Korešpondenčný seminár z programovania
XXX. ročník, 2012/13
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola letnej časti

vzorák napísala Maja

1. Žena či muž?!

(max. 2 b za popis, 8 b za program)

Dagmar, Iľja, Gejza... Zistiť pohlavie týchto troch bolo prekvapivo najťažšou časťou úlohy. Ale poďme pekne od začiatku.

Priamočiare riešenie

Pri tejto úlohe sa dalo postupovať napríklad tak, že sme si urobili dva zoznamy, do jedného vložili všetky ženské mená, ktoré sme našli a do druhého všetky mužské. Pri písaní programu si len tieto dva zoznamy prekopírujeme do dvoch polí. Program potom pri každom mene na vstupe iba vyhľadá, v ktorom z polí sa dané meno nachádza. Takéto riešenie má pamäťovú zložitosť $O(m)$, kde m je počet všetkých mien v zozname na wikipédii. Časová zložitosť je $O(n \cdot m)$, pri každom mene na vstupe musíme prejsť celý zoznam dĺžky m a dané meno v ňom vyhľadať. Takéto riešenie by samozrejme nedostalo žiadne body za stručnosť. A našťastie všetci ste (sme) boli aj príliš leniví niečo také programovať :).

Vzorové riešenie

Pre vylepšenie priamočiareho riešenia na vzorové stačilo urobiť jedno jednoduché pozorovanie. Skoro všetky ženské mená v zozname končia na `a` a skoro žiadne mužské nekončia. A potom tu bola ešte otázka: sú nejaké výnimky? Najlepšie bolo prehľadať poriadne meno po mene zoznam a pri každej nejasnosti konzultovať s Googlom. Ak ste si napriek tomu boli istí, že Dagmar je mužské meno, fajn spôsob je tiež dať skontrolovať zoznam pár kamarátom, prípadne rodičom, starým rodičom... Ak už máte nájdené všetky tri výnimky (teda Gejza, Iľja, Dagmar), váš program sa iba spýta, či dané meno nie je výnimka, a ak nie, tak iba zistí, či je posledné písmeno v mene `a`.

Toto riešenie má časovú zložitosť $O(n)$, keďže pre každé meno urobí iba konštantný počet operácií. Pamäťová zložitosť je $O(1)$, program si počas svojho behu nemusí pamätať žiadne zoznamy mien, iba tri výnimky, ktoré môžeme prehlásiť za konštantu. Dĺžku mien môžeme v obidvoch riešeniach zanedbať, keďže všetky mená v zozname sú pravdepodobne kratšie ako povedzme 30 znakov. A časová zložitosť $O(30n)$ je to isté ako $O(n)$, pri O-notácii nás prenásobenie konštantou alebo prirátanie konštanty nezaujíma. Takéto riešenie by malo dostať aj dva body za stručnosť. Ešte si ukážeme, ako bolo treba bojovať o bonusový bod za stručnosť.

Listing programu (Pascal)

```
var n,i:integer;
        m:string;
begin
        readln(n);
        for i:=1 to n do begin
                readln(m);
                if ((m[byte(m[0])]='a') or (m='Dagmar')) and (m<>'Gejza') and (m<>'Ilja') then
                        writeln('femmina')
                else  writeln('maschio');
        end;
end.
```

Krátke vzorové riešenie

Vzorové riešenie napísané vyššie by samozrejme bonusový bod nezískalo. Jeho hlavnou chybou je, že je napísane v Pascale. Pascal je veľmi ukecaný jazyk, so všetkými svojimi `begin`mi, `end`ami, `var`mi, `integer`mi a pod. Preto sa bolo treba pozrieť, aké všetky možnosti ponúka testovač. Na napísanie takéhoto krátkeho programu nemusíte daný jazyk nejak super ovládať, stačí základná syntax, takže pokojne ste mohli skúsiť programovať aj v niečom, čo ste ešte v živote nevideli. Tak si niečo skúsme vybrať. Napríklad Java hneď vypadáva, už len kvôli `System.out.println()`. V C/C++ zase musíte urobiť `#include<...>`,`int main()`..., zase sa tam príliš veľa napíšete. Ako ideálny sa ukazuje Python, netreba sa starať o deklarácie a typy premenných, veľa vecí je v ňom elegantne skrátených a kopa užitočných funkcií predkódená, netreba ani nič importovať.

Pekné, ale trochu trikové riešenie tejto úlohy napísal v Pythone Syseľ. Má pekných 101 znakov. Najkratšie účastnícke riešenie napísal Pavel Madaj (má 107 znakov) a týmto mu patrí večná sláva a naozaj zaslúžený bonusový bod :). Začnime s trocha dlhším riešením, 111 znakovým riešením, ktoré potom vylepšíme.

Listing programu (Python)

```
p=["maschio","femmina","Dagmar","Gejza","Ilja"]
for i in range(int(input())):
    m=input()
    print(p[(m[-1]=='a') ^ (m in p)])
```

Poďme si vysvetliť, čo dané riešenie robí. Prvý riadok vytvorí pole s odpoveďami a výnimkami. Nasleduje cyklus, ktorý bude `int(intput())`-krát (teda n-krát) opakovať tretí a štvrtý riadok. Tretí riadok načíta meno do premennej m. Čo však robí štvrtý riadok? Vypisuje niečo z poľa. Ale čo?

Najskôr si vysvetlíme funkciu tej striešky (`^`) v strede riadku. Je to logická spojka ako napríklad `and` alebo `or`, ale táto sa volá `xor`. Vracia `true` alebo 1 vtedy, ak je splnená práve jedna podmienka a vracia `false` alebo 0 vtedy, keď nie je splnená žiadna alebo sú splnené obidve. No a naše dve podmienky sú: končí sa meno na `a`?, nachádza sa naše meno v zozname výnimiek? Či sa meno končí na `a`, testujeme tým, že sa pozrieme na posledné písmeno v mene. V Pythone k poslednému písmenu vieme pristúpiť pomocou indexu -1 (-2 by bol predposledný index atď). Či sa nachádza v poli sa spýtame pomocou meno `in` pole, to je celkom zjavné.

Tak a čo bude náš program teda vypisovať? Ukážeme si to na nejakých prípadoch. Na začiatok bezproblémové meno Zuzana. Toto meno sa končí na `a`, takže prvá časť podmienky vráti 1, nenachádza sa vo výnimkách, takže druhá časť vráti 0. Keďže je pravdivá práve jedna časť podmienky, tak výraz celkovo vráti 1. Takže pristúpime k $p[1]$, čo je femmina a to aj vypíšeme. A presne to sme chceli. Ak bude meno na vstupe Gejza, prvá časť podmienky vráti 1, keďže sa končí na `a`, druhá časť podmienky tiež vráti 1, keďže Gejza je výnimka. A keďže sú pravdivé obidve podmienky, xor nám vráti 0 a pristúpime k nultému prvku poľa p, čo je slovo maschio. Rovnako to bude fungovať aj na ostatných mužských alebo ženských menách (ak neveríte, urobte si tabuľku a overte :)).

Teraz poďme naše riešenie skrátiť. Máme tam dvakrát výskyt slova `input`, takže si ho v prvom riadku premenujeme na `v`, čím ušetríme jeden znak. Ďalšie znaky ušetríme tým, že si pole vyrobíme rozsekaním reťazca funkciou `split`.

