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

Vzorové riešenia 1. kola zimnej časti

opravuje Syseľ
(max. 0 b za popis, 10 b za program)

1. Zwarte Doos

Level 1

Po pár pokusoch si bolo možné všimnúť, že k vstupnému číslu sa vždy pripočítalo číslo 1. Na dosiahnutie čísla 47 teda stačilo zadať 46.

Level 2

Tu sa zobralo vstupné číslo a prevrátilo sa v ňom poradie cifier. Číslo 123 získame pomocou 321.

Level 3

Tento level počítal ciferný súčet vstupu. Keďže $42 = 6 \cdot 7$, mohli ste napríklad zadať 777777.

Level 4

Fakt, že máte dosiahnuť OK, mohol byť mierne odstrašujúci. V skutočnosti vám však skrinka odpovedala, či má byť číslo väčšie, alebo menšie.

Pre človeka bolo najjednoduchšie najskôr nájsť správnu dĺžku výsledného čísla a potom postupne hľadať jednotlivé cifry. Výsledok nebol zvolený úplne náhodne. Bolo to prvých 8 desatinných miest čísla π: 14159265.

Ak by sme však chceli minimalizovať počet pokusov, mohli by sme skúsiť ešte efektívnejší postup. Najskôr nájdeme prvé minimum a maximum. Minimum môže byť napríklad nula a maximum prvé číslo, ktoré nám už bude vypisovať odpoveď menšie. Tieto nám určia nejaký interval, v ktorom sa hľadané číslo môže nachádzať. Spýtame sa presne na stred tohto intervalu. Nech nám krabička odpovie akokoľvek, práve sme zmenšili interval možných odpovedí na polovicu (ak sme, náhodou, číslo neuhádli). Opakovaním tohto postupu sa rýchlo dopracujeme k výsledku. V informatike sa táto metóda nazýva binárne vyhľadávanie.

Level 5

Prvá vec, ktorú ste si mohli všimnúť, je, že skrinka zachováva počet cifier. Prvú cifru nikdy nemení. Druhú vždy zvýši o 1, tretiu zvýši o 2, atď. Ale ak je napríklad druhá cifra 9, tak z nej spraví 0, takže číslice sa cyklicky točia. Stačilo preto 25252525 cyklicky posunúť späť, čím získame 24028068.

Level 6

Predstavte si, že zoberiete všetkých 26 písmen anglickej abecedy a poskladáte z nich všetky možné slová. Tieto slová potom zoradíte. Najskôr podľa dĺžky a potom podľa abecedy. Takto získate postupnosť, ktorej prvky vám vracala šiesta krabička.

Na začiatku je prázdne slovo dĺžky 0. Nasleduje 26 samostatných písmen dĺžky 1. Potom sú všetky dvojice zoradené abecedne, atď. Niečo podobné využívajú tabuľkové procesory (napr. Microsoft Excel alebo LibreOffice Calc) na označenie stĺpcov.

Jeden spôsob, ako nájsť medzi týmito slovami $PAPAGAJ$-a, je postupovať podobne ako v leveli 4. Vieme totiž povedať, či slovo, ktoré sme dostali, je pred alebo za $PAPAGAJ$-om.

Druhá, rýchlejšia možnosť je toto číslo vypočítať. Najskôr musíme spočítať všetky menejpísmenkové slová. Prázdne slovo je jedno, jednopísmenkových je 26, dvojpísmenkových $26 \cdot 26$ a n-písmenkových je 26^n. Spolu je to teda $26^0 + 26^1 + 26^2 + \cdots + 26^6$ slov. Potom ešte musíme pripočítať všetky 7 písmenové slová, ktoré začínajú niečím pred P $(15 \cdot 26^6)$, všetky slová začínajúce na PA ktoré majú tretie písmeno pred P $(15 \cdot 26^4)$ a obdobné slová pre G a J $(6 \cdot 26^2$ a $9 \cdot 26^0)$. Po sčítaní dostaneme 4961867752 čo je počet slov pred slovom $PAPAGAJ$ a teda aj jeho poradie.

Level 7

Nech napíšete takmer čokoľvek, výsledok je 101010. A máte nájsť *POKLAD*! Čo to má znamenať? Robia si z nás snáď srandu? Alebo že by 101010 niečo znamenalo? Nemôže to byť stopa na ceste k pokladu? Neskúsime ho zadať? BINGO! Získali sme ďalšiu stopu. Ak zadáme ju a postup zopakujeme ešte 20 krát, dostaneme sa k pokladu.

Wait! What? Má to aj nejaký hlbší zmysel? Samozrejme! Ak si totiž cestu k pokladu zaznačíte do mapy (na číselnú os), zistíte, že neskáče len tak náhodne, ale vždy skočí o polovicu toho, čo skákala predtým. A vždy zmení smer. Presnejšie povedané, v n-tom kroku skočí o $(-2)^{20-n}$ dopredu (kroky číslujeme od 0). Znamená to, že musí skončiť približne v dvoch tretinách prvého skoku. Na domácu úlohu si rozmyslite prečo! Alebo si to aspoň nakreslite.

Level 8

Veľmi dobrý testovací vstup je napríklad 8887777. Vráti nám 3847. Čo to má znamenať? No presne to, čo ste napísali. Tri osmičky a štyri sedmičky. Takže 123456 nám vlastne hovorí, že správny vstup má byť jedna dvojka, tri štvorky a päť šestiek. Teda 244466666.

Level 9

Čas. Výborne. Dimenzia, ktorá nám chýbala. Vstup 0 vypíše aktuálny čas.

Nápad: Submitnúť o 4:13 ráno. Zamietnutý. (Či?)

Pozorovanie: číslo na vstupe vynásobené 47 je počet minút, ktorý sa k aktuálnemu času pripočíta.

Nápad: Nájsť najbližší čas pred 4:13, ku ktorému vieme pričítať nejaký jednoduchý násobok 47, a počkať tých najviac 47 minút. Zamietnutie závisí od jedinca[1] a aktuálneho času.

Jedinec, ktorý predchádzajúci nápad zamietol, si mohol uvedomiť, že nie je špecifikovaný deň a teda sa vieme posunúť dopredu o viac ako 24 hodín. Takto sa vie k želanému času dostať bližšie a čakať menej. Alebo to môže vypočítať presne a potom zistiť, že sa medzičasom čas zmenil. Aj tak mu však patrí sláva.

Level 10

Konečne koniec. Čo si pre nás prichystal? Adama a Aničku z Alekšiniec. Pozorný riešiteľ si môže všimnúť, že máme k dispozícii 13 mužských mien, 11 ženských mien a 7 obcí. Tieto sa pri postupnom zvyšovaní vstupu neustále cyklia. A vďaka tomu sa postupne vystrieda všetkých 1001 možností.

Riešenie? Hrubá sila. Borisa dostaneme z 1 a potom z každého trinásetho čísla. Tak teda postupne pripočítavame 13. Keď to spravíme 2-krát (27), dostaneme Borisa a Filoménu z Gánoviec. Kombinácia muža a obce sa však opakuje iba raz za $13 * 7 = 91$ krokov. Teda pripočítavame 91, kým nedostaneme v strede Hanku. Je to pri 755.

Ak by vás zaujímal všeobecnejší spôsob ako riešiť podobné úlohy, kľúčový pojem je „Čínska zvyšková veta“.

Záver

Časom (pár týždňov po odovzdaní) sa tento text ešte doplní štatistikami ohľadne počtu pokusov pre jednotlivé úlohy. Dúfam, že ste sa pri riešení Zwarte Doos zabavili, vzoráky pochopili a že sa tešíte na Zwarte Doos 2.

2. Zaujímavé poháre

opravuje Rasťo
(max. 6 b za popis, 4 b za program)

V tomto vzorovom riešení si predstavíme viacero možných spôsobov, ako riešiť tento problém, a po ich porovnaní vyberieme ten najefektívnejší.

Pomerne jednoduchý spôsob je postupne skúšať, ktorý pohár nám chýba, a potom overiť, či súčet objemov ostatných pohárov je rovný v. Na spočítanie objemu zvyšných pohárov potrebujeme urobiť n operácií. A musíme to spraviť n krát, pre každý pohár, ktorý by mohol chýbať, takže urobíme $n \cdot n$ operácií. Časová zložitosť tohto algoritmu je teda $O(n^2)$[2].

Nedalo by sa toto riešenie zlepšiť? Musíme skutočne skúšať každý pohár? Začnime tým, že sčítame objem všetkých pohárov a tento súčet si označíme s. Objem pohára, ktorý nám chýba je zjavne $s - v$. Takto sme zlepšili zložitosť nášho programu na $O(n)$, keďže nám stačí sčítať n čísel.

[1] riešiteľa

[2] Ak ste ešte nikdy nepočuli o O-notácii, tak si o nej môžete niečo prečítať na stránke ksp.sk/riesenie/zlozitost

Všimnime si, že najpomalšia fáza v našom algoritme je vypočítať súčet objemov všetkých pohárov, čo je v podstate súčet $1+2+\cdots+n$. Pre takýto špeciálny súčet, ale existuje matematický vzorec. A na jeho odvodenie sa používa pekný matematický trik.

