Riešenia kategórie A

A-I-1 Hotel

Počas riešenia našej úlohy sa musíme rozhodovať: vždy, keď príde nový hosť, musíme sa rozhodnúť, na ktoré poschodie ho ubytujeme.

V prvom rade sa zamyslime, či na našich rozhodnutiach vôbec záleží: nie je odpoveď náhodou vždy rovnaká? Tu je zjavná odpoveď že áno, záleží. Ak by sme napríklad mali na každom poschodí jednu izbu a príde nám prvý hosť s $v_1 = 2$ a potom druhý hosť s $v_2 = 1$, potrebujeme prvého hosťa umiestniť na druhé poschodie. Potom totiž môžeme ubytovať ešte aj druhého hosťa na prvom poschodí. Naopak by to nefungovalo: ak by sme prvého hosťa ubytovali na prvom poschodí, pre druhého by sme už nemali voľnú izbu.

Tento príklad nám zároveň naznačuje, že asi bude vo všeobecnosti lepšie ubytúvať ľudí čo najvyššie. Intuitívne: ak v budúcnosti príde vyšší hosť, tomu je to jedno, a ak príde nižší, tak mu nezavadziame. Toto tvrdenie si teraz poriadne sformulujeme a dokážeme, že platí.

Veta: Ak každého hosťa ubytujeme na najvyššom dostupnom poschodí tak zaručene zostrojíme optimálne riešenie. (Inými slovami, ak pre každého hosťa použijeme najvyššie položenú voľnú izbu v ktorej môže bývať, tak sa nám podarí ubytovať najväčší možný počet ľudí.)

Túto vetu si dokážeme tak, že ukážeme, že ľubovoľné optimálne riešenie vieme prerobiť na rovnako dobré riešenie, v ktorom všetky rozhodnutia robíme podľa vyššie uvedeného pravidla.

Predstavme si teda, že máme ľubovoľné optimálne riešenie. Ak sa pri ňom vždy dodržalo naše pravidlo, sme hotoví. Ak nie, nájdime prvého hosťa X, pri ktorom sme ho porušili – teda ubytovali sme ho na poschodí x v izbe I_x, hoci na poschodí y (kde $y > x$) sme ešte mali voľnú izbu I_y v ktorej mal bývať podľa nášho pravidla.

Pozrime sa na to, čo sa v optimálnom riešení stalo s izbou I_y. Ak ostala do konca prázdna, môžeme naše riešenie upraviť tak, že hosťa X ubytujeme v izbe I_y a ostane prázdna izba I_x. A čo ak sme v izbe I_y neskôr ubytovali nejakého iného hosťa Y? Keďže tento hosť je dosť vysoký na to aby býval v izbe I_y, je dosť vysoký aj na to, aby býval v izbe I_x. To ale znamená, že môžeme týchto dvoch hostí jednoducho vymeniť: hosť X pôjde pekne do izby I_y kam patrí, a keď neskôr príde hosť Y, ten dostane voľnú izbu I_x.

Nech začneme z ľubovoľného optimálneho riešenia, opakovaním vyššie uvedenej úvahy ho v konečnom počte krokov zjavne prerobíme na rovnako dobré (a teda tiež optimálne) riešenie, v ktorom každého hosťa ubytujeme podľa nášho pravidla.

Implementácia pomocou usporiadanej množiny

Z tvrdenia, ktoré sme si práve dokázali, vyplýva, že zadanú úlohu vieme riešiť *pažravo*: každého hosťa vieme rovno ubytovať bez toho, aby sme sa museli pozerať na to, akí ďalší hostia nám prídu v budúcnosti. Jediné, čo si ešte potrebujeme rozmyslieť, je efektívna implementácia. Potrebujeme si udržiavať údaje o izbách v nejakej vhodnej dátovej štruktúre, ktorá nám umožní rýchlo nájsť izbu, do ktorej chceme ubytovať práve spracúvaného hosťa.

Potrebujeme teda dátovú štruktúru, ktorá bude poskytovať nasledovnú operáciu: *nájdi* najvyššiu z prázdnych izieb na poschodiach 1 až v_i a následne nájdenú izbu *odstráň* z množiny prázdnych izieb.

