Korešpondenčný seminár z programovania
XXXIII. ročník, 2015/16
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. série zimnej časti

1. Zlacnené disky

Dávid
(max. 6 b za popis, 4 b za program)

Úlohou bolo zistiť, v ktorý deň je najvýhodnejšie zaplatiť zvyšok dlhu.

Potrebujeme teda pre každý deň zistiť, koľko eur by Matúš celkovo zaplatil, ak by vyplatil zvyšok dlhu v tento deň. Potom nám stačí vybrať najskorší deň kedy je táto suma najnižšia.

Otázkou ostáva, ako počítať sumu v eurách, ktorú by zaplatil, ak by ukončil splácanie v i-ty deň. Táto suma sa skladá z dvoch častí:

Prvá časť je to, čo musel zaplatiť každý deň splácania – i-krát výška splátky, teda $i \cdot s$.

Druhá časť musí pokrývať nesplatený zvyšok ceny v taiwanských dolároch. Tento zvyšok vypočítame ako rozdiel ceny disku a súčtu hodnôt všetkých doterajších splátok. Ak si teda označíme kurz i-teho dňa ako k_i, zvyšok je $c - s \cdot (k_1 + k_2 + ... + k_i)$.

Môžeme si všimnúť, že ak poznáme zvyšok v i-ty deň, ľahko vieme vypočítať zvyšok v $i+1$ deň, a to tak že od zvyšku v i-ty deň odrátame hodnotu splátky v $i+1$ deň – $s \cdot k_{i+1}$[1].

Keď už poznáme zvyšok, ktorý treba pokryť, vydelíme ho kurzom na tento deň a získame sumu v eurách ktorú treba doplatiť. Nakoľko chceme platiť celočíselne, túto sumu ešte zaokrúhlime nahor.

Celkovú sumu, ktorú Matúš zaplatí, ak sa rozhodne doplatiť dlh v i-ty deň vieme teda vypočítať ako

$$i \cdot s + \left\lceil \frac{c - s \cdot (k_1 + k_2 + ... + k_i)}{k_i} \right\rceil$$

Keď poznáme celkovú sumu, stačí ju porovnať s doteraz najnižšou sumou ktorú sme vyrátali, a ak je lepšia, tak si uložíme číslo tohto dňa. Treba si dať pozor, aby sme výpočet zastavili potom, ako sa zvyšok stane záporným, lebo vtedy je už disk splatený.

Zložitosť

Ak postupujeme tak, ako sme popisovali vyššie, stačí nám v každom z n krokov prečítať hodnotu kurzu na tento deň a spraviť konštantne veľa jednoduchých výpočtov a porovnaní, ktoré majú konštantnú časovú zložitosť. Preto celková časová zložitosť bude $O(n)$.

Lepšie sa to dokonca ani nedá, keďže na vstupe máme kurzy na n dní, ktoré musíme načítať[2].

Pamäťová zložitosť je $O(1)$, pretože si stačí pamätať len zvyšok, ktorý treba zaplatiť z predošlého dňa a kurz v daný deň.

Listing programu (C++)

```
#include <iostream>
#include <cmath>

using namespace std;

int main(){
    int n, cena, splatka, best = 1234567890, den = 1, kurz;
    cin >> n >> cena >> splatka;
    for (int i=0; i<n; i++){
        cin >> kurz;
        cena -= kurz * splatka; // kolko taiwanskych dolarov zostava zaplatit

        // kolko eur by zaplatil ak skonci tento den
        int teraz = (i + 1) * splatka;
        if(cena>0) teraz += ceil(cena/kurz);
```

[1] Ak by sme pre každý deň zvlášť počítali súčet $(k_1 + k_2 + ... + k_i)$, potebovali by sme si kurzy pamätať v poli, no čo je horšie, výpočet by mohol trvať dlho. Pre n dní by sme postupne sčítavali $1, 2, 3, \ldots n$ čísel, teda celkovo by sme sčítali $1+2+3+\cdots+n = \frac{n(n-1)}{2} = O(n^2)$ čísel. Takýto program by úspešne stihol vyriešiť vstupy s n nanajvýš 10 000.

[2] Odhad zložitosti robíme všeobecne, teda uvažujeme najhorší možný prípad – ak je najvýhodnejšie doplatiť splátku v posledný deň, musíme načítať všetky kurzy.

```
        if (teraz < best){
            best = teraz;
            den = i + 1;
        }

        if (cena < 0) break;
    }
    cout << den << endl;
}
```

Listing programu (Pascal)

```
uses crt, math;
var n, cena, splatka, best, teraz, den, kurz, i:longint;
begin
    best := 1234567890;
    den := 1;
    read(n, cena, splatka);
    for i := 1 to n do
    begin
        read(kurz);
        dec(cena, kurz * splatka);

        teraz := i * splatka;
        if cena > 0 then inc(teraz, ceil(cena / kurz));

        if teraz < best then
        begin
            best := teraz;
            den := i;
        end;

        if cena < 0 then break;
    end;
    writeln(den);
end.
```

výskumný tím Katedry Sprchovania a Plávania
(max. 6 b za popis, 4 b za program)

2. Znova zašpinení programátori

Táto úloha sa dala riešiť viacerými spôsobmi. V tomto vzoráku si ukážeme jedno pomalé a dve rýchle riešenia.

Naivná simulácia

Budeme robiť presne to, čo nám hovorí zadanie. V jednom poli si budeme pamätať, v akom poradí sú gély na kope a postupne budeme simulovať sprchovanie programátorov. Vždy, keď sa nejaký programátor osprchuje, nájdeme jeho gél v našom poli, všetky gély nad ním posunieme o jednu pozíciu nižšie a náš hľadaný gél dáme na samý vrch. Na konci vypíšeme poradie, v akom sú gély na kope. Pri implementácii si treba dať pozor na to, aby sme gély posúvali v dobrom poradí, inak sa nám môže stať, že si jedným gélom postupne prepíšeme celú posúvanú časť poľa.

Zložitosť

Vždy, keď sa nejaký programátor osprchuje, budeme musieť posunúť všetky gély, ktoré sú v kope nad jeho gélom. V najhoršom prípade sa nám môže stať, že po každom sprchovaní budeme musieť posunúť všetkých n gélov. Časová zložitosť je teda $O(mn)$, pamäťová zložitosť je $O(n)$.

Listing programu (C++)

```
#include <cstdio>

int kopa[200000];

int main() {
    int n, m;
    scanf("%d %d",&n,&m);
    for(int i=0; i<n; i++)
        kopa[i] = i+1;

    for(int i=0; i<m; i++) {
        int aktualny;
        scanf("%d",&aktualny);
        int kde_je;
        for(int j=0; j<n; j++)
            if(kopa[j] == aktualny) // najdeme gel v kope
                kde_je = j;

        for(int j=kde_je; j>0; j--) // vsetky gely nad aktualnym klesnu
            kopa[j] = kopa[j-1];
        kopa[0] = aktualny;         // a aktualny pojde na vrch kopy
    }
```

```
    for(int i=0; i<n; i++) {
        printf("%d",kopa[i]);
        if(i < n-1) printf("␣");
        else printf("\n");
    }
}
```

Spájaný zoznam

Opäť budeme celý dej simulovať, ale informáciu o kope si budeme pamätať trochu šikovnejšie. Namiesto toho, aby sme si pamätali poradie všetkých gélov, budeme si pre každý gél pamätať, ktorý gél je v kope hneď pod ním a ktorý gél je hneď nad ním (prípadne, že daný gél je na úplnom vrchu, alebo spodku kopy). K tomu si budeme navyše pamätať číslo gélu, ktorý je aktuálne na vrchu kopy. Všimnime si, že keď presúvame gél z nejakého miesta na vrch kopy, väčšine gélov sa susedia nezmenia. Konkrétne, ak chceme presunúť gél A, pričom nad ním bol gél B, pod ním bol gél C a na vrchu kopy bol gél D, tak po presune pod gélom B už nebude A, ale C, nad gélom C bude gél B, nad gélom D bude A a gél A bude na vrchu kopy, pričom pod ním bude gél D. Nič iné už v pamäti meniť nepotrebujeme.

Na konci vypíšeme gél, ktorý je na vrchu kopy, potom gél tesne pod ním, potom gél, ktorý bol pod ním atď., až kým neprídeme na spodok kopy.

Aby sme pri implementácii nemuseli špeciálne ošetrovať spodok a vrch kopy, môžeme urobiť nasledovný trik: na spodok aj na vrch kopy si pridáme po jednom virtuálnom géli (s číslami 0 a $n+1$), s ktorými nikdy nebudeme hýbať. Vďaka tomu nikdy nebudeme musieť hýbať s gélami, ktoré sú na vrchu, alebo na spodku kopy. Potom si ani nebudeme musieť pamätať, ktorý gél je na vrchu skutočnej kopy, keďže to vždy bude gél tesne pod gélom 0.

Zložitosť

Na začiatku musíme nastaviť všetkým gélom susedov (v čase $O(n)$). Pri každom sprchovaní musíme zmeniť konštantne veľa premenných (dokopy v čase $O(m)$) a na konci musíme prejsť cez všetky gély a vypísať ich (v čase $O(n)$). Celková časová zložitosť je teda $O(m+n)$. Pamäťová zložitosť je $O(n)$

Listing programu (C++)

```
#include<cstdio>

int nad[200002], pod[200002];

int main() {
    int n, m;
    scanf("%d␣%d",&n,&m);

    for(int i=0; i<=n; i++) pod[i] = i+1;
    for(int i=1; i<=n+1; i++) nad[i] = i-1;

    for(int i=0; i<m; i++) {
        int aktualny;
        scanf("%d",&aktualny);
        pod[nad[aktualny]] = pod[aktualny];
        nad[pod[aktualny]] = nad[aktualny]; // vyberieme aktualny gel z kopy
        pod[aktualny] = pod[0];
        nad[aktualny] = 0;                 // pridame ho na vrch kopy
        nad[pod[0]] = aktualny;
        pod[0] = aktualny;                 // a aktuailzujeme gely na vrchu kopy
    }

    int gel = pod[0];
    for(int i=0; i<n; i++) {
        printf("%d", gel);
        if(i < n-1) printf("␣");
        else printf("\n");
        gel = pod[gel];
    }
}
```

Naspäť v čase

Nakoniec si ukážeme jedno riešenie, ktoré je myšlienkovo trochu náročnejšie, ale má jednoduchšiu implementáciu. Všimnime si, že ak sa programátor niekedy počas dňa osprchoval, potom všetci, ktorí sa sprchovali po ňom (presnejšie povedané, po tom, ako sa náš programátor osprchoval posledný raz), budú mať svoje gély v kope nad jeho gélom a všetci ostatní budú mať gély pod jeho gélom. Inými slovami, na konci bude na vrchu kopy gél programátora, ktorý sa sprchoval ako posledný. Pod ním bude gél programátora, ktorý sa sprchoval ako predposledný, atď.. Gély programátorov, ktorí sa nesprchovali, budú úplne na spodku kopy v rovnakom poradí, ako boli na začiatku.

