



Vzorové riešenia 1. kola letnej časti

Sebastián

1. Malá mäťúca prechádzka

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

Pre naše riešenie sa najprv musíme pozrieť na prípad, kedy prídeme odkiaľ sme prišli. To dovŕšime iba tým, že sa posunieme rovnako hore-dole a doprava-dolava. Súradnice si môžeme zapisovať do dvoch premenných x, y korešpondujúce s danými osami a pre jednoduchosť výpočtu začnime na súradnici $x = 0, y = 0$.

Kroky môžeme hodnotiť nasledovne:

1. $H=y+1$
2. $D=y-1$
3. $P=x+1$
4. $L=x-1$

Zaujímavá myšlienka

Musíme zadať **vzdialenosť od začiatočného bodu**. Bude to je súčet absolútnych hodnôt našich súradníc ($|x| + |y|$). Napríklad by sme mali cestu "DP". Vzďialenosť od začiatku by bola 2, lebo $|1| + |-1| = 2$. Nemusíme kontrolovať a počítať kombinácie možností, kde sme sa mohli pomýliť (to by sme sa zbytočne narobili). Keď v ceste urobíme nejakú chybu a zle zabočíme, tak na konci cesty budeme nejakou vzdialený od začiatku.

Zamyslime sa nad tým, že čo ak by sme spravili nejaký krok zle. Jedna vec je istá a to, že sme sa nepriblížili ku koncu o jedno políčko, čiže sme ho "stratili" a pripočíta sa ku vzdialenosti od začiatku. Druhá vec je, že sa ešte vzdialime od začiatku o jedno políčko iným smerom ako ku koncu a pripočíta sa k vzdialenosti od začiatku.

Optimálne riešenie

Z tohto uvažovania nám vyplýva, že keď spravíme zlý krok, tak sa náš koncový bod vzdiali o 2 políčka od začiatočného bodu. Čiže celkový počet zlých krokov vypočítame ako vzdialenosť od začiatku predelenú 2 ($\frac{(|x|+|y|)}{2}$).

Takéto riešenie bude mať časovú zložitosť $O(n)$ a pamäťovú zložitosť $O(n)$

Listing programu (Python)

```
1 n = int(input())
2 x = y = 0
3
4 for c in input():
5     if c == "H":
6         y += 1
7     elif c == "D":
8         y -= 1
9     elif c == "L":
10        x -= 1
11    elif c == "P":
12        x += 1
13
14 print((abs(x) + abs(y)) // 2)
```

Listing programu (C++)

```

1  #include "iostream"
2  using namespace std;
3
4  int main() {
5      int n, x = 0, y = 0;
6      char character;
7
8      cin >> n;
9
10     for (int i = 0; i < n; i++) {
11         cin >> character;
12         switch (character) {
13             case 'H':
14                 y++;
15                 break;
16             case 'D':
17                 y--;
18                 break;
19             case 'L':
20                 x--;
21                 break;
22             case 'P':
23                 x++;
24                 break;
25             default:
26                 continue;
27         }
28     }
29
30     cout << (abs(x) + abs(y)) / 2 << endl;
31 }

```

Ale pamäťovú zložitosť dokážeme zmenšiť na $O(1)$, lebo si nepotrebuje pamätať celú cestu naraz, stačí nám poznať iba jednotlivé znaky.

Listing programu (Python)

```

1  import sys
2
3  n = int(input())
4  x = y = 0
5
6  for i in range(n):
7      c = sys.stdin.read(1)
8      if c == "H":
9          y += 1
10     elif c == "D":
11         y -= 1
12     elif c == "L":
13         x -= 1
14     elif c == "P":
15         x += 1

```

```
16
17 print((abs(x) + abs(y)) // 2)
```

Listing programu (C++)

```
1  #include "iostream"
2  using namespace std;
3
4  int main() {
5      int n, x = 0, y = 0;
6      char character;
7
8      cin >> n;
9
10     for (int i = 0; i < n; i++) {
11         cin >> character;
12         switch (character) {
13             case 'H':
14                 y++;
15                 break;
16             case 'D':
17                 y--;
18                 break;
19             case 'L':
20                 x--;
21                 break;
22             case 'P':
23                 x++;
24                 break;
25             default:
26                 continue;
27         }
28     }
29
30     cout << (abs(x) + abs(y)) / 2 << endl;
31 }
```

Strižo

2. Ako to len ofarbiť

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

Zaujímavá myšlienka

Dôležité je si hneď na začiatku uvedomiť, že Miškina podmienka – žiadne dve kachličky dotýkajúce sa hranou nemôžu mať rovnakú farbu – v kombinácii s tým, že máme iba dve farby, znamená, že výsledná terasa bude vyzeráť ako šachovnica. Ďalej si môžeme všimnúť, že z tohto dôvodu existujú iba dva finálne stavy po tom, čo by sme všetky potrebné kachličky premaľovali. Buď bude v ľavom hornom rohu ružová farba, alebo fialová. Zvyšok je v podstate daný týmto jedným rozhodnutím.

Optimálne riešenie

Na riešenie problému môžeme pristúpiť tak, že si rozdelíme políčka na šachovnici do dvoch skupín. Nazvime ich párne a nepárne, pričom medzi párne zaradíme ľavé horné políčko spolu so všetkými ďalšími, ktoré by na šachovnici mali rovnakú farbu.

Ak si zvolíme, že v ľavom hornom rohu má byť ružová kachlička, vieme spočítať, na koľkých párných políčkach sú fialové a na koľkých nepárnych sú ružové. Súčet týchto dvoch čísel predstavuje počet kachličiek, ktoré by

sme museli premaľovať. Podobne si môžeme vypočítať, koľko by sme museli premaľovať v prípade, že v ľavom hornom rohu by mala byť fialová kachlička – stačí spočítať počet ružových kachličiek na párnych pozíciách a počet fialových na nepárnych pozíciách.

Nakoniec už len porovnáme, ktoré z týchto dvoch čísel je menšie, a vypíšeme ho. (Poznámka: Nemusíme našou plochou prechádzať dvakrát – stačí raz a údaje si priebežne zapamätáme.)

Časová zložitosť: $O(N \times M)$, keďže každú kachličku prejdeme raz. Pamäťová zložitosť: $O(1)$, keďže si pri postupnom načítavaní vstupu stačí pamätať iba počty jednotlivých typov políčok (napr. „párne fialové = 3“), a teda si nemusíme ukladať celý vstup – postačí nám zopár premenných.

Listing programu (Python)

```
1 n,m = map(int, input().split())
2 parne = True
3 slovník = {'Truer':0, 'Falser':0, 'Truef':0, 'Falsef':0}
4 for _ in range(n):
5     riadok = input()
6     for farba in riadok:
7         slovník[str(parne)+farba] += 1
8         parne = not parne
9     if not m % 2: parne = not parne
10 zacruzova = slovník['Truer'] + slovník['Falsef']
11 zacfialova = slovník['Falser'] + slovník['Truef']
12
13 print(min((n*m - zacruzova, n*m - zacfialova)))
```

Listing programu (C++)

```
1 #include <iostream>
2 #include <unordered_map>
3 #include <algorithm>
4 using namespace std;
5
6 int main() {
7     int n, m;
8     cin >> n >> m;
9
10    bool parne = true;
11    unordered_map<string, int> slovník = { {"Truer", 0}, {"Falser", 0}, {"Truef", 0}, {"Falsef", 0}
↪ };
12
13    for (int i = 0; i < n; ++i) {
14        string riadok;
15        cin >> riadok;
16
17        for (char farba : riadok) {
18            string key = (parne ? "True" : "False") + string(1, farba);
19            slovník[key]++;
20            parne = !parne;
21        }
22
23        if (m % 2 == 0) {
24            parne = !parne;
25        }
26    }
```

```

26     }
27
28     int zacruzova = slovník["Truer"] + slovník["Falsef"];
29     int zacfialova = slovník["Falser"] + slovník["Truef"];
30
31     cout << min((n * m - zacruzova), (n * m - zacfialova)) << endl;
32
33     return 0;
34 }

```

Stanko

3. Častá naháňka

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

Pozrime sa na to, ako sa dá popísať stav hry. To vieme pomocou troch hodnôt - naša pozícia, v ktorej línii sa nachádzame a to, že ako ďaleko je mačka. Označme si tento stav ako trojicu (p, l, m) . Povedzme teraz, že máme takýto stav a pozrime sa na to, do akých stavov sa môžeme dostať na jeden krok. Máme teda 3 možnosti: - Ak sa posunieme dopredu, tak naša pozícia sa zväčší o 1 a mačka sa tiež posunie o 1. Nový stav bude $(p+1, l, m+1)$. - Ak sa posunieme dozadu, tak naša pozícia sa zmenší o 1 a mačka sa tiež posunie o 1. Nový stav bude $(p-1, l, m+1)$. - Ak skočíme, tak sa pohneme o k krokov dopredu a zmeníme línii. Nový stav bude $(p+k, l, m+1)$.

Na začiatku sme v stave $(1, 0, 0)$. Víťazný stav je hocičo, ktoré má p väčšie ako n .

Bruteforce

Ako prvé čo nás napadne je skúšať všetky možné ťahy. Teda začneme počiatočným stavom a rekurzívne sa zavoláme pre všetky stavy do ktorých sme sa vedeli dostať na jeden krok a zároveň musí platiť, že mačka nás nezožerie ($m < p$) a taktiež nemôžeme skončiť na prekážke, teda políčko na ktorom skončíme musí byť prázdne. Tým, že mačka sa dostane na koniec levelu, tak toto riešenie musí skončiť.