Napokon sa ešte zbavíme `for`cyklu, a nahradíme ho príkazom `exec`. Príkaz `exec` dostane ako parameter reťazec znakov a jednoducho ho vykoná, akoby to bol Pythonovský kód. A mať napísaný kód ako reťazec, má veľkú výhodu, lebo ho vieme ľahko zopakovať n-krát. Napríklad „ahoj“`*4` vyrobí „ahojahojahojahoj“.

Nasledovný program robí teda presne to, čo predošlý, ale tento je dlhý len 101 znakov.

Listing programu (Python)

```
v=input
p="maschio femmina Dagmar Gejza Ilja".split()
exec('m=v();print(p[(m[-1]=="a")^(m in p)]);' *int(v()))
```

vzorák napísal Jano

2. Zázračná opička

(max. 7 b za popis, 3 b za program)

Asi nás napadne, že pohyb opičky je periodický, teda že postupnosť stoličiek, na ktoré skáče sa časom začne opakovať. Samozrejme, potrebujeme si uvedomiť, že skok dlhý $n + 1$ je to isté ako skok dlhý 1. Všeobecnejšie, z ľubovoľnej stoličky sa po skoku dlhom $k \cdot n + d$ dostaneme na rovnakú stoličku, ako po skoku dlhom d.

Pokiaľ teda chceme zistiť, na ktoré stoličky opička skočí, stačí nám napríklad odsimulovať prvých niekoľko skokov. Presnejšie toľko, aby sme si boli istí, že sa pohyb opičky ďalej bude len opakovať (a teda neskočí na žiadnu novú stoličku). Počas simulácie si (napríklad do poľa booleanov) budeme značiť, na ktoré stoličky opička skočila. Na konci iba prejdeme toto pole a spočítame počet `True` hodnôt v ňom.

Tým pádom na vyriešenie úlohy stačilo dobre odhadnúť periódu skákania – po koľkých skokoch sa opička dostane do rovnakej situácie (bude sedieť na rovnakej stoličke a bude sa chystať spraviť rovnako dlhý skok, až na pripočítanie $k \cdot n$), v akej už niekedy bola. Ak totiž poznáme stoličku a dĺžku nasledovného skoku, vieme jednoznačne určiť, čo sa bude diať ďalej.

Tak, keďže rôznych stoličiek je n a rôznych dĺžok skokov je tiež n, tak po n^2 skokoch sa opička určite dostane do už navštívenej situácie a teda stačí odsimulovať n^2 skokov. To je však príliš veľa, skúsme to zlepšiť.

Určite počet skokov, ktoré budeme musieť spraviť, bude násobok n, pretože za toľko skokov sa zopakuje rovnaká dĺžka skoku. A my k tomu ešte chceme, aby sa zopakovalo aj políčko.

Stačí sa s tým trocha pohrať, vyskúšať si skákať pre malé n a uvidíme, že sa to stane už po $2n$ skokoch. Totiž po $2n$ skokoch sme dokopy prešli vzdialenosť $1 + 2 + 3 + \cdots + 2n = 2n(2n + 1)/2 = n(2n + 1)$. Čiže sme dokopy spravili $2n + 1$ kolečiek a skončili sme na pôvodnej stoličke.

Z toho vyplýva, že pokiaľ odsimulujeme len prvých $2n$ skokov, tak opička skočí na všetky tie stoličky, na ktoré skočí aj niekedy v budúcnosti. Algoritmus na simulovanie bude mať časovú aj pamäťovú zložitosť $O(n)$ (pamätať si potrebujeme vyššie spomínané pole booleanov).

Listing programu (Pascal)

```
var n,x,i:longint;
    V:array[1..1000047] of boolean;
begin
    read(n);
    // odskaceme si prvych 2*n krokov
    for i:=1 to 2*n do begin
        V[x]:=true;
        x:=(x+i) mod n;
    end;
    // spocitame, na kolko stoliciek sme skocili
    x:=0;
    for i:=1 to n do if V[i-1] then inc(x);
    writeln(x);
end.
```

V skutočnosti však stačilo odsimulovať len n skokov, hoci sa nemusíme ocitnúť na pôvodnom políčku, symetrie spravia prácu za nás – zamyslite sa prečo. Časovú zložitosť to ale nezlepší.

Listing programu (Python)

```
import sys
n,x = int(sys.stdin.read()),0
V = [False]*n
# odskaceme si prvych n krokov
for i in range(0,n):
    x = (x + i)%n
    V[x] = True
# spocitame stolicky, na ktore sme skocili
print V.count(True)
```

Úloha sa však dala riešiť aj s oveľa lepšou časovou zložitosťou – to sme od vás nechceli, ale ako zaujímavosť sa nad tým určite oplatí zamyslieť.

Napríklad sa dá všimnúť, že opička poskáče po všetkých stoličkách práve vtedy, keď n bude mocninou dvojky. Následne sa dá zistiť aj vzorec pre mocniny iných prvočísel, dôkaz je však nad rámec tohoto vzorového riešenia. Pre kruh veľkosti p^k skočí opička na $\lfloor \frac{p^{k+1}}{2p+2} \rfloor + 1$ stoličiek, kde p je prvočíslo väčšie než dva.

Pre ostatné čísla dostaneme výsledok rozložením na mocniny prvočísel a ponásobením hodnôt pre jednotlivé mocniny. Napríklad pre $n = 150 = 2 \cdot 3 \cdot 5^2$ dostaneme $v(150) = v(2) \cdot v(3) \cdot v(5^2) = 2 \cdot (\lfloor \frac{9}{5} \rfloor + 1) \cdot (\lfloor \frac{125}{12} \rfloor + 1) = 2 \cdot 2 \cdot 11 = 44$, čo je naozaj správna odpoveď. Bez dôkazu teda uvádzame riešenie so zložitosťou $O(\sqrt{n})$.

Listing programu (C++)