Úlohu teda vyriešime nasledovne: budeme prechádzať zoznamom sprchovaní **v opačnom poradí**, ako bol na vstupe. Vždy, keď nájdeme nejakého programátora, ktorého sme ešte nevideli (teda sme našli posledné

sprchovanie tohto programátora), vypíšeme jeho číslo. Keďže všetkých programátorov, ktorí sa sprchovali po ňom, sme už videli (a teda aj vypísali), tak sa nám nestane, že by sme niektoré číslo vypísali príliš skoro. Nakoniec ešte prejdeme všetky gély v poradí, v akom boli na začiatku (teda od 1 do n) a vypíšeme tie, ktoré sme ešte nevypísali.

Zložitosť

Musíme načítať a prejsť m-prvkový zoznam sprchovaní a potom ešte n-prvkový zoznam gélov. Časová zložitosť je teda $O(m+n)$. Potrebujeme si pamätať celý zoznam sprchovaní (lebo ho budeme prechádzať v opačnom poradí, ako bol na vstupe) a pre každého programátora si potrebujeme pamätať, či sme ho už videli (či sme už vypísali číslo jeho gélu). Preto aj pamäťová zložitosť je $O(m+n)$.

Listing programu (C++)

```
#include <cstdio>

bool videl_som[200001];
int sprchovanie[200000];
int vypisanych = 0;

int main()
{
    int n, m;
    scanf("%d %d",&n,&m);
    for(int i=1; i<=n; i++)
        videl_som[i] = 0;
    for(int i=0; i<m; i++)
        scanf("%d",&sprchovanie[i]);

    for(int i=m-1; i>=0; i--) {
        if(!videl_som[sprchovanie[i]]) {
            printf("%d",sprchovanie[i]);
            vypisanych++;
            if(vypisanych < n) printf(" ");
            else printf("\n");
            videl_som[sprchovanie[i]] = 1;
        }
    }
    for(int i=1; i<=n; i++) {
        if(!videl_som[i]) {
            printf("%d",i);
            vypisanych++;
            if(vypisanych < n) printf(" ");
            else printf("\n");
        }
    }
}
```

Baška

3. Zygrov dlhoročný sen . . .

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

Ciferný súčet

Na začiatok si povieme, ako zistiť ciferný súčet čísla c. Na výpočet ciferného súčtu použijeme operáciu modulo – zvyšok po delení. Posledná cifra čísla c je zvyšok po delení číslom 10, alebo $c \mod 10$. Vydelením čísla c číslom 10 ho „skrátime" o poslednú cifru.

Rovnakú myšlienku preto môžeme použiť aj na predposlednú cifru, predpredposlednú cifru... Postup budeme opakovať, až kým nebudeme vedieť všetky cifry. Skončíme vtedy, keď sa nám číslo zmenší na nulu. Cifry čísla c si budeme postupne sčítavať do premennej.

Listing programu (C++)

```
long long ciferny_sucet(long long cislo){
    long long sucet = 0;
    while (cislo > 0) {
        long long posledna_cifra = cislo % 10;
        sucet += posledna_cifra;
        cislo /= 10;
    }
    return sucet;
}
```

Úloha

Zázračné platenie fungovalo nasledovne: Nech je skutočná cena nákupu c. Zygro pri zázračnom platení zaplatí $ciferny\_sucet(c)$ peňazí a teda pri tomto nákupe ušetrí $n = c - ciferny\_sucet(c)$ peňazí.

Úlohou bolo nájsť každé číslo c, pre ktoré platí, že ak od neho odčítame jeho ciferný súčet, tak dostaneme číslo na vstupe, teda n.

Nech máme zadanú ušetrenú sumu n. Keďže vieme, že každá pôvodná cena c musela byť aspoň tak veľká ako číslo n, môžeme úlohu riešiť tak, že postupne skúšame všetky čísla $c \geq n$ a testujeme, či po odčítaní ich ciferného súčtu dostaneme číslo n. Otázkou však zostáva, kedy môžeme prestať skúšať ďalšie, väčšie, čísla c.

Horné ohraničenie na hodnotu c

Môžeme si všimnúť, že za nákup v hodnote c môžeme zaplatiť najviac $9 \cdot len(c)$ peňazí, kde $len(c)$ je počet cifier čísla c. Vždy teda ušetríme aspoň $c - 9 \cdot len(c)$ peňazí, preto platí nerovnica:

$$n \geq c - 9 \cdot len(c)$$

Z tohto zápisu vieme odvodiť horné ohraničenie pre hodnotu c

$$c \leq n + 9 \cdot len(c)$$

potrebujeme však odhadnúť $len(c)$ pomocou n. Na rýchlostnej programátorskej súťaži by sme si zvolili nejakú hodnotu, ktorá by určtite stačila pre nájdenie všetkých možných riešení (napr.: $len(c) < 2 \cdot len(n)$). Nezhoršili by sme tak asymptotickú časovú zložitosť a riešenie by sme mali naprogramované rýchlo. Pre úplnosť si ale dokážeme presnejší odhad.

Aká je teda dĺžka čísla c vzhľadom na dĺžku čísla n? Ukážeme si, že dĺžky týchto dvoch čísel sa môžu líšiť najviac o jedna.

- Ak je c jednociferné, po odčítaní ciferného súčtu sa dĺžka výsledku nezmení, n má teda tiež jednu cifru.
- Ak je c dvojciferné, $ciferny\_sucet(c) \leq 18$. Pre všetky čísla $c \geq 28$ sa po odčítaní $ciferny\_sucet(c)$ jeho dĺžka nezmení, takže n bude tiež dvojciferné. No a ak je c menšie ako 28, výsledné číslo n bude možno jednociferný (napr. pri číslach 11 až 19).
- Ak je c k-ciferné, $ciferny\_sucet(c) \leq 9k$. Pre všetky $c \geq 10^{k-1} + 9k$, odčítaním ciferného súčtu jeho dĺžku nezmeníme. Inak môže byť n len o 1 cifru kratšie. Na skrátenie o dve cifry by totiž muselo platiť, že ciferný súčet je aspoň $9 \cdot 10^{k-2}$.[3] Dostaneme teda nasledovnú nerovnicu: $9 \cdot 10^{k-2} < ciferny\_sucet(c) \leq 9k$. Táto nerovnica však zjavne nie je splnená pre k väčšie alebo rovné trom.

Vieme teda, že po odčítaní ciferného súčtu z čísla c nemôže vzniknúť číslo s počtom cifier zmenšeným o 2. Toto môžeme zapísať ako:

$$len(c) - len(n) = len(c) - len(c - ciferny\_sucet(c)) \leq 1$$

čím dostávame odhad $len(c) \leq 1 + len(n)$ a teda spojením týchto dvoch úvah môžeme ohraničiť c pomocou n:

$$c \leq n + 9 \cdot (1 + len(n))$$

Posledný krok, ktorý ešte musíme spraviť, je zistenie dĺžky čísla v desiatkovom zápise.

Môžeme použiť funkciu podobnú tej, čo rátala ciferný súčet. Kým je číslo väčšie ako 0, budeme ho deliť 10 a zväčšovať si počítadlo počtu cifier.

Iný spôsob je využiť desiatkový logaritmus. Pre všetky kladné celé čísla platí $len(n) = \lfloor \log_{10}(n) \rfloor + 1$, kde $\lfloor n \rfloor$ je dolná celá časť n. Pre 0 pridáme špeciálny prípad, keďže počet cifier čísla nula je 1, čo nesedí s naším vzorčekom. Horná hranica pre c bude teda $n + 9 \cdot (\lfloor \log_{10}(n) \rfloor + 2)$.

Listing programu (C++)

```
#include<iostream>
#include<cmath>

using namespace std;

long long ciferny_sucet(long long cislo){
    long long sucet = 0;
    while (cislo > 0) {
        long long posledna_cifra = cislo % 10;
        sucet += posledna_cifra;
        cislo /= 10;
    }
    return sucet;
```

[3] Ak chceme zo 6-ciferného čísla (najmenšie je 100,000) dostať číslo štvorciferné (najväčšie je 9,999), musíme odčítať aspoň 90001).

```
}

int main() {
    long long z;
    cin>>z;

    for (int i = 0; i<z; ++i) {
        long long n;
        cin>>n;
        long long horna = 10;
        if (n > 0) horna = 9*int(log10(n)+2);
        bool je_riesenie = false;
        for (long long c = n; c <= n+horna; ++c) {
            if (c - ciferny_sucet(c) == n) {
                if (je_riesenie) {
                    cout << " ";
                }
                je_riesenie = true;
                cout << c;
            }
        }
        cout << endl;
    }
    return 0;
}
```

Zložitosť

Ciferný súčet čísla c zistíme v čase $O(\log_{10}(c))$. Avšak, c a n má takmer rovnako veľa cifier (c má nanajvýš o jednu viac), preto je to zároveň čas $O(\log_{10}(n))$. Pre zadané n skúšame hodnoty od n po $n + 9(\lfloor \log_{10}(n) \rfloor + 2)$, teda dokopy $9\lfloor \log_{10}(n) + 2 \rfloor$ čísel. To znamená, že vypočítať riešenie pre jedno n trvá $O(\log^2_{10}(n))$. No a na vstupe máme z záznamov, preto je výsledná časová zložitosť $O(z \cdot \log^2_{10}(n))$.

Pamäťová zložitosť je $O(1)$, pretože máme iba konštantný počet premenných a výsledky môžeme rovno vypisovať.