Z implementačného hľadiska je asi najjednoduchším riešením použiť niektorú z dátových štruktúr určených na reprezentovanie usporiadanej množiny. V tých vieme aj vyhľadávať aj mazať v čase logaritmickom od počtu uložených prvkov.

V nižšie uvedenom programe sme použili dátovú štruktúru `multiset` v C++. Na začiatku sme do nej vložili všetkých pn izieb (teda n kópií každého z čísel od 1 po p). No a následne sme postupne pre každého hosťa pomocou metódy `upper_bound` našli najlepšiu izbu. Každá operácia s touto dátovou štruktúrou beží v čase $O(\log(pn))$, celková časová zložitosť je teda $O(pn \log(pn))$.

Ešte o chlp lepšiu časovú zložitosť $O(pn \log p)$ by sme vedeli dosiahnuť použitím dátovej štruktúry `map`, v ktorej by sme si pre každé poschodie, ktoré ešte nie je plne obsadené, pamätali počet voľných izieb na ňom. Takúto istú časovú zložitosť vieme dosiahnuť aj použitím intervalového stromu, v ktorom listy zodpovedajú

jednotlivým poschodiam a v ktorom si v každom vnútornom vrchole pamätáme index najvyššieho nie-úplne-plného poschodia v príslušnom podstrome.

Listing programu (C++)

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

int main() {
    // načítame údaje o hoteli a počte hostí
    int P, N, H;
    cin >> P >> N >> H;

    // vygenerujeme si všetky prázdne izby v hoteli
    multiset<int> hotel;
    for (int p=1; p<=P; ++p) for (int n=0; n<N; ++n) hotel.insert(p);

    // postupne čítame a spracúvame hostí, kým sa neminú alebo sa nezasekneme
    vector<int> riesenie;
    while (H--) {
        int V; cin >> V;
        auto it = hotel.upper_bound(V); // "it" ukazuje na najnižšiu NEvhodnú izbu
        if (it == hotel.begin()) break; // nemáme pod ňou žiadnu vhodnú voľnú izbu? -> koniec
        --it;                           // teraz "it" ukazuje na správnu izbu
        riesenie.push_back(*it);        // tú si zapamätáme v riešení...
        hotel.erase(it);                // ... a odstránime ju z množiny voľných izieb
    }

