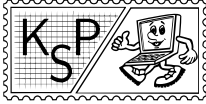


Korešpondenčný seminár z programovania
XXIX. ročník, 2011/12
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola letnej časti

1. Zavádzame plánovanie

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

Základný princíp riešenia odhalil takmer každý z Vás. Vytvoríme si N-prvkové pole booleanov, ktoré nám budú hovoriť, či obec s daným číslom ešte (možno) chceme, alebo sme ju už zamietli. Na začiatku sme ešte nezamietli žiadnu obec, preto každému prvku priradíme hodnotu `true`. Potom postupne načítame všetky zlé obce a nastavíme im hodnotu na `false`. Nakoniec prejdeme celé pole ešte raz a vyberieme 10 vhodných obcí (napríklad tie s najmenším číslom).

Akú máme časovú zložitosť? Nainicializovanie poľa nás stojí $O(N)$ operácií, zamietnutie K miest $O(K)$ a vybranie vhodných miest $O(N)$ – celkovo teda $O(N+K) = O(N)$ (nakoľko $K \leq N$). Pamäťová zložitosť je $O(N)$ – máme jedno veľké pole veľkosti N a konštantný počet iných (malých) premenných.

A teraz otázka znie: prečo takéto riešenie nedostane plný počet bodov? Ide to totiž aj trochu lepšie: nepotrebujeme vedieť rozhodovať o tom, či je *všetkých* N obcí dobrých alebo nie – bude nám stačiť túto informáciu mať pre $(K + 10)$ z nich. Keď nám z týchto obcí zakážu K, stále nám ostane 10 použiteľných. Celý algoritmus a program vyzerá rovnako ako v predchádzajúcom riešení až na to, že pole bude mať veľkosť $K + 10$ a pokiaľ nám niekto zakáže obec s číslom väčším ako $K + 10$, tak tento zákaz odignorujeme. Takýmto spôsobom dosiahneme časovú zložitosť $O(K + 10) = O(K)$ a pamäťovú zložitosť $O(K + 10) = O(K)$.

Listing programu:

```
var n,k,i,a:longint;ok:array of boolean;

begin
  read(n, k);
  SetLength(ok, k+11);
  for i:=1 to k+10 do ok[i] := true;

  for i:=1 to k do begin
    read(a);
    if a <= k+10 then ok[a] := false;
  end;

  a := 0;
  i := 1;
  while a < 10 do begin
    if ok[i] then begin
      if a <> 0 then write(' ');
      write(i);
      inc(a);
    end;
    inc(i);
  end;
  writeln;
end.
```

2. Zabudnutá hra

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

Väčšina riešení tejto úlohy, ktoré sme dostali, by sa dalo zhrnúť nasledovne. Načítame si do dvojrozmerného poľa počiatočné rozloženie mriežky, postupne načítavame jednotlivé ťahy, pokúsime sa ich vykonať a na záver vypíšeme výsledné rozloženie mriežky. Ťah sa dal vykonať

pomaly, ak by sme vždy postupne hľadali číslo, ktoré chceme posunúť. Na urýchlenie si stačí všimnúť, že namiesto hľadania tohto čísla stačí sledovať polohu medzery (číslo 0) v mriežke. Ak totiž vieme urobiť ťah, tak potom nutne číslo, s ktorým chceme hýbať, musí ležať hneď vedľa medzery. Preto stačí sa pozrieť od medzery napravo, naľavo, hore a dole a hneď vidíme, že či a ktorým smerom vieme potiahnuť. Časová zložitosť tohto riešenia je $O(n^2 + k)$ – treba načítať a vypísať mriežku, spracovanie jedného ťahu je v konštantnom čase. Keďže si potrebujeme pamätať celú mriežku, pamäťová zložitosť je $O(n^2)$. Toto je aj optimálne riešenie, ktorého implementačné detaily budeme ďalej viac rozoberať. Ukážeme si zopár trikov, ktorými sme si mohli ušetriť pár porovnaní a rozlišovaní niekoľkých prípadov.

V prvom rade si umiestníme mriežku do poľa tak, aby mala z každej strany okraj. Tento okraj inicializujeme na nejakú hodnotu mimo rozsahu (v našom prípade je to číslo -1). Na čo je to dobré? Pokiaľ sa medzera nachádza na kraji poľa, pri kontrole či sa posúvané číslo nachádza hneď vedľa nej, by sme zasahovali mimo poľa. Keďže tu ale máme náš okraj, nezasahujeme mimo poľa. Naviac, test či sa posúvané číslo nachádza v okrajovom políčku bude vždy negatívny (je tu predsa číslo -1). Takýmto spôsobom si vieme ušetriť kontrolu na okraje poľa v kóde a urobiť ho tak jednoduchším.

Veľa z vašich riešení uvažovalo možné polohy posúvaného čísla vzhľadom na medzeru (hore, dole, vpravo, vľavo) nezávisle ako štyri rôzne prípady. V skutočnosti je to ale to isté – ak sa na jednom zo štyroch políčok nachádza naše číslo, tak ho vymeníme s medzerou. Tieto štyri možnosti môžeme spojiť dokopy nasledovným trikom.

Všimnime si, že súradnice políčka hneď vedľa medzery sa líšia od súradníc medzery práve o jedna. Preto, budeme mať dve polia (každé pre jednu súradnicu) kde si budeme pamätať 4 záznamy (pre každý smer jeden) ako sa mení patričná súradnica keď sa posunieme v tomto smere. Napríklad, ak by sme chceli ísť doprava, tak riadková súradnica sa nám nezmení a stĺpcová narastie o jedna. Čiže náš záznam by obsahoval 0 pre riadkový smer a +1 pre stĺpcový smer. Keď teraz chceme skontrolovať aké číslo je vedľa medzery v nejakom zo štyroch smerov, stačí postupne vyskúšať všetky štyri diferencie a prepočítať si súradnice z medzery na susedné políčko. Potom sa vieme rovno pozrieť, že či sa na tomto políčku nachádza posúvané číslo a ak áno, vykonať ťah a aktualizovať pozíciu medzery. (V zdrojovom kóde je prezeranie štyroch smerov v časti s for-cyklom s riadiacou premennou d.) Už iba ako poznámka, tento istý trik vieme použiť aj keď sú pohyby v mriežke komplikovanejšie, napríklad pri simulácii pohybu šachového koňa.

Body za popis ste mohli stratiť ak vaše riešenie bolo pomalšie ako optimálne, ak vaše odhady časovej alebo pamäťovej zložitosti boli chybné alebo ak váš zdrojový kód obsahoval chyby (a neboli odhalené testovaním).

Listing programu:

```
program zhra;

var H: array of array of longint;
    nulaR,nulaS: longint;                {riadok a stlpec, kde sa nachadza medzera}
    novyR,novyS: longint;                {suradnice vedla medzery}
    cislo: longint;                      {cislo, ktorym ideme hybat v tahu}
    r,s,i,k,d,n: longint;
    dr: array[1..4] of integer = (1, 0,  0, -1);
    ds: array[1..4] of integer = (0, 1, -1,  0);

begin
  read(n);
  SetLength(H,n+2,n+2);
  for r:=0 to n+1 do                     {ulozime si do pola hodnoty mimo rozsahu}
    for s:=0 to n+1 do
      H[r,s]:= -1;

  for r:=1 to n do                       {nacitame vstup}
    for s:=1 to n do
    begin
      read(H[r,s]);
      if H[r,s]=0 then                   {ulozime si poziciu medzery}
      begin
```

```
        nulaR:= r;
        nulaS:= s;
      end;
    end;

  read(k);
  for i:=1 to k do
  begin
    read(cislo);                            {nacitavame tahy}
    for d:=1 to 4 do                        {pozrieme sa naokolo medzery}
    begin
      novyR:= nulaR + dr[d];
      novyS:= nulaS + ds[d];
      if (H[novyR,novyS] = cislo) then      {ak je naokolo nase cislo}
      begin                                 {urobime tah}
        H[novyR,novyS]:= 0;
        H[nulaR,nulaS]:= cislo;
        nulaR:= novyR;                      {ulozime si poziciu novej medzery}
        nulaS:= novyS;
        break;
      end;
    end;
  end;

  for r:=1 to n do                          {vypiseme vystup}
    for s:=1 to n do
    begin
      write(H[r,s]);
      if s = n then writeln()
      else write(' ');
    end;
end.
```

opravoval MišoF
(max. 12 b za popis, 0 b za program)

3. Zaplav to! (vzorák 1 z 3)

Vitajte pri milom a jednoduchom vzorovom riešení tejto úlohy. V tomto vzorovom riešení si povieme, ako som vaše riešenia bodoval a ako sme očakávali, že budú vyzerať vaše riešenia, ktoré dostanú plný počet bodov.

Pár slov o bodovaní úlohy

Zadanie kázalo „napíš program". Štyri body som teda dával za dobre spravený program, ktorý by bez problémov vedel zvládnuť napr. $n = 5000$.

Zadanie hovorilo „veľkosti strán hracieho plánu vyskúšaj rôzne, *vrátane čo najväčších*". Ďalších päť bodov som dával za to, ak ste správne namerali výsledky pre rozumne veľké n. Kto sa uspokojil s n niekde do 30, musí sa teraz uspokojiť aj s menším počtom bodov.

(Ako chcete len na základe tak malého n povedať niečo o tom, čo sa deje pre veľké n? Pre $n = 30$ by ste napríklad neuvideli, aký obrovský rozdiel je medzi časovou zložitosťou dobrých a zlých triediacich algoritmov – napr. MergeSort verzus BubbleSort. To sa prejaví až pre omnoho väčšie n. A podobne je to aj v tejto úlohe. Asi hlavné poučenie – takúto úlohu sa neoplatí nechávať na posledný deň `:-)`.)

Posledné tri body sa dali získať za vhodné a pekné zobrazenie získaných výsledkov v grafe. Grafy, ktoré ste vyrobili, boli poväčšine strašné `:-)`. To hlavné, čo im bežne chýbalo: čitateľ musí ľahko vedieť (zistiť), čo že to vlastne graf zobrazuje. Čomu zodpovedá x-ová os a čomu y-ová? Aký majú rozsah hodnôt?

A ešte tu boli bonusy. Bonusový trinásty bod sa dal získať za správne uhádnutie toho, ako zhruba závisí veľkosť výsledku od hodnoty n. No a extra bonusový štrnásty bod sa získať v podstate nedal, lebo nik nemal rozumnú šancu uhádnuť, ako závisí veľkosť výsledku od f.

Ako merať?

Aby sme niečo mohli merať, potrebujeme urobiť to, čo nám priamo kázalo zadanie: napísať program, ktorý bude vedieť pre dané n a f vygenerovať náhodný hrací plán a nájsť na ňom

najväčšiu jednofarebnú oblasť. Kľúčová je samozrejme efektívnosť takéhoto programu. Čím bude rýchlejší, tým väčšie n si môžeme dovoliť vyskúšať.