Po každom našom pohybe sa mačka pohne o políčko hore, takže ak neutečieme o n krokov, neutečieme vôbec. Takže skúsime najviac každú postupnosť n možných ťahov, a v každom momente máme navyber najviac 3 možné ťahy. Teda takéto riešenie vyskúša nanajvyš 3^n postupností ťahov.

Optimálne riešenie

Všimnime si, že predošlé riešenie skúša nejaké stavy zbytočne viackrát. Napríklad, ak $k=1$, tak do stavu $(3, 0, 2)$ sa vieme dostať dvoma spôsobmi - dvakrát ideme dopredu, alebo dvakrát skáčeme. Potom z tohto stavu by sme ďalej skúšali všetky stavy odtiaľto zbytočne 2-krát. Čiže si potrebujeme sledovať, že v akých stavoch sme už boli, aby sme ich neriešili viackrát. To vieme docieľiť napríklad pomocou slovníka v pythone.

Pozrime sa ďalej na príklad, že sa nejakú pozíciu vieme dostať dvoma rôznymi spôsobmi, pričom v jednom je mačka ďalej od nás. Zjavne ak vieme uniknúť v neskoršom čase, tak to zvládneme aj keby sme unikli skôr. Naopak ak si vyberieme tú dlhšiu cestu, tak sa to neskôr môže pokaziť a mačka nás chytí. Teda vieme povedať, že ak to pôjde, tak to nutne pôjde tou najkratšou cestou. Na vyriešenie tohto problému vieme použiť klasické prehľadávanie do šírky.

Namiesto rekurzie máme frontu v ktorej si uchováваме stavy do ktorých sme sa nejak dostali. Na začiatku do nej pridáme počiatočný stav a potom opakujeme kým sa fronta nevyprázdni, alebo sme sa dostali do cieľa. Pri každej iterácii si z fronty vyberieme prvý stav, označíme si, že sme v tomto stave už boli a potom z neho skúsime spraviť všetky 3 ťahy a stavy do ktorých sme sa dostali pridáme do fronty. Pridávame samozrejme iba povolené ťahy, také pri ktorých neskončíme na prekážke a ani nás mačka nezožerie.

Pozrime sa na časovú a pamäťovú zložitosť tohto riešenia. Tým, že na každé políčko sa pozeráme iba raz, tak časová zložitosť bude lineárna od počtu políčok, teda celkovo to bude $O(n)$. Potrebujeme si pamätať, na ktorom políčku sme už boli, aby sme žiadne políčko nenavštívili viackrát. Teda pamäťová zložitosť bude taktiež $O(n)$.

Listing programu (Python)

```

1 n, k = map(int, input().split())
2 first = input()
3 second = input()

```

```

4 first_visited = [False for x in range(n)]
5 second_visited = [False for s in range(n)]
6
7 q = []
8 q.append((0,0,0))
9 ans = False
10
11 while len(q) > 0:
12     curr = q.pop(0)
13     if curr[0] >= n:
14         ans = True
15         break
16
17     if curr[2] >= curr[0]:
18         continue
19     if curr[1] == 1 and second_visited[curr[0]]:
20         continue
21     if curr[1] == 0 and first_visited[curr[0]]:
22         continue
23
24     if curr[1] == 0:
25         first_visited[curr[0]] = True
26         if curr[0] - 1 > 0 and first[curr[0] - 1] != 'X':
27             if curr[0] - 1 > curr[2]:
28                 q.append((curr[0] - 1, 0, curr[2] + 1))
29         if curr[0] + 1 >= n or first[curr[0] + 1] != 'X':
30             q.append((curr[0] + 1, 0, curr[2] + 1))
31         if curr[0] + k >= n or second[curr[0] + k] != 'X':
32             q.append((curr[0] + k, 1, curr[2] + 1))
33     elif curr[1] == 1:
34         second_visited[curr[0]] = True
35         if curr[0] - 1 > 0 and second[curr[0] - 1] != 'X':
36             if curr[0] - 1 > curr[2]:
37                 q.append((curr[0] - 1, 1, curr[2] + 1))
38         if curr[0] + 1 >= n or second[curr[0] + 1] != 'X':
39             q.append((curr[0] + 1, 1, curr[2] + 1))
40         if curr[0] + k >= n or first[curr[0] + k] != 'X':
41             q.append((curr[0] + k, 0, curr[2] + 1))
42
43 if ans:
44     print('Vyhrál som')
45 else:
46     print('Prehrál som')

```

Listing programu (C++)

```

1 #include <bits/stdc++.h>
2 #include <queue>
3 #include <vector>
4 using namespace std;
5 using ll = long long;
6

```

```

7  struct Stav {
8      int macka;
9      int linia;
10     int pozicia;
11 };
12
13 int main() {
14     cin.tie(0)->sync_with_stdio(0);
15
16     int n, k;
17     cin >> n >> k;
18     vector<char> prva_linia(n), druha_linia(n);
19     vector<char> first_visited(n, 0), second_visited(n, 0);
20     for (int i = 0; i < n; i++) {
21         cin >> prva_linia[i];
22     }
23     for (int i = 0; i < n; i++) {
24         cin >> druha_linia[i];
25     }
26
27     queue<Stav> q;
28     q.push({0, 0, 0});
29     bool odpoved = false;
30
31     while (!q.empty()) {
32         Stav terajsi_stav = q.front();
33         q.pop();
34
35         if (terajsi_stav.pozicia >= n) {
36             odpoved = true;
37             break;
38         }
39
40         if (terajsi_stav.macka > terajsi_stav.pozicia) {
41             continue;
42         }
43         if (terajsi_stav.linia == 1 && second_visited[terajsi_stav.pozicia]) {
44             continue;
45         }
46         if (terajsi_stav.linia == 0 && first_visited[terajsi_stav.pozicia]) {
47             continue;
48         }
49
50         Stav nasledujuci;
51         nasledujuci.macka = terajsi_stav.macka + 1;
52         nasledujuci.linia = terajsi_stav.linia;
53
54         if (terajsi_stav.linia == 0) {
55             first_visited[terajsi_stav.pozicia] = 1;
56             if (terajsi_stav.pozicia - 1 > 0 && prva_linia[terajsi_stav.pozicia - 1] != 'X') {
57                 nasledujuci.pozicia = terajsi_stav.pozicia - 1;
58                 if (nasledujuci.pozicia >= nasledujuci.macka) {
59                     q.push(nasledujuci);

```

```

60     }
61 }
62
63 if (prva_linia[terajsi_stav.pozicia + 1] != 'X') {
64     nasledujuci.pozicia = terajsi_stav.pozicia + 1;
65     q.push(nasledujuci);
66 }
67
68 if (terajsi_stav.pozicia + k >= n  druha_linia[terajsi_stav.pozicia + k] != 'X') {
69     nasledujuci.pozicia = terajsi_stav.pozicia + k;
70     nasledujuci.linia = 1;
71     q.push(nasledujuci);
72 }
73 } else if (terajsi_stav.linia == 1) {
74     second_visited[terajsi_stav.pozicia] = 1;
75     if (terajsi_stav.pozicia - 1 > 0 && druha_linia[terajsi_stav.pozicia - 1] != 'X') {
76         nasledujuci.pozicia = terajsi_stav.pozicia - 1;
77         if (nasledujuci.pozicia > nasledujuci.macka){
78             q.push(nasledujuci);
79         }
80     }
81
82     if (terajsi_stav.pozicia + 1 >= n  druha_linia[terajsi_stav.pozicia + 1] != 'X') {
83         nasledujuci.pozicia = terajsi_stav.pozicia + 1;
84         q.push(nasledujuci);
85     }
86
87     if (terajsi_stav.pozicia + k >= n  prva_linia[terajsi_stav.pozicia + k] != 'X') {
88         nasledujuci.pozicia = terajsi_stav.pozicia + k;
89         nasledujuci.linia = 0;
90         q.push(nasledujuci);
91     }
92 }
93 }
94
95 if (odpoved) {
96     cout << "Vyhral som" << '\n';
97 } else {
98     cout << "Prehral som" << '\n';
99 }
100 }

```

Janči

4. Iné jazýčky

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

Naším cieľom je spočítať súčet dĺžok všetkých preložiteľných intervalov. Najjednoduchší spôsob, ako to spraviť, je postupne všetky takéto intervaly nájsť.

Pomalé prehľadávanie podreťazcov

Všetky preložiteľné intervaly môžeme nájsť napríklad tak, že sa pozrieme na všetky možné intervaly, a pre každý zistíme, či sa dá preložiť alebo nie.

Každý interval je jednoznačne definovaný svojim začiatkom a koncom, takže na prejdienie všetkých nám stačia dva for cykly – jeden pre začiatok a druhý pre koniec (pričom si dáme pozor, aby koniec nikdy nebol pred začiatkom). Každý interval potom prejdeme tretím cyklom, v ktorom otestujeme, či sa dá preložiť.

To vieme spraviť veľmi jednoducho. Vyrobíme si dve polia dĺžky anglickej abecedy (malé a veľké písmená) a pre každý znak, ktorý by sa mohol vyskytnúť vo vrchnom alebo spodnom reťazci si budeme pamätať, na aký znak v opačnom reťazci sa prekladá. Na začiatku testovania intervalu budú všetky hodnoty samozrejme prázdne (špeciálne označené, že zatiaľ žiadny preklad nemáme).