Napíšme si na papier postupnosť 1 až n a pod ňu ešte raz tú istú postupnosť, ale v obrátenom poradí.

$$\begin{array}{ccccccccccccc} 1 & + & 2 & + & 3 & + & \cdots & + & (n-2) & + & (n-1) & + & n \\ n & + & (n-1) & + & (n-2) & + & \cdots & + & 3 & + & 2 & + & 1 \end{array}$$

Všimnime si, že súčet v každom stĺpci je $n+1$. Takže súčet čísel v oboch riadkoch je $n(n+1)$. Avšak, každé číslo sme zarátali dvakrát, takže súčet jedného riadku je polovica celkového súčtu, t.j. $\frac{n(n+1)}{2}$. Takto vieme súčet objemov všetkých pohárov vypočítať v konštantnom čase dosadením do vzorca. Pamäťová zložitosť je tiež konštantná.

Listing programu (C++)

```
#include <cstdio>

int main() {
    long long N,V;
    scanf("%lld%lld",&N,&V);
    printf("%lld",(N * (N + 1)) / 2 - V);
}
```

Listing programu (Pascal)

```
program pohare;

var N,V : int64;

begin
    readln(N,V);
    writeln((N * (N + 1)) div 2 - V);
end.
```

3. Základ úspechu

opravuje Luxusko
(max. 5 b za popis, 5 b za program)

Najjednoduchšie riešenie, priamočiaro "hrubou silou" vyzerá nasledovne: Označme si reťazce fembotky a Helboja F a H. Vyskúšame všetky možné podreťazce H – postupným výberom začiatkov a koncov. Každý podreťazec po znakoch skontrolujeme, či sa také písmenko nenachádza aj v F – prejdením celého F. Ak tam bola aspoň jedna spoločná téma-písmenko, podreťazec zarátame. Časová zložitosť tohoto prístupu je $O(h^3 \cdot f)$ – máme $h(h+1)/2$ podreťazcov po najviac h písmen, z ktorých každý skontrolujeme za najviac f krokov.

Listing programu (C++)

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

int main() {
  int f, h;
  string F, H;
  cin >> f >> h;
  cin >> F >> H;
  long long pocet = 0;
  for (int i = 0; i < h; i++) {
    for (int j = i; j < h; j++) {
      for (int k = i; k <= j; k++) {
        bool spolocna = false;
        for (int l = 0; l < f; l++) if (H[k] == F[l]) { spolocna = true; break; }
        if (spolocna) { pocet++; break; }
      }
    }
  }
  cout << pocet << '\n';

  return 0;
}
```

Všimnime si, že netreba stále prechádzať celý reťazec F – namiesto toho vieme prejsť F na začiatku raz, pre každé písmenko si zapamätať, či je v ňom a následne to rýchlo kontrolovať pri prechádzaní. Dostaneme čas $O(f+h^3)$.

Listing programu (C++)

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

string F, H;
bool je_v_F[32]; // tabulka pre pismenka, v globalnych je default vsade 0/false

int main() {
  int f, h;
  cin >> f >> h;
  cin >> F >> H;
  for (int i = 0; i < f; i++) je_v_F[F[i]-'a'] = true;
  long long pocet = 0;
  for (int i = 0; i < h; i++)
    for (int j = i; j < h; j++)
      for (int k = i; k <= j; k++)
        if (je_v_F[H[k]-'a']) { pocet++; break; }
  cout << pocet << '\n';

  return 0;
}
```

Ďalej sa zamyslime, čo sa stane, keď v H pre daný začiatok úseku nájdeme koniec taký, že tento úsek už obsahuje spoločnú tému s F. Tento koniec môžeme ľubovoľne posúvať ďalej a stále budeme mať aspoň jednu túto istú spoločnú tému. Ako nám to pomôže? Pre každý možný začiatok nájdeme najbližšie písmenko, ktoré sa vyskytuje aj v F. Ak žiadne nenájdeme, tak zjavne nemáme podreťazec so spoločnou témou nikde ďalej. Nech začiatok je na pozícii i a najbližšie spoločné písmenko na $j >= i$. Všetkých podreťazcov začínajúcich v i obsahujúcich j je $h - j$ (ak indexujeme od 0 po $h - 1$). Takže tento príslušný počet zarátame a pohneme sa na ďalší začiatok. Začiatkov je h a pre každý prejdeme najviac h písmen, čo nám dáva časovú zložitosť $O(f + h^2)$.

Listing programu (C++)

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

string F, H;
bool je_v_F[32];

int main() {
  int f, h;
  cin >> f >> h;
  cin >> F >> H;
  for (int i = 0; i < f; i++) je_v_F[F[i]-'a'] = true;
  long long pocet = 0;
  for (int i = 0; i < h; i++){
    int prva_spol;
    for (prva_spol = i; prva_spol < h; prva_spol++) {
      if (je_v_F[H[prva_spol]-'a']) break;
    }
    pocet += h-prva_spol;
  }
  cout << pocet << '\n';
  return 0;
}
```

Nakoniec sa pozrime, čo sa deje pri hýbaní sa na ďalší začiatok úseku. Kde mohla byť predošlá najbližšia spoločná téma? Ak bola hneď na predošlom začiatku, tak je už za nami a musíme nájsť novú. Inak je stále najbližšia tá istá. Budeme mať premennú pre pozíciu najbližšej spoločnej témy a keď sa už začiatok ocitne za ňou, budeme ju zvyšovať, kým nenájdeme dalšiu spoločnú tému alebo koniec. Túto premennú zvýšime celkovo iba h-krát a každému začiatku vieme povedať, koľko príslušných podreťazcov z neho zarátame. Dostávame rýchle vzorové riešenie v $O(f + h)$.

Listing programu (C++)

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

string F, H;
bool je_v_F[32];

int main() {
  int f, h;
  cin >> f >> h;
  cin >> F >> H;
  for (int i = 0; i < f; i++) je_v_F[F[i]-'a'] = true;
  long long pocet = 0;
  int ns = -1; // najblizsia spolocna tema (pozicia)
  for (int i = 0; i < h; i++){
    if (ns < i) {
      for (ns = i; ns < h; ns++) if (je_v_F[int(H[ns]-'a')]) break;
    }
    pocet += h-ns;
  }
  cout << pocet << '\n';
```

```
  return 0;
}
```

Treba si dať pozor na veľkosť odpovede – môže presiahnuť rozsah 32-bitovej premennej.

Všimnite si, ako sme z jednoduchšieho riešenia zrýchľovali. Vždy vyhodíme niečo, čo zbytočne prechádzame. Takto vieme často nájsť použiteľné riešenia. Pamäťová zložitosť bola vo všetkých prípadoch $O(f + h)$, nakoľko sme si pamätali iba načítané reťazce a konštantne veľa pomocnej pamäte.

4. Zdobenie torty

opravuje Peťa a Jano
(max. 9 b za popis, 6 b za program)

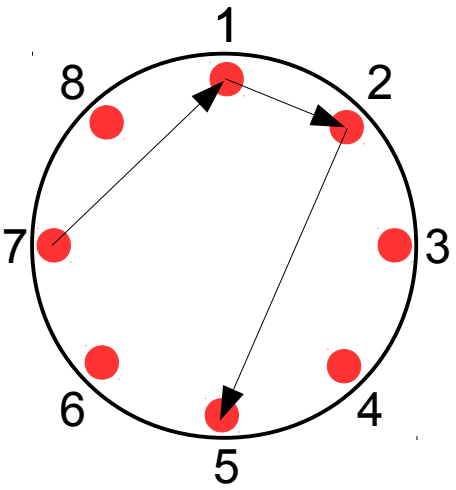
Najprv sa zamyslíme nad tým, ako by sa príklad dal riešiť, keby čiara na torte išla cez všetky jahody. Jahodám, cez ktoré čiara počas kreslenia už prešla, budeme hovoriť *pokreslené*.

Keby sa nám počas kreslenia čiary na tortu niekedy stalo, že sme čiarou práve prišli k nejakej jahode, pričom aj naľavo aj napravo od tejto jahody je nepokreslená jahoda, je jasné, že plán sa už nebude dať dokončiť bez prekríženia čiary. (Ak by sme sa totiž v ďalšom kroku vybrali napravo od poslednej jahody, už sa nedostaneme k tej, čo je naľavo, bez toho, aby sa čiara prekrížila – a naopak analogicky.)

Túto myšlienku môžeme veľmi ľahko zovšeobecniť – pri kreslení čiary od jahody k jahode musí vždy platiť, že každá jahoda (okrem prvej), ku ktorej pôjdeme čiarou, má aspoň jednu z dvoch susedných jahôd už pokreslenú. To znamená, že počas kreslenia budeme mať vždy jeden súvislý úsek jahôd pokreslený a zvyšné jahody budú nepokreslené – teda pri každom ďalšom kuse čiary, ktorý budeme kresliť (okrem posledného), budeme mať na výber z dvoch jahôd – budú to tie nepokreslené, ktoré susedia s pokreslenými.