    // vypíšeme riešenie
    cout << riesenie.size() << "\n";
    for (unsigned i=0; i<riesenie.size(); ++i) cout << riesenie[i] << (i+1==riesenie.size() ? "\n" : " ");
}
```

Ešte lepšie riešenie

Predchádzajúce riešenie stačilo na zisk plného počtu bodov. Našu úlohu však vieme riešiť ešte o čosi efektívnejšie: v čase $O(pn \cdot \alpha(pn))$, kde α označuje inverznú Ackermannovu funkciu.[1] Uvedieme základnú myšlienku tohto riešenia.

Predstavme si, že sme všetky izby v hoteli uložili do radu usporiadané podľa poschodia. Ako ubytúvame hostí, izby postupne obsadzujeme. Tým nám časom začnú vznikať celé úseky v ktorých sú všetky izby obsadené. Čo sa deje, keď chceme ubytovať ďalšieho hosťa? Pozrieme sa na najvyššie poschodie, kde ešte môže bývať. Ak tam vidíme voľnú izbu, je dobre. No a ak sme tam narazili na nejaký úsek obsadených izieb, zaujíma nás, kde tento úsek začína – bezprostredne pred ním je totiž izba, kde chceme nášho hosťa ubytovať.

V našom riešení použijeme na reprezentáciu jednotlivých úsekov obsadených izieb algoritmus Union-Find. Ku každému úseku si navyše zapamätáme, ktorou izbou začína. Operáciou Find vieme nájsť k optimálnej hosťovej izbe celý úsek obsadených izieb v ktorom leží. Operácie Union použijeme vždy po obsadení izby na jej pripojenie k úsekom ležiacim bezprostredne pred a za ňou (ak už sú izby pred/za ňou tiež obsadené).

A-I-2 Žabka

Úlohu samozrejme vieme riešiť hrubou silou: rekurzívne skúšať všetky možnosti ako mohla žabka postupne skákať. Tých však môže byť strašne veľa[2] a teda takéto riešenie bude použiteľné len pre veľmi malé n.

Kubické riešenie

Neefektívnosť predchádzajúceho riešenia spočíva v tom, že zbytočne znova a znova skúša tie isté postupnosti skokov. Ako to vieme zlepšiť?

Predstavme si, že žabka už spravila niekoľko skokov a zaujíma nás, ako najlepšie má z tej situácie pokračovať ďalej do cieľa. Kľúčové pozorovanie: Odpoveď na túto otázku *takmer nezávisí* od toho, ako žabka skákala doteraz.

[1] Táto funkcia rastie tak pomaly, že pre všetky praktické potreby je tento algoritmus lineárny od počtu izieb v hoteli.
[2] Koľko vlastne? Exponenciálne od n, či ešte viac? Skúste si to rozmyslieť.

Závisí len od posledného skoku, ktorý spravila. Tým je totiž presne určené aj kde sa teraz nachádza, aj aké skoky môže spraviť počas zvyšku svojej púte.

Zaujímajú nás teda otázky nasledovného tvaru: „Ak žabka práve skočila z kameňa a na kameň b, koľko najviac skokov ešte môže spraviť na ceste do cieľa?“

To je $O(n^2)$ rôznych otázok. Každú otázku vieme zodpovedať tak, že rekurzívne vyskúšame $O(n)$ možností, kam žabka skočí v nasledovnom skoku. Pre každú možnosť tak dostaneme jednu novú otázku, na ktorej zodpovedanie použijeme rekurzívne volanie.

Ak by sme len priamo implementovali vyššie popísané riešenie, dostali by sme presne vyššie popísané riešenie používajúce hrubú silu. My však môžeme použiť *memoizáciu*: Odpoveď na každú otázku budeme počas behu počítať len raz. Keď nejakú odpoveď vypočítame, zaznačíme si ju do tabuľky. Ak sa rovnaká otázka vyskytne neskôr počas behu programu, rovno vrátime uloženú hodnotu – teda nerobíme už žiadne skúšanie možností ani žiadne rekurzívne volania.

Takto upravený program pre každú otázku najviac $1\times$ strávi $O(n)$ času výpočtom jej odpovede. Dokopy teda takto upravený program spraví $O(n^3)$ krokov. Samotným riešením úlohy je hodnota $1 +$ maximum z odpovedí na otázky v ktorých je $a = 1$.

Listing programu (C++)

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

typedef pair<long long, long long> kamen;

int N;
vector<kamen> rybnik;
vector< vector<int> > memo;

long long square_dist(int a, int b) {
    // štvorec vzdialenosti medzi kameňmi a, b
    long long dx = rybnik[a].first - rybnik[b].first;
    long long dy = rybnik[a].second - rybnik[b].second;
    return dx*dx + dy*dy;
}

int otazka(int a, int b) {
    // ak sme už túto otázku niekedy dostali, rovno vrátime odpoveď na ňu
    if (memo[a][b] != INT_MIN) return memo[a][b];

    // inicializujeme odpoveď: ak sme práve v cieli, je to 0 (môžeme tu skončiť), inak -nekonečno
    int odpoved = (b==N-1 ? 0 : -47);

    // vyskúšame všetky možnosti kam skočiť ďalej a vyberieme najlepšiu
    for (int c=0; c<N; ++c) {
        if (c == b) continue; // nemôžeme ostať na mieste
        if (square_dist(a,b) <= square_dist(b,c)) continue; // musíme spraviť kratší skok
        int moznost = otazka(b,c);
        if (moznost == -47) continue; // po tomto skoku sa nedá dostať do cieľa
        odpoved = max( odpoved, 1+moznost );
    }

    // zapamätáme si vypočítanú odpoveď a vrátime ju na výstup
    memo[a][b] = odpoved;
    return odpoved;
}

int main() {
    // načítame vstup
    cin >> N;
    for (int n=0; n<N; ++n) {
        long long x,y; cin >> x >> y;
        rybnik.push_back( {x,y} );
    }

    // inicializujeme pamäť
    memo.resize( N, vector<int>(N,INT_MIN) );

    // vyskúšame všetky možnosti pre prvý skok a vyberieme najlepšiu z nich
    int odpoved = 0;
    for (int b=1; b<N; ++b) odpoved = max( odpoved, 1+otazka(0,b) );
    cout << odpoved << endl;
}
```