Rýchlejšie, matematické riešenie

Na túto úlohu sa ale dá pozrieť aj čisto matematicky. Povedzme, že by mal Zygro zaplatiť sumu c. Toto číslo si môžeme zapísať pomocou jeho cifier ako

$$c = 1 \cdot c_0 + 10 \cdot c_1 + 100 \cdot c_2 \ldots$$

pričom c_i je i-ta cifra čísla c. Koľko je potom $n = c - ciferny\_sucet(c)$? Ciferný súčet je len súčet cifier a teda dostávame:

$$n = 1 \cdot c_0 + 10 \cdot c_1 + 100 \cdot c_2 \ldots - (c_0 + c_1 + c_2 + \ldots) = (10 - 1) \cdot c_1 + (100 - 1) \cdot c_2 \ldots$$

Z tohto čísla potom potrebujeme zrekonštruovať pôvodné c, teda jeho cifry. Vidíme, že n nám o nultej cifre c absolútne nič nepovie. To znamená, že posledná cifra m môže byť ľubovoľná. Ak teda nejaké riešenie existuje, je ich rovno 10.

Ďalej potrebujeme nájsť cifry $c_1, c_2, \ldots c_q$, pre ktoré platí vyššie uvedená rovnosť $n = (10 - 1) \cdot c_1 + (100 - 1) \cdot c_2 + \ldots + (10^q - 1) \cdot c_q$, kde q je počet cifier c, čo vieme, že je najviac počet cifier n plus jedna.

Algoritmus je potom nasledovný. Zoberieme n a nájdeme q. Zistíme si celočíselný podiel $\frac{n}{10^q - 1}$, čím dostaneme c_q. Od n odčítame $(10^q - 1) \cdot c_q$ a zmenšíme q o jedna. Takto pokračujeme, až kým q neklesne na nulu, čo nastane po $O(\log n)$ krokoch.

Posledné, čo musíme skontrolovať, (môžeme to robiť aj počas rátania jednotlivých cifier) je zistiť, či sú nájdené čísla $c_1, c_2, \ldots c_q$ naozaj cifry – čísla od 0 po 9. Ak sú všetky naozaj ciframi, zistili sme všetky, okrem nultej, ktorá ale môže byť ľubovoľná. Stačí už len vypísať.

Vhodným udržiavaním si potrebných čísel vieme úlohu pre jedno n vyriešiť v čase $O(\log(n))$ s pamäťou $O(1)$. Celkové riešenie má potom časovú zložitosť $O(z \log(n))$ a pamäťovú zložitosť $O(1)$.

O počte cifier a odhade časovej zložitosti

Mnohí z vás radi v riešeniach prehlasovali počet cifier každého n za konštantu, a teda aj časovú zložitosť za konštantnú.

Odhad časovej zložitosti je veľmi praktická vec. Robíme ho preto, aby sme vedeli približne predpovedať (ešte pred naprogramovaním), ako dlho bude program bežať, keď mu budeme dávať rôzne veľké vstupy. V našej úlohe je n jedným z dvoch parametrov. Bolo by teda veľmi *praktické* vedieť, čo sa bude diať keď ho budeme meniť. Preto je odhad $O(\log n)$ informatívnejší.

Samozrejme, keďže n mohlo mať najviac 18 cifier, tak to, či $n = 12$ alebo $n = 123456789012345678$ ovplyvní čas behu programu minimálne. Preto môžeme prakticky tvrdiť, že beh programu nezávisí od veľkosti n (teda, že má program konštantnú časovú zložitosť). Takéto zdôvodnenie ale treba v popise riešenia vždy spomenúť.

Ak by sme úlohu riešili pre veľaciferné čísla, nemohli by sme používať štandartné aritmetické operácie (napr. v C++) alebo by čas výpočtu týchto operácií veľmi závisel od veľkosti čísel (Python, vlastná aritmetika s veľkými číslami). Takýto program by si vyžadoval iné odhady zložitosti.

Mišo

4. Zajtra dávam výpoveď

(max. 9 b za popis, 6 b za program)

Táto úloha sa dala riešiť viacerými spôsobmi. Ukážeme si výhody a nevýhody riešení, ktoré ste submitovali. Prvé z nich je nedostatočné, druhé vtipné ale správne, a tretie je správne a elegantné.

Hlavná myšlienka za všetkými riešeniami je nasledovná: postupne vypočítať výsledok po jednotlivých cifrách. To znamená, že najprv si vygenerujeme jednociferné číslo, ktoré spĺňa prvú požiadavku, potom k nemu pridáme ďalšiu cifru tak, aby spĺňalo druhú požiadavku, a tak ďalej. Týmto spôsobom postupne splníme všetky požiadavky, a keďže tie požadujú len deliteľnosť číslom od 1 po 10, tak sa nikdy nestane, že by nebolo možné požiadavku splniť. Toto tvrdenie si neskôr dokážeme.

Teraz sa postupne pozrieme na všetky spomínané riešenia.

Skúšanie cifry

Toto je najjednoduchšie riešenie. Výsledok máme uložený v číselnej premennej a vždy, keď dostaneme novú požiadavku, pridáme k nemu novú cifru tak, aby sme požiadavku splnili.

Urobíme to tak, že budeme postupne skúšať všetky cifry od 0 po 9. Cifru pridáme k výsledku tak, že ho vynásobíme desiatimi a pripočítame cifru. Pridáme prvú (najmenšiu) cifru, ktorá spraví výsledok deliteľný číslom z požiadavky.

Listing programu (Python)

```
n = int(input())

vysledok = 0

# postupne spracujeme všetky požiadavky
# input().split() načíta riadok vstupu a rozdelí ho podľa medzier
for poziadavka in input().split():
    poziadavka = int(poziadavka)

    # vyskúšame všetky cifry
    for cifra in range(0, 9+1):

        # prvá cifra nemôže byť nula, preto musíme tento prípad ošetriť
        if vysledok == 0 and cifra == 0:
            continue

        # pridáme cifru na koniec výsledku
        novy_vysledok = vysledok * 10 + cifra

        # ak je to deliteľné číslom z požiadavky, pridáme túto cifru
        if novy_vysledok % poziadavka == 0:
            vysledok = novy_vysledok
            break

print(vysledok)
```

Jedna nevýhoda tohoto riešenia je zjavná: požiadaviek na vstupe môže byť až 200 000. Keďže počet cifier výsledného čísla sa rovná počtu požiadaviek, tak je jasné, že ukladať si výsledok v číselnej premennej nie je veľmi dobrý nápad. Ani do 64-bitovej premennej (napríklad `long long int` v C++) sa nezmestí viac ako 20-ciferné číslo.

Samozrejme, máme tu Python, a v ňom sú predsa číselné premenné neobmedzené, nie? Áno, ale zistiť zvyšok po delení sedmičkou 200 000 cifernému číslu dá zabrať aj Pythonu a program nestihne vypočítať výsledok v časovom limite.

Takéto riešenie mohlo dostať najviac 4 body.

Zložitosť

V tomto riešení n-krát počítame zvyšky po pridaní každej možnej cifry. Počítanie zvyšku po delení má časovú zložitosť $O(n)$[4], kde n je počet cifier čísla, z ktorého robíme zvyšok. Ostatné operácie, ktoré v programe robíme nemajú časovú zložitosť horšiu ako $O(n)$.

Preto je časová zložitosť tohoto programu $O(n \cdot 10 \cdot n)$, čo sa rovná $O(n^2)$.

Keďže si uchovávame celé výsledné číslo a toto číslo má na konci n cifier, na jeho uloženie potrebujeme $O(n)$ pamäte.

[4] To, že operácie s číslami trvajú konštantne dlho môžeme tvrdiť vtedy, ak sú tieto čísla ohraničené konštantou, teda nezávisia od veľkosti vstupu.

„Orlojové" riešenie

Z predchádzajúceho riešenia je jasné, že potrebujeme vymyslieť niečo lepšie. Slabina spomínaného riešenia je v tom, že zakaždým potrebujeme vypočítať zvyšok po delení celého výsledku, čo, okrem iného, spôsobuje zlú časovú zložitosť.

Jednoduchým vylepšením bude, že si o našom výsledku budeme pamätať nejaké informácie, aby sme rýchlo vedeli vypočítať zvyšok po delení. V riešení, ktoré viete vymyslieť na základe toho, čo ste sa učili na hodinách matematiky, si môžeme pamätať takéto informácie:

- paritu, pre deliteľnosť 2
- ciferný súčet, pre deliteľnosť 3, 9
- posledné trojčíslie, pre deliteľnosť 4, 5, 8, 10
- deliteľnosť 6 sa dá odvodiť z deliteľnosti 2 a 3
- zvyšok po delení 7, pre deliteľnosť 7

Keď si pamätáme tieto informácie, výsledok vieme generovať po cifrách tak, že postupne vyskúšame všetky cifry od 0 po 9 a na základe týchto informácii zistíme, či bude výsledok po pridaní danej cifry deliteľný číslom z požiadavky. Ak bude, tak cifru vypíšeme a aktualizujeme všetky tieto informácie.

Listing programu (Python)

```
# kód tohto programu nemusíte čítať, stačí, ak kriticky zhodnotíte jeho eleganciu

n = int(input())

vysledok = []

parne = True
ciferny_sucet = 0
posledne_trojcislie = 0
zvysok_po_7 = 0

for poziadavka in input().split():
    poziadavka = int(poziadavka)

    for cifra in range(0, 9+1):
        if len(vysledok) == 0 and cifra == 0:
            continue

        # vypočítame si, aké by boli všetky informácie, keby sme použili danú cifru
        nove_parne = cifra % 2 == 0
        novy_ciferny_sucet = (ciferny_sucet + cifra) % 9 # potrebujeme len jeho zvyšok po delení 3 a 9
        nove_posledne_trojcislie = (posledne_trojcislie * 10 + cifra) % 1000
        novy_zvysok_po_7 = (zvysok_po_7 * 10 + cifra) % 7

        # je nové číslo deliteľné číslom z požiadavky?
        dobra_cifra = False

        if poziadavka == 1:
            dobra_cifra = True
        elif poziadavka == 2:
            dobra_cifra = nove_parne
        elif poziadavka == 3:
            dobra_cifra = novy_ciferny_sucet % 3 == 0
        elif poziadavka == 4:
            dobra_cifra = nove_posledne_trojcislie % 4 == 0
        elif poziadavka == 5:
            dobra_cifra = nove_posledne_trojcislie % 5 == 0
        elif poziadavka == 6:
            dobra_cifra = nove_parne and novy_ciferny_sucet % 3 == 0
        elif poziadavka == 7:
            dobra_cifra = novy_zvysok_po_7 == 0
        elif poziadavka == 8:
            dobra_cifra = nove_posledne_trojcislie % 8 == 0
        elif poziadavka == 9:
            dobra_cifra = novy_ciferny_sucet % 9 == 0
        elif poziadavka == 10:
            dobra_cifra = cifra == 0

        if dobra_cifra:
            vysledok.append(cifra)

            parne = nove_parne
            ciferny_sucet = novy_ciferny_sucet
            posledne_trojcislie = nove_posledne_trojcislie
            zvysok_po_7 = novy_zvysok_po_7

            break

# nasledujúci príkaz vypíše vypočítané cifry výsledku pospájané do jedného reťazca
# funkcia str.join(values) vytvorí nový string tak, že string str postrká medzi všetky hodnoty values
# náš zoznam obsahuje inty, teda ich musíme najprv skonvertovať funkciou map
print(''.join(map(str, vysledok)))
```

Zložitosť

Keďže premenné, v ktorých si udržujeme vyššie popísané informácie dosahujú maximálne hodnoty nezávislé od veľkosti vstupu, tak práca s nimi je v konštantnom čase. Keďže musíme spracovať všetkých n požiadaviek, tak časová zložitosť je $O(n)$.

