P-I-1 Hotel

Během řešení naší úlohy se musíme opakovaně rozhodovat: pokaždé, když přijde nový host, musíme rozhodnout, ve kterém poschodí ho ubytujeme.

Nejprve se zamyslíme, zda na našich rozhodnutích vůbec záleží. Není náhodou výsledek pokaždé stejný? Snadno ukážeme, že na volbě poschodí opravdu záleží. Kdybychom například měli v každém poschodí jeden pokoj a přišel by první host s $v_1 = 2$ a potom druhý host s $v_2 = 1$, musíme prvního hosta umístit na druhé poschodí. Potom totiž budeme moci ubytovat také druhého hosta v prvním poschodí. Naopak by to nefungovalo – pokud bychom prvního hosta ubytovali v prvním poschodí, pro druhého bychom již neměli žádný volný pokoj.

Tento příklad nám zároveň naznačuje, že asi bude obecně lepší ubytovávat hosty vždy co možná nejvýše. Přijde-li v budoucnu vyšší host, je mu to jedno, ale když přijde nižší host, nebudeme mu překážet. Toto tvrzení si nyní pořádně zformulujeme a dokážeme, že platí.

> *Věta: Jestliže každého hosta ubytujeme v nejvyšším dostupném poschodí, pak určitě sestrojíme optimální řešení.* Jinými slovy řečeno, jestliže pro každého hosta použijeme nejvýše položený volný pokoj, v němž může bydlet, potom se nám podaří ubytovat nejvyšší možný počet lidí.

Větu dokážeme tak, že libovolné optimální řešení převedeme na stejně dobré řešení, ve kterém všechna rozhodnutí provádíme podle výše uvedeného pravidla.

Představte si, že máme libovolné optimální řešení. Pokud se v něm vždy dodržuje naše pravidlo, jsme hotoví. Pokud ne, určíme prvního hosta X, u kterého jsme pravidlo porušili – tedy ubytovali jsme ho na poschodí x v pokoji I_x, ačkoliv na poschodí y (kde $y > x$) jsme ještě měli volný pokoj I_y, v němž měl host X bydlet podle našeho pravidla.

Podíváme se, co se v optimálním řešení stalo s pokojem I_y. Jestliže zůstal až do konce prázdný, můžeme naše řešení upravit tak, že hosta X ubytujeme v pokoji I_y a zůstane nám volný pokoj I_x. A co když jsme v pokoji I_y později ubytovali nějakého jiného hosta Y? Když je tento host dost vysoký na to, aby bydlel v pokoji I_y, je jistě dost vysoký i na to, aby bydlel v pokoji I_x. To ovšem znamená, že můžeme tyto dva hosty jednoduše vyměnit: host X půjde do pokoje I_y, kam patří, a až později přijde host Y, dostane volný pokoj I_x.

Když vyjdeme od libovolného optimálního řešení, opakováním výše uvedené úvahy ho v konečném počtu kroků zjevně převedeme na stejně dobré (a tedy také optimální) řešení, v němž každého hosta ubytujeme podle našeho pravidla.

Implementace pomocí uspořádané množiny

Z právě dokázaného tvrzení vyplývá, že zadanou úlohu můžeme řešit *hladově:* každého hosta dokážeme rovnou ubytovat, aniž bychom se potřebovali dívat, jací

další hosté přijdou v budoucnosti. Zbývá pouze navrhnout efektivní implementaci. Potřebujeme si udržovat údaje o pokojích v nějaké vhodné datové struktuře, která nám umožní rychle nalézt pokoj, kam chceme ubytovat právě zpracovávaného hosta.

Potřebujeme tedy datovou strukturu, která bude poskytovat následující operaci: *najdi* nejvyšší z prázdných pokojů na poschodích 1 až v_i a následně nalezený pokoj *odstraň* z množiny prázdných pokojů.

Z implementačního hlediska je asi nejjednodušším řešením použít některou z datových struktur určených pro reprezentaci uspořádané množiny. V těch umíme vyhledávat i mazat v čase logaritmickém vzhledem k počtu uložených prvků.