Program teda v tomto prípade funguje nasledovne: celý čas si bude pamätať interval pokreslených jahôd (na začiatku si ho inicializuje na prvú jahodu) a pri každom ďalšom kroku len skontroluje, či nová jahoda susedí s jednou z krajných jahôd intervalu, a ak áno, interval si príslušne zväčší. Ak nie, vyhlási, že plán je zlý a čiara sa pretne.

Čo ale v prípade, že plán neobsahuje všetky jahody? Keď venujeme chvíľku kresleniu rôznych plánov na papier, uvidíme, že z každej jahody sa môžeme pohnúť dvoma rôznymi smermi – doprava alebo doľava. Navyše, ak ideme vľavo, môžeme si vyberať len z tých jahôd vľavo, ktoré patria do nepokresleného intervalu začínajúceho ľavou susednou jahodou a končiaceho najbližšou pokreslenou jahodou. (Analogicky aj pre druhú stranu – ak ideme vpravo, môžeme ísť len na takú jahodu, ktorá patrí do nepokresleného intervalu začínajúceho pravou susedou jahody, na ktorej sme, a končiaceho prvou pokreslenou jahodou.) Napríklad v situácii na obrázku interval vpravo obsahuje jahody 3 a 4 a interval vľavo jahodu 6.



Nášmu programu by teda stačilo, keby si udržiaval dva intervaly, a pri každej novej jahode sa pozrel, či patrí do jedného z nich; ak áno, príslušne si intervaly upravil, a ak nie, vyhlásil by, že sa čiara pretne. Môžeme si ale všimnúť, že ak za začiatok pravého intervalu vyhlásime prvú pokreslenú jahodu, ktorá je napravo od naposledy navštívenej jahody a za koniec tú, ktorá je pravou susedou naposledy navštívenej jahody, a pre ľavý interval to spravíme naopak (začiatok na susede naposledy navštívenej jahody), tak interval, ktorý začína začiatkom pravého intervalu a končí koncom ľavého intervalu, obsahuje práve všetky jahody navštíviteľné v najbližšom

kroku plus jahodu, na ktorej práve sme. Keďže zadanie nám zaručuje, že jahody sa v pláne nebudú opakovať, môžeme si prácu uľahčiť tým, že budeme kontrolovať príslušnosť nasledujúcej jahody len do jedného intervalu, a nie do dvoch. Na obrázku by to znamenalo, že začiatok spojeného intervalu bude na jahode 2 a koniec na jahode 7 (pričom interval "ide" v smere hodinových ručičiek).

Ako teda náš program funguje? Keď načítame prvú jahodu, nastavíme si začiatok (z) a koniec (k) intervalu a aktuálnu (a) aj poslednú (p) jahodu na tú, ktorú sme práve načítali. Pri načítaní každej ďalšej jahody (uložíme si ju do a) skontrolujeme, či a patrí do intervalu $[z, k]$ – to spravíme zistením, či je a vzdialená od z v smere hodinových ručičiek najviac o toľko, o koľko je v rovnakom smere vzdialené k (keďže k aj z sú už pokreslené, nemôže sa stať, že by $a = z$ alebo $a = k$). Ak nie, zapamätáme si, že čiara sa pretne. Nakoniec si zaktualizujeme interval (čiže jeden z jeho koncov zmeníme na p – to, ktorý, určíme podľa toho, v ktorej jeho časti leží aktuálna jahoda) a nastavíme $p = a$.

Na záver ešte pár slov o zložitosti. Keďže náš program používa len zopár premenných (ich počet nezávisí od veľkosti vstupu), jeho pamäťová zložitosť je konštantná – $O(1)$. Pre každý vstup spraví v cykle, ktorého dĺžka závisí od veľkosti vstupu, konštantne veľa krokov, takže jeho časová zložitosť je lineárna od veľkosti vstupu – $O(m)$. Všimnime si, že obe zložitosti sú optimálne – pamäť $O(1)$ sa nedá vylepšiť zo zrejmých dôvodov, a čas $O(m)$ sa nedá vylepšiť preto, lebo toľko trvá už samotné načítanie vstupu.

Listing programu (C++)

```
#include <cstdio>

int n, m;

bool je_v_intervale(int a, int z, int k) {
    return (a-z+n)%n <= (k-z+n-1)%n; //-1 je tam kvoli prvej jahode, vtedy k = z
}

void sprav() {
    scanf("%d%d", &n, &m);
    int z, k, p, a;
    bool pretne = false;
    for(int i = 0; i < m; ++i) {
        scanf("%d", &a);
        a--; //chceme pracovat s cislami od 0 po n-1, nie od 1 po n
        if (i==0) {
            z = k = p = a;
        } else {
            if (!je_v_intervale(a, z, k)) pretne = true;
            if (je_v_intervale(a, z, p)) k = p;
            else z = p;
            p = a;
        }
    }
    printf("%sPRETNE\n", pretne?"":"NE");
}

int main(){
    int T;
    scanf("%d", &T);
    while(T--) sprav();
}
```

5. Obytná štvrť

opravuje Jano
(max. 3 b za popis, 12 b za program)

Úloha sa dala riešiť nasledujúcimi troma nástrojmi (prípadne kombináciou týchto nástrojov). Počty bodov, ktoré sa dali získať, sú len odhadované a dosť záviseli od šikovnosti a množstva času venovaného tejto úlohe.

1. Použitím ceruzky a papiera sa dalo získať 7 až 9 bodov z 12.
2. Využitím počítača na prechádzanie všetkých možností a automatizovanie manuálnej roboty sa dalo získať 9 až 11 bodov.
3. Využitím počítača a programov špecializovaných na rýchle riešenie optimalizačných úloh, sa dalo získať 10 až 12 bodov.

Bez ohľadu na to, ktorý nástroj použijeme, ak chceme získať veľa bodov, tak postup riešenia bude vyzerať približne takto:

1. Malé plány (s rozmermi do 8) vyriešime samostatne.
2. Následne vymyslíme čo najlepší spôsob, ako vyrobiť nekonečne veľký plán mesta. Teda odmyslíme si okraje a skúsime nájsť nejaký plán – nejakú "kachličku" – ktorú môžeme ľubovoľne veľa krát ukladať veľa seba a pod seba, a stále bude spĺňať podmienky zo zadania. Budeme sa snažiť dosiahnuť čo najvyššiu priemernú výšku kachličky.

3. Veľké plány mesta vyrobíme tak, že naukladáme vedľa seba niekoľko kópií kachličiek a pridáme okraje plánu.

Malé plány

Takto vyzerajú optimálne riešenia pre $n \in \{3, 5, 8\}$ aj so súčtom výšok budov.

```
20          170
241         12413231
132         14324152
241         25135241
            43251432
63          11342514
23131       24515323
14523       13234131
42314       24142142
31452
21231
```

Prvé dva sa dajú nájsť ručne alebo inteligentným skúšaním všetkých možností na počítači. Pre $n = 8$ vieme bezbolestne ručne/pomocou počítača nájsť riešenie s počtom poschodí medzi 160 a 166, ale na optimum treba trocha väčšie kladivo.

Väčšie kladivo môže byť napríklad nejaký heuristický algoritmus – to je taký, ktorý síce nemusí vždy dosiahnuť najlepší výsledok, ale väčšinou sa mu to podarí a navyše je veľmi rýchly. Ako príklady uvedieme simulované žíhanie, genetické programovanie alebo hillclimbing, viac si o nich môžete nájsť na internete, ale konkrétne v tejto úlohe pravdepodobne nebudú veľmi úspešné.

Lepšie je použiť ILP, Integer Linear Programing, čiže celočíselné lineárne programovanie, o ktorom si podrobnejšie môžete prečítať v zadaní 5. príkladu 4. série, 31. ročníka (old.ksp.sk/wiki/uploads/Zadania/ps314.pdf). Pointa je v tom, že úlohu zapíšeme ako skupinu podmienok a výraz, čo chceme maximalizovať, v špecifickom formáte. Táto skupina podmienok a výraz sa nazýva celočíselný lineárny program. Potom na internete dokážeme nájsť nástroje, ktoré vedia hľadať optimálne riešenia pre takéto programy. Ako dobré cvičenie si môžete naštudovať ILP z vyššie uvedeného odkazu alebo inú heuristickú metódu a pomocou nej vytvoriť čo najlepší plán mesta. V tomto vzorovom riešení sa však uspokojíme s 10 bodmi za program a ďalej sa týmto sofistikovaným nástrojom nebudeme venovať.

Inteligentné skúšanie všetkých možností.

Zlé skúšanie možností je nasledovné: Pre každé políčko máme 5 možností aká budova tam môže byť, takže vygenerujeme všetkých 5^{n^2} rôznych miest. Overíme, ktoré z miest vyhovujú podmienkam zo zadania, a vyberieme z nich najlepšie (podľa počtu poschodí).