Na šikovné zistenie veľkosti najväčšej oblasti vieme použiť niektorý z algoritmov na prehľadávanie grafu: napr. prehľadávanie do hĺbky alebo prehľadávanie do šírky. My si popíšeme to druhé. Prehľadávanie do šírky sa po anglicky volá breadth-first search (BFS), niekedy sa mu tiež hovorí flood fill. Predstavuje spôsob, akým by daný graf (resp. v našom prípade súvislú jednofarebnú oblasť) zaplavila voda, šíriaca sa zo začiatočného vrcholu (políčka hracieho plánu) rovnomerne do všetkých smerov.

Prehľadávanie do šírky z konkrétneho políčka ľahko implementujeme pomocou dátovej štruktúry fronta:

```
prehľadaj(políčko p):
    označ p ako navštívené
    vyrob frontu Q obsahujúcu jediný prvok p
    kým Q nie je prázdna:
        vyber zo začiatku fronty Q políčko x
        pre každého suseda y políčka x:
            ak y ešte nebolo navštívené a má rovnakú farbu ako malo p:
                označ y ako navštívené
                vlož y na koniec fronty Q
```

V programe použijeme ako farby políčok čísla od 1 po f. Následne si navštívené políčka budeme jednoducho značiť tak, že ich farbu zmeníme na číslo 0.

Je zjavné, že počet krokov potrebný na prehľadanie hocijakej súvislej jednofarebnej oblasti je priamo úmerný jej počtu políčok – totiž každé jej políčko práve raz označíme ako navštívené, práve raz ho vložíme do fronty, práve raz ho odtiaľ vyberieme a práve raz ho spracujeme.

Celý algoritmus teraz môžeme v pseudokóde zhrnúť nasledovne:

```
pre každý riadok r:
    pre každý stĺpec s:
        ak je políčko (r,s) ešte nenavštívené:
            prehľadaj( políčko (r,s) )
```

Teda v dvoch cykloch prechádzame pole a vždy, keď nájdeme neoznačené políčko, spustíme z neho prehľadávanie do šírky. Takto postupne po jednom nájdeme a ofarbíme všetky súvislé oblasti. Celková časová zložitosť je priamo úmerná súčtu veľkostí oblastí, čiže celkovému počtu políčok na pláne. Formálne je teda časová zložitosť priamo úmerná číslu n^2, čo zapisujeme „časová zložitosť je $\Theta(n^2)$“. Lepšie to nejde, lebo rádovo toľko isto krokov potrebujeme na samotné vygenerovanie hracieho plánu.

Tu je celý program (v C++):

Listing programu:

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

int N, F, K;            // strana stvorca, pocet farieb, pocet levelov
char A[10010][10010];   // mapa levelu
int dr[] = {-1,1,0,0}, dc[] = {0,0,-1,1}; // smery na susedne policka

int main() {
    cin >> N >> F >> K;
    for (int k=0; k<K; ++k) {
        // vygenerujeme level (okolo neho su 0, lebo A je globalna premenna)
        for (int r=1; r<=N; ++r) for (int c=1; c<=N; ++c) A[r][c] = 1 + rand()%F;
        // ... a prehladame ho do sirky
        int odpoved = 0;
        for (int r=1; r<=N; ++r) for (int c=1; c<=N; ++c) if (A[r][c]!=0) {
            // nasli sme novy komponent, prefarbime ho na 0 a spocitame jeho velkost
```