Keď potom interval prechádzame, dostávame postupne dvojice znakov, ktoré sa pokúšame do poľa zapísať. Ak je pre nejaké písmeno v poli už zapísaný jeho preklad, a tento preklad nie je rovnaký, ako by sme chceli doplniť, znamená to, že aktuálny interval je nepreložiteľný a môžeme jeho testovanie ukončiť. Ak sa nám podarí úspešne dokončiť testovanie intervalu, pridáme jeho dĺžku (rozdiel konca a začiatku plus jedna) k odpovedi.

Tento prístup má časovú zložitosť $O(n^3)$ – prechádzame každý z kvadraticky mnoho intervalov dĺžky až n . Pamäťová zložitosť je lineárna, keďže si musíme pamätať celé pôvodné pole, aby sme ho mohli prechádzať opakovane.

Zafixovanie začiatku

Zrejme je ale jasné, že pri predošlom prístupe počítame mnohokrát to isté. Napríklad pokiaľ by bol interval $[1, 10]$ preložiteľný, samostatne by sme počítali všetkých 10 intervalov dĺžky jedna ($[1, 1], [2, 2], \dots, [10, 10]$), potom samostatne všetkých 9 intervalov dĺžky 2 ($[1, 2], [2, 3], \dots, [9, 1]$), až po jeden interval dĺžky 10 ($[1, 10]$).

Ak by sme vedeli využiť, že interval $[1, 10]$ je preložiteľný a zároveň maximálny (teda jeho predĺženie doprava už preložiteľné nie je), mohli by sme si ušetriť veľa práce tým, že ho nájdeme iba raz a dĺžky všetkých intervalov v ňom spočítame naraz. Tu by však mohol nastať problém – pokiaľ by bol preložiteľný aj interval $[2, 11]$ (avšak $[1, 11]$ nie), mohlo by sa stať, že započítame množstvo kratších intervalov viackrát.

Skúsme si teda zakaždým zafixovať začiatok intervalu, nájsť koniec maximálneho intervalu, ktorý sa ešte dá preložiť, a pripočítať k výsledku len dĺžky takých jeho podintervalov, ktoré začínajú na našom zafixovanom začiatku. Napríklad pre začiatok na pozícii 1 a maximálny preložiteľný interval $[1, 10]$ zarátame intervaly $[1, 1], [1, 2], \dots, [1, 10]$ s dĺžkami $1 + 2 + \dots + 10 = 55$, teda dokopy $(d + 1) \times (d + 2)/2$, kde d je rozdiel začiatku a konca.

Takto určite zarátame každý preložiteľný interval práve raz, takže dostaneme správnu odpoveď. Časová zložitosť je $O(n^2)$, pre každý začiatok intervalu musíme nájsť jeho koniec, ktorý môže byť až n znakov vzdialený.

Optimálne okienkovanie

Stále počítame niektoré veci zbytočne. Napríklad pokiaľ by sme si pri počítaní maximálneho intervalu pre začiatok na pozícii 1 označili dvojicu $A - a$, rovnakú dvojicu by sme našli na pozícii 5 a na pozícii 11 by sme oproti písmenu A našli b , vieme, že pre začiatky na pozíciách 2, 3, 4 a 5 by sme tiež našli rovnaké konce intervalov na pozícii 11. Tieto intervaly by sme teda vôbec nemuseli počítat.

Rozviňme túto myšlienku do vzorového riešenia: budeme si udržiavať dva indexy – začiatok a koniec intervalu – a okrem prekladov si pre každé písmeno budeme pamätať aj počet výskytov daného prekladu, ktoré v aktuálnom intervale máme.

Vždy budeme najskôr dopredu posúvať koniec intervalu, dokým bude celý interval preložiteľný. Keď už to nepôjde, pretože nájdeme pre niektoré písmeno iný preklad (čo sa môže stať aj pre obe písmená na danej pozícii naraz), než máme aktuálne uložený, budeme interval skracovať od začiatku, až dokým sa nezbavíme všetkých prekladov pre dané písmeno. Práve preto, aby sme vedeli, dokedy máme interval skracovať, si potrebujeme pre každý preklad udržiavať aj počet jeho výskytov.

Pripočítavať výsledky budeme vždy pri predlžovaní intervalu – pri každom predĺžení započítame všetky intervaly, ktoré končia na aktuálne pridanom mieste. Ich počet budeme počítat rovnako, ako pri počítaní intervalov s fixným začiatkom. Určite započítame všetky intervaly, pretože oproti predošlému prístupu sme zmenili iba smer, ktorým ich započítavame (nič nemení, či ich počítame podľa koncov alebo podľa začiatkov).

Zmení sa tiež časová zložitosť – keďže začiatok aj koniec posunieme dokopy najviac n -krát a pre každý posun spravíme konštantné množstvo operácií, bude aj celková časová zložitosť lineárna. Pamäťová zložitosť zostane lineárna – stále si musíme pamätať, v najhoršom prípade, celé pôvodné pole. Veľkosť anglickej abecedy je konštantná, a teda aj pamäť potrebná na uloženie aktuálnych prekladov.

Listing programu (Python)

```
1 #!/usr/bin/env python 3
2
3 def main():
4     N = int(input())
5     L = input()
```

```

6     M = input()
7
8     translation_L = dict() # slovník: znak --> (preložený znak, počet výskytů)
9     translation_M = dict()
10    begin, end = 0, 0
11    ans = 0 # výsledek
12
13    while end < N:
14        end += 1 # skusíme posunout koniec
15        l = L[end - 1]
16        m = M[end - 1]
17
18        while ((l in translation_L.keys() and translation_L[l][0] != m) or
19              (m in translation_M.keys() and translation_M[m][0] != l)):
20            # ak musíme najskôr skrátovať zo začiatku
21            begin += 1
22            remove_L = L[begin - 1]
23            translation_L[remove_L] = (translation_L[remove_L][0],
24                                      translation_L[remove_L][1] - 1)
25            if translation_L[remove_L][1] == 0:
26                del translation_L[remove_L] # odoberáme výskyt zo slovníka
27
28            remove_M = M[begin - 1]
29            translation_M[remove_M] = (translation_M[remove_M][0],
30                                      translation_M[remove_M][1] - 1)
31            if translation_M[remove_M][1] == 0:
32                del translation_M[remove_M] # odoberáme výskyt zo slovníka
33
34            # už môžeme posunúť koniec
35
36            # pridáme nový preklad alebo zvýšime počet jeho výskytov
37            if not l in translation_L.keys(): translation_L[l] = (m, 0)
38            translation_L[l] = (translation_L[l][0],
39                              translation_L[l][1] + 1)
40
41            # pridáme nový preklad alebo zvýšime počet jeho výskytov
42            if not m in translation_M.keys(): translation_M[m] = (l, 0)
43            translation_M[m] = (translation_M[m][0],
44                              translation_M[m][1] + 1)
45
46            # započítame intervaly končiace na novej end pozícii
47            ans += (end - begin) * (end - begin + 1) // 2
48
49    print(ans)
50
51    main()

```

Listing programu (C++)

```

1 #include <bits/stdc++.h>
2 using namespace std;
3

```

```

4  int main() {
5      int N; cin >> N;
6      string L, M; cin >> L >> M;
7
8      unsigned long long ans = 0; // vysledok
9      vector<pair<char, int>> translations_L(128), translations_M(128); // pismeno, pocet vyskytov
10     // 128 je malo, nemusime teda komplikovane pocitat indexy pismen v abecede
11     unsigned long long begin = 0, end = 0;
12
13     while (end < N){
14         ++end; // skusime posunut koniec
15         char l = L[end - 1];
16         char m = M[end - 1];
17
18         while ((translations_L[l].second != 0 && translations_L[l].first != m)
19                (translations_M[m].second != 0 && translations_M[m].first != l)){
20             // ak musime najskor skracovat zo zaciatku
21             begin += 1;
22
23             char remove_L = L[begin - 1];
24             --translations_L[remove_L].second; // odoberame vyskyt zo slovnika
25
26             char remove_M = M[begin - 1];
27             --translations_M[remove_M].second; // odoberame vyskyt zo slovnika
28         }
29
30         // uz mozeme posunut koniec
31
32         translations_L[l].first = m;
33         ++translations_L[l].second; // pridame novy vyskyt
34
35         translations_M[m].first = l;
36         ++translations_M[m].second; // pridame novy vyskyt
37
38         // zapocitame intervaly konciace na novej end pozicii
39         ans += ((end - begin) * (end - begin + 1) / 2);
40     }
41
42     cout << ans << endl;
43 }

```

Fipo

5. A koľko mu mám dať?

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

Bruteforce

Najjednoduchšie riešenie je, že spravíme OR každého súvislého intervalu. Toto riešenie má časovú zložitosť $O(n^3)$, pretože máme n^2 intervalov a na väčšine z nich musíme spraviť OR rádomo n čísiel. Toto riešenie má pamäťovú zložitosť $O(n)$, pretože si potrebujeme zapamätať celý vstup dĺžky n a zopár pomocných premenných.