Riešenie by sa najjednoduchšie implementovalo tak, že plochu budeme generovať po jednotlivých políčkach rekurzívne. Teda si spravíme funkciu `vygeneruj(x,y)`, ktorá vyskúša všetkých 5 možností čo môže byť na políčku `x,y` a pre každú možnosť zavolá `vygeneruj(nasledujúce políčko po x,y)`. Nech `x` je číslo riadku a `y` číslo stĺpca. Riadky aj stĺpce číslujeme od `1` po `n`. Keď chceme políčka prechádzať po riadkoch, tak nasledujúce políčko po `x,y` má súradnice `x + y/n, y mod n + 1`. Keď sa takto dostaneme na riadok `n+1`, tak sme vygenerovali celú plochu a ostáva nám overiť, či vyhovuje zadaniu. Keďže pre každé políčko sme skúsili všetky možné výšky, nestane sa nám, že by sme na nejaký plán zabudli.

```
vygeneruj(x, y):
    pokiaľ y>n:
        pokiaľ Plocha vyhovuje zadaniu a má zatiaľ najväčší počet poschodí:
            aktualizuj najlepšie riešenie
        koniec
    pre i od 1 do 5:
        Plocha[x][y] = i
        vygeneruj(x + y/n, y%n + 1)

vygeneruj(1,1)
vypíš najlepšie riešenie
```

Prečo je toto zlé? Lebo je to pomalé a veľa roboty sa robí zbytočne. Už pre $n = 4$ by výpočet trval asi hodinu. A pre $n = 5$ by sme sa nedočkali výsledku ani do konca roka. Skúsime to teda zlepšiť.

Pri inteligentnom skúšaní všetkých možností musíme priebežne kontrolovať, či vôbec má zmysel pokračovať ďalej. Pokiaľ napríklad prvé dve budovy budú 5 a 5, tak už nemá zmysel skúšať, ako by vyzeral zvyšok mesta, pretože určite nebude vyhovovať zadaniu.

```
vygeneruj(x, y):
    pokiaľ y>n:
        skontroluj Plochu a prípadne aktualizuj najlepšie riešenie
        koniec
    skontroluj Plochu, či ma zmysel pokračovať
    ak nemá zmysel pokračovať:
        koniec
    pre i od 1 do 5:
        Plocha[x][y] = i
        vygeneruj(x + y/n, y%n + 1)
        Plocha[x][y] = 0
```

Čím lepšie naprogramujeme funkciu na kontrolu, tým menej možností bude program skúšať. Jednoduchá kontrola, ktorá veľmi pomôže, je prejsť všetky nenulové políčka v poli `Plocha` a pre každé z nich overiť, či by sa dali doplniť okolité nuly tak, aby malo políčko okolo seba všetky menšie čísla (tak ako to káže zadanie). Celá kontrola sa dá implementovať v čase $O(n^2)$. Konkrétnu implementáciu si môžete pozrieť na spodku tohoto vzorového riešenia.

Trocha času sa dá ušetriť aj zvolením si vhodného programovacieho jazyka, napríklad C++ je asi desaťnásobne rýchlejšie než Python.

Nekonečne veľký plán mesta

Chceli by sme mať čo najvyššie budovy, ale celé nám to kazia okraje, lebo budovy na okraji majú málo susedov. Keby sme povedali, že budovy úplne vľavo susedia s tými úplne v pravo a budovy v prvom riadku susedia s budovami v spodnom riadku, mohli by sme dosahovať plány s oveľa lepším skóre.

Napríklad plán

```
1323
2414
1525
```

má priemernú výšku budovy 2.75. (Pre porovnanie, optimálny plán pre 5×5 podľa pôvodných pravidiel, mal priemernú výšku 2.52). Keď sa potrápime trocha viac, nájdeme optimálny plán s rozmermi 5×5 (s tým, že protiľahlé okraje susedia):

```
12345
45123
23451
51234
34512
```

Tento plán má priemernú výšku 3. Ukladaním takejto "kachličky" ľubovoľne veľakrát pod seba a vedľa seba vieme vytvoriť nekonečnú plochu s priemernou výškou 3. Aby sme nemuseli hľadať ďalšie plány, ukážeme si, že lepšia priemerná výška sa nedá dosiahnuť.

Horné ohraničenie

Každá budova, ktorá nemá 1 podlažie, musí susediť s jednopodlažnou budovou. Jedna jednopodlažná budova môže susediť s najviac 4 ďalšími budovami, inými slovami, jedna jednopodlažná budova dovolí najviac štyrom budovám mať viac poschodí. Z toho vyplýva, že **aspoň jedna pätina všetkých budov je jednopodlažná**.

Podobne dvojpodlažná budova môže mať okolo seba najviac 3 vyššie budovy (lebo musí mať vedľa aspoň jednu jednopodlažnú). Preto aspoň **štvrtina z budov, ktoré nie sú jednopodlažné, sú dvojpodlažné**. Analogicky ukážeme, že **aspoň tretina zo zvyšných budov sú trojpodlažné** a **aspoň polovica zo zvyšku je štvorpodlažná**.

Jednoduchou matematikou môžeme spočítať, že priemerná výška budov potom nemôže byť viac ako tri.

Skladanie veľkých konečných plôch

Po tom, ako sme našli "kachličku" 5×5, môžeme nadobudnúť presvedčenie, že najlepší plán mesta pre veľké n bude vyzerať zhruba takto:

```
????????????????...?
?123451234512345   ?
?451234512345123   ?
?234512345123451...?
?512345123451234   ?
?345123451234512   ?
.       .          .
.       .          .
.       .          .
????????????????...?
```

Avšak rýchlo zistíme, že v druhom stĺpci a piatom riadku nemôže byť päťposchodová budova, pretože v prvom stĺpci by musela byť štvorka, ktorá by nevedela mať okolo seba 1, 2 aj 3. Preto skutočný plán mesta by vyzeral takto (rozdiel je len v druhom stĺpci):

```
????????????????...?
?123451234512345   ?
?451234512345123   ?
?234512345123451...?
?412345123451234   ?
?345123451234512   ?
.       .          .
.       .          .
.       .          .
????????????????...?
```

Alebo sa nám možno oplatí nezačať číslom 1 ale spraviť druhý riadok `234512345...`, `345123451...`, `451234512...` či `512345123...`

A program by už len mal uhádnuť, aké výšky budov sú schované pod otáznikmi. Pre veľké n to môže byť stále veľa možností, tak sa môžme obmedziť len na okraje, ktoré sa periodicky opakujú. Teda 10-te číslo v prvom riadku musí byť rovnaké ako 5-te číslo v prvom riadku.

Za týchto podmienok je možností dostatočne málo na to, aby náš program, ktorý skúša všetky možnosti, bežal len chvíľu.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
using namespace std;
#define REP(i, n) for(int i = 0; i<int(n); ++i)
#define FOR(i, n) for(int i = 1; i<=int(n); ++i)
typedef vector<int> vi;

int n, best, sum;
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
vector<vi> M,Naj,V; // Mapa, Najlepsia mapa, Vynimky
int vedla[7] = {0};

// skontrolujeme, ci mapa vyhovuje podmienkam zo zadania
bool spravne(){
    FOR(i, n) FOR(j, n){
        if (M[i][j] <= 0) continue;
        FOR(d, 6) vedla[d] = 0;
        REP(d, 4) vedla[M[i+dx[d]][j+dy[d]]+1]++;
        FOR(d, M[i][j]-1) if (!vedla[d+1]) vedla[1]--;
        if (vedla[1] < 0) return false;
    }
    return true;
}

// kolko mam pripocitat na poziciu x,y
// (meni to cisla 5 v druhom stlpci na cisla 4)
int vynimka(int x, int y){
    if (y>3) return 0;
    if (y==2 && M[x-1][y] == 2) {
        V[x][y] = 1;
```

```
        return -V[x][y];
    }
    return max(V[x][y-1],V[x-1][y]);
}

// toto makro trocha skratilo zdrojovy kod
#define VYGENERUJ(d, p, q) {\
    M[x][y]=d;\
    if (p)\
        vygeneruj(x+y/n, y%n+1);\
    V[x][y] = M[x][y]=0;\
    q\
}

