Korešpondenčný seminár z programovania
XXXVI. ročník, 2018/19
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

1. Trefa do čierneho

aja (aja@ksp.sk)
(max. 12 b za popis, 8 b za program)

Hrubá sila

Pri tomto riešení budeme často potrebovať zistiť, či sme pod alebo nad priamkou $y = x$. To je jednoduché: pod priamkou sa nachádzame, ak je súradnica x väčšia ako y, v opačnom prípade sme nad priamkou.

Najjednoduchšie riešenie, ktoré by nám pri tejto úlohe mohlo napanúť, je postupné simulovanie Samových posunov a rátanie pretnutí s priamkou. Problém ale je, že takéto posuny by mohli trvať potenciálne donekonečna. Preto sa treba najprv zamyslieť, kedy môžeme so simuláciou prestať. Podľa toho, v akom vzťahu sú veľkosti posunov d_x a d_y, môžu nastať nasledujúce prípady:

- $d_x = d_y$: po každých 2 posunoch sa ocitneme opäť na priamke $x = y$, teda dotykov s priamkou bude nekonečne veľa.
- $d_x > d_y$: s priamkou sa stretneme iba raz, a to v bode $(0, 0)$. Od nášho prvého pohybu sa už budeme stále nachádzať iba pod priamkou (rozmyslite si).
- $d_x < d_y$: tu sa budeme s priamkou stretávať rôzne veľa krát. Stačí nám ale uvedomiť si, že ak sa pri nejakom pohybe doprava nepretneme s priamkou, už nikdy sa s ňou nepretneme. Od tohto momentu budeme totiž celý čas už iba nad priamkou.

Keď už poznáme tieto tri situácie, môžeme sa pustiť do programovania nášho riešenia. Na začiatku skontrolujeme, či sa veľkosti posunov nerovnajú, alebo či nie je posun doprava väčší ako ten smerom hore. Ak áno, vypíšeme potrebný výsledok. Ak nie, simulujeme pohyb po mriežke a po každom kroku zistíme, či sme sa pretli s priamkou. S priamkou sa pretneme vtedy, ak sme išli spod priamky nad ňu, alebo naopak (samozrejme, nezabúdame na ošetrenie dotyku s priamkou). Ak sme sa pri niektorom pohybe doprava nepretli s priamkou, simuláciu ukončíme a vypíšeme počet pretnutí.

Pamäťová zložitosť tohto riešenia je $O(1)$, keďže miesto, kde sme, a veľkosti posunov si udržiavame v pár premenných.

Časová zložitosť riešenia je priamo úmerná počtu krokov, ktoré odsimulujeme. Ten je zhruba rovný počtu pretnutí s priamkou (ak dva kroky po sebe priamku nepretneme, simuláciu ukončíme). Ako si ukážeme v časti o ideálnom riešení, počet pretnutí je (v zaujímavom prípade $d_x < d_y$) zhruba $d_x/(d_y - d_x)$, časová zložitosť je preto $O(d_x/(d_y - d_x)) = O(d_x)$.

Listing programu (C++)

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

int main() {
  int t;
  cin >> t;
  for (int i = 0; i < t; i++) {
    long long dx, dy;
    cin >> dx >> dy;
    if (dx == dy) {
      cout << -1 << endl;
      continue;
    }
    if (dx > dy) {
      cout << 1 << endl;
      continue;
    }
    long long somx = 0, somy = 0, pocet = 0;
    while (true) {
      if (somx == somy) pocet ++;
      if (somx < somy && somx + dx > somy) pocet ++;
      somx += dx;
      if (somx < somy) break;
      if (somx == somy) pocet ++;
      if (somy < somx && somy + dy > somx) pocet ++;
      somy += dy;
```

```
    }
    cout << pocet << endl;
  }
}
```

Ideálne riešenie

Ako sme si pri riešení hrubou silou mohli všimnúť, v dvoch prípadoch sme mali odpoveď hneď, bez akejkoľvek simulácie. Ak sa bližšie zamyslíme a nakreslíme si pár obrázkov, zistíme, že aj v poslednom prípade vieme počet dotykov vypočítať v konštantnom čase.

Simuláciu sme ukončovali, keď sme prvý raz urobili krok doprava, pri ktorom sme našu priamku nepretli, ale zostali sme nad ňou. Bol to vlastne prvý krok doprava, po ktorom bola naša y-ová súradnica väčšia, než x-ová. Pozrime sa preto na body, v ktorých budeme po pohybe smerom vpravo, čiže po prvom kroku, po treťom kroku, atď. a všímajme si, o koľko je ich x-ová súradnica väčšia, než y-ová. Po prvom posune má x-ová súradnica “náskok” d_x, keďže sme v bode $(d_x, 0)$. Po každých dvoch krokoch však y-ová súradnica tento “náskok” stiahne o $d_y - d_x$. Na to, aby y-ová súradnica “dobehla” tú x-ovú, teda treba $\lceil d_x/(d_y - d_x) \rceil$ takýchto dvojkrokov. Pri každom takomto dvojkroku, možno s výnimkou posledného, pretneme priamku dvakrát, počet pretnutí teda bude zhruba $2d_x/(d_y - d_x)$.

Do vzorca nesmieme zabudnúť pripočítať pretnutie v bode $(0, 0)$ kde začíname. Ďalej je potrebné si uvedomiť, že nám vzniknú 2 prípady, pri ktorých sa počty mierne líšia:

- V prípade, že je d_x deliteľné číslom $d_y - d_x$, po $d_x/(d_y - d_x)$ dvojkrokoch budeme presne na priamke. To znamená, že v každom dvojkroku priamku pretneme dvakrát, spolu s pretnutím v bode $(0, 0)$ teda dostávame vzorec: $2 \cdot d_x/(d_y - d_x) + 1$
- Ak d_x nie je deliteľné číslom $d_y - d_x$, po $\lceil d_x/(d_y - d_x) \rceil$ dvojkrokoch skončíme nad priamkou. To znamená, že v poslednom z týchto dvojkrokov sme priamku pretli iba raz. Spolu s pretnutím v $(0, 0)$ teda dostaneme vzorec: $2 \cdot \lceil d_x/(d_y - d_x) \rceil$.

Časová aj pamäťová zložitosť tohto riešenia je konštantná, keďže potrebujeme iba pár premenných na výpočet vzorca, teda $O(1)$.

Technická poznámka

Pri výpočte hodnoty $\lceil d_x/(d_y - d_x) \rceil$ potrebujeme vydeliť dve čísla a výsledok zaokrúhliť nahor. Programovacie jazyky ako Python a C++ však pri celočíselnom delení zaokrúhľujú nadol. Preto v kóde vzorového riešenia využívame fakt, že d_x nie je deliteľné číslom $d_y - d_x$, teda $\lceil d_x/(d_y - d_x) \rceil = \lfloor d_x/(d_y - d_x) \rfloor + 1$.

Listing programu (C++)

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

int main() {
  long long t;
  cin >> t;
  long long dx, dy;
  for (int i = 0; i < t; i++) {
    cin >> dx >> dy;
    if (dx == dy) {
      cout << -1 << endl;
      continue;
    }
    if (dx > dy) {
      cout << 1 << endl;
      continue;
    }
    if (dx % (dy - dx) != 0)
      cout << 2 * (dx / (dy - dx)) + 2 << endl;
    else
      cout << 2 * (dx / (dy - dx)) + 1 << endl;
  }
}
```

Listing programu (Python)

```python
t = int(input())
for testcase in range(t):
    dx, dy = map(int, input().split())
    if dx == dy:
        print(-1)
    elif dx > dy:
        print(1)
    else:
```

```
if dx % (dy - dx) == 0:
    print(dx // (dy - dx) * 2 + 1)
else:
    print(dx // (dy - dx) * 2 + 2)
```

Dávid (davidb@ksp.sk)

2. Ukladanie kartičiek

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

Našou úlohou bolo zistiť pre daný vstup, či existuje výherná stratégia pre prvého hráča. Teda či počnúc prvým ťahom, bez ohľadu na to ako zareaguje John, môže Vlejd spraviť taký ťah že Johna dostane do situácie, z ktorej nemôže vyhrať (ak bude Vlejd hrať dobre).

Kedy Vlejd vyhrá

Ak je v hre jediná kartička s najväčším číslom, je to ideálna situácia, Vlejd ju zoberie a vyhrá. Ak je ale najväčších kartičiek viac, rozlišujeme dve situácie. Ak je kartičiek nepárny počet, Vlejd opäť zoberie prvú z nich a ďalej budú striedavo brať po jednej kartičke, až zoberie poslednú (John zase nemá na výber ani v jednom ťahu).

Problém nastane ak ich je párny počet. Vtedy Vlejd potrebuje aby prvú z nich zobral John. Znamená to teda že Vlejd je nútený zobrať nejakú z menších kartičiek, aby hneď neprehral. Musí si však dať pozor, aby sa nedostal do situácie, kedy mu John zoberie poslednú menšiu kartičku a on už bude nútený zobrať prvú z najväčších kariet. Jeho cieľom je teda opäť zobrať poslednú kartu, ale teraz zo všetkých okrem najväčších. Dostali sme sa ku problému, ktorý sme už vyššie vyriešili.

