Korešpondenčný seminár z programovania
XXXII. ročník, 2014/15
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

Baklažán
(max. 0 b za popis, 15 b za program)

1. Zerg Bot

1 - Komnata

Tu stačilo robota odnavigovať na správne tlačidlo, prepnúť ho a odnavigovať do cieľa. To, ktoré tlačidlo je správne, sa dalo zistiť napríklad tak, že ste postupne vyskúšali všetky. Bolo to tretie tlačidlo zdola.

Kód:

```
doprava
krok
krok
doprava
krok
prepni
doprava
krok
krok
doprava
krok
krok
krok
```

2 - Učko

V tomto leveli je na prvý pohľad jasné, čo má robot urobiť. Problém ale je, že nemá dosť pamäte na všetky príkazy. Môžeme si však všimnúť, že U-čko sa skladá z troch rovnakých častí. Napíšeme si teda funkciu, ktorá nám prejde jednu časť U-čka a túto funkciu potom zavoláme trikrát zasebou. Treba si však dať pozor, aby bol robot po dokončení jedného ramena U-čka správne otočený, na konci funkcie ho preto otočíme doprava.

Kód:

```
funkcia0
funkcia0
funkcia0

aha funkcia0:
krok
prepni
krok
krok
doprava
```

3 - Schody

Potrebujeme prejsť veľa rovnakých schodov, preto sa nám oplatí napísať si funkciu na prejdenie jedného schodu:

```
aha funkcia0:
krok
doprava
krok
dolava
```

Teraz však narazíme na problém: môžeme použiť už iba dva príkazy a schodov je oveľa viac, než dva.

Použijeme preto jeden zaujímavý trik: napíšeme funkciu, ktorá prejde jeden schod a potom *zavolá sama seba*. Keď sa táto funkcia začne vykonávať, prejde jeden schod a sama sa zavolá znovu. Opäť prejde jeden schod a znovu sa zavolá. A takto bude náš program pokračovať až donekonečna, resp. kým robotovi nedôjde batéria, alebo nepríde do cieľa.

Kód:

```
funkcia0

aha funkcia0:
krok
doprava
krok
dolava
funkcia0
```

Ako už viete zo študijného textu, tento princíp sa volá *rekurzia*. V našom prípade ide o nekonečnú rekurziu, čo pekne vystihuje Obr. 4. Na našu `funkciu0` sa môžeme pozerať tak, že je to funkcia, ktorá prejde nekonečne veľa schodov. Urobí to tak, že najprv prejde jeden schod (prvé 4 príkazy) a potom ešte nekonečne veľa ďalších (posledný príkaz).

4 - Šachovnica

Najjednoduchšie asi bude ísť po riadkoch a cestou prepnúť správne políčka.

Existuje viacero spôsobov, ako prejsť takúto cestu. Jednou z možností je napísať si štyri funkcie, ktoré sa navzájom volajú v nekonečnej rekurzii. My si ale vystačíme aj s niečím jednoduchším.

Budeme potrebovať, aby robot prepol prepínač na každom druhom políčku, cez ktoré prejde. Napíšme si teda funkciu, ktorá prejde dve políčka, pričom to, na ktoré stúpi skôr, prepne:

```
aha funkcia0:
krok
prepni
krok
```

S jej pomocou si môžeme napísať funkciu, ktorá prejde jeden riadok šachovnice a prepne pri tom potrebné políčka za predpokladu, že štartujeme na tom konci riadka, ktorý nemá byť zapnutý:

```
aha funkcia1:
funkcia0
funkcia0
funkcia0
funkcia0
```

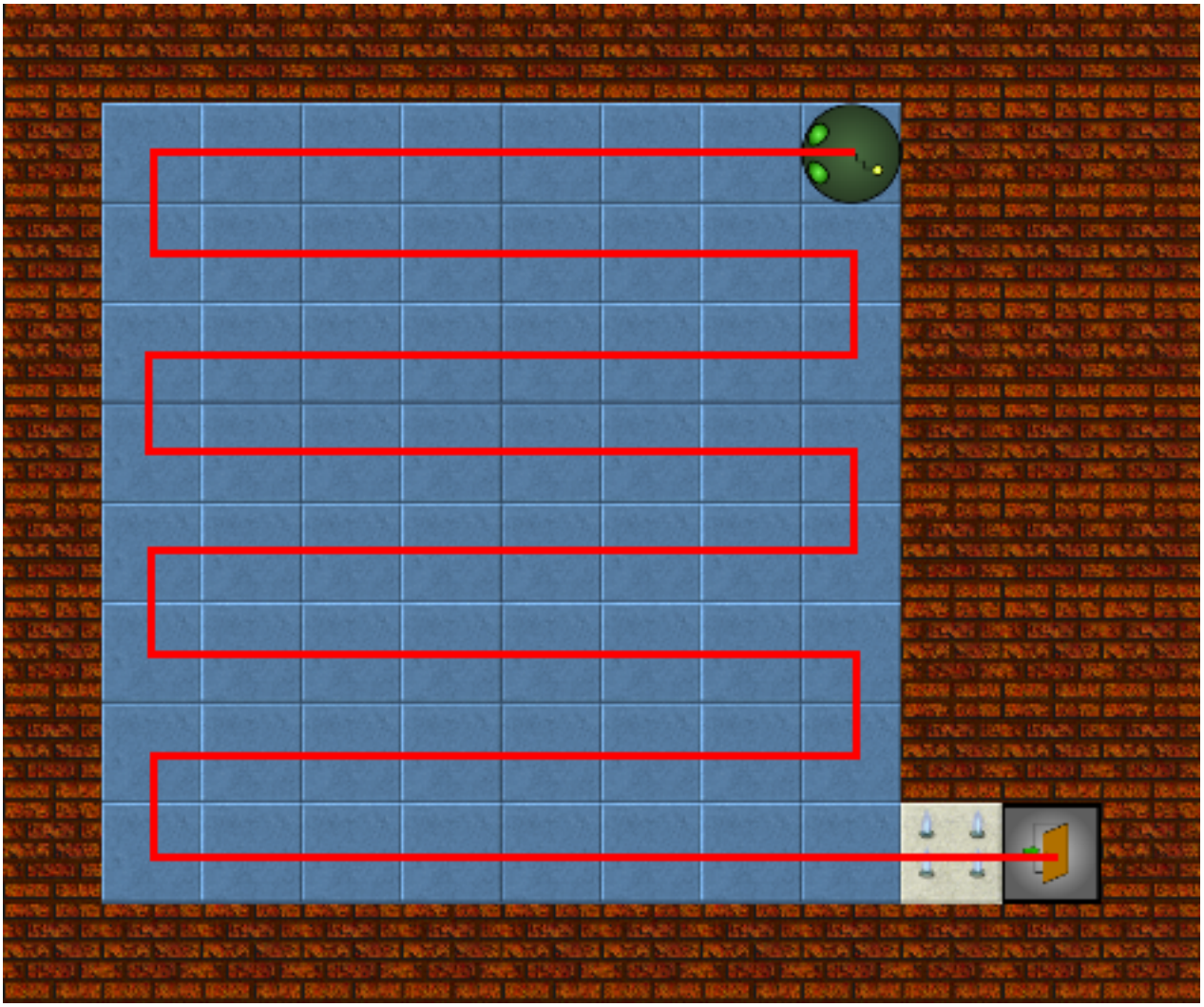
Všimnime si, že táto funkcia sa po zapnutí posledného prepínača v riadku bude snažiť urobiť ešte jeden krok. To nám ale neprekáža, robot sa iba pokúsi urobiť krok do steny (alebo na JetTorch, ktorý už v tom čase bude vypnutý).

Aj na prejdenie ľavotočivej a pravotočivej zákruty si môžeme napísať funkcie:

```
aha funkcia2:
dolava
krok
dolava

aha funkcia3:
doprava
krok
doprava
```

Na ceste, po ktorej má robot prejsť, sa striedajú rovné úseky a zákruty, pričom zákruty sú striedavo ľavotočivé a pravotočivé. Napíšme si teda funkciu, ktorá nám prejde dva rovné úseky a zákruty nasledujúce po nich:



Obr. 1: Zamýšľaná trajektória

```
aha funkcia4:
funkcia1
funkcia2
funkcia1
funkcia3
```

Keď túto funkciu zavoláme štyrikrát po sebe, robot by mal prejsť celú šachovnicu. Problém ale je, že po prejdení posledného riadka sa bude snažiť prejsť ešte jednu pravotočivú zákrutu. Preto ju radšej zavoláme iba trikrát a posledné dva rovné úseky (aj so zákrutou medzi nimi) rozpíšeme. Robot by potom skončil na políčku s JetTorchom (nezabúdajte, že `funkcia1` sa po prejdení riadka snaží urobiť ešte jeden krok), preto na koniec pridáme ešte jeden krok.

Hlavný kód:

```
funkcia4
funkcia4
funkcia4
funkcia1
funkcia2
funkcia1
krok
```

5 - Zeleno modrá cesta

Kľúčovým pozorovaním v tomto leveli je to, že po zapnutom tlačidle treba ísť najbližšie dva kroky smerom na juh a po vypnutom prepínači treba ísť najbližšie dva kroky smerom na východ. Po urobení dvoch krokov správnym smerom sa bude robot nachádzať na nasledujúcom prepínači.

Tiež si môžeme všimnúť, že máme povolených iba veľmi málo príkazov. Preto opäť využijeme nekonečnú rekurziu: napíšeme si funkciu, ktorá (za predpokladu, že robot stojí na prepínači) urobí dva kroky správnym smerom, a potom sa zavolá znovu.

Tu ale narazíme na jeden problém: robot nevie zistiť, ktorým smerom je práve otočený, teda nebude vedieť, ktorým smerom je východ a ktorým smerom je juh. To môžeme vyriešiť tak, že vždy potom, čo robot prejde na ďalší vypínač, sa otočí na východ (teda ak išiel na juh, otočí sa doľava a ak išiel na východ, ostane tak, ako je). Na to si ale bude musieť pamätať, či bolo predošlé tlačidlo stlačené alebo nie. To docielime tak, že namiesto toho, aby sa robot na tlačidle iba otočil alebo neotočil (podľa toho, či je zapnuté) a potom urobil dva kroky, sa na tlačidle zavolá buď funkcia, ktorá urobí dva kroky smerom na východ, alebo funkcia, ktorá urobí dva kroky smerom na juh a potom sa otočí na východ.

Kód:

```
funkcia0

aha funkcia0:
funkcia1 ak svieti
funkcia2 ak nesvieti
funkcia0

aha funkcia1:
doprava
krok
krok
dolava

aha funkcia2:
krok
krok
```

Počas behu programu si môžeme všimnúť, že niekedy sa v jednom volaní `funkcie0` zavolá aj `funkcia1` aj `funkcia2` (nastane to vtedy, keď je robot na zapnutom tlačidle ale po vykonaní `funkcie1` sa ocitne na vypnutom tlačidle). To nám v našom prípade neškodí, ale keby sme chceli, aby sa v jednom volaní `funkcie0` zavolala vždy

iba jedna z funkcií 1 a 2, mohli by sme to docieliť napríklad tak, že by sme `funkciu0` zavolali aj na konci `funkcie1`, teda v prípade, že tlačidlo bolo zapnuté, by sa ani netestovalo, či sa má spustiť `funkcia2`.