// vygeneruj dalsie policko (na pozicii x,y)
void vygeneruj(int x, int y){
    if (x>n) {
        if (!spravne()) return;
        sum = 0;
        FOR(i, n) FOR(j, n) sum+=M[i][j];
        if (sum>best){
            best = sum;
            Naj = M;
        }
        return;
    }
    // cela mapa je dost periodicka, takze vacsinou vieme
    // urcit hotnotu na zaklade predoslych
    if (x-5>1 && x<n) VYGENERUJ(M[x-5][y], true, return;)
    if (y-5>2 && y<n) VYGENERUJ(M[x][y-5], true, return;)
    if (x>1 && x<n && y>1 && y<n) {
        if (y>2) VYGENERUJ((M[x][y-1]+vynimka(x,y))%5+1, true, return;)
        if (x>2) VYGENERUJ((M[x-1][y]+2+vynimka(x,y))%5+1, true, return;)
    }
    // v opacnom pripade skusime vsetkych 5 moznosti, ale vzdy kontrolujeme,
    // ci ma zmysel pokracovat
    FOR(d, 5) {
        VYGENERUJ(d, spravne(), ;)
    }
}

int main(){
    scanf("%d", &n);
    Naj = M = vector<vi>(n+2, vi(n+2, -1));
    V = vector<vi>(n+2, vi(n+2, 0));
    FOR(i, n) FOR(j, n) M[i][j] = 0;

    vygeneruj(1,1);
    FOR(i, n) {
        FOR(j, n) printf("%d", Naj[i][j]);
        printf("\n");
    }
}
```

Tento program vstupy s n do 5 rieši optimálne, pri $n = 8$ a $n = 13$ sa mu veľmi nedarí, ale pri vyšších n je veľmi blízky najlepšiemu vedúcovskemu riešeniu. (V najhoršom prípade dosahuje o 3 poschodia menej ako najlepšie riešenie.) Nie je úplne najrýchlejší takže na výsledok si budeme musieť počkať asi pol hodinu.

No a ako dostať viac bodov? Naše riešenia skúšalo doplniť len okraj veľkosti jedna a ten okraj navyše musel byť periodický. Keď skúsime okraj veľkosti dva alebo tri a dovolíme robiť aj neperiodické okraje, dosiahneme lepšie výsledky. Aby sme sa dočkali výsledku do konca série, musíme použiť nejaké z väčších kladív, ktoré sme spomínali v prvej časti vzoráku.

My sme použili ILP, pre n do 13 sa stíhal okraj veľkosti 3, pre vyššie n okraj veľkosti 2. Počty poschodí najlepších plánov boli postupne 7, 20, 63, 170, 467, 1249, 3342, 8863, 23416 a 61640.

6. Okultistický fundraising

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

Táto úloha bola na pomery O-čka pomerne ľahká. Hlavná časť jej riešenia používa metódu dynamického programovania a ak sa vám túto úlohu nepodarilo vyriešiť, určite by ste si mali prečítať toto vzorové riešenie, pretože táto metóda je často používaná.

Prezeranie všetkých riešení

Skôr ako sa pustíme do vzorového riešenia, povedzme si čo-to o menej optimálnych prístupoch. Úloha po nás chce, aby sme sa o každej z n kariet rozhodli, ktorou stranou ju otočíme (ktoré číslo sa objaví na vrchu) a chceme, aby súčet týchto čísel dal dokopy hodnotu s.

Ako prvé riešenie teda môžeme vyskúšať všetky možné otočenia kariet. Pre každé otočenie vyskúšame, aký súčet dostaneme a ak sa nejaký súčet rovná s, prehlásime toto otočenie za výsledok. Ak ani jedno otočenie nebude vyhovovať, riešenie zjavne neexistuje. Otázka je, koľko je možných otočení n kariet. A asi je jasné, že týchto možností je 2^n. Ak naviac každú možnosť spracujeme v čase $O(n)$ dostávame riešenie so zložitosťou

$O(n2^n)$. Nič úžasné, ale dosť na to, aby sme získali sľubované 3 body.

Otázkou teraz zostáva, ako takéto riešenie naprogramovať a ako ho naprogramovať čo najbezbolestnejšie. Celý trik leží v tom, ako si reprezentujeme dané otočenie. Ak sa pozrieme na výstup, vidíme, že máme vypísať postupnosť čísel 1 a 2, kde i-te číslo reprezentuje otočenie i-tej karty. Čísla 1 a 2 nie sú pre počítač až také pekné – čo ak by sme ich zmenili na čísla 1 a 0? Potom by predsa každé otočenie bol bitový reťazec, čo je vlastne **číslo v binárnom zápise**.

Reprezentovať si otočenie pomocou čísla môže byť pomerne pohodlné, hlavne ak vezmeme do úvahy, že každé číslo od 0 po $2^n - 1$ je nejaké platné otočenie, a navyše každé dve rôzne čísla reprezentujú dve rôzne otočenia. Otázkou však zostáva, či sa vieme nejakým jednoduchým spôsobom dostať k i-temu bitu čísla x, teda zistiť, či je tento bit 0 alebo 1. Odpoveď je, že to vieme dokonca v konštantnom čase, použitím dvoch binárnych operácií – *shift left* (zapisovaný ako <<) a *and* (zapisovaný ako &). Operácia $x\&(1 << i)$ potom vráti na výstup i-ty bit čísla x.[3]

Listing programu (C++)

```
for(int i=0; i<(1<<n); i++) {
    int sucet = 0;
    for(int j=0; j<n; j++)
        if(i&(1<<j)) sucet += druha_strana[j];
        else sucet += prva_strana[j];
    if(sucet == s) riesenie = i;
}
```

Meet in the middle

Keď už máme tri body, mohli by sme poškuľovať po tých piatich. Obmedzenie, ktoré máme zadané, nám vraví, že n je menšie ako 50, čo je zhruba dvojnásobok predchádzajúceho obmedzenia. Mohli by sme teda stále zostať v exponencionálnej časovej zložitosti, akurát ju trošku zlepšiť. V tomto prípade, ak by sa nám podarilo vytvoriť algoritmus so zložitosťou približne $O(2^{n/2})$, ešte stále by to mohlo fungovať. Vyzerá to teda, ako by sme mohli naraz pracovať len s polovicou vstupu. A presne na tom sa zakladá riešenie metódou meet in the middle.

Zoberme si prvú polovicu vstupu a pustime naň predchádzajúci algoritmus. Vyskúšame teda všetky možné otočenia prvej polovice kariet. Ak je súčet nejakého otočenia väčší ako s, toto otočenie môžeme rovno zahodiť, lebo je nepoužiteľné. V opačnom prípade si ho ale zapamätáme aj s príslušným súčtom.

Keď sme spracovali prvú polovicu, pustime sa do druhej. Zoberieme si nejaké otočenie druhej polky kariet a zistíme, že jeho súčet je x. To ale znamená, že ak sa nám podarí otočiť prvú polovicu kariet tak, aby dávala súčet $s - x$, máme vhodné otočenie. Dokopy totiž obe časti dajú súčet s. Stačí sa teda pozrieť do riešení prvej polovice – a aby to bolo dosť efektívne, tieto riešenia chceme mať uložené v *map*e alebo utriedené podľa súčtov a v nich binárne vyhľadávať.

Spracovanie prvej polovice nám trvá $O(n2^{n/2})$ (najväčší prínos má triedenie, ktorého zložitosť je $2^{n/2}\log(2^{n/2})$, to je však po odstránení logaritmu $n2^{n/2}$) a tento čas nám trvá aj spracovanie druhej polovice. Keďže sme ich však od seba oddelili a spracovávame ich samostatne, výsledný časová zložitosť bude tiež $O(n2^{n/2})$, a to by nám malo stačiť na 5 bodov. Riešenie vyzerá podobne ako predchádzajúce, minimálne sa v ňom využívajú rovnaké časti programu[4].

Listing programu (C++)

