A-III-1 Konferenčné zákusky

Najskôr sa zamyslíme nad optimálnym výberom koláčov keby existovala iba prestávka, potom si ukážeme, ako toto riešenie upraviť tak, aby sme doň zakomponovali aj výber koláča, ktorý zje Peťka až cez prednášku.

Vyberanie koláčov jedených počas prestávky

Rozmyslime si, ako vyzerá Peťkina situácia na začiatku prestávky. Pred sebou má n koláčov, do konca prestávky ostáva p sekúnd a jej spokojnosť je 0 (keďže zatiaľ žiaden koláč nezjedla). Sama seba sa pýta otázku: *Koľko najviac spokojnosti viem získať zjedením niekoľkých z týchto n koláčov tak, aby čas ich jedenia bol nanajvýš p?*

Ako prvé si môžeme uvedomiť, že na poradí, v akom zje vybrané koláče vôbec nezáleží. Keďže koláče sú očíslované od 1 po n, môžeme si povedať, že ich bude jesť v poradí od najvyššieho čísla. Pozrime sa teda na koláč číslo n. Buď sa rozhodne, že ho zje (samozrejme ak $t_n \leq p$) alebo, že ho nechá nezjedený.
Ak sa rozhodne, že ho **jesť nebude**, jej spokojnosť sa nezmení a do konca prestávky jej bude stále ostávať p sekúnd. K dispozícii však bude mať už len $n-1$ koláčov s nižšími číslami. Bude preto chcieť vedieť odpoveď na otázku: *Koľko najviac spokojnosti viem získať zjedením niekoľkých z týchto $n-1$ koláčov tak, aby čas ich jedenia bol nanajvýš p?*
Ak sa rozhodne, že **zje n-tý koláč**, jej spokojnosť sa zväčší o s_n, jeho jedením však stratí t_n času a do konca prestávky bude ostávať už len $p-t_n$ sekúnd. No a k dispozícii jej ostanú zvyšné koláče. Aby maximalizovala svoju spokojnosť, bude sa pýtať: *Koľko najviac spokojnosti viem získať zjedením niekoľkých z týchto $n-1$ koláčov tak, aby čas ich jedenia bol nanajvýš $p-t_n$?*

Všimnite si, že sa zakaždým pýta tú istú otázku, akurát s inými hodnotami pre počet koláčov a dĺžku prestávky. Odpoveď na otázku „*Koľko najviac spokojnosti viem získať zjedením niekoľkých z týchto x koláčov tak, aby čas ich jedenia bol nanajvýš y?*“ si preto vieme reprezentovať ako funkciu $f(x,y)$.
A celú situáciu popísanú vyššie vieme zapísať jednoduchým rekurzívnym vzťahom

$$f(n,p) = \max(f(n-1,p), s_n + f(n-1,p-t_n))$$

Z daných dvoch možností si totiž Peťka chce vybrať tú, ktorá jej zaručí väčšiu celkovú spokojnosť.

Keďže sa jedná o rekurzívnu funkciu, vieme ju veľmi priamočiaro implementovať v ľubovoľnom programovacom jazyku. Treba si len uvedomiť, že pre výpočet hodnoty $f(x,y)$ je druhá možnosť (že Peťka koláč zje) možná iba vtedy, ak je $t_x \leq y$, a ak je $x = 0$ nepokračujeme vo výpočte, keďže z 0 koláčov vie Peťka získať nanajvýš 0 spokojnosti.
Samozrejme, priamočiara rekurzívna implementácia by bola neefektívna, vieme však veľmi jednoducho použiť memoizáciu. Funkciu $f(x,y)$ chceme počítať nanajvýš pre všetky hodnoty $0 \leq x \leq n$ a $0 \leq y \leq p$, máme teda len $O(np)$ rôznych možných výpočtov. Každý z nich trvá konštantne dlho, a preto je aj výsledná časová zložitosť takéhoto riešenia $O(np)$.

Vyberania koláča na prednášku

Ostáva zistiť, ako efektívne vybrať koláč, ktorý si má Peťka zobrať na prednášku. Problémom však je, že výber tohto koláča ovplyvňuje to, ktoré koláče si vyberie cez prestávku. Nestačí jednoducho spustiť vyššie spomenuté riešenie a pridať k nemu jeden z nevybraných koláčov, pretože to nemusí viesť k maximálnej spokojnosti.
Pozrime sa teda na riešenie ako celok. Predstavme si, že poznáme optimálnu sadu koláčov, ktoré má Peťka zjesť počas prestávky a cez prednášku. Označme si koláč, ktorý zje Peťka v tomto optimálnom riešení počas prednášky ako k. Čo o ňom vieme povedať?

Určite musí platiť, že jeho spokojnosť s_k **je väčšia** alebo rovná ako spokojnosť ľubovoľného z nevybraných koláčov. Ak by totiž existoval koláč, ktorý sme do riešenia nevybrali a mal by väčšiu spokojnosť ako s_k, mohli by sme ho vymeniť za koláč k, čím by sme zväčšili celkovú spokojnosť nášho riešenia. To je však v spore s tým, že predpokladáme, že naše riešenie je optimálne.

Čo bude platiť o čase t_k koláča k? Keďže na prednáške môžeme jesť koláč ľubovoľne dlho, je prirodzené vyberať si koláče, ktorých čas jedenia je dlhý. Môžeme si preto uvedomiť, že existuje také optimálne riešenie, v ktorom je hodnota t_k najvyššia spomedzi hodnôt t pre všetky vybrané koláče.

Predstavme si, že tomu tak nie je a máme optimálne riešenie, v ktorom počas prestávky zjeme koláč l, pre ktorý platí, že $t_l > t_k$. Tieto dva koláče môžeme vymeniť. Koláč k zjeme už počas prestávky a koláč l si necháme na prednášku. Je jasné, že celková spokojnosť sa nezmení, keďže sme len vymenili poradie, v ktorom jeme koláče. Zároveň sme však neporušili ani podmienku toho, že celkový čas jedenia koláčov počas prestávky je nanajvýš p. Predtým sme totiž počas prestávky stihli zjesť koláč s časom jedenia t_l, ktorý sme teraz nahradili koláčom s kratším časom jedenia t_k. Takouto výmenou preto dostaneme rovnako dobré riešenie, ktoré má však väčšiu hodnotu t pre koláč zjedený počas prednášky.