V níže uvedeném programu jsme použili datovou strukturu `multiset` v C++. Na začátku jsme do ní vložili všech pn pokojů (tedy n kopií každého z čísel od 1 do p). Následně jsme postupně pro každého hosta pomocí metody `upper_bound` určili nejvhodnější pokoj. Každá operace s touto datovou strukturou běží v čase $\mathcal{O}(\log(pn))$, celková časová složitost je proto $\mathcal{O}(pn \log(pn))$.

Ještě o něco lepší časové složitosti $\mathcal{O}(pn \log p)$ bychom mohli dosáhnout použitím datové struktury `map`, v níž bychom si pro každé poschodí, které ještě není plně obsazené, pamatovali počet volných pokojů. Stejnou časovou složitost dostaneme i při použití intervalového stromu, ve kterém listy odpovídají jednotlivým poschodím a v každém vnitřním vrcholu si pamatujeme index nejvyššího ne zcela zaplněného poschodí v příslušném podstromu.

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

int main() {
    // Načteme údaje o hotelu a počtu hostů.
    int P, N, H;
    cin >> P >> N >> H;

    // Vygenerujeme si všechny prázdné pokoje v hotelu.
    multiset<int> hotel;
    for (int p=1; p<=P; ++p) for (int n=0; n<N; ++n) hotel.insert(p);

    // Postupně čteme a zpracováváme hosty, dokud nejsou všichni nebo
    // dokud se nezasekneme.
    vector<int> reseni;
    while (H--) {
        int V; cin >> V;
        auto it = hotel.upper_bound(V); // "it" ukazuje na nejnižší NEvhodný pokoj
        if (it == hotel.begin()) break; // není pod ním vhodný volný pokoj => konec
        --it;                           // nyní "it" ukazuje na správný pokoj
        reseni.push_back(*it);          // ten si zapamatujeme v řešení...
        hotel.erase(it);                // ...a smažeme ho z množiny volných pokojů
    }

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

Ještě lepší řešení

Předcházející řešení stačilo k získání plného počtu bodů. Úlohu ale dokážeme vyřešit ještě o něco efektivněji, a to v čase $\mathcal{O}(pn \cdot \alpha(pn))$, kde α označuje inverzní Ackermannovu funkci.* Uvedeme alespoň základní myšlenku tohoto řešení.

Představte si, že jsme všechny pokoje v hotelu umístili do řady uspořádané podle poschodí. Když ubytováváme hosty, pokoje postupně obsazujeme. Časem nám tak začnou vznikat celé úseky, v nichž jsou všechny pokoje obsazené. Co se děje, když chceme ubytovat dalšího hosta? Podíváme se na nejvyšší poschodí, kde ještě může tento host bydlet. Je-li tam volný pokoj, hosta ubytujeme. Pokud jsme ale narazili na nějaký úsek obsazených pokojů, zajímá nás, kde tento úsek začíná – bezprostředně před ním je totiž pokoj, do kterého chceme našeho hosta ubytovat.

Pro reprezentaci jednotlivých úseků obsazených pokojů použijeme algoritmus Union-Find. Ke každému úseku si navíc zapamatujeme, kterým pokojem začíná. Operací Find umíme nalézt k optimálnímu hostovu pokoji celý úsek obsazených pokojů, v němž leží. Operaci Union použijeme vždy po obsazení pokoje na jeho připojení k úsekům ležícím bezprostředně před a za ním (pokud už jsou pokoje před/za ním také obsazené).

P-I-2 Žabka

Úlohu samozřejmě můžeme řešit hrubou silou – rekurzivně zkoušet všechny možnosti, jak mohla žabka postupně skákat. Těch ale může být mnoho,† takže takové řešení bude použitelné jen pro velmi malé hodnoty n.