Cifry výsledku môžeme vypisovať postupne, jednu po druhej, bez toho, aby sme si ich ukladali, teda pamäťová zložitosť môže byť až konštantná, $O(1)$[5].

Zvyšky – pôjde to aj jednoduchšie

Predošlé riešenie som nazval „orlojové", lebo je zbytočne zložité. Pamätáme si 4 odlišné informácie a implementácia tohto riešenia obsahuje množstvo `if - else` alebo `case` príkazov – špeciálne prípady, v ktorých sa dá ľahko pomýliť.

Skúsme teda zjednodušiť myšlienku a tým aj implementáciu. Pre deliteľnosť sedmičkou sme si pamätali zvyšok po delení sedmičkou... A nedá sa to tak spraviť pre všetky cifry? Áno, dá! V tomto jednoduchšom riešení si budeme pamätať zvyšky po delení všetkými číslami od 1 po 10. Pre novú požiadavku skúsime postupne pridávať novú cifru, a vyberieme najmenšiu vyhovujúcu. V predošlom riešení tak stačí nahradiť `if - else` príkazy jedným `for`-cyklom a namiesto počítania 4 špecifických premenných spočítame v jednom `for`-cykle nové zvyšky po delení číslami 1 až 10, uložené v poli.

Listing programu (Python)

```
n = int(input())

vysledok = []

# v zozname si pamätáme zvyšky po všetkých číslach od 1 po 10
zvysok = [0] * 11

for poziadavka in input().split():
    poziadavka = int(poziadavka)

    cifra = None

    # ošetríme prvú cifru
    if len(vysledok) == 0:
        cifra = poziadavka

    # zistíme, akú cifru potrebujeme pridať
    else:
        for i in range(0, 9+1):
            if (zvysok[poziadavka] * 10 + i) % poziadavka == 0:
                cifra = i
                break

    vysledok.append(cifra)

    # aktualizujeme všetky zvyšky
    for i in range(1, 10+1):
        zvysok[i] = (zvysok[i] * 10 + cifra) % i

print(''.join(map(str, vysledok)))
```

Zložitosť

S časovou zložitosťou je to v tomto riešení rovnaké ako v prechádzajúcom, „orlojovom". Máme pole s konštantnou veľkosťou, ktorého prvky môžu nadobúdať len ohraničené hodnoty, nezávislé od n. Práca s nimi je teda v konštantnom čase, a keďže spracúvame všetky požiadavky, tak celková časová zložitosť je $O(n)$.

Pamäťová zložitosť je na tom rovnako, ako v prechádzajúcom riešení, ak by sme cifry vypisovali jednu po druhej, bola by konštantná, $O(1)$, ale keďže to kvôli výkonu nerobíme, tak je lineárna, $O(n)$.

Počítanie namiesto skúšania možností

Na záver si môžeme uvedomiť, že keď poznáme zvyšky po delení všetkými potrebnými číslami, už nepotrebujeme skúšať všetky cifry. Potrebnú cifru vieme vypočítať!

Výpočtom ďalšej cifry tiež dokážeme, že pre ľubovoľné číslo a ľubovoľnú požiadavku existuje cifra, ktorú vieme pridať. Toto tvrdenie sme využívali vo všetkých predošlých úvahách, no neboli sme presvedčení o tom, že skutočne platí.

Čo musí spĺňať cifra, ktorú chceme pridať? Výsledok po pridaní tejto cifry musí byť deliteľný číslom z požiadavky. Musí teda platiť, že nový zvyšok po delení číslom požiadavky sa rovná nule. Ak Z_i je zvyšok doteraz napísaného čísla po delení i, p je požiadavka a c je cifra, ktorú pridávame, tak musí platiť toto:

[5] V Pythone je ale rýchlejšie vypísať celý riadok naraz, než vypisovať po znakoch, a teda je lepším rozhodnutím pamätať si celé číslo a na konci ho vypísať

$$(Z_p \cdot 10 + c) \bmod p = 0$$

Z tohto vzorca si jednoducho odvodíme, ako vypočítať želanú cifru:

$$c = (-Z_p \cdot 10) \bmod p$$

Ak by ste mali vy alebo váš programovací jazyk problém počítať zvyšky záporných čísel, môžeme si uvedomiť, že $(-Z_p \cdot 10) \bmod p = (p \cdot 10 - Z_p \cdot 10) \bmod p$.

Iný spôsob, ako sa dopracovať k c je pouvažovať: Doteraz napísané číslo má zvyšok Z_p. Po pridaní cifry 0 bude mať zvyšok $(Z_p \cdot 10) \bmod p$. O koľko potrebujeme zväčšiť pridanú cifru, aby sme zvyšok $(Z_p \cdot 10) \bmod p + c$ dostali na nulu? Jednoducho o $p - (Z_p \cdot 10) \bmod p$. Ak však $(Z_p \cdot 10) \bmod p = 0$, nechceme pridať cifru p, ale cifru 0 a to dosiahneme takto ďalším zmodulovaním p:

$$c = (p - (Z_p \cdot 10) \bmod p) \bmod p$$

Počítanie novej cifry nám prinesie len zrýchlenie o konštantu, no dôležitá je myšlienka, že nová cifra sa vždy dá pridať. Ak by napríklad mohla byť požiadavka 11, mohli by sme sa dostať do situácie, že doteraz napísané číslo má $(Z_{11} \cdot 10) \bmod 11 = 1$. V takomto prípade by žiadna cifra výsledný zvyšok nedotiahla na nulu.

Optimalizované zvyšky

Riešenie so zvyškami je úplne dobré. Avšak, ak sme veľkí fajnšmekri, môžeme riešenie ešte raz konštantne zrýchliť. Nemusíme si zvyšky pamätať jednotlivo v zozname, stačí, ak si budeme pamätať zvyšok po najmenšom spoločnom násobku čísel 1 až 10 a z toho si vždy vieme vypočítať každý potrebný zvyšok. Takéto riešenie dáva v Pythone dvakrát až trikrát lepšie časy než riešenie s poľom zvyškov.

Listing programu (Python)

```
nasobok = 2 * 2 * 2 * 3 * 3 * 5 * 7

n = int(input())
vysledok = []

zvysok = 0

for poziadavka in input().split():
    poziadavka = int(poziadavka)

    cifra = None
    if len(vysledok) == 0:
        cifra = poziadavka
    else:
        cifra = (poziadavka - zvysok * 10) % poziadavka

    vysledok.append(cifra)
    zvysok = (zvysok * 10 + cifra) % nasobok

print(''.join(map(str, vysledok)))
```

Žaba

5. Optimálna šifrovačka

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

Zopakujme si, ako znie zadanie. Na vstupe máme niekoľko tatranských križovatiek, každú v inej nadmorskej výške. Medzi nimi vedie niekoľko rôzne dlhých chodníkov. Našou úlohou je nájsť čo najdlhšiu cestu, ktorá celý čas klesá.

Celú situáciu si môžeme predstaviť ako graf. Križovatky sú vrcholy a chodníky predstavujú hrany medzi nimi. Uvedomme si však, že po každej hrane môžeme chodiť len jedným smerom – v smere od vyššie položeného vrchola k tomu nižšiemu. Takéto hrany nazývame orientované. Takisto v tomto grafe nemôžu existovať orientované cykly, lebo by to znamenalo, že v nejakom momente sme porušili podmienku klesania. Graf, ktorý nám zaručuje zadanie, je preto orientovaný a acyklický a zvykne sa označovať ako DAG[6].

Skúsme sa teraz pozrieť na to, ako vyzerajú najdlhšie cesty v ňom. Je jasné, že musia začínať vo vrchole, do ktorého nevedie žiadna hrana. V opačnom prípade by sme predsa začali o kúsok vyššie a cestu si predĺžili. Takisto, končiť bude v nejakom vrchole, z ktorého už žiadna hrana nevedie. Oveľa viac však zaručiť nevieme. To však nevadí, pokúsme sa vymyslieť aspoň nejaké prvé riešenie.

[6] Z anglického directed acyclic graph.

Hľadanie všetkých ciest

Prvá možnosť, ktorá nás môže napadnúť, je vyskúšať všetky možné cesty, ktoré v našom grafe existujú a vybrať z nich tú, ktorá bude mať najväčšiu dĺžku. Myšlienkovo je to samozrejme správne riešenie, aj keď tušíme, že nebude až také rýchle. Keď sa ale zamyslíme, ako by sa niečo také dalo naprogramovať, zistíme, že to nebude také jednoduché. Ukážme si preto, ako naprogramovať toto prvé riešenie a potom si ukážeme, ako ho môžeme zlepšiť.

Ako vôbec vyzerá nejaká cesta? Začneme v niektorom najvyššom vrchole. Odtiaľ pôjdeme jednou z jeho hrán do susedného vrchola. Odtiaľ si opäť vyberieme niektorú z hrán, ktoré z neho vedú a takto budeme pokračovať, až kým sa nedostaneme na spodok. Uvedomme si však, že v každom vrchole, v ktorom sa ocitneme, sme v úplne rovnakej situácii. Chceme sa z neho presunúť všetkými možnými spôsobmi do jeho susedov a odtiaľ pokračovať v hľadaní.