Približne kvadratické riešenie

Predchádzajúce riešenie vieme ešte zlepšiť. Ušetriť sa dá totiž na tom, že v riešení s kubickou časovou zložitosťou pri každej otázke prezeráme všetky možné ďalšie skoky, a to vrátane takých, ktoré už v danej situácii spraviť nemôžeme. Ako teda spracúvať skoky šikovnejšie?

Všetky možné skoky si roztriedime na niekoľko kôpok podľa dĺžky akú majú. (Presnejšie, v programe použijeme štvorec dĺžky, keďže ten vieme uložiť do celočíselnej premennej.) Samotné kôpky si potom podľa dĺžky usporiadame, začínajúc najväčšou.

Pre každý kameň si budeme pamätať, na koľko *najviac* skokov sa naň vieme dostať. Na začiatku je to 0 pre štart (tam začíname) a $-\infty$ pre každý iný kameň (tam zatiaľ nevieme byť). Postupne budeme teraz skúšať použiť skoky v poradí od najdlhších po najkratšie, a zakaždým prepočítame len tie údaje, ktoré sa týkajú práve spracúvaných skokov.

Ako vyzerá spracovanie nového skoku? Veľmi jednoducho: ak ide o skok z kameňa a na kameň b, tak sa pozrieme, na koľko najviac skokov sme sa vedeli (dlhšími skokmi) dostať na kameň a. Nech je tento počet rovný k. Na kameň b sa teraz vieme dostať na $k+1$ skokov. Ak bola doteraz pamätaná hodnota pre kameň b menšia, zväčšíme ju na $k+1$.

Ak máme viac skokov rovnakej dĺžky, treba si dať pozor na to, aby sme ich všetky spracovali naraz – aby sa nám nestalo, že žabka spraví viac rovnako dlhých skokov po sebe.

Časová zložitosť tohto riešenia je $O(n^2 \log n)$. Jeho najpomalšou časťou je usporiadanie všetkých skokov podľa dĺžky. Samotné spracovanie skokov je už efektívnejšie, vieme ho spraviť v konštantnom čase na skok, teda dokopy v $O(n^2)$.

Listing programu (Python)

```
from collections import defaultdict
def square_dist(A,B): return (A[0]-B[0])**2 + (A[1]-B[1])**2

# načítame vstup

N = int( input() )
rybnik = [ tuple( int(x) for x in input().split() ) for n in range(N) ]

# vygenerujeme všetky možné skoky, roztriedime si ich podľa dĺžky, dĺžky usporiadame

skoky = defaultdict(list) # pre každú dĺžku zoznam skokov

for a in range(N):
    for b in range(N):
        if a != b:
            d = square_dist( rybnik[a], rybnik[b] )
            skoky[d].append( (a,b) )

dlzky = reversed( sorted( skoky.keys() ) )

# postupne prechádzame dĺžky a updatujeme si kde vieme byť na koľko skokov

najviac = [ -2**30 for n in range(N) ]
najviac[0] = 0

for d in dlzky:
    # zo starých informácií vypočítame nové
    nove_info = []
    for a,b in skoky[d]: nove_info.append( ( b, najviac[a]+1 ) )
    # zapíšeme tie nové informácie ktoré sú lepšie ako staré
    for b,s in nove_info: najviac[b] = max( najviac[b], s )

print( najviac[N-1] )
```

A-I-3 Triediaca hra

Riešenie pre k=1

Pre $k = 1$ je situácia jednoduchá: každá hra sa dá vyhrať, kamienky totiž jednoducho postupne po jednom poposúvame kam patria. Začneme tým, že kamienok číslo 1 necháme na mieste. Na kamienok číslo 2 budeme

klikať dovtedy, kým sa nedostane vedľa kamienku číslo 1. To isté potom spravíme postupne s kamienkami 3 až $n-1$. Keďže sa kamienok po každom kliknutí posunie len o 1, správne miesto nikdy nepreskočíme.