Kubické řešení

Neefektivita předcházejícího řešení spočívá v tom, že zbytečně znovu a znovu zkouší stejné posloupnosti skoků. Představte si, že žabka už provedla několik skoků a zajímá nás, jak nejlépe může z této situace pokračovat dále do cíle. Důležité pozorování: Odpověď na tuto otázku *téměř nezávisí* na tom, jak žabka skákala dosud. Závisí jenom na posledním skoku, který provedla. Tím je totiž přesně určeno, kde se nyní nachází a jaké skoky může provést ve zbytku své cesty.

Zajímají nás tedy otázky následujícího tvaru: „Když žabka právě skočila z kamene a na kámen b, kolik nejvýše skoků ještě může udělat cestou do cíle?“ To je $\mathcal{O}(n^2)$ různých otázek. Každou z nich dokážeme zodpovědět tak, že rekurzivně vyzkoušíme $\mathcal{O}(n)$ možností, kam žabka skočí následujícím skokem. Pro každou možnost tak dostaneme jednu novou otázku, k jejímuž zodpovězení použijeme rekurzivní volání.

Kdybychom přímo implementovali právě popsaný postup, dostali bychom dříve uvedené řešení hrubou silou. Můžeme ale použít *memoizaci*: odpověď na každou

---

* Tato funkce roste tak pomalu, že pro všechny praktické potřeby je tento algoritmus lineární vzhledem k počtu pokojů v hotelu.

† Kolik vlastně? Exponenciálně vzhledem k n, nebo ještě více? Zkuste si to rozmyslet.

otázku budeme během výpočtu počítat jenom jednou. Jakmile nějakou odpověď zjistíme, zaznamenáme si ji do tabulky. Když se pak stejná otázka objeví později při výpočtu programu, jednoduše vrátíme uloženou hodnotu – tedy neprovádíme už žádné zkoušení možností ani žádné rekurzivní volání.

Takto upravený program potřebuje pro každou otázku na výpočet odpovědi čas nejvýše $\mathcal{O}(n)$. Celkově tedy program vykoná $\mathcal{O}(n^3)$ kroků výpočtu. Výsledkem úlohy je hodnota $1 +$ maximum z odpovědí na otázky, v nichž je $a = 1$.

```
#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) {
    // čtverec vzdálenosti mezi kameny 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) {
    // Pokud jsme už tuto otázku někdy dostali, rovnou vrátíme odpověď.
    if (memo[a][b] != INT_MIN) return memo[a][b];

    // Inicializujeme odpověď: jsme-li právě v cíli, je to 0 (můžeme zde skončit),
    // jinak -nekonečno.
    int odpoved = (b==N-1 ? 0 : -47);

    // Vyzkoušíme všechny možnosti, kam skočit déle, a vybereme nejlepší.
    for (int c=0; c<N; ++c) {
        if (c == b) continue; // nemůžeme zůstat na místě
        if (square_dist(a,b) <= square_dist(b,c))
            continue; // musíme provést kratší skok
        int moznost = otazka(b,c);
        if (moznost == -47) continue; // po tomto skoku nelze dojít do cíle
        odpoved = max( odpoved, 1+moznost );
    }

    // Zapamatujeme si spočítanou odpověď a vrátíme ji na výstup.
    memo[a][b] = odpoved;
    return odpoved;
}

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

```
    // Inicializujeme paměť.
    memo.resize( N, vector<int>(N,INT_MIN) );