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

int main(){
    int n = 0, x = 1, m;
    scanf("%d",&n);
    while(n%2==0) n/=2, x*=2;
    m = n;
    for(int i = 3; i*i<=n; ++i){
        int p = 0;
        while(m%i==0) m/=i, p++;
        x*=floor(pow(double(i),p+1.0)/(2.0*(i+1.0)))+1;
    }
    if (m>1) x*=floor(pow(double(m),2.0)/(2.0*(m+1.0)))+1;
    printf("%d\n", x);
}
```

vzorák napísal Žaba

3. Zavládne poriadok

(max. 8 b za popis, 4 b za program)

Ďalšia úloha, ktorá vyzerá škaredšie, ako nakoniec je. Dokonca aj implementácia je pomerne priamočiara, treba si len dať pozor, že dáte všade správne nerovná sa :). Ale to by takým skúseným programátorom, ako ste vy, nemalo robiť problém.

Takže ako sa popasovať s touto úlohou? No, už jej zadanie vám napovedá . . . spravte si v tom poriadok. Ako uvidíme, toto naozaj pomôže. Vieme, že chceme rozdeliť našu halu na dve časti nejakou lomenou čiarou z bodu $(0, 0)$ do bodu (w, h), pričom v jednej časti budú len balvany a v druhej len šutre. Tak poďme na to.

Upozornenie: V nasledujúcom texte sa budeme tváriť, že čiaru kladieme na kamene, aj keď je to zadaním zakázané. My tam však len tichučko využívame vlastnosť, že skutočná čiara môže byť o maličký kúsoček posunutá a nič sa nezmení, keďže určite nezasiahneme iný kameň, ktorý je vzdialený aspoň o 1.

Povedzme si najprv, že v ľavej hornej časti haly budú ležať samé šutre. (Druhý prípad sa vyrieši rovnako.) To znamená, že z pozícií šutrov vieme zostrojiť nejakú čiaru, čo ich oddeľuje.

Vyhovujúcich čiar je samozrejme viac, nič však nepokazíme, ak čiaru natlačíme čo najviac na jednotlivé šutre. Teda, ak čiara môže ísť dohora, tak pôjde a doprava ide len vtedy, keď naozaj nemá na výber. Je vcelku zjavné, že takáto čiara je istým spôsobom minimálna a každá iná čiara, ktorá odeľuje šutre, obsahuje v ľavej hornej časti aj celú túto čiaru.

Prečo je to naozaj tak, vyplynie z konštrukcie tejto čiary. Ako ju teda skonštruujeme? Tu prichádza na rad upratovanie. Zoberme si všetky šutre a utrieďme ich podľa y-ovej súradnice od najmenšej po najväčšiu. Pôjdeme teraz postupne po týchto šutroch a budeme si pamatať premennú c, ktorá bude určovať, ako ďaleko na x-ovej osi musí byť naša čiara. Na začiatku je $c = 0$ (začíname vľavo dole). Pozrime sa na prvý šuter. Až poňho sme mohli ísť stále hore. Tento šuter však musíme pokryť, preto musíme ísť doprava, až kým neprídeme k nemu. Hodnotu c nastavíme na x-ovú súradnicu tohto šutra. A znova ideme hore až po ďalší šuter. Toho x-ová súradnica je buď menšia ako c a teda nemusíme robiť nič, alebo je väčšia a musíme k nemu prísť pohybom doprava.

Takto vieme v $O(n)$ skonštruovať tú čiaru ako pozície a dĺžky úsekov, kde pôjdeme doprava. Zároveň si uvedomme, že z konštrukcie vyplýva, že na žiadnom mieste nevieme byť viac vľavo, lebo by sme odokryli nejaký šuter, tie však musíme pokryť. To znamená, že ak sa nejaký balvan predsa len nachádza v tejto oblasti, tak sa tá čiara nedá postaviť. Pre každý balvan teda potrebujeme zistiť, či je naľavo od našej minimálnej čiary.

Toto by sa dalo robiť nejakým, nie až tak pekným binárnym vyhľadávaním, ale krajšie je to robiť rovno počas vyrábania čiary. Utriedime si všetky kamene podľa y-ovej súradnice. Vždy, keď narazíme na šuter, updatneme si hodnotu c (pohyb vpravo opísaný vyššie). Ak však narazíme na balvan, pozrieme sa, či nie je viac naľavo ako aktuálna hodnota c. Ak áno, vieme, že sa čiara nedá spraviť.

Nakoniec toto isté vyskúšame aj opačne – teda balvany budú naľavo hore a zistíme, či sa aspoň jeden spôsob podarí. Časová zložitosť bude $O(n \log n)$, pretože potrebujeme utriediť všetky kamene. Pamäťová zložitosť je $O(n)$.

Poznámka k implementácii: Aby som nemusel robiť to isté dvakrát, budujem si obe čiary naraz. Jediné, čo potrebujem ošetriť je, aby som kamene spracovával v správnom poradí. To je také, že ak majú kamene rovnakú y-ovú súradnicu, ten s väčšou x-ovou má prednosť. Zamyslite sa, prečo je to tak.

Listing programu (C++)

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

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef pair<int,int> pii;
typedef pair<pii,int> tri;

bool cond(tri a, tri b) {
    if(a.first.first<b.first.first) return true;
    if(a.first.first==b.first.first && a.first.second > b.first.second) return true;
    return false;
}

int main() {
    int w,h,n;
    scanf("%d %d %d",&w,&h,&n);
    vector<tri> A;
    For(i,n) {
        int x,y,t;
        scanf("%d %d %d",&x,&y,&t);
        A.push_back(mp(mp(y,x),t));
    }
    sort(A.begin(),A.end(),cond);
    int cs=0,cb=0;
    bool v1=true,v2=true;
    For(i,n) {
        if(A[i].second == 1) {
            cs=max(cs,A[i].first.second);
            if(A[i].first.second<=cb) v1=false;
        }
        else {
            cb=max(cb,A[i].first.second);
            if(A[i].first.second<=cs) v2=false;
        }
    }
```

```
    if(v1 || v2) printf("ano\n");
    else printf("nie\n");
return 0;
}
```

vzorák napísal Jano

4. Žaba na olympiáde

(max. 5 b za popis, 10 b za program)

Pokiaľ súradnice potokov (resp. k_n) nie sú príliš veľké, tak sa dá použiť pomerne jednoduché dynamické programovanie.

Pre každé políčko si chceme spočítať, na koľko najmenej skokov sa naň vieme dostať, pričom ak sa na políčko dostať nedá, zapamätáme si hodnotu ∞. Prečo? Lebo z toho ľahko zistíme odpoveď na úlohu. Na nulté políčko sa, samozrejme, vieme dostať na 0 skokov a na zvyšné suché políčka sa dostaneme buď skokom alebo krokom.

Keď si $D[x]$ označíme najmenší počet skokov potrebných k dostaniu sa na políčko x, tak $D[0] = 0$ (to už sme si vysvetlili). Ďalej, ak na políčku x nie je potok, $D[x] = min(D[x-1], D[x-p]+1)$, lebo zoberieme výhodnešiu z možností skok a krok. Pri kroku sme sa na x dostali z políčka $x-1$ a nič nás to nestálo. Pri skoku sme sa na x dostali z $x-p$, ale musíme pripočítať jednotku k počtu skokov. Aby sme nemuseli ošetrovať špeciálne prípady, tak dodefinujeme $D[x] = \infty$, pre $x < 0$[1].

Pokiaľ na políčku x je potok, tak samozrejme $D[x] = \infty$, lebo tam sa dostať nevieme.

Na výstup nám stačí napríklad vypísať hodnotu $D[k_n + p]$, alebo -1 ak $D[k_n + p]$ je ∞. (Prečo $k_n + p$? Menšie hodnoty veľmi nechceme, lebo by sa nám mohlo stať, že Žaba políčko preskočí. Väčšie hodnoty by zasa iba zbytočne predlžovali počítanie.)

Hodnotu každého políčka takto zistíme v konštantnom čase, čo nám dokopy dáva čas $O(k_n + p)$. To nám nestačí, ba čo je ešte horšie, aj pamäťová zložitosť je $O(k_n + p)$, pretože si pamätáme hodnoty $D[x]$ pre všetky x. Poďme najprv zlepšiť tú pamäť.

Pamäťovú zložitosť zlepšíme pomerne jednoducho, stačí sa pozrieť, kde si niečo pamätáme a nepotrebujeme to. Jediná veľká štruktúra, ktorú si pamätáme je D, ale dá sa všimnúť, že v momente keď počítame $D[x]$, tak si nemusíme pamätať hodnoty $D[x-p-1]$ a skoršie, pretože už sa na ne nikdy neskôr nebudeme pýtať.