Ostáva nám už len využiť tieto pozorovania pri riešení pôvodného problému. Uvedomme si, že naše rekurzívne riešenie spracováva koláče v zadanom poradí, principiálne však nezáleží na tom, v akom poradí sú tieto koláče zadané. Zoraďme si preto koláče vzostupne podľa hodnôt t_i. Ak teraz spočítame hodnotu $f(x, y)$ dostaneme optimálnu spokojnosť pre koláče, ktorých čas jedenia nepresahuje t_x.
Ak si zoberieme optimálne riešenie a pozrieme sa, ktoré koláče sme zjedli počas prestávky, jeden z týchto koláčov, označme ho z, bude mať najväčšiu hodnotu t. Toto riešenie teda musí mať optimálnu spokojnosť pre všetky koláče, ktorých čas nepresahuje t_z. Čo je presne riešenie $f(z, y)$, keď využijeme utriedené poradie.
Navyše si na tomto usporiadanom poli pre každý sufix spočítajme maximum hodnôt s_i. Toto maximum pre sufix začínajúci i-tym prvkom poľa označme $g(i)$. Potom musí platiť, že optimálne riešenie má spokojnosť rovnú $f(y, p) + g(y)$.

My samozrejme nepoznáme vhodné y, ale nič nám nebráni vyskúšať všetky možnosti. Všetky hodnoty $f()$ vieme spočítať v čase $O(np)$, hodnoty $g()$ v čase $O(n)$ a vyskúšanie všetkých možných y potom trvá tiež $O(n)$. Nezabudneme na usporiadanie poľa podľa hodnôt t_i a dostaneme výslednú časovú zložitosť $O(n \log n + np)$.

Hľadanie konkrétneho rozdelenia

Nezabudnime, že pri riešení nás nezaujíma iba celková najväčšia spokojnosť, chceme nájsť aj jednu konkrétnu možnosť, ktoré koláče má Peťka kedy zjesť. Musíme preto vedieť naše riešenie zrekonštruovať.
To však nie je náročné, stačí ak si pri počítaní funkcie $f()$ zapamätáme, ktorá z dvoch možností viedla k optimálnemu riešeniu. Výsledkom pre $f(x, y)$ teda nebude iba najväčšie množstvo spokojnosti, ale aj to, či toto množstvo spokojnosti dosiahne Peťka zjedením alebo nezjedením koláču x. Vďaka tejto informácii potom vieme spätne zrekonštruovať riešenie, keďže vieme, ako sa zmenila Peťkina situácii pri každom koláči.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
using namespace std;

int pocet_kolacov, prestavka;
vector<pair<pair<int, int>, int> > kolace;
pair<long long, int> MemF[10047][10047];

pair<long long, int> f(int pocet, int cas) {
    int cas_jedenia = kolace[pocet].first.first;
    int spokojnost = kolace[pocet].first.second;

    if(pocet == -1) return {0, -1};
    if(MemF[pocet][cas].first != -1) return MemF[pocet][cas];

    long long nezje = f(pocet-1, cas).first;
    long long zje = -1;
    if(cas_jedenia <= cas) {
        zje = spokojnost + f(pocet-1, cas - cas_jedenia).first;
    }

    pair<long long, int> vysledok;
    if(zje > nezje) vysledok = {zje, 1};
    else vysledok = {nezje, 0};
    return MemF[pocet][cas] = vysledok;
}

pair<long long, int> MemG[10047];

pair<long long, int> g(int pocet) {
    int spokojnost = kolace[pocet].first.second;
    if(pocet == pocet_kolacov) return {0, -1};
    if(MemG[pocet].first != -1) return MemG[pocet];
```

```
    pair<long long, int> suffix = g(pocet + 1);
    pair<long long, int> vysledok = suffix;
    if(spokojnost > suffix.first) vysledok = {spokojnost, pocet};
    return MemG[pocet] = vysledok;
}

int main() {
    scanf("%d␣%d", &pocet_kolacov, &prestavka);
    for(int i = 0; i < pocet_kolacov; i++) {
        kolace.push_back({{0, 0}, i});
    }
    for(int i = 0; i < pocet_kolacov; i++) scanf("␣%d", &kolace[i].first.second);
    for(int i = 0; i < pocet_kolacov; i++) scanf("␣%d", &kolace[i].first.first);
    for(int i = 0; i < pocet_kolacov; i++) {
        MemG[i] = {-1, -1};
        for(int j = 0; j <= prestavka; j++) MemF[i][j] = {-1, -1};
    }

    sort(kolace.begin(), kolace.end());

    int najlepsie_riesenie = -1;
    for(int i = 0; i < pocet_kolacov; i++) {
        if(f(i, prestavka).first + g(i + 1).first > f(najlepsie_riesenie, prestavka).first + g(najlepsie_riesenie + 1).first)
            najlepsie_riesenie = i;
    }