O niečo lepšie riešenie

Povedzme, že máme vypočítaný OR intervalu $P[i, j]$, ktorý sa rovná $o_{i,j}$. Potom OR intervalu $P[i, j + 1]$ vypočítame: $o_{i,j+1} = o_{i,j}$ OR $P[j + 1]$. Časová zložitosť tohto riešenia je $O(n^2)$, keďže musíme prejsť každý

interval, ktorých je rádovo n^2 . Toto riešenie má pamäťovú zložitosť $O(n)$, pretože si potrebujeme zapamätať celý vstup dĺžky n a zopár pomocných premenných.

Zaujímavé myšlienky

Povedzme, že máme vypočítaný OR intervalu $P[i, j]$, ktorý je $o_{i,j}$. Vieme z tejto informácie zistiť $o_{i+1,j}$?

Áno, vieme! Len si potrebujeme pamätať zopár extra vecí intervalu $P[i, j]$. Potrebujeme si pamätať, koľko krát boli jednotlivé bity “zapnuté” v intervale $P[i, j]$. Pamätajme si túto informáciu v poli *BITS*. Keď potom chceme vedieť $o_{i+1,j}$, tak od každej hodnoty v *BITS*[i] odpočítame 1, ak ten bit bol zapnutý v čísle $P[i]$. $o_{i+1,j}$ zistíme tak, že za každú hodnotu *BITS*, ktorá je väčšia ako 1 pripočítame príslušnú mocninu 2. Pole *BITS* má veľkosť rádovo $\log(\max(P[i]))$, ale keďže $P[i] \leq 10^9$, čiže $\log(\max(P[i])) \leq 31$ tak môžeme povedať, že pole *BITS* má konštantnú veľkosť.

Podme sa teraz pozrieť na jednotlivé čísla v poli. Konkrétne o každom čísle chceme vedieť, či môže byť v niektorom intervale, ktorý má OR presne k . Keď má $P[i]$ zapnutý bit b , ktorý je vypnutý v k , tak nemôže byť v žiadnom intervale, ktorý má OR presne k . $P[i]$ nám teda rozdelí pole P na 2 časti, pričom žiadny interval, ktorý začína v prvej časti a končí v druhej časti nemá bitový OR k .

Niečo málo o 2 bežcoch

Ak už viete, ako funguje metóda 2 bežcov, môžete túto časť preskočiť

Optimálne riešenie využíva metódu dvoch bežcov. Tá spočíva v tom, že si spravíme 2 premenné, ktoré budú ukazovať na rôzne prvky v poli. Títo dvaja bežci môžu bežať oproti sebe (na začiatku ich nastavíme, aby jeden ukazoval na začiatok a druhý na koniec poľa) alebo rovnakým smerom (na začiatku obaja ukazujú na začiatok poľa) - toto je prípad tejto úlohy.

Dobrá rule of thumb je, že keď potrebujeme niečo vedieť o podúseku poľa, tak hýbeme bežcami rovnakým smerom. Keď niečo potrebujeme vedieť o dvojici prvkov poľa, tak hýbeme bežcami oproti sebe.

Podľa toho, aký je momentálny podúsek hýbeme 2 bežcami. Keď je podúsek príliš “malý”, tak pohneme bežcom, ktorý ukazuje na koniec podúseku. Naopak, keď je podúsek príliš “veľký”, tak pohneme bežcom, ktorý ukazuje na začiatok podúseku.

Optimálne riešenie

Teraz to už iba celé musíme dať dokopy. Najskôr si pole P rozdelíme na časti podľa čísiel, ktoré majú zapnutý niektorý bit b , ktorý je vypnutý v k . Keď už ho takto máme rozdelené, tak ideme pre každú časť zistiť, koľko je v nej intervalov, ktoré majú OR k .

To spravíme pomocou 2 bežcov. 1. bežec bude ukazovať na začiatok intervalov, ktoré majú OR k , 2. bežec bude ukazovať na koniec najkratšieho intervalu, ktorý začína na mieste 1. bežca a má OR k . Všetky dlhšie intervaly až po koniec daného úseku majú OR k a teda ich započítame - je ich toľko, koľko je rozdiel 2. bežca a konca intervalu. Keď už sme našli úsek, ktorý má OR k , tak posunieme 1. bežca. Pri posúvaní využijem výpočet OR-u popísaný vyššie. 2. bežca posúvame, až dokým nenájdeme začiatok najkratšieho úseku, ktorý má OR k (niekedy ho vôbec nemusíme posunúť). Intervaly budeme zarátavať vtedy, keď pôjdeme pohnúť 1. bežcom. Keďže interval je jednoznačne určený svojím začiatkom a 1. bežcom prejdeme cez všetky možné začiatky intervalov, tak každý interval, ktorý vyhovuje zarátame raz.

Toto riešenie má časovú zložitosť $O(n \log(\max(P[i])))$, keďže musíme prejsť celým polom a na každý prvok môžu bežcovia byť najviac 2-krát. $\log(\max(P[i]))$ je tam kvôli spracovávaniu OR čísiel poľa. Keďže $\log(\max(P[i])) \leq 31$, tak môžeme povedať, že časová zložitosť je $O(n)$. Pamäťovú zložitosť má toto riešenie $O(n)$, pretože si potrebujeme zapamätať celý vstup dĺžky n .

Sebik

6. TokTik mačiatka

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

Najprv rýchle odrozprávkovanie zadania. Máme strom s ohodnotenými vrcholmi. Chceme vybrať nejakú množinu vrcholov s čo najväčším súčtom hodnôt tak, aby žiadne dva vrcholy v našej vybranej množine boli spojené hranou. Bude takúto nejakú množinu vrcholov volať **pokrytie**, a súčet hodnôt vrcholov v tomto pokrytí budem volať **hodnota pokrytia**. Ak pokrytie zároveň spĺňa podmienku zo zadania, teda že žiadne dva vrcholy v pokrytí nesmú byť spojené hranou, budem ho volať **platné pokrytie**.

Bruteforce

Najprv rýchlo spomenieme pomerne neefektívny algoritmus hrubej sily, ktorý rieši prvú sadu - vyskúšame ním všetky možnosti. Počet vrcholov je iba malý, takže môžeme jednoducho vyskúšať všetky možné pokrytia,

pre každé skontrolovať, či je platné, a vybrať to s najväčšou hodnotou. Počet takýchto pokrytí je však pomerne veľký, pretože každý vrchol môže v každom pokrytí byť, alebo byť. Počet pokrytí je teda 2^n , a pre každé pokrytie treba ešte skontrolovať, či je platné. To vieme spraviť jedným prechodom stromu, čiže nejaké DFS. Celková časová zložitosť teda bude $O(n \cdot 2^n)$, čo stačí na prvú sadu.

Greedy riešenie tretej sady

Pred riešením na plný počet bodov sa ešte podme pozrieť na riešenie tretej sady, kde mali všetky vrcholy rovnakú hodnotu. Táto sada sa dala vyriešiť greedy (pažravým) riešením. Vieme si totiž uvedomiť, že v optimálnom pokrytí určite neškodí, ak doňho zahrnieme nejaký list stromu. Zakoreňme si strom a povedzme, že vrchol A je list a jeho rodič je vrchol B . V optimálnom pokrytí určite chceme mať buď vrchol A , alebo vrchol B , a to práve jeden z nich. Ak do pokrytia zahrnieme vrchol A , vieme, že nič okrem vrcholu B tým neobmedzíme. Naopak, ak vezmeme vrchol B , môže sa stať, že si uškodíme na iných miestach. Teda vždy sa nám oplatí vziať list A , a poznamenať si, že B zobrať nemôžeme. Inými slovami, ak zoberieme rodiča listu, tak skóre si vylepšíme rovnako, ale vrcholy, ktoré sú v pokrytí, alebo ich nemôžeme zobrať, budú stále nadmnožina vrcholov v pokrytí, alebo vrcholov ktoré nemôžeme zobrať ak zoberieme list. Teda nedosiahneme lepšie riešenie ako keď zoberieme list. No ale ak už sme teda nejaký list do pokrytia zahrnuli, tak nás už až tak nazaujíma, tak ho môžeme zo stromu odpojiť. Toto nám zachová strom, teda určite bude existovať nejaký iný list na spracovanie.

Algoritmus teda postupuje nasledovne: budeme si udržiavať stack listov na spracovanie. Vždy so spracovaným listom spravíme nasledovné: ak sme si niekedy poznamenali, že ho vziať nemôžeme, tak nič nerobíme, ak nie, zahrnieme ho do pokrytia a pre jeho rodiča si poznamenáme, že ho nemôžeme zobrať. Potom tento list odstránime, updatneme rodiča, a ak sa rodič práve stal listom, zaradíme rodiča do stacku listov na spracovanie. Tento algoritmus beží v $O(n)$ a k riešeniu na plný počet bodov nám nepomôže, avšak je fajn vedieť, že sa podúloha dá riešiť aj takto.

Čo s cestou?

Presuňme sa zaujímavejšej sade - sade 2. V tejto sade máme garantované, že graf je cesta, čo však znamená, nemusíme rozmýšľať nad nijakým grafom. Jednoducho si úlohu pretransformujeme na nasledovnú: máme pole čísel, a chceme z nich vybrať množinu čísel čo najväčšiu v súčte, avšak žiadne dva vybrané prvky nemôžu byť susedné. Vela z vám táto úloha príde pomerne povedomá - nachádza sa totiž na testovači, prípadne sme ju vedeli nájsť aj v tohtoročných zadaniach Kasiopei. Táto úloha sa dá riešiť dynamickým programovaním.