```
            int farba = A[r][c], velkost = 1;
            A[r][c] = 0;
            queue<int> Q; Q.push(r); Q.push(c);
            while (!Q.empty()) {
                int cr=Q.front(); Q.pop();
                int cc=Q.front(); Q.pop();
                for (int d=0; d<4; ++d) {
                    int nr=cr+dr[d], nc=cc+dc[d];
                    if (A[nr][nc]!=farba) continue;
                    A[nr][nc]=0; ++velkost;
                    Q.push(nr); Q.push(nc);
                }
            }
            if (velkost > odpoved) odpoved = velkost;
        }
        cout << odpoved << endl;
    }
}
```

Tento program má jeden nedostatok – konkrétne ten, že nepoužíva skutočne náhodné čísla, len *pseudonáhodné*. V tomto vzorovom riešení tento nedostatok spokojne odignorujeme.

Čo vlastne merať?

Skôr, než začneme merať, neuškodí zamyslieť sa nad tým, čo vlastne bude dobré merať. V prvom rade si treba uvedomiť, že keď vygenerujeme dva rôzne hracie plány pre rovnaké n a f, je možné (a zväčša aj veľmi pravdepodobné), že sa veľkosti najväčších jednofarebných miestností budú líšiť. Aby sme teda vedeli povedať niečo zmysluplné, nestačí vygenerovať len jeden hrací plán, ani desať, ale čo najviac. Až z dostatočne veľkej *štatistickej vzorky* budeme vedieť povedať niečo zmysluplné o tom, ako sa veľkosť najväčšej jednofarebnej oblasti správa pre pevne zvolené n a f.

No a keď toto budeme dostatočne presne vedieť, môžeme potom prikročiť k úvahe komplikovanejšej: pokúsiť sa zistiť niečo o tom, ako sa správa veľkosť najväčšej jednofarebnej oblasti keď meníme hodnoty n a f.

Prvou fázou riešenia teda bude, že si zvolíme pevné n a f a vygenerujeme veľa hracích plánov. Vyššie uvedený program s tým už počíta, premenná k udáva počet hracích plánov, ktoré program vygeneruje a spracuje. Výsledkom spustenia programu bude postupnosť k celých čísel – veľkosti najväčších oblastí na vygenerovaných plánoch. Túto postupnosť sa nám samozrejme nechce študovať celú. (Obzvlášť ak budeme mať takých postupností stovky pre rôzne n a f.)

Ak teda chceme o našej postupnosti čísel rýchlo zistiť niečo zmysluplné, máme dve možnosti: prvou sú *štatistiky* a druhou je *vizualizácia*. Najskôr sa zamyslíme nad zmysluplnými štatistikami.

- „Povinnou" štatistikou, ktorú ste aj merali snáď všetci, je *priemer* (v štatistike tiež nazývaný *stredná hodnota*).
- Štatistikou, ktorú naopak nemeral nikto (pretože ju ešte nepoznáte), je *disperzia*, resp. po slovensky *rozptyl*. Toto je číslo, ktoré udáva, ako ďaleko okolo priemeru sa pohybujú naše merania. (Ak by všetky merania vyšli presne rovnako, vzorec pre rozptyl vypočíta hodnotu 0.)

  Keďže ale už zachádzame za hranice stredoškolskej matematiky, skúsime sa zaobísť bez rozptylu. Ukážeme si o chvíľu, že podobnú informáciu vieme získať aj ináč.
- Ďalšou zaujímavou štatistikou je *najčastejšie nameraná hodnota*, odborne nazývaná *modus*.

  V našej úlohe by sme o moduse zistili dve veci. Prvou je, že modus vychádza o niečo menší ako priemer. A druhou je, že modus príliš nemá zmysel merať. Ukáže sa totiž, že pre veľké n sú viaceré hodnoty zhruba rovnako pravdepodobné – a teda kým nepoužijeme veľmi veľký počet plánov k, bude konkrétny nameraný modus ovplyvnený náhodným šumom. V ďalšom texte teda už modus neuvažujeme.

- Mnohých tiež napadlo pozerať sa na *minimum* a *maximum* – teda nájsť najhorší a najlepší spomedzi všetkých vygenerovaných hracích plánov.
- A táto myšlienka sa dá ešte pekne zovšeobecniť: postupnosť nameraných hodnôt totiž môžeme usporiadať. Na jej začiatku tak uvidíme minimum, na jej konci zase maximum.

  A do usporiadanej postupnosti sa môžeme pozerať aj inam. Najintuitívnejším ďalším miestom je hodnota uprostred, nazývaná *medián*.

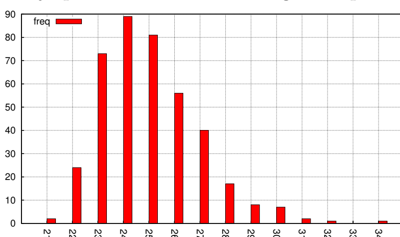
  V grafoch, ktoré si ukážeme neskôr, sa okrem mediánu budeme pozerať aj na ďalšie miesta. Presnejšie, pozrieme sa na hodnoty, ktoré ležia v 10%, 30%, 50% (toto je medián), 70% a 90% usporiadanej postupnosti nameraných hodnôt.

  Toto je práve ten vyššie spomínaný spôsob, ako ľahko získať nejaké informácie o rozptyle: Ak by sme vždy namerali to isté, týchto päť hodnôt bude rovnakých. A ak sa namerané hodnoty často od priemeru výrazne líšia, na týchto piatich hodnotách sa to dostatočne výrazne prejaví.

Druhým nástrojom, ktorý nám umožní rýchlo uvidieť, čo sa deje, je vizualizácia. Potrebujeme si vedieť vhodne znázorniť výsledok meraní, ktoré sme spravili. Vhodnou voľbou je v tomto prípade *histogram* – stĺpcový graf ukazujúci, ktorú hodnotu sme namerali ako často.

(Rozmyslite si, že z histogramu je pekne vidieť minimum, maximum aj modus. Takisto vieme aspoň približne odhadnúť priemer a medián, tie ale až tak pekne neuvidíme. Možným vylepšením by bolo dodatočne ich do grafu na správne miesta dokresliť.)

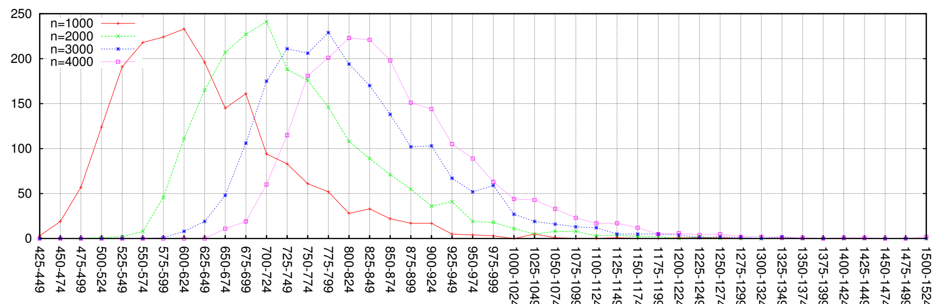
Nasledujúci obrázok ukazuje príklad takéhoto histogramu pre $n = 6400$, $f = 6$ a $k = 401$:



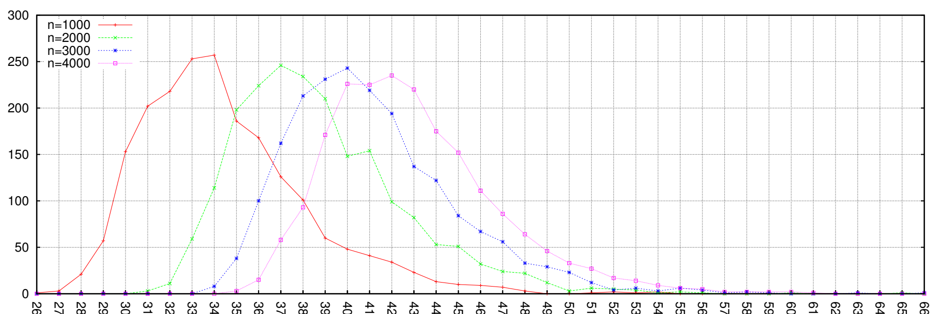
Meníme veľkosť plánu

Teraz sa už pozrieme aj na to, čo sa deje, keď nám rastie veľkosť hracieho plánu n. Uvažovali sme hodnoty n od 1000 po 4000 (s krokom 1000).

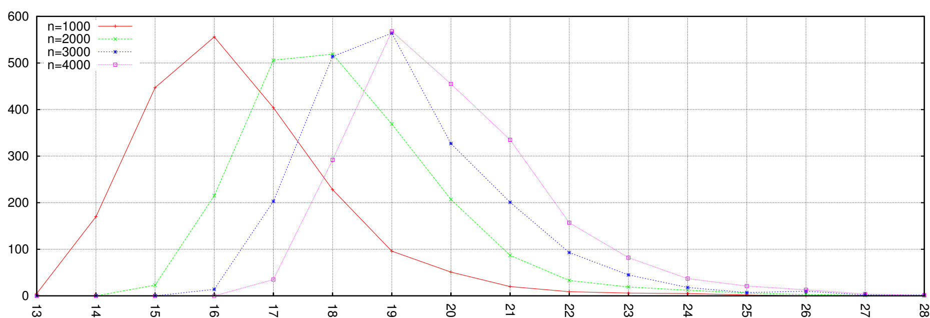
Najskôr sa pozrieme na situáciu pre $f = 2$ farby. Nasledujúci graf ukazuje rozloženia nameraných hodnôt pre jednotlivé n a $k = 2000$ meraní.



To isté pre $f = 4$ farby:



A pre veľký úspech ešte raz pre $f = 7$ farieb:



O maxime a minime

Zo všetkých vyššie uvedených grafov vidíme to isté: Minimum je pomerne „stabilné", ľavá strana „kopca" je strmá. Na druhej strane, na pravom konci je veselo, keď sa zadarí, maximum občas „uletí" pekne ďaleko. (Toto má potom za dôsledok aj to pozorovanie, že medián aj modus sú oba o trošku menšie ako priemer. Veľmi veľké maximálne hodnoty totiž zvýšia priemer, ale nemajú vôbec žiaden vplyv na medián ani na modus.)

Toto pozorovanie si vieme intuitívne zdôvodniť napríklad nasledovne: V závislosti od n a f sa zrejme dá určiť nejaká minimálna veľkosť v taká, že je skoro isté, že sa aspoň jedna oblasť veľkosti aspoň v niekde na pláne vyskytne. Napríklad pre $f = 4$ je už pre $n = 10$ takmer nepredstaviteľné aby bola odpoveď 1 – aspoň nejaká dvojica susedných políčok skoro určite bude mať tú istú farbu. Veľmi malé výsledky budú teda veľmi veľmi vzácne.

Na druhej strane pri maxime často stačí trocha šťastia na to, aby sa výrazne zmenila veľkosť najväčšej oblasti. Predstavte si napríklad, že hrací plán postupne vzniká tak, že sa vyberie *náhodné* neofarbené políčko a to sa náhodne ofarbí.[1] Na hracom pláne nám postupne vznikajú čoraz väčšie jednofarebné oblasti. A často sú dve oblasti jednej farby pomerne blízko pri sebe. No a potom „ak sa trochu zadarí" a časom ofarbíme políčko či políčka medzi nimi tou istou farbou, zrazu sa nám tieto dve oblasti spoja do jednej väčšej.

To isté inými slovami: Na to, aby sme namerali veľký výsledok, stačí mať šťastie na jednom mieste hracieho plánu, kde vznikne veľká oblasť. Ale na to, aby sme namerali výsledok malý, musia vzniknúť samé malé oblasti všade, na úplne celom hracom pláne.

[1] Tento postup vyrába presne rovnako pravdepodobné ofarbenia ako keď ideme za radom a každé políčko náhodne ofarbíme. Pre nasledujúcu úvahu však je tento postup názornejší.