Teda z D si chceme pamätať len posledných p hodnôt, pretože nám to stačí. Lenže ako to naprogramujeme? Pomerne jednoduchý trik je namiesto $D[x]$ mať $D[x \bmod p]$. Všetky hodnoty $x \bmod p$ sú v rozsahu od 0 do p, takže bude stačiť pole veľkosti p. Navyše v okamihu, ako vypočítame hodnotu $D[x]$, hodnotu $D[x-p]$ prestaneme potrebovať, takže si ju môžeme bezpečne prepísať.

Pamäťová zložitosť sa týmto zlepší na $O(p)$. Takže ostáva zrýchliť čas, ak chceme riešiť ulohu aj pre veľké k_n.

Problémom predošlého riešenia je, že počíta hodnotu všetkých políčok a tých môže byť veľa. Takže potrebujeme nejaké políčka nepočítať. Vhodnými kandidátmi na nepočítanie sú dlhé úseky, kde sa hodnota $D[x]$ nemení. Ale tým pádom potrebujeme dlhý úsek, kde sa žiaden potok nekončí ani nezačína.

Takže nás zaujímajú dlhé mokré úseky a dlhé súše. Dlhé potoky môžeme vyriešiť ľahko, pretože akonáhle je nejaký potok dlhší ako p, tak sa nedá preskočiť a odpoveď je -1. Dlhé súše sú zasa zaujímavé tým, že sa na nich neoplatí skákať. (Ak nejakým skokom nepreskočíme potok, tak vieme skok nahradiť p krokmi a ušetriť.) Teda ak napríklad políčka x až y neobsahujú potok, tak $D[x+p], D[x+p+1] \ldots D[y]$ budú všetky rovnaké. Je to preto, že na políčkach x až $y-p$ nebude žiaden skok začínať, takže na políčkach $x+p$ až y nebude skok končiť. Na týchto políčkach sa teda hodnota D nemení.

Z každého úseku, ktorý je dlhší ako $2p$ teda stačí vypočítať prvých p hodnôt a najmenšiu hodnotu skopírovať na posledných p políčok úseku. Medzi každými dvoma potokmi teda spočítame hodnoty najviac $2p$ políčok, každý potok je dlhý najviac p alebo je odpoveď -1, takže celkovo pre n potokov stačí spočítať najviac $3n \cdot p = O(p \cdot n)$ políčok. A zhodou okolností[2] bolo v zadaní obmedzenie na súčin p a n.

Stačí to naprogramovať a máme plnopočtové riešenie s časovou zložitosťou $O(p \cdot n)$, resp. $O(min(p \cdot n, k_n + p)$ a pamäťovou $O(p)$.

Listing programu (C++)

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

[1] Navyše si všimnime, že použitím ∞ sme si uľahčili robotu a ošetrovanie ďalších prípadov, lebo minimum nám zachová potrebné vlastnosti (napr. $min(4, \infty) = 4$, teda keď sa na nejaké políčko viem dostať krokom ale skokom nie, tak sa naozaj použije hodnota pre krok). V programovaní samozrejme ∞ nemáme, zvykne sa definovať v konštante na začiatku programu ako dosť veľké číslo.

[2] Alebo úmyselne?

```
#define For(i, n) for(int i = 0; i<(n); ++i)
#define LINF 1023456789123456789LL
typedef long long ll;

ll D[1000047];
ll n,p,pos, z,k;
void neda_sa(){ printf("-1\n"); exit(0); }

int main(){
    scanf("%lld%lld",&n,&p);
    For(i,p) D[i] = LINF;
    D[0] = 0;
    pos = 1;
    For(i,n){
        scanf("%lld%lld",&z,&k);
        // ak je tam široký potok, nedá sa
        if (k-z > p) neda_sa();
        // široké súše rátame rýchlejšie
        if (z-pos > 2*p){
            ll najmenej = LINF;
            For(j,p) najmenej = min(najmenej, D[j]);
            najmenej+=(i>0);
            if (najmenej >= LINF) neda_sa();
            For(j,p) D[j] = najmenej;
            pos = z-1;
        }
        // úzka súš:
        while(pos<z) {
            D[pos%p] = min(D[(pos+p-1)%p], D[pos%p]+1);
            pos++;
        }
        // úzky potok:
        while(pos<k) {
            D[pos%p] = LINF;
            pos++;
        }
    }
    // vypočítame odpoveď
    D[pos%p] = min(D[pos%p]+1, D[(pos+p-1)%p]);
    For(i,p) D[pos%p] = min(D[pos%p], D[(pos+i)%p]+1);
    printf("%lld\n", (D[pos%p]>=LINF)?-1LL:D[pos%p]);
}
```

vzorák napísal Mišof

5. Opakovanie – matka múdrosti

(max. 0 b za popis, 16 b za program)

V každej podúlohe výsledné riešenie uvádzam v podobe, v akej sa odovzdávalo – teda v prvom riadku je reťazec, z ktorého sa začína, a v druhom riadku je kompletný program pre sed.

Podúloha 1: mocniny dvoch

Tak tu sa ešte príliš nebolo treba zamýšľať. Začnem s jedným písmenom a v každom kole potrebujem počet písmen zdvojnásobiť. To ide zjavne tak, že každé písmeno nahradíme dvomi jeho kópiami.

```
a
s/a/aa/g
```

Podúloha 2: cyklus dĺžky 50

Toto tiež ešte ide ľahko. Začneme s `a` a potom v každom kole pridáme jedno ďalšie a ak už ich je 51, tak ich všetky nahradíme jedným a začína sa odznova:

```
a
s/^/a/ ; s/a{51}/a/
```

Šlo to samozrejme aj všelijako obšírnejšie, napríklad tak ako uvádzame nižšie, vyššie uvedené riešenie je však krajšie a stručnejšie.

```
b
s/b/ab/ ; s/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab/b/
```

Podúloha 3: zdvojnásob počet len raz za tri kolá

Chceme postupnosť dĺžok 1, 1, 2, 2, 2, 4, 4, 4, ... Použijeme na to tri rôzne písmená. Budeme teda mať k kusov `a`, tie prepíšeme na k kusov `b`, tie na k kusov `c` a tie potom na $2k$ kusov `a` (každé `c` sa zmení na `aa`).

Treba si dať trochu pozor na to, v akom poradí robíme jednotlivé prepisovania. My sme pri tom použili pomocné písmeno `d`, pozrite sa ako:

```
b
s/b/d/g ; s/a/b/g ; s/c/aa/g ; s/d/c/g
```

Podúloha 4: štvorce

Budeme používať tri písmená: `a`, `b` a `c`. Písmeno `c` bude stále jedno, písmen `b` bude v n-tom reťazci presne $n-1$, a písmená `a` budú tvoriť zvyšok – bude ich teda $n^2 - n$.

Ako sa majú počty písmen zmeniť, keď vyrábame reťazec $n+1$ z reťazca n? Má pribudnúť jedno `b` a má pribudnúť $((n+1)^2 - (n+1)) - (n^2 - n) = 2n$ kusov písmena `a`.

Výsledný program:

```
c
s/b/aab/g ; s/c/aabc/
```

Podúloha 5: k reťazcov dĺžky k

V tejto podúlohe prvýkrát využijeme zložitejšie príkazy `sedu`, konkrétne príkaz `t`. Príkaz `t` (trochu zjednodušene, ako uvidíme neskôr) znamená „ak sa už podarilo niečo prepísať, ukonči vykonávanie programu".