Myšlienka dynamického programovania

Budeme postupne počítat najlepšie pokrytie pre časť poľa. Pre prvok i budeme chcieť vypočítat hodnoty B (beriem) a N (neberiem). Hodnota $B[i]$ nám hovorí, aký najväčší súčet vieme dostať pre nejaké pokrytie prvých i prvkov, pričom prvok i sa v pokrytí určite nachádza. Naopak, hodnota N nám hovorí, aký najväčší súčet môžeme dostať pre prvých i prvkov, ak sa v prvok i v pokrytí určite nenachádza. Ako vieme tieto hodnoty vypočítat?

Predpokladajme, že poznáme hodnoty $B[i - 1]$ a $N[i - 1]$. Vieme, že pri pokrytí, kde určite berieme i -tý prvok, musí byť $i - 1$ prvok nezobraný, ale zároveň chceme prvých $i - 1$ prvkov vybrať optimálne. To však vieme, poznáme predsa hodnotu $N[i - 1]$. Teda vieme povedať, že $B[i] = N[i - 1] + \text{hodnota}[i]$. Podobne vieme vypočítat $N[i]$. Vieme totiž, že ak určite neberieme i -tý prvok, môžeme si prvých $i - 1$ prvkov navoliť ľubovoľne, tak aby boli optimálne. Teda $N[i] = \max(B[i - 1], N[i - 1])$.

Teda sme si ukázali, že vieme hodnoty $B[i]$ a $N[i]$ vypočítat iba s vedomosťou hodnôt $B[i - 1]$ a $N[i - 1]$. Vieme potom povedať, že $B[0] = \text{hodnota}(0)$ a $N[0] = 0$. No ale z týchto dvoch hodnôt vieme vypočítat hodnoty B a N pre všetky prvky v poli, pretože z hodnôt pre prvok 0 si vieme vypočítat hodnoty pre prvok 1, z tých pre prvok 2... a tak ďalej. Pre každé políčko tiež vypočítame iba dve rovnice, čiže časová zložitosť programu bude $O(n)$.

Optimálne riešenie - dynamika na stromoch

Čo však s riešením na plný počet bodov? Zakoreňme si strom, a z predošlého odseku si odnesme jednu kľúčovú myšlienku - aj v strome vieme totiž efektívne rátať hodnoty B a N , ktoré však budú mať v strome trochu lepšiu definíciu. $B[X]$ bude najlepšie pokrytie podstromu vrcholu X také, že X berieme (teda sa nachádza v pokrytí). $N[X]$ bude podobne najlepšie pokrytie podstromu vrcholu X také, že X neberieme (teda sa v pokrytí nenachádza). Podme si teda tiež upraviť vzorček pre dynamické programovanie.

Povedzme, že chceme hodnoty B a N vypočítat pre nejaký vrchol A , a predpokladajme, že poznáme hodnoty B a N pre všetkých jeho synov. Potom ak chceme vypočítat hodnotu $B[A]$, tak vieme, že pre tento prípad musia

byť všetci synovia tohoto vrcholu nezobraní, pretože vrchol A zobrazený je. Preto viem jednoducho sčítať hodnoty N pre synov vrcholu A , pripočítať k tomu hodnotu A , a tento výsledok bude žiadané $B[A]$.

$N[A]$ vieme tiež vyriešiť podobne ako predtým. Opäť platí, že tým, že vrchol A neberieme, vieme si pre každého jeho syna vybrať to najoptimálnejšie pokrytie, bez žiadnych iných obmedzení. Preto pre každého syna S vrcholu A vypočítame hodnotu $\max(B[S], N[S])$, a tieto hodnoty sčítame. To bude hodnota $N[A]$. Nakoniec si vieme rozmyslieť, že ak vrchol nemá synov, vieme pre neho vypočítať hodnoty B a N veľmi jednoducho, a teda pre list i bude $B[i] = \text{hodnota}(i)$ a $N[i] = 0$. A ak nejaký vrchol synov má, jeho optimálne hodnoty B a N vieme vypočítať vyššie popísaným spôsobom.

Ako to nakódiť? Ukáže sa, že pomerne jednoducho. Stačí spustiť funkciu DFS , ktorá pre zavolanie do nejakého vrcholu vráti dve hodnoty - B a N tohoto vrcholu. Tieto hodnoty spočítame pekne rekurzívne, až kým v koreni nedostaneme výsledok.

A nakoniec: akú časovú zložitosť bude mať toto riešenie? DFS funkcia sa zavolá do každého vrcholu raz, pričom v každom vrchole bude rekurzívne dostávať hodnoty B a N pre synov tohoto vrcholu. Tieto hodnoty si iba upravíme a sčítame ako popísané vyššie. Naš algoritmus bude mať teda časovú zložitosť $O(n)$.

Listing programu (Python)

```
1 import sys
2 sys.setrecursionlimit(10**6)
3
4 pct = int(input())
5 uzitocnost = list(map(int, input().split()))
6 graf = [[] for _ in range(pct)]
7
8 def dfs(vtx, parent):
9     on = uzitocnost[vtx]
10    off = 0
11    for to in graf[vtx]:
12        if to != parent:
13            x = dfs(to, vtx)
14            on += x[0]
15            off += max(x[0], x[1])
16    return [off, on]
17
18 for i in range(pct-1):
19    frm, to = list(map(int, input().split()))
20    graf[frm].append(to)
21    graf[to].append(frm)
22
23 f = dfs(0,0)
24 print(max(f[0], f[1]))
```

Listing programu (C++)

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 using ll = long long;
4
5 vector<ll> uzitocnost;
6 vector<vector<int>> graf;
7
8 pair <ll, ll> dfsko(int vtx, int parent) {
9     ll zapnuty = uzitocnost[vtx];
```

```

10     ll vypnuty = 0;
11
12     for (int i = 0; i < graf[vtx].size(); i++)
13     {
14         int to = graf[vtx][i];
15         if (to != parent)
16         {
17             pair <ll,ll> vals = dfsko(to, vtx);
18             zapnuty += vals.first;
19             vypnuty += max(vals.first, vals.second);
20         }
21
22     }
23
24     return {vypnuty, zapnuty};
25 }
26
27 int main() {
28     cin.tie(0)->sync_with_stdio(0);
29
30     int N;
31     cin >> N;
32
33     uzitocnost.resize(N);
34     graf.resize(N);
35
36     for (int i = 0; i < N; i++)
37     {
38         ll x; cin >> x;
39         uzitocnost[i] = x;
40     }
41
42
43     for (int i = 0; i < N-1; i++)
44     {
45         int from, to;
46         cin >> from >> to;
47         graf[to].push_back(from);
48         graf[from].push_back(to);
49     }
50
51     pair <ll,ll> vysledok = dfsko(0,0);
52     cout << max(vysledok.first, vysledok.second) << endl;
53 }

```

Paulinka

7. Kocúre a Klbko

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

Vítazné pozície

V tejto úlohe sme sa pohrali s teóriou hier, kde sme chceli spočítať koľko zo začiatočných pozícií je **vítazných** pre Aničku – teda ak začnú hrať hru s dĺžkami špagátov a a b , či vie Anička vyhrať ak vie optimálne.

Keďže hra je *symetrická* (možné ťahy sú rovnaké bez ohľadu na to, kto je na ťahu), vieme všeobecne hovoriť o každej dvojici dĺžok špagátov či je víťazná pre mačiatko na ťahu. *Stav hry* (alebo pozíciu) si označíme ako

(a, b) – dĺžky špagátov.

Ako zistíme ktoré pozície sú víťazné? Zjavne, ak sa mačiatko dostalo na ťah a ostal mu jeden špagát nenulovaj dĺžky, a druhý špagát došiel, znamená to že mačiatko na prechádzajúcom ťahu prehralo. Preto sú pozície $(a, 0)$ a $(0, b)$ *vyhrávajúce* pre všetky celé čísla $a, b > 0$.

Podme sa pozrieť na pozíciu (a, b) , pričom $a \leq b > 0$ (prípád, že $b \leq a > 0$ je symetrický). Mačiatko na ťahu vie skrátiť prvý špagát na dĺžky $a - b, a - 2b, \dots, a - kb \geq 0$ (pre nejaké celé číslo k).

Predstavme si, že vieme pre každú z pozícií $(a - b, b), \dots, (a \bmod b, b)$ či je vyhrávajúca alebo nie. Ak medzi nimi je aspoň jedna prehrávajúca, mačiatko na ťahu spraví tento ťah a zrazu sa dostane na ťah druhé mačiatko, ale je v prehrávajúcej pozícii. A keďže prehrávajúca pozícia nie je vyhrávajúca, znamená to, že nech robí, čo robí, ak jeho oponent hrá optimálne, prehrá. V tomto prípade je tak pozícia (a, b) vyhrávajúca.

Na druhú stranu, ak sú pozície $(a - b, b), \dots, (a - kb, b)$ všetky vyhrávajúce, mačiatko na ťahu nemá na výber, nech robí, čo robí, každý jeho ťah dôjde do pozície odkiaľ vie vyhrať súper. Vtedy teda musí byť pozícia (a, b) prehrávajúca.

Z tohto pozorovania sa vieme dostať k jednoduchému dynamickému programovaniu ktoré nám vyrieši prvú sadu: iterujeme a od 1 po a_2 a b od 1 po b_1 , a pre každú pozíciu si skontrolujeme, či existuje medzi ťahmi nejaký do prehrávajúcej pozície (pre všetky “menšie” pozície sme si už v dynamickom programovaní spočítali odpoveď). Ak áno, zaznačíme si pozíciu ako vyhrávajúcu. Ak nie, tak ako prehrávajúcu.

Po vypočítaní odpovede pre všetky pozície jednoducho spočítame počet vyhrávajúcich pozícií v intervale hier ktoré Bláčik a Anička budú hrať.

Pre každú z t otázok vieme použiť tú istú tabuľku – netreba ju navyše prepočítavať ak ju spravíme dost veľkú.

Toto riešenie funguje v časovej zložitosti približne $O(\max(a_2, b_2)^2 \log(\max(a_1, b_1)))$ (pomocou zložitejšej matematiky si vieme odhadnúť že priemerný počet možných ťahov na pozíciu je v skutočnosti približne logaritmický od možného počtu pozícií, a nie lineárny).

Listing programu (C++)