Vieme teda, že ak je najväčších kariet nepárny počet, Vlejd vyhrá. Ak je najväčších párny, ale druhých najväčších nepárny, Vlejd taktiež vyhrá. Ak by sme takto pokračovali, zistíme, že ak v postupnosti kartičiek usporiadanej od najväčšej narazíme na nejakú, ktorej je nepárny počet, Vlejd určite vyhrá.

Pozrime sa, ako by teda prebiehala nejaká hra: Karty: 1 2 2 3 3 3 4 4 5 5 5 5 Vlejd: 3 Karty: 3 3 4 4 5 5 5 5 John: ?

Vlejd sa pozrel na kartičky a najväčšia, ktorej je nepárny počet, bola trojka, tak ju zobral. Po takomto ťahu Johnovi ostala každá karta v párnom počte, a neostáva mu nič iné ako to zmeniť a dať Vlejdovi opäť víťaznú pozíciu. Ďalej Vlejdovi stačí ťahať to isté, čo potiahne John.

Vyhrávajúca a prehrávajúca pozícia

Kartičky, ktoré ostávajú na stole budeme volať pozícia v hre. Ak je teda na stole len jedna najväčšia kartička, hráč, ktorý je na ťahu, môže vyhrať, a teda je vo vyhrávajúcej pozícii. Ak sú na stole len dve rovnaké kartičky a nič iné, hráč na ťahu je v prehrávajúcej pozícii, lebo nech by spravil čokoľvek, protihráč sa dostane do výhernej pozície.

Z popisu vyššie môžeme vidieť, že pozícia, v ktorej je každá kartička v párnom počte, je prehrávajúca. Všetky ostatné – teda pozície, v ktorých je *nejaká* kartička v nepárnom počte – sú vyhrávajúce.

Riešenie

Potrebujeme zistiť, či je úvodná pozícia vyhrávajúca – teda či sa na vstupe nachádza aspoň jedno číslo nepárny počet krát. Existuje niekoľko spôsobov ako niečo takéto naprogramovať.

Jeden spôsob je, že si čísla zoradíme od najmenšieho po najväčšie, a budeme počítať, koľko rovnakých je za sebou. Takéto riešenie dostane plný počet bodov, existuje však o niečo krajšie riešenie:

Vyrobíme si dosť veľké pole a na i-tom políčku si budeme pamätať, koľko krát bolo na vstupe číslo i. Takýto postup sa volá tiež *Counting sort*[1], ale môžeme si ho dovoliť len vtedy, keď sa nám *dosť veľké pole* zmestí do pamäte, teda keď rozsah čísel na kartičkách je dosť malý. Toto riešenie má časovú zložitosť $O(n)$, pretože iba raz prejdeme naše vstupné pole. Pamäťová zložitosť je taktiež $O(n)$, kde n je najväčšie možné číslo kartičky na vstupe.

Listing programu (Python)

```
n = int(input())
cards = [int(x) for x in input().split()]
counts = [0 for i in range(10**5 + 1)]

for c in cards:
  counts[c] += 1

for c in counts:
  if c % 2 == 1:
```

[1] https://sk.wikipedia.org/wiki/Counting_sort

```
    print("Vlejd")
    exit(0)

print("John")
```

Listing programu (C++)

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

using namespace std;

int main(){
    int n;
    cin>>n;
    vector<int> counts(100001,0);
    for(int i = 0; i < n; i++){
        int t;
        cin>>t;
        counts[t]++;
    }

    for(int i = 0; i <= 100000; i++){
        if(counts[i] % 2 == 1){
            cout << "Vlejd" << endl;
            return 0;
        }
    }
    cout << "John" << endl;
}
```

Je zaujímave uvedomiť si, že počet výskytov čísla si v skutočnosti nemusíme pamätať. Zaujíma nás iba, či sa dané číslo vyskytuje na vstupe párny alebo nepárny počet krát. Takúto hodnotu si môžeme uložiť ako $True/False$ a keď príde nový výskyt daného čísla, iba hodnotu na pozícií tohoto čísla znegujeme – párny počet sa zmení na nepárny a naopak. Je dôležité si uvedomiť, že toto nám zmenší množstvo pamäte, ktorá bude použitá, ale asymptotická pamäťová zložitosť bude stále $O(n)$, teda lineárna.

Listing programu (Python)

```
n = int(input())
cards = [int(a) for a in input().split()]
count = [False for i in range(100001)]

for c in cards:
  count[c] = not count[c]

for c in count:
  if c:
    print("Vlejd")
    exit(0)

print("John")
```

3. Hurá leto!

kubik (kubik@ksp.sk)
(max. 12 b za popis, 8 b za program)

Toto vzorové riešenie má dve časti. V prvej časti sa pozrieme, ako má Žaba ukladať škatule, aby vytvoril čo najmenej kôp. V druhej časti budeme riešiť, ako toto uloženie efektívne vypočítať.

Mohli by sme sa do toho pustiť intuitívne. Postavíme si najsilnejšiu škatuľu a začneme na ňu ukladať ostatné škatule. Tento prístup má ale svoje problémy. Napríklad, ak máme ďalšiu, rovnako pevnú škatuľu, nevieme, či ju máme položiť na tú prvú, alebo si ju máme šetriť do ďalšej kopy. Konkrétny príklad: ak máme iba dve škatule s pevnosťou 1, oplatí sa ich postaviť na seba. Na druhú stranu, ak máme navyše ešte dve škatule s pevnosťou 0, viac sa nám oplatí "jednotkové" škatule rozdeliť.

Austrália

Skúsme sa na to teda pozrieť opačne. Stavajme kopy "odvrchu". Novú kopu začneme jej najslabšou škatuľou. Ďalej budeme pridávať čoraz silnejšie škatule, vždy na *spodok* kopy.

Teraz už k samotnému ukladaniu škatúľ. Berme škatule od najslabšej po najpevnejšiu a ukladajme ich do kôp. Vždy, keď ukladáme nejakú škatuľu, musíme sa rozhodnúť, či ňou začneme novú kopu, alebo ju pridáme pod nejakú už existujúcu. Ak sme sa rozhodli pridať škatuľu pod niektorú z už existujúcich kôp, musíme si navyše vybrať pod ktorú. Prirodzene sa nám núka položiť škatuľu pod najväčšiu kopu, ktorú je ešte schopná uniesť, aby sme "neplytvali pevnosťou". Pokiaľ naša škatuľa nie je schopná uniesť žiadnu z existujúcich kôp, začneme novú kopu. Takýmto spôsobom vieme rozdeliť všetky škatule do niekoľkých kôp. Dostaneme ale zaručene najmenší možný počet kôp?

Zdôvodnenie správnosti

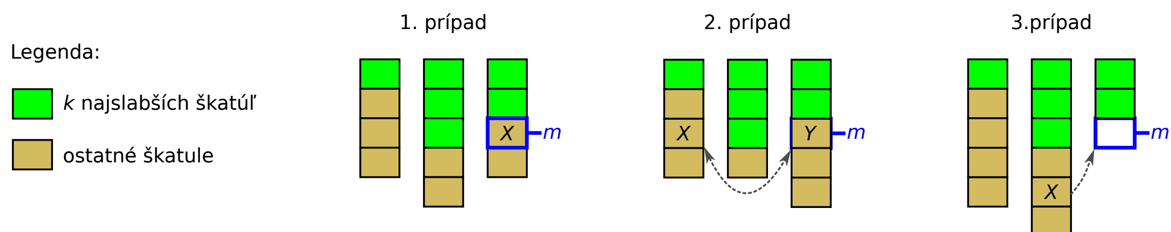
Treba si uvedomiť, že môže existovať viacero spôsobov, ako škatule rozdeliť na najmenší možný počet kôp. Tieto spôsoby budeme ďalej volať *optimálne riešenia*. My chceme ukázať, že náš postup rozdelí škatule jedným z nich. Nato si ukážeme, že po každej pridanej škatuli v našom postupe je ešte možné doplniť ostatné škatule tak, aby sme dostali jedno z optimálnych riešení. Ak toto bude platiť aj po pridaní poslednej škatule, znamená to, že sme vytvorili optimálne riešenie.

Predpokladajme, že sme už umiestnili prvých k škatúľ a ešte stále je možné doplniť ostatné škatule tak, aby vzniklo optimálne riešenie. Škatuľu, ktorú by náš algoritmus umiestnil ako ďalšiu (teda $k+1$-vú najslabšiu škatuľu) označme X a miesto, kam by ju dal, označme m. Doplňme ostatné škatule tak, aby vzniklo optimálne riešenie a pozrime sa, kde v ňom je škatuľa X. Ak náš algoritmus nechcel škatuľou X začať novú kopu, môžu nastať 3 prípady:

1. Škatuľa X sa nachádza na mieste m.

2. Škatuľa X je niekde inde a na mieste m sa nachádza nejaká iná škatuľa Y. Vieme, že všetky škatule slabšie než X sú umiestnené tam, kam ich dal náš algoritmus. To znamená, že škatuľa Y nemôže byť jedna z nich, a teda musí byť aspoň tak pevná, ako škatuľa X. Ak teraz vymeníme škatule X a Y, škatuľa Y určite bude schopná niesť záťaž, ktorú predtým niesla škatuľa X. Škatuľa X sa tým ocitne na mieste m, kam ju chcel dať aj náš algoritmus, a teda tiež určite unesie svoj náklad. Takto sme dostali iné riešenie Žabovej úlohy, ktoré je tiež optimálne (má rovnako veľa kôp) a navyše má škatuľu X na mieste m.

3. Škatuľa X je inde ako na mieste m a miesto m je prázdne. V takom prípade môžeme škatuľu X preniesť na miesto m. Škatuľa X svoj nový náklad určite unesie, rovnako aj všetky ostatné škatule (škatuliam, ktoré boli pod X sa náklad odľahčil o škatuľu X, ostatným sa nezmenil). Opäť sme dostali optimálne riešenie, kde je škatuľa X na mieste m.



Ak náš algoritmus chcel škatuľou X začať novú kopu, znamená to, že by neuniesla žiadnu z kôp, ktoré existovali po umiestnení prvých k škatúľ. Aj v optimálnom riešení preto X musí byť v nejakej inej kope. Za miesto m teraz budeme považovať vrch kopy obsahujúcej škatuľu X. Ak je X navrchu tejto kopy, situácia je rovnaká ako v prípade 1, ak nie je navrchu, situácia je rovnaká ako v prípade 2.

V každom prípade existuje optimálne riešenie, v ktorom je škatuľa X na mieste m, teda aj keby sme nechali náš algoritmus umiestniť prvých $k+1$ škatúľ, zvyšné škatule by sa určite dali doplniť do optimálneho riešenia.

Ak našu predošlú úvahu urobíme pre $k = 0$, dostávame, že prvú škatuľu náš algoritmus umiestni dobre, t.j. bude možné doplniť zvyšné škatule do optimálneho riešenia. Zopakovaním úvahy pre $k = 1$ dostávame, že aj druhú škatuľu náš algoritmus umiestni dobre. Úvahu postupne zopakujeme pre $k = 2, 3, \ldots, n-1$ a dostaneme, že aj keď necháme náš algoritmus umiestniť všetkých n škatúľ, bude možné doplniť zvyšných 0 škatúľ do optimálneho riešenia[2]. To ale znamená, že náš algoritmus vytvoril optimálne riešenie.

Implementácia

Ukázali sme si algoritmus, ktorým Žaba môže ukladať škatule. Teraz ešte napísať program, ktorý to odsimuluje.

Jedno z pozorovaní, ktoré nám pomôžu pri implementácii je, že ono si nám vlastne netreba pamätať, čo v kope je, stačí nám vedieť veľkosť kopy. Prečo? Keď škatule vkladáme na spodok kopy, potrebujeme iba skontrolovať, či počet vecí na tejto kope nie je väčší, než pevnosť škatule. Toto nám o dosť zjednoduší programovanie nášho riešenia.

Listing programu (Python)

[2] Technike, ktorú sme práve použili, sa hovorí *matematická indukcia*