Táto myšlienka je teda vo svojej podstate rekurzívna a to vieme patrične využiť a tak ju aj implementovať. Spravíme si teda funkciu `najdi(v)` s jedným parametrom. Táto funkcia bude mať nasledovnú úlohu: **Prejdi všetky cesty, ktoré vedú z vrchola v.**

Nech má vrchol v susedov $w_1, w_2 \dots w_k$. Ak chceme prejsť všetkými cestami, ktoré vedú z vrchola v, tak musíme ísť z vrchola v do vrchola w_1 a potom prejsť všetky cesty z tohto vrchola, čo je vlastne len rekurzívne volanie `najdi(w1)`. No a potom musíme tiež ísť z v do vrchola w_2 a odtiaľ prejsť všetko, čo je volanie `najdi(w2)`. A tak ďalej pre všetkých susedov vrchola v.

Nás však nezaujímajú všetky cesty, ale len tá najdlhšia. Teda chceme **prejsť všetkými cestami vedúcimi z vrchola v a zistiť dĺžku najdlhšej z nich**. Je viacero spôsobov, ako vypočítať túto hodnotu. Mohli by sme si dĺžku aktuálne objavovanej cesty posúvať ako ďalší parameter našej funkcie. Mohli by sme si značiť prejdené vrcholy do pomocného poľa a vždy na spodku cesty pomocou neho vyrátať jej dĺžku. Nám sa však bude hodiť (a podľa mňa je to najelegantnejší spôsob), keď si ju budeme vracať ako návratovú hodnotu tejto funkcie. Naša funkcia `najdi(v)` bude teda vracať dĺžku najdlhšej cesty, ktorá začína vo vrchole v.

Túto hodnotu nie je problém vyrátať. Je to maximum z nasledovných hodnôt: dĺžka hrany z v do w_i plus dĺžka najdlhšej cesty z vrchola w_i, čo je `najdi(wi)`, postupne pre všetky možné i. Vďaka tomu dostávame prvé riešenie, ktorého program nájdete nižšie. Odporúčam podrobnejšie si ho preštudovať.

Listing programu (C++)

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

int n, m;
vector<vector<pair<int, long long> > > Graf;
vector<int> Vysky;
vector<int> pocet_vchadzajucich_hran;

long long najdi(int v) {
    long long najdlhsia=0;
    // pre vsetkych susedov
    for(int i=0; i<Graf[v].size(); i++) {
        int w = Graf[v][i].first;
        int c = Graf[v][i].second;
        najdlhsia = max(najdlhsia, c+najdi(w));
    }
    return najdlhsia;
}

int main() {
    scanf("%d %d", &n, &m);
    Vysky.resize(n);
    Graf.resize(n);
    pocet_vchadzajucich_hran.resize(n);
    for(int i=0; i<n; i++)
        scanf(" %d", &Vysky[i]);
    for(int i=0; i<m; i++) {
        int x, y, c;
        scanf(" %d %d %d", &x, &y, &c);
        x--; y--;
        if(Vysky[x] < Vysky[y]) swap(x, y);
        Graf[x].push_back(make_pair(y, c));
        pocet_vchadzajucich_hran[y]++;
    }
    long long vysledok=0;
    // pre vsetky vrcholy, do ktorych nic nevchadza
    for(int i=0; i<n; i++)
        if(pocet_vchadzajucich_hran[i] == 0)
            vysledok = max(vysledok, najdi(i));
    printf("%lld\n", vysledok);
}
```

Memoizácia

Po odovzdaní tohto riešenia zistíme, že je síce korektné, lebo na prvom vstupe prejde, ale strašne pomalé

a na zvyšných prekročí časový limit. To ale neznamená, že postupujeme zle. Možno stačí malá úprava, aby sa naše riešenie zmenilo na vzorové. A ako uvidíte, od vzorového riešenia sa toto skúšanie všetkých možností líši v štyroch riadkoch.

Takmer vždy, keď máme správne, ale pomalé riešenie, (a už tobôž keď ho riešime pomocou rekurzívnej funkcie) oplatí sa položiť si nasledovné otázky: Nepočítame niečo zbytočne? Nepočítame niečo viackrát?

Predstavme si, že máme graf, v ktorom z vrchola 1 vedie strašne dlhá cesta – bez odbočiek, jednoducho stále iba jedna možnosť ako sa pohnúť ďalej. Naviac nech existuje veľa vrcholov, do ktorých nevedie žiadna hrana a vychádza z nich jediná hrana do vrchola 1.

Výpočet nášho programu na tomto grafe vyzerá nasledovne: začnem v jednom z vrchných vrcholov, posuniem sa do vrchola 1, pričom sa zavolá funkcia `najdi(1)`. Tá zistí, že najdlhšia cesta vedúca z vrchola 1 má dĺžku x, lebo celú túto cestu prejde. Potom si zoberieme druhý vrchol, do ktorého nič nevedie a opäť sa presunieme do vrchola 1. Potom sa opäť zavolá funkcia `najdi(1)`, ktorá vo svojich volaniach prejde celú tú dlhočiznú cestu nadol, aby nám povedala to, čo už dávno vieme. Teda, že najdlhšia cesta z vrchola 1 má dĺžku x. A toto sa zopakuje ešte niekoľkokrát.

Môžeme si teda všimnúť, že vždy, keď v našom programe zavoláme funkciu `najdi(v)`, dostaneme tú istú hodnotu. Tá sa totiž nemá prečo meniť. Najdlhšia cesta z tohto vrcholu zostane najdlhšou bez ohľadu na to, z ktorého vrchola sme sem prišli. A to platí pre ľubovoľný vrchol v. Namiesto toho, aby sme spúšťali funkciu `najdi(v)` viackrát, zapamätáme si, akú hodnotu nám vrátila pri prvom zavolaní. Keď ju budeme chcieť neskôr zavolať znova, jednoducho zistíme, že sme tú hodnotu už vyrátali predtým a vrátime rovno toto vyrátané číslo, čím sa vyhneme celému zbytočnému výpočtu.

Toto celé sa dá implementovať veľmi jednoducho. Vytvoríme si pole `memoizacia[]`, ktoré si bude na i-tej pozícii pamätať, aká je najdlhšia cesta vedúca z vrchola i. Na začiatku naplníme toto pole hodnotami -1 (alebo niečím iným zbytočným). Následne sa funkcia `najdi(v)` vždy po zavolaní najskôr pozrie, čo je v poli `memoizacia[v]`. Ak je tam číslo -1, znamená to, že sme túto funkciu pred tým nerátali. Preto pokračujeme rovnako, ako v prvom riešení, akurát si na konci výsledok zapamätáme na pozícii `memoizacia[v]`. Ak je ale v `memoizacia[v]` nejaké nezáporné číslo, vieme, že sme túto funkciu už volali s parametrom v a toto je jej výsledok, ktorý môžeme rovno vrátiť.

Táto technika, ktorá sa volá memoizácia, je naviac veľmi všestranná, lebo sa dá použiť na každú rekurzívnu funkciu. Občas nie až s tak dobrým výsledkom, ale často krát nám to riešenie výrazne zrýchli.

Zložitosť

Zostáva nám odhadnúť zložitosť tohto riešenia. Uvedomme si, že funkciu `najdi(v)` budeme rátať najviac raz pre každé v, lebo pri jej ďalších volaniach už stačí vyberať výsledok z poľa `memoizacia[]`. Počas všetkých týchto rátaní dokopy prejdeme cez každú hranu práve raz, a teda výsledná časová zložitosť bude $O(n + m)$.

Pamäťová zložitosť je rovnaká, lebo si musíme zapamätať celý vstupný graf. No a ako som sľúbil, tu je vzorový program, v ktorom sa oproti pomalému riešeniu zmenili naozaj iba štyri riadky. Odporúčam vám overiť si to ;)

Listing programu (C++)

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

int n, m;
vector<vector<pair<int, long long> > > Graf;
vector<int> Vysky;
vector<int> pocet_vchadzajucich_hran;
vector<long long> memoizacia;

long long najdi(int v) {
    if(memoizacia[v] != -1) return memoizacia[v];
    long long najdlhsia=0;
    // pre vsetkych susedov
    for(int i=0; i<Graf[v].size(); i++) {
        int w = Graf[v][i].first;
        int c = Graf[v][i].second;
        najdlhsia = max(najdlhsia, c+najdi(w));
    }
    memoizacia[v] = najdlhsia;
    return najdlhsia;
}

int main() {
    scanf("%d %d", &n, &m);
    Vysky.resize(n);
    Graf.resize(n);
    pocet_vchadzajucich_hran.resize(n);
    memoizacia.resize(n, -1);
    for(int i=0; i<n; i++)
        scanf(" %d", &Vysky[i]);
```

```
    for(int i=0; i<m; i++) {
        int x, y, c;
        scanf("%d %d %d", &x, &y, &c);
        x--; y--;
        if(Vysky[x] < Vysky[y]) swap(x, y);
        Graf[x].push_back(make_pair(y, c));
        pocet_vchadzajucich_hran[y]++;
    }
    long long vysledok=0;
    // pre vsetky vrcholy, do ktorych nic nevchadza
    for(int i=0; i<n; i++)
        if(pocet_vchadzajucich_hran[i] == 0)
            vysledok=max(vysledok, najdi(i));
    printf("%lld\n", vysledok);
}
```

Kubo

6. Objednaná elektronika

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

Hrubá sila

Máme zistiť, koľko rôznych kombinácií štyroch čisel zo vstupu je dobrých. Teda, koľko je štvoríc (A, B, C, D) takých, že $x_A \cdot x_B = x_C \cdot x_D$ a $1 \leq A < B < C < D \leq n$. Spravíme to teda prvým spôsobom, ktorý nás napadne – vyskúšame všetky štvorice. Tento bruteforce spravíme vnorením štyroch for-cyklov a jeho časová zložitosť je $O(n^4)$.

Nepočítajme nič dvakrát

Problém s naším riešením je, že veľa vecí v ňom počítame stále dookola. Pre každú kombináciu indexov A, B totiž raz prejdeme celý zvyšok poľa, kde kontrolujeme súčin čísel na pozíciách C, D. Ak by sme vedeli využiť informáciu o súčinoch C, D viackrát, bez prechádzania poľa, mohli by sme náš algoritmus výrazne zrýchliť.

Rovnako dobré by bolo, keby sme vedeli rýchlo nájsť počet dvojíc indexov A, B pre konkrétne C, D.

Vezmime si nasledujúce riešenie hrubou silou: postupne vyberáme dvojice C, D, a pre ne počítame výskyty dvojíc A, B s rovnakým súčinom.

Zo zadania vieme, že A, B sú vždy naľavo od C, D. Náš algoritmus teda prejde všetky dvojice čísel naľavo od C a porovná ich súčin s $x_C \cdot x_D$. Keď prejde všetky, tak si posunie D doprava a začne počítať od začiatku. Keď program doráta všetky D, tak si posunie C, nastaví nové D na $C+1$ a znova prezerá dvojice A, B a posúva D, kým skontroluje všetky.