    // Vyzkoušíme všechny možnosti pro první skok a vybereme nejlepší z nich.
    int odpoved = 0;
    for (int b=1; b<N; ++b) odpoved = max( odpoved, 1+otazka(0,b) );
    cout << odpoved << endl;
}
```

Přibližně kvadratické řešení

Předchozí řešení můžeme ještě zlepšit. Ušetřit se dá na tom, že v řešení s kubickou časovou složitostí u každé otázky procházíme všechny možné další skoky, a to včetně těch, které už v dané situaci nemůžeme udělat. Jak tedy zpracovávat skoky šikovněji?

Všechny možné skoky si roztřídíme do skupin podle jejich délky. (V programu raději použijeme druhou mocninu délky, jelikož tu si můžeme uložit do celočíselné proměnné.) Vzniklé skupiny potom uspořádáme podle délky skoku, počínaje největší.

Pro každý kámen si budeme pamatovat, kolika *nejvýše* skoky se na něj dokážeme dostat. Na začátku je to 0 pro start (tam začínáme) a $-\infty$ pro každý jiný kámen (tam zatím neznáme žádnou cestu). Postupně budeme zkoušet použít skoky v pořadí od nejdelších po nejkratší a vždy přepočítáme jen ty údaje, které se týkají právě zpracovávaných skoků.

Zpracování nového skoku je jednoduché: když se jedná o skok z kamene a na kámen b, podíváme se, kolika nejvýše skoky (tzn. delšími skoky) jsme se dokázali dostat na kámen a. Označme si tento počet jako k. Na kámen b se nyní umíme dostat pomocí $k+1$ skoků. Jestliže byla dosud uložená hodnota pro kámen b menší, zvýšíme ji na $k+1$.

Máme-li více skoků stejné délky, musíme dát pozor na to, abychom je všechny zpracovali najednou – aby se nám nestalo, že žabka vykoná více stejně dlouhých skoků po sobě.

Časová složitost tohoto řešení je $\mathcal{O}(n^2 \log n)$. Jeho nejpomalejší částí je uspořádání všech skoků podle délky. Samotné zpracování skoků je pak už efektivnější, dokážeme ho provést v konstantním čase na skok, tedy celkem v čase $\mathcal{O}(n^2)$.

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

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

# Vygenerujeme všechny možné skoky, roztřídíme je podle délky, délky uspořádáme.
skoky = defaultdict(list) # pro každou délku seznam skoků

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) )

delky = reversed( sorted( skoky.keys() ) )
```

```
# Postupně procházíme délky a počítáme, kam se dokážeme dostat na kolik skoků.
nejvic = [ -2**30 for n in range(N) ]
nejvic[0] = 0

for d in delky:
    # Ze starých informací vypočítáme nové.
    nove_info = []
    for a,b in skoky[d]: nove_info.append( ( b, nejvic[a]+1 ) )
    # Zapíšeme ty nové údaje, které jsou lepší než staré.
    for b,s in nove_info: nejvic[b] = max( nejvic[b], s )

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

P-I-3 Řazení kamenů

Řešení pro k=1

Pro $k = 1$ je situace jednoduchá. Každou hru lze vyhrát, hrací kameny postupně jeden po druhém přesuneme tam, kam patří. Začneme tím, že kámen číslo 1 necháme na místě. Na kámen číslo 2 budeme klikat tak dlouho, až se dostane vedle kamene číslo 1. Totéž pak uděláme postupně s kameny 3 až $n - 1$. Jelikož se kámen každým kliknutím posune jen o 1, jeho správné místo nikdy nepřeskočíme.

Řešení pro k=3

Ukážeme, že každou hru lze vyhrát – a to tak, že ji převedeme na hru předcházejícího typu. K tomu postačí nalézt posloupnost tahů, která vymění dva sousední kameny.

Protože $n > k + 1$, pro $k = 3$ máme aspoň 5 kamenů, tedy $n \geq 5$. Některých pět po sobě jdoucích kamenů si označíme $ABCDE$. Všimněte si, co se stane, když postupně klikneme na kameny D, E, B, C, A. Tato posloupnost kliknutí vůbec nezmění zbytek kruhu. Bude se měnit jenom pořadí naší pětice kamenů, a to následovně:

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

Vidíme, že po této posloupnosti tahů jsou všechny kameny na původních místech, pouze sousední kameny A a B si své pozice vyměnily. Na hru pro $k = 3$ tedy můžeme použít algoritmus pro $k = 1$, jenom na každou výměnu dvou sousedních kamenů budeme místo jednoho kliknutí potřebovat pět kliknutí.

Řešení pro k=2 a sudé n

Pro $k = 2$ umíme vyhrát všechny hry, v nichž je počet kamenů sudý. Nechť je $n = 2x$. Sledujme libovolný zvolený kámen. Co se stane, když na něj $(x - 1)$-krát po sobě klikneme? Pokaždé předběhne dva jiné kameny, takže celkem předběhne $2x - 2$ ze zbývajících $2x - 1$ kamenů. Jinými slovy, právě se nachází o jednu pozici dále, než byl na začátku. Výsledný efekt je stejný, jako kdybychom ho vyměnili s kamenem, který byl původně vedle něho (ve směru hodinových ručiček). Opět tedy můžeme použít přímočaré řešení pro $k = 1$, jenom se více naklikáme.

Téměř řešení pro k=2 a liché n

Pro $k = 2$ a liché n použijeme trochu jiný postup. Představte si, že jsme už umístili několik prvních kamenů vedle sebe a nyní k nim chceme přidat další. Postupně po obvodu máme tedy kameny rozmístěny například takto: $(1, 2, 3, 4, 5, x, x, x, 6, y, y)$.