    pair<long long, int> prva_cast = f(najlepsie_riesenie, prestavka);
    pair<long long, int> druha_cast = g(najlepsie_riesenie + 1);
    printf("%lld\n", prva_cast.first + druha_cast.first);
    vector<int> kolace_prestavka;
    int kolac = najlepsie_riesenie;
    int zvysny_cas = prestavka;
    while(kolac != -1) {
        if(f(kolac, zvysny_cas).second == 1) {
            kolace_prestavka.push_back(kolace[kolac].second);
            zvysny_cas -= kolace[kolac].first.first;
        }
        kolac--;
    }
    for(int i = 0; i < kolace_prestavka.size(); i++)
        printf("%d%c", kolace_prestavka[i] + 1, (i != kolace_prestavka.size() - 1)?'␣':'\n');
    if(kolace_prestavka.size() == 0) printf("\n");
    if(g(najlepsie_riesenie + 1).second != -1)
        printf("%d\n", g(najlepsie_riesenie + 1).second + 1);
    else printf("\n");
}
```

Bonus: riešenie s lepšou pamäťou

Vyššie popísané riešenie na rekonštrukciu optimálnej sady koláčov potrebuje nielen čas ale aj pamäť $\Theta(np)$: potrebujeme si pamätať celú tabuľku hodnôt $f(x,y)$. Ak by nám stačilo poznať optimálnu spokojnosť Peťky, vedeli by sme zlepšiť pamäť počas tejto časti výpočtu na $O(p)$: pri počítaní hodnôt $f(x+1,\cdot)$ si stačí pamätať hodnoty $f(x,\cdot)$.

Existuje však spôsob, ako získať všetky výhody: mať stále rovnakú asymptotickú časovú zložitosť, vedieť zostrojiť riešenie, a pritom mať stále celkovú pamäťovú zložitosť len $O(n+p)$. Načrtneme hlavnú myšlienku. Predstavme si, že prvú polovicu koláčov ofarbíme na červeno a druhú na modro. Kým spracúvame červené koláče, robíme iba to, čo sme robili pri pôvodnom riešení s $O(p)$ pamäte. Keď spracúvame modré koláče, naďalej používame $O(p)$ pamäte, ale pre každý počet sekúnd si pamätáme nielen optimálnu spokojnosť Peťky ale ešte jedno číslo navyše: koľko času sme pri tomto optimálnom riešení mali po spracovaní všetkých červených koláčov.

Keď teraz nájdeme hodnotu globálneho optima, vieme navyše, koľko času pri ňom strávime jedením červených koláčov – a zvyšok času strávime optimálnym jedením modrých. (V najhoršom prípade nám ostanú všetky červené a skoro všetky modré koláče, ale ak bolo optimálne zobrať na prednášku už niektorý skorý koláč, môžeme napríklad vyberať tie, čo zjeme cez prestávku, len spomedzi prefixu červených.)

Na zostrojenie optimálneho riešenia teraz spravíme (nanajvýš) dve rekurzívne volania. Pri každom z nich máme nejaké množstvo času a nejaký súvislý úsek koláčov, ktoré môžeme jesť.

Dôležitá vlastnosť teraz je, že pri pôvodnom volaní sme vypĺňali tabuľku rozmerov $n \times p$. Pri každom z týchto dvoch rekurzívnych volaní budeme vypĺňať tabuľku s prvým rozmerom $n/2$ a súčet ich druhých rozmerov bude p, dokopy teda spravíme len polovičné množstvo práce.

No a ako sme už povedali, obe tieto časti rekonštrukcie riešenia budú rekurzívne volania toho istého algoritmu. V každom z nich teda opäť zoberieme príslušný úsek koláčov, prvú polovicu z nich ofarbíme na červeno a druhú

na modro, a tak ďalej.
Každá ďalšia úroveň tejto rekurzie bude mať polovičné množstvo práce v porovnaní s predchádzajúcou. Dokopy tento algoritmus teda spraví približne dvakrát toľko práce ako ten pôvodný – a teda jeho asymptotická časová zložitosť ostane rovnaká.

A-III-2 Lezenie

Postupne dokážeme viacero pozorovaní, ktoré nás dovedú k efektívnemu riešeniu. Prvé pozorovanie bude zjavné: pri optimálnom riešení dosiahneme aspoň jednu zo želaných hodnôt β a λ presne. Totiž ak máme ľubovoľný tréningový plán, ktorý skončí tým, že sme prekročili aj β aj λ, tak nemôže byť optimálny. Ľahko vyrobíme kratší plán – napríklad tak, že prestaneme trénovať skôr, akonáhle dosiahneme druhý z cieľov presne.
Iný spôsob ako prerobiť platné riešenie, ktoré natrénovalo $\beta' > \beta$ a $\lambda' > \lambda$ na kratšie platné riešenie je *preškálovať ho*. Uvažujme konštantu $\alpha = \max(\beta/\beta',\ \lambda/\lambda')$. Zjavne $0 < \alpha < 1$. Ak všetky časy tréningu prenásobíme α, dostaneme nové kratšie riešenie, ktoré aspoň jednu zo želaných hodnôt dosiahne presne.

Riešenie pre jednu stenu

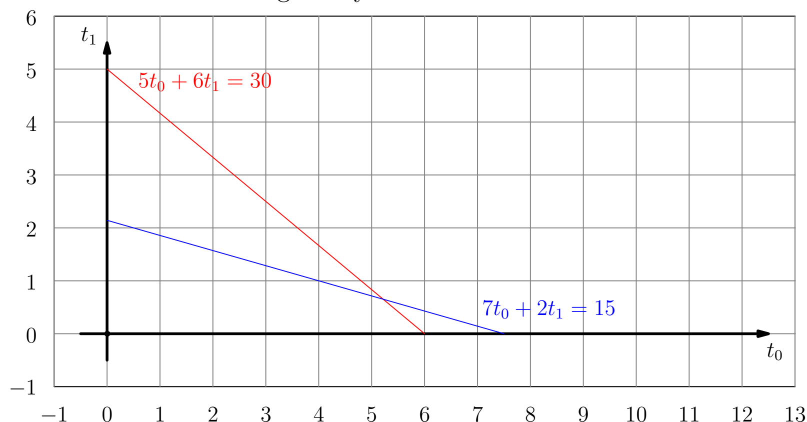
Ak máme len jednu stenu s parametrami (b_0, ℓ_0), riešenie je zjavné: potrebujeme aspoň β/b_0 hodín na dosiahnutie dostatočnej schopnosti v boulderingu, aspoň λ/ℓ_0 hodín na lead, riešením je teda maximum z týchto dvoch časov.

Riešenie pre dve podobné steny

Na steny sa oplatí dívať ako na vektory (b_i, ℓ_i).
Všimnime si situáciu, v ktorej máme dve steny také, že jedna je konštantným násobkom druhej: jedna má parametre (b_0, ℓ_0) a druhá $(\alpha b_0, \alpha \ell_0)$.
Ak $\alpha > 1$, je zjavné, že sa oplatí trénovať len na druhej z týchto stien: na presne rovnaký zisk schopností ako na prvej stene nám stačí trénovať α-krát kratšie. A naopak, ak $\alpha < 1$, ľubovoľný tréning na druhej stene vieme nahradiť kratším tréningom na prvej.
Takéto situácie teda vieme riešiť použitím riešenia pre jednu stenu.

Všeobecné riešenie pre dve steny

Majme teraz dve steny, ktoré nie sú podobné (vo vyššie uvedenom zmysle – odborne tomu hovoríme vektory, ktoré nie sú lineárne závislé).
Keď Adam strávi t_0 času na stene 0 a t_1 času na stene 1, získa $t_0 b_0 + t_1 b_1$ schopností v boulderingu a $t_0 \ell_0 + t_1 \ell_1$ schopností v leade. Nás zaujíma minimálna hodnota $t = t_0 + t_1$, pre ktorú je zároveň prvá z týchto hodnôt aspoň β a druhá aspoň λ. Máme teda dve lineárne nerovnice s dvoma premennými.