```python
def pocet_kopok(skatule, n):
    kopky = [0] * n
    for sila in skatule:
        for i in range(n):
            if sila >= kopky[i]:
                kopky[i] += 1
                break
    # zoberieme len tie kôpky, ktoré sme použili
    return len(kopky) - kopky.count(0)
```

Ako vidíme, skutočne len pre každú škatuľu prechádzame všetky potenciálne kôpky a uložíme to na prvú, na ktorú má naša aktuálna škatuľa dosť pevnosti. Toto riešenie má ale časovú zložitosť $O(n^2)$. Zoberme si napríklad, že všetky škatule čo dostaneme, by boli z papundekla, teda pevnosti 0. Pre každú škatuľu musíme prejsť všetky doteraz urobené kôpky a vyrobiť si novú.

Optimalizácia

Ďalšie z našej série pozorovaní je, že vyššie uvedený algoritmus má vedľajší efekt. V každom kroku nášho algoritmu sú všetky kôpky zoradené zostupne, podľa veľkosti! Ako správny KSP-áci toto predsa hneď musíme zneužiť. Čo vieme robiť na zoradenom poli? No predsa binárne vyhľadávanie!

Medzi zoradenými kôpkami teda vieme pomocou binárneho vyhľadávania, v čase $O(\log n)$, nájsť našu ideálnu kôpku pre aktuálnu škatuľu. Toto nám zlepší celkovú časovú zložitosť na $O(n \log n)$.

Listing programu (Python)

```python
def pocet_kopok(skatule, n):
    kopky = [0] * n
    for sila in skatule:
        vhodna_kopka = bin_najdi_kopku(kopky, n, sila)
        kopky[vhodna_kopka] += 1
    return len(kopky) - kopky.count(0)
```

Implementovanie samotnej funkcie `bin_najdi_kopku` nechávame ako cvičenie pre čítateľa.

Viac optimalizácií

Keď sa trochu zamyslíme nad vyššie spomínanou optimalizáciou, zistíme, že my vlastne vôbec nemusíme binárne vyhľadávať v takomto poli kôpok.

Predpokladajme, že prvých zopár škatúľ už máme nejako rozostavaných a teraz ideme umiestniť skupinu škatúľ s rovnakou pevnosťou. Kam by ich dal náš $O(n \log n)$ algoritmus? Najskôr by ich dával pod prvú kôpku, ktorú sú schopné udržať. Následne by pokladal ďalšie škatule pod túto istú kôpku, až dokedy by nebola príliš veľká pre našu silu škatúľ. Pokiaľ je už aktuálna kôpka príliš veľká, posunieme sa na ďalšiu. Ďalšia kôpka je zaručene dostatočne malá na to, aby sme pod ňu mohli umiestniť aspoň jednu škatuľu.

Keď prechádzame na škatule s väčšou pevnosťou, tieto pevnejšie škatule sú určite schopné uniesť všetky existujúce kopy. Má teda zmysel sa po tomto prechode vrátiť opäť na prvú kôpku a postup opakovať, teraz už ale so silnejšími škatuľami.

Tento algoritmus má časovú zložitosť $O(n)$, keďže pre každú škatuľu nájdeme príslušnú kopu v konštantnom čase. Všetky doteraz spomínané riešenia majú pamätovú zložitosť $O(n)$, pretože si potrebujeme pamätať iba dve polia dĺžky najviac n: pevnosti škatúľ a veľkosti kôp.

Listing programu (Python)

```python
def skatule_sily(skatule):
    """
    Zo síl zo vstupu vráti dvojice (sila, pocet).
    """
    posledna_sila = skatule[0]
    pocet = 0
    pocty_sil = []
    for sila in skatule:
        if sila == posledna_sila:
            pocet += 1
        else:
            pocty_sil.append([posledna_sila, pocet])
            posledna_sila = sila
            pocet = 1
    pocty_sil.append([posledna_sila, pocet])
    return pocty_sil


def pocet_kopok(skatule, n):
    sily = skatule_sily(skatule)
    kopky = [0] * n
    for sila, pocet in sily:
        aktualna_kopka = 0
        while pocet > 0:
            while sila >= kopky[aktualna_kopka] and pocet > 0:
                kopky[aktualna_kopka] += 1
```

```
            pocet -= 1
        aktualna_kopka += 1
    return len(kopky) - kopky.count(0)