1 - Riedke bludisko

Jeden z možných spôsobov, ako dôjsť do cieľa, je ísť vždy čo najdlhšie rovno a keď prídeme ku stene, otočiť sa doprava. To vieme, s pomocou nekonečnej rekurzie (Obr. 4 v študijnom texte o rekurzii), zapísať napríklad takto:

```
funkcia0

aha funkcia0:
doprava ak je stena vpredu
krok
funkcia0
```

2 - Hrebeň

Keďže tlačidlá sú v rôznych zuboch hrebeňa rôzne ďaleko, budeme potrebovať funkciu, ktorá robotom pohne po najbližšie zapnuté tlačidlo. Tá môže vyzerať takto:

```
aha funkcia0:
krok
funkcia0 ak nesvieti
```

Táto funkcia sa teda bude volať a hýbať robotom, až kým nestúpi na tlačidlo. V tom okamihu nebude platiť podmienka `ak nesvieti`, teda funkcia sa už znovu nezavolá a robot zastane. Takýto druh rekurzie sa nazýva *chvostová rekurzia*. Podobne si môžeme napísať funkciu, ktorá robotom pohne po najbližšiu stenu:

```
aha funkcia1:
krok
funkcia1 ak nie je stena vpredu
```

S pomocou týchto dvoch funkcií už ľahko napíšeme funkciu, ktorá vsúpi do zubu, nájde tlačítko, stlačí ho, otočí sa a presunie sa k spodnej stene, kde sa rekurzívne zavolá. Musíme si však dať pozor, aby bol robot pri rekurzívnom volaní v takej istej situácii ako na začiatku, čo bude znamenať, že musí skončiť postavený na (teraz už vypnutom) JetTorchi otočený doprava.

```
aha funkcia2:
krok
dolava
funkcia0
prepni
dolava
dolava
funkcia1
dolava
krok
funkcia2
```

V hlavnom programe nám stačí zavolať `funkciu2`.

3 - Zeleno modrá cesta 2

Pred čítaním tohto riešenia vám odporúčam prečítať si vzorové riešenie levelu Zeleno modrá cesta (posledný level v predošlej úlohe).

Podobne ako pri prvej zeleno modrej ceste platí, že stav prepínača určuje, ktorým smerom z neho máme ísť:

- Z vypnutého tlačidla treba ísť dva kroky na sever.
- Zo zapnutého tlačidla treba ísť dva kroky smerom na východ.
- Z ukradnutého tlačidla (miesto, kde by malo byť tlačidlo, ale nie je) treba ísť dva kroky smerom na juh.

Podobne ako v zeleno modrej ceste, napíšeme si funkciu, ktorá urobí dva kroky správnym smerom, otočí sa na východ a potom zavolá sama seba.

Tu ale narazíme na problém: ako vieme rozlíšiť ukradnuté tlačidlo od vypnutého, keď pre obe platí, že nesvietia? Jednoducho tak, že sa ho pokúsime prepnúť. Ak je po prepnutí zapnuté, znamená to, že bolo iba vypnuté. Ak stále nie je zapnuté, znamená to, že je ukradnuté. Kód by teda vyzeral takto:

```
funkcia0

aha funkcia0:
funkcia1 ak svieti
prepni
funkcia2 ak svieti
funkcia3 ak nesvieti
funkcia0

aha funkcia1:
krok
krok

aha funkcia2:
dolava
krok
krok
doprava

aha funkcia3:
doprava
krok
krok
dolava
```

Tu ale príde ďalší problém: môže sa stať, že sa v jednom volaní `funkcie0` zavolá `funkcia2` a potom aj `funkcia3` (ak sa vykonaním `funkcie2` robot dostane na nesvietiace políčko). Pri volaní `funkcie3` pritom už nemusí platiť, že robot je na ukradnutom vypínači, keďže môže byť aj na vypnutom. A naozaj sa to aj stane, už pri treťom vypínači. Preto zaručíme, aby sa v jednom volaní `funkcie0` zavolala vždy iba jedna z funkcií 1, 2, 3. To urobíme tak, že aj na konci každej z funkcií 1, 2, 3 zavoláme `funkciu0` (volanie `funkcie0` na konci `funkcie0` sa tým pádom stane zbytočným).

4 - Bludisko

Na vyriešenie tohto levelu použijeme algoritmus, ktorý sa nazýva *pravidlo pravej ruky* a slúži na prehľadávanie bludísk. Je veľmi jednoduchý: na začiatku sa pravou rukou chytíme múru a celý čas ideme pozdĺž neho bez toho, aby sme ho pustili. Takýmto spôsobom prejdeme po celom obvode bludiska a buď sa vrátime späť tam, kde sme začali, alebo cestou nájdeme nejaký iný východ z bludiska (čo sa v našom prípade nestane, keďže bludisko iný východ nemá). Všimnime si, že všetky políčka nášho bludiska sa nachádzajú pri jeho obvode, teda pravidlom pravej ruky prejdeme cez všetky prepínače.

Pre nášho robota to bude znamenať, že vždy, keď vojde na nové políčko, bude sa snažiť ísť čo najviac doprava. To znamená, že ak vpravo nie je stena, tak pôjde doprava, ak je vpravo stena, tak pôjde rovno, ak je stena aj vpredu, tak pôjde doľava a ak je stena aj vľavo, tak sa vráti, odkiaľ prišiel.

Kód:

```
funkcia0

aha funkcia0:
krok
prepni ak nesvieti
doprava ak nie je stena vpravo
dolava ak je stena vpredu
dolava ak je stena vpredu
funkcia0
```

Po tom, čo robot prepne všetky vypínače, pôjde ďalej pozdĺž múru až príde do cieľa.

5 - JetTorchové peklo

Cesta obsahuje štyri rovné úseky, pričom prvý a tretí sú "bezpečné", keďže sú zakončené stenou, kým druhý a štvrtý sú "nebezpečné", keďže za nimi nasleduje JetTorch. Kľúčovým pozorovaním tohto levelu je, že každý nebezpečný úsek je rovnako dlhý ako bezpečný úsek pred ním.

Napíšme si teda funkciu, ktorá pôjde po najbližšiu stenu, potom sa otočí doľava a urobí rovnako veľa krokov, ako urobila cestou k stene. Ako začiatok si môžeme zobrať funkciu z levelu 2, ktorá ide po najbližšiu stenu:

```
aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
```

Po tom, čo robot príde k stene, sa má otočiť doľava, preto doplníme jeden príkaz:

```
aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
dolava ak je stena vpredu
```

Všimnime si, že táto funkcia sa zavolá presne toľko krát, koľko krokov urobí robot cestou k stene. A to je presne toľko, koľko má urobiť cestou od steny. Tiež si môžeme všimnúť, že žiadne volanie funkcie sa neskončí, kým robot nepríde k stene. To znamená, že ak by robot v každom zavolaní funkcie urobil ešte jeden krok po tom, ako sa otočí pri stene, urobil by presne to, čo potrebujeme. Odporúčam si pozrieť obrázok 5 zo študijného textu. Ak si nakreslíte, ako sa budú vykonávať príkazy, bude vám to oveľa jasnejšie.

Funkcia teda bude vyzerať takto:

```
aha funkcia0:
krok
funkcia0 ak nie je stena vpredu
dolava ak je stena vpredu
krok
```

Ak sa vám toto zdalo nezrozumiteľné, môžeme sa na našu funkciu pozrieť aj ako na vysvetľovanie významu slova pomocou toho istého slova. Príkaz "choď rovno po najbližšiu stenu, potom sa otoč doľava a choď rovnako dlho rovno" vieme totiž popísať nasledovne:

1. Najprv urob krok
2. - Ak je už pred tebou stena, otoč sa doľava a urob ešte jeden krok (kvôli kroku, ktorý si urobil v bode 1).
   - V opačnom prípade choď rovno po najbližšiu stenu, potom sa otoč doľava a choď rovno rovnako dlho. Nakoniec urob ešte jeden krok (za krok, ktorý si urobil v bode 1).

A presne to naša funkcia robí.

S touto funkciou bude hlavný kód jednoduchý:

```
funkcia0
dolava
funkcia0
dolava
krok
```

Ak vás zaujíma viac o rekurzii, prečítajte si študijný text o nej na strane 7 vzorových riešení Prasku, ktoré nájdete na adrese prask.ksp.sk/ulohy/solutions_pdf/6.

2. Zarovnaný kalendár

vzorák napísala Maja
(max. 6 b za popis, 4 b za program)

Úloha sa dala riešiť veľmi veľa spôsobmi. Väčšina z nich, ak sa naimplementovali šikovne, mali dobrú časovú aj pamäťovú zložitosť. Išlo teda hlavne o eleganciu vášho riešenia.

Najmenej elegantné riešenie

Úloha sa dala riešiť úplne priamo. Urobíme si zoznam prekladov dátumov z jedného kalendára do druhého. Aj toto sa dá urobiť rôzne elegantne. Ukážeme si príklad úplne najhoršieho možného riešenia:

Listing programu (C++)

```
#include<cstdio>

int main(){
        int a,n;
        scanf("%d",&a);
        if(a==1){
                scanf("%d",&n);
                for(int i=0;i<n;i++){
                        int d,m;
                        scanf("%d %d",&d,&m);
                        if((d==1)&&(m==1)){
                                printf("1 1\n");
                        }
                        if((d==2)&&(m==1)){
                                printf("2 1\n");
                        }
                        if((d==3)&&(m==1)){
                                printf("3 1\n");
                        }
                        if((d==4)&&(m==1)){
                                printf("4 1\n");
                        }
                        //atď, asi si viete predstaviť ako to pokračuje :)
                }
        }
}
```

Prečo je toto riešenie zlé? Pretože zakaždým, keď chceme preložiť nejaký dátum, musíme prechádzať cez veľmi veľa ifov. A čo je ešte horšie, všetky tie ify musíme aj napísať. Takže takéto riešenie nie je ani veľmi efektívne (urobí síce pre každý dátum iba konštantný počet operácií, ale tá konštanta je dosť veľká, pre 31.12. musíme prejsť cez 365 ifov). A už vôbec to nie je elegantné.

Trochu elegantnejšie riešenie

Trochu lepšie riešenie nám ponúka Python, kde vieme použiť dictionary (slovník), čo je dátová štruktúra, ktorej každý prvok sa skladá z kľúča a hodnoty, pričom ku hodnote vieme pristúpiť, ak poznáme kľúč. Je to naozaj podobné ako prekladový slovník, predstavte si, že máme prekladový slovník z angličtiny do slovenčiny, potom anglické slová by boli kľuče a slovenské hodnoty. Výhoda slovníka je, že prístup ku každej hodnote vieme vykonať v čase $O(1)$, teda sa nemusíme otravovať s ifmi a je to aj efektívnejšie. Ako by to vyzeralo implementované?

Listing programu (Python)

```
preklad = {
    1: {
        (1, 1): (1, 1),
        (2, 1): (2, 1),
        (3, 1): (3, 1),
        (4, 1): (4, 1),
        (5, 1): (5, 1),
        (6, 1): (6, 1),
        # ...
    },
    2: {
        (1, 1): (1, 1),
        (2, 1): (2, 1),
        (3, 1): (3, 1),
        (4, 1): (4, 1),
        (5, 1): (5, 1),
        (6, 1): (6, 1),
        # ...
    }
}


smer_prekladu, n = map(int, input().split())
for i in range(n):
    d1, m1 = map(int, input().split())
    d2, m2 = preklad[smer_prekladu][(d1, m1)]
    print("%d %d" % (d2, m2))
```

Toto riešenie má už síce dobrú časovú zložitosť, ale stále nie je pekné. Čo keby mal rok 20000 dní? Museli by sme vypísať do slovníka všetky. A to sa nám samozrejme nechce.

Vzorové riešenie

Vzorové riešenie bude robiť konverziu v obidvoch smeroch v dvoch krokoch. Najskôr si každý dátum preve-

dieme na deň v roku, (napr. 3.2. v gregoriánskom kalendári je 34. deň v roku). A následne si tento deň prevedieme na dátum podľa pravidiel druhého kalendára. Prevod z 28 dňového kalendára do dňa v roku je jednoduchšia časť, stačí nám nasledovná rovnica $denvroku = 28(mesiac-1)+den$. Prevod z klasického gregoriánskeho kalendára je trošku zložitejší, musíme si nejakým spôsobom zadrôtovať v programe dĺžky mesiacov, a kedže sa nestriedajú pravidelne a je ich len 12, tak sa nám to celkom oplatí. Najjednoduchší spôsob je urobiť si pole dĺžky 12, v ktorom si budeme udržiavať dĺžky jednotlivých mesiacov. Ale trochu viac sa nám hodí urobiť si pole dĺžky 12, kde budú uložené hodnoty hovoriace, koľko dní v roku prešlo do začiatku daného mesiaca (môžete si uvedomiť, že to by sme v konečnom dôsledku museli rátať aj z pôvodného poľa). Akú ma toto výhodu? Deň v roku potom vieme vyrátať rovnicou $denvroku = pole[mesiac-1]+den$. Ok, takže už sa vieme dostať z obidvoch kalendárov ku dňu v roku.