Pozrime sa na výpočet pre konkrétne C. Všimnime si, že pre každé D hľadáme jemu príslušné A, B medzi tými istými číslami. Nám však stačí, ak si na začiatku raz **predrátame počet výskytov súčinov** $x_A \cdot x_B$. Potom už pre jednotlivé D vieme rýchlo zistiť, koľko dvojíc A, B má súčin $x_C \cdot x_D$.

Takto si pre každé C najprv spočítame počty dvojíc A, B s rôznymi súčinmi, uložíme si počty do nejakej dátovej štruktúry a potom stačí posúvať D a rýchlo vyberať z dátovej štruktúry počty dvojíc s daným súčinom $x_C \cdot x_D$. Ak by sme odhadli čas vkladania/výberu z dátovej štruktúry ako $O(t)$, mohli by sme odhadnúť čas potrebný pre tento algoritmus ako $O\left(n \cdot (tn^2 + tn)\right) = O(tn^3)$.

Keď už sme zrýchlili výpočet pri posúvaní D, pozrime sa, čo vieme zlepšiť na posúvaní C. Tu si môžeme všimnúť, že po posunutí C sú počty súčinov vľavo od neho skoro také isté, ako pri starom C. Všetko, čo sa stalo je, že sme pridali súčiny so starým C – možnosti, kde sa vybrané B rovná starému C. Teda keď si posunieme C, tak nemusíme zahodiť všetky predpočítané súčiny, ale stačí k nim pridať tieto nové.

Pre každé C si teda do dátovej štruktúry najprv pridáme súčiny dvojíc prvkov na pozíciách A a starého C a potom opäť posúvame D a vyberáme záznamy o počtoch dvojíc s požadovaným súčinom, čo trvá $O(tn + tn)$. Čas tohto algoritmu vieme odhadnúť ako $O\left(n \cdot (tn + tn)\right) = O(tn^2)$.

Ako si pamätať počty dvojíc s rovnakými súčinmi?

Už len zostáva vymyslieť, ako si počet súčinov zapamätáme – akú dátovú štruktúru použijeme. Najjednoduchšie by bolo ukladať si počty do poľa, kde index bude hodnota súčinu. Avšak súčiny môžu nadobúdať hodnoty až 10^{18}, a pole s takouto veľkosťou sa nám do pamäte nezmestí. Čísel na vstupe je ale najviac 1 000, čiže rôznych súčinov bude najviac 10^6. Ak by sme použili pole, väčšina indexov by teda mala hodnotou 0, a sem-tam by sa nám tam objavila iná hodnota.

Našťastie, problém, ako si zapamätať a rýchlo pracovať s rozumne malým počtom (10^6) veľkých záznamov, už niekto vyriešil. Použijeme dátovú štruktúru map, ktorá nám dovolí ukladať si dvojice <`súčin, počet dvojíc indexov A, B s daným súčinom`> tak, ako budeme potrebovať, ale nebude si zbytočne ukladať hodnoty, ktoré nepotrebujeme.

Zložitosť

Podľa toho, či si zvolíme implementáciu mapy ako binárneho vyhľadávacieho stromu (`map`) alebo ako hashovacej tabuľky (`unordered_map`) dostaneme časovú zložitosť vkladania/upravovania/čítania $t = O(\log n)$ alebo $t = O(1)$, teda celkovo $O(n^2 \log n)$ alebo $O(n^2)$, keď použijeme predošlé odhady.

Pre pamäť si zoberieme najhorší prípad. Ak sú súčiny čísel na vstupe rôzne, tak si musíme do mapy uložiť $n(n-1)$ čísel. Pamäťová zložitosť bude teda $O(n^2)$

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <map>

using namespace std;

// vyrobime si mapu na suciny
map<long long, long long> suciny;

// pridavanie sucinu do mapy
void pridaj_sucin(long long key){
    if(suciny.find(key)==suciny.end()){
        suciny[key]=1;
    }else{
        suciny[key]++;
    }
}

// hladanie poctu vyskytov sucinu v mape
long long najdi_pocet(long long key){
    if(suciny.find(key)!=suciny.end()){
        return suciny[key];
    }else{
        // ak sucin nie je v mape, tak sme ho este nikde nevideli
        return 0;
    }
}

int main(){
    // nacitame vstup
    int n;
    cin >> n;
    vector<long long> vstup(n);
    for (int i=0; i<n; i++){
        cin >> vstup[i];
    }

    long long result = 0;

    // postupne prejdeme vsetky C
    for (int c=0; c<n; c++){

        // postupne prejdeme D a najdeme pocet sucinov C*D v mape
        for (int d=c+1; d<n; d++){
            result += najdi_pocet(vstup[c]*vstup[d]);
        }

        // pridame nove suciny do mapy
        for (int d = 0; d<c; d++){
            pridaj_sucin(vstup[c]*vstup[d]);
        }
    }