V našom riešení budeme postupne vyrábať nasledujúce reťazce: `z`, `az`, `za`, `aaz`, `aza`, `zaa`, `aaaz`, `aaza`, `azaa`, `zaaa`, `aaaaz`, … Vo väčšine prípadov teda robíme jednoduchú zmenu: posunieme `z` o pozíciu doľava. To vieme spraviť tak, že prvý (a jediný) výskyt `az` zmeníme na `za`. Ak sa táto zmena nepodarí, vieme, že máme reťazec tvaru `za*`. V takomto prípade najskôr prepíšeme `z` na `aa` (čím zväčšíme dĺžku o 1) a následne posledné `a` na `z`.

```
z
s/az/za/ ; t ; s/z/aa/ ; s/a$/z/
```

Podúloha 6: Collatzova postupnosť

V tejto podúlohe sme mali vyrobiť postupnosť definovanú nasledovne: Prvé číslo je 97. Po čísle k vždy nasleduje $k/2$ ak je k párne, resp. $3k+1$ ak je k nepárne.

Takéto postupnosti sa podľa matematika, ktorý ich skúmal, volajú Collatzove. Každá známa postupnosť dosiahne po konečnom počte krokov číslo 1. Ale dokázať, že je to tak vždy, zatiaľ nik nevie – ide o jeden z veľmi starých otvorených problémov v matematike.

Pri konštrukcii tejto postupnosti `sedom` si vystačíme s písmenom `a`. Opäť použijeme príkaz `t`, no tentokrát v jeho všeobecnejšej verzii: príkaz `t` totiž môže ako parameter dostať názov labelu (návestia), na ktorý má skočiť, ak sa už nejaké prepísanie podarilo. Label s názvom `x` značíme `:x`.

Prepisovanie teda začneme tým, že si zistíme, či máme párny počet `a`. Ako to vieme spraviť? Napríklad takto: „`s/^((aa)*)$/\1/`". Ak je v reťazci párny počet `a`, celý ho označíme a vďaka spätnej referencii ho prepíšeme na jeho samého. Ak je počet `a` nepárny, prepísanie sa nepodarí.

No a následne si príkazom `t` vyberieme, ako budeme prepisovať: či zmenšíme počet `a` na polovicu, alebo naopak každé strojíme a ešte jedno pridáme.

Celé riešenie:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
s/^((aa)*)$/\1/ ; t x ; s/a/aaa/g ; s/$/a/ ; t ; :x ; s/aa/a/g
```

Program ešte môžeme skrátiť a zjednodušiť pomocou ďalšej syntaxe jazyku `sed`. Konkrétne vieme o príkaze povedať, že sa má vykonať len pre riadky, ktoré zodpovedajú danému regulárnemu výrazu. Keď napr. napíšeme „`/vyraz/ s/x/y/`", tak sa zámena prvého `x` na `y` vykoná len vtedy, ak aktuálny riadok zodpovedá vzorke `vyraz`.

My teda dostávame nasledovné riešenie:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
/^(aa)*$/ s/aa/a/g ; t ; s/a/aaa/g ; s/$/a/
```

Podúloha 7: logaritmus

Táto podúloha bola ľahká, len si bolo treba uvedomiť, aké reťazce majú dĺžku priamo úmernú logaritmu: obyčajné čísla. A keď ide o logaritmus so základom 2, tak ide o čísla v dvojkovej sústave. Správne dĺžky má teda postupnosť reťazcov `1`, `10`, `11`, `100`, `101`, `110`, `111`, `1000`, … alebo, keďže máme používať písmená a nie cifry, tak postupnosť `b`, `ba`, `bb`, `baa`, `bab`, `bba`, `bbb`, `baaa`,…

Máme teda vyrobiť obyčajné počítadlo: v každom kroku chceme „pripočítať 1" k aktuálnemu reťazcu. Ako na to? Potrebujeme všetky koncové 1 zmeniť na 0, a 0 pred nimi zmeniť na 1. Plus je tam špeciálny prípad: tá 0 tam občas nemusí byť, napr. keď ideme z 111 na 1000.

Skôr, než sa pustíme do riešenia, povieme si ešte jednu vec o užitočnom príkaze `t`: každým jeho použitím sa resetuje to, či už úspešne prebehlo nejaké prepísanie. Navyše existuje príkaz `T`, ktorý robí to isté ako `t`, len sa aktivuje, ak sa dovtedy nič prepísať nepodarilo. Úplne správny význam príkazu `t` je teda: „ak sa od začiatku spracúvania aktuálneho riadku, resp. od posledného príkazu `t` alebo `T`, už podarilo niečo prepísať, ...“

V našom riešení postupne spravíme nasledovné:

- Pridáme na koniec reťazca jeden `@`.
- Kým vidíme `b@`, prepíšeme ho na `@@` a opakujeme (pomocou príkazu `t`).
- Ak vidíme `a@`, prepíšeme ho na `b@`, inak nastal špeciálny prípad spomínaný vyššie (začínali sme z `bb...b`) a teda musíme jedno `b` na začiatok pridať.
- Zmažeme jeden `@` – jeden je navyše, lebo sme ho na začiatku pridali.
- Všetky `@` prepíšeme na `a` a sme hotoví.

Celé riešenie:

```
b
s/$/@/ ; T ; :x ; s/b@/@@/ ; t x ; s/^@/b@/ ; s/a@/b@/ ; s/@// ; s/@/a/g
```

Všimnite si, ako sme použili `T` na vymazanie toho, že prepísanie `s/$/@/` skutočne prebehlo. Následne sa cyklus „`:x ; s/b@/@@/ ; t x`“vykoná len toľkokrát, koľkokrát skutočne prepisovanie prebehne. (Ale v skutočnosti tam ten príkaz `T` netreba. Jediné, čo sa stane, ak ho zmažeme: na reťazcoch končiacich na `a` sa náš cyklus raz vykoná naprázdno.)

Podúloha 8: prvočísla

Najskôr sa zamyslime nad pomôckou zo zadania. Podľa tej reťazce tvorené samými `a`, ktorých dĺžka je zložené číslo, vieme rozpoznať napr. regulárnym výrazom „`^(aa+)\1+$`“. Prečo je tomu tak? Prvá časť nám označí skupinu niekoľkých, ale aspoň dvoch, `a`, no a následne druhá časť (`\1+`) hovorí, že presne rovnaký úsek `a` sa tam musí ešte niekoľkokrát, ale aspoň raz, zopakovať. Vhodné rozdelenie reťazca na prvú a druhú časť teda zjavne existuje vtedy a len vtedy, ak má jeho dĺžka n nejakého deliteľa d iného ako 1 a n. Prvú skupinu bude tvoriť d kusov `a`, a táto skupina sa potom ešte $((n/d) - 1)$-krát zopakuje.

Pre tento regulárny výraz sa dá pomerne ľahko rozpoznať, či mu daný reťazec zodpovedá – je len lineárne veľa možností, čo môže byť v zátvorke. Knižnica, ktorú na regulárne výrazy používajú `grep` a `sed`, však robí všeobecnejšiu konštrukciu, ktorá si s týmto výrazom nevie dobre poradiť. Môžete si to predstaviť tak, že v knižnici prebieha akýsi neoptimálny backtracking, ktorý niektoré možnosti skúša zbytočne veľakrát.