A ako sa teraz dostaneme k dátumu? V 28 dňovom kalendári to bude celkom priamočiare. Mesiac dostaneme vydelením dňa v roku číslom 28 a deň ako zvyšok z tejto hodnoty (plus tam máme problémy s nejakými jednotkami, ale na to ste určite prišli aj sami :)). Vyjadrené pomocou vzorcov to bude vyzerať nasledovne:

$$mesiac = (denvroku-1)/28+1$$

$$den = (denvroku-1) \bmod 28+1$$

V gregoriánskom kalendári znova použijeme naše pole s predrátanými dňami v roku po začiatok daného mesiaca. Ktoré políčko z toho poľa budeme potrebovať? Predsa najväčšie menšie ako náš hľadaný deň v roku. To samozrejme musí označovať mesiac, v ktorom sa náš deň v roku nachádza. Teda mesiac už máme. A deň potom získame ako $den = denvroku - pole[mesiac-1]$. Ak ste sa dostali vo vašom riešení sem, stačilo neseknúť sa v detailoch a 10 bodov za riešenie je vašich :).

Listing programu (C++)

```
#include <cstdio>
int mesiace [] ={0,31,59,90,120,151,181,212,243,273,304,334};
int main(){
        int t,n;
        scanf("%d %d",&t,&n);
        for (int i = 0; i < n; ++i){
                int d,m;
                scanf("%d %d",&d,&m);
                if(t==1){
                        int num=mesiace[m-1]+d;
                        printf("%d %d\n",(num-1)%28+1,(num-1)/28+1);
                }else{
                        int num=28*(m-1)+d;
                        int i=0;
                        while(num>mesiace[i]&&i<12){
                                i++;
                        }
                        printf("%d %d\n",num-mesiace[i-1],i);
                }
        }
}
```

Ešte dodám, že časová zložitosť tohto riešenia je $O(n)$ a pamäťová zložitosť tohto riešenia je $O(1)$, ako aj každého iného tu prezentovaného riešenia, kedže nezávisle na vstupe urobí každé riešenie iba konštantný počet krokov (najviac $365 \cdot konstanta$ pri najhoršom a najmenej $12 \cdot konstanta$ pri najlepšom). Takisto každú hodnotu môžeme spracovať hneď po načítaní, nemusíme si nikde pamätať všetkých n hodnôt.

3. Zoraď všetky slová, vypíš n-té

vzorák napísal Rasťo
(max. 7 b za popis, 3 b za program)

Začneme tým, že si spočítame akú dĺžku má n-té slovo. Slov dĺžky 1 je 26. Slov dĺžky 2 je $26 \cdot 26$, lebo na prvej pozícii môže byť 26 rôznych písmen a na druhej môže byť tiež 26 písmen. Vo všeobecnosti, slov dĺžky k je 26^k. Takže v cykle budeme postupne odčítavať počty slov dĺžky i od n kým bude platiť, že $n > 0$. Ak totiž platí $n - 26^i > 0$[1], tak sa hľadané slovo nachádza v postupnosti až za všetkými slovami dĺžky i, čiže musí byť nutne dlhšie ako i. Ak $n - 26^i \leq 0$, tak to znamená, že hľadané slovo má dĺžku i.

V ďalšej fáze zistíme, z akých písmen sa toto slovo skladá. Nech je naše slovo dĺžky d. Potom všetkých slov dĺžky $d-1$ je 26^{d-1}. Teda slov dĺžky d začínajúcich na písmeno `A` je 26^{d-1}. Zasa by sme mohli v cykle odčítavať počty slov začínajúcich na písmeno `A`, potom na písmeno `B`, písmeno `C` atď. Myšlienka je rovnaká ako pri hľadaní dĺžky.

[1] Uvedomte si, že hodnota n v tejto nerovnici vyjadruje pôvodnú hodnotu n, od ktorej sme už odčítali všetky menšie mocniny 26.

Všimnite si, že zakaždým odčítavame to isté číslo číslo, takže rozumnejšie bude použiť celočíselné delenie. Teda $n/26^{d-1}$ hovorí, koľkokrát môžeme odčítať 26^{d-1} od n aby sme neklesli pod 0 a zvyšok po delení je to čo nám z n ostane po odčítavaní. Takýmto spôsobom zistíme každé písmeno slova. (V zdrojovom kóde, ktorý sa nachádza nižšie sme využili to, že hodnoty 26^{d-1} sme si predpočítali dopredu v predchádzajúcom cykle.)

Pamäťová zložitosť je zjavne $O(1)$, lebo sme na zapamätávanie použili len pár premenných.

Pozrime sa na časovú zložitosť tohto algoritmu. Nech dĺžka slova je d. Potom aj prvý, aj druhý cyklus urobí zhruba d operácii. Poďme spočítať aké veľké je číslo d. Z predchádzajúceho algoritmu pre d a n platí:

$$n \leq 26^0 + 26^1 + \cdots + 26^d$$

Toto nie je nič iné ako súčet členov geometrickej postupnosti na čo použijeme vzorec[2]:

$$n \leq \frac{26^{d+1} - 1}{25}$$

To znamená, že n je zhruba tak veľké ako hodnota 26^d (Zanedbali sme nejaké konštanty, ktoré sa stratia v O-notácii.). Hodnotu d preto môžeme slovne popísať ako: číslo, na ktoré keď umocním číslo 26, dostanem hodnotu n.

No a na vyjadrenie takýchto hodnôt používajú matematici pojem logaritmus. Zapisuje sa to ako $\log_a b = c$, číta sa to ako logaritmus z b pri základe a sa rovná c a vyjadruje to, že ak umocním hodnotu a na číslo c dostanem číslo b ($a^c = b$). Časovú zložitosť preto môžeme vyjadriť ako $O(\log_{26} n) = O(\log n)$. Keďže tento výpočet urobíme pre každý riadok vstupu, tak celková zložitosť je $O(t \log n)$.

Opäť si odporúčam pogoogliť niečo o logaritmoch, je to v informatike veľmi používaný pojem a oplatí sa poznať vzorce na ich úpravu a vedieť, čo zhruba znamenajú.

Listing programu (C++)

```
#include <cstdio>

void solve()
{
        long long N;
        scanf("%lld",&N); N++;
        long long words = 1;
        //zistime dlzku
        while (N - words > 0)
        {
                N -= words;
                words *= 26;
        }
        words /= 26;//2^d
        N--;
        while (words)
        {
                putchar( N / words + 'A');
                N %= words;
                words /= 26;
        }
        putchar('\n');
}

int main()
{
        int T;
        scanf("%d",&T);
        for (int i=0;i<T;++i) solve();
        return 0;
}
```

vzorák napísal Jaro
(max. 10 b za popis, 5 b za program)

4. Och, tie darčeky!

Riešenie tejto úlohy pozostávalo z viacerých krokov. Najprv sa teda pozrieme, aké riešenie vám stačilo na polovicu bodov a nakoniec sa pozrieme na celý problém.

Zjednodušená úloha

Ľahšou úlohou bolo vyriešiť problém, pokiaľ pre každú pozíciu vieme bez akýchkoľvek výpočtov povedať, s ktorými pozíciami ju vieme vymeniť. Celkom zjavne najmenšia permutácia vznikne tak, že na jej prvé miesto dáme najmenšie možné číslo. Podobne na každú ďalšiu pozíciu chceme dať najmenšie možné číslo, pričom niektorým číslam sme už priradili ich pozíciu.

[2] Odporúčam vygoogliť.

To nám dáva priamočiary postup, ako našu permutáciu vygenerovať. Postupne spracujeme pozície zľava doprava. Pre každú pozrieme všetky pozície od nej ďalej, z nich vyberieme pozíciu s najmenším číslom vymeníme čísla na týchto dvoch pozíciach. Takto dostaneme riešenie úlohy v časovej zložitosti $O(n^2)$. Ak navyše riadky matice zo vstupu spracujeme priebežne, bude nám stačiť lineárna pamäťová zložitosť.

Teraz sa na to pozrieme z inej strany. Rozoberme si pozície na skupiny tak, že pozície v jednej skupine sú vymeniteľné každá s každou, ale s nijakou mimo tejto skupiny. Keď teraz začneme vypĺňať našu výslednú permutáciu od prvej pozície, čo môžeme spraviť, je pozrieť sa, do ktorej skupiny daná pozícia patrí a zo všetkých nepoužitých čísel tejto skupiny tam presunúť to najmenšie. Nakoniec si už len všimneme, že v riešení sme čísla vrámci jednej skupiny vlastne usporiadali podobným spôsobom, ako funguje minsort. Neskôr to využijeme.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;

int main () {
    int n, x;
    scanf("%d", &n);
    vector <int> permutacia(n);
    vector <bool> riadok(n);

    //nacitame povodnu permutaciu
    for (int i = 0; i < n; i++) {
        scanf("%d", &permutacia [i]);
    }

    //spracujeme po poziciach, zlava doprava
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {

            //nacitame, s ktorymi poziciami sa i-ta moze vymenit
            scanf("%1d", &x);
            riadok [j] = (x == 1);
        }

        //zistime, ake najmensie cislo vieme dostat na i-tu poziciu
        //z este nepouzitych
        int najmensia_pozicia = i;
        for (int j = i+1; j < n; j++) {
            if (riadok [j] && permutacia [najmensia_pozicia] > permutacia [j])
                najmensia_pozicia = j;
        }