Túto sústavu nerovníc si môžeme znázorniť graficky:



V prvom kvadrante nám každá nerovnica „zakáže“ jeden trojuholník – ten ležiaci pod priamkou, pre ktorú platí rovnosť. My potom hľadáme v nezakázanom zvyšku bod s najmenším súčtom súradníc.

Je zjavné, že keď si vyberieme ľubovoľnú priamku a hýbeme sa po nej, súčet súradníc bodu, v ktorom sa nachádzame, sa spojito mení. Preto je zjavné, že optimálne riešenie stačí hľadať v extrémnych bodoch hranice: v bode, kde sa obe hraničné priamky pretínajú, a v bodoch, kde niektorá hraničná priamka pretína súradnicovú os.

Toto geometrické pozorovanie si teraz môžeme preložiť späť do reči našej súťažnej úlohy: pre dve steny vždy platí, že v optimálnom riešení buď trénujeme len na jednej z nich, alebo trénujeme na oboch tak, že presne dosiahneme obe hodnoty β a λ.

Priesečník oboch hraničných priamok vieme nájsť nasledovne: Z $t_0b_0+t_1b_1 = \beta$ si vyjadríme, že $t_1 = (\beta-t_0b_0)/b_1$. Toto dosadíme do rovnice druhej priamky a dostaneme lineárnu rovnicu s jednou neznámou. Tá po úprave vyzerá nasledovne:

$$t_0 = \frac{b_1\lambda - \beta\ell_1}{b_1\ell_0 - b_0\ell_1}$$

Výraz v menovateli zlomku je nulový len ak sú naše dve priamky rovnobežné. To je ale presne prípad, ktorý sme už ošetrili vyššie – dve podobné steny.

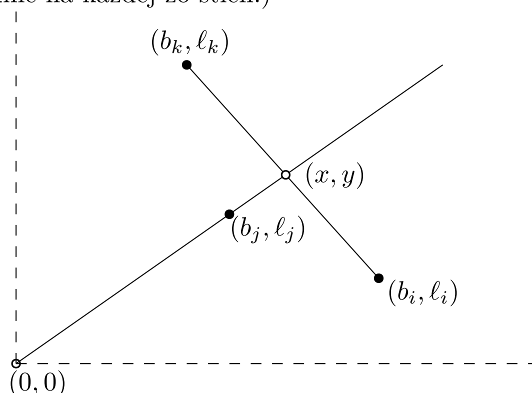
Riešením sústavy rovníc môžeme občas dostať riešenie, v ktorom neplatí, že t_0 a t_1 sú kladné – hraničné priamky sa pretínajú mimo prvého kvadrantu. V takomto prípade táto možnosť proste nevedie k optimálnemu riešeniu našej úlohy a môžeme ju ignorovať.

Vždy stačia najviac dve steny

Dôležitým krokom ku všeobecnému riešeniu bude pozorovanie, ktoré si dokážeme v tejto časti: vždy existuje optimálne riešenie, v ktorom sú najviac dva z časov t_i nenulové – teda optimálne riešenie, v ktorom Adam trénuje na nanajvýš dvoch z n dostupných stien.

Začnime tým, že si tréningové steny pretriedime: ak máme nejaké skupiny podobných stien, z každej si necháme len tú, na ktorej sa trénuje najrýchlejšie. Steny, ktoré nám ostali, si teraz môžeme usporiadať podľa uhlu, ktorý zviera vektor (b_i, ℓ_i) s kladnou poloosou osi x.

Uvažujme ľubovoľné tri steny s indexami $i < j < k$ Znázornime si ich graficky ako body v prvom kvadrante súradnicovej sústavy. Pozrime sa na úsečku spájajúcu (b_i, ℓ_i) a (b_k, ℓ_k). Ľahko nahliadneme, že body tejto úsečky predstavujú všetky možné výsledky tréningu, ak dokopy strávime hodinu času na stenách i a k. (Napr. stred tejto úsečky zodpovedá pol hodine na každej zo stien.)



Pozrime sa teraz, kde táto úsečka pretne polpriamku z počiatku súradníc cez bod (b_j, ℓ_j). Ak je to ako na obrázku vyššie, teda tento priesečník leží na polpriamke aspoň tak ďaleko ako bod (b_j, ℓ_j), je zjavne zbytočné trénovať na stene j. Totiž ak namiesto hodiny tréningu na stene j rozdelíme v správnom pomere hodinu tréningu na stenách i a k, získame tým aspoň rovnako oboch schopností.

No a naopak, ak je priesečník (x, y) bližšie ako bod (b_j, ℓ_j), môžeme urobiť presne opačnú optimalizáciu. Ak by oba časy t_i a t_k boli kladné, môžeme zobrať vhodne zvolené dva kusy tréningu na stenách i a k ktoré sú v tom správnom pomere a nahradiť ich menším časom tréningu na stene j, ktorý bude mať presne rovnaký efekt. V takejto situácii teda každé optimálne riešenie má buď $t_i = 0$ alebo $t_k = 0$ (alebo oboje).

Takto dostávame riešenie s časovou zložitosťou $O(n^2)$: pre každú dvojicu stien spočítame, ako najrýchlejšie sa Adam vie vytrénovať na nich, a z týchto možností vyberieme optimálnu. Za toto riešenie sa dalo získať 5 bodov.

Menej kandidátov

Spomedzi tréningových stien nemusíme zahadzovať len tie, ktoré sú podobné iným a menej efektívne. Omnoho účinnejšie pravidlo je nasledovné: ak máme dve steny (b_i, ℓ_i) a (b_j, ℓ_j) také, že $b_i \leq b_j$ a $\ell_i \leq \ell_j$, tak zjavne nikdy nemusíme použiť stenu i – ľubovoľný tréning na nej vieme nahradiť rovnako dlhým tréningom na stene j. Ak sú všetky b_i aj ℓ_i kladné celé čísla neprekračujúce m, ostane nám po použití tohto pravidla nanajvýš m rôznych hodnôt – totiž spomedzi stien s rovnakým b vždy necháme nanajvýš jednu: tú s najväčším ℓ.
Celé toto filtrovanie zbytočných stien vieme spraviť v čase $O(n \log n)$ napr. tak, že si zoznam stien usporiadame primárne podľa b a sekundárne podľa ℓ a potom ho prejdeme od konca. Následne na steny, ktoré sme nevyhodili, použijeme vyššie popísané riešenie.
Toto riešenie má celkovú časovú zložitosť $O(n \log n + m^2)$ a mohli ste zaň získať 7 bodov.

Optimálna sada kandidátov: konvexný obal

Úvahu, ktorú sme spravili vyššie pre dve steny, vieme zovšeobecniť aj pre dva ľubovoľné tréningové programy. Ak existuje možnosť, ako za hodinu natrénovať (β_1, λ_1) a iná možnosť, ako natrénovať (β_2, λ_2), tak vieme za hodinu natrénovať aj hociktorý bod, ktorý leží na úsečke medzi týmito dvoma – stačí príslušne naškálovať oba tréningy. (Napr. na stred úsečky by sme spravili polovičné časy prvého a potom polovičné časy druhého tréningu.)
Z toho už vyplýva, že množinu všetkých bodov, ktoré vieme za *najviac* hodinu natrénovať, tvorí práve konvexný obal počiatku a n bodov predstavujúcich jednotlivé tréningové steny.
No a keďže všetko v konvexnom obale vieme vyrobiť len pomocou tých tréningových stien, ktoré tvoria vrcholy tohto konvexného obalu, môžeme vyššie uvedenú optimalizáciu vylepšiť tak, že v čase $O(n \log n)$ nájdeme konvexný obal a zahodíme všetky steny okrem jeho vrcholov. Detailný popis tohto algoritmu nájdete napríklad tu: `https://www.ksp.sk/kucharka/konvexny_obal/`