```
map<int,int> prva_polovica;
int pol = n/2;
for(int i=0; i<(1<<pol); i++) {
    int sucet = 0;
    for(int j=0; j<pol; j++)
        if(i&(1<<j)) sucet += druha_strana[j];
        else sucet += prva_strana[j];
    prva_polovica[sucet] = i;
}
pol = n - pol;
for(int i=0; i<(1<<pol); i++) {
    int sucet = 0;
    for(int j=0; j<pol; j++)
        if(i&(1<<j)) sucet += druha_strana[n/2 + j];
        else sucet += prva_strana[n/2 + j];
    if(prva_polovica.find(s - sucet) != prva_polovica.end()) {
        riesenie_prva = prva_polovica[s - sucet];
        riesenie_druha = i;
    }
```

[3] Prezradím, že číslo $1 << i$ vráti hodnotu 2^i a & je bitový *and* dvoch čísel pracujúci bit po bite. Zvyšok si domyslite (dogooglite) sami.

[4] Opäť odporúčam dogoogliť všetky pojmy, ktorým ste neporozumeli úplne. Napríklad set v C++ alebo aj samotný meet in the middle postup.

}

Vzorové riešenie

Podľa obmedzení, ktoré máme zadané, je jasné, že žiadne exponenciálne riešenie nebude vyhovovať, preto to potrebujeme nejakým spôsobom zlepšiť. Začnime teda malým trikom.

Hlavný problém je, že sa musíme rozhodovať, ktorú stranu zoberieme, a ktoré číslo prirátame k výsledku. Zoberme si teraz kartu, na ktorej sú čísla 8 a 3. Je jasné, že nech otočíme kartu ľubovoľnou stranou, k výsledku pripočítame **aspoň** hodnotu 3. Čo ak by sme teda nahradili túto kartu kartou, ktorá má na sebe čísla 5 a 0, a odčítali číslo 3 od s? Dostali by sme novú sadu kariet a ich otočením by sme chceli dosiahnuť hodnotu $s - 3$. Vidíme, že naša úloha sa vlastne vôbec nezmenila, a ak nájdeme otočenie, ktoré rieši túto úlohu, toto otočenie bude dobré aj pre nezmenené karty.

Takýmto spôsobom môžeme upraviť všetky karty. Zistíme minimum z ich strán a toto minimum odčítame jednak od oboch strán karty, jednak od hodnoty s. Na prvý pohľad sa zdá, že sa nič nezmenilo, stále hľadáme otočenie n kariet, akurát hľadáme inú hodnotu s'. Na jednej zo strán karty je však zakaždým hodnota 0. To znamená, že náš problém sa dá preformulovať: Ktoré karty máme vybrať, aby ich súčet bol s'?

Znamená to, že máme množinu n čísel a chceme zistiť, či niektorá z množín týchto čísel má súčet s'. Možno, že to tak nevyzerá, úloha sa nám však trochu zjednodušila a skúsenejší z vás v nej možno spoznali klasický problém o napĺňaní batoha (aspoň jednu z mnohých verzií tohoto problému).

Ako však riešiť túto úlohu? Ešte stále sa nám ponúka vyššie ukazované exponenciálne riešenie, pomocou ktorého vieme prechádzať všetky podmnožiny. Samozrejme toto riešenie je príliš pomalé. Treba sa preto pozrieť na to, prečo je tak pomalé. Spýtajme sa teda veľmi dôležitú otázku, ktorú by ste sa mali pýtať zakaždým: **Nerobíme niečo viac krát?** Ak totiž rátame nejakú informáciu viacej krát, dá sa toto opakovanie odstrániť a tým zrýchliť naše riešenie.

Zoberme si množinu čísel $\{1, 3, 2, 5\}$. Ak spracujeme prvé tri prvky, zistíme, že pomocou prvého a tretieho prvku vieme dosiahnuť hodnotu 3, ale túto hodnotu vieme dosiahnuť aj pomocou druhého prvku. Máme teda dva rôzne spôsoby ako dosiahnuť tú istú hodnotu a k obom týmto možnostiam následne skúsime pridať hodnotu 5. Nám však stačí, ak zistíme, že hodnotu 3 vieme vyskladať pomocou prvých troch prvkov a potom k tejto hodnote pridať 5 len raz. Samozrejme, dá sa argumentovať, že toto nám nemusí pomôcť, lebo každá podmnožina čísel môže dávať rôzny súčet. Možných súčtov by teda bolo 2^n, pre nás zaujímavé sú však len tie súčty, ktoré sú menšie ako s'. V okamihu ako nájdeme väčší súčet, môžeme ho rovno zahodiť, lebo nám nemôže pomôcť, keďže pracujeme len s kladnými číslami. A hodnota s' je podľa zadania najviac 50 000, čo je často menej než 2^n.

Zostáva už len správne využiť spomenuté pozorovanie. Spravíme si pole V dĺžky s' a na políčko x si budeme značiť, či vieme dosiahnuť súčet x pomocou nejakej podmnožiny prvých i čísel. Na začiatku, keď nepoužívame žiadne čísla, jediný súčet, ktorý vieme dosiahnuť, je 0. Preto si nastavíme hodnotu $V[0] = 1$ a zvyšné hodnoty na 0.

Nech už máme pridaných prvých $i - 1$ čísel a chceme pridať i-te číslo, ktoré má hodnotu a_i. Prezerajme postupne pole V, prvok po prvku. Ak je na x-tej pozícii hodnota 0, nepotrebujeme urobiť nič – hodnotu x nevieme vyskladať z prvých $i - 1$ kariet a teda nemôžeme pridať túto kartu. Ak je však $V[x]$ rovné 1, tak to znamená, že existuje podmnožina prvých $i - 1$ kariet, ktorá dáva súčet x. Ak k tejto (ľubovoľnej z nich) podmnožine pridáme kartu a_i, dostaneme množinu so súčtom $x + a_i$ a zaznačíme si do V, že $V[x + a_i] = 1$. Ak náhodou $x + a_i > s'$ tak túto hodnotu zahodíme.

Ak po spracovaní všetkých kariet sa $V[s']$ rovná 1, tak naša úloha má riešenie, v opačnom prípade nevieme nájsť podmnožinu s požadovaným súčtom. Zostávajú nám posledné dve veci, ktoré musíme vyriešiť. Prvou z nich je, ako spätne zistiť, ktoré čísla použijeme vo výslednej množine, ktorá dáva súčet s', aby sme vedeli späť zistiť, ktoré karty otočiť na ktorú stranu. Druhá vec je trochu nejasnejšia, ale pomôže nám v tom, aby nám celý algoritmus fungoval korektne.

Začnime tou druhou. Problém nášho algoritmu môže vzniknúť pri prechádzaní poľa V a pridávaní ďalšej karty. Ak budeme prechádzať pole od hodnoty 0 po s', tak sa stane nasledovná vec. Nájdeme jednotku vo $V[0]$ a nastavíme na jedna $V[a_1]$. Potom ale nájdeme aj túto novo pridanú jednotku a nastavíme $V[2a_1]$ na jednotku. To však nie je platná množina, lebo pomocou jednej karty s hodnotou a_1 ju nevieme dosiahnuť. Preto si treba dať pozor, aby sme pole prechádzali **od väčších hodnôt k menším**. Tým zaručíme, že upravujeme už spracované indexy a náš algoritmus bude fungovať správne.

Posledný problém, ako spätne zistiť, ktoré čísla tvoria výslednú množinu, vieme vyriešiť pomocou jedného poľa navyše. V tomto poli S si na pozícii x zaznačíme, ktoré číslo sme pridali ako posledné do množiny so súčtom x. Ak teda nastavíme hodnotu $V[x+a_i]$ na jednotku, do políčka $S[x+a_i]$ nastavíme hodnotu i. Pomocou tohoto poľa a hodnôt a_i vieme spätne zrekonštuovať riešenie.

Rekapitulácia

Zopakujme si teda celé naše riešenie. Začneme tým, že upravíme všetky naše karty tak, aby na jednej strane bola 0 a na druhej kladná hodnota, a príslušne upravíme hodnotu s. S novovytvorenými číslami potom spustíme algoritmus, ktorý bude zisťovať, ktoré všetky súčty vieme vytvoriť z prvých i čísel. Na konci zistíme, či vieme vytvoriť súčet s' a pomocou pomocného poľa S, do ktorého si ukladám posledné použité číslo, zistíme, ktoré čísla patria do riešenia. Následne otočíme každú kartu, ktorej patrí číslo z tejto množiny, na stranu, kde má maximum a zvyšné karty na stranu, kde majú minimum.

Aká bude časová a pamäťová zložitosť tohoto riešenia? Na zisťovanie dosiahnuteľnosti súčtov a pamätanie si kroku späť potrebujeme pole dĺžky s a na zapamätanie kariet pole veľkosti n. Máme teda pamäťovú zložitosť $O(n + s)$. Čo sa týka času, tak na pridanie jednej karty k našim vytvoreným množinám potrebujeme prejsť celým poľom dĺžky s a toto opakujeme n krát. Dosiahneme teda zložitosť $O(ns)$, ktorá nám bude stačiť na získanie plného počtu bodov.

Na záver poznamenám, že riešenie využíva to, že hodnota s je dostatočne malá a teda treba vždy zvážiť, kedy takéto riešenie použiť. Naviac je dobré si uvedomiť, že riešenie nemá polynomiálnu časovú zložitosť, lebo je závislé od hodnoty s, ktorá nezodpovedá veľkosti vstupu. Vstup je veľký $O(n)$ a s môže byť ľubovoľne veľké bez toho aby sa zväčšila veľkosť vstupu. Takéto riešenie nazývame pseudopolynomiálne.

Listing programu (C++)

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

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

typedef pair<int,int> pii;

int main() {
    int n,s;
    scanf("%d %d",&n,&s);
    vector<pii> K; K.resize(n);
    For(i,n) scanf(" %d %d",&K[i].first,&K[i].second);
    vector<int> Vys; Vys.resize(n);
    vector<int> M; M.resize(n);
    For(i,n)
        if(K[i].first < K[i].second) {M[i]=K[i].second-K[i].first; s-=K[i].first; Vys[i]=0;}
        else {M[i]=K[i].first-K[i].second; s-=K[i].second; Vys[i]=1;}
    if(s<0) {printf("A je to v ...\n"); return 0;}
    vector<int> V; V.resize(s+47,0); V[0]=1;
    vector<int> S; S.resize(s+47,-1);
    For(i,n) {
        for(int j=s; j>=0; j--) {
            if(V[j] == 0) continue;
            if(j+M[i]>s) continue;
            if(V[j+M[i]]==1) continue;
            V[j+M[i]]=1; S[j+M[i]]=i;
        }
    }
    if(V[s] == 0) {printf("A je to v ...\n"); return 0;}
    int kde=s;
    while(kde!=0) {
        Vys[S[kde]]=1-Vys[S[kde]];
        kde-=M[S[kde]];
    }
    For(i,n) printf("%d",Vys[i]+1);
    printf("\n");
}
```

opravuje Tomi

7. Ohavný čin

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

Chceme stiahnuť n súborov, pričom každý súbor môžeme buď stiahnuť celý, alebo si ušetriť čas a stiahnuť iba jeho rozdiely oproti nejakému inému súboru, ktorý už máme stiahnutý. O všetkom z toho vieme, koľko by to trvalo.