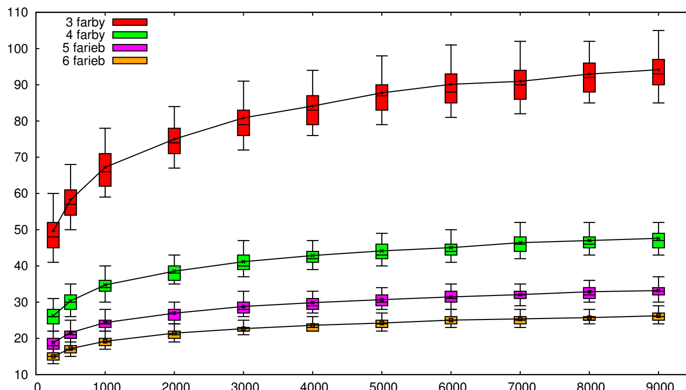
Toto má samozrejme do exaktných argumentov ďaleko. Ale nebojte sa,[2] aj na tie príde.

Ešte pár iných grafov

Predchádzajúce grafy nám síce naznačili niečo o tom, čo sa deje, keď pre pevné f zvyšujeme n, ale vyčítať z nich nejakú presnejšiu závislosť je ťažké. My si preto to isté skúsime znázorniť ešte raz, ale ináč. (Toto je práve ťažiskom celej vizualizácie – hľadať k zložitým dátam taký spôsob ich znázornenia, pri ktorom vieme o dátach pohľadom toho čo najviac zistiť.)

Pred chvíľkou sme si zdôvodnili, že konkrétna hodnota, ktorú nameriame ako maximum, dosť výrazne závisí od toho, aké máme zrovna šťastie. Nebudeme sa teda pozerať na maximum (a ani na minimum), ale obmedzíme sa na hodnoty, kde už šťastie nehrá až takú významnú rolu. Ako sme už spomínali vyššie, do grafu si znázorníme hodnoty, ktoré ležia v 10%, 30%, 50%, 70% a 90% usporiadanej postupnosti nameraných hodnôt.

Toto vieme spraviť do jedného veľkého grafu, ktorý vidíte nižšie. Rôzne farby predstavujú rôzne hodnoty f. Body na x-ovej osi predstavujú rôzne hodnoty n. Každej dvojici (n, f) teda zodpovedá jeden „farebný obdĺžniček s anténkami“. Môžete si všimnúť, že každý z týchto útvarov obsahuje presne 5 vodorovných čiariek. Tie udávajú, zdola hore, dotyčných 5 významných nameraných hodnôt. (Prostredná čiarka je teda vždy medián, najvyššia čiarka je veľkosť v taká, že len v 10% prípadov sme namerali odpoveď väčšiu ako v.)



Čierne značky spojené lomenou čiarou predstavujú priemer nameraných hodnôt. (Môžete si všimnúť, že je zväčša o čosi vyšší ako medián, presne ako sme si to vyššie zdôvodnili.) Hodnoty pre $f = 2$ farby sme do grafu nedávali – sú natoľko veľké, že sa doň nevošli. Vyzerajú však veľmi podobne.

Tiež si môžete všimnúť, že z tohto grafu vieme vyčítať aj to, že pri maxime je väčší rozptyl hodnôt ako pri minime. Stačí si všimnúť, že skoro vždy je horná „anténka“ dlhšia ako dolná.

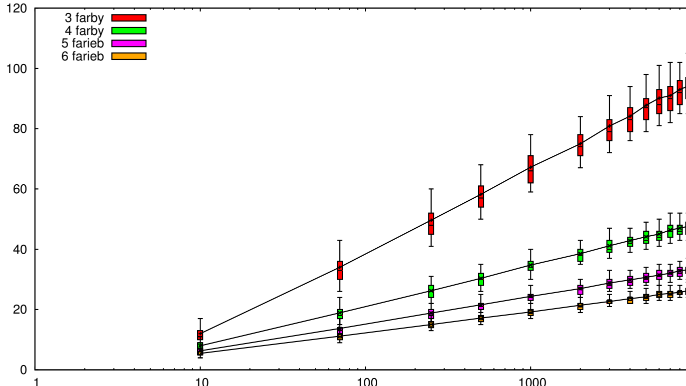
Vyslovenie nejakej tej hypotézy

Tie čierne krivky na predchádzajúcom grafe rozhodne nevyzerajú ako lineárne funkcie, obzvlášť keď si domyslíme, že smerom doľava pokračujú všetky veľmi rýchlo do nuly. Zhruba takéto grafy v matematike najčastejšie vidíme v dvoch situáciách: prvou sú odmocniny (presnejšie mocniny s exponentom medzi 0 a 1) a druhou sú logaritmy.

Dobrým nástrojom na zistenie, čo že sa to deje v našom prípade, je logaritmická mierka. Pri výrobe grafov sa často oplatí jednu či obe súradnice nezobrazovať lineárne (čo centimeter to rovnaká zmena) ale v logaritmickej škále (čo centimeter to rovnaký násobok). Skúsme si teda

[2] Môžete aj behať!

predchádzajúci graf prepnúť do logaritmickej škály na osi x – teda budeme skúmať, ako závisí veľkosť najväčšej oblasti od hodnoty $\log n$. Pre väčšiu názornosť sme do obrázku ešte doplnili dve menšie hodnoty n.



No aha ho aké pekné polpriamky. Takže máme hypotézu: priemerná veľkosť najväčšej jednofarebnej oblasti je zhruba priamo úmerná logaritmu dĺžky strany hracieho plánu. Koeficient priamej úmernosti zjavne klesá s rastúcim f, ale zatiaľ nebudeme bližšie skúmať túto závislosť.

opravoval MišoF
(max. 12 b za popis, 0 b za program)

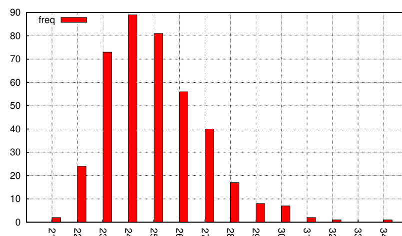
3. Zaplav to! (vzorák 2 z 3)

Vitajte pri vzorovom riešení úlohy Zaplav to!

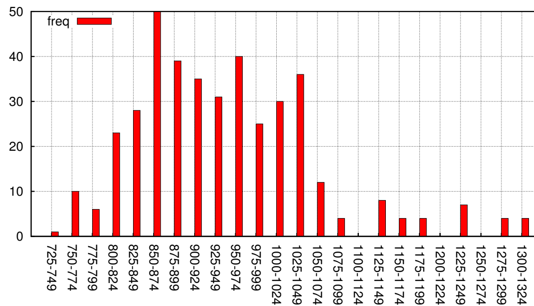
Čo, že ste už jedno čítali? To je úplne v poriadku. Toto vzorové riešenie bude úplne o inom. Povieme si v ňom niečo viac o generovaní náhodných čísel a tiež skúsime uhádnuť, ako že to závisí odpoveď od počtu farieb. Obe tieto veci už neboli nijak zohľadňované pri bodovaní, uvádzame ich len pre zaujímavosť.

Čísla náhodné a pseudonáhodné

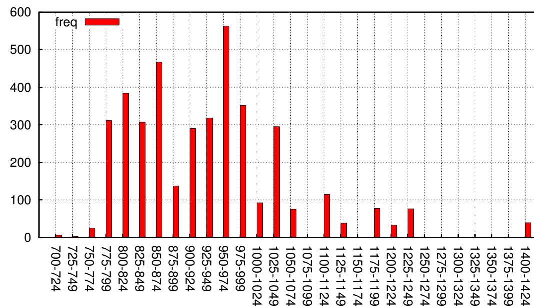
Prvý graf, ktorý sme si minule ukazovali, bol histogram pre $n = 6400$, $f = 6$ a $k = 401$:



Jéj, aké pekné, kopec. Pozrime sa, čo sa stane, keď počet farieb zmenšíme na dve. Tam sme namerali odpovede od 735 po 1304. Ich približné rozložene vidíme v druhom histograme. (Keďže bolo nameraných odpovedí veľa, každý bod x-ovej osi teraz zodpovedá jednému malému intervalu hodnôt. Všetky tieto intervaly sú rovnako dlhé.)



Ale čo to? Kde je pekný kopec? Tu nám tie výšky stĺpčekov pomerne divoko skáču. Ale nie je to ešte nič neuveriteľné – rozsah nameraných hodnôt je totiž v stovkách a my sme spravili len 401 meraní. Niet divu, že je to ešte rozhádzané ako papiere v mojej izbe. Tak teda skúsme spraviť rádovo viac meraní. My sme spravili $k = 4001$ nových meraní a dostali sme toto:



Moment. Tu asi niečo nesedí. Máme tu štyritisíc meraní a stále ten graf ani zďaleka nemá ten pekný tvar, ktorý mal pre 6 farieb. A čo sa to tam deje na tej pravej strane? Žiadny pokus nedal odpoveď od 1250 po 1399, ale hneď asi 40 ich dá odpoveď väčšiu ako 1400? Ešte zvláštnejšie to vyzerá, ak sa pozrieme na presné hodnoty, ktoré náš program vygeneroval. Najväčšia nájdená odpoveď je 1418, a to hneď 39×. A druhá najväčšia je 1237, a to 38×. Hmm, že by sme boli na stope nejakej matematickej zákonitosti?

Ale kdeže. V skutočnosti sme len zvládli kolosálne zblbnúť samých seba. A to tým, že mizerne generujeme náhodné čísla. A vlastne ani nie náhodné, len pseudonáhodné.

V našom programe používame knižničnú funkciu `rand`. Tá vyrába pseudonáhodnú postupnosť čísel. Hodnoty vyrábané funkciou `rand` majú mnohé vlastnosti skutočne náhodných čísel. Ukazuje sa ale, že bohužiaľ v našom prípade sa ich nie úplná náhodnosť môže prejaviť. Pri našich výpočtoch sme vygenerovali $6400^2 \cdot 4001$ náhodných čísel, čo je vyše 160 miliárd. A pri takto obrovských počtoch sa nám niektoré závislosti prejavili tým, že sa niektoré typy vstupov nevygenerovali vôbec a niektoré iné naopak častejšie ako by sa mali. (Situáciu zhoršilo aj to, že sme si ako n zvolili číslo deliteľné veľkou mocninou 2 a tiež to, že pre $f = 2$ sa každému pseudonáhodnému číslu vlastne pozeráme len na posledný bit.)

Skúsme teda použiť lepší generátor náhodných čísel. Jednu lepšiu možnosť nám ponúka nové C++11, v ktorom máme k dispozícii viacero kvalitných pseudonáhodných generátorov. Nasledujúci kód používa generátor Mersenne twister, ktorý má periódu až $2^{19937} - 1$ a viacero dobrých kombinatorických vlastností.

Listing programu:

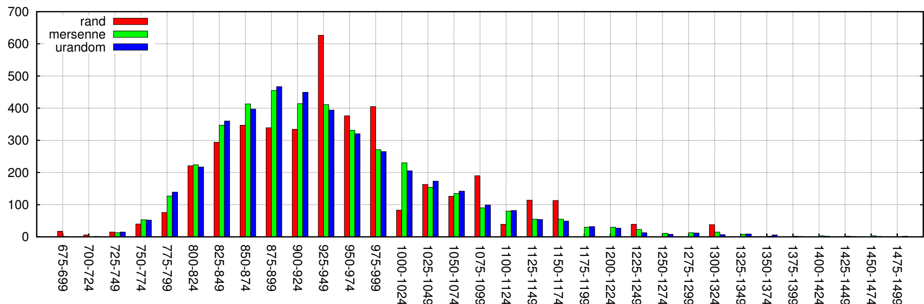
```
// len casti, ktore sa zmenili:
#include <random>
mt19937 myRNG; // Mersenne twister generator pseudonahodnych cisel
myRNG.seed( time(0)^getpid() );
uniform_int_distribution<uint32_t> myDist(1,F); // distribucia rovnomerne vracajuca hodnoty 1..F
for (int r=1; r<=N; ++r) for (int c=1; c<=N; ++c) A[r][c] = myDist(myRNG); // vygeneruje level
```

V Linuxe je ďalšou možnosťou zobrať ako zdroj náhodných čísel súbor (presnejšie device) `/dev/random`. Ten nám poskytuje „skutočne“ náhodné bity pochádzajúce z hardvéru v počítači (napr. fluktuácie posledného bitu teplotných senzorov). Alebo, keďže máme málo času, radšej súbor `/dev/urandom`. Totiž `/dev/random` vypisuje len toľko náhodných bitov, koľko má práve k dispozícii. Preto `/dev/urandom` kombinuje obe predchádzajúce techniky: náhodnými bitmi z `/dev/random` inicializuje pseudonáhodný generátor, ktorý nám následne dogeneruje ďalšie bity podľa potreby.

Listing programu:

```
// len casti, ktore sa zmenili:
#include <cstdio>
FILE *frand;
inline int random_bit() {
    unsigned char c; fscanf(frand,"%c",&c);
    c ^= c>>4; c &= 15; c ^= c>>2; c &= 3; c ^= c>>1; c &= 1;
    return c; // return (parita poctu jednotkovych bitov v c)
}
frand = fopen("/dev/urandom","r");
for (int r=1; r<=N; ++r) for (int c=1; c<=N; ++c) A[r][c] = 1+random_bit();
```

Porovnanie týchto troch náhodných generátorov vidíme na nasledujúcom grafe (stále pre $n = 6400$, $f = 2$, $k = 4001$). Mersenne twister aj urandom oba dávajú, na rozdiel od štandardného náhodného generátora, očakávané pekné rozdelenie veľkosti najväčšej oblasti.

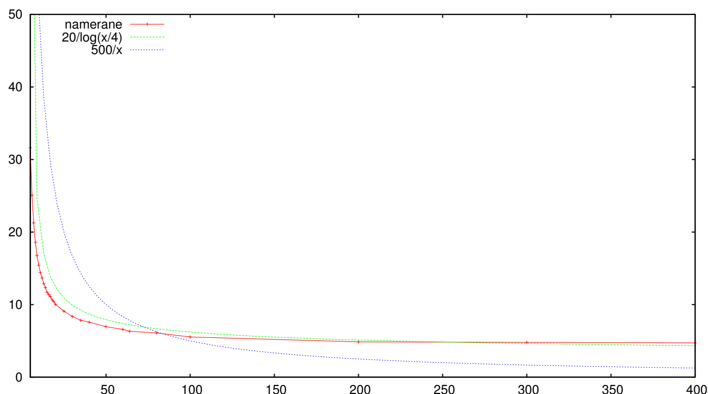


Dáta a ich grafy, ktoré ste videli v prvom vzorovom riešení, boli v skutočnosti všetky generované programom používajúcim Mersenne twister pseudonáhodný generátor.

Závislosť výsledku od počtu farieb

Už máme hypotézu, že veľkosť najväčšej oblasti rastie rádovo rovnako rýchlo ako logaritmus čísla n. Zároveň ale táto veľkosť závisí aj od počtu použitých farieb – čím viac farieb, tým menšie budú jednofarebné oblasti. Ale ako presne závisí veľkosť najväčšej z nich od f?

Urobili sme teda ešte poslednú sadu experimentov: pre $n = 6400$ sme si postupne vyskúšali rôzne hodnoty od $f = 2$ až po $f = 400$. Priemerné namerané hodnoty sú v nasledujúcom grafe červenou farbou. Zelenou farbou je tam veľmi podobne klesajúca funkcia, a pre porovnanie modrou je tam násobok funkcie $1/x$, ktorá klesá rádovo rýchlejšie ako nami namerané dáta.



Záverečná hypotéza teda znie: veľkosť najväčšej jednofarebnej oblasti je *približne priamo úmerná* hodnote $\log n$ a *približne nepriamo úmerná* hodnote $\log f$.

opravoval Mio

4. Zamyslený alchymista

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

Triviálne kvadratické riešenie funguje takto: Dokola skúšame všetky pravidlá, až kým sa nám nestane, že už sme nevyrobili žiaden prvok.

Toto sa dá vylepšiť, ak si uvedomíme, že každé pravidlo treba použiť najviac raz, ale vo vhodnom momente. Nemá zmysel zaoberať sa pravidlami, pre ktoré ešte nemáme vyrobený ani jeden vstupný prvok. Vždy keď vyrobíme nový prvok, tak sa nám môže zväčšiť aj množina pravidiel, ktoré môžme použiť - za žiadnych iných okolností sa nám táto množina určite nezväčší. Keď už raz nejaké pravidlo použijeme, znovu ho už použiť nemusíme.

Z týchto pozorovaní sa nám už črtá algoritmus. Načitame si počiatočné prvky a zoznam pravidiel. Ku každému prvku si zapamätáme v ktorých pravidlách sa nachádza - aby sme ich moholi vyhľadať v konštantnom čase. Budeme si pamätať, koľko prvkov pravidla ešte musíme vyrobiť, aby sme ho moholi použiť. Na začiatku je táto hodnota u všetkých pravidiel 2. Vždy, keď vyrobím nové prvky zapamätám si počet krokov, na ktorý som ho vyrobil[3] a znížim počet nevyrobených prvkov všetkým pravidlám v ktorých sa daný prvok nachádza[4] a ak táto hodnota dosiahla 0, pridám pravidlo do fronty pravidiel. Na začiatku vyrobím základné prvky a potom postupne vyberám pravidlá z fronty a vyrábam nové prvky. Keď už vo fronte nie sú žiadne nové pravidlá, môžem skončiť a vypísať výsledky. Každý prvok pozerám len raz, tak na každé pravidlo sa pozrem práve 2x - za každý vstupný prvok pravidla raz. Spolu s načítaním vstupu nám to dáva zložtosť $O(m+k)$. Pamäťová zložitosť bude tiež $O(m+k)$, pretože si potrebujeme zapamätať všetky prvky a pravidlá.

Prečo nefunguje prístup z opačnej strany? Opačný prístup vyzerá takto: Chceme zistiť na koľko krokov sa dá vyrobiť prvok C. Nájdeme si všetky pravidlá ktoré vyrábajú C a $\forall i$ spočítame $P(C_i) = max(P(A_i), P(B_i)) + 1$ a $P(C) = min(P(C_i))$. Ak potrebnú hodnotu ešte nemáme vypočítanú tak si ju rekurzívne vypočítame. Problémom sú však cyklické pravidlá. Pre takýto korektný vstup sa nám môže program zacykliť.

[3] ak máme pravidlo $A + B = C$, tak $P(C) = max(P(A), P(B)) + 1$

[4] ak je pravidlo typu $X + X = Y$, tak sa táto hodnota zníži o 2

vstup