Toto riešenie je o čosi lepšie ako predchádzajúce. Ľahko nahliadneme, že konvexný obal bodov so súradnicami neprevyšujúcimi m má nanajvýš m vrcholov, keďže jeho vrcholy musia mať navzájom rôznu prvú súradnicu. Platí však aj silnejšie tvrdenie – dá sa dokázať, že takýto konvexný obal má vždy vrcholov nanajvýš rádovo $m^{2/3}$. (Jedna možná myšlienka dôkazu je taká, že keď postupne ideme po jeho obvode, pohybom po hranách musia zodpovedať navzájom rôzne vektory s celočíselnými súradnicami.) To znamená, že celková časová zložitosť nášho riešenia je $O(n \log n + m^{4/3})$.

Ešte presnejšie pozorovanie je, že z konvexného obalu nám stačí uvažovať len body na jeho „pravej hornej štvrtine", teda od bodu s maximálnym b po bod s maximálnym ℓ. Všetky ostatné vrcholy sú od týchto horšie v zmysle, ktorý sme si popísali v časti Menej kandidátov. Toto už asymptotiku časovej zložitosti neovplyvní, ale zjednoduší to naše úvahy vo zvyšku riešenia.

Optimálne steny na konvexnom obale susedia

Na záver riešenia spojíme všetko, čo sme už o našej úlohe objavili. Vieme, že existuje optimálne riešenie, v ktorom sa používajú najviac dve tréningové steny. Optimálne riešenie s jednou stenou nájdeme triviálne, ostáva teda zistiť, či ho nevieme zlepšiť použitím práve dvoch. Vieme, že ak sa v optimálnom riešení naozaj použijú dve steny, vždy natrénujeme presne β v boulderingu a presne λ v leade. Poďme teda najlepšie takéto riešenie nájsť.
Potrebujeme sa čo najrýchlejšie dostať do bodu (β, λ). Keby sme už takéto optimálne riešenie mali a všetky tréningové časy vydelili jeho celkovým časom t, dostali by sme nejaký platný tréningový program, ktorý trvá hodinu a pohne nás v smere vektora (β, λ). A naopak, ľubovoľný hodinu trvajúci program, ktorý nás pohne v tomto smere, vieme naškálovať a tým dostať platné riešenie. Optimálne riešenie teda nájdeme tak, že nájdeme najväčšiu vzdialenosť, o ktorú sa vieme v tomto smere za hodinu dostať.
Pozrime sa znova na náš konvexný obal – teda všetky body, kam sa vieme dostať za nanajvýš hodinu. Tréningové plány, ktoré nás hýbu správnym smerom, ležia na polpriamke z počiatku cez bod (β, λ). Optimálne riešenie s dvomi stenami zjavne zodpovedá najvzdialenejšiemu bodu, ktorý leží aj na tejto polpriamke, aj v konvexnom obale.
Ak polpriamka celá ide mimo konvexného obalu, teda zdieľajú len počiatok súradnicovej sústavy, je optimálne použiť len jednu stenu. Vo všetkých ostatných prípadoch sa okamih, kedy polpriamka opustí náš konvexný obal,

nachádza v jeho hornej štvrtine. Ak sme trafili presne vrchol, použijeme jemu zodpovedajúcu stenu. Ak sme vyšli von cez stranu konvexného obalu, máme bod, ktorému zodpovedajúci tréning vieme dostať ako vážený priemer tréningov na tých dvoch stenách, ktorým zodpovedajú susedné dva vrcholy konvexného obalu.

Detaily implementácie

Do množiny bodov zodpovedajúcich tréningovým stenám ešte pred robením konvexného obalu pridáme aj hraničné body $(\max b_i, 0)$ a $(0, \max \ell_i)$. Horná štvrtina konvexného obalu potom spája obe súradnicové osi a nemusíme špeciálne ošetrovať prípady, kedy polpriamka do (β, λ) konvexný obal nepretne. Ak polpriamka pretne prvú alebo poslednú stranu takéhoto konvexného obalu alebo ak prejde jeho vrcholom, je optimálne použiť príslušnú jednu stenu, ak pretne hociktorú inú stranu, je optimálne použiť príslušné dve steny.
Po tom, ako v čase $O(n \log n)$ zostrojíme konvexný obal, už nemusíme robiť nič zložité. Stačí v čase $O(n)$ postupne pre každú jeho stranu skontrolovať, či pretína polpriamku vedúcu do nášho vytúženého cieľa. Celková časová zložitosť tohto riešenia je teda $O(n \log n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>
#define INF 99999999999.0
using namespace std;
typedef long long ll;
typedef pair<ll, ll> pll;

int n;
double beta, lambda;
vector<pll> steny;

double jedna_stena (pll x) {
        return max(beta / x.first, lambda / x.second);
}

// Predpokladame, ze x aj y lezia na hornej casti konvexniho obalu, a teda nemusime
// riesit specialne pripady, len pripad kedy nastanu dve rovnosti.
double dve_steny (pll x, pll y) {
        double t2 = (lambda*x.first - beta*x.second) / (x.first*y.second - x.second*y.first);
        double t1 = (beta - t2*b2) / b1;
        if (t2 >= 0 && t1 >= 0) return t1 + t2; else return INF;
}

int main() {
        cin >> n >> beta >> lambda;
        ll b, l;
        ll maxb = -1, maxl = -1
        for (int i = 0; i < n; ++i) {
                cin >> b >> l;
                steny.push_back({ b, l });
                maxb = max(maxb, b);
                maxl = max(maxl, l);
        }
        steny.push_back({ maxb, 0 });
        steny.push_back({ 0, maxl });

        // Funkcia upper_convex_hull najde hornu polovicu konvexneho obalu.
        // Na jej zaciatku a konci budu body { maxb, 0 } a { 0, maxl }.
        vector<pll> obal = upper_convex_hull(steny);
        int m = obal.size();

        // Vyskusame moznosti trenovat len na prvej / len na poslednej stene konvexneho obalu.
        double result = min(jedna_stena(obal[1]), jedna_stena(obal[m - 2]));

        // Vyskusame moznosti trenovat na dvoch "susednych" stenach.
        for (int i = 1; i < m-2; ++i) result = min(result, dve_steny(obal[i], obal[i+1]));
        return result;

}
```