V našom riešení teda musíme knižnici nejak pomôcť. Napríklad tak, že do reťazca vložíme jedno `b`, ktoré bude označovať hranicu medzi prvou a druhou časťou. Výraz `^(aa+)b\1+$` už knižnica rozpozná ľahko: je jasné, kde je `b`, a teda aj čo tvorí prvú a čo druhú časť.

No a ako teraz pomocou `sed`u z jedného prvočísla vyrobíme nasledujúce? Základná logika je jasná: pridaj `a`, otestuj či máš zložené číslo, a ak áno, začni odznova. Jediné, čo je tu ťažké na implementáciu, je spomínaný test, či je číslo zložené.

Na ten použijeme konštrukciu podobnú podúlohe 5: pridáme do reťazca jedno `b`, postupne vyskúšame všetky jeho polohy a ak hociktorá z nich zafunguje, tak máme zložené číslo.

Kontrolu konkrétnej polohy `b` spravíme nasledovne: „`s/^(aa+)b(\1+)$/\1\2/`“. Pri úspešnej kontrole ho rovno zmažeme, pri neúspešnej sa nič nestane.

Skúšanie jednotlivých polôh `b` bude prebiehať v cykle: „`:z ; s/ba/ab/ ; T w ; *** ; t x ; T z ; :w`“. Na mieste označenom `***` je vyššie uvedená kontrola. Kód treba čítať ako cyklus `:z ... T z`. V každej iterácii najskôr príkazom `s/ba/ab/` posunieme `b` o jedno doprava. Ak sa toto nepodarilo, máme už `b` na konci reťazca, cyklus má skončiť – tak z neho príkazom `T w` vyskočíme. Ak sa `b` posunúť podarilo, spustíme kontrolu, či máme zložené číslo. Ak tá uspela, skočíme na label `x`. (Ten sme zatiaľ nevideli, ale ako uvidíme neskôr, tento skok znamená, že ideme zvýšiť počet `a`). A ak kontrola neuspela, príkaz `T z` spustí ďalšiu iteráciu tohto cyklu.

No a ostáva vonkajší cyklus: „`:x ; s/^/ab/ ; &&& ; s/b//`“. Na mieste označenom `&&&` je vyššie popísaná kontrola, či je v aktuálnom reťazci počet `a` zložené číslo. Hlavný cyklus začneme tým, že do reťazca pridáme jedno nové `a` a následne doň na začiatku vložíme jedno `b`, ktoré potom posúvame pri kontrole. Obe tieto veci my spravíme jedným príkazom `s/^/ab/`. No a akonáhle niekedy kontrola ani pre jednu polohu `b` neuspeje, máme nový prvočíselný počet `a`, môžeme zmazať `b` a skončiť.

Celý program:

```
aa
:x ; s/^a/aab/ ; :z ; s/ba/ab/ ; T w ; s/^(aa+)b(\1+)$/\1\2/ ; t x ; T z ; :w ; s/b//
```

Výzva na záver: Tento program zjavne skúša zbytočne veľa polôh `b`. Viete ho výrazne urýchliť?

vzorák napísal Bob

6. Obrovské pavúky

(max. 13 b za popis, 7 b za program)

Nestabilné bunky spočítame pre každý sektor osobitne. Vezmime si sektor i a jeho dvoch susedov $i-1$ a $i+1$ (pre jednoduchosť budeme vynechávať operáciu *modulo* n, ktorá by nám zaručila, že číslo sektora je z množiny $0, 1, \ldots, n-1$). Aby sme zistili, či je bunka sektora i stabilná, potrebujeme poznať počty krajných bodov premostení, ktoré ležia na jej hraniciach so sektormi $i-1$ a $i+1$.

Hranica j-tej bunky je úsečka na hlavnom vlákne, ktorá začína vo vzdialenosti $d_{i,j-1}$ od stredu pavučiny (prípadne priamo v strede, ak ide o prvú bunku) a končí vo vzdialenosti $d_{i,j}$. Keď teda preberieme všetky premostenia v susednom sektore, ľahko pre každé zistíme, či jeho krajný bod leží na hranici bunky j. Na overenie, či je bunka stabilná, by sme takto potrebovali čas $O(k_{i-1} + k_{i+1})$. Celý sektor i by sme potom spracovali v čase $O(k_i \cdot (k_{i-1} + k_{i+1}))$. V najhoršom prípade by mohla mať pavučina väčšinu premostení v dvoch susedných sektoroch, vtedy by naše riešenie malo celkovú časovú zložitosť až $O(k^2)$, kde k je počet premostení v celej pavučine.

Premostenia sú v rámci jedného sektora usporiadané podľa vzdialenosti od stredu. Vďaka tomu ich nemusíme lineárne preberať všetky – stačí binárnym vyhľadávaním nájsť začiatok a koniec intervalu premostení, ktoré susedia s j-tou bunkou. Čas spracovania jedného sektora sa nám tak zlepší na $O(k_i \cdot (\log k_{i-1} + \log k_{i+1}))$. Celková časová zložitosť bude $O(\sum k_i \cdot (\log k_{i-1} + \log k_{i+1})) \subseteq O(\log k \cdot \sum k_i) \subseteq O(k \log k)$.

Posledné zrýchlenie: Uvedomme si, že premostenia susediace s bunkou $j+1$ nasledujú hneď za premosteniami susediacimi s bunkou j. To znamená, že začiatok ich intervalu už nemusíme hľadať, a že koniec môžeme nájsť obyčajným lineárnym prechodom. Presnejšie, bunky i-teho sektora budeme spracovávať postupne, podľa rastúcej vzdialenosti od stredu. Pre každý zo susedných sektorov $i-1$ a $i+1$ si budeme pamätať index posledného spracovaného premostenia (teda takého, ktoré ako posledné susedilo s nejakou už spracovanou bunkou). Tieto indexy potom budeme inkrementovať dovtedy, kým zodpovedajúce premostenia susedia s aktuálnou bunkou.

Spracovanie jednej bunky teraz síce môže trvať až $O(k_{i-1} + k_{i+1})$, ale celý sektor zvládneme spracovať v čase $O(k_{i-1} + k_i + k_{i+1})$, keďže pamätané indexy susedných premostení môžeme inkrementovať len $(k_{i-1} + k_{i+1})$-krát. Toto vzorové riešenie má teda celkovú časovú zložitosť $O(k)$.