Riešenie pre k=3

Ukážeme, že každá takáto hra sa dá vyhrať – a to tak, že ju prevedieme na hru predchádzajúceho typu. Na to nám stačí nájsť postupnosť ťahov, ktorá vymení dva susedné kamienky.

Keďže $n > k+1$, máme aspoň 5 kamienkov. Niektorých päť po sebe idúcich kamienkov si označme $ABCDE$. Všimnime si teraz, čo sa stane, keď postupne klikneme na kamienky D, E, B, C, A. Zvyšok kruhu táto postupnosť kliknutí vôbec nezmení. Všetky zmeny budú len preusporadúvať našu päticu kamienkov, a to nasledovne:

$$ABCDE \to DABCE \to DEABC \to BDEAC \to BCDEA \to BACDE$$

Vidíme, že po tejto postupnosti ťahov sú všetky kamienky na pôvodných miestach, len susedné kamienky A a B si miesto vymenili. Na hru pre $k = 3$ teda môžeme použiť algoritmus pre $k = 1$, len na každú výmenu dvoch susedných kamienkov budeme namiesto jedného kliknutia potrebovať spraviť kliknutí až päť.

(Zamyslite sa, čo by sa stalo, keby kamienky mohli byť len štyri. Šlo by aj vtedy každú hru vyhrať?)

Riešenie pre k=2 a párne n

Pre $k = 2$ vieme vyhrať všetky hry, v ktorých je počet kamienkov párny. Nech totiž $n = 2x$. Všimnime si ľubovoľný kamienok. Čo sa stane, keď naň x-krát po sebe klikneme? Zakaždým predbehne dva iné, a teda dokopy predbehne $2x - 2$ zo zvyšných $2x - 1$ kamienkov. Inými slovami, práve sa nachádza o pozíciu skôr ako bol na začiatku – čiže výsledný efekt je ten istý, ako keby sme ho vymenili s kamienkom ktorý bol na začiatku vedľa neho (v smere hodinových ručičiek). A teda opäť vieme použiť priamočiare riešenie pre $k = 1$, len sa viac naklikáme.

Takmer-riešenie pre k=2 a nepárne n

Pre $k = 2$ a nepárne n použijeme trochu iný postup riešenia. Predstavme si, že sme už uložili niekoľko prvých kamienkov vedľa seba a teraz k nim chceme dostať ďalší. Postupne po obvode máme teda kamienky vyzerajúce napr. takto: $(1, 2, 3, 4, 5, x, x, x, 6, y, y)$.

Kým máme aspoň dva kamienky označené x (teda pred kamienkom ktorý chceme práve dostať na správne miesto), môžeme klikať na náš kamienok – v príklade teda na kamienok 6. Ak potom ostane práve jeden kamienok x, odstránime ho tak, že naň budeme klikať, až kým sa nedostane medzi kamienky označené y.

Vyššie uvedený postup zjavne funguje kým máme aspoň dva neumiestnené kamienky. Ako však zistíme, občas nastane na konci problém: napríklad pre $n = 5$ sa nám môže stať, že po umiestnení kamienka 3 dostaneme poradie $(1, 2, 3, 5, 4)\ldots$ ale kamienky 4 a 5 sa nám už vymeniť nepodarí. V poslednej časti tohto vzorového riešenia si dokážeme, že je to naozaj tak – aj keby sme ťahali úplne ináč, takúto hru sa vyhrať nedalo.