A-III-3 Hviezdne impérium a hypercestovanie

Podúloha A: hyperdiaľnice

Ak existuje nejaký spôsob, ako postaviť hyperdiaľnice, tak určite existuje taký spôsob, pri ktorom je hyperdiaľnic presne $n - 1$ a tvoria strom.

Dôkaz: Majme ľubovoľnú vyhovujúcu sadu hyperdiaľnic. Zoberme teraz ľubovoľnú jej kostru – teda podgraf, ktorý je strom. Tento je tiež platným riešením: je súvislý a jeho stupne vrcholov sú nanajvýš rovné pôvodným. Stačí nám teda overiť, či existuje riešenie, ktoré je strom.

Ukážeme si riešenie, ktoré potrebuje $k = 2\lceil\log_2 n\rceil$ vyveštených bitov. Hlavnou myšlienkou tohto riešenia bude, že si dáme vyveštiť jeden konkrétny strom hyperdiaľnic a o ňom následne overíme, či spĺňa všetky ostatné podmienky.
Vo výslednom strome budeme vrchol 0 považovať za koreň. Pre každý iný vrchol v platí, že v strome existuje práve jedna cesta z v do koreňa. Dĺžku tejto cesty nazveme hĺbkou vrcholu v a prvý vrchol za v na tejto ceste nazveme rodičom vrcholu v.
Pre každý hviezdny systém iný ako 0 si dáme vyveštiť lokálne číslo s toho jeho suseda, ktorý má byť jeho rodičom vo vyveštenom strome. (Upozorňujeme, že pod *susedom* vrcholu v vždy myslíme ľubovoľný systém, ktorý s v susedí v Hviezdnom impériu, bez ohľadu na to, či tieto systémy naozaj budú spojené niektorou z vyveštených hyperdiaľnic.)
Pre každý vrchol si ešte dáme vyveštiť jeho hĺbku d vo vyveštenom strome.
Následne každý vrchol pošle všetkým svojim susedom dva údaje: každému tú istú hodnotu d a navyše jeden bit hovoriaci, či tohto suseda považujeme za svojho rodiča v strome.
Od vyvešteného rodiča nám musí prísť vzdialenosť $d-1$. Od susedov, ktorí nám oznámia, že my sme ich rodič, nám musí prísť vzdialenosť $d+1$. Správy od ostatných susedov nás netrápia, môžu obsahovať ľubovoľné vzdialenosti – títo naši susedia môžu byť v úplne inej časti vyvešteného stromu.
Ešte musíme skontrolovať, či nemá vyveštený strom vrchol priveľkého stupňa. Vrchol 0 môže mať najviac troch susedov, ktorí ho považujú za rodiča, a každý iný vrchol najviac dvoch.
Ak riešenie existuje, je zjavné, že veštci si môžu vybrať ľubovoľný strom, ktorý je riešením. Ten vyveštia a všetky vyššie popísané kontroly úspešne prebehnú. A naopak: ak všetky systémy úspešne skontrolujú všetky vyššie popísané podmienky, tak je zjavné, že vyveštené hrany (z každého vrcholu okrem koreňa do jeho rodiča) naozaj tvoria strom a že žiaden vrchol v ňom nemá stupeň väčší ako 3.