    //vypiseme vysledok
    cout << result << endl;
    return 0;
}
```

Buj

7. Ozajstné vzrušenie

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

Zamyslime sa nad tým, ako by mohol Zdeno sám overiť, ktorým smerom mohol cestovať. Zaoberajme sa len cestou z Bratislavy do Košíc, cestu opačným smerom overíme analogicky. Zdeno môže postupovať nasledovne: začne v Bratislave, a pôjde smerom do Košíc. Pritom si postupne vyškrtáva úseky, ktoré už videl – vždy, keď vlak prejde časť cesty reprezentovanú jedným znakom, tak Zdeno overí, či práve *nedopozeral* aktuálne overovaný úsek. Ak áno, tak si ho zaškrtne (nakoľko nič nestratí tým, že ho zaškrtne najskôr, ako vie), a začne overovať nasledujúci úsek.

Zamyslime sa teraz nad tým, ako môže Zdeno overovať, či práve nedopozeral aktuálny úsek. Jeho reťazec si označíme B a jeho znaky si postupne označíme $B_1, B_2, \ldots, B_{|B|}$. Reťazec reprezentujúci cestu z Bratislavy do Košíc si označíme A, a jeho znaky postupne $A_1, A_2, \ldots, A_{|A|}$. Aktuálnu pozíciu vlaku označíme i.

Riešenie hrubou silou

Uvedomíme si, čo chceme zistiť. To, či Zdeno dopozeral B, znamená presne to, že reťazec posledných videných

$|B|$ znakov je rovný B. Formálne, je to ekvivalentné tomu, že $B = A_{i-|B|+1}A_{i-|B|+2}\dots A_{i-1}A_i$. A to vieme overiť jedným prechodom – najprv overíme, či $A_i = B_{|B|}$. Ak nie, tak Zdeno B nemohol dopozerať, a skončíme. Ak áno, tak overíme $A_{i-1} = B_{|B|-1}$, ak aj to platí, tak overíme $A_{i-2} = B_{|B|-2}$, a tak ďalej, až po posledný znak. Ak sme nenašli žiaden rozdiel, Zdeno práve dopozeral B.

Zamyslime sa nad časovou zložitosťou tohto algoritmu. Na jednej pozícii i môžeme potrebovať overiť až $O(|B|)$ znakov. Dokopy môže Zdeno spraviť až $O|A| \cdot |B|)$ operácií – presnejšie $O(d \cdot |B|)$, kde d je vzdialenosť prejdená vlakom, kým Zdeno nenájde pozíciu, kde končí hľadaný úsek. A naozaj existuje vstup, na ktorom ich toľko spraví – napríklad $A = aaaaaa\dots a$ a $B = aaaa\dots aaab$.

Všimnime si, že náš algoritmus v skutočnosti robí nasledovné: zistí, aký najdlhší **sufix** B končí na pozícii i[7]. Keď ale Zdeno uvidí nový znak, tak nevieme jednoducho z predchádzajúceho najdlhšieho sufixu povedať, aký bude najdlhší sufix teraz.

Skoro algoritmus KMP

Čo keby sme sa ale nepozerali na najdlhší sufix, ale **najdlhší prefix B končiaci na pozícii i**? Z takej informácie tiež vieme ľahko overiť, či Zdeno práve dopozeral B – vtedy je najdlhší prefix dlhý $|B|$.

Predpokladajme, že poznáme najdlhší prefix končiaci na pozícii i. Označme jeho začiatok l_1. (Potom je tento najdlhší prefix dlhý $i+1-l_1$.) Zamyslime sa nad tým, ako by sme vedeli zistiť najdlhší prefix končiaci na $i+1$. Na to nám stačí zistiť jeho začiatok l_2 – takže hľadáme najmenšie k také, že $A_kA_{k+1}\dots A_iA_{i+1}$ je prefixom B. (Potom $l_2 = k$.) Keďže $A_{l_2}A_{l_2+1}\dots A_iA_{i+1}$ je prefixom B, tak aj $A_{l_2}A_{l_2+1}\dots A_i$ je prefixom B. Preto je najviac taký dlhý, ako prefix začínajúci na l_1 a končiaci na i, ktorý je najdlhší možný končiaci v i. Takže nový prefix, končiaci v $i+1$ nemôže začínať skôr ako na pozícii l_1, kde začína $A_{l_1}A_{l_1+1}\dots A_i$. Teda musí platiť $l_2 \geq l_1$.

Potom nám stačí overovať, či pridaním znaku A_{i+1} na koniec *predĺžime* starý prefix. Ak áno, tak sme našli najdlhší prefix B končiaci na $i+1$ (teda platí $l_2 = l_1$). Ak nie, tak vyskúšame $k = l_1 + 1$, ak ani to nevyhovuje, tak vyskúšame $k = l_1 + 2$, a tak ďalej. Pri overovaní nového k musíme porovnať $i-k+1$ znakov prefixu B, či sa zhodujú so znakmi $A_k\dots A_i$.

Zamyslime sa nad časovou zložitosťou. Ak sa nám podarilo starý prefix *predĺžiť*, tak sme spravili len 1 porovnanie. V opačnom prípade sme pri každom posune k spravili najviac $O(|B|)$ porovnaní. k posúvame vždy len *doprava* (zvyšuje sa), a týchto posunov môže byť najviac $|A|$, alebo d – vzdialenosť prejdená vlakom, kým Zdeno nenájde pozíciu. Takže celková časová zložitosť môže byť najviac $O(|A| \cdot |B|)$, resp. $O(d \cdot |B|)$. Vyzerá to teda tak, že sme si veľmi nepomohli...

Algoritmus KMP

V predošlej časti sme zistili, že ak si pamätáme najdlhší prefix B, ktorý končí na pozícii i, občas vieme rýchlo nájsť prefix B končiaci na $i+1$ – ak je znak A_{i+1} ďalším znakom v B. Ak ale $A_{l_1}A_{l_1+1}\dots A_{i+1}$ nie je prefixom B, potrebujeme zistiť, kde má začínať najdlhší prefix B končiaci na pozícii $i+1$. Teraz si ukážeme, ako program vylepšiť tak, aby sme pre l_2 nemuseli skúšať všetky hodnoty $l_1+1, l_1+2, \dots, i+1$ ale len tie, ktoré by *veľmi pravdepodobne* mohli byť začiatkami nejakého prefixu B – začiatky druhého najdlhšieho, tretieho najdlhšieho, ... prefixu B, ktorý končí v i.

Zopakujme si, že l_2 má nasledujúce vlastnosti: $l_2 \geq l_1$. $P_2 = A_{l_2}A_{l_2+1}\dots A_i$ je prefixom B a $P_1 = A_{l_1}A_{l_1+1}\dots A_i$ je tiež prefixom B. Potom P_2 je sufixom P_1. Takže by nám stačilo skúšať také k**, pre ktoré $A_kA_{k+1}\dots A_i$ je sufixom P_1 (a zároveň prefixom B)**. Označme si všetky takéto k od najmenšieho postupne $l_1 = k_1 < k_2 < \dots < k_K$. Všimneme si, že $A_{k_2}A_{k_2+1}\dots A_i$ je najdlhší sufix $A_{k_1}A_{k_1+1}\dots A_i$, ktorý je aj prefixom B. Podobne pre k_3 a k_2, k_4 a k_3, a tak ďalej až po k_K a k_{K-1}.

Ak by sme vedeli **pre každý prefix B, označme ho P, zistiť jeho najdlhší vlastný (rôzny od P) sufix taký, že je tiež prefixom** B, tak by sme našu postupnosť vedeli ľahko zostrojiť:

Nech P je prefix B a $suf(P)$ je najdlhší vlastný sufix P. Naša postupnosť možných začiatkov prefixov B – $k_1 < k_2 < \dots < k_K$ – je potom:

$$l_1 = i - |P| + 1,\ i - |suf(P)| + 1,\ i - |suf(suf(P))| + 1,\ \dots,\ i+1$$

Ak sa nám podarí tieto sufixy prefixov B spočítať, program si bude pre každú overovanú pozíciu i v A pamätať P_i – najdlhší prefix B končiaci v i. Keď chceme zistiť najdlhší prefix končiaci na pozícii $i+1$:

[7] Každý podreťazec slova W vieme reprezentovať dvojicou (l, r), kde l je pozícia začiatku podreťazca, a r je pozícia konca podreťazca. Sufix je každý taký podreťazec, pre ktorý $r = |W|$. Prefix je každý taký podreťazec, pre ktorý $l = 1$. Sufixy slova *abca* sú teda $a, ca, bca, abca$. Prefixy sú $a, ab, abc, abca$.

- Ak bude ďalšie písmenko v A *dobré* (teda rovnaké ako ďalšie písmenko v B, formálne $A_{i+1} = B_{|P_i|+1}$), P_{i+1} bude o 1 dlhší prefix B ako P_i.
- Ak bude ďalšie písmenko v A *zlé*, skúsime sa pozrieť, či toto písmenko A_{i+1} nepasuje za prefix $suf(P_i)$, teda či $A_{i+1} = B_{|suf(P_i)|+1}$. Ak áno, našli sme prefix B končiaci v $i+1$ a P_{i+1} je $suf(P_i)$ s pridaným znakom A_{i+1} na koniec.
- Ďalej, ak sme nenašli prefix B končiaci na $i+1$, skúsime kratší sufix P a overíme $A_{i+1} = B_{|suf(suf(P_i))|+1}$
- ...
- Na konci overíme $A_{i+1} = B_1$ a dostaneme buď $P_{i+1} = \emptyset$ alebo $P_{i+1} = B_1$.

Program skončí (nájde B v A), ak $P_i = B$, teda ak najdlhší prefix B končiaci na pozícii i je samotné B.

Aká je časová zložitosť takéhoto programu? Vždy, keď Zdeno uvidí nový znak A_{i+1}, overíme, či vieme aktuálnemu prefixu $P_i = B_1B_2 \dots B_{|P_i|}$ (B označuje úsek, ktorého *dopozeranie* overujeme) pridať na koniec tento znak, teda či $B_1B_2 \dots B_{|P_i|}A_{i+1}$ je prefix B. Stačí nám teda len overiť, či $A_{i+1} = B_{|P_i|+1}$. Ak nie, tak to skúsime pre prefix dĺžky $|suf(P_i)|$. Overenie každého kratšieho prefixu trvá konštantne dlho – porovnáme 2 znaky. Pozícia začiatku prefixu B, ktorý končí na pozícii $i+1$, sa navyše vždy len zvyšuje. Z toho vyplýva, že celkovo spravíme týmto spôsobom najviac $O(|A|)$ operácií. Ak teda máme čiernu krabičku, ktorá nám konštantne rýchlo odpovedá hodnoty $suf(B_1B_2 \dots B_n)$ pre každé $n \in \{0, 1, 2, \dots, |B|\}$, tak časová zložitosť pre nájdenie B v A bude $O(|A|)$.

Posledná vec, ktorú potrebujeme vyriešiť je, **ako spočítame hodnoty** $suf(B_1B_2 \dots B_n)$ – najdlhší sufix $B_1B_2 \dots B_n$, ktorý je prefixom B. Môžeme si uvedomiť, že každý prefix B je jednoznačne určený jeho dĺžkou. Stačí nám teda pre prefix dĺžky n spočítať dĺžku jeho najdlhšieho vlastného sufixu, čo označíme $suf(n)$.

Všimnime si, čo sme robili v horeuvedenom algoritme. Keď sme chceli zistiť najdlhší sufix slova $B_1B_2 \dots B_nc$, tak nám stačilo skúšať za prefixy slova $B_1B_2 \dots B_n$ pridať c. To funguje aj v prípade, že $c = B_{n+1}$. suf potom vieme spočítať dynamickým programovaním.

Na začiatku nastavíme $suf(0)$, následne, ak poznáme $suf(0), suf(1), \dots, suf(n)$, tak na spočítanie $suf(n+1)$ robíme nasledovné: vyskúšame za prefix určený $suf(n)$ pridať c, ak to nevyjde, tak za $suf(suf(n))$, ak ani to nevyjde, tak za $suf(suf(suf(n)))$, a tak ďalej ... Až kým sa nám podarí pridať c, alebo nepodarí – vtedy nastavíme $suf(n+1) = 0$. Časová zložitosť je rovnaká, ako vyššie uvedenej časti algoritmu (ale už nepotrebujeme čiernu krabičku, keďže že si tie hodnoty suf po spočítaní pamätáme), lebo sme v podstate ten istý algoritmus spustili na vstupe B. Teda časová zložitosť predpočítania dĺžok prefixov bude $O(|B|)$.

Dokopy je časová zložitosť prinajhoršom $O(|A| + |B|)$ – presnejšie $O(d + |B|)$ (d je vzdialenosť prejdená vlakom).

Návrat k pôvodnému problému

Pôvodný problém teda vieme riešiť nasledovne: pomocou KMP vieme vytvoriť funkciu, ktorá na vstupe dostane reťazec A reprezentujúci cestu, reťazec B reprezentujúci úsek, ktorého dopozeranie Zdeno overuje, a pozíciu vlaku na nej (prvý znak, ktorý Zdeno zatiaľ nevidel). Tá nám vráti pozíciu, na ktorej Zdeno prvýkrát dopozerá B (resp. nám povie, že taká pozícia neexistuje nejakou nezmyselnou hodnotu, napríklad -1). Označíme si A reťazec reprezentujúci cestu z Bratislavy do Košíc, a A' cestu opačným smerom (zrkadlovo obrátený reťazec). p, p' označíme aktuálne pozície vlakov. Postupne načítavame úseky B a zisťujeme, kde Zdeno dopozerá B na úseku $A_pA_{p+1} \dots A_{|A|}$, čo zistíme volaním funkcie $dopozeraj(A, B, p)$ a túto pozíciu zapíšeme do p, podobne $dopozeraj(A', B, p')$ zapíšeme do p'.

Nakoniec overíme, či mohol Zdeno cestovať prvým smerom (či $p \neq -1$), a druhým smerom (či $p' \neq -1$), a podľa toho vypíšeme odpoveď.

Časová zložitosť je $O((d_1 + |B_1|) + (d_2 + |B_2|) + \dots + (d_b + |B_b|))$, kde d_i je vzdialenosť prejdená vlakom, kým Zdeno nedopozerá úsek B_i. To je rovné $O((d_1 + \dots + d_b) + (|B_1| + \dots + |B_b|)) = O(|A| + (|B_1| + \dots + |B_b|))$, teda je lineárna od veľkosti vstupu.

Pamäťová zložitosť je $O(A + max(|B_1|, |B_2|, \dots, |B_b|))$ – nemusíme si pamätať všetky $B_1, \dots, B_b$ naraz, stačí ich načítavať postupne (po každom načítaní zistíme nové p, p' – pozície vlakov, a následne reťazec môžeme zahodiť, lebo ho už ďalej nepotrebujeme).

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

int dopozeraj(const string& A, int poz, const string& B) {
    if (poz == -1)
        return -1;

    int suf[B.size()+1];
```

```
    suf[0] = -1;
    for (int i=1; i<=B.size(); i++) { // dlzka prefixu, pre ktory hladame jeho najdlhsi dobry sufix
        int kandidat = suf[i-1];      // dlzka kandidata -- potom posledny znak kandidata je B[kandidat - 1]
                                      // (pricom B[-1] signalizuje ziaden znak), a nasledujuci znak je B[(kandidat-1)+1]
        while (kandidat!=-1 && B[(kandidat-1)+1]!=B[i-1]) {
            kandidat = suf[kandidat];
        }
        suf[i] = kandidat+1;
    }

    // dlzka najdlhsieho prefixu konciaceho na danej pozicii
    int najpref=0;
    while (poz!=(int)A.size() && najpref!=B.size()) {
        while (najpref!=-1 && B[(najpref-1)+1]!=A[poz]) {
            najpref= suf[najpref];
        }
        najpref++;
        poz++;
    }
    if (najpref!=B.size()) {
        return -1;
    }
    return poz;
}

int main () {
    string A1; // z Bratislavy do Kosic
    cin >> A1;
    string A2 = A1; // z Kosic do Bratislavy
    reverse(A2.begin(),A2.end());

    int n;
    cin >> n;
    int p1=0, p2=0;
    for (; n>0; n--) {
        string B;
        cin >> B;
        p1 = dopozeraj(A1,p1,B);
        p2 = dopozeraj(A2,p2,B);
    }

    bool moze1 = (p1 != -1);
    bool moze2 = (p2 != -1);
    if (moze1 && moze2) {
        cout << "neviem\n";
    }
    if (moze1 && !moze2) {
        cout << "z Bratislavy do Kosic\n";
    }
    if (!moze1 && moze2) {
        cout << "z Kosic do Bratislavy\n";
    }
    if (!moze1 && !moze2) {
        cout << "zabludil\n";
    }
    return 0;
}
```

8. Ozajstná veda

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

Skoro riešenie

Najprv si úlohu zjednodušme. Čo by sa stalo, keby sme nemali šťastné čísla? Potom by sme vedeli spočítať, koľkými spôsobmi sa dá vybrať k čísel zo skupiny n čísel. Je to takzvané kombinačné číslom

$$\binom{n}{k} = \frac{n!}{k!(n-k!)}$$

Čo teraz s tými nešťastnými šťastnými číslami?