Základná myšlienka riešenia je previesť naše zadanie na problém najlacnejšej kostry. To je štandardná úloha, v ktorej dostaneme neorientovaný ohodnotený graf, a máme vybrať takú podmnožinu hrán, že má minimálnu cenu a graf obsahujúci len tieto hrany bude súvislý. Takže keď v grafe nájdeme najlacnejšiu kostru, stále sa po nej zvládneme dostať odvšadiaľ všade, ale celkový súčet váh hrán bude minimálny.

Tak to skúsme. Povedzme, že každý vrchol bude reprezentovať jeden stiahnutý súbor, a medzi každými dvoma vrcholmi bude hrana s takou váhou, koľko trvá stiahnuť rozdiely medzi tými súbormi. Keby sme v tomto grafe našli najlacnejšiu kostru, zistili by sme, ako by sa dalo stiahnuť všetky súbory, keď už jeden máme.

Ale zatiaľ sme obmedzení len na sťahovanie rozdielov. Stále sme sa nedozvedeli, ktoré súbory máme stiahnuť celé. Našťastie už nechýba veľa: Stačí do grafu pridať nový vrchol, ktorý predstavuje situáciu "zatiaľ žiadne

súbory nemám". Medzi každým súborom a týmto novým vrcholom bude hrana s takou váhou, koľko trvá stiahnuť celý ten súbor.

Veľkosť najlacnejšej kostry pre tento graf je naša odpoveď. Najlacnejšia kostra vlastne hovorí návod, ako všetko najrýchlejšie stiahnuť. Súbory, čo susedia s vrcholom "ešte nič nemám", stiahneme celé, a potom podľa kostrových hrán postupne stiahneme všetko ostatné (tam už stačí sťahovať rozdiely).

Hľadanie najlacnejšej kostry

Na naše konkrétne zadanie už môžme zabudnúť. Pozrime sa, ako sa hľadá najlacnejšia kostra vo všeobecnom grafe s v vrcholmi a e hranami. Jeden z algoritmov, čo túto úlohu vedia riešiť, sa volá Primov algoritmus.

Primov algoritmus buduje výslednú kostru vrchol po vrchole. Na začiatku bude kostra mať jediný vrchol (je jedno, ktorý). Potom v každom ťahu našu kostru o niečo zväčšíme – nájdeme najlacnejšiu hranu, ktorá prepája nejaký kostrový a nejaký nekostrový vrchol, a pridáme ju do našej kostry. To opakujeme, až kým nie je v kostre každý vrchol (čiže $v - 1$ ráz). Kostra, ktorú týmto postupom nájdeme, je zaručene najlacnejšia.

Ako nájdeme tú najlacnejšiu hranu? Väčšinou sa na to používa binárna halda, ale teraz si ukážeme, ako to spraviť lineárnym prehľadávaním. Budeme mať pole, v ktorom si pre každý vrchol p uložíme cenu pridania p do kostry – čiže najlacnejšiu hranu z ľubovoľného kostrového vrchola do p. Keď vyberáme, ktorý vrchol pridať, proste zaradom prejdeme celé toto pole a vyberieme najlacnejší. Z tohto vrchola sa tým pádom stáva kostrový vrchol, takže sa pozrieme na všetkých jeho nekostrových susedov a zapíšeme do poľa, či sme pre nich nenašli lepšiu cenu. A to je celé.

Tento algoritmus má časovú zložitosť $O(v^2)$. To je zvyčajne dosť veľa – keby sme použili Primov algoritmus s binárnou haldou, mali by sme zložitosť iba $O(e \log v)$, a Kruskalov algoritmus by nám tiež dal $O(e \log v)$. Lenže my máme kompletný graf, kde $v = O(n)$ a $e = O(n^2)$. V ňom majú Kruskal aj haldový Prim časovú zložitosť $O(n^2 \log n)$, zatiaľ čo trápnemu lineárnemu prehľadávaniu stačí $O(n^2)$. V praxi to síce nie je veľký rozdiel, takže asi budú fungovať aj programy s pomalšími algoritmami, ale v popise ste za to mohli stratiť jeden bod.

Listing programu (C++)

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

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

int main() {
    int n;
    int G[1047][1047];
    scanf("%d",&n);
    For(i,n) For(j,n) scanf("␣%d",&G[i][j]);
    long long res=0;
    vector<int> V; V.resize(n,1000000047);
    For(i,n) V[i]=G[i][i];
    For(i,n) {
        int p=-1;
        For(j,n) if(V[j]!=-1 && (p==-1 || V[p]>V[j])) p=j;
        res+=V[p];
        V[p]=-1;
        For(j,n) if(V[j]>G[p][j]) V[j]=G[p][j];
    }
    printf("%lld\n",res);
}
```

opravuje Jaro

8. Organizačné opletačky

(max. 15 b za popis, 10 b za program)

Najprv sa pozrime na to, čo po nás vlastne zadanie chce. Potrebujeme nejako vhodne povyberať z každého dňa konkrétnu aktivitu tak, aby sme čo najlepšie pokryli všetky aktivity, ktoré chce Samko stihnúť. Pre nás je ale vhodnejší iný pohľad: hľadajme také pospájanie aktivít s dňami (v ktorých sa daná aktivita vyskytuje), kde každý deň je spojený s najviac jednou aktivitou zo Samkovho zoznamu a podobne aj každá aktivita je spojená s najviac jedným dňom. Skúsenejší riešitelia hneď zbadajú párenie, dokonca (a našťastie) na bipartitnom grafe.[5]

Zostavíme si teda graf, v ktorom ako vrcholy jednej partície pôsobia aktivity, ktoré chce Samko stihnúť, a ako vrcholy druhej partície pôsobia jednotlivé dni, ktoré má k dispozícii. Hrana medzi dňom a aktivitou existuje práve vtedy, keď ju môže Samko v ten deň vykonávať. Nás bude zaujímať, koľko najviac hrán z tohto grafu vieme vybrať tak, aby žiadne dve nemali spoločný koncový vrchol. Takúto množinu hrán nazveme maximálnym párením. Počet hrán maximálneho párenia bude našou hľadanou odpoveďou.

[5] Bipartitný graf je taký, u ktorého vieme vrcholy rozdeliť do dvoch množín, tzv. partícií, tak, aby žiadne hrany neviedli v rámci jednej partície.

Maximálne párenie

Pre popis algoritmu hľadajúceho maximálne párenie si zavedieme ešte jeden pojem: zlepšujúca cesta v nejakom už vybratom (nie maximálnom) párení P. Ide o postupnosť neopakujúcich sa vrcholov (spojených hranami), ktorá začína v nespárenom vrchole v_0, pokračuje po nepoužitej hrane do nejakého vrchola u_0. Kým táto cesta neskončí v nespárovanom vrchole, pokračuje ďalej z vrchola u_i do vrchola v_{i+1}, po hrane z párenia P, z ktorého pokračuje do vrchola u_{i+1} opäť po hrane, ktorá sa v párení P nenachádza. Všimnime si, že cesta ide z vrcholov v po nespárenej hrane a z vrcholov u po spárenej. Keďže cesta začína vo vrchole v_0 a končí v nejakom vrchole u_k, bude obsahovať viac nespárených, než spárených hrán.

Keď nájdeme zlepšujúcu cestu v nejakom párení, vieme toto párenie zväčšiť o 1, pokiaľ „preklopíme“ hrany na tejto zlepšujúcej ceste, čiže tie hrany z cesty, ktoré sme mali pôvodne v párení, z neho odstránime a zvyšné doňho pridáme. Existuje dôkaz, že párenie je maximálne práve vtedy, keď v ňom neexistuje zlepšujúca cesta[6].

Preto sa bude celé riešenie párenia zaoberať hľadaním zlepšujúcich ciest na bipartitnom grafe. Označme si partície tohto grafu ako A a B. Stačí nám používať obyčajné prehľadávanie do hĺbky, ktoré vždy začne napríklad v (každom) nespárenom vrchole z A, pôjde do každého možného vrchola z B po nespárenej hrane, potom po spárenej hrane späť do vrchola z A, atď. atď., až dôjde do nespáreného vrchola z B. V tomto momente „preklopí“ všetky hrany.

Toto celé sa bude opakovať, kým sa nestane, že nenájdeme žiadnu zlepšujúcu cestu. Vtedy však môžeme byť spokojní, lebo sme objavili maximálne párenie.

Časová zložitosť takéhoto riešenia je $O(m \cdot (v + h))$, kde m je veľkosť maximálneho párenia, v je počet vrcholov nášho grafu a h je počet hrán tohto grafu. m vieme zhora odhadnúť ako $min(n, k)$, $v = n + k$. V najzákernejšom prípade však môže byť počet hrán až $n \cdot k$, napríklad ak máme k-krát tú istú aktivitu, ktorá sa môže vykonať v každý deň.

Listing programu (C++)

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

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

int n,k;
int A[100047][4];
vector<vector<int> > G;
vector<bool> T;
vector<int> P;

bool zlepsi(int v) {
    T[v]=true;
    For(i,G[v].size()) {
        int w=G[v][i];
        if(P[w]==-1) {P[w]=v; return true;}
        if(!T[P[w]] && zlepsi(P[w])) {P[w]=v; return true;}
    }
    return false;
}

int matching() {
    P.resize(n+k,-1);
    int res=0;
    For(i,n) {
        T.clear(); T.resize(n,false);
        if(zlepsi(i)) res++;
    }
    return res;
}

int main() {
    scanf("%d",&n);
    For(i,n) For(j,4) scanf("␣%d",&A[i][j]);
    scanf("␣%d",&k);
    G.resize(n+k);
    For(i,k) {
        int x;
        scanf("␣%d",&x);
        For(j,n) For(k,4) if(A[j][k]==x) G[j].push_back(n+i);
    }
    printf("%d\n",matching());
return 0;
}
```