Listing programu (Python)

```
# ak sme jediny system v celom imperiu, odpoved je ano
if N == 1: return ANO

hlbka = vyvesti_cislo(N)
if ID > 0:
    rodic = vyvesti_cislo(stupen)
    if rodic >= stupen: return NIE
else:
    rodic = -1

# posleme kazdemu susedovi nasu hlbku + bit ci je nas rodic
for s in range(stupen):
    outbox[s] = ( d, int(s == rodic) )

# zistime z prijatych sprav, ci nemame privela deti
deti = sum( x[1] for x in inbox )
if ID > 0 and deti > 2: return NIE
if ID == 0 and deti > 3: return NIE

# zistime z prijatych sprav, ci maju nasi susedia v strome spravne hlbky
if inbox[s][0] != hlbka-1: return NIE
for s in range(stupen):
    if inbox[s][1] == 1: # my sme jeho rodic
        if inbox[s][0] != hlbka+1: return NIE

# pockame a ak vesmir nie je ruzovy, nik nemal odpoved NIE, a teda vsetci vratime odpoved ANO
if ruzovy_vesmir(): return NIE
return ANO
```

Podúloha B: hyperchodníky

Keby sme túto úlohu chceli riešiť na klasickom počítači, riešenie by bolo priamočiare: spočítame stupne vrcholov a výslednú hodnotu porovnáme s $2m$.
Toto sa robí ľahko, ak vieme mať globálnu premennú na súčet stupňov. Lenže práve jedným z problémov, na ktoré narazíme pri programovaní nášho Hviezdneho impéria, je absencia globálnych premenných. Možno sa

teda oplatí zamyslieť nad tým, ako by sme našu úlohu vyriešili na klasickom počítači *ale bez použitia globálnych premenných.*
Aj toto je ešte rozumne priamočiare. Jedno možné riešenie vyzerá nasledovne: z jedného vrcholu spustíme rekurzívne prehľadávanie do hĺbky (DFS), ktoré postupne ofarbí všetky vrcholy. Základnú implementáciu tejto rekurzívnej funkcie upravíme tak, aby ako návratovú hodnotu vracala súčet stupňov vrcholov, ktoré boli počas jej behu ofarbené.
Pre každý vrchol v teda postupne rekurzívne spracujeme všetkých jeho ešte nespracovaných susedov. Pre každého z nich spravíme rekurzívne volanie a to nám vráti súčet stupňov vrcholov, ktoré sme počas neho ofarbili. Tieto čísla si sčitujeme. Keď prejdeme všetkých susedov, k tomuto súčtu ešte pridáme stupeň samotného vrchola v a výslednú hodnotu vrátime.

Naše vzorové riešenie sa bude fungovať veľmi podobne tomu, ktoré sme si práve popísali. Použijeme pri tom podobný trik ako v podúlohe A z krajského kola: Každý vrchol si dá vyveštiť číslo, ktoré sme počas výpočtu vypočítali v ňom. Následne si každý vrchol so susedmi skontroluje, či jeho číslo zodpovedá tomu, ktoré by sme vypočítali z čísel u susedov.
Presnejšie to celé bude vyzerať nasledovne:

- Dáme si vyveštiť nejaký zakorenený strom, ktorý je *kostrou* Hviezdneho impéria. (Nemusí ísť konkrétne o DFS strom, výpočet bude fungovať presne rovnako aj pre ľubovoľnú inú kostru, takže nám na tvare stromu nijak nezáleží. Môžeme si teda vybrať taký strom, ktorý budeme vedieť efektívne vyveštiť.)
- V tomto strome má každý vrchol jedného rodiča (až na koreň, ktorý nemá žiadneho) a nezáporný počet potomkov.
- Každý vrchol si dá vyveštiť, aké číslo on posielal ako návratovú hodnotu z prehľadávania.
- Každý vrchol svoje vyveštené číslo pošle všetkým susedom. Okrem toho ešte pošle aj správu, z ktorej sa bude dať skontrolovať, že sme správne vyveštili strom.
- Každý vrchol skontroluje, či jeho súčet sedí – teda sčíta hodnoty, ktoré mu prišli od jeho potomkov, pridá k tomu svoj stupeň a výsledok porovná so svojim číslom, ktoré mu na začiatku bolo vyveštené.
- Koreň porovná svoje vyveštené (a teraz už aj skontrolované) číslo s hodnotou $2m$.

Vyveštenie stromu. Existuje veľa možností, ako presne si dať vyveštiť jeden konkrétny zakorenený strom – niektoré efektívnejšie, iné menej. My si ukážeme jednu veľmi priamočiaru: budeme si veštiť strom, v ktorom je vrchol 0 koreňom. Každý iný vrchol si dá vyveštiť číslo svojho rodiča v strome a tomu potom pošle správu, že je jeho rodič.
Moment. V krajskom kole sme predsa videli, že toto môže robiť problémy. Vo vyveštených údajoch môžu vzniknúť neželané cykly – teda napr. situácia, kedy dostaneme vyveštené, že rodičom a je b, rodičom b je c a rodičom c je a. Ako skontrolujeme, že nič také nenastalo a naozaj máme platný strom?
Tu nás zachráni druhá časť kontrol. Ak by nám veštci vyveštili rodičov tak, že v nich bude nejaký cyklus, nikdy nezafunguje kontrola toho, či sú správne aj druhé vyveštené čísla. Intuitívne je to preto, že keď výpočet prejde dokola po cykle, nemôže skončiť s rovnakým číslom ako začínal.
Formálnejšie, ak by sme vo vyveštených rodičoch mali akýkoľvek cyklus, vyberme si jeden z nich. Na tomto cykle sa v každom vrchole pozrime na jeho vyveštený súčet stupňov. Nech v je ľubovoľný vrchol cyklu, v ktorom je tento súčet minimálny. Potom kontrola súčtov vo v nemôže byť úspešná: totiž v dostane od svojho potomka na cykle aspoň tak isto veľké číslo a následne k nemu pripočíta svoj (určite kladný) stupeň.