Listing programu (C++)

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

int main() {
        int n;
        scanf("%d", &n);
        vector<vector<int> > A(n);
        for (int i = 0; i < n; ++i) {
                int k;
                scanf("%d", &k);
                A[i].resize(k);
                for (int j = 0; j < k; ++j)
                        scanf("%d", &A[i][j]);
        }
        int res = 0;
        for (int i = 0; i < n; ++i) {
                int prev = (i + n - 1) % n, prev_j = 0;
                int next = (i + 1) % n, next_j = 0;
                for (int j = 0; j < (int) A[i].size(); ++j) {
                        int diff = 0;
                        while (prev_j < (int) A[prev].size() && A[prev][prev_j] < A[i][j])
                                ++prev_j, ++diff;
                        while (next_j < (int) A[next].size() && A[next][next_j] < A[i][j])
                                ++next_j, --diff;
                        if (diff)
                                ++res;
                }
        }
        printf("%d\n", res);
}
```

7. Opustené oddelenie ochranných odevov okupované oddychujúcou Olíviou

vzorák napísal Žaba

(max 13 b za popis, 7 b za program)

Ďalší pekný príklad s Olíviou, a dokonca nie ani taký ťažký. Poďme však ale pekne poporiadku. Úloha od nás chce, aby sme vybrali nejakú podmnožinu prvkov, ktorých súčet vitamínovosti bude v intervale od d po h a zároveň budú mať čo najväčšiu cenu. To sa priam núka, aby sme naozaj pozreli všetky podmnožiny.

Samozrejme vieme, že počet podmnožín je 2^n, čo by bola (prinajlepšom) aj zložitosť nášho riešenia, a to je vo väčšine prípadov príliš pomalé. Keď sa však pozrieme na limity, vidíme, že $n \leq 32$, takže by to možno nemusel byť až taký zlý nápad...

Prezeranie podmnožín

Ostáva teda problém, ako rýchlo prezrieť všetky podmnožiny. Na rýchlosť sa už nehráme, keďže vieme, že to bude trvať aspoň $O(2^n)$. Ostáva však otázka, či sa to dá implementovať pekne a jednoducho. Samozrejme, že dá – trik je nasledovný.

Priraďme krabičkám vitamínov čísla od 0 po $n-1$, napríklad podľa poradia, v akom boli na vstupe. Zoberme si teraz nejakú množinu vitamínov. Nech $n = 5$, množina by mohla vyzerať takto: vyberiem nultú, tretiu a štvrtú krabičku vitamínov. To sa však dá zapísať aj ako reťazec čísiel 0 a 1, kde 1 predstavuje vybratie príslušnej krabičky a 0 nevybratie. Dostaneme teda reťazec 10011 (vľavo sú malé pozície). Takto sa dá zapísať ľubovoľná množina a bude mať priradený jednoznačný reťazec.

Na tento reťazec sa však dá pozerať aj inak. Je to predsa číslo v binárnom zápise, v našom prípade (ak je najmenej významný bit naľavo) je to číslo 25. Takže vieme spraviť aj obrátené tvrdenie a to, že každé číslo od 0 po $2^n - 1$ reprezentuje jednu podmnožinu n prvkov. No a ak vieme zistiť, aká je to množina, len sčítame cenu a vitamínovosť krabičiek v nej, zistíme, či spĺňajú podmienky, a nájdeme najcennejšiu. Tu si môžete pozrieť časť programu, ktorá prezerá všetky podmnožiny (pozrite sa hlavne na bitové operácie, ktorými sa zisťuje, či je daný prvok v množine reprezentovanej číslom i).

Listing programu (C++)

```
//mam pole A obsahujúce n prvkov, chcem pozrieť všetky podmnožiny
for(int i=0; i<(1<<n); i++) { // << je operácia shiftleft
    for(int j=0; j<n; j++) { //hľadám prvky, ktoré sú v množine reprezentovanej i
        if(i&(1<<j)) { //ak na j-tej pozícii binárneho čísla i je 0, and vráti 0 = false
            //j-ty prvok je v množine i, spravím čo potrebujem
        }
    }
}
```

Zložitosť tohto riešenia je $O(n \cdot 2^n)$. Ale aj keď je číslo n len do 32, stále to bude pomalé – reálne to bude stíhať pre $n \leq 20$. Treba teda ešte niečo zlepšiť.

Meet in the middle

Táto technika je občas použitá na súťažiach a my si teraz vysvetlíme, v čom spočíva. Je dobré si zapamätať, že sa vyskytuje spolu s n medzi 30 až 50. Nie vždy je to tak, ale občas to je dobrý hint :).

Najskôr si trochu zjednodušíme našu úlohu. Povieme si, že nebudeme chcieť vitamínovosť medzi d a h, ale že chceme, aby sa presne rovnala h.

To čo robí technika Meet in the middle je nasledovné: namiesto toho, aby rátala všetky podmnožiny, povie si, že každú množinu vie dostať ako spojenie dvoch nezávislých množín – jedna je vytvorená z časti prvých $n/2$ prvkov, druhá je vytvorená z časti zvyšných $n/2$ prvkov (pri nepárnom n je jedna množina väčšia). Zrátať $2^{n/2}$ podmnožín nám ale pre $n \leq 32$ nerobí žiaden problém. Samozrejme, ak by sme teraz spájali každú podmnožinu s každou inou, nikam by sme sa nepohli, lebo by to stále trvalo $O(2^n)$. My však máme informáciu naviac: ak si totiž vyberieme nejakú množinu z prvej polovice a súčet vitamíností v nej je x, tak z druhej polovice chceme vybrať množinu s vitamínovosťou $h - x$. Stačí teda pre druhú polovicu zrátať raz každú z jej $2^{n/2}$ podmnožín a pre každú vitamínovosť si zapamätať najväčšiu cenu – napríklad uložením do mapy. Potom sa nám pre množiny z prvej polovice stačí pozrieť na jedno konkrétne miesto a máme vyhraté. Celková zložitosť – $O(\log n \cdot 2^{n/2})$.

Už to len dokopnúť

A už máme všetko, čo potrebujeme, akurát neriešime pôvodnú úlohu. Tam sme mali interval od d po h. To nám však vraví, že ak si vyberieme z prvej polovice množinu s vitamínovosťou x, z druhej chceme vybrať najdrahšiu množinu, ktorej vitamínovosť je z intervalu $d - x$ až $h - x$. A to je jeden súvislý interval, ktorého maximum vieme zistiť pomocou intervalového stromu. V tomto prípade súradnice stromu budú vitamínovosť množín a budeme doňho ukladať ceny množín, pričom sa pýtame vždy na maximum. Samozrejme nebude to také ľahké – keďže vitamínovosť sa pohybuje v rozmedzí až do 10^{18} nemôžeme si intervaláč pamätať len tak. Tu sa však dá použiť ďalší klasický trik a to prečíslovanie. To znamená, že si nahradíme vitamínovosti novými číslami z rozsahu 1 až $2^{n/2}$, čo sa nám už zmestí do rozumne veľkého poľa. Na efektívne pamätanie, čo som na čo prečísloval, použijeme mapu. Keďže vloženie aj otázka nás stoja logaritmický čas, časová zložitosť bude $O(\log 2^{n/2} \cdot 2^{n/2})$ čo je vlastne $O(n2^{n/2})$.

Ešte treba dodať, že na hľadanie maxima intervalu sa dá použiť tiež algoritmus RMQ (range minimum/maximum query), ktorý nám ale dá rovnakú zložitosť, aj keď vie niektoré operácie robiť rýchlejšie. Musíme si totiž naše čísla usporiadať podľa vitamínovosti a to nám zaberie čas $O(n \cdot 2^{n/2})$.

Toť vše.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <map>
#include <cmath>
using namespace std;
#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
#define INF 100000000000000004711
typedef long long ll;
typedef pair<ll,int> pli;

pli Tree[1000047];
int treesize=1;

void setValue(int vertex, pli value) {
    if(vertex>=treesize) Tree[vertex]=value;
    else Tree[vertex]=max(Tree[2*vertex],Tree[2*vertex+1]);
    if(vertex!=1) setValue(vertex/2,value);
}

pli getValue(int vertex, int from, int to, int treefrom, int treeto) {
    if(from<=treefrom && to>=treeto) return Tree[vertex];
    if(to<=treefrom || from>=treeto) return mp(-1,0);
    int middle=(treefrom+treeto)/2;
    return max(getValue(2*vertex,from,to,treefrom,middle),getValue(2*vertex+1,from,to,middle,treeto));
}

int main() {
    int n;
    ll d,h;
    scanf("%d %lld %lld",&n,&d,&h);
    vector<ll> V(n),C(n);
    For(i,n) scanf(" %lld %lld",&V[i],&C[i]);
    if(n==1) {
        if(d<=V[0] && h>=V[0]) printf("1\n1\n");
        else printf("0\n");
        return 0;
    }
    int half=n/2;
    vector<pair<ll,pli> > A;
    For(i,1<<half) {
        ll v=0,c=0;
        For(j,half)
            if(i&(1<<j)) {
                v+=V[j]; c+=C[j];
            }
        A.push_back(mp(v,mp(c,i)));
    }
    A.push_back(mp(INF,mp(-1,0)));
    sort(A.begin(),A.end());
    int count=0;
    for(int i=1; i<A.size(); i++) if(A[i].first!=A[i-1].first) count++;
    count++;
    while(count>treesize) treesize*=2;
    For(i,2*treesize+47) Tree[i]=mp(-INF,0);
    map<ll,int> M;
    int pom=0;
    For(i,A.size()) {
        if(M.find(A[i].first)!=M.end()) setValue(treesize+M[A[i].first],A[i].second);
        else {
            setValue(treesize+pom,A[i].second);
            M[A[i].first]=pom++;
        }
    }
    int rest=n-half;
    ll res,maxi=0;
    For(i,1<<rest) {
        ll v=0,c=0;
        For(j,rest)
            if(i&(1<<j)) {
                v+=V[half+j]; c+=C[half+j];
            }
        ll b=d-v,e=h-v;
        int beg=(*M.lower_bound(b)).second,end=(*M.upper_bound(e)).second;
        if(e<0 || beg==pom-1 || beg==end) continue;
        pli get=getValue(1,beg,end,0,treesize);
        if(c+get.first>maxi) {maxi=c+get.first; res=((ll)i<<half)+get.second;}
    }
    vector<int> Res;
    For(i,n) if(res&(1ll<<i)) Res.push_back(i);
    printf("%d\n",Res.size());
    For(i,Res.size()) printf("%d%c",Res[i]+1,((i!=Res.size()-1)?' ':'\n'));
}
```