n = int(input())
krabice = list(map(int, input().split()))
print(pocet_kopok(krabice, n))
```

Toto je naše optimálne riešenie a dostaneme zaň pekných 8 bodov.

Poznámka: Pochopiteľne, konverzia, ktorú funkcia `skatule_sily` vykonáva nie je vôbec potrebná ku korektnej funkčnosti nášho algoritmu. Tento upravený formát vstupu nám iba umožňuje mať prehľadnejší kód v `pocet_kopok`.

Exotika

Náš miestny zelovocár s exotickými riešeniami vymyslel aj riešenie, ktoré má časovú zložitosť $O(n)$ a pamäťovú $O(1)$. Funguje na trochu inom príncípe ako všetky naše riešenia, ktoré sú konštruktívne - snažia sa škatule naozaj rozdeliť na kopy. Toto riešenie je nekonštruktívne. Pre každú škatuľu na základe jej pevnosti a počtu slabších škatúľ vypočíta dolný odhad na počet všetkých kôp. Nájsť toto riešenie je pekným cvičením pre trochu skúsenejších riešiteľov.

mikx (`mikx@ksp.sk`)

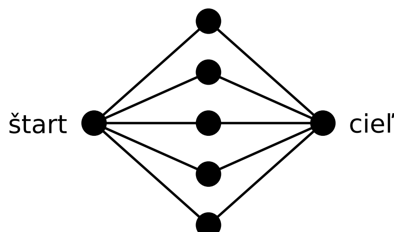
4. Éra pečúceho slnka

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

Konštrukčná úloha, ktorá má neskutočné množstvo správnych riešení. Ukážeme si, čo je kameň úrazu tejto úlohy, čo s ním a nakoniec aj nejaké jedno možné[3] riešenie.

Kameň úrazu

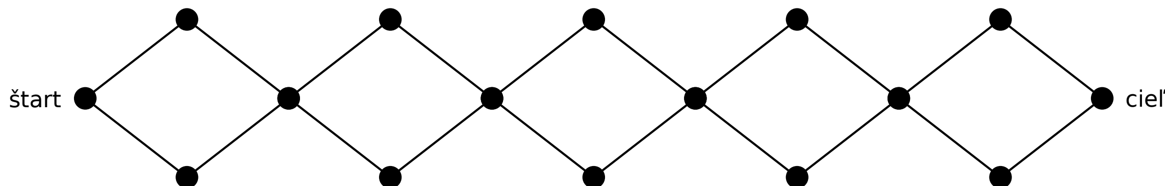
Najtragickejšia veta zadania je tá, že počet domov vami navrhnutého mesta nesmie presiahnuť 1 000. Prečo? Predstavme si, že by sme na toto nemali žiadne obmedzenie. Stačí nám vytvoriť niekoľko neprekrývajúcich sa ciest. Okrem štartovacieho a cieľového domu budeme mať v meste ešte k ďalších, prechodných domov. Každý prechodný dom spojíme chodníkom so štartom aj s cieľom.



V našej úlohe je však počet ciest podstatne väčší, ako počet domov, ktoré môžeme použiť. Nejakým spôsobom potrebujeme na malý počet pridaných domov dosiahnuť veľký počet rovnako dlhých ciest. Dámy a páni, predstavujem vám:

Diamant

Diamantom budeme nazývať štyri domy spojené do kosoštvorca. Keď jeden z domov prehlásime za začiatok a dom oproti (cez uhlopriečku) za koniec, tak medzi nimi existujú práve 2 najkratšie cesty (dĺžky 2). Na tomto útvare je super, že sa dajú skladať za seba. Ak naskladáme 5 diamantov za seba (koniec jedného bude začiatkom druhého), dosiahli sme $2^5 = 32$ možných rôznych najkratších ciest (každá postupnosť vľavo-vpravo-vľavo-vľavo-vľavo, vľavo-vpravo-vľavo-vľavo-vpravo, atď. je iná cesta). A to celé len za cenu $3 \times 5 + 1 = 16$ domov.



Ak chceme $2^{30} = 1\,073\,741\,824$ ciest, stačí naskladať 30 diamantov za seba, za cenu $3 \times 30 + 1 = 91$ domov. Celkom dobré, nie?

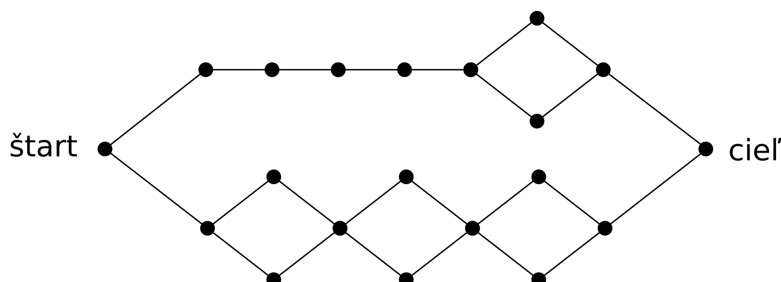
[3] samozrejme najkrajšie

Takže ak k je mocnina dvojky, len poskladáme príslušný počet diamantov. Čo však, ak nie je? Žiaden problém.

Príklad

Predstavme si, že $k = 10 = 8 + 2 = 2^3 + 2^1$. Toto nám vraví, že nejako by sa to mohlo dať, ak použijeme dve diamantové reťaze (série naskladaných diamantov), jednu zloženú z troch a druhú z jedného diamantu.

Reťaz s tromi diamantami obsahuje 8 ciest dĺžky 6, reťaz s jedným diamantom 2 cesty dĺžky 2. Keďže všetky cesty musia byť rovnako dlhé (aby boli všetky najkratšie), potrebujeme si cesty v kratšej reťazi nejako predĺžiť. To urobíme tak, že pred ňu pripojíme 4 vrcholy, ktoré nebudú mať inú úlohu, ako predlžovať cestu cez krátku reťaz. Takto upravené reťaze potom už len zapojíme vedľa seba (pridáme im spoločný štart a spoločný cieľ). Tadá.



Takýmto postupom vieme vyskladať ľubovoľné číslo.

V kocke

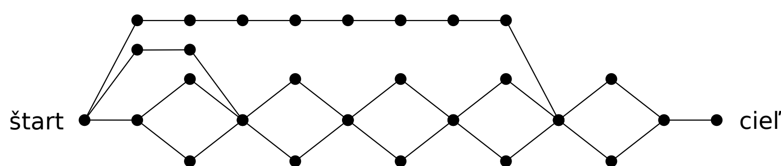
Každé číslo sa dá zapísať v binárnom zápise. To predstavuje, z akých mocnín dvojky sa skladá. Napr. $13 = 1101_2 = 2^3 + 2^2 + 2^0$. Pre tieto mocniny dvojky postavíme príslušné reťaze a všetky reťaze predĺžime na dĺžku najdlhšej. Takto upravené reťaze zapojíme vedľa seba a hotovo!

Koľko domov minieme?

Tak a teraz nepríjemná otázka s ešte horším zistením. Aby sme dokázali vytvoriť mesto s $2^{29} - 1 = 536\,870\,911 = 11111111111111111111111111111_2$ najkratšími cestami (čo je stále menej ako maximálna hodnota zo zadania), potrebujeme všetky reťaze dĺžok $1, 2, \ldots, 28$. Už len vyskladanie tohto nám minie $4 + 7 + 10 + \cdots + 85 = 1246$ domov. Príliš veľa. Čo s tým vieme robiť?

Posledný trik

Trikom je nestavať veľa reťazí, ale iba jednu. Zo všetkých našich reťazí si necháme iba tú s najväčšou dĺžkou. Do tejto reťaze sa na niekoľkých miestach pripojíme vhodne dlhými cestičkami zo začiatku tak, aby sme využili len nejakú jej podčasť. Napríklad pre $k = 50 = 110010_2$ to bude vyzerať takto:



Koľko stojí toto? Diamantová reťaz môže mať dĺžku najviac 29 diamantov, čo je $29 \times 3 + 1 = 88$ domov. Okrem toho máme ešte začiatočný dom, koncový dom a cestičky pripájajúce sa na reťaz (takzvané *slíže*). Slíž pripájajúci sa na reťaz za x-tým diamantom má dĺžku $2x$. Ak by sme mali slíže všetkých možných dĺžok od 1 po 29 diamantov, dokopy by stáli $2 + 4 + 6 + \cdots + 58 = 870$ domov.

Dokopy teda nebudeme potrebovať viac než $88 + 2 + 870 = 960$ domov a sme strašne šťastní, pretože to je pod 1 000 :). Samozrejme, nejaké malé (a možno aj väčšie) optimalizácie sa ešte dajú porobiť, netreba ich však.

Listing programu (Python)

```
n = int(input())

MAX_POWER = 29   # 2^30 > 10**9
path_len = 1 + 2*MAX_POWER + 1

V = 2
E = []

# vytvorime chainu MAX_POWER diamantov na sebe, na vrchole 2
prev = V
```

```
V += 1
for _ in range(MAX_POWER):
    E.append([prev, V])
    E.append([prev, V+1])
    E.append([V, V+2])
    E.append([V+1, V+2])

    prev = V+2
    V = V+3

# vrchol pripojim na ciel
# este ale neviem presne, ake bude maximalne V, cize pouzijem nejaky sentinel
# preto taka velka hodnota
opalovak = 10**20
E.append([V-1, opalovak])

mocn, i = 1, 0
while mocn <= n:
    if mocn & n:
        # spravime sliz prislusnej dlzky
        # za kazdy missnuty diamant 2 vrchole
        prev = 1
        for _ in range(path_len - 2 - 2*i):
            E.append([prev, V])
            prev = V
            V += 1

        # a pripojime na spravne miesto do diamantu
        E.append([prev, 3*(MAX_POWER-i+1)-1])

    i += 1
    mocn *= 2

print(V, len(E))
for e in E:
    # iba opalovak bol vyssi nez V, takze iba unho bude hodnota teraz menena
    print(min(e[0], V), min(e[1], V))
```

Andrej (ajok@ksp.sk)

5. Letné nakupovanie

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

Najprirodzenejším spôsobom ako sa dá celá úloha reprezentovať je zrejme grafom[4]. Vrcholmi sú v tomto grafe obchody a chata, pričom hranami sú cesty medzi nimi. Grafová reprezentácia je výhodná najmä v tom, že na grafoch poznáme mnoho algoritmov.

Riešenie hrubou silou

Ak sa pýtame na najmenší čas, za ktorý sa dá nakúpiť p fixiek za cenu najviac c, môžeme postupovať jednoduchým spôsobom a pýtať sa : "ide to za čas 1?", "ide to za čas 2?"...

Akonáhle je odpoveď na nejakú takúto otázku áno, vieme, že sme našli najmenší čas, za ktorý vieme p fixiek nakúpiť. Ide ale jednoducho nájsť odpoveď na takéto otázky? Ukážeme si, že áno. Vezmime si otázku : "ide to za čas x?" a poďme na ňu skúsiť nejak odpovedať.

Čo najskôr potrebujeme urobiť, je nájsť obchody z ktorých máme na výber, teda zistiť, ktoré ležia do vzdialenosti x. To vieme ľahko jedným prehľadávaním do šírky[5] v lineárnom čase. Keď máme tieto obchody, chceme vedieť či ide len pomocou nich kúpiť p fixiek za cenu najviac c. Ak si zoradíme obchody, ktoré máme k dispozícií podľa ceny za jednu fixku, vieme postupne nakupovať fixky od najlacnejších obchodov až po najdrahšie. Čo sa môže stať sú 3 veci:

```
1. fixky úspešne nakúpime za cenu dokopy menej ako c
2. počas nakupovania fixiek nám dojdú peniaze
3. počas nakupovania nám dojdú obchody, z ktorých by sme mohli nakúpiť
```

Asi je jasné, že iba prvá možnosť znamená, že to "ide za čas x", ostatné znamenajú, že zatiaľ sme s touto vzdialenosťou nepochodili a musíme sa pozrieť na obchody o kus ďalej.

Akú má toto riešenie časovú zložitosť? Koľko môže byť otázok typu : "ide to za čas x?". Je to zjavne n. S časom $x = n$ už vieme určite použiť všetky obchody, žiadny obchod nemôže byť ďalej a pridaním času sa nám teda už ponuka nezvýši, inými slovami, ak to nejde za čas n, nepôjde to ani za čas $n + 1$, $n + 2$, $n + 3$... Koľko najviac môže trvať odpoveď na takúto otázku? Zoradiť všetky obchody a raz ich prejsť bude trvať pokaždé $O(n \cdot log(n))$. Teraz vieme v čase $O(n \cdot n \cdot log(n) + m)$ riešiť túto úlohu. Akú ma toto pamäťovú zložitosť? Ukazuje sa, že tak ako všetky naše riešenia to bude $O(n + m)$. V každom našom riešení nám bude stačit zapamätať si konštantne veľa vecí pre každý obchod a potom ešte zoznam susedov, reprezentujúci graf.

[4] https://www.ksp.sk/kucharka/grafy_uvod/
[5] https://www.ksp.sk/kucharka/bfs/

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

// prehladavanie do sirky na grafe "G" z vrcholu "zac"
void bfs(const int zac, const vector<vector<int> >& G, vector<int>& vzd)
{
    queue<int> Q;

    vzd[zac] = 0;
    Q.push(zac);

    while(!Q.empty())
    {
        int v = Q.front();
        Q.pop();

        for(const int kandidat : G[v]) if(vzd[kandidat] == -1)
        {
            vzd[kandidat] = vzd[v] + 1;
            Q.push(kandidat);
        }
    }

    return;
}

// vrati true ak ide nakupit "max_pocet" fixiek za cenu najviac "max_cena" len s obchodmi
// v poli "moznosti"
bool ide(vector< pair<int, int> >& moznosti, const int max_cena, const int max_pocet)
{
    // chceme prechadzat od najlacnejsich obchodov
    sort(moznosti.begin(), moznosti.end());

    int spolu = 0, cena = 0;
    for(int j=0;j<moznosti.size();++j)
    {
        if(cena > max_cena) break;

        if(spolu + moznosti[j].second <= max_pocet)
        // pridame vsetky fixky z obchodu
        {
            spolu += moznosti[j].second;
            cena += moznosti[j].first * moznosti[j].second;
        }
        else
        // pridame cast fixiek z obchodu a skoncime
        {
            cena += moznosti[j].first * (max_pocet-spolu);
            spolu = max_pocet;
            break;
        }
    }

    if(spolu == max_pocet && cena <= max_cena) return true;

    return false;
}

int main()
{
    ios_base::sync_with_stdio(false);

    int n, m, max_pocet, max_cena;

    cin >> n >> m >> max_pocet >> max_cena;

    vector<vector<int> > G(n+1, vector<int>());
    vector<int> pocty(n, 0), ceny(n, 0);

    for(int i=0;i<n;++i) cin >> pocty[i];

    for(int i=0;i<n;++i) cin >> ceny[i];

    for(int i=0;i<m;++i)
    {
        int a, b;
        cin >> a >> b;

        G[a].push_back(b);
        G[b].push_back(a);
    }

    vector<int> vzd(n+1, -1);

    bfs(n, G, vzd);

    for(int i=0;i<=n+1;++i)
    // "i" nam hovori aku vzdialenost teraz testujeme
    {
        vector<pair<int, int> > moznosti;

        // do vectoru moznosti nahadzeme vsetko, co je do vzdialenosti "i"
        for(int j=0;j<n;++j) if(vzd[j] <= i)
```

```
        {
            moznosti.push_back( {ceny[j], pocty[j]} );
        }

        // spytame sa, ci existuje riesenie, ak ano, vypiseme ho a skoncime
        if( ide(moznosti, max_cena, max_pocet) )
        {
            cout << i << endl;
            return 0;
        }
    }

    // ak sme riesenie nenasli, neexistuje
    cout << "-1" << endl;

    return 0;
}
```

Ako naše riešenie teraz časovo zlepšiť?

Zlepšenie riešenia hrubou silou

K jednoduchému zlepšeniu vedie nasledovné pozorovanie: Ak vieme p fixiek nakúpiť za cenu c a to všetko za čas x (teda pomocou obchodov do vzdialenosti x), vieme to určite aj za čas $x+1$, $x+2$, $x+3$ atď. Pokiaľ je x najmenší čas, za ktorý to vieme, tak zároveň platí, že to nevieme za čas $x-1$, $x-2$, $x-3\ldots$ 0. To ale znamená, že vieme odpoveď jednoducho binárne vyhľadať. Povedali sme si, že najskôr ide p-fixiek nakúpiť za čas 1, najneskôr za čas n. Ak si vezmeme nejaký čas x, môžu sa stať dve veci:

```
1. za čas x ide nakúpiť p fixiek za cenu najviac c a teda musíme najmenšiu odpoveď
hľadať vo vzdialenosti menšej alebo rovnej ako x.
2. za čas x nejde nakúpiť p fixiek za cenu najviac c a teda musíme najmenšiu odpoveď
hľadať vo vzdialenosti väčšej ako x.
```

Tento malý trik si zapamätajte. V úlohách, kde treba hľadať najmenší čas za ktorý sa niečo dá spraviť, sa tentro trik používa pomerne často. Akú má tento postup časovú zložitosť? Počet otázok, na ktoré vieme stále odpovedať v čase $O(n \cdot log(n))$, sa zmenšil z n na $log(n)$. Výsledná časová zložitosť je teda $O(log(n) \cdot n \cdot log(n))$ alebo inak $O(n \cdot log(n)^2 + m)$. Toto už stačí na prejdenie všetkými vstupmi.

Listing programu (C++)

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

void bfs(const int zac, const vector<vector<int> >& G, vector<int>& vzd)
{
    queue<int> Q;

    vzd[zac] = 0;
    Q.push(zac);

    while(!Q.empty())
    {
        int v = Q.front();
        Q.pop();

        for(const int kandidat : G[v]) if(vzd[kandidat] == -1)
        {
            vzd[kandidat] = vzd[v] + 1;
            Q.push(kandidat);
        }
    }

    return;
}

bool ide(vector< pair<int, int> >& moznosti, const int max_cena, const int max_pocet)
{
    sort(moznosti.begin(), moznosti.end());

    int spolu = 0, cena = 0;
    for(int j=0;j<moznosti.size();++j)
    {
        if(cena > max_cena) break;

        if(spolu + moznosti[j].second <= max_pocet)
        {
            spolu += moznosti[j].second;
            cena += moznosti[j].first * moznosti[j].second;
        }
        else
        {
            cena += moznosti[j].first * (max_pocet-spolu);
```

```
            spolu = max_pocet;
            break;
        }
    }

    if(spolu == max_pocet && cena <= max_cena) return true;

    return false;
}

int main()
{
    ios_base::sync_with_stdio(false);

    int n, m, max_pocet, max_cena;
    cin >> n >> m >> max_pocet >> max_cena;

    vector<vector<int> > G(n+1, vector<int>());
    vector<int> pocty(n, 0), ceny(n, 0);

    for(int i=0;i<n;++i) cin >> pocty[i];
    for(int i=0;i<n;++i) cin >> ceny[i];

    int a, b;
    for(int i=0;i<m;++i)
    {
        cin >> a >> b;
    // Tu doslo k zmene
    ///////////////////////////////////////////////////////////////////
        G[a].push_back(b);
        G[b].push_back(a);
    }

    vector<int> vzd(n+1, -1);
    bfs(n, G, vzd);

    int zac = 1, stred, kon = n+1;

    while(kon - zac > 2)
    {
        stred = (zac+kon)/2;

        vector<pair<int, int> > moznosti = nahadz()

        for(int j=0;j<n;++j) if(vzd[j] <= stred)
        {
            moznosti.push_back( {ceny[j], pocty[j]} );
        }

        if( ide(moznosti, max_cena, max_pocet) ) kon = stred+1;
        else zac = stred;
    }

    for(int i=zac;i<kon;++i)
    {
        vector<pair<int, int> > moznosti;

        for(int j=0;j<n;++j) if(vzd[j] <= i)
        {
            moznosti.push_back( {ceny[j], pocty[j]} );
        }

        if( ide(moznosti, max_cena, max_pocet) )
        {
            cout << i << endl;
            return 0;
        }
    }

    cout << "-1" << endl;