        //a vymenime
        swap(permutacia [i], permutacia [najmensia_pozicia]);
    }

    //nakoniec vsetko vypiseme a dame si pozor na medzeru
    //na konci riadka
    for (int i = 0; i < n-1; i++) printf("%d␣", permutacia [i]);
    printf("%d\n", permutacia [n-1]);
}
```

Grafy

Tu si rozoberieme, čo budeme rozumieť pod pojmom graf, cesta v grafe a komponent grafu. Ďalej sa pozrieme na prehľadávanie grafu, ktorým budeme hľadať komponenty. Ak sú vám tieto pojmy známe, odporúčam preskočiť túto podkapitolu.

Graf je štruktúra pozostávajúca z vrcholov a hrán, kde hrany spájajú dvojicu vrcholov. Prirovnať si to môžeme k nejakej cestnej sieti, kde vrcholmi budú mestá pospájané asfaltkami.

Postupnosť vrcholov, kde každé dva po sebe idúce vrcholy sú spojené hranou, začínajúcu vrcholom a a končiacu vrcholom b, budeme volať cesta medzi a a b. Množinu vrcholov, pre ktorú existuje cesta medzi ľubovoľnými dvoma vrcholmi tejto množiny, ale neexistuje žiadna cesta medzi vrcholom z tejto množiny a vrcholom mimo tejto množiny, budeme nazývať komponent grafu.

Ešte si popíšeme, ako zistiť, aké vrcholy ležia v tom istom komponente, ako nejaký zadaný vrchol grafu. Použijeme rekurzívnu funkciu, ktorá ako argument dostane vrchol, ktorý má spracovať a rekurzívne sa zavolá do všetkých ešte neprehľadaných vrcholov, ktoré s ním susedia. K zisteniu, či nejaký vrchol ešte nebol prehľadaný, bude naša funkcia používať jedno globálne pole.

Permutácia ako graf

Tu sa najprv poriadne popozeráme, ako vlastne môžeme narábať s permutáciou v nezjednodušenej verzii úlohy. K tomu sa na ňu pozrieme ešte z iného uhla, predstavíme si to celé ako graf, ktorého vrcholmi budú pozície v našej permutácii a medzi dvoma pozíciami bude hrana, pokiaľ môžeme ich čísla priamo vymeniť.

Keď sa pozrieme na komponenty takéhoto grafu, zistíme, že vhodnou postupnosťou výmen sme schopní vytvoriť ľubovoľné preusporiadanie čísel v jednom komponente (a celkom zjavne do neho nedokážeme dostať číslo z iného komponentu). Ukážeme si, ako vymeniť ľubovoľné dve čísla v tomto komponente. Pomocou dostatočného počtu takýchto výmen by sme už dokázali bez problémov ľubovoľné rozloženie čísel naozaj zmeniť na ľubovoľné iné.

Vezmime si nejakú cestu medzi pozíciami, ktoré chceme vymeniť. Ak vymeníme čísla na prvej a druhej, potom na druhej a tretej a tak ďalej, až na predposlednej a poslednej pozícii z tejto cesty, docielime, že prvý prvok bude na konci, tam kde ho chceme mať a posledný bude na predposlednej pozícii. Potom môžeme podobne presunúť číslo z predposlednej pozície (čiže to, ktoré bolo pôvodne na konci) na prvú pozíciu, čím naozaj ostanú všetky okrem prvého a posledného čísla na svojej pozícii a prvé a posledné budú vymenené[3].

Tu už len využijeme riešenie ľahšej podúlohy. Stačí nám prehľadaním zistiť, ktoré pozície sú v tom istom komponente a potom v každom komponente usporiadať čísla od najmenšieho po najväčšie, k čomu môžeme použiť ľubovoľný nanajvýš kvadratický algoritmus.

Celé to bude fungovať s časovou zložitosťou $O(n^2)$ a rovnakou pamäťovou zložitosťou.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

vector <int> permutacia, visited, values;
vector <vector <bool> > maru;
int n;

//Vsetkym poziciam v jednom komponente priradime v poli
//visited hodnotu 2. Do pola values nahadzeme hodnoty
//z tychto pozicii.
void dfs (int v) {
    values.push_back(permutacia [v]);
    visited [v] = 2;
    for (int i = 0; i < n; i++) {
        if (visited [i] == 0 && maru [v] [i]) {
            dfs(i);
        }
    }
}

int main () {
    int x;
    scanf("%d", &n);
    permutacia.resize(n);
    visited.resize(n, 0);
    maru.resize(n, vector <bool> (n));

    //nacitame permutaciu
    for (int i = 0; i < n; i++) {
        scanf("%d", &permutacia [i]);
    }

    //nacitame maticu vymenitelnosti
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%1d", &x);
            maru [i] [j] = (x == 1);
        }
    }

    //pre kazdu poziciu zistime, ci sme ju uz nahodou
    //nespracovali a ak nie, spracujeme ju
    for (int i = 0; i < n; i++) {
        if (visited [i] == 0) {

            //zistime, co je s nou v komponente
            dfs(i);

            //usporiadame hodnoty
            sort (values.begin(), values.end());

            //priradime ich v poradi na prazdne pozicie
            //a danym poziciam nastavime hodnoty v poli
            //visited na 1
            int point = 0;
            for (int j = 0; j < n; j++) {
                if (visited [j] == 2) {
                    visited [j] = 1;
                    permutacia [j] = values [point];
                    point ++;
                }
            }
            values.clear();
        }
    }
```

[3] Nakreslite si to!

```
    //vypiseme
    printf("%d", permutacia [0]);
    for (int i = 1; i < n; i++) printf("␣%d", permutacia [i]);
    printf("\n");
}
```

Existuje ešte aj riešenie s lineárnou pamäťovou zložitosťou, ktoré sme po vás ale nevyžadovali. Mohli ste zaň dostať jeden bonusový bod. Základnou myšlienkou je vhodne si ukladať informáciu o komponentoch a informáciu o vymeniteľnosti pozícií spracovávať priebežne po riadkoch.

5. Zlovestný svet

vzorák napísal Vlejd
(max. 0 b za popis, 15 b za program)

Na začiatok

Toto vzorové riešenie nebude také, na aké ste možno zvyknutí. Dôvod je, že táto úloha vzorové riešenie ako také *nemá* a nemôže mať. Samozrejme, určite existuje zvieratko, ktoré má najväčšiu odolnosť, ale všeobecný, zaručený a rýchly spôsob jeho hľadania neexistuje. Prečo? Lebo inak by sme vedeli jednoducho riešiť ľubovoľný informatický problém.

Celá táto úloha bol taký experiment s vami. Úloha skúmala, ako sa dokážete vysporiadať s neznámym problémom. Možných prístupov je mnoho a do vzorového riešenia sa všetky nezmestia. Ukážeme si teda niekoľko techník a niekoľko prístupov k úlohe.

Lacný bod

Napíšeme program, ktorý skúsi zvieratká, ktoré ponúka ukážkový vstup. Pár riadkov kódu a máme dva body ani nevieme ako. Ale aspoň sme sa presvedčili, že rozumieme ako má fungovať vypisovanie výstupu.

Náhodné tipovanie

Čo tak predchádzajúci prístup zlepšiť, a pýtať sa aj na ďalšie možné zvieratká. Budeme veselo generovať náhodné stringy a posielať ich do sveta. A máme tri body, s trochou šťastia štyri.

Lozenie hore kopcom

Ako rozprávka napovedá, ideme si založiť po kopcoch a použijeme metódu zvanú hill-climbing. Predstavme si, že sme v neznámom teréne uprostred tmy a chceme vyliezť na kopec. Môžeme spraviť jeden krok náhodným smerom a overiť, či sme takto išli hore kopcom. Ak sa nám nepodarilo ísť hore kopcom, tak sa vrátime a skúsime ísť iným smerom. Opakovaním tohto postupu sa dostaneme na vrchol nejakého kopca, odkiaľ už nebudeme môcť ísť vyššie.

Problém tohto postupu je, že keby sme to skúšali v Tatrách, tak pravdepodobne ostaneme zaseknutí na nejakom veľkom kameni v údolí a na Gerlachovský štít sa zrejme nedostaneme. Tento postup totiž nájde len takzvané lokálne maximum (bod, ktorý je vyšší od všetkých okolitých) a nie globálne maximum (najvyšší bod celkovo). Môžeme však skúsiť šťastie a začať naše putovanie z 200 náhodných miest a z každého spraviť 500 krokov smerom hore.

V reči DNA začneme vždy z nejakého náhodného DNA reťazca a budeme opakovane meniť jedno náhodné písmeno na iné. Zistíme, či vtákopysk so zmeneným jedným písmenom vydržal dlhšie a ak áno, necháme si ho. Inak zmenu zahodíme a skúsime spraviť inú náhodnú zmenu. Môžme tiež skúsiť aj iné typy zmien, napríklad vymieňať dve písmenka, meniť písmenko na náhodnej pozícií, kopírovať náhodné kusy zvieratka na iné miesto, čo nás len napadne. Vo všeobecnosti sa ale odporúča skúsiť naraz viac podobných zmien a potom vybrať tú najlepšiu. Ak sa napríklad rozhodneme, že budeme meniť druhý gén, tak skúsime všetky možnosti a tú najlepšiu si necháme.

V závislosti od konkrétnej implementácie získame pravdepodobne 5 až 8 bodov.

Takmer simulované žíhanie

Toto je postup podobný lozeniu do kopca. Líši sa v tom, že dovolíme robiť aj zmeny k horšiemu. Navyše v žíhaní vystupuje aj parameter *teplota*, ktorý ovplyvňuje správanie algoritmu.

V každom kroku zoberieme aktuálny reťazec a spravíme v ňom nejaké drobné zmeny (počet zmien zvykne závisieť od teploty). Následne zistíme, či nový reťazec má lepšie skóre (t.j. odolnosť zodpovedajúceho vtákopyska) ako predošlý a ak áno, necháme si nový reťazec. Ak nie, tak sa náhodne rozhodneme, či si necháme nový alebo starý reťazec. Pravdepodobnosť toho, že si aj pri neúspechu necháme nový reťazec, závisí od teploty.

Potom existuje viacero prístupov ako meniť teplotu, môžeme začať s vysokou a vždy, keď sa nám podarí spraviť dobrú zmenu, tak ju trocha znížime (prenásobíme konštantou trochu menšou ako 1). Alebo môžeme dokola striedať 4 hodnoty teploty. Fantázii sa medze nekladú.

Týmto spôsobom by ste pravdepodobne dostali 10 až 11 bodov, a ak správne odladíte konštanty, tak aj 12.

Labák

Ak chceme ísť ďalej, môžeme sa inšpirovať prírodou a skúsiť takzvané genetické algoritmy. Idea je nasledovná. Budeme sa tváriť, že naše zvieratká sú naozajstné živočíchy, že si skutočne spolu nažívajú v skupine zvanej populácia a že ich DNA reťazce sú naše stringy. Na začiatku si vyrobíme náhodne niekoľko zvieratiek a zistíme si ich odolnosť (pošleme ich do sveta). Tieto zvieratká nám budú tvoriť prvú generáciu našej populácie.

Tak ako bežne v prírode, zvieratká sa budú medzi sebou krížiť a ich DNA bude podliehať náhodným mutáciam (zmenám). Aby sme celý proces urýchlili (a tým zlepšili) budú mutácie oveľa pravdepodobnejšie ako naozaj v prírode. Tiež tak ako bežne v prírode, prežijú len najsilnejší. Každé zvieratko má nejakú pravdepodobnosť, že sa dožije ďalšej generácie a tá pravdepodobnosť by mala byť tým vyššia, čím odolnejšie je zvieratko.

Novú generáciu z predošlej vyrobíme nasledovne: Zahodíme náhodne vybraté zvieratká s nízkou odolnosťou (asi tretinu). Nakopírujeme najodolnejších pár zvieratiek. Náhodným zvieratkám spravíme náhodné mutácie. Nové náhodné zvieratká vytvoríme kombináciou predošlých. Spravíme ďalšie veci čo nám napadnú, napríklad môžeme pridať niekoľko nových úplne náhodných zvieratiek.

V celom postupe sa skrýva veľké množstvo magických konštánt, od veľkosti populácie po jednotlivé pravdepodobnosti. Tiež sa treba rozhodnúť, aké mutácie budeme robiť (tu sa dajú robiť rovnaké ako v hill-climbingu, náhodná zmena písmen, vymieňanie písmen atď.), a ako sa budú zvieratá krížiť. Často kríženie prebieha tak, že zoberieme dve (kľudne ale aj viac) zvieratiek a nejako ich spojíme dokopy. Napríklad začiatok reťazca zoberieme z prvého zvieratka a koniec z druhého. Tiež môžeme občas nechať nejaké zvieratko vyvinúť, napríklad tak, že ho dáme ako počiatočné pre hill-climbing a do ďalšej generácie pošleme už vyoptimalizované zvieratko.

Je tu obrovský priestor na experimentovanie a veľmi veľa možností, čo robiť. S hala-bala neporiadnym nastavením konštánt sa dalo získať 10 bodov.

Vedúcovské riešenie bolo tiež genetické programovanie s trocha lepšie nastavenými konštantami (aj keď určite sa dali ešte zlepšiť). Väčšinou dostalo 12 až 14 bodov.

Drobné optimalizácie na zváženie

Pokiaľ sme použili v algoritme nejakú náhodu, alebo aj keď nie, mohlo sa stať, že sme do sveta posielali to isté zvieratko dva krát. To je ale plytvanie drahocennými pokusmi. Čo môžeme spraviť je, že si každé zvieratko, na ktoré sme sa už spýtali, uložíme do mapy aj s jeho odolnosťou (a.k.a. memoizácia). Potom vždy, keď budeme chcieť otestovať nejaké zvieratko, tak sa najprv pozrieme, či už nie je v mape.

Pokiaľ by sme mali pamäťové problémy (zvieratiek je predsa len celkom dosť), môžeme si zvieratká trochu úspornejšie zakódovať, a to nie do stringov, ale do dvojíc longov (64 bitový integer). Stačí si uvedomiť, že na zakódovanie jedného génu nám stačia dva bajty. Potom vieme jedno zvieratko uložiť ako 100 bitov, čo sú dva longy. Toto prináša isté implementačné problémy a námahu, ale v pamäťovo rozšafnejších jazykoch na to treba myslieť.

Vedúcovský program takéto optimalizácie nerobil.

Tiež sme rátali s tým, že svoje riešenie odovzdáte viac krát a necháte tam to najlepšie. (Napríklad počty bodov za submity toho istého programu mohli vyzerať 13,12,14,11,12,13,14,12,14. V zadaní sme však písali aby ste to nepreháňali a odovzdať viac ako 100 programov nie je úplne fér.)