Dokud máme aspoň dva kameny označené x (tzn. před kamenem, který chceme právě dostat na správné místo), můžeme klikat na náš kámen – v příkladu tedy na kámen číslo 6. Když nám potom zůstane právě jeden kámen x, odstraníme ho tak, že na něj budeme opakovaně klikat, až se přesune mezi kameny označené y.

Uvedený postup zjevně funguje, jestliže máme aspoň dva neumístěné kameny. Občas ale nastane na konci problém: například pro $n = 5$ se nám může stát, že po umístění kamene 3 dostaneme pořadí $(1, 2, 3, 5, 4)$. Kameny 4 a 5 se nám však už nepodaří vyměnit. V poslední části tohoto vzorového řešení si dokážeme, že je tomu skutečně tak – i kdybychom postupovali úplně jinak, takovouto hru nelze vyhrát.

Řešení pro k=2 a liché n

Mějme posloupnost navzájem různých čísel. Inverzí nazveme takovou dvojici čísel, v níž je větší číslo nalevo od menšího. Například posloupnost $(1, 2, 5, 3, 4)$ obsahuje dvě inverze: číslo 5 je nalevo od čísla 3 a zároveň číslo 5 je nalevo od čísla 4. Ukážeme si několik zajímavých vlastností inverzí:

Výměna dvou po sobě jdoucích prvků postupnosti vždy změní paritu počtu inverzí.

Toto je zřejmé: výměnou buď jednu inverzi odstraníme, nebo jednu přidáme.

Výměna prvního prvku s posledním změní paritu počtu inverzí.

Tvrzení snadno dokážeme: Mějme posloupnost délky n. Označíme si první prvek posloupnosti A a poslední prvek B. Výměnu prvků A a B můžeme provést tak, že nejprve $(n-1)$-krát vyměníme A s prvkem, který je napravo od něho (tím A přesuneme až za B na konec posloupnosti) a následně $(n-2)$-krát vyměníme B s prvkem nalevo od něho.

Celkem jsme tedy $(2n-3)$-krát vyměnili dva sousední prvky. Každá výměna změnila paritu počtu inverzí. Protože $2n-3$ je liché číslo, na konci bude parita počtu inverzí opačná, než jaká byla na začátku.

Když v posloupnosti liché délky přesuneme první prvek na konec, parita počtu inverzí se nezmění.

Důvod je obdobný: tento přesun můžeme provést pomocí $n-1$ výměn sousedních prvků.

Nechť posloupnost čísel liché délky n je zapsána do kruhu. Ať ji začneme číst (ve směru hodinových ručiček) od libovolného jejího členu, vždy dostaneme posloupnost délky n se stejnou paritou počtu inverzí.

Libovolnou takovou posloupnost získáme z libovolné jiné tak, že několikrát přesuneme první prvek na konec – a o tom již víme, že nám to paritu počtu inverzí nezmění. Nadále tedy budeme hovořit přímo o paritě počtu inverzí cyklické posloupnosti (liché délky) a budeme tím myslet paritu počtu inverzí libovolné z posloupností, které dostaneme jejím „rozstřižením“.

V naší hře pro liché n a pro $k = 2$ žádný tah nezmění paritu počtu inverzí.

Každý tah odpovídá dvěma postupným výměnám sousedních prvků.

A to už je všechno. V předchozí části jsme ukázali algoritmus, který pro liché n a pro $k = 2$ libovolnou výchozí pozici změní buď na pozici $(1, 2, 3, \ldots, n-2, n-1, n)$, nebo na pozici $(1, 2, 3, \ldots, n-2, n, n-1)$. Teď už navíc víme, že parita počtu inverzí počáteční pozice určuje, který z těchto případů nastane. Je-li parita počtu inverzí sudá, hru vyhrajeme. Je-li lichá, hru nelze vyhrát, neboť po každém provedeném tahu zůstane parita počtu inverzí lichá, takže nikdy nedosáhneme uspořádané posloupnosti, v níž je počet inverzí roven 0.

P-I-4 Sufixové stromy

A) Počet různých písmen *(2 body)*

Každým písmenem řetězce *začíná* právě jeden ze sufixů. Počáteční písmena sufixů odpovídají hranám vedoucím z kořene sufixového stromu. Počet různých písmen v řetězci tedy můžeme určit v konstantním čase jako stupeň kořene sufixového stromu.

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

B) Nejdelší opakující se řetězec *(4 body)*

Uvažujme libovolné dva výskyty téhož řetězce T v řetězci S. Každým výskytem začíná jeden ze sufixů řetězce S. V sufixovém stromu tedy můžeme vyjít z kořene a podle písmen řetězce T sestupovat dolů. Pod místem, kde skončíme, se budou nacházet konce obou sufixů.

Naopak, zvolme libovolné místo v sufixovém stromu, pod nímž leží konce aspoň dvou sufixů. Řetězec, který nás dovede z kořene na toto místo, se zjevně nachází aspoň dvakrát v původním řetězci – každý ze sufixů, které jím začínají, představuje jeden jeho výskyt.

Nejdelší opakující se řetězec tedy určíme tak, že v sufixovém stromu najdeme nejhlubší vrchol, v němž nebo pod nímž se nacházejí konce aspoň dvou sufixů. K tomu stačí použít funkci `spocitej_konce` ze studijního textu a následně jednou projít celý strom například prohledáváním do hloubky.

Zde uvedený program najde délku nejdelšího opakujícího se podřetězce. K sestrojení tohoto podřetězce bychom následně mohli použít podobnou funkci, která však dostane na vstupu také hledané optimum a na výstupu vrátí přímo řetězec příslušné délky.