```
1 #include<bits/stdc++.h>
2
3 using namespace std;
4
5 #define FOR(i,n)      for(int i=0;i<(int)n;i++)
6 #define lli long long int
7
8 bool memo(lli a, lli b, vector<vector<int>> &W) {
9     if (a == 0 || b == 0) return true;
10    if (W[a][b] != -1) return W[a][b];
11
12    bool seen_l = false;
13    if (a < b) {
14        int k = 1;
15        while (k * a <= b) {
16            if (!memo(a, b - k * a, W)) seen_l = true;
17            k ++;
18        }
19    }
20    else {
21        int k = 1;
22        while (k * b <= a) {
23            if (!memo(a - k * b, b, W)) seen_l = true;
24            k ++;
25        }
26    }
27
28    W[a][b] = seen_l;
```

```

29     return seen_l;
30 }
31
32 int main() {
33     cin.sync_with_stdio(false); cin.tie();
34     cout.sync_with_stdio(false); cout.tie();
35     int t; cin >> t;
36     vector<vector<int>> > winning(501, vector<int>(501, -1));
37
38     FOR(i, t) {
39         lli a1, a2, b1, b2;
40         cin >> a1 >> a2 >> b1 >> b2;
41
42         lli count = 0;
43         for (int a = a1; a <= a2; a++) {
44             for (int b = b1; b <= b2; b++)
45                 count += (int)(memo(a, b, winning));
46         }
47         cout << count << "\n";
48     }
49 }

```

Prvé pozorovanie

Oplatí sa nám naozaj skúšať všetkých $\max a_2 \max b_2$ pozícií?

Ak $a \leq b < 2a$ (alebo $b \leq a \leq 2b$), vtedy nemá mačiatko na ťahu na výber – má presne jeden ťah ktorý môže spraviť. Zamyslime sa tak nad prípadom, že $b \geq 2a$. Mačiatko na ťahu má aspoň dva možné ťahy: do pozície $(b \bmod a, a)$, alebo do pozície $(a + (b \bmod a), a)$. Všimnite si, že v pozícii $(a + b \bmod a, a)$ je možný len jediný ťah: do $(b \bmod a, a)$.

Rozoberme si najskôr prípad, že pozícia $(b \bmod a, a)$ je vyhrávajúca. V tom prípade, ak mačiatko na ťahu (povedzme že je to Anička) prejde do pozície $(a + (b \bmod a), a)$, Bláčík musí následne ísť do pozície $(b \bmod a, a)$, kde sa dostane na ťah Anička. Táto pozícia je však vyhrávajúca, teda v tomto prípade má Anička vyhrávajúcu stratégiu s pozície (a, b) .

No a v prípade, že je pozícia $(b \bmod a, a)$ prehrávajúca? Potom vieme, že keď sa tam Anička pohne, nech Bláčík robí čo robí, určite prehrá ak Anička ďalej hrá optimálne.

Ukázali sme si tak, že v oboch prípadoch má Anička vyhrávajúcu stratégiu, teda vždy keď $b \geq 2a$ je pozícia (b, a) vyhrávajúca. Všimnite si, že nemusíte počítať presnú Aničkinu stratégiu – stačí nám vedieť či z pozície vyhrá, alebo nie.

Takto vieme jednoducho overiť, či je nejaké pozícia (a, b) vyhrávajúca (bez ujmy na všeobecnosti uvažujme $a \leq b$, inak je situácia symetrická): ak $b \geq 2a$ (v tomto prípade je zahrnutý prípad $a = 0$), pozícia je vyhrávajúca. Inak sa posuňme na pozíciu $(b - a, a)$ a postupujeme rovnakým spôsobom, pričom pozícia (a, b) je vyhrávajúca práve ak je pozícia $(b - a, a)$ prehrávajúca.

Kolko nám takýto výpočet bude trvať? Po dvoch ťahoch z (a, b) dostaneme na $(2a - b, b - a)$, $b - a < a$, takže sa nám najväčšie z čísel zmenšilo o polovicu. Teda určite na overenie výhernosti pozície (a, b) ($s a \leq b$) nepotrebujeme viac ako $2 \log b$ krokov.

Takto si jednoducho vieme overiť kto vyhráva každú z hier ktorú mačiatka spolu hrajú, a algoritmus celkovo zaberie $O((a_2 - a_1 + 1)(b_2 - b_1 + 1) \log \max(a_2, b_2))$ času, a dokonca ho vieme implementovať v konštantnej pamäti¹. Toto riešenie stačí na štyri body.

[1] V prípade, že ho implementujete cez cykly. Ak používate rekúziu ako doleuvedený kód, tak zásobník v rekúzii zožerie $O(\log \max(a_2, b_2))$ pamäti.

Listing programu (C++)

```

1 #include<bits/stdc++.h>
2

```

¹

```

3  using namespace std;
4
5  #define FOR(i,n)      for(int i=0;i<(int)n;i++)
6  #define lli long long int
7
8  bool memo(lli a, lli b) {
9      if (a == 0 && b == 0) return true;
10     if (a > b) return memo(b, a);
11     if (b > a * 2) return true;
12     return (!memo(b - a, a));
13 }
14
15 void solve() {
16     lli a1, a2, b1, b2;
17     cin >> a1 >> a2 >> b1 >> b2;
18
19     lli count = 0;
20     for (lli a = a1; a <= a2; a++) {
21         for (lli b = b1; b <= b2; b++) count += (int)(memo(a, b));
22     }
23     cout << count << "\n";
24 }
25
26 int main() {
27     cin.sync_with_stdio(false); cin.tie();
28     cout.sync_with_stdio(false); cout.tie();
29     int t; cin >> t;
30     FOR(i, t) solve();
31 }

```

Ako to zlepšiť?

V druhej polovici vstupov mačiatka hrajú priveľa hier aby sme každú kontrolovali separátne². Musíme vymyslieť niečo lepšie.

[2] Pýtate sa kedy to stihnú? Mačiatka sú *veľmi* hyperaktívne

Jedna možnosť, ktorá podľa šikovnosti implementácie vie získať medzi 6 a 8 bodmi je pre každé a medzi a_1 a a_2 simulovať všetky b naraz, a v každom kroku počítať koľko z pozícií vyhráva okamžite (podľa horeuvedeného pravidla).

Implementácia tejto verzie nie je úplne triviálna, a nechávam ju ako zamyslenie sa na čitateľa (hoci je vzorák, naopak, implementačne veľmi jednoduchý, ako čoskoro uvidíte, skúsiť si nakódovať túto verziu je samo o sebe vcelku zaujímavá úloha na zamyslenie :)

Optimálne riešenie

Podme sa ďalej zamyslieť nad našou úvahou, že pozícia (a, b) s $a \leq b$ je vyhrávajúca ak $b \geq 2a$. Ak nie je, potom je vyhrávajúca ak je $(b - a, a)$ prehrávajúca. Teraz máme dva prípady. Ak $a \leq 2(b - a)$, potom je táto pozícia vyhrávajúca, podľa predchádzajúceho pravidla. Toto je ekvivalentné prípadu, že $b \geq \frac{3}{2}a$.

V druhom prípade platí $a < 2(b - a)$, a hra pokračuje do stavu $(2a - b, b - a)$. Zás máme dva prípady: ak $b - a \geq 2(2a - b)$, čiže $b \geq \frac{5}{3}a$, vtedy je pozícia víťazná, inak...

$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \dots$, to znie skoro povedome! A veru, ak takto pokračujeme ďalej, dostaneme sériu zlomkov susediacich fibonacciho čísel. Indukciou si vieme dokázať, že po $2i + 1$ iteráciách z (a, b) takto dostaneme $(b \cdot F_{2i+1} - a \cdot F_{2i+2}, a \cdot F_{2i+1} - b \cdot F_{2i})$, a po $2(i + 1)$ iteráciách z (a, b) dostaneme $(a \cdot F_{2i+3} - b \cdot F_{2i+2}, b \cdot F_{2i+1} - a \cdot F_{2i+2})$, pričom F_n zaznačuje n -tého fibonacciho číslo ($F_0 = 0, F_1 = 1, F_{n+2} = F_n + F_{n+1}$).

Z tohto dostaneme, že na to, aby bola pozícia (a, b) vyhrávajúca, musí platiť $b \geq aF_{2k+4}/F_{2k+3}$ a zároveň $b \leq aF_{2k+5}/F_{2k+4}$ (ako sa ukáže, pre všetky k)

²

Ak ste niekedy počuli z zlatom reze, tak viete že pomer vedľajších Fibonacciho čísel konverguje ku ϕ , a teda (a, b) je vyhrávajúca pozícia práve vtedy ak $b \geq \phi a$ (alebo $a \geq \phi b$).

Takto vieme počet vyhrávajúcich pozícií pre Aničku spočítať v konštantnom čase: jednoducho spočítame pre každé možné a , koľko b v intervale $[b_1, b_2]$ platí $b \geq a\phi$, alebo $b \leq a/\phi$.

ϕ vieme jednoducho odhadnúť ako pomer dostatočne veľkých Fibonacciho čísel – takto sa vieme vyhnúť problémom s zaokrúhľovaním necelých čísel pri overovaní koncov víťazných intervalov.

Celé riešenie tam má lineárnu časovú, a konštantnú pamäťovú zložitosť.

Listing programu (C++)