Krutý svet príkorí

Určite ste všetci zvedaví, ako vlastne fungoval svet – ako sa hodnotila odolnosť vtákopyskov.

Najprv sa z DNA reťazca dlhého 50 znakov vyrobil binárny reťazec dlhý 100 znakov a to tak, že každé písmeno si zmeníme na dva bity ($A = 00$, $C = 01$, $G = 11$, $T = 10$) a prvých 50 bitov binárneho reťazca bude tvorených prvými bitmi písmen z DNA a druhých 50 bitov sú druhé bity z písnen z DNA. [4]

Keď máme binárny reťazec, tak vypočítame takzvaný počet rôznych štvorcov v ňom. Teda spočítame koľko existuje reťazcov w takých, že ww sa nachádza v binárnom reťazci. Napríklad 101011001100 má 6 rôznych štvorcov a vyhovujúce w sú $0, 1, 10, 01, 0110, 1100$. Počítanie štvorcov sa robí pomocou hashovania, aby bolo rýchlejšie, a teda je tam nejaká, veľmi malá šanca, že sa pomýlime. (Napr. vtákopysk mal šťastný život a prežil o deň dlhšie ako by mal.)

[4] Teda ešte pred tým sa celý vstup zrotoval nejakou náhodnou cyklickou rotáciou, ktorá bola fixná počas celého behu programu. Toto tam bolo na to, aby sa funkcia trocha líšila pri rôznych behoch a aby sa nedali jednoducho využívať informácie z predošlých submitov.

Napokon, ešte penalizujeme reťazce, ktoré obsahujú dlhé úseky rovnakých bitov, lebo tie sú príliš nudné. Mohli ste mať však 20 rovnakých bitov za sebou beztrestne.

Áno, hodnotiaca funkcia bola pomerne komplikovaná, ale našťastie ste sa ňou nemuseli trápiť :). Keď ju poznáme, vieme spraviť o trocha lepšie riešenie, napríklad vedúcim sa podarilo nájsť zvieratko, ktoré prežilo 83 dní. Avšak nevieme, či je to najlepšie riešenie, či neexistuje nejaký lepší vtákopysk. Totiž túto úlohu, pre dané n nájsť binárny reťazec dĺžky n s najväčším počtom štvorcov nevieme efektívne riešiť. Na riešenie úloh, ktoré nevieme efektívne riešiť, často používame také heuristické algoritmy[5] ako sme si spomínali vyššie. Ak spravíme dobrý heuristický algoritmus, tak na malých vstupoch dáva optimálny výsledok oveľa rýchlejšie ako exaktný algoritmus a na veľkých vstupoch, kde by exaktný algoritmus bežal milióny rokov, dáva celkom dobré výsledky pomerne rýchlo.

Algoritmus, ktorý bodoval vaše riešenia vyzeral takto:

Listing programu (C++)

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
#define For(i, n) for(int i = 0; i<int(n); ++i)

#define QUERYCNT 100000
#define DNALENGTH 50
#define PRIME 47

string line, rotline;
string alph = "ACGT";
int H[2*DNALENGTH+17];
int powers[2*DNALENGTH+17];
int map[256];
vector<int> V;

int zivot(const string &str) {
    // spocitaj HASH a najdlhsi usek rovnakych bitov
    int mono = 0, maxmono = 0;
    H[0] = 0;
    For(i, DNALENGTH*2) {
        H[i+1] = H[i]*PRIME + str[i];
        mono = (i>0 && str[i]==str[i-1])?mono+1:0;
        maxmono = max(maxmono, mono);
    }
    maxmono /= 20;

    // najdi vsetky rozne vyskyty ww
    int h;
    int pocet = 0;
    for(int d = 1; 2*d<=2*DNALENGTH; ++d) {
        V.clear();
        for(int i = 0; i+2*d<=2*DNALENGTH; ++i)
            if ((h = H[i+d] - H[i]*powers[d]) == (H[i+2*d] - H[i+d]*powers[d])) {
                V.push_back(h);
            }
        sort(V.begin(), V.end());
        For(i, V.size()) if (i==0 || V[i]!=V[i-1]) pocet++;
    }

    return (pocet-maxmono)/(maxmono+1);
}

void zle(const string &message) {
    cout << message << endl;
    cerr << "Skore 0" << endl;
    cerr << "Chyba " << message << endl;
    exit(1);
}

int skore(int s) {
    if (s >= 80) return 15;
    if (s >= 78) return 14;
    if (s >= 76) return 13;
    if (s >= 73) return 12;
    if (s >= 70) return 11;

    if (s >= 67) return 10;
    if (s >= 65) return 9;
    if (s >= 63) return 8;
    if (s >= 61) return 7;
    if (s >= 58) return 6;

    if (s >= 53) return 5;
    if (s >= 47) return 4;
    if (s >= 36) return 3;
    if (s >= 21) return 2;
    if (s >= 10) return 1;
    return 0;
}

int main() {
```

[5] Teda nemáme žiadnu záruku, že dajú správnu odpoveď

```
    int rot = rand()%DNALENGTH;
    For(i, 256) map[i] = -1;
    For(i, alph.size()) map[alph[i]] = i;
    powers[0] = 1;
    For(i, DNALENGTH+16) powers[i+1] = powers[i]*PRIME;
    int maxzivot = 0;
    rotline.resize(2*DNALENGTH);

    For(q, QUERYCNT) {
        if (! (cin >> line)) zle("Nenasiel som DNA na vstupe.");
        if (line.size() == 1 && line == "K") break;
        if (line.size() != DNALENGTH) zle("Zla dlzka riadku!");
        For(i, line.size()) if (map[line[i]]<0) zle("Zly format riadku! Ocakavam len znaky ACGT");

        For(i, DNALENGTH) {
            // zmen vstup na retazec bitov
            int p = (i+rot)%DNALENGTH;
            rotline[i] = map[line[p]]/2;
            rotline[DNALENGTH+i] = map[line[p]]%2;
        }
        int z = zivot(rotline);
        maxzivot = max(maxzivot, z);
        cout << z << endl;
    }
    cerr << "Zivot " << maxzivot << endl;
    cerr << "Skore " << skore(maxzivot) << endl;
}
```

Záver

Genetické programovanie, simulované žíhanie a hill-climbing sú príklady heuristických algoritmov, ktoré sa pomerne ľahko programujú a dávajú celkom dobré výsledky.

Konkrétne v tejto úlohe má podľa nás najväčšiu nádej na úspech práve genetické programovanie, ktoré (ak ste tomu venovali dosť času) vám umožnilo získať 14 bodov. Ak by ste boli veľkí šťastlivci alebo by ste sa s úlohou naozaj dlho hrali, bola nádej získať 15 bodov. Nikomu sa to však nepodarilo.

vzorák napísal Mário

6. Obedové menu: ryža

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

Najprv si povieme, prečo je výhodné rozdeliť čísla zo vstupu na reťaze. Ďalej sa dozviete, ako spočítať celkový výsledok z čiastkových výsledkov pre reťaze a potom ukážeme rôzne spôsoby ako rozdeliť vstup na reťaze a ako vypočítať najlepší výber prvkov a počet týchto výberov pre jednu reťaz.

Stačí sa pozerať na 60 čísel z 1 000 000

Podľa zadania je potrebné z n čísel odstrániť čo najmenej tak, aby nezostali žiadne 2 čísla $x, 2x$. Prvé pozorovanie, ktoré urobíme je, že každé číslo x sa vylučuje s najviac dvoma inými číslami: $\frac{x}{2}, 2x$. Ďalej sa $2x$ vylučuje s x a $4x$, $4x$ s $2x$ a $8x$ atď.

$x, 2x, 4x, ..., 2^k x$ sa navzájom ovplyvňujú. Ostatné čísla, ku ktorým sa nevieme dostať z x násobením a delením dvomi, môžeme pri riešení 2^k-násobkov x úplne ignorovať.

- Každé prirodzené číslo vieme jednoznačne zapísať ako $x = 2^u \cdot z$, kde z je nepárne číslo, ktoré z čísla x získame tak, že ho delíme 2 kým sa dá, teda u-krát. Pokiaľ majú dve čísla rôzne z, určite sa nebudú ovplyvňovať.
- Ak máme vo vstupe čísla $x, 2x, 8x$ ale nie $4x$, zjavne ani $8x$ nebude nijak závisieť od toho, či $x, 2x$ odstránime alebo necháme.

Vstup sa nám teda oplatí porozdeľovať na **reťaze** čísel, ktoré spolu súvisia. Podľa predošlých dvoch argumentov budú dve čísla v jednej reťazi, ak vieme **jedno z nich prerobiť na druhé opakovaným násobením** 2 **a každý medzivýsledok je vo vstupnej postupnosti**.

Keďže čísla na vstupe sú najviac $10^{18} \approx 2^{60}$, reťaze budú mať dĺžku najviac 60 a budú navzájom nezávislé. Ak teda zistíme najlepší výber čísel a počet týchto výberov pre každú reťaz zvlášť, budeme vedieť vypočítať aj celkové výsledky.

Ako spočítať celkové výsledky z čiastkových

Vstup sme si porozdeľovali na reťaze a predpokladajme, že vieme pre každú reťaz zistiť, aký najväčší **počet čísel** v nej môže zostať[6] – označíme $C(retaz)$. **Počet rôznych možností** ako poodstraňovať čísla z jednej reťaze si označíme ako $M(retaz)$.

[6] Ako to spočítať si ukážeme neskôr.

Keďže každé číslo sa nachádza v práve jednej reťazi, najväčší počet čísel, ktoré vieme zachovať spočítame jednoducho ako súčet $C(r)$ pre každú reťaz.

Keďže reťaze sa nijak neovplyvňujú, pre každú z nich môžeme zvoliť ľubovoľný spôsob výberu čísel, a vďaka tomu bude celkový výsledok súčin $M(r)$ pre všetky reťaze.

Ako porozdeľovať čísla do reťazí

Po rozdelení čísel do reťazí už nebude záležať na hodnote čísel, iba na ich poradí v reťazi a na počtoch rovnakých čísel. Reťaz $3, 3, 3, 6, 6, 12, 24, 48, 48$ si preto zapamätáme len ako postupnosť počtov čísel s rovnakými mocninami dvojky (podľa hodnoty u v zápise $x = 2^u \cdot z$), t.j. ako $3, 2, 1, 1, 2$.

Na vytvorenie reťaze potrebujeme rýchlo zisťovať, **či je vo vstupnej postupnosti číslo x, ak áno, koľkokrát**. Pri tvorení reťaze si vyberieme začiatočné číslo a pridáveme dvojnásobky. Ak sú také čísla vo vstupnej postupnosti, pridáme ich do reťaze, ak nie sú, ukončíme reťaz a pokračujeme s ďalšou.

Rozdeľovanie do reťazí – Binárne vyhľadávanie

Vstup je utriedená postupnosť a preto v nej vieme binárne vyhľadávať. Ak chceme zistiť koľkokrát sa nachádza x v postupnosti, stačí binárne vyhľadať najmenšiu pozíciu čísla x a pozíciu najmenšieho väčšieho čísla. V C++ presne tieto úlohy plnia funkcie `lower_bound, upper_bound`. Pokiaľ sa číslo x nachádza v poli, počet jeho výskytov je jednoducho `upper_bound(x) - lower_bound(x)`.[7] V najhoršom prípade – ak sú všetky čísla vstupu rôzne – musíme každé z nich binárne vyhľadať, teda časová zložitosť takéhoto delenia na reťaze bude $O(n \log n)$.

Listing programu (C++)