    ///////////////////////////////////////////////////////////////////

    return 0;
}
```

Vzorové riešenie

Vzorové riešenie bude akosi trochu kopírovať naše prvé riešenie hrubou silou. Zas sa bude náš algoritmus pozerať na problém po úrovniach. Chceme totiž využiť vlastnosť, že akonáhle fixky, ktoré máme k dispozícií spĺňajú podmienky počtu a ceny, vieme, že naše riešenie je najlepšie možné. Riešenie hrubou silou malo nevýhodu, že po každej úrovni zahodilo všetky informácie, ktoré o grafe získalo. Ako budeme teda postupovať?

Budeme prechádzať obchody postupne po úrovniach (podľa vzdialenosti od chaty) a udržiavať si v akomsi virtuálnom nákupnom košíku dostatočný počet, doteraz najlacnejších fixiek. Do košíku najskôr naložíme všetky fixky z obchodov v nejakej úrovni. Ak je počet fixiek stále moc nízky, pokračujeme ďalšou úrovňou. Ak je počet fixiek v košíku moc veľký, začneme fixky vyhadzovať pokým ich nie je v košíku toľko, koľko chceme. Vyhadzovať ich samozrejme budeme od najdrahších. Po dovyhodzovaní fixiek je ich určite v košíku toľko, koľko potrebujeme. Ak je ale cena privysoká, pokračujeme ďalšou úrovňou. Akonáhle nájdeme riešenie, je jasné, že žiadne lepšie neexistuje, kedže by sme ho boli objavili skôr.

Ako bude vyzerať implementácia? Od košíka chceme, aby sme doňho vedeli rýchlo vkladať dvojice (cena_za_fixku_v_obchode, obchod) a vedeli z neho rýchlo vyberať a pozerať sa na akutálne najdrahší obchod, z ktorého máme nejaké fixky. Pokaždé keď fixky vyhadzujeme, stačí sa nám pozrieť na vrchný najdrahší obchod, z ktorého fixky v košíku máme. Asi tušíte, že na toto je vhodná dátová štruktúra maximová halda[6].

Aká bude časová zložitosť tohto riešenia? Obchody musíme určite zoradiť, každý obchod pridáme do košíka iba raz a z košíka vyhadzujeme iba na jednotlivých úrovniach. Časová zložitosť je teda $O(n * log(n) + m)$. Pamäťová zložitosť ostáva $O(n + m)$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

void bfs(int zac, const vector<vector<int> >& G, vector<int>& vzd)
{
    queue<int> Q;

    vzd[zac] = 0;
    Q.push(zac);

    while(!Q.empty())
    {
        int v = Q.front();
        Q.pop();

        for(const int kandidat: G[v]) if(vzd[kandidat] == -1)
        {
            vzd[kandidat] = vzd[v] + 1;
            Q.push(kandidat);
        }
    }

    return;
}

int main()
{
    ios_base::sync_with_stdio(false);

    int n, m, max_pocet, max_cena;
    cin >> n >> m >> max_pocet >> max_cena;

    vector<vector<int> > G(n+1, vector<int>());
    vector<int> pocty(n, 0), ceny(n, 0);

    for(int i=0;i<n;++i) cin >> pocty[i];
    for(int i=0;i<n;++i) cin >> ceny[i];

    for(int i=0;i<m;++i)
    {
        int a, b;
        cin >> a >> b;

        G[a].push_back(b);
        G[b].push_back(a);
    }

    vector<int> vzd(n+1, -1);
    bfs(n, G, vzd);

    // v poli dvojic "por" su obchody zoradene podla vzdialenosti
    vector<pair<int, int> > por(n);

    for(int i=0;i<n;++i) por[i] = {vzd[i], i};

    // klasicke zoradovanie dvojic podla prveho cisla, teda vzdialenosti
    sort(por.begin(), por.end());

    // maximalna vzdialenost, do ktorej hladame obchody, i index prveho obchodu,
    // ktory zatial nepouzivame
    int i = 0, max_vzd = 1;
    // cena veci v kosiku a pocet veci v kosiku
    long long cena = 0, pocet = 0;

    // PQ symbolizuje nas nakupny kosik
    priority_queue< pair<int, int> > PQ;
    // v poli nakupene si pamatame pre kazdy obchod, kolko fixiek mame z neho v kosiku
    vector<int> nakupene(n, 0);

    // i je index prveho obchodu, ktory zatial neuvazujeme
    while(i<n)
    {
        // najskor pridavame obchody pokym su do vzdialenosti max_vzd
        while(i<n && por[i].first <= max_vzd)
        {
            int obchod = por[i].second;
```

[6] https://www.ksp.sk/kucharka/halda/

```
            nakupene[obchod] += pocty[obchod];
            pocet += pocty[obchod];
            cena += pocty[obchod] * ceny[obchod];

            PQ.push( {ceny[obchod], obchod} );

            ++i;
        }

        ++max_vzd;

        // ak pocet nie je dostatocny, skaceme na dalsiu uroven
        if(pocet < max_pocet) continue;

        // vyhadzujeme pokym nie je pocet fixiek v kosiku presne max_pocet
        while(1)
        {
            int obchod = PQ.top().second;

            // prva moznost je ze chceme kompletne z kosika odstranit fixky z najdrahsieho
            // obchodu "obchod" lebo ich aj tak budeme mat dost
            if( pocet - nakupene[obchod] >= max_pocet)
            {
                PQ.pop();
                pocet -= nakupene[obchod];
                cena -= nakupene[obchod] * ceny[obchod];
                nakupene[obchod] = 0;
            }
            else
            {
                // druha moznost je, ze nechceme odstranit vsetky z najdrahsieho obchodu
                // lebo by sme nemali dost fixiek
                long long cast = pocet - max_pocet;
                pocet -= cast;
                cena -= cast * ceny[obchod];
                nakupene[obchod] -= cast;
                break;
            }
        }

        // ked sme skoncili a fixiek je urcite v kosiku max_pocet,
        // skontrolujeme ci sedi cena
        if(cena <= max_cena)
        {
            cout << max_vzd-1 << endl;
            return 0;
        }
    }

    cout << "-1" << endl;

    return 0;
}
```

Žaba (zaba@ksp.sk)

6. Egyptské pyramídy

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

Zopakujme si zadanie úlohy. Máme spočítať počet pyramíd, ktoré majú h poschodí, vrchné je veľkosti 1×1, spodné $x \times x$ a poschodia medzi sú “takmer štvorce”, teda ich šírka a dĺžka sa nelíši o viac ako 3. Navyše sa poschodia postupne od spodu zužujú, presnejšie, šírka aj výška každého poschodia je aspoň o 2 menšia ako poschodia priamo pod ním.

Vždy, keď riešime problém s počítaním možností mali by sme sa zamyslieť, či nevieme využiť dynamické programovanie. Pri tejto technike si najskôr sformulujeme problém a následne skúšame vypočítať jeho riešenie na základe jemu podobných podproblémov.

Nie vždy je vhodné si ako problém zobrať úlohu zadania, skôr by malo platiť, že pomocou zvoleného problému vieme ľahko na zadanie odpovedať. V tomto prípade je to však pomerne ľahké a priamo zo zadania vyplýva nasledovná otázka: *Koľkými spôsobmi vieme postaviť korektnú pyramídu výšky n, ktorej spodné poschodie má veľkosť $w \times h$?*

Keďže pyramídy nemusia mať vždy štvorcové podstavy, problém sme si trochu zovšeobecnili na podstavy $w \times h$. Stále však vieme ľahko odpovedať na pôvodnú otázku, stačí, keď nastavíme $w = x$ a $h = x$.

Ďalším krokom je zistiť, či vieme na túto otázku odpovedať pomocou výsledkov pre jednotlivé podproblémy. Čo to vlastne ten podproblém je? Je to pôvodná otázka, do ktorej dosadíme iné, menšie parametre. Teda napríklad, koľko je pyramíd výšky $n - 1$ s veľkosťou základne 4×3. Aby sme nemuseli našu otázku vždy rozpisovať, označme si počet pyramíd výšky n so základňou $w \times h$ ako $P(n, w, h)$.

Náš pôvodný problém je nájsť hodnotu $P(n, x, x)$. Ako takéto pyramídy vyzerajú? Ich spodné poschodie je veľké $x \times x$ a potom sú tam poukladané zvyšné poschodia. Tieto zvyšné poschodia však tvoria korektnú pyramídu výšky $n - 1$. Jediné čo nevieme je, akú podstavu tieto menšie pyramídy majú. Uvedomme si však, že napríklad, každú pyramídu výšky $n - 1$, ktorej podstava je $(x - 2) \times (x - 2)$ vieme postaviť na poschodie $x \times x$ a dostať tak pyramídu, ktorú hľadáme. Tým pádom, hodnota $P(n, x, x)$ obsahuje všetky možnosti $P(n - 1, x - 2, x - 2)$.

A to je už skladanie podproblémov. Vidíme, že ak vypočítame $P(n-1, x-2, x-2)$, táto hodnota nám pomôže pri výpočte hľadaného $P(n, x, x)$. No ale $P(n-1, x-2, x-2)$ je ten istý problém, len trochu menší, na jeho riešenie preto môžeme použiť ten istý postup. A to je základnou myšlienkou dynamického programovania.

Ak chceme vypočítať $P(n, x, x)$, musíme zistiť, aké všetky pyramídy veľkosti $n-1$ vieme položiť na poschodie $x \times x$. Ich výšku poznáme, stačí teda skúsiť všetky možnosti pre podstavu. Z toho plynie nasledovný jednoduchý program. Odporúčam si ho prečítať, ak ste sa s dynamickým programovaním ešte nestretli. Pekne ukazuje, aká jednoduchá je to technika, keď si položíte správnu otázku.