Efektívnosť riešenia. Vo vyššie popísanom riešení si dávame veštiť dve čísla. Prvé je číslom vrcholu. Toto je z rozsahu od 0 do $n-1$, stačí nám naň teda $\log_2 n$ bitov. Druhé je súčtom stupňov nejakých vrcholov. Keďže vrcholov je n a každý má stupeň nanajvýš $n-1$, toto číslo je určite menšie ako n^2 a teda naň stačí $\log_2(n^2) = 2\log_2 n$ bitov. Po pridaní príslušného zaokrúhlenia nahor na celé bity teda dostávame, že vyššie popísanému riešeniu určite stačí $k = \lceil \log_2 n \rceil + \lceil 2\log_2 n \rceil \leq 3\lceil \log_2 n \rceil$ bitov.

Pridáme ešte jednu drobnú optimalizáciu. Pre $m \geq n(n-1)/2$ zjavne môžeme rovno odpovedať `ANO`. V opačnom prípade budeme ako súčty stupňov veštiť len čísla z rozsahu od 0 po $2m$. Ak je odpoveď kladná, všetko zafunguje presne rovnako ako vo vyššie popísanom riešení. Ak je odpoveď záporná, veštci nebudú vedieť vyveštiť správne

súčty stupňov, keďže niektoré z nich (obzvlášť ten v koreni) sú väčšie ako $2m$, to ale stále znamená, že náš algoritmus zistí, že v niektorom vrchole súčet stupňov nesedí, a následne dá správnu odpoveď `NIE`.
(Toto riešenie je efektívnejšie pre malé m, ale najhorší prípad má stále rovnaký ako vyššie popísané vzorové riešenie, takže sme ho bodovali rovnako.)

Listing programu (Python)

```
# ak sme si isti, ze M je dost velke, vratime ano
if M >= N * (N-1) // 2: return ANO

# vyvestime si strom a medzivysledky na nom
# na medzivysledky nam uz staci tolko bitov ako na cislo 2*M
sucet_stupnov_v_podstrome = vyvesti_cislo(2*M+1)
if ID > 0:
    rodic = vyvesti_cislo(stupen)
    if rodic >= stupen: return NIE
else:
    rodic = -1

# posleme kazdemu susedovi nas sucet + bit ci je nas rodic
for s in range(stupen):
    outbox[s] = ( sucet_stupnov_v_podstrome, s == rodic )

# skontrolujeme, ci sucet stupnov sedi
spravny_sucet = stupen
for s in range(stupen):
    if inbox[s][1]:
        spravny_sucet += inbox[s][0]

if spravny_sucet != sucet_stupnov_v_podstrome: return NIE

# rovno kazdy vrchol skontroluje, ci mame dostatocne malo hran
if sucet_stupnov_v_podstrome > 2*M: return NIE

# pockame a ak vesmir nie je ruzovy, nik nemal odpoved NIE, a teda vsetci vratime odpoved ANO
if ruzovy_vesmir(): return NIE
return ANO
```

Iné vyveštenie stromu. Pre zaujímavosť si ukážeme ešte jeden spôsob, ako si vyveštiť strom pomocou len $\log_2 n$ bitov na vrchol. Tentokrát si budeme veštiť hĺbky vrcholov. Hlavnou myšlienkou je samozrejme že ak naša hĺbka je d, tak sused, ktorý dostal vyveštenú hĺbku $d-1$, je náš rodič, a susedia, ktorí dostali hĺbku $d+1$, sú naši potomkovia. Ale ide to vôbec? Dajú sa vždy vyveštiť hĺbky tak, aby sme z nich hľadaný strom vedeli jednoznačne zostrojiť?
Táto otázka je ťažšia ako by sa možno na prvý pohľad zdalo. Napríklad úvaha „ak som v hĺbke d a mám viac susedov, ktorí dostali vyveštenú hĺbku $d-1$, môj rodič bude ten z nich, ktorý má najmenšie číslo“ nefunguje, lebo dotyčný rodič sa už nemá ako dozvedieť, že sme si za rodiča vybrali zrovna jeho.
Trik je nasledovný: spomedzi všetkých stromov s koreňom 0 uvažujme taký, pre ktorý je súčet vzdialeností vrcholov od koreňa najväčší možný. Tvrdíme, že pre tento strom platí, že každý vrchol, ktorý má v strome hĺbku $d>0$, má medzi susedmi len jediného, ktorý má hĺbku $d-1$.
Dôkaz sporom: Nech nejaký vrchol v má (aspoň) dvoch takých susedov, x a y. Potom by sme ale vedeli upraviť strom nasledovne: zmažeme hranu z y do jeho rodiča a namiesto toho pridáme hranu z y do v. V tomto novom strome je y (a aj každý vrchol v jeho podstrome) o 2 ďalej od koreňa ako v pôvodnom strome. To je spor s tým, že už pôvodný strom mal maximálny súčet vzdialeností vrcholov od koreňa.
V každom súvislom grafe teda existuje takýto strom. To znamená, že si môžeme dať od veštcov vyveštiť hĺbky vrcholov a skontrolovať, že každý vrchol okrem koreňa má práve jedného suseda s hĺbkou o jedna menšou od svojej. Každý vrchol v hĺbke $d>0$ potom vie svojho rodiča a symetricky tiež vie, že jeho potomkami v strome sú všetci tí jeho susedia, ktorí majú vyveštenú hĺbku $d+1$.

TRIDSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Jakub Šimo
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2023