Zabudnime na chvíľu na čísla, ktoré nie sú šťastné a zoberme si len tie, ktoré obsahujú štvorky a sedmičky. Nech ich je dokopy S. Koľkými spôsobmi vieme z tejto postupnosti S čísel vybrať k prvkovú množinu tak, aby tam nebolo žiadne číslo dvakrát? Odpoveď vieme zistiť dynamickým programovaním.

Usporiadajme si šťastné čísla bez duplikátov do postupnosti $s_1, s_2, \ldots, s_m$. Nech je šťastné číslo s_i na vstupe p_i-krát. Do tabuľky s rozmermi $D[m+1][k+1]$ si na políčku $D[i][j]$ vyrátame, **koľkými spôsobmi vieme vybrať množinu obsahujúcu j prvkov ak používame len šťastné čísla** $s_1, \ldots, s_i$ a každé šťastné číslo sa môže vo vybranej množine objaviť najviac raz. Pri rátaní $D[i][j]$ máme vždy dve možnosti. Ak sme s_i nevložili do vybranej množiny, tak máme toľko možností, ako pre $i-1$ čísel a veľkosť množiny j. Ak sme ale s_i vložili do množiny, tak sme si mohli vybrať jedno z p_i čísel, ktoré majú hodnotu s_i a jednu množinu veľkosti $j-1$, ktorá sa opäť skladá z prvých $i-1$ čísel. Toto zapíšeme vzorcom:

$$D[i][j] = D[i-1][j] + p_i \cdot D[i-1][j-1]$$

V našom dynamickom programovaní zistíme hodnoty $m \cdot k$ stavov a každý vypočítame v konštantnom čase, takže časová zložitosť bude $O(mk)$.

Posledné, čo si musíme uvedomiť, je, že počet rôznych šťastných čísel na vstupe, teda hodnota m, je malý. Zhora ho môžeme odhadnúť číslom $2^{10} - 2$, čo je počet všetkých, najviac 9-ciferných štastných čísel. Takisto, aj keď podľa zadania je k veľké až 100 000, tak v našom dynamickom programovaní sa nám neoplatí skúšať k väčšie ako m. Z každého šťastného čísla totiž môžeme zobrať najviac jedno. Časová zložitosť bude teda $O(m^2)$ a keďže m je rádovo 1 000, takéto riešenie sa nám v pohode zmestí do časového aj pamäťového limitu.

Stačí už len spojiť tieto dve myšlienky a máme úlohu teoreticky vyriešenú. Pre každé $l \leq m$ vyrátame, koľkými spôsobmi vieme vybrať postupnosť dlhú $k - l$ len z nie šťastných čísel (čo je kombinačné číslo) a postupnosť dlhú l zo šťastných čísel (čo je číslo z našej dynamiky). Tieto dve čísla vynásobíme, výsledok sčítame cez všetky l a máme riešenie.

Kombinačné čísla v modulárnej aritmetike

Nakoľko sú čísla veľké a stačí nám výsledok modulo $10^9 + 7$, potrebujeme (chceme (musíme)) používať modulárnu aritmetiku. Konkrétne vieme, že $(a+b) \bmod P = (a \bmod P) + (b \bmod P)$ a $(a \cdot b) \bmod P = (a \bmod P) \cdot (b \bmod P)$. Problém ale je, že potrebujeme rátať veľké kombinačné čísla, v ktorých sa delí. Teda by sme chceli vedieť, koľko je

$$\frac{a}{b} \bmod P$$

Na pomoc príde Malá Fermatova veta[8], ktorá hovorí, že ak je P prvočíslo (čo číslo $10^9 + 7$ je) a b je ľubovoľné číslo, tak $b^{P-1} = 1 \bmod P$. Keď túto rovnicu vynásobíme číslom a a predelíme b, dostaneme:

$$\frac{a}{b} \bmod P = a \cdot b^{P-2} \bmod P$$

To znamená, že delenie sme si nahradili násobením, ktoré vieme modulovať priebežne, vďaka vyššie spomenutému vzťahu. Číslo $b^{P-2} \bmod P$ sa inak nazýva inverzný prvok k číslu b.

Predposledná vec, ktorú potrebujeme na riešenie, je rýchle umocňovanie, aby sme hodnotu b^{P-2} stihli zrátať. Hlavná myšlienka tohto algoritmu je, že ak chceme vypočítať hodnotu x^{2y}, tak nám stačí vypočítať číslo x^y, a to umocniť na druhú. Pokiaľ by sme chceli vypočítať x^{2y+1}, tak opäť vypočítame x^y, umocníme na druhú a výsledok ešte raz vynásobíme x. Samozrejme, všetky operácie robíme modulo P.

Túto ideu samozrejme aplikujeme rekurzívne, takže pri počítaní x^y budeme rátať niečo štýlu $x^{y/2}$, čím dostaneme algoritmus, ktorý umocňuje s časovou zložitosťou $O(\log(y))$, keďže zakaždým sa exponent, ktorý potrebujeme vyrátať zmenší na polovicu.

Na rátanie kombinačných čísel si teda dopredu predpočítame všetky faktoriály do poľa $F[\,]$ a k nim si vypočítame inverzné prvky, ktoré si uložíme do poľa $FI[\,]$. Následne vieme, že

$$\binom{n}{k} = F[n] \cdot FI[k] \cdot FI[n-k]$$

Zložitosť

Úspešným zvládnutím všetkých týchto vecí dostaneme správne a dostatočne rýchle riešenie. Zostáva už len odhadnúť časovú zložitosť. Nech m je počet rôznych šťastných čísel vo vstupnej postupnosti. Potom dynamické programovanie zaberie čas $O(m^2)$. Predrátanie si hodnôt p_i, ktoré počas neho potrebujeme nám pri použití štruktúry `map` zaberie čas $O(n \log m)$.

Naviac si potrebujeme predrátať všetky faktoriály, čo zaberie čas $O(n)$. No a pri postupnom skúšaní hodnoty l potrebujeme pre každú z m možností vyrátať jedno kombinačné číslo v čase $O(\log(P))$, čo nám pridá zložitosť $O(m \log P)$.

Keď to dáme všetko dokopy, dostaneme zložitosť $O(n \log m + m^2 + m \log P)$. Pamäťová zložitosť bude $O(n + m^2)$, ale ak by sme si v našej dynamike pamätali len posledné dva riadky, dostali by sme zložitosť $O(n + m)$.

Listing programu (C++)

```
#include<iostream>
#include<algorithm>
#include<map>
using namespace std;
#define For(Q,W) for(long long Q=0; Q<W; Q++)

typedef long long LLD;
```

[8] Kuk wikipédia

```
LLD PRIME = 1000000007;
LLD faktorial[1000000];

LLD power(LLD a, LLD b){ // umocnovanie
    if(b == 0ll) return 1ll;
    if(b == 1ll) return a;
    LLD pom = power(a, b/2ll);
    if(b%2ll == 0ll) return (pom * pom) % PRIME;
    else return (((pom * pom) % PRIME) * a) % PRIME;
}

LLD inv(LLD x){ // ratanie inverzneho prvku
    return power(x, PRIME-2ll);
}

LLD choice(LLD a, LLD b){ // ratanie kombinacneho cisla
    return (faktorial[a] * inv((faktorial[b] * faktorial[a-b]) % PRIME)) % PRIME;
}

bool is_lucky(LLD x){ // zistime, ci je cislo stastne
    while(x>0){
        if (x%10 != 4 && x%10 != 7) return false;
        x = x/10;
    }
    return true;
}

LLD min(LLD a, LLD b){ // lebo klasicke C++ nema min na long longoch
    if(a<b) return a;
    else return b;
}

int main(){
    faktorial[0]=1ll;
    For(i, 1000000-1) // predratame si faktorial
        faktorial[i+1] = (faktorial[i]*(i+1)) % PRIME;
    map<LLD, LLD> mapa;
    LLD n, k;
    cin >> n >> k;
    LLD pole[n];
    For(i, n) cin >> pole[i];

    LLD unlucky =0;
    For(i, n){ // stacia nam pocty jednotlivych stastnych cisel
        if (is_lucky(pole[i])){
            if(mapa.count(pole[i])==0){
                mapa[pole[i]]=1;
            }
            else{
                mapa[pole[i]]+=1;
            }
        }
        else unlucky+=1;
    }

    LLD D[mapa.size()+1];
    For(i, mapa.size()+1) D[i]=0;
    D[0] = 1;
    for (map<LLD, LLD>::iterator it=mapa.begin(); it!=mapa.end(); ++it){ // dynamika
        for(LLD i=mapa.size(); i>0; i--){
            D[i] = (D[i] + D[i-1] * it->second) % PRIME;
        }
    }

    LLD ans = 0;
    For(i, min(k+1,  unlucky+1)){ // spocitame pre jednotlive pocty nie stastnych
        if(k-i <= mapa.size())
            ans += choice(unlucky, i) * D[k-i];
        ans = ans%PRIME;
    }
    cout<<ans<<endl;
    return 0;
}
```