```
infinity = float('inf')

def nejhlubsi_se_dvema(vrchol):
    if vrchol.data < 2: return -infinity
    odpoved = 0
    for hrana in vrchol.deti.values():
        delka = hrana.do - hrana.od
        odpoved = max( odpoved, (hrana.do - hrana.od) +
            nejhlubsi_se_dvema(hrana.kam) )
    return odpoved
```

```
def nejdelsi_repeat(S):
    strom = vytvor_strom(S)
    spocitej_konce( strom.koren )
    return max( 0, nejhlubsi_se_dvema( strom.koren ) )
```

C) Počet různých podřetězců *(4 body)*

Pomůže nám podobná úvaha: Každým podřetězcem začíná některý sufix, a proto každému podřetězci odpovídá cesta z kořene sufixového stromu dolů.

Kdyby sufixový strom nebyl komprimovaný, tzn. kdyby na každé hraně bylo napsané jen jedno písmeno, dokázali bychom počet různých podřetězců spočítat jako počet vrcholů v sufixovém stromu (nepočítaje kořen – ten odpovídá prázdnému řetězci, který podle zadání nepočítáme). Každému vrcholu odpovídá jiný podřetězec – ten, který je napsán na cestě z kořene do tohoto vrcholu.

Rovněž v samotném sufixovém stromu můžeme ale tuto informaci snadno získat: jednoduše sečteme délky všech hran sufixového stromu. Hraně délky k v sufixovém stromu totiž v jeho nekomprimované verzi odpovídá právě k vrcholů.

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

def pocet_podretezcu(S):
    strom = vytvor_strom(S)
    return soucet_delek_hran( strom.koren )
```