```
1  #include<bits/stdc++.h>
2
3  using namespace std;
4
5  #define FOR(i,n)      for(int i=0;i<(int)n;i++)
6  #define lli long long int
7
8  void solve(lli f1, lli f2) {
9      lli a1, a2, b1, b2;
10     cin >> a1 >> a2 >> b1 >> b2;
11
12     if (a2 - a1 > b2 - b1) {
13         swap(a1, b1); swap(a2, b2);
14     }
15
16     lli count = 0;
17     for (lli a = a1; a <= a2; a++) {
18         lli lb = a * f1 / f2;
19         lli ub = a * f2 / f1;
20         if (ub * f1 < f2 * a) ub++;
21
22         count += max(0LL, min(lb, b2) + 1LL - b1) + max(0LL, b2 - max(ub, b1) + 1LL);
23     }
24     cout << count << "\n";
25 }
26
27 int main() {
28     cin.sync_with_stdio(false); cin.tie();
29     cout.sync_with_stdio(false); cout.tie();
30     int t;
31     cin >> t;
32
33     lli f1 = 1, f2 = 1;
34     FOR(i, 45) {
35         swap(f1, f2);
36         f2 += f1;
37     }
38
39     FOR(i, t) solve(f1, f2);
40 }
```

8. Algoritmické problémy

Najskôr zlé riešenie

Poznámka od autora: *veľmi ťažko sa mi vyrábali vstupy, ktoré by toto nesprávne riešenie neprešlo, pretože sa veľmi podobá správne. Niekedy nie je zlé začať aj s niečím nesprávnym a skúsiť to opraviť.*

Pre každý problém skúsime nakódovať čo najviac jeho algoritmov, ale len toľko, na koľko nám ešte “vyjde potešenie”. Teda postupne skúsime kódovať ešte nenakódené algoritmy potrebné na vyriešenie problému a zakaždým znížime potešenie, ktoré za problém získame. Skončíme v momente, keď by pridaním ďalšieho algoritmu získané potešenie vyriešenia problému kleslo pod nulu.

Toto riešenie samo o sebe prejde na nejakých vstupoch.

Môžeme skúsiť spraviť viac iterácií tohoto postupu, možno sa to zlepši... Ak by sme boli veľmi šikovní a mali nejaké ďalšie nápady a k tomu trochu šťastia, mohli sme už teraz oklamať testovača a získať aj nejaké body. My si však ukážeme ako sa táto úloha mala riešiť.

Len to opravme

Uvedomme si, kde toto riešenie zlyhá. Na to si najprv lepšie pomenujme, čo vlastne robí.

Pre každý algoritmus, ktorý nakódujeme, vyberieme nejaký problém, ktorý za neho “zaplatí”. Takto to budeme volať aj vo zvyšku vzoráku. Náš program naivne rozhoduje, ktorý problém zaplatí za ktorý algoritmus. Keďže ešte nemáme OK, tak to zjavne robí neoptimálne.

Môžeme si všimnúť 2 problémy:

1. V optimálnom riešení môže niekedy zaplatiť za jeden algoritmus aj viac ako jeden problém, čo náš program nezistí.
2. Niekedy môžeme určiť, že za algoritmus x zaplatí problém p určitú hodnotu potešenia, ale optimálne by bolo, aby zaplatil menej. To sa nám však už nikdy nepodarí, lebo sa nevraciam späť.

Prvý problém je ľahké vyriešiť. Namiesto toho, aby sme vždy určili nejaký problém, ktorý zaplatí za algoritmus, určíme len nejaký problém, ktorý algoritmu zaplatí 1 potešenie. Síce musíme náš postup zopakovať veľa krát, ale to sa ukáže ako jednoduchá vec.

Druhý problém už nie je až tak triviálne riešiť, ale predstavme si to takto. Sme v nejakom stave, kde máme záznamy tipu problém p platí algoritmu x jedno potešenie. Predstavme si, že chceme nájsť problém, ktorý zaplatí algoritmu x , ale všetky problémy, ktoré potrebujú x už nemajú potešenie navyše. Potom môžeme spraviť nasledovnú úpravu. Vyberieme nejaký problém p , ktorý potrebuje algoritmus x , ale platí nejakému inému algoritmu y . Ak existuje problém q , ktorý potrebuje algoritmus y a ešte má potešenie navyše, môže q zaplatiť jedno potešenie algoritmu y , tým pádom p nemusí platiť y a môže zaplatiť x .

Takto sme vlastne presunuli potešenie z q do x , a nič iné sa nezmenilo. Teda okrem toho, kto komu platí, ale to nás v skutočnosti nezaujíma.

V pôvodnom nesprávnom riešení hľadáme len problémy p . Teraz máme ešte stále nesprávne riešenie, lebo hľadáme len problémy q vzdialené jednu iteráciu hľadania (len jeden krát sa vrátíme naspäť).

Túto myšlienku však môžeme aj zovšeobecniť. Nebudeme hľadať len problémy q , ktoré ešte majú potešenie, ale budeme hľadať aj hlbšie a vždy vzdialenejšie problémy, ktoré ešte majú potešenie.

Toto je vlastne celá myšlienka vzoráku, vo zvyšku už len dokážeme správnosť a pozrieme sa na implementáciu. Naozaj to nebolo až tak ťažké, že? A ukáže sa to ešte omnoho jednoduchšie.

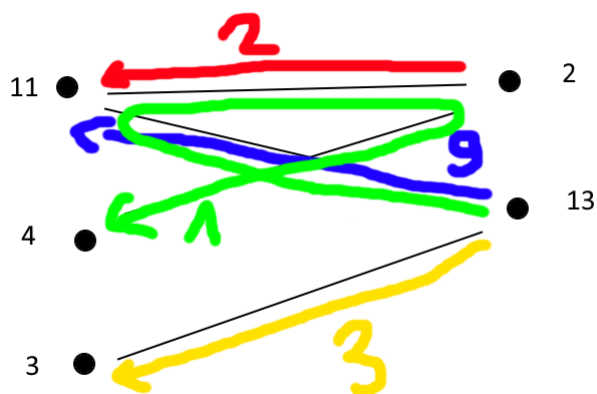
Vyzerá to povedome

Môžeme si celú úlohu predstaviť ako bipartitný graf. Problémy sú napravo a algoritmy naľavo. Hrany vedú medzi algoritmom a problémom vtedy, ak problém algoritmus potrebuje.

Vždy, keď určíme, že nejaký problém platí nejakému algoritmu, vlastne sme povedali, že hrana medzi nimi má smerom k algoritmu o 1 väčšiu hodnotu (je jedno, ktorým smerom to povieme, ale je dobré si nejaký vybrať).

Keď hľadáme problém, ktorý zaplatí za algoritmus x , robíme vlastne jednoduché prehľadávanie. Vždy sa najskôr skúsime vrátiť nejakou hranou (toto je ekvivalentné tomu, že nejaký problém potrebuje algoritmus x), čím najdeme naše p . Potom z p ideme hranou, po ktorej ide niečo v smere (teda p platí niekomu inému) ďalej znovu nejakou hranou... Toto opakujeme, až kým nenájdeme problém q , ktorý má ešte voľné potešenie. Teraz pošleme všetkými hranami, ktorými sme prešli jedno potešenie opačným smerom.

Môžeme si ukázať, ako by taký beh nášho programu mohol vyzeráť. Máme 3 algoritmy s cenami 11, 4 a 3 potešenia a 2 problémy, ktoré dajú 2 a 13 potešenia. Takto nejakým spôsobom by mohol náš algoritmus pridelať, kto komu platí:



Poradie krokov: červená, modrá, žltá, zelená

Prvé 3 kroky by zvládol aj nesprávny algoritmus zo začiatku. Preto je pre nás zaujímavý zelený krok. V tomto kroku má algoritmus 13 ešte jedno voľné potešenie. Toto voľné potešenie pridelí algoritmu 11, tomu zoberie jedno potešenie od problému 2 a to pridelí 4.

DOKEDY???

Ako môžeme vidieť na obrázku, niektoré kroky nášho algoritmu je dobré robiť, lebo nám zlepšujú odpoveď. Ak by sme si však pred zeleným krokom povedali, že už máme dosť, a chceli by sme – nedajbože – teraz skúmať odpoveď spôsobom: 13 má ešte jedno voľné potešenie a jej algoritmy sú zaplattené teda odpoveď je 1. Dopracujeme sa k nesprávnemu riešeniu.

Ukážeme si ale, že ak vykonáme maximálny počet krokov, dokážeme zistiť, aké je optimálne riešenie.

Zjavne ak má nejaký problém na konci algoritmu ešte nejaké potešenie, určite ho v optimálnom riešení vyriešime. POZOR okrem toho môžeme vyriešiť aj nejaké iné problémy, ktoré majú už teraz potešenie 0. Keďže stačí výsledok len povedať a nie aj skonštruovať, ukáže sa, že na získanie výsledku stačí len sčítať pozitívne hodnoty. Naozaj to je ľahké.

Nech po vykonaní maximálneho počtu krokov existuje nejaký problém p , ktorý má ešte voľné potešenie. Z maximálnosti počtu krokov vieme, že všetky algoritmy, s ktorými je spojený, už sú zaplattené. Zlé jazyky by však mohli povedať nasledovné: *Čo ak je medzi algoritmami susednými s p nejaký algoritmus x , za ktorý platí problém q , ktorý ale nevyriešime?*

Toto je seriózný problém. Potom by sme nemohli povedať, že vyriešime p , lebo by sme vyriešenými problémami nezaplatili za algoritmi potrebné na jeho vyriešenie – q nie je vyriešený, ale platí za algoritmy pre p . Avšak zlé jazyky si neuvedomujú silu toho, že niečo robíme maximálny počet krát! Keďže q sme nevyriešili, existuje algoritmus y susedný s q , ktorý nie je zaplattený. Teda existuje cesta $p \rightarrow x \rightarrow q \rightarrow y$, po ktorej môžeme poslať nejaké potešenie, keďže p ešte potešenie má. Teda sme mohli ešte raz zopakovať náš algoritmus, čo je spor.

Ak teda náš postup opakujeme, kým sú v grafe cesty, po ktorých vieme poslať potešenie, mali by sme ľahko vedieť povedať, aký bude maximálny profit – iba sčítame tie, ktoré ešte majú potešenie nazvyš.

Už máme všetko dokázané, preto sa môžeme veselo pustiť do implementácie. Ukážeme si však trik, ktorý nám pomôže použiť radšej staré známe algoritmy, ako sa mordovať s implementáciou, aj keď verím, že by to nebolo zas tak náročné.

Implementácia

Namiesto toho, aby sme mali pre každý vrchol to, koľko potešenia nám pridá, môžeme do grafu pridať jeden vrchol, ktorý je nekonečným zdrojom potešenia a je spojený so všetkými problémami hranou. Po tejto hrane môže prejsť maximálne toľko potešenia, koľko potešenia nám dá vyriešenie daného problému. Zjavne v takomto upravenom grafe hľadáme cestu z tohto zdroja do nejakého algoritmu, za ktorý ešte nie je zaplattené, pričom nechceme prekročiť kapacitu na žiadnej hrane.

Ďalej si vieme podobne pridať ešte jeden vrchol spojený so všetkými algoritmami. Tento vrchol bude nekonečným pohlcovačom potešenia, kde hrany do neho budú mať rovnaké kapacity ako ceny daných algoritmov.

Hrany od problémov k algoritmom majú nekonečné kapacity.

Teraz keď nájdeme cestu medzi pridanými vrcholmi⁴, pošleme po nej nejaké potešenie a zapamätáme si aj, koľko potešenia môžeme vrátiť naspäť. (zelená cesta z obrázka vráti po hornej hrane jedno potešenie, lebo ňou

³<https://www.youtube.com/watch?v=6E1zUaTnLx0>

⁴Použijeme terminológiu **zdroj** – nekonečný zdroj a **stok** – nekonečný požierač.

išli 2 v opačnom smere) To vieme napríklad implementovať tak, že opačným hranám sa zvýši kapacita.

V tomto grafe môžeme postupne hľadať nejakú cestu zo zdroja do stoku pomocou prehľadávania (napríklad DFS) a poslať po nej jedno potešenie.

Dokonca môžeme po ceste rovno poslať maximum potešenia, ktoré sa ešte zmestí do kapacity hrán na ceste.

Takéto riešenie mohlo dostať 6 z 8 bodov. Na 8 bodov stačilo DFS vymeniť za BFS. Teda vždy nájsť najkratšiu cestu zo zdroja do stoku⁵.

Toto je koniec vzoráku, ale ako by to bolo, kebyže sa to nedá napísať celé oveľa kratšie pohromade.

A čo to vlastne robíme?

Už som naznačil aj terminológiu, že hľadáme maximálny tok v sieti. Nám sa však oplatí pozrieť na celú úlohu ako na minimálny rez, keďže tieto hodnoty sú rovnaké.

Čo je rez v sieti? Vyberieme nejaké hrany, tie “prerežeme” a musí platiť, že v novej sieti sa nedá dostať zo zdroja do stoku.

Jedna možnosť je napríklad prerezať všetky hrany medzi algoritmi a problémami. My však chceme nájsť takú možnosť, kde súčet hodnôt potešenia na hranách rezu je najmenší možný.

Takže celý vzorák by vlastne mohol byť takýto:

Zoberieme náš graf aj s pridaným zdrojom a stokom a nájdeme v ňom minimálny rez. Tento rez zjavne nebude obsahovať žiadne hrany, ktoré vedú z algoritmov do problémov, lebo tie majú kapacitu nekonečno.

Rez si interpretujeme takto:

1. Ak prerežeme nejakú hranu zo zdroja do problému, znamená to, že tento problém nevyriešime.
2. Ak prerežeme nejakú hranu z algoritmu do stoku znamená to, že tento algoritmus nakódime.

Všimnime si, že ak sa rozhodneme nejaký problém vyriešiť, znamená to, že nakódime všetky jeho susedné algoritmy, lebo inak by sme nedostali rez.

Obe možnosti pre nás znamenajú stratu potešenia, ak teda na začiatku sčítame potešenie pre všetky problémy a odčítame hodnotu minimálneho rezu, dostaneme požadovaný výsledok.

Počítať minimálny rez môžeme Edmonds-Karpovým algoritmom, ktorý sme si vlastne trochu obrazne ukázali vyššie – to je ten s BFS.

Listing programu (C++)

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4  typedef long long ll;
5
6  #define inf 1e18
7
8  struct EdmondsKarp {
9      vector<unordered_map<ll, ll>> g;
10     vector<ll> vis;
11     vector<ll> pred;
12     EdmondsKarp(ll n) : g(n), vis(n), pred(n, -1) {}
13     void addEdge(ll a, ll b, ll c, ll rc = 0) {
14         g[a][b] = c;
15         g[b][a] = rc;
16     }
17     ll bfs(ll s, ll t) {
18         fill(begin(pred), end(pred), -1);
19         queue<ll> q;
20         q.push(s);
21         pred[s] = -2;
22         while (!q.empty() && pred[t] == -1) {
```