Kam ďalej

Prvé možné zlepšenie je v samotnom algoritme hľadajúcom maximálne párenie. Keby sme namiesto prehľa-

[6] Ak by ste si to chceli formálne dokázať, najjednoduchší postup pre netriviálnu implikáciu je nepriamy dôkaz. Vezmete si nejaké nie maximálne párenie a nejaké maximálne, pozriete sa, ako by vyzerali komponenty grafu, ktorý bude obsahovať všetky vrcholy, ale len hrany, koré sú aspoň v jednom z párení, a zistíte, že musí existovať zlepšujúca cesta.

dávania do hĺbky použili prehľadávanie do šírky, vieme nájsť viacero zlepšujúcich ciest naraz a vieme to ukopať k časovej zložitosti $O(h \cdot \sqrt{v})$[7].

S tým sa však neuspokojíme a potiahneme to kúsok ďalej. Čo nám vedelo strašne pokaziť časovú zložitosť, boli viacnásobné aktivity. S nimi sa vieme vysporiadať tak, že všetky rovnaké aktivity pospájame do jedného vrchola, v ktorom si nejako označíme, že s ním môže byť pospájaných viacero rôznych dní. Namiesto toho, aby sme nejako čarovne upravovali pôvodný algoritmus pre párenie, si ukážeme, ako sa celá úloha dá previesť na úlohu o maximálnom toku v grafe.

Na to si zostrojíme ešte trošku iný graf. Ten bude ohodnotený a orientovaný a bude obsahovať dva špeciálne vrcholy, takzvaný *source* a *sink*, ďalej vrcholy pre každý deň a pre každý druh aktivity. Zo *source* pôjde hrana do každého dňa s váhou 1. Z každého dňa pôjdu hrany váhy 1 do tých aktivít, ktoré sa dajú v ten deň vykonávať. Tu si treba uvedomiť, že každý druh aktivity bude mať priradený nanajvýš jeden vrchol, takže týchto hrán už bude nanajvýš $4n$. Nakoniec pôjdu hrany z jednotlivých aktivít do *sink*, s váhou rovnou tomu, koľkokrát chce Samko danú aktivitu počas prázdnin vykonať. Ak chceme znížiť nejaké konštanty, vrcholy budeme vytvárať len pre tie aktivity, ktoré chce Samko vykonať počas prázdnin.

Maximálne toky

V takomto novom grafe nás bude zaujímať, koľko najviac vody vie tiecť zo *source* do *sink*. Obmedzenia sú nasledovné: Z každého vrchola okrem *source* a *sink* musí odtekať presne toľko vody, koľko doňho priteká a po každej hrane môže tiecť najviac toľko vody, aká je jej váha. Ľahko si všimneme, že každý validný tok zodpovedá nejakému konkrétnemu pareniu na pôvodnom grafe.[8]

Ako sa taký najväčší tok hľadá? Opäť budeme potrebovať zaviesť rafinovaný pojem zlepšujúcej cesty. K tomu budeme potrebovať do nášho orientovaného grafu prirobiť fiktívnu spätnú hranu ku každej normálnej hrane, na začiatku s kapacitou 0. V pôvodných hranách si budeme pamätať, koľko vody nimi môže **ešte** pretiecť, v spätných, koľko vody tečie pôvodnými hranami. Teraz budeme hľadať také cesty, ktoré začínajú v *source*, prechádzajú po hranách s nenulovými hodnotami a končia v *sink*.

Čomu to zodpovedá: Budeme hľadať nejaký spôsob, ako cez náš graf pretlačiť ďalší prúd veľkosti 1. Keď pôjdeme po obyčajných hranách, bude to zodpovedať tomu, že cez ne posielame nový prúd. Keď pôjdeme po spätnej hrane z u do v, znamená to, že sme si rozmysleli, kade tiekol prúd predtým z v a pošleme ho nejakým novým smerom.

Tieto zlepšujúce cesty môžme opäť hľadať starým osvedčeným prehľadávaním do hĺbky zo *source*. Akonáhle dôjdeme do *sink*, hodnoty všetkých hrán, po ktorých sme prešli, znížime o 1 a k nim opačným hranám naopak zvýšime hodnoty o 1. Rozmyslite si, že opäť platí príjemná vlastnosť, že ak nevieme nájsť zlepšujúcu cestu, znamená to, že náš tok je už naozaj najväčší možný.

Časová zložitosť takéhoto riešenia bude $O(p \cdot (v + h))$, kde p je veľkosť toku, čiže naša odpoveď, v a h sú počty vrcholov a hrán, pričom tentoraz vieme odhadnúť $v = n+k+2$, $h \leq 5n+k$, takže po vhodnom dosádzaní a prepisovaní je horný odhad $O((n+k)^2)$. Pamäťová zložitosť takéhoto riešenia bude $O(n+k)$[9].

Listing programu (C++)

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

#define For(i, n) for(int i = 0; i<int(n); ++i)
#define ForEach(it, T) for(auto it = T.begin(); it!=T.end(); ++it)
typedef vector<int> vi;
typedef vector<vi> vvi;

int n, d, k, a;
vi V; // Visited
vvi D,O,S,P; // Dni, Ostava_kapacita, Sused, oPacny_tok
map<int, int> M;

inline void makeEdge(int f, int t, int c){ //from, to, kapacita
    P[f].push_back(O[t].size());
    P[t].push_back(O[f].size());
    S[f].push_back(t);
    S[t].push_back(f);
    O[f].push_back(c);
    O[t].push_back(0);
}

bool najdi(int v){ //dfs na vrchole v
```

[7] Pre záujemcov

[8] Stačí sa pozrieť, po ktorých hranách medzi dňami a aktivitami tečie voda.

[9] Tu sa hodí ešte poznamenať, že pre všeobecné grafy existujú aj lepšie tokové algoritmy, ale vďaka malému hornému odhadu veľkosti toku nám v tomto prípade stačí aj takéto „pomalé“ riešenie.

```
    if (V[v]) return 0;
    V[v] = 1;
    if (v==1) return 1;
    For(i, S[v].size())
        if(O[v][i] > 0 && najdi(S[v][i])) {
            O[v][i]--;
            O[S[v][i]][P[v][i]]++;
            return 1;
        }
    return 0;
}

int flow(){ //hlada najvacsi tok
    int f = 0;
    while(true) {
        V.clear();
        V.resize(n,0);
        if(!najdi(0)) break;
        f++;
    }
    return f;
}

int main(){
    scanf("%d", &d);
    D.resize(d, vi(4));
    For(i, d) For(j, 4) scanf("%d", &D[i][j]);
    scanf("%d", &k);

    //na zapamatanie mnozstva aktivit pouzijeme mapu <cislo_akivity,doterajsi_pocet>
    For(i, k) {
        scanf("%d", &a);
        M[a]++;
    }

    //pocet vrcholov bude 2(source+sink) + pocet dni + pocet roznych aktivit
    n = 2+d+M.size();
    P.resize(n);
    S.resize(n);
    O.resize(n);

    //pridame hrany z aktivit do sinku, hodnotu mame ulozenu v mape
    //vrcholy tvorime len pre aktivity, ktore chce Samko spravit
    int p = 0;
    ForEach(it, M) {
        makeEdge(2+d+p, 1, it->second);

        //ked vyuzijeme to, kolko bolo aktivit, v mape prepiseme hodnotu
        //na cislo vrchola, aby sme ho neskor (*) vedeli zistit
        it->second = 2+d+p;
        p++;
    }

    //pridavame hrany k dnom
    For(i, d) {
        //zo source do dna, kapacity 1
        makeEdge(0, 2+i, 1);

        //z dna do kazdej aktivity (ktoru chce Samko vobec vykonat),
        //ktora sa da v dany den vykonat
        //Z (*) vieme, ze cislo vrchola pre danu aktivitu mame teraz v mape
        //Vsimnite si, ze nasobne hrany nam nic nepokazia,
        //lebo tok do dna nikdy nepresiahne 1.
        For(j, 4) if (M.count(D[i][j])) makeEdge(2+i, M[D[i][j]], 1);
    }
    printf("%d\n", flow());
}
```