
            FOR (b, 2) {
                FOR(l, k) {
                    if (a & (1 << l)) continue;
                    dp[l][a | (1 << l)][b] = min(dp[l][a | (1 << l)][b], dp[j][a
                        ↪ ][b] + E[j][l]);
                    if (!b) dp[l][a | (1 << l)][1] = min(dp[l][a | (1 << l)][1],
                        ↪  dp[j][a][b] + start[l][j]);
                }
            }
        }
    }
    FOR(j, k) {
        res = min(res, dp[j][(1 << k) - 1][0] + start[0][j]);
        res = min(res, dp[j][(1 << k) - 1][1] + E[j][0]);
    }
    cout << res << endl;


}

int main() {
```

```
    cin.sync_with_stdio(false);
    cout.sync_with_stdio(false);
    solve();
}
```