```
4 2 3
1 4
1 2 3
4 3 2
1 1 2
```

Bodovanie za popis:

- Riešenie v $O(m+k)$ získalo max 10 bodov.
- Riešenie v $O(m+k.logk)$ alebo $O(m.logm+k)$ získalo max 8 bodov.
- Riešenie v $O(k*(m+k))$ získalo max 6 bodov.
- Horšie funkčné riešenie získalo max 5 bodov.

Ďalšie body boli strhnuté za rôzne nedostatky, ako napríklad chýbajúce, alebo zlé odhady zložitostí, či horšiu pamäťovú zložitosť.

Listing programu:

```
program alchymista;

uses math;

type
pravidlo = record
  a,b,c:longint;
end;

var i,j,k,n,m,fronta_zactiatok, fronta_koniec, t:longint;
    pociatocne,pocetkrokov,fronta:array of longint;
    pravidla:array of pravidlo;
    index:array of array of longint;
    pocetnesplnenych:array of integer;


begin
  readln(k,n,m);
  SetLength(pociatocne,n);
  for i:=0 to n-1 do
  begin
    read(pociatocne[i]);
    dec(pociatocne[i]);
  end;
  readln;


  SetLength(pravidla,m);
  SetLength(pocetnesplnenych,m);
  SetLength(fronta,m);
  SetLength(index,k);
  SetLength(pocetkrokov,k);

  for i:=0 to m-1 do
  begin
    readln(pravidla[i].a,pravidla[i].b,pravidla[i].c);
    dec(pravidla[i].a);
    dec(pravidla[i].b);
    dec(pravidla[i].c);
    SetLength(index[pravidla[i].a],length(index[pravidla[i].a])+1);
    index[pravidla[i].a][length(index[pravidla[i].a])-1] := i;
    SetLength(index[pravidla[i].b],length(index[pravidla[i].b])+1);
    index[pravidla[i].b][length(index[pravidla[i].b])-1] := i;
    pocetnesplnenych[i] := 2;
  end;

  for i:=0 to k-1 do
  begin
```

```
    pocetkrokov[i]:=1000000;
  end;

  fronta_zactiatok:=0;
  fronta_koniec:=0;

  for i:=0 to n-1 do
  begin
    pocetkrokov[pociatocne[i]] := 0;
    for j:=0 to length(index[pociatocne[i]])-1 do
    begin
      dec(pocetnesplnenych[index[pociatocne[i]][j]]);
      if (pocetnesplnenych[index[pociatocne[i]][j]] = 0) then begin
        fronta[fronta_koniec]:=index[pociatocne[i]][j];
        inc(fronta_koniec);
      end;
    end;
  end;

  while fronta_koniec>fronta_zactiatok do begin
    t := fronta[fronta_zactiatok];
    inc(fronta_zactiatok);
    if (pocetkrokov[pravidla[t].c] = 1000000) then begin
      pocetkrokov[pravidla[t].c] := 1 + max(pocetkrokov[pravidla[t].a],
pocetkrokov[pravidla[t].b]);
      for i:=0 to length(index[pravidla[t].c])-1 do
      begin
        dec(pocetnesplnenych[index[pravidla[t].c][i]]);
        if (pocetnesplnenych[index[pravidla[t].c][i]] = 0) then begin
          fronta[fronta_koniec]:=index[pravidla[t].c][i];
          inc(fronta_koniec);
        end;
      end;
    end;
  end;

  for i:=0 to k-1 do
  begin
    writeln(pocetkrokov[i]);
  end;
end.
```

opravoval Usamec

5. Optimálne skákanie

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

Z toho, že hodnotenie úlohy je postavené na tom, ako dobré riešenie sa vám podarí vyrobiť, sa dalo usúdiť, že pre túto úlohu nie je známe optimálne riešenie (resp. nie je známy spôsob, ako vyrobiť optimálne riešenie dostatočne rýchlo).

Poďme sa pozrieť na to ako s čo najmenšou námahou nažmýkať, čo najviac bodov. V prvom rade pre malé N môžeme napísať riešenie, ktoré vyskúša fakt všetky možnosti a garantuje nám optimálne riešenie. Vhodný spôsobom ako napísať takéto riešenie je rekurzia. V každom kroku vygenerujeme všetky možnosti na nasledujúce číslo (rozmyslite si pokiaľ má zmysel skúšať nasledujúce číslo) a následne overíme ako ďaleko sme sa dostali a vylepšíme globálne maximum. Keď túto rekurziu napíšeme dobre a pustíme ju na jednu noc, tak získame optimálne riešenia pre $N \leq 11$.

Čo sa zvyšnými číslami? Tuná máme niekoľko možností, ukážeme si dve. Ani jedna z nich negarantuje nájdenie najlepšieho riešenia, ale zato nájde nejaké riešenie dostatočne rýchlo.

Prvou možnosťou je vhodne osekať prehľádavanie všetkých možností. Vygenerujme všetky možné riešenia pre nejaké vhodne malé n. Teraz si ponechajme len najlepších k. Riešenia pre $n+1$ zgenerujeme tak, že ku každému z týchto k riešení pridáme všetky možné ďalšie čísla. Tieto riešenia vyhodnotíme a opäť si necháme najlepších k. A toto opakujeme, až kým sa nedostaneme ku 30-ke. Toto je celkom slušný postup a kombináciou so spomínaným bruteforcom sa s ním pri $k = 2000$ dalo nahrabať skóre okolo 730 (čo viedlo asi ku 9.5 bodom).

Ja osobne som tento postup ešte vylepšil tým, že som algoritmu zafixoval prvé 4 čísla v postupnosti (myšlienka za tým je taká, že zmena v týchto číslach má celkom slušnú váhu a nie všetky možnosti sa pri osekávaní dostanú ďalej). A pustil som ho pre všetky možnosti na prvých 4 číslach. Môj výstup potom bol:

```
1
1 3
1 4 5
1 4 7 8
1 4 6 14 15
1 3 7 9 19 24
1 4 5 15 18 27 34
1 3 6 10 24 26 39 41
1 3 8 9 14 32 36 51 53
1 2 6 8 19 28 40 43 91 103
1 2 3 8 11 26 38 56 69 85 89
1 3 4 13 21 27 32 46 83 98 105 146
1 4 5 15 18 24 46 58 66 93 145 170 194
1 3 8 12 18 27 30 43 50 71 134 155 218 239
1 3 8 12 18 27 30 43 50 71 134 155 218 232 281
1 3 8 12 18 27 30 43 50 71 134 155 218 232 281 344
1 3 7 9 20 25 35 65 75 80 100 163 186 207 209 210 223
1 3 8 12 18 27 30 43 50 71 134 155 197 211 260 323 386 449
1 3 4 13 21 27 32 46 83 98 105 146 218 235 270 285 325 392 439
1 3 4 11 20 25 38 43 71 97 124 154 160 202 204 292 341 402 417 527
1 3 4 9 20 30 42 55 73 78 113 128 175 220 254 315 326 342 447 512 604
1 4 3 10 18 23 34 55 84 96 124 153 181 225 252 291 338 356 391 398 431 636
1 3 8 10 11 31 35 48 74 90 134 160 175 227 256 289 366 398 404 410 575 581 692
1 4 3 10 15 23 28 62 81 97 118 144 186 217 250 295 325 395 445 505 521 557 750 814
1 4 6 13 17 24 32 57 67 105 132 146 191 216 255 276 346 433 457 519 567 596 620 737 758
1 4 5 15 18 24 46 58 66 93 95 125 180 236 261 299 359 408 445 513 563 589 693 839 902 952
1 2 6 9 14 29 39 55 73 95 116 158 201 233 236 283 366 375 449 539 584 637 732 790 860 917 1053
1 4 6 13 17 24 32 57 67 105 132 146 191 216 255 276 346 424 452 507 559 672 682 824 868 1031 1045 1152
1 3 5 6 19 30 39 45 62 114 122 126 163 216 263 313 361 382 392 497 543 675 707 785 882 979 1061 1196 1225
1 3 7 8 18 21 37 60 72 103 118 130 186 208 263 287 349 429 461 511 581 625 736 785 928 987 1036 1204 1215 1334
```

Dosiahlo to skóre 763.

Pre veľké čísla sa môže hodiť viac matematický postup. Našim cieľom je nájsť nejaký vhodný pattern ako postupnosť vyrábať tak, aby dosiahla, čo najlepší výsledok. Toto vyžaduje značnu mieru autizmu a nejde to tak ľahko[5]. Pozrime sa na pattern typu:

$$1, 2, 6, 8, 19, 24, 27, 33, 45, 45 + 30 \cdot 1, 45 + 30 \cdot 2, 45 + 30 \cdot 3, \ldots, 45 + 30k$$

Dá sa ukázať, že pomocou neho je posledné dosiahnuteľné číslo $45 + 30(k + 1)$. To je zatiaľ celkom nanič. Pridajme ale zaňho ešte čísla $45 + 30k + 10, 45 + 30k + 20$. Odrazu je posledné dosiahnuteľné číslo $139 + 60k$. Keby sme to aplikovali pre $n = 30$, tak by sme dosiahli výsledok 1279, čo je stále menej ako pri predchádzajúcom spôsobe (tam sme mali 1605).

Na úplný úspech treba ešte pridať čísla $140 + 60k, 150 + 60k, 160 + 60k$. Napr. pre $n = 16$ máme postupnosť:

$$1, 2, 6, 8, 19, 24, 27, 33, 45, 75, 105, 115, 125, 260, 270, 280$$

Ktorá doskáče do vzdialenosti 414. Pre $n = 30$ dostaneme postupnosť, ktorá dosiahne vzdialenosť 1674. Toto nie je až taký veľký rozdiel na druhej strane pri číslach okolo 20-25 je ten rozdiel rádovo 150. Celkové skóre, keď zameníme postupnosti pre $n = 16$ a vyššie je 837.

6. Ostrieľaní zlomyseľníci

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

Vedenie KSP popiera akékoľvek prepojenie so spoločenstvom druidov a dištancuje sa od ich nekalých a zlomyseľných praktík. Preto sa nenašiel nik, kto by bol ochotný napísať vzorák tejto úlohy. Vyhovieme však vašim početným (i keď nepochopiteľným) žiadostiam a uverejňujeme prepis otváracej reči brata Ztrateného. Nech vám slúži na výstrahu.

„Vážení bratia, drahé sestry, milí spoludruidi! Vítam vás na ďalšom z našich jarných sympózií a dúfam, že sa počas nasledujúceho týždňa nikomu nič zlé nestane, lebo sme si zabudli doniesť lekárničku. *(hrdelný smiech)*

[5] autor tohoto vzoráku myšlienku odkukal z `http://www.recmath.org/contest/Darts/FinalReport.html`

Možno sa pýtate, prečo sa naše sympózium koná práve tu, u brata Zakopaného. Dôvodom je, že nám záleží na vašom pohodlí.

Cestná sieť, ktorá spája naše chatrče, tvorí z pohľadu teórie grafov strom. To okrem iného znamená, že medzi každou dvojicou vrcholov (chatrčí) vedie práve jedna cesta. Keď tade cestuje druid s ťažkou batožinou, spôsobuje mu to veľké nepohodlie. No a my sme sa snažili minimalizovať celkové nepohodlie všetkých druidov. Ďalší slide.

Keď si za miesto konania sa sympózia zvolíme chatrč u, nepohodlie jednotlivých druidov môžeme vypočítať jedným prehľadávaním (DFS alebo BFS) z vrcholu u. Potrebujeme totiž len zistiť vzdialenosti jednotlivých vrcholov od u.

Na jedno prehľadávanie potrebujeme lineárny čas. Ak by sme teda za u volili postupne každú z chatrčí, našli by sme riešenie v čase $O(n^2)$. Ďalší slide.

Pozrime sa, ako sa zmení celkové nepohodlie, ak presunieme sympózium z vrcholu u do jeho suseda v vzdialeného d. Niektorým druidom to bude bližšie (o d), ostatným ďalej (tiež o d), no dôležitý nie je ich počet, ale súčet hmotností ich batožín. Nepohodlie jedného druida sa totiž zmení o hmotnosť jeho batožiny vynásobenú vzdialenosťou d.

Písmenom V označme súčet hmotností batožín tých druidov, ktorí to budú mať po zmene miesta sympózia bližšie; písmenom U označme súčet hmotností zvyšných batožín. Celkové nepohodlie sa potom presunom sympózia z u do v zväčší o $d \cdot (U - V)$. Ďalej.

Na tomto vzťahu založíme efektívnejšie riešenie našej úlohy. Najprv strom zakoreníme – niektorý z jeho vrcholov prehlásime za koreň (napríklad vrchol 1). Prehľadávaním do hĺbky z koreňa vypočítame súčet hmotností batožín pre každý z podstromov a tiež celkové nepohodlie v prípade, keď by sa sympózium konalo v koreni.

Potom spustíme z koreňa ďalšie prehľadávanie. V ňom už budeme počítať celkové nepohodlie pre zvyšné možnosti umiestnenia sympózia. Zakaždým, keď sa presunieme z vrcholu s už známou hodnotou celkového nepohodlia do jeho syna, na výpočet novej (neznámej) hodnoty použijeme vzťah, ktorý sme si už skôr odvodili.

Toto riešenie má časovú zložitosť $O(n)$, lebo potrebujeme len dve prehľadávania stromu. Pamäťová zložitosť je tiež $O(n)$.

Brat Zelený mi už naznačuje, že mám končiť. Ešte raz vás teda vítam a prajem vám veľa šťastia a dobrú kondičku. Nasleduje hra s názvom Bezbrehé behanie z kopca na kopec. *(búrlivý potlesk obecenstva)*“

Listing programu:

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

int n;
vector<int> W;  // batozina
vector<int> S;  // sucet hmotnosti batozin v podstrome
vector<vector<pair<int, int> > > G;  // (ciel, dlzka)
long long vysledok = 0;

void rataj_batozinu(int vrchol, int rodic, int vzdialenost) {
    vysledok += (long long) vzdialenost * W[vrchol];
    S[vrchol] = W[vrchol];
    for (int i = 0; i < (int) G[vrchol].size(); ++i)
        if (G[vrchol][i].first != rodic) {
            rataj_batozinu(G[vrchol][i].first, vrchol, vzdialenost + G[vrchol][i].second);
            S[vrchol] += S[G[vrchol][i].first];
        }
}

void rataj_nepohodlie(int vrchol, int rodic, long long nepohodlie) {
    vysledok = min(vysledok, nepohodlie);
    for (int i = 0; i < (int) G[vrchol].size(); ++i)
        if (G[vrchol][i].first != rodic)
            rataj_nepohodlie(G[vrchol][i].first, vrchol, nepohodlie +
                (long long) G[vrchol][i].second * (S[0] - 2 * S[G[vrchol][i].first]));
}
```

```
int main() {
    cin >> n;
    W.resize(n);
    for (int i = 0; i < n; ++i)
        cin >> W[i];
    G.resize(n);
    for (int i = 0; i < n - 1; ++i) {
        int a, b, d;
        cin >> a >> b >> d;
        --a, --b;
        G[a].push_back(pair<int, int>(b, d));
        G[b].push_back(pair<int, int>(a, d));
    }

    S.resize(n);
    rataj_batozinu(0, -1, 0);
    rataj_nepohodlie(0, -1, vysledok);

    cout << vysledok << endl;
}
```

opravoval Žaba

7. Obecná pošta

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

To čo vás táto úloha mala naučiť bolo, že sa nemáte zastrašiť škaredo vyzerajúcim zadaním, ale je treba sa zamyslieť, nakresliť nejaké tie obrázky a takýmto spôsobom postupne odhaliť podstatu príkladu, ktorá je mnohokrát oveľa jednoduchšia ako sa na prvý pohľad zdá. Napr. pôvodne som túto úlohu počul takto: „Máš zadaný graf a jeho kostru, v tvare cesty, vyber čo najmenej hrán tak aby bol výsledný graf dvojsúvislý“. Uznáte, že moje zadanie bolo trošku miernejšie.

Prvé čo chceme spraviť je preznačiť si čísla domov, do nejakého rozumného poradia, takže najlepšie od 0 po n podľa toho, koľkí v poradí na ceste je. Mnohí z vás na to použili mapu, alebo dokonca unordered mapu, to je však úplne zbytočné, keďže čísla domov boli len do 100000. Stačí si spraviť pole takej veľkosti a následne si pamätať, že na políčku s indexom i je j-ty dom v poradí.

Teraz, keď sme si už domy očíslovali rozumnejšie, poďme stavať cesty. Ako prvú musíme spraviť cestu z domu 0 do nejakého iného. Kebyže to nespravíme, tak pod odstránení cesty $0->1$, by sme sa z pošty nevedeli dostať nikam inam. Máme teda zopár možností na postavenie takejto cesty. Chceme ukázať, že nič nepokazíme tým, že postavíme cestu, ktorá vedie do najvzdialenejšieho domu.

Predstavme si, že máme nejaké optimálne riešenie, v ktorom cesta z domu 0 vedie do domu i. Ale existuje aj cesta do domu j kde $i < j$. Je ľahko nahliadnuť, že sa nič nepokazí ak hranu $0->i$ nahradíme hranou $0->j$.

Keď teda pridáme hranu $0->j$ stane sa nám to, že úsek domov 0 až j spĺňa vlastnosť zo zadania. Teda po odstránení ľubovoľnej hrany $k->k+1$, $k < j$ sa stále vieme dostať všade, do domov $\leq k$ pomocou starých ciest a do zvyšných pomocou našej pridanej hrany.

Teraz sme v situácii, že všetky domy 0 až j sa tvária ako jeden „veľký“ dom a ten chceme pripojiť k zvyšným domom. To znamená, že chceme pridať nejakú cestu, ktorá vedie z „veľkého“ domu do zvyšných, čo znamená, že hľadáme takú cestu $i->k$, že $i \leq 0$ a k je čo najväčšie, lebo stále platí rovnaká greedy stratégia.

Takto sa nám postupne zväčšuje náš „veľký“ dom, až nakoniec bude obsahovať všetky domy. V tom prípade máme najmenší možný počet hrán, lebo sme vždy vybrali hranu, ktorá nám to určite nepokazila.

Túto myšlienku naimplementujeme nasledovne: Najskôr si pomocou poľa prečíslujeme domy a následne si pre každý dom zapamätáme, kam najďalej z neho vedie cesta. Keďže vždy vyberáme najdlhšiu cestu, zvyšné nebudeme potrebovať. Najskôr pridáme hranu, ktorá vedie z 0. Dostali sme interval $(0, j>$, tak sa pozrieme na všetky tieto domy a zapamätáme si, kam najďalej sa z nich vieme dostať, nech je to nejaké k. Teraz si treba uvedomiť, že už nepotrebuujeme prezerať celý interval $(0, k>$ ak chceme nájsť novú cestu, ktorú chceme pridať. Lebo

v intervale $(0, j>$ viedla najdlhšia cesta len do domu k a my chceme nejaký väčší, takže stačí prezrieť interval $(j, k>$. Skončíme, keď k bude rovné $n-1$, teda číslu poslednému domu.

Keďže sa na každý dom pozrieme najviac raz, tak nám to bude trvať $O(n)$, ale keďže musíme aj načítať celý vstup, tak výsledná časová zložitosť je $O(n+m)$. Pamätáme si len n hrán - pre každý dom číslo najvzdialenejšieho a plus pole, ktoré používame na prečíslovanie, to má však veľkosť 100000 čo je rádovo n, takže pamäťová zložitosť je $O(n)$.