```
vector<int> patri_retazi(n,-1);
vector<vector<int> > retaze;

for(int i=0; i<n; i++){
  if (patri_retazi[i] != -1) continue;                    // ak uz sme i spracovali, pokracujeme s i+1
  ll cur = v[i];
  retaze.push_back(vector<int>(0));                       // vytvorime novu, zatial prazdnu, retaz
  int terajsia_retaz = int(retaze.size())-1;
  while(1){
    int lb = lower_bound(v.begin(), v.end(), cur) - v.begin();
    if(lb == int(v.size()) || v[lb] != cur) break;        // ak 2-nasobok uz vo vstupe nie je, ukoncime retaz
    int ub = upper_bound(v.begin(), v.end(), cur) - v.begin();
    retaze[terajsia_retaz].push_back(ub-lb);
    for(int j=lb; j<ub; j++) patri_retazi[j] = terajsia_retaz; // oznacime cisla pridane do retaze ako spracovane
    cur *= 2LL;
  }
}
```

Rozdeľovanie do reťazí – Fronta / dvaja bežci

Vďaka utriedenému vstupu sa čísla dajú rozdeliť do reťazí aj v lineárnom čase.

Predstavme si, že čísla načítavame postupne tak, ako sú na vstupe a vkladáme ich do fronty – queue. Načítali sme číslo x a pozrieme sa, čo s ním môžeme urobiť:

- ak je x rovnaké ako predošlé číslo, vložíme ho do rovnakej reťaze
- ak je vo fronte číslo $\frac{x}{2}$ vložíme x na koniec reťaze, v ktorej je $\frac{x}{2}$
- ak vo fronte nie je $\frac{x}{2}$ vytvoríme novú reťaz so začiatkom x

Vieme rýchlo odpovedať na otázku, či je vo fronte $\frac{x}{2}$? Z fronty vieme vyberať prvky iba zo začiatku, teda z nej budeme musieť vyhadzovať všetky prvky menšie ako $\frac{x}{2}$. Nakoniec buď nájdeme $\frac{x}{2}$, alebo najmenšie väčšie číslo. Dôležité je uvedomiť si, že týmto spôsobom nikdy nevyhodíme čísla, ktoré budeme v budúcnosti hľadať. Zjavne polovice väčších čísel ako x sú tiež väčšie ako $\frac{x}{2}$, teda zostanú vo fronte.

Každé číslo zo vstupnej postupnosti najviac raz vložíme do fronty a najviac raz vyberieme, a tak spravíme len $O(n)$ operácií. Ak máme čísla načítané v poli, frontu môžeme simulovať pomocou dvoch ukazovateľov (bežcov), ktorí budú ukazovať na začiatok a koniec fronty.

Listing programu (C++)

```
vector<int> patri_retazi(n,0);
int zac = 0, dalsi = 1;
retaze.push_back(vector<int>(1,1));

while(dalsi < n){
  if(v[dalsi] == v[dalsi-1]){          // ak su prvky rovnake, zvysime posledne
```

[7] Tieto funkcie v C++ vracajú iterátory, ktoré sú adresami prvkov, nie indexami do poľa. Pokiaľ chceme zistiť index x, potrebujeme od iterátora odpočítať adresu začiatku poľa.

```
    int ret = patri_retazi[dalsi-1];  // cislo v posledne upravovanej retazi
    retaze[ ret ].back()++;
    patri_retazi[dalsi] = ret;
  }
  else{
    while(v[zac]*2 < v[dalsi]) zac++; // posuvame zaciatok - z fronty vyberame male prvky
    if(v[zac]*2 == v[dalsi]){         // ak je na zaciatku prave o polovicu mensi prvok
      int ret = patri_retazi[zac];
      patri_retazi[dalsi] = ret;
      retaze[ret].push_back(1);
    }
    else{                             // ak na zaciatku je uz vacsi, pridame novu retaz
      int ret = int(retaze.size());
      patri_retazi[dalsi] = ret;
      retaze.push_back( vector<int>(1,1) );
    }
  }
  dalsi++;
}
```

Rozdeľovanie do reťazí – Hashovacia tabuľka

Pole veľkosti 10^{18} by zaberalo niekoľko miliónov terabajtov, ale ak by boli čísla na vstupe malé ($\approx$ menšie ako 10 000 000), na riešenie by nám stačilo pole malej veľkosti (napr. 10 miliónov prvkov pre čísla menšie ako 10 miliónov). Políčku $pole[x]$ by sme pripočítali 1 za každý výskyt x vo vstupe a na požadovanú otázku – koľko x-ov je na vstupe by sme vedeli odpovedať v konštantnom čase. Reťaze by sme tak vedeli vytvoriť v čase $O(n)$.

Pre veľké čísla si presne takéto údaje môžeme ukladať a odpovedať na otáky v konštantnom čase pomocou hash mapy, ktorá si „premenuje" prvky – zahashuje – aby sa zmestili do poľa, do ktorého potom indexuje. V C++11 ju nájdete pod názvom `unordered_map`. Jej réžia si však vyžaduje konštantne viac času a pamäte oproti riešeniu s frontou/2 bežcami.

Výpočet pre jednu reťaz

V tejto časti popíšeme ako pre jednu reťaz spočítať najväčší možný počet zachovaných prvkov a počet výberov ktorými sa to dá dosiahnuť. Budeme sa sústrediť už len na nasledovnú úlohu:

Máme postupnosť čísel – počty prvkov v reťazi. Chceme vybrať niekoľko čísel tak, aby sme nevybrali žiadne dve vedľa seba a aby počet vybratých bol čo najväčší. Chceme zistiť aj počet rôznych výberov.

Výpočet pre jednu reťaz – Neopakujúce sa čísla

3 body sa dali získať aj za vyriešenie jednoduchšej úlohy – pre neopakujúce sa čísla na vstupe. V takejto verzii úlohy nám stačí poznať len dĺžky jednotlivých reťazí – v reprezentácii, ako bola spomenutá na začiatku časti o rozdeľovaní čísel do reťazí, by sme mali reťaze uložené ako postupnosti samých jednotiek.

Výber čísel z reťaze budeme značiť ako postupnosť 0/1, kde 0 bude znamenať, že sme číslo odstránili a 1, že sme ho zachovali.

- Pokiaľ je dĺžka reťaze nepárna existuje len jedna možnosť ako dosiahnuť najvačší počet zachovaných čísel a zachová sa ich $\lfloor \frac{l}{2} \rfloor + 1$, kde l je dĺžka reťaze a $\lfloor x \rfloor$ je dolná celá časť x. Výber čísel bude vyzerať nasledovne: $1010 \ldots 0101$.
- Ak je dĺžka párna, najväčší počet zachovaných je $\lfloor \frac{l}{2} \rfloor$. Koľko je ale rôznych možností výberov? Pre $l = 4$ to môže byť 0101, 1010, ale aj 1001. Pre ľubovoľnú párnu dĺžku sú 2 možnosti výberov, kedy sú na krajoch $0/1, 1/0$. Ak sú však na oboch krajoch jednotky, vnútri postupnosti sme museli odstrániť 2 čísla vedľa seba. Takýchto potenciálnych miest je vo vnútri reťaze $\lfloor \frac{l}{2} \rfloor - 1$, teda počet možností výberov je celkovo pre párne dlhú reťaz $\lfloor \frac{l}{2} \rfloor + 1$.

Výpočet pre jednu reťaz – Hrubá sila

Ak by boli všetky reťaze krátke, môžeme dovoliť použiť hrubú silu. Skonštruovali by sme všetky výbery, ako niektoré čísla vynechať – všetky podmnožiny reťaze. V každom takomto výbere by stačilo skontrolovať, či sme v ňom nenechali 2 nasledujúce čísla. Počet rôznych výberov by sme jednoducho zväčšovali o 1 za každý výber s najlepším výsledkom.

Ako prakticky prezerať všetky možné výbery? Ak si opäť označíme výber prvkov ako reťazec núl a jednotiek, potrebujeme skontrolovať všetky výbery medzi $00 \ldots 00$ a $11 \ldots 11$. Tieto binárne reťazce ale môžeme považovať aj za čísla od 0 po $2^l - 1$. Stačí nám teda v cykle prejsť cez tieto čísla a pre každé číslo – výber, skontrolovať či v ňom nie sú 2 jednotky za sebou a spočítať súčet čísel, čo zostanú v reťazi.

Ak by sme ohraničili dĺžku reťazí číslom l, časová zložitosť takéhoto riešenia by sa dala odhadnúť ako $O(n \cdot 2^l)$. Za takéto riešenie ste mohli získať až 6 bodov z praktickej časti.

Listing programu (C++)

```
int najvacsi_pocet = 0;
ll najviac_moznosti = 1LL;
ll MOD = 1000000009LL;

For(r,int(retaze.size())){
  ll moznosti = 0LL;
  int pocet = 0;
  int len = retaze[r].size();
  For(i, (1<<len)){                              // pre vsetky mozne vybery reprezentovane ako cisla
    bool ok = 1;                                 // kontrola, ci nejdu 2 jednotky po sebe
    For(j,len)
        if( (i & (1<<j)) && (i & (1<<(j+1)))) ok = 0;
    if(!ok) continue;

    int poc = 0;                                  // inak aktualizujeme najvacsi pocet cisel a pocet moznosti
    For(j,len)
      if(i & (1<<j) ) poc += retaze[r][j];        // spocitavame pocet zachovanych cisel v retazi
    if(poc > pocet){ pocet = poc; moznosti = 0LL; }
    if(poc ==pocet) moznosti++;
  }
  najvacsi_pocet += pocet;
  najviac_moznosti = (najviac_moznosti * moznosti) % MOD;
}

printf("%d %lld\n", najvacsi_pocet, najviac_moznosti);
```

Výpočet pre jednu reťaz – Dynamické programovanie

Na záver si ukážeme, ako spočítať všetko, čo chceme, optimálne – teda v lineárnom čase.

Skúsme **výber prvkov z reťaze postupne konštruovať zľava doprava**. Na začiatku nech je výber prázdny a postupne do neho budeme pridávať niektoré čísla z reťaze. Pri každom čísle budeme mať na výber dve možnosti: „pridať, či nepridať?"

Začnime prvým prvkom zľava. Ak ho nepridáme, je zjavne len jeden výber s najlepším súčtom – nulovým. Ak ho pridáme, najlepší možný súčet je prvé číslo z reťaze a počet výberov je 1.

Poďme pridať ďalšie číslo z reťaze. Ak sme pridali predošlé číslo, druhé číslo pridať nemôžeme. Ak sme predošlé číslo nepridali, môžeme sa rozhodnúť či pridáme alebo nepridáme ďalšie.

Všimnite si, že akonáhle sa rozhodneme jedno číslo nepridať do výberu, zvyšok výberu bude úplne nezávislý od všetkého naľavo. Nemusíme teda výbery konštruovať, stačí nám zapamätať si, aký je doterajší najlepší súčet a koľkými rôznymi spôsobmi sa dá dosiahnuť.

Na základe týchto úvah nájdeme najväčší súčet a počet rôznych výberov postupne pre prvých $1, 2, \ldots, i$ prvkov reťaze.

Označme si ako `S[i][0]` **najväčší súčet**, ktorý môžeme dosiahnuť výberom z prvých i prvkov, ak prvok i nepridáme. `S[i][1]` bude označovať maximálny možný súčet z prvkov $0, 1, \ldots, i$ ak i-te číslo pridáme.

Podobne označíme **počet rôznych výberov** z prvkov $0, 1, \ldots, i$ s najlepším súčtom ako `V[i][0]` a `V[i][1]` opäť podľa toho, či sme i-te číslo pridali.

Celkový výsledok pre reťaz zistíme ako najväčší súčet čísel z výberu všetkých prvkov, teda `max(S[dĺžka reťaze - 1][0], S[dĺžka reťaze - 1][1])` a počet výberov, pri ktorých dosiahneme tento súčet, sa dozvieme z príslušného `V[dĺžka reťaze - 1][0], V[dĺžka reťaze - 1][1]`.

Najlepšie súčty pre prvých i prvkov vieme určiť z informácií pre prvých $i - 1$ prvkov takto:

- Ak chceme pridať prvok i, $(i-1)$-vý sme pridať nemohli. Preto `S[i][1] = S[i-1][0] + retaz[i]`.
- Ak prvok i nepridávame, pre výber prvých i prvkov sa môžeme rozhodnúť či v ňom bude $(i-1)$-vé číslo z reťaze alebo nie. Vtedy `S[i][0] = max(S[i-1][0], S[i-1][1])`.

Počet výberov prvých i prvkov s najlepším súčtom vieme tiež zistiť len z informácií pre prvých $i - 1$ prvkov:

- Keď pridávame prvok i, počet možností výberu zostane rovnaký ako počet výberov prvých $i - 1$ prvkov, keď $(i-1)$-vý v tomto výbere nebude, teda `V[i][1] = V[i-1][0]`.
- Ak i do výberu prvých i prvkov nepridáme, môžeme sa rozhodnúť, či je lepší výber prvých $i - 1$ prvkov s alebo bez $(i-1)$-vého prvku.
  - Ak je jeden výber lepší, počet možností výberov `V[i][0]` bude rovnaký ako `V[i-1][ten lepší]`.
  - Ak sú najlepšie výbery s a bez $i-1$ rovnocenné, teda ak vieme dosiahnuť rovnaký najlepší súčet bez aj s $(i-1)$-vým prvkom, môžeme použiť všetky tieto výbery a tak výsledný počet rôznych výberov je súčtom počtov výberov `V[i][0] = V[i-1][0] + V[i-1][1]`.

Pre každý prvok reťaze vypočítame štyri čísla `S[i][0], S[i][1], V[i][0], V[i][1]`. Každý takýto výpočet je nanajvýš súčtom alebo maximom dvoch čísel, teda sa stíha v konštantnom čase a teda každú reťaz vieme spracovať v čase lineárnom v závislosti od jej dĺžky, teda $O(l)$. Spracovanie všetkých reťazí dokopy nám potrvá $O(n)$, lebo každé políčko tabuľky `S, V` zodpovedá aspoň jednému číslu zo vstupu.

Keďže vo výpočte i-teho políčka používame len údaje z $(i-1)$-vého, mierne množstvo pamäte sa dá ušetriť tak, že si nepamätáme celé pole ale len tieto posledné údaje o $(i-1)$ prvých prvkoch.

Listing programu (C++)