vzorák písal Usámec

8. Olepený ohybný pásik

(max. 13 b za popis, 12 b za program)

Ohýbanie pásika si môžeme predstaviť ako vytváranie špirály. Presnejšie, keďže môžeme pásik ohýbať z oboch koncov, tak máme dve špirály – jednu z každej strany, pričom každá zo špirál začína neohnutým kúskom dlžky aspoň a, resp. aspoň b.

Špirála sa dá popísať postupnosťou dĺžok jednotlivých neohnutých kúskov. Majme m čísel $a_1, a_2, \ldots, a_m$. Pre dĺžky týchto kúskov musí platiť: $a_1 \leq a_2 \leq \cdots \leq a_m$.

Poďme najprv vyriešiť nasledovný problém: koľko existuje špirál z n častí, pričom najmenší kúsok má dĺžku aspoň k? Odpoveď na túto otázku označme ako $S[n][k]$.

Toto vieme spočítať pomocou dynamického programovania. Ak $k > n$, tak $S[n][k] = 0$. Ak $n = k$, tak $S[n][k] = 1$. Inak môžeme ohnúť práve k častí pásika, alebo ohnúť aspoň $k+1$. Toto vieme zapísať ako $S[n][k] = S[n-k][k] + S[n][k+1]$.

Teraz by si tí naivní medzi nami mohli myslieť, že na nájdenie výsledku stačí spočítať nasledovné: výskúšajme prestrihnúť každý pásik na každom mieste a spočítať počet možností, ako spraviť špirálu vľavo a vpravo. Toto je celkom nanič, lebo napríklad neohnutý pásik započítame viac ako raz.

Bolo by fajn aby sme špirálu trochu obmedzili. V predchádzajúcom prípade napríklad špirály s dĺžkami $1, 3$; $2, 4$ a špirály s dĺžkami $1, 4$; $2, 3$ viedli k rovnakému riešeniu. (Pásik by bol pre obe možnosti ohnutý rovnako.)

Problémy robia posledné kúsky, ktorými špirály potom spojíme. Presnejšie, ak máme nejakú špirálu a predĺžime jej posledný kúsok, tak tým nevyrobíme novú možnosť. (Teda napr. špirály $2, 3, 3$; $2, 3, 4$; $2, 3, 5$;... vedú k rovnako ohnutému pásiku, lýšia sa len tým, koľko častí spotrebujúi.) Preto si zo všetkých možností vyberieme len tú s najmenšou poslednou časťou a pri počítaní možností budeme celý pásik deliť nie na dve časti, ale na tri – ľavá špirála, nepoužitý úsek, pravá špirála.

Zaveďme si teda pojem pekná špirála, čo bude taká, kde dĺžka posledného kúska bude rovnaká ako dĺžka predposledného kúska, t.j. $a_{m-1} = a_m$. Všimnite si, že pekná špirála je aspoň raz ohnutá. Opäť si budeme klásť otázku, koľko existuje pekných špirál z n častí, ktorých najmenší kúsok má dĺžku aspoň k. Toto vieme tiež riešiť dynamickým programovaním; hodnotu odpovede označme ako $T[n][k]$. Zoberieme dynamické programovanie pre počet špirál ale miesto $S[n][n] = 1$ položíme $T[2n][n] = 1$.

Teraz už môžeme rátať celkový počet možností. Môžu nastať nasledovné prípady:

- Pásik neohýbame: 1 možnosť
- Pásik ohýbame iba z jednej strany. $\sum_{1 \leq i \leq n-b} T[i][a] + \sum_{1 \leq i \leq n-a} T[i][b]$.
- Pásik ohýbame z oboch strán $\sum_i \sum_j T[i][a]T[n-i-j][b]$. T.j. jednu špirálu urobíme z i častí, necháme j častí v strede a druhú urobíme zo zbytku.

Celá táto procesia má časovú a pamäťovú zložitosť $O(n^2)$.

Listing programu (C++)

```
#include <cstdio>

int mod = 10301;

int T[1050][1050];

// memoizacia je v nasom pripade pohodlnejsia ako dynamika
long long GetT(int n, int k) {
  if (n < 2*k) return 0;
  if (n < 0 || k < 0) return 0;
  if (T[n][k] != -1) return T[n][k];

  if (n == 2*k) T[n][k] = 1;
  else T[n][k] = (GetT(n-k, k) + GetT(n, k+1)) % mod;
  return T[n][k];
}

int main() {
  for (int i = 0; i < 1050; i++)
    for (int j = 0; j < 1050; j++)
      T[i][j] = -1;
  int n, a, b;
  scanf("%d %d %d", &n, &a, &b);

  long long res = 1;
  for (int i = 1; i <= n; i++) {
    if (i <= n - b)
      res += GetT(i, a);
    if (i <= n - a)
      res += GetT(i, b);
    res %= mod;
  }
```

```
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j + i < n; j++) {
      res += GetT(i, a) * GetT(n-i-j, b);
      res %= mod;
    }
  }
  printf("%lld\n", res);
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09