Riešenie pre k=2 a nepárne n

Majme postupnosť navzájom rôznych čísel. Inverziou voláme takú dvojicu čísel, v ktorej je väčšie naľavo od menšieho. Napr. postupnosť $(1, 2, 5, 3, 4)$ má dve inverzie: kamienok 5 je naľavo od kamienka 3, a zároveň kamienok 5 je naľavo od kamienka 4. O inverziách toho vieme dosť povedať:

Výmena dvoch po sebe idúcich prvkov postupnosti vždy zmení paritu počtu inverzií.
Toto je očividné: buď jednu inverziu odstránime, alebo jednu pridáme.

Výmena prvého prvku s posledným zmení paritu počtu inverzií.
Toto ľahko dokážeme nasledovne: Majme postupnosť dĺžky n. Označme si prvý prvok A a posledný prvok B. Výmenu A a B vieme spraviť tak, že najskôr $(n-1)$-krát vymeníme A s prvkom ktorý je napravo od neho (čím A presunieme až za B na koniec) a následne $(n-2)$-krát vymeníme B s prvkom naľavo od neho.

Dokopy sme teda $(2n-3)$-krát vymenili dva susediace prvky. Každá výmena nám zmenila paritu počtu inverzií. A keďže $2n-3$ je nepárne, na konci je parita počtu inverzií opačná ako na začiatku.

Ak v postupnosti nepárnej dĺžky presunieme prvý prvok na koniec, parita počtu inverzií sa nezmení.
Zafunguje rovnaký argument: tento presun vieme realizovať pomocou $n-1$ výmen susediacich prvkov.

Na kruhu je napísaná postupnosť nepárnej dĺžky n. Nech ju začneme čítať (v smere ručičiek) od ľubovoľného jej členu, vždy dostaneme postupnosť dĺžky n s tou istou paritou počtu inverzií.
Ľubovoľnú takúto postupnosť vieme vyrobiť z ľubovoľnej inej tak, že niekoľkokrát presunieme prvý prvok na koniec – a o tom už vieme, že nám to paritu počtu inverzií nezmení. Odteraz teda budeme hovoriť priamo o parite počtu inverzií cyklickej postupnosti (nepárnej dĺžky) a budeme tým myslieť paritu počtu inverzií ľubovoľnej z postupností, ktoré vieme dostať jej „rozstrihnutím“.

V našej hre pre nepárne n a pre $k = 2$ žiaden ťah nemení paritu počtu inverzií.
Každý ťah zodpovedá dvom postupným výmenám susediacich prvkov.

A to už je všetko. V predchádzajúcej časti sme ukázali algoritmus, ktorý pre nepárne n a pre $k = 2$ ľubovoľnú pozíciu prerobí buď na pozíciu $(1, 2, 3, \ldots, n-2, n-1, n)$ alebo na pozíciu $(1, 2, 3, \ldots, n-2, n, n-1)$. A teraz už navyše vieme, že to, ktorá z nich to bude, závisí od toho, akú paritu počtu inverzií mala pozícia, z ktorej sme začínali. Ak párnu, hru vyhráme, ak nepárnu, vieme, že hru sa vyhrať nedá, lebo po každom ťahu, nech by bol akýkoľvek, bude parita počtu inverzií opäť nepárna – a teda nikdy nedosiahneme usporiadanú postupnosť, pre ktorú je počet inverzií 0.

Algoritmus

Poslednou časťou nášho riešenia bude efektívny algoritmus, ktorý pre daný popis pozície povie, či je riešiteľná. Jedinou netriviálnou časťou riešenia je tá, ktorou sme ukončili vyššie uvedený popis: pre $k = 2$ a nepárne n potrebujeme vedieť k danej postupnosti zistiť paritu počtu jej inverzií.

To sa samozrejme dá spraviť v čase $\Theta(n^2)$ tak, že pre každú dvojicu prvkov zistíme, či tvorí inverziu. Existujú však aj efektívnejšie algoritmy. Počet inverzií vieme spočítať v čase $\Theta(n \log n)$. Jedným možným postupom je použiť algoritmus MergeSort na usporiadanie danej postupnosti a navyše vždy, keď v časti Merge spájame dve usporiadané postupnosti do jednej, rátať koľko inverzií odstraňujeme keď menší prvok z pravej časti poľa „predbehne“ väčšie z ľavej časti.