```
int najvacsi_pocet = 0;
ll najviac_moznosti = 1LL;
ll MOD = 1000000009LL;

For(r,int(retaze.size())){
  // trikovo pridame este 1 prvok na koniec retaze, aby sme na zaver nemuseli "vypocty dynamiky" pisat 2x
  retaze[r].pb(0);
  int l = int(retaze[r].size());
  vector< vector<int> > S(l,vector<int>(2,0));
  vector< vector<int> > V(l,vector<int>(2,0));

  S[0][0] = 0; S[0][1] = retaze[r][0];    // nastavime vysledky pre prvy prvok
  V[0][0] = 1; V[0][1] = 1;
  for(int i=1; i<l; i++){
    S[i][1] = S[i-1][0] + retaze[r][i];   // ak chceme vybrat prvok i, nas osud je predurceny
    V[i][1] = V[i-1][0];

    if(S[i-1][0] == S[i-1][1]){           // ak su vybery s a bez (i-1) rovnocenne
      S[i][0] = S[i-1][0];
      V[i][0] = V[i-1][0]+V[i-1][1];
    }
    else{                                 // ak je jeden z vyberov (i-1) lepsi
      int k = (S[i-1][0] > S[i-1][1] ? 0 : 1);
      S[i][0] = S[i-1][k];
      V[i][0] = V[i-1][k];
    }
  }
  najvacsi_pocet += S[l-1][0];
  najviac_moznosti = (najviac_moznosti * ll(V[l-1][0])) % MOD;
}

printf("%d %lld\n", najvacsi_pocet, najviac_moznosti);
```

Záver

Rozdelenie vstupu na reťaze sa dá robiť nezávisle od počítania údajov pre jednu reťaz a tak ste si mohli zvoliť rôzne kombinácie algoritmov. Celkovo sa dala úloha vyriešiť spojením prístupu s frontou s dynamickým programovaním v $O(n)$. Toto je zjavne aj najlepšia asymptotická zložitosť, ktorá sa dá dosiahnuť, keďže už len načítanie vstupu nám potrvá lineárne dlho.

vzorák napísal Žaba

7. Och, koho len mám zabiť?

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

Začnime tým, že si zopakujeme zadanie úlohy, keďže je mierne komplikované. Na začiatku sme mali permutáciu n čísel, ktorú si označíme P. Zobrali sme všetky cyklické rotácie tejto permutácie a lexikograficky sme ich usporiadali. Z každej permutácie sme postupne zobrali posledné číslo (v takom poradí ako boli usporiadané), čím sme dostali permutáciu R. Nanešťastie, zachovalo sa nám iba niekoľko začiatočných čísel permutácie R, ale vieme, že zo všetkých vhodných permutácii, bola R lexikograficky najmenšia. No a úlohou je nájsť permutáciu R.

A tu sa nachádza chyták tejto úlohy. Aj keď budeme hľadať permutáciu R, oveľa dôležitejšia bude pre nás permutácia P, hlavne fakt, že je to permutácia. Ak sme spravili cyklické permutácie P a lexikograficky ich zoradili, dostali sme tabuľku $n \times n$. Permutácia R nám tvorí posledný stĺpec tejto tabuľky. Ale my poznáme aj prvý stĺpec. Musia to byť predsa čísla od 1 po n zoradené za sebou[8].

A čo nám určuje prvý a posledný stĺpec tejto tabuľky? Hovorí to o vzájomnom vzťahu za sebou idúcich čísel v permutácii P. Ak máme v nejakom riadku na začiatku číslo x a na konci číslo y, znamená to, že v permutácii P sa číslo x nachádza hneď za číslom y. Permutáciu P teda môžeme zapísať ako graf, kde vrcholy predstavujú čísla od 1 po n a šípka z vrcholu x do vrcholu y bude znamenať, že číslo x sa v permutácii P nachádza hneď za číslom y (pričom prvé číslo permutácie P sa nachádza hneď za posledným číslom P). Takto dostaneme jeden cyklus dĺžky n.

Toto je jediná podmienka, ktorú sa budeme snažiť dodržať. Pri vytváraní R budeme postupne zisťovať závislosti, ktoré platia v P a hrany zodpovedajúce týmto závislostiam budeme pridávať do nášeho grafu. Budeme

[8] Ak robíme cyklické rotácie nejakej permutácie, pre každé možné číslo od 1 po n nám vznikne permutácia s týmto začiatkom. No a lexikografické usporiadanie potom musí triediť podľa týchto rôznych začiatkov.

sa pritom snažiť, aby sme na konci naozaj dostali jeden veľký cyklus. Hocičo iné by znamenalo, že naše R nevzniklo z **permutácie**.

Pre jednoduchosť vzoráku budeme predpokladať, že sa nezachovalo žiadne z čísel R, čiže $m = 0$. V opačnom prípade sa zmení akurát to, že niektoré závislosti budú určené vopred a môže sa nám stať, že dáta nebudú konzistentné a povedú k sporu, ktorý spôsobí, že riešenie nebude existovať. V ďalšej časti si však aj tak povieme, čo tento spor je a preto táto časť programu je takmer totožná tej druhej.

Uvedomme si, ako vyzerá náš graf, ktorý reprezentuje permutáciu P, ak sme ešte nepridali všetky závislosti. V tomto grafe môže z každého vrcholu vychádzať najviac jedna hrana a naviac tu nemôže byť žiaden cyklus. Cyklus totiž môže vzniknúť až na úplnom konci. Ľubovoľný menší cyklus by znamenal, že už nikdy sa nám neporadí spraviť jeden veľký cyklus dĺžky n. Tento graf teda musí byť tvorený niekoľkými cestami.

Naviac, vnútorné vrcholy týchto ciest nás už nezaujímajú, lebo vieme, ktorý vrchol je pred nimi aj za nimi a viac hrán do nich ísť nemôže. Jediné zaujímavé sú teda vrcholy na začiatku a konci cesty. Počas nášho algoritmu si teda musíme udržiavať tieto začiatky a konce. Jedna rozumná reprezentácia je nasledovná: pre každý koncový vrchol nejakej cesty si budem pamätať začiatok tejto cesty (toto budeme mať v poli K) a pre každý začiatočný vrchol si pamätám koniec tejto cesty (pole Z).

Predstavme si, že už sme určili (alebo boli určené) prvých $i-1$ čísiel permutácie R a máme vytvorený graf obsahujúce príslušené zistené závislosti. Pozeráme sa preto na i-ty riadok našej $n \times n$ tabuľky. Na jej začiatku je číslo i a chystáme sa určiť, aké najmenšie číslo x môžeme dať na jej koniec. V našom grafe to vytvorí hranu z vrchola i (ktorý je začiatok nejakej cesty) do vrchola x (ktorý musí byť koniec nejakej cesty[9]). Vieme teda, že číslo x musí zodpovedať vrcholu na konci nejakej cesty a kedže R sa snažíme spraviť lexikograficky najmenšie, vyberieme najmenšie možné x.

Toto má ale jeden prípad, keď to nemôžeme spraviť. A to práve vtedy, ak je x na konci cesty, kde je i na začiatku. Pridaním takejto závislosti by sme totiž vytvorili nechcený cyklus. V takom prípade ale môžeme zobrať druhé najmenšie x, keďže na konci cesty začínajúcej s i je len jedno číslo. Toto je zároveň prípad, ktorý môže spôsobiť nekonzistenciu vstupných dát.

Posledné čo zostáva je, ako rýchlo nájsť najmenšie (poprípade druhé najmenšie) číslo z koncových vrcholov. Hneď by vás malo napadnúť, že môžeme použiť dátovú štruktúru *halda*, kde vieme skladovať tieto čísla a rýchlo vybrať najmenšie z nich (a na vybratie druhého najmenšieho proste vyberieme dve najmenšie čísla a to prvé vrátime).

Následne už len správne upravíme polia Z a K. Začiatok tejto novej cesty bude vrchol $Z[x]$ a koniec bude $K[i]$. Správne úpravy budú teda $K[Z[x]] = K[i]$ a $Z[K[i]] = Z[x]$.

Zostáva nám už len odhadnúť pamäťovú a časovú zložitosť. Pamäte nám stačí $O(n)$, lebo si pamätáme len niekoľko lineárnych polí. Pri časovej musíme zarátať logaritmický faktor haldy, použijeme ju však lineárne veľa krát a preto bude časová zložitosť $O(n \log n)$.

Listing programu (C++)

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

#define For(i,n) for(int i=0; i<(n); i++)

int main() {
    int n;
    scanf("%d",&n);
    vector<int> Z,K;
    For(i,n) Z.push_back(i);
    For(i,n) K.push_back(i);
    vector<int> V;
    bool spravne = true;
    int m;
    scanf("%d",&m);
    For(i,m) {
        int x;
        scanf("%d",&x);
        x--;
        V.push_back(x);
        if((K[i]==-1 || K[i]==x) && i!=n-1) {spravne=false; continue;}
        Z[K[i]]=Z[x]; K[Z[x]]=K[i];
        K[i]=Z[x]=-1;
    }
    priority_queue<int> Q;
    For(i,n) if(Z[i]!=-1) Q.push(-i);
    for(int i=m; i<n; i++) {
        if(Q.size()==0) continue;
        int x=-Q.top(); Q.pop();
```

[9] Rozmyslite si, prečo to platí, nie je to ťažké.

```
        if(K[i]==x && Q.size()!=0) {
            int y=-Q.top(); Q.pop();
            Q.push(-x);
            x=y;
        }
        V.push_back(x);
        Z[K[i]]=Z[x]; K[Z[x]]=K[i];
        K[i]=Z[x]=-1;
    }
    if(!spravne) {printf("Chybny vstup\n"); return 0;}
    For(i,V.size()) printf("%d\n",V[i]+1);
}
```

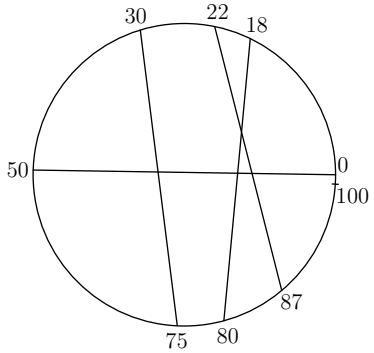
vzorák napísal mišof

8. Ostrovný maják

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

Zadanie úlohy bolo jednoduché: máme kruh a ním prechádzajúce priamky, spočítajte dvojice priamiek, ktoré sa vo vnútri kruhu križujú. Celé je to ešte zjednodušené tým, že priesečníky priamiek s kruhom sú všetky navzájom rôzne.