Listing programu:

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

int main()
{
  int n;
  scanf("%d ",&n);
  vector<int> Precislovanie; Precislovanie.resize(100047);
  for(int i=0; i<n; i++)
  {
    int x;
    scanf("%d ",&x);
    Precislovanie[x]=i;
  }
  int m;
  scanf("%d ",&m);
  vector<int> Max; Max.resize(n+42);
  for(int i=0; i<m; i++)
  {
    int x,y;
    scanf("%d %d ",&x,&y);
    Max[Precislovanie[x]]=max(Max[Precislovanie[x]],Precislovanie[y]);
    Max[Precislovanie[y]]=max(Max[Precislovanie[y]],Precislovanie[x]);
  }
  int koniec=0,vys=0,maxi=0;
  for(int i=0; i<n; i++)
  {
    maxi=max(maxi,Max[i]);
    if(koniec==i) {vys++; koniec=maxi;}
  }
  printf("%d\n",vys-1);
return 0;
}
```

opravoval MišoF

3. Zaplav to! (vzorák 3 z 3)

(max. 12 b za popis, 0 b za program)

Tak. Sem sme schovali poslednú, tretiu časť tohto prerasteného vzorového riešenia. Táto má podtitul „výlet do zoologickej záhrady“. Ukážeme si pár besných zvieratiek. Presnejšie teda skúsime aspoň približne vypočítať, ako že sa to správa tá veľkosť najväčšej jednofarebnej miestnosti. Ďalej čítate len na vlastné nebezpečenstvo :-)

Matematický pohľad na úlohu

Experimentálne merania sú fajn, ale omnoho lepšie je vedieť matematicky zdôvodniť, že sa daná vec naozaj správa tak, ako sme odpozorovali. Napríklad sa môžeme pokúsiť povedať niečo o pravdepodobnosti toho, že sa nám niekde na hracom pláne objaví aspoň s-políčková jednofarebná oblasť.

Ľahko vieme zodpovedať jednoduchšiu otázku. Predstavme si, že nám už niekto vyznačil konkrétnu s-políčkovú oblasť. Aká je pravdepodobnosť, že bude celá jednej farby? Tu vieme odpoveď presne: je to $1/f^{s-1}$. To preto, že spomedzi všetkých možných f^s ofarbení našej oblasti je len f jednofarebných. (Inými slovami, keď políčka našej oblasti ofarbujem postupne, prvé môžem ofarbiť ľubovoľne a pre každé zo zvyšných $s-1$ mám pravdepodobnosť $1/f$ že mu vygenerujem tú istú farbu ako prvému.)

Teraz nás bude zaujímať počet spôsobov $\varphi(n, s)$, ktorými sa na pláne $n \times n$ dá vyznačiť súvislá s-políčková oblasť. (Pre pevne zvolené n a s budeme namiesto $\varphi(n, s)$ písať len φ.)

Od φ zjavne závisí hľadaná pravdepodobnosť, že aspoň jedna oblasť veľkosti s bude celá jednofarebná. Povedať presný matematicky vzťah popisujúci túto závislosť je však veľmi ťažké.

(Možná chyba: Na tomto mieste by mnohí prehlásili: „Ale veď to je jednoduché: pre konkrétnu oblasť je pravdepodobnosť $p = 1/f^{s-1}$, že nebude jednofarebná. A teda pravdepodobnosť toho, že žiadna z φ oblastí nebude jednofarebná, je p^{φ}.“ V tejto úvahe je chyba. Niektoré dvojice možných oblastí sa totiž čiastočne prekrývajú, a teda nejde o nezávislé javy a nemôžeme len tak násobiť ich pravdepodobnosti.)

Skúsme teda hľadanú pravdepodobnosť aspoň nejak odhadnúť. Na jeden odhad môžeme použiť trik nazývaný *lineárnosť strednej hodnoty*. Celé si to predstavíme nasledovne: Zamestnáme φ trpaslíkov, zoženieme si kopec cukríkov a jedno veľké vrece. Každý trpaslík si vyberie jednu z φ s-políčkových oblastí a bude sa na ňu pozerať. Teraz začneme generovať hracie plány. Zakaždým, keď vygenerujeme hrací plán, tak každý trpaslík, ktorého oblasť je na ňom jednofarebná, hodí do vreca cukrík.

Koľko bude vo vreci cukríkov po ω pokusoch? Pozerajme sa na jedného konkrétneho trpaslíka. Ten má v každom pokuse pravdepodobnosť $p = 1/f^{s-1}$, že tá jeho konkrétna oblasť bude jednofarebná. Teda do vreca hodí cukrík *v priemere* raz za f^{s-1} kôl. Tento trpaslík teda do vreca dokopy prispeje *v priemere* ω/f^{s-1} cukríkmi. No a rovnakú úvahu môžeme spraviť pre každého iného trpaslíka. Tým dostávame, že dokopy bude po ω pokusoch vo vreci *v priemere* $\varphi\omega/f^{s-1}$ cukríkov.

Nech teraz q je pravdepodobnosť toho, že sa na náhodnom hracom pláne objaví aspoň jedna jednofarebná s-políčková oblasť. Čo o nej vieme povedať? Keď spravíme ω pokusov, dajú trpaslíci určite *v priemere* do vreca *aspoň* $q\omega$ cukríkov. (Slovo „aspoň“ je tam preto, že v každom pokuse, v ktorom sa taká oblasť zjaví, dajú spolu trpaslíci do vreca *aspoň* jeden cukrík. Môže ich byť aj viac, ak sa zjavila oblasť väčšia ako s alebo dokonca viac veľkých miestností.)

Teraz vieme zhora odhadnúť q. Priveľké q totiž vieme vylúčiť na základe toho, že by do vreca pribúdalo viac cukríkov ako má. Nutne teda musí platiť nasledovná nerovnosť: $q\omega \leq \varphi\omega/f^{s-1}$. Tým sme teda dokázali horný odhad pre hľadanú pravdepodobnosť q: ak zvládneme spočítať počet oblastí φ, budeme vedieť, že $q \leq \varphi/f^{s-1}$. A vlastne nepotrebujeme φ poznať presne, stačí túto hodnotu rozumne tesne zhora odhadnúť.

Každý, kto sa dočítal až sem, povinne teraz hneď napíše na KSP fórum (do časti *Fórum / KSP / Úlohy* do threadu *Zoznam frajerov*) že je _____ frajer. Nech máme štatistiku. Navyše: Ak si porozumel(a) všetkému, čo sa tu doteraz písalo, tak za _____ zvoľ prídavné meno, ktoré ešte nebolo v threade použité a má „u“ alebo „ú“ ako prvú samohlásku (teda napr. „kuchynský frajer“). Ak si sa už stratil(a) a niečo nedávalo zmysel, použi ako prvú samohlásku „y“ alebo „ý“ (napr. „mydlový frajer“). Je povolené a odporúčané kruto sa vysmievať každému, kto nie je frajer, len sa v threade priživuje a za frajera pretvaruje.

Hodnotu $\varphi(n, s)$ vieme vypočítať nasledovne: nájdeme všetky polyominá[6] veľkosti s a pre každé z nich zistíme počet spôsobov, ktorými sa dá umiestniť do mriežky $n \times n$. A tento počet je zjavne pre každé polyomino nanajvýš rovný n^2. Ak si teda počet polyomín veľkosti s označíme $\Psi(s)$, zjavne platí $\varphi(n, s) \leq n^2 \cdot \Psi(s)$.

Je známe, že počet polyomín $\Psi(s)$ veľkosti s rastie v závislosti od s exponenciálne. Teda existuje nejaká konštanta ψ taká, že funkcia $\Psi(s)$ rastie asymptoticky rovnako rýchlo ako funkcia ψ^s.

Uvedomte si, že táto skutočnosť nie je zjavná. Napríklad taká funkcia $n!$ rastie *rádovo rýchlejšie* ako všetky exponenciálne funkcie. A prečo by nemal počet polyomín byť až rádovo rovný n faktoriál?

(Tu by vám mohla intuícia napovedať niečo typu: Keď vyrábam polyomino veľkosti n, zoberiem ľubovoľné polyomino veľkosti $n-1$ a k jednému z jeho $n-1$ políčok prilepím susedné.

[6] Presnejšie, tzv. fixné polyominá, keďže záleží na rotácii. Napr. pre $s = 5$ existuje 63 fixných pentomín, každé z nich vznikne vhodnou rotáciou a možno preklopením jedného z 12 rôznych tvarov.

Pomocou takejto úvahy by sa dalo dokázať, že polyomín veľkosti n je *nanajvýš rádovo* $n!$. Ukáže sa ale, že to isté polyomino veľkosti n sa dá vyrobiť veľa rôznymi spôsobmi, takže všetkých polyomín bude rádovo menej ako $n!$.)

Presná hodnota ψ nie je známa. Doteraz sa vedcom o nej podarilo dokázať len to, že naozaj existuje, a že jej hodnota spĺňa nerovnosti $3.98 < \psi < 4.64$. Konkrétne dôkazy si nebudeme ukazovať, ale pre názornosť si ukážeme dôkazy slabšieho dolného aj horného odhadu, aby ste mali predstavu, ako na to.

Dolný odhad počtu polyomín

Keď chceme zdola odhadnúť počet polyomín, potrebujeme ukázať nejaký spôsob, ako ich vyrobiť veľa tak, aby boli zaručene všetky navzájom rôzne.

Uvažujme napríklad nasledujúci spôsob výroby: Zoberiem nekonečnú štvorcovú sieť, postavím sa na políčko $(0,0)$ a z neho sa $(s-1)$-krát pohnem buď doprava alebo dohora. Takto určite navštívim presne s políčok. A navyše rôznym voľbám cesty zodpovedajú rôzne s-tice políčok. Preto existuje aspoň 2^{s-1} rôznych polyomín veľkosti s.

(Všetky polyominá, ktoré sme práve popísali, majú tvar „dlhého hada“. Zjavne je aj veľa úplne ináč vyzerajúcich. Viete dokázať nejaký lepší dolný odhad ako 2^{s-1}?)

Horný odhad počtu polyomín

Horný odhad hodnoty ψ môžeme spraviť nasledovne:

Uvažujme nasledovný algoritmus. Zoberiem nekonečnú štvorcovú sieť, postavím sa na políčko $(0,0)$ a z neho spustím prehľadávanie do hĺbky, ktoré sa počas behu $(s-1)$-krát „vnorí hlbšie“ (teda prejde na nejaké dovtedy nenavštívené políčko) a tiež $(s-1)$-krát sa vynorí späť. Keď teda tento algoritmus skončí, máme ofarbené nejaké s-políčkové polyomino.

Zjavne pre každé konkrétne fixné polyomino existuje aspoň s rôznych priebehov nášho algoritmu, ktoré ho nakreslia. (Zväčša je ich dokonca výrazne viac. Tých zaručených s, ktoré máme na mysli, sa líši voľbou začiatočného políčka.)

Ak chceme zhora odhadnúť počet polyomín, stačí nám zhora odhadnúť počet rôznych priebehov vyššie popísaného algoritmu.

V prvom rade sa pozrime na to, kedy sa prehľadávanie vnára hlbšie a kedy sa vynára. Keď si zapíšeme kroky „hlbšie“ ako ľavé zátvorky a kroky, kedy sa z rekurzie vynárame, ako pravé zátvorky, dostaneme dobre uzátvorkovaný výraz s $s-1$ pármi zátvoriek. Počet dobre uzátvorkovaných výrazov udávajú Catalanove čísla: je ich presne $\binom{2s-2}{s-1}/s$. No a zjavne platí veľmi voľný horný odhad:

$$\frac{\binom{2s-2}{s-1}}{s} \leq \binom{2s-2}{s-1} \leq 2^{2s-2} = 4^{s-1}$$

No a ešte musíme každej ľavej zátvorke priradiť smer pohybu. Pre prvú (úplný začiatok prehľadávania) máme 4 možnosti, pre každú ďalšiu už len najviac 3 (lebo nejdeme smerom, z ktorého sme práve prišli). Dokopy teda vieme ľavým zátvorkám priradiť pohyby nanajvýš $4 \cdot 3^{s-2}$ spôsobmi. (Pravým zátvorkám pohyby priradzovať netreba, lebo predsa vždy vieme, kam sa práve vraciame.)

Dostávame teda nasledovný odhad: Vyššie popísaný algoritmus má nanajvýš $4^{s-1} \cdot 4 \cdot 3^{s-2} < 12^s$ možných priebehov. Niektoré tieto priebehy urobia blbosti a pobehajú samé po sebe, takže nakreslia oblasť s menej ako s políčkami. Ale určite je pre každú súvislú s-políčkovú oblasť medzi priebehmi nášho algoritmu aspoň s rôznych takých, pri ktorých vznikne tá konkrétna oblasť. Preto platí $\Psi(s) < 12^s$, a teda $\psi < 12$.

(Aj tento náš horný odhad je veľmi voľný. Vidíš nejaký spôsob, ako ho spresniť?)

Závery pre našu úlohu

Na `http://oeis.org/A001168` sa okrem iného môžeme dočítať, že $\Psi(21) = 88\,983\,512\,783$. Keď si vyčíslime 4.64^21, dostaneme číslo vyše 1000-krát väčšie. Nedopustíme sa teda veľkej nepresnosti, ak povieme, že pre $s \geq 21$ platí $\Psi(s) < 4.64^s/1000$.

Pripomeňme si, že sme si dokázali nasledovný horný odhad: Nech q je pravdepodobnosť toho, že na pláne $n \times n$, ofarbenom f farbami, uvidíme aspoň jednu aspoň s-políčkovú jednofarebnú oblasť. Potom pre q platí $q \leq n^2\Psi(s)/f^{s-1}$.

Po dosadení horného odhadu pre $\Psi(s)$ dostávame pre $s \geq 21$ nasledovný odhad: $q \leq (n^2 f/1000) \cdot (4.64/f)^s$.

Tento odhad je zjavne úplne nanič pre $f \leq 4$, keďže na s-tú budeme umocňovať hodnotu väčšiu ako 1. Ale pre väčšie počty farieb začína byť tento odhad celkom schopný.

Uvažujme napr. $n = 10000$ a $f = 8$. Čo nám o takýchto hracích plánoch povie náš vzorec? Vyskúšajme si dosadiť rôzne hodnoty s:

Pre $s = 25$ dostávame $q < 97.46\%$.
Pre $s = 26$ dostávame $q < 56.53\%$.
Pre $s = 30$ dostávame $q < 6.40\%$.
Pre $s = 35$ dostávame $q < 0.42\%$.
Pre $s = 50$ dostávame $q < 0.00012\%$.

Teda slovne: keby sme generovali veľmi veľa hracích plánov s týmito n a f, určite by sme plány, na ktorých najväčšia oblasť dosiahne aspoň 35 políčok, videli veľmi zriedka – ani nie raz za 200 pokusov. A keďže náš horný odhad q je zrejme voľný, v skutočnosti to bude ešte zriedkavejšie. Keďže pravdepodobnosť prekročenia čísla 26 je dokázateľne nanajvýš zhruba 50%, medián meraní určite túto hodnotu nemôže prekročiť.

Pre porovnanie samozrejme výsledky experimentu: vygenerovali sme si $k = 100$ hracích plánov s týmito n a f. Medián bol 19, priemer okolo 19.5. Najmenšie namerané veľkosti oblasti boli 17 (2×) a 18 (20×), najväčšie boli 24 (1×) a 25 (1×). A to je celkom blízko toho, čo nám zaručoval náš dôkaz.

Zdôvodnenie hypotéz

V predchádzajúcom riešení sme dospeli k empirickému pozorovaniu, že veľkosť najväčšej jednofarebnej oblasti je zhruba priamo úmerná $\log n$ a zhruba nepriamo úmerná $\log f$. Dokázať to síce nevieme, ale vieme si to aspoň bližšie zdôvodniť – teda aspoň pre tie f, pre ktoré je náš horný odhad použiteľný.

Predpokladajme, že náš odhad je aspoň rádovo tesný – teda že pre nejaké vhodné konštanty c a ψ platí $q \sim (n^2 f/c) \cdot (\psi/f)^s$. Pre jednoduchosť budeme namiesto $\sim$ písať $=$.

Položme $q = 1/2$. Hľadáme teraz tie trojice (n, f, s), pre ktoré pravdepodobnosť výskytu s-políčkovej oblasti vychádza na 50%. Inými slovami, tie trojice (n, f, s), pre ktoré je s mediánom nameraných výsledkov pre (n, f).

Dostávame teda $(f/\psi)^s = n^2 f(2/c)$. Vyjadríme z tohto vzťahu s: zlogaritmujeme obe strany a upravíme. Dostaneme:

$$(\log f - \log \psi)s = 2\log n + \log f + \log(2/c)$$
$$s = \frac{2\log n + \log f + \log(2/c)}{\log f - \log \psi}$$

Z posledného vzťahu už je zjavné, že keď počítame s ako funkciu n, porastie s priamo úmerne hodnote $\log n$. A ak sa naň dívame ako na funkciu f, môžeme ho ľahko upraviť do tvaru $s = 1 + \mu/(\log f - \log \psi)$ pre nejakú vhodnú konštantu μ. A teda s je približne nepriamo úmerné hodnote $\log f$.

A to už je na dnes naozaj všetko. Dobrú noc!

opravoval JaNo

8. Obozretný tukan

(max. 17 b za popis, 8 b za program)

Hra, ktorú Tukan a Holub hrajú, je *kombinatorická*. Ak chcete vedieť, čo to znamená, pozrite si vzorák 7. úlohy minulej série.

Pozícia v tejto hre je jednoznačne určená veľkosťou kopy.

Keď chceme zistiť, ktoré z pozícií sú výhrávajúce a ktoré prehrávajúce, pomôže nám vedieť, že vyhrávajúce sú tie, z ktorých sa vieme nejakým ťahom dostať do prehrávajúcej. Všetky ostatné pozície sú prehrávajúce. Z toho je napríklad jasné, že pozície, z ktorých nevedie ťah sú tiež prehrávajúce.

Keď úž máme túto predstavu, pomerne jednoducho napíšeme program, ktorý povie, koľká pozícia je aká. Stačí napríklad cyklom prejsť cez všetky pozície od práznej kopy, až po kopu veľkosti N.

Pre každú pozíciu sa pozrieme na všetký pozície, do ktorých vieme ťahať. Keďže ťahaním kopu zmenšíme, tak o všetkých týchto pozíciách už vieme, aké sú. Ak je aspoň jedna z pozícií, do ktorých sa vieme dostať, prehrávajúca, označíme našu aktuálnu pozíciu vyhrávajúcou, inak bude prehrávajúca.

Takýmto spôsobom zistíme v čase $O(KN)$ koľko pozícií je vyhrávajúcich a teda odpoveď na našu úlohu. Toto riešenie je však príliš pomalé.

Vzorák

Všimnime si, že keď $o_i \leq 20$, tak moja pozícia je ovplyvňovaná len poslednými 20 políčkami. Vo všeobecnoti, nech M je maximum z o_i. Potom je pozícia ovplyvňovaná len M políčkami pred ňou. Preto, ak sa nejaká M-tica políčok vyskytne viac krát, tak po oboch vyskytnutiach nasledujú rovnaké pozície. Napríklad ak $M = 3$ a po trojici 101 nasleduje 1, tak je jedno, kde sa tá trojica nachádza, po každom výskyte 101 bude nasledovať jednotka. Navyše, keďže možných M-tíc je najviac 2^M, tak z Dirichletovho princípu vyplýva, že v prvých $2^M + 1$ M-ticiach sa určite nejaká zopakuje.

Potom teda existujú čísla A a B, $0 \leq A < B$ a $A + B \leq 2^M$ (V praxi sú A a B rádovo menšie než 2^M, nám je to však skoro jedno) také, že postupnosť pozícii vyzerá nasledovne. Začína nejakým úsekom dlhým A a za ním sa nekonečne veľa krát opakuje úsek dlhý B. Hovoríme tomu, že postupnosť je periodická s periódou B a predperiódou A.

Ak poznáme čísla A, B a máme vypočítané pozície do $A+B$, tak vieme ľahko povedať koľko z pozícií $1..N$ je vyhrávajúcich a teda riešiť úlohu. Nech $P[x]$ je počet vyhrávajúcich pozícií od 1 po x.

Potom

$$P[N] = P[kB + Z] = P[k(B - Z + Z) + Z] = k \cdot (P[Z + B] - P[Z]) + P[Z]$$

pričom $k = (N - Z)/B$ a $Z = ((N - A) \; mod \; B) + A \Rightarrow Z < A + B$, rozmyslite si, prečo to funguje.

No a $P[N]$ je riešenie našej úlohy, takže sme už skoro skončili.

Potrebujme však nejaký rozumný a rýchly spôsob, ako zistiť, kedy sa nám postupnosť zacyklila – teda kedy sa nám vyskytne nejaká 20tica druhýkrát. Rouzmný a priamočiari spôsob je vytvoriť si pole intov, kde si pamätáme, ktoré M-tice sa už vyskytli a kde sa vyskytli. Ak nájdeme M-ticu, ktorá sa už vyskytla, tak A bude pozícia tej prvej a B bude rozdiel ich pozícií.

V čase $O(K2^M)$, si vieme vypočítať prvých dosť pozícií.

Vzorák plusplus

Keď chceme náš program ešte trocha urýchliť, tak môžeme spraviť nasledovný trik. M nasledujúcich pozícií zakódujeme ako jedno celé číslo. Toto číslo bude mať v binárnom zápise jednotku na i-tej pozícii práve vtedy, keď na i-tom mieste od konca M-tice je vyhrávajúca pozícia.

Nasledujúcu M-ticu vypočítam ako `((predosla<<1) & range) | bool((~predosla) & mask)`.

Operáciou `((predosla<<1) & range)` posuniem predošlú pozíciu o 1 doľava a zahodí najstarší bit. Následne sa na najnovší bit zapíše 1 ak je moja pozícia vyhrávajúca, inak sa zapíše 0. Keď mask je ďalší binárny vektor taký, že má jednotku na $(o_i - 1)$-tom mieste, pre $i \in \{1..K\}$ tak ho viem zANDovať s bitovou negáciou predošlého čísla a dostanem 1 iff je moja pozícia vyhrávajúca. Neveríte? Vyskúšajte si to na papieri.

Týmto sa časová zložitosť zmenší na $O(2^M)$. Pamäťová zložitosť bude tiež $O(2^M)$.

Listing programu:

```
//Fruit of Light
//FoL CC
//Apple Strawberry

#include<cstdio>
#include<algorithm>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)
typedef long long ll;

ll N;
int K,mask,range,a, was[1<<20], cnt[1<<20];

int next_v(int state){
    return ((state<<1) & range) | bool((~state) & mask);
}

int main(){
    scanf("%lld %d",&N, &K);
    range = (1<<20)-1;
    For(i,K) {
        scanf("%d", &a);
        mask |= (1<<(a-1));
    }
    int state = 0;
    int T = 0;
    while(++T<=N){
        state = next_v(state);
        cnt[T] = cnt[T-1]+state%2;
        if (was[state]) break;
        was[state] = T;
    }
    ll P = T - was[state], PP = was[state];
    printf("%lld\n", (N-PP)/P*(cnt[T]-cnt[PP]) + (cnt[(N-PP)%P+PP]));
}
```


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