Listing programu (C++)

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

int MOD = 1000000;
int Mem[300][700][700];

// počet pyramíd výšky n, ktorých podstava je veľká w krát h
int P(int n, int w, int h) {
    // skontroluj najjednoduchšie prípady a už zapamätané riešenia
    if(n == 1 && w == 1 && h == 1) return 1;
    if(n == 1) return 0;
    if(w <= 0 || h <= 0) return 0;
    if(Mem[n][w][h] != -1) return Mem[n][w][h];
    // vyskúšaj všetky možné podstavy
    int res = 0;
    for(int i = 0; i < w-1; i++)
        for(int j = 0; j < h-1; j++) {
            if(abs(i-j) > 3) continue; // skontroluj, či je to takmer štvorec
            res += P(n-1, i, j);
        }
    return Mem[n][w][h] = (res % MOD);
}

int main() {
    int n,x;
    scanf("%d %d",&n,&x);
    // nastavím pole Mem na -1 - nič ešte nemáme spočítané
    for(int i=0; i<300; i++)
        for(int j=0; j<700; j++)
            for(int k=0; k<700; k++)
                Mem[i][j][k] = -1;

    printf("%d\n",P(n,x,x));
}
```

Vo vyššie uvedenom programe si všimnite dve dôležité veci. Jednak, naša rekurzia musí mať ukončenie. To je väčšinou určené podproblémom, ktorý vieme vyriešiť ručne. V našom prípade pyramída výšky 1 s podstavou 1×1, ktorá je naozaj len jedna. Takisto si však treba dať pozor na prípady, ktoré už k riešeniu nevedú. Druhým pozorovaním je, že aby sme dookola nepočítali tie isté veci, pamätáme si už vypočítané výsledky v poli `Mem[]`. Táto technika sa volá memoizácia a výrazne zrýchľuje väčšinu rekurzívnych funkcií. Naozaj jediné čo s `Mem[]` robíme je, že si do tohto poľa vkladáme vypočítané hodnoty a ak sa pýtame na niečo, čo už máme zapamätané, vrátime namiesto toho túto hodnotu.

Aká je časová zložitosť nášho programu. Pri takomto type rekurzií sa to v skutočnosti odhaduje pomerne ľahko. Stačí zistiť koľko rôznych podproblémov môžeme riešiť a ako dlho nám trvá riešenie jedného z nich. Možných podproblémov je $n \cdot x \cdot x$ – všetky možnosti pre výšku, šírku a dĺžku pyramídy. Výpočet jednej možnosti je pritom $x \cdot x$, pretože musíme vyskúšať všetky možnosti pre šírku a dĺžku podstavy o poschodie vyššie. Celková zložitosť je preto $O(nx^4)$.

Zapojenie "takmer štvorcov"

Naše riešenie je príliš pomalé, je však zrejmé, že to tak vôbec nemusí byť a veľa vecí sme si zbytočne zjednodušili. Napríklad sme nijak nevyužili fakt, že jednotlivé poschodia musia byť takmer štvorce. Skúšame preto úplne zbytočné možnosti ako $P(10, 9, 3)$. Síce vieme spočítať počet takýchto pyramíd, ale na čo nám sú, ak podstavu 9×3 nikdy nevieme použiť? Toto platí o rekurziách a dynamikách aj všeobecne. Keď sa ich snažíme zrýchliť, vždy si treba klásť otázku, či niečo nepočítame dvakrát, poprípade zbytočne.

Zapojenie takmer štvorcov ovplyvňuje dve miesta v kóde – počítanie možných podstáv pre nižšie pyramídy a samotný počet a tvar podproblémov. Začnime tým prvým. V predchádzajúcom riešení sme pre pyramídu výšky $n-1$ skúšali všetky možné šírky a dĺžky pre podstavu. My však vieme, že ich rozdiel môže byť najviac 3. Ako v riešení vyskúšať všetky takéto veľkosti? Uvedomme si, že stále chceme skúšať každú možnú šírku, k danej šírke však už neskúšame všetky dĺžky, ale iba tých 7 zaujímavých – o 3 menšiu až o 3 väčšiu ako šírka. Toto zrýchli počítanie možností na $O(x)$, keďže 7 je iba malá konštanta.

Druhú úpravu musíme spraviť priamo vo formáte podproblému, nechceme počítať a vôbec dovoľovať podproblémy, ktoré nás nezaujímajú. Jeden z dôvodov je napríklad aj ten, že sa nám tým zmenší veľkosť nášho

poľa `Mem[]`, čo je dôležité, pretože vytvoriť pole nejakej veľkosti trvá rovnako veľa času. Ak by sme teda tieto podproblémy nepočítali, ale mali by sme pre ne vytvorené miesto v pamäti, vôbec by sme si nepomohli. Náš problém si preto preformulujeme na *Koľkými spôsobmi vieme postaviť korektnú pyramídu výšky n, ktorej spodné poschodie má šírku w a dĺžku o dx inú?* Pričom samozrejme dx môže mať iba hodnoty -3 až 3.

Ako sa zmenila časová zložitosť našeho riešenia? Počet podproblémov je už iba $n \cdot 7x$ – určíme si výšku, šírku a dĺžka už môže mať iba jednu zo 7 možností na základe šírky. A jeden podproblém vypočítame v čase $7x$ – skúsime každú šírku a iba 7 okolitých dĺžok. Výsledné riešenie má teda zložitosť $O(n \cdot x^2)$.

K implementácii tohto riešenia ešte dodajme, že je ťažké si pole indexovať zápornými číslami, preto dx nebude nadobúdať hodnoty -3 až 3 ale 0 až 6. Počet hodnôt je rovnaký, sú kladné, je však na nás, aby sme sa v nich nedoplietli a správne ich používali.

Listing programu (C++)

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

int MOD = 1000000;
int Mem[2000][5000][7];

// počet pyramíd výšky n, ktorých podstava je veľká w krát w+dx-3
int P(int n, int w, int dx) {
    int h = w + dx - 3;
    // skontroluj najjednoduchšie prípady a už zapamätané riešenia
    if(n == 1 && w == 1 && dx == 3) return 1;
    if(n == 1) return 0;
    if(w <= 0 || h <= 0) return 0;
    if(Mem[n][w][dx] != -1) return Mem[n][w][dx];
    // vyskúšaj všetky podstavy tvaru takmer štvorca
    int res = 0;
    for(int i = 0; i < w-1; i++)
        for(int j = -3; j < 4; j++) {
            if(h - (i + j) < 2) continue;
            res += P(n-1, i, j + 3);
        }
    return Mem[n][w][dx] = (res % MOD);
}

int main() {
    int n,x;
    scanf("%d %d",&n,&x);
    // nastavím pole Mem na -1 - nič ešte nemáme spočítané
    for(int i=0; i<2000; i++)
        for(int j=0; j<5000; j++)
            for(int k=0; k<7; k++)
                Mem[i][j][k] = -1;

    printf("%d\n", P(n,x,3));
}
```

Rýchlejšie počítanie jedného podproblému

Predchádzajúce riešenie však stále nie je dostatočne rýchle. Nie je však veľa vecí, ktoré by sme vedeli zmeniť. Počet podproblémov už veľmi nezmeníme, predsa len, musíme si pamätať výšku a veľkosť základne. To znamená, že treba zrýchliť výpočet jedného podproblému. Pri aktuálnom riešení postupne skúšame všetky o jedno nižšie pyramídy, ktoré sa zmestia na zadanú podstavu. Bohužiaľ, my poznáme odpoveď iba pre konkrétnu veľkosť podstavy a preto musíme robiť všetky tie skúšania. Čo by sa ale stalo, keby sme namiesto toho vedeli odpovedať na nasledovný problém: *Koľkými spôsobmi vieme postaviť korektnú pyramídu výšky n, ktorej **spodné poschodie je nanajvýš** w široké a $w+dx$ dlhé?* Označme si túto hodnotu $T(n, w, dx)$. Potom predsa platí, že $P(n, w, dx) = T(n-1, w-2, dx)$, pretože sa viem rovno dozvedieť počet pyramíd, ktoré viem dať na podstavu $w \times (w + dx)$.

No dobre, ale pomohli sme si vôbec? Lebo sme si iba vytvorili nový dynamický problém, ktorý však stále potrebujeme vedieť vypočítať. Ako však uvidíme, vyriešiť tento problém je o niečo jednoduchšie. Takže ešte raz, hľadáme počet pyramíd výšky n, ktorých podstava je nanajvýš $w \times (w + dx)$. Medzi ne určite patria pyramídy, ktorých podstava je presne takto veľká. A tých vieme, že je $T(n-1, w-2, dx)$. Následne už len potrebujeme pripočítať pyramídy, ktorých podstava je menšia. Menšia môže byť v šírke alebo dĺžke. Čo keby ich šírka bola najviac $w-1$? Potom počet takýchto pyramíd je $T(n, w-1, dx+1)$ – sú vysoké n poschodí, ich šírka je $w-1$ a ich dĺžka je rovnaká ako pred tým, čo spôsobí, že dx sa zväčší o 1 (lebo w sa o 1 zmenšilo). A keď chceme pyramídy, ktoré majú kratšiu dĺžku, dostaneme hodnotu $T(n, w, dx-1)$. No a hodnoty $T()$ zahŕňajú všetky **nanajvýš** takto veľké pyramídy, nemusíme preto skúšať všetky možné veľkosti podstavy.