Na záver dodáme, že existuje aj lineárne riešenie našej úlohy. To je založené na inom *invariante*: každá výmena dvoch prvkov v permutácii zmení paritu počtu jej *cyklov*, no a počet cyklov vieme ľahko zistiť v lineárnom čase.

A-I-4 Sufixové stromy

Podúloha A: počet rôznych písmen

Každým písmenom reťazca *začína* jeden zo sufixov. No a začiatočné písmená sufixov zodpovedajú práve hranám vedúcim z koreňa sufixového stromu. Počet rôznych písmen v reťazci teda vieme určiť v konštantnom čase: ako stupeň koreňa.

```
print( len( strom.koren.deti ) )
```

Podúloha B: najdlhší opakujúci sa reťazec

Zoberme si ľubovoľné dva výskyty toho istého reťazca T v reťazci S. Každým výskytom začína jeden zo sufixov reťazca S. V sufixovom strome teda vieme začať v koreni a podľa písmen reťazca T sa hýbať dodola. Pod miestom, kde skončíme, sa budú nachádzať konce oboch sufixov.

A naopak, všimnime si ľubovoľné miesto v sufixovom strome, pod ktorým sú konce aspoň dvoch sufixov. Reťazec, ktorý nás z koreňa dovedie na toto miesto, sa zjavne v pôvodnom reťazci nachádza aspoň dvakrát – každý zo sufixov, ktoré ním začínajú predstavuje jeden jeho výskyt.

Najdlhší opakujúci sa reťazec teda nájdeme tak, že v sufixovom strome nájdeme najhlbší vrchol, v ktorom alebo pod ktorým sa nachádzajú konce aspoň dvoch sufixov. Na to stačí použiť funkciu `spocitaj_konce` zo študijného textu a následne raz prejsť celý strom napríklad prehľadávaním do hĺbky.

Nižšie uvedený program nájde dĺžku najdlhšieho opakujúceho sa podreťazca. Na jeho zostrojenie by sme následne mohli použiť funkciu podobnú `najhlbsi_s_dvoma` ktorá však dostane na vstupe aj hľadané optimum a na výstupe vráti priamo reťazec príslušnej dĺžky.

```
infinity = float('inf')

def najhlbsi_s_dvoma(vrchol):
    if vrchol.data < 2: return -infinity
    odpoved = 0
    for hrana in vrchol.deti.values():
        dlzka = hrana.po - hrana.od
        odpoved = max( odpoved, (hrana.po - hrana.od) + najhlbsi_s_dvoma(hrana.kam) )
    return odpoved

def najdlhsi_repeat(S):
    strom = vyrob_strom(S)
    spocitaj_konce( strom.koren )
    return max( 0, najhlbsi_s_dvoma( strom.koren ) )
```

Podúloha C: počet rôznych podreťazcov

Tu nám opäť pomôže podobná úvaha: Každým podreťazcom začína niektorý sufix, a preto každému podreťazcu zodpovedá cesta z koreňa sufixového stromu dodola.

Keby sufixový strom nebol komprimovaný, teda keby na každej hrane bolo napísané len jedno písmeno, vedeli by sme počet rôznych podreťazcov vypočítať ako počet vrcholov v sufixovom strome. (Nerátajúc koreň – ten zodpovedá prázdnemu reťazcu, ktorý podľa zadania nerátame.) Každému vrcholu zodpovedá iný podreťazec – ten, ktorý je napísaný na ceste z koreňa doň.

Aj v samotnom sufixovom strome vieme túto informáciu ľahko získať: jednoducho sčítame dĺžky všetkých hrán sufixového stromu. Hrane dĺžky k v sufixovom strome totiž v jeho nekomprimovanej verzii zjavne zodpovedá práve k vrcholov.

```
def sucet_dlzok_hran(vrchol):
    odpoved = 0
    for hrana in vrchol.deti.values():
        odpoved += (hrana.po - hrana.od) + sucet_dlzok_hran( hrana.kam )
    return odpoved

def pocet_podretazcov(S):
    strom = vyrob_strom(S)
    return sucet_dlzok_hran( strom.koren )
```

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Askar Gafurov
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2015