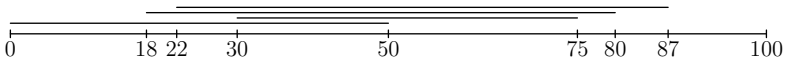
Za túto úlohu je hanba mať nulu, pretože riešenie hrubou silou – presnejšie, v čase $\Theta(n^2)$ – je veľmi jednoduché. Pre každú dvojicu priamiek vieme v konštantnom čase spočítať, či sa križujú alebo nie: Ak máme priamku spájajúcu body a a b, pričom $a < b$, a druhú priamku spájajúcu body c a d, tak sa križujú vo vnútri kruhu vtedy a len vtedy, ak práve jedno z čísel c a d leží v intervale (a, b). Rozmyslite si to pri pohľade na nasledujúci obrázok:



Veselšie to samozrejme bude akonáhle sa pokúsime o riešenie s lepšou časovou zložitosťou.

Základná myšlienka nášho riešenia pritom nebude vôbec zložitá. Mnohé problémy na kruhu sa dajú previesť na problémy na priamke, a tie sa väčšinou dajú riešiť jednoduchšie. Presne o to sa pokúsime aj v našom vzorovom riešení.

Predstavme si, že sme zobrali nožnice a prestrihli kruh medzi ľubovoľnými dvomi z bodov zadaných na vstupe – napríklad medzi bodmi s číslom 0 a r. (Na našom obrázku teda medzi bodmi 0 a 100.) Kruh teraz chytíme za oba konce a vystrieme ho na úsečku. Pre kruh z príkladu vyššie by sme takto dostali nasledovnú úsečku:



Každá z pôvodných priamok (presnejšie, tetív našej kružnice) sa teraz zmenila na nejaký interval na našej úsečke. Dôležité je uvedomiť si, že tieto intervaly naďalej nesú všetku informáciu potrebnú na vyriešenie úlohy. (Totiž nijak sme nezmenili čísla zo vstupu, len sa na ne inak pozeráme.) A navyše túto informáciu z nich vieme ľahko získať:

- Dvojica tetív zodpovedajúca disjunktným intervalom sa nepretína.
- Dvojica tetív zodpovedajúca intervalom z ktorých je jeden celý vnútri druhého sa nepretína.
- Všetky ostatné dvojice tetív sa pretínajú.

Inými slovami, stačí nám zistiť počet dvojíc intervalov, ktoré sa len čiastočne prekrývajú: teda idúc zľava doprava najskôr začne prvý interval, potom začne druhý, potom skončí prvý a až po ňom skončí druhý interval.

Tento počet dvojíc vieme ľahko určiť zametaním. Začneme tým, že si všetky začiatky a konce našich intervalov uložíme do jedného poľa a toto pole usporiadame. V tomto poradí ich teraz budeme spracúvať.

A čo sa bude diať počas tohto spracúvania? Budeme si (v nejakej šikovnej podobe, ktorú upresníme neskôr) pamätať, ktoré intervaly sú momentálne otvorené – teda množinu intervalov, ktorých začiatok sme už spracovali ale koniec ešte nie. Vždy, keď spracujeme ďalší začiatok intervalu, pribudne nám nejaký interval do tejto množiny, a vždy, keď spracujeme koniec, tak nám z nej jeden interval ubudne.

Navyše vždy, keď spracúvame koniec nejakého intervalu, započítame nejaké dvojice čiastočne sa prekrývajúcich intervalov – tie, v ktorých je interval, ktorý práve skončil, "prvý". Koľko je takých dvojíc? "Druhým" intervalom v každej takejto dvojici je určite niektorý z intervalov, ktoré sú práve otvorené. Treba si ale uvedomiť, že my nechceme úplne všetky takéto intervaly – len tie z nich, ktoré začali neskôr ako náš interval, ktorého koniec práve spracúvame.

Potrebujeme teda vedieť efektívne odpovedať na otázky nasledujúceho typu: "Koľko spomedzi intervalov ktoré sú práve otvorené, má začiatok medzi x a y?" Toto vieme spraviť veľa rôznymi spôsobmi. Napríklad môžeme použiť vyvažovaný binárny vyhľadávací strom, v ktorého vrcholoch si pamätáme začiatky aktuálne otvorených intervalov, a navyše informáciu o tom, koľko vrcholov stromu sa pod ním nachádza. Pomocou takejto dátovej štruktúry vieme ľubovoľnú otázku vyššie uvedeného typu zodpovedať v logaritmickom čase.

Existujú však aj stručnejšie možnosti implementácie. K tým nám môže pomôcť napríklad to, že si uvedomíme, že na konkrétnych súradniciach začiatkov a koncov našich intervalov vôbec nezáleží. Keď už ich raz usporiadame, riešenie sa vôbec nezmení, ak ich následne prečíslujeme na 1 až $2n$. No a udržiavať si podmnožinu množiny $\{1, \ldots, 2n\}$ je už výrazne ľahšie ako robiť to vo všeobecnosti. Môžeme na to použiť napríklad intervalový strom (ľudovo nazývaný intervaláč) alebo Fenwickov strom (u nás ľudovo nazývaný fínsky strom, pozri http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees). Ten druhý používame aj v našej nižšie uvedenej implementácii.

Všetky vyššie popísané riešenia majú zjavne časovú zložitosť $\Theta(n \log n)$ a pamäťovú $\Theta(n)$.

Listing programu (C++)

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

struct FenwickTree {
    int fsize;
    vector<int> F;

    FenwickTree(int sz) { fsize=1; while (fsize<sz) fsize <<= 1; F.clear(); F.resize(fsize+1); }

    void update(int x, int d) { while (x <= fsize){ F[x]+=d; x+=x&-x; } }

    int sum(int x1, int x2) { // sucet v zlava-otvorenom intervale (x1,x2]
        int res = 0;
        while (x2) { res += F[x2]; x2 -= x2 & -x2; }
        while (x1) { res -= F[x1]; x1 -= x1 & -x1; }
        return res;
    }
};

struct event {
    long long kde;
    int id, typ;
    event(long long kde, int id, int typ) : kde(kde), id(id), typ(typ) {}
};

bool operator< (const event &A, const event &B) { return A.kde < B.kde; }

int main() {
    int N; cin >> N;
    long long R; cin >> R;

    vector<long long> Z(N); // Z[i] je zaciatok intervalu cislo i
    vector<event> events;
    for (int n=0; n<N; ++n) {
        long long z, k; cin >> z >> k; if (z>k) swap(z,k);
        Z[n] = z;
        events.push_back( event(z,n,+1) );
        events.push_back( event(k,n,-1) );
    }
    sort( events.begin(), events.end() );

    for (int n=0; n<2*N; ++n) { // precislujeme suradnice na 1 az 2N
        events[n].kde = n+1;
        if (events[n].typ == +1) Z[ events[n].id ] = n+1;
    }

    FenwickTree F(2*N+7);
    long long answer = 0;
    for (auto e : events) {
        if (e.typ == +1) {
            F.update(e.kde,+1);
        } else {
            int my_start = Z[e.id];
            F.update(my_start,-1);
            answer += F.sum(my_start,e.kde);
        }
    }
    cout << answer << endl;
}
```

Bonus na záver

Štandardný `set` v C++ veci potrebné na riešenie tejto úlohy robiť nevie. Konkrétne, nevie nám odpovedať na otázku, koľko spomedzi v ňom uložených prvkov leží v danom intervale. Netreba však hneď implementovať vlastný strom. V novších verziách g++ kompilátora nájdeme aj takzvané "policy-based data structures" a medzi nimi aj stromy, ktoré potrebnú fičúriu majú. A s nimi je už riešenie našej úlohy hračkou.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#include <vector>
using namespace std;
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree< long long, null_type, less<long long>, rb_tree_tag, tree_order_statistics_node_update > ordered_set;

struct event {
    long long kde, zac;
    int id, typ;
    event(long long kde, long long zac, int id, int typ) : kde(kde), zac(zac), id(id), typ(typ) {}
};

int main() {
    int N; cin >> N;
    long long R; cin >> R;

    vector<event> events;
    for (int n=0; n<N; ++n) {
        long long z, k; cin >> z >> k; if (z>k) swap(z,k);
        events.push_back( event(z,z,n,+1) );
        events.push_back( event(k,z,n,-1) );
    }
    sort( events.begin(), events.end(), [](const event &A, const event &B) { return A.kde < B.kde; } );

    long long answer = 0;
    ordered_set open;
    for (auto e : events) {
        if (e.typ == +1) {
            open.insert(e.zac);
        } else {
            open.erase(e.zac);
            answer += open.order_of_key(e.kde) - open.order_of_key(e.zac);
        }
    }
    cout << answer << endl;
}
```

(Tento strom podporuje aj operáciu inverznú k `order_of_key`: metóda `find_by_order(x)` vráti iterátor na x-tý najmenší prvok, číslujúc od nuly. Obe operácie bežia v čase logaritmickom od počtu uložených prvkov.)

Bonus za záverom

Ak by niekedy došlo na najhoršie a bolo naozaj treba od základov implementovať vlastný vyvažovaný strom, dôležité je dodržať dve zásady:

- nepodľahnúť panike
- vedieť, aký strom implementovať

Hrdinom dnešných dní je treap (http://en.wikipedia.org/wiki/Treap): strom, ktorý na to, aby bol *s veľkou pravdepodobnosťou* vyvážený, používa náhodné čísla. Keď porozumiete tomu, ako treap funguje, je jeho implementácia (spravená správnym spôsobom) až prekvapivo stručná a takmer bez špeciálnych prípadov. Trik na dobrú implementáciu je nasledovný:

- jediným rekurzívnym prechodom zhora dole vieme treap rozbiť na dva menšie (`split`)
- taktiež jediným prechodom zhora dole vieme tie dva menšie spojiť späť (`merge`)
- vkladanie prvku aj výber prvku vieme triviálne realizovať pomocou `split` a `merge`
- čokoľvek ďalšie dorobíme rovnako ako by sme to robili v nevyvažovanom strome

Tu je v celej jej kráse implementácia, ktorá sa dá priamo použiť v predchádzajúcom riešení namiesto `ordered_tree`.

Listing programu (C++)

```
struct treap {
    struct treap_node {
        long long x; // prvok ktory si pamatame
        int y;       // nahodna priorita
        int size;    // velkost podstromu pod tymto vrcholom
        treap_node *l, *r;

        treap_node(long long x) : x(x), y(rand()), size(1), l(NULL), r(NULL) {}
        ~treap_node() { delete l; delete r; }

        void refresh() { size = 1 + (l ? l->size : 0) + (r ? r->size : 0); }
    };

    void split(treap_node *root, long long x, treap_node* &l, treap_node* &r) {
        // rozdeli treap na dva: lavy obsahujuci hodnoty <x a pravy obsahujuci >=x
        l = r = NULL;
        if (!root) return;
        if (root->x < x) { split(root->r, x, root->r, r); l=root; }
        else             { split(root->l, x, l, root->l); r=root; }
        root->refresh();
    }

    treap_node* merge(treap_node *l, treap_node *r) {
        // undo split
        if (!l || !r) return l ? l : r;
        treap_node *p;
        if (l->y < r->y) { p=l; p->r = merge(p->r,r); } else { p=r; p->l = merge(l,p->l); }
        p->refresh();
        return p;
    }

    treap_node *root;

    treap() : root(NULL) {}
    ~treap() { delete root; }

    void insert(long long x) {
        treap_node *l, *r;
        split(root,x,l,r);
        root = merge(merge(l,new treap_node(x)),r);
    }

    void erase(long long x) {
        treap_node *l, *m, *r;
        split(root,x,l,r);
        split(r,x+1,m,r);
        if (m) delete m;
        root = merge(l,r);
    }

    int order_of_key(long long x) {
        treap_node *l, *r;
        split(root,x,l,r);
        int answer = l ? l->size : 0;
        root = merge(l,r);
        return answer;
    }
};
```