⁵Analýzu časovej zložitosti neuvádzame, ale verte mi, stihne sa to.

```

23     ll v = q.front();
24     q.pop();
25     for (auto e : g[v]) {
26         if (pred[e.first] == -1 && e.second) {
27             pred[e.first] = v;
28             q.push(e.first);
29         }
30     }
31 }
32 if (pred[t] != -1) {
33     ll f = inf;
34     for (ll i = t; i != s; i = pred[i]) {
35         f = min(f, g[pred[i]][i]);
36     }
37     for (ll i = t; i != s; i = pred[i]) {
38         if ((g[pred[i]][i] - f) <= 0) g[pred[i]].erase(i);
39         g[i][pred[i]] += f;
40     }
41     return f;
42 }
43 return 0;
44 }
45 ll calc(ll s, ll t) {
46     ll ans = 0;
47     while (ll p = bfs(s, t)) {
48         ans += p;
49     }
50     return ans;
51 }
52 };
53
54 int main() {
55     cin.tie(0)->sync_with_stdio(0);
56     cin.exceptions(cin.failbit);
57     ll n, m;
58     cin >> n >> m;
59     vector<ll> a(n), b(m);
60     for (auto &i : a) cin >> i;
61     for (auto &i : b) cin >> i;
62     ll nn = n + m, s = nn, t = nn + 1;
63     EdmondsKarp f = EdmondsKarp(nn + 2);
64     for (ll i = 0; i < n; i++) {
65         ll ni;
66         cin >> ni;
67         for (ll j = 0; j < ni; j++) {
68             ll tool;
69             cin >> tool;
70             tool--;
71             f.addEdge(tool + n, i, inf);
72         }
73     }
74     for (ll i = 0; i < m; i++) {
75         f.addEdge(s, i + n, b[i]);

```

```
76     }
77     ll sum = 0;
78     for (ll i = 0; i < n; i++) {
79         sum += a[i];
80         f.addEdge(i, t, a[i]);
81     }
82     cout << sum - f.calc(s, t) << endl;
83 }
```

Zložitosť tohoto algoritmu, je trochu paradoxne $O(VE^2)$, kde V je počet vrcholov a E je počet hrán. V našom prípade $V = n + m$ a $E \in O(nm)$, čo pre obmedzenia zo zadania dáva dosť zlú časovú zložitosť.

Dôležité sú 2 veci:

1. Táto zložitosť je najhoršia možná a veľakrát bude algoritmus bežať omnoho rýchlejšie, ako aj v našom prípade, čo závisí od toho ako sieť vyzerá.
2. Dá sa to aj rýchlejšie nejakými inými tokovými algoritmami, avšak pre rámec tejto úlohy a vzoráku to nie je podstatné.