Toto avšak nie je všetko, musíme si uvedomiť chybu v našom riešení. Hodnoty $T(n, w-1, dx+1)$ a $T(n, w, dx-1)$ obsahujú totiž niekoľko rovnakých pyramíd. Presnejšie, každá pyramída výšky n, ktorej podstava je nanajvýš $(w-1) \times (w+dx-1)$ spadá pod obe tieto čísla, pretože takto veľké podstavy sú rovnako nanajvýš veľké ako

podstavy $(w-1) \times (w+dx)$ a aj podstavy $w \times (w+dx-1)$. No dobre, ale tieto pyramídy sú práve pyramídy $T(n, w-1, dx)$ a ak sme ich započítali dvakrát, tak ich potrebujeme raz odpočítať. To nás dostáva k výslednému vzorcu:

$$T(n, w, dx) = T(n-1, w-2, dx) + T(n, w-1, dx+1) + T(n, w, dx-1) - T(n, w-1, dx)$$

Malá poznámka na záver. Tento vzorec je dobrý, ale občas nemusí byť úplne pravidvý. Totiž podstavy $(w-1) \times (w+dx)$ nemusia už byť platné, lebo ich rozdiel je priveľký. V takom prípade túto hodnotu nepripočítavame, čím nám nevzniknú duplikáty a teda nechceme ani odčítavať $T(n, w-1, dx)$. To sú však skôr implementačné detaily, na ktoré si treba dať pozor, keď sa však budete zamýšľať nad tým, čo píšete a čo tie veci znamenajú, ľahko sa im vyhnete.

Vo výsledku hodnoty $P()$ ani nepotrebujeme počítať, celú úlohu vieme vyriešiť pomocou hodnôt $T()$. A aká je zložitosť? Počet podproblémov $T()$ je stále $n \cdot 7x$. Akurát ich počítanie sa zrýchlilo, pretože nám stačí vypočítať vyššie uvedený vzorec, čo trvá konštantne veľa času. Celková časová zložitosť je $O(nx)$. Pamäťová zložitosť je totožná počtu podproblémov, teda tiež $O(nx)$. Túto by sme vedeli síce zmenšiť, ak by sme použili dynamické programovanie namiesto rekurzie s memoizáciou, takéto riešenie však nebolo vyžadované.

Listing programu (C++)

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

int MOD = 1000000;
int Mem[2000][5000][7];

// počet pyramíd výšky n, ktorých podstava je nanjvýš veľká w krát w+dx-3
int T(int n, int w, int dx) {
    int h = w + dx - 3;
    // skontroluj najjednoduchšie prípady a už zapamätané riešenia
    if(n == 1 && w >= 1 && h >= 1) return 1;
    if(w <= 0 || h <= 0) return 0;
    if(Mem[n][w][dx] != -1) return Mem[n][w][dx];
    // spočítaj možné pyramidy
    int res = T(n-1, w-2, dx);
    bool t = true;
    if(abs(w-1-h) <= 3) res += T(n, w-1, h-(w-1)+3);
    else t = false;
    if(abs(w-h+1) <= 3) res += T(n, w, (h-1)-w+3);
    else t = false;
    // ak som zarátal duplikáty
    if(t) res -= T(n, w-1, dx);
    return Mem[n][w][dx] = ((res%MOD + MOD) % MOD);
}

int main() {
    int n,x;
    scanf("%d␣%d",&n,&x);
    // nastavím pole Mem na -1 - nič ešte nemáme spočítané
    for(int i=0; i<2000; i++)
        for(int j=0; j<5000; j++)
            for(int k=0; k<7; k++)
                Mem[i][j][k] = -1;

    printf("%d\n",T(n-1, x-2, 3));
}
```

7. Telefonát

Samo (samo@ksp.sk)
(max. 12 b za popis, 8 b za program)

Na začiatok si uvedomme, že ciferný súčin nám dáva akýsi rozklad čísla n na súčin, ktorého každý člen tvorí samostatná cifra. To ale znamená, že v prvočíselnom rozklade čísla n (čo je jemnejší rozklad) sa môžu nachádzať iba prvočísla menšie ako 10. V opačnom prípade číslo s daným ciferným súčinom neexistuje a odpoveď je 0.

Neprvočíselné cifry a ciferný súčet

Medzi ciframi našeho čísla sa môžu nachádzať aj neprvočíselné hodnoty, preto nám rozklad n na prvočísla nedá jednoznačnú sadu cifier, ktorú musíme použiť. Toto sa týka cifier 4, 6, 8 a 9, v našom riešení sa preto budeme musieť rozhodnúť, koľko prvočísel 2 a 3 použijeme na ich vytváranie. Presnejšie, budeme sa musieť rozhodnúť, koľko cifier 4, 6, 8 a 9 chceme mať vo výslednom čísle.

Zabudnúť nemôžeme ani na druhú podmienku zadania – ciferný súčet, ktorý sa tiež musí rovnať n. Ciferný súčet je však väčšinou oveľa menší ako súčin, číslo preto vieme doplniť potrebným počtom cifier 1, ktoré ciferný súčin nemenia.

Ako vybrať sadu použitých cifier?

Koľko možností máme na výber počtu jednotlivých cifier? Vieme napríklad, že nemôžeme použiť žiadnu cifru 0, pretože celý ciferný súčin by bol tiež 0. A keďže poznáme prvočíselný rozklad n, vieme, že naše číslo musí obsahovať rovnaký počet 5 a 7 ako je v tomto rozklade. Otázne sú iba cifry 4, 6, 8 a 9, keby sme vedeli, koľko ich bude vo výsledku, počet 2 a 3 si vieme dopočítať z celkového počtu týchto čísel v rozklade.

Skúsime teda všetky možnosti, lebo vieme, že maximálny počet jednej cifry môže byť najviac $\log n$, keďže ich súčin nesmie presiahnuť hodnotu n. Určením týchto cifier sú už všetky ostatné jednoznačne dané, počet 1 dopočítame tak, aby sedel ciferný súčet (ak by nám tento počet vyšiel záporný, tak počty ostatných cifier nevyhovujú žiadnemu číslu).

Koľko možných čisiel máme pre danú sadu cifier?

Keď už vieme koľko jednotlivých cifier chceme použiť v našom čísle, potrebujeme zvoliť ich poradie. To totiž nemení ciferný súčet ani súčin, chceme preto zarátať všetky možné preusporiadania. Toto je už však pomerne ľahká kombinatorická úloha, ak si počet cifry i označíme ako P_i a počet všetkých cifier dokopy ako x, počet preusporiadaní je:

$$\binom{x}{P_1}\binom{x-P_1}{P_2}...\binom{x-P_1-...-P_8}{P_9} = \frac{x!}{P_1!P_2!...P_9!}$$

.

Na daný vzorec sa môžete pozerať cez kombinačné čísla (ľavá strana), keď si postupne volíme na ktoré pozície dáme ktoré cifry a počet voľný pozícií sa zmenšuje, alebo ako na permutácie s opakovaním (pravá strana), kde najskôr započítame všetky možné preusporiadania a potom odstraňujeme (delíme) tie, kde sme iba zmenili poradie rovnakých cifier.

Celkový počet cifier bude kvôli cifernému súčtu určite najviac n a preto si vieme predpočítať všetky faktoriály až po hodnotu x dopredu a to dokonca modulo 10^9+7. Tu však nastáva problém, pretože v okamihu, keď začneme čísla modulovať, normálne delenie prestane správne fungovať. Našťastie, na vyriešenie tohto problému poznáme pomerne klasickú techniku **inverzných prvkov**. Pre každý faktoriál si vypočítame aj jeho inverzný prvok a keď týmto faktoriálom potrebujeme deliť, namiesto toho delené číslo vynásobíme inverzným prvkom delenca. Keďže hodnota $p = 10^9 + 7$, ktorú používame na modulovanie je prvočíslo platí, že inverzný prvok čísla a je rovný a^{p-2}.

V prípade, že sa chcete o inverzných prvkoch dozvedieť niečo viac, prečo ich potrebujeme a počítame práve daným spôsobom, navštívte našu kuchárku a prečítajte si článok o Počítaní modulo prvočíslo[7].

Časová a pamäťová zložitosť

Najprv si potrebujeme predpočítať všetky faktoriály a ich inverzy, čo nám zaberie $O(n \log p)$ času ($p = 10^9 + 7$), keďže na vypočítanie inverzu čísla ho musíme umocniť na $p-2$, čo vieme spraviť v logaritmickom čase. Následne musíme vygenerovať všetky možné počty cifier 4, 6, 8 a 9, každej z nich môže byť najviac $O(\log n)$, čo dáva dokopy zložitosť $O(\log^4 n)$. Počty týchto cifier jednoznačne určujú počty zvyšných cifier, ostáva teda výpočet všetkých preusporiadaní vybranej sady cifier. To vieme spraviť v konštantnom čase, keďže faktoriály a ich inverzy už máme predpočítané. Celková časová zložitosť nášho riešenia je $O(n \log p)$.

V pamäti musíme mať predrátané všetky faktoriály, teda pamäťová zložitosť bude $O(n)$

Listing programu (C++)

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

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

typedef long long ll;

int n;
ll res=0, MOD = 1000000007;

vector<int> P;
vector<ll> F, I;

ll umocni(ll a, ll b) {
    ll res1=1;
    while(b>0) {
        if(b%2 == 1) res1=(res1*a)%MOD;
        a=(a*a)%MOD;
        b/=2;
    }
    return res1;
```

[7] https://www.ksp.sk/kucharka/modularna_aritmetika/

```
}

void init() {
    F.push_back(1);
    for(int i=1; i<300047; i++) F.push_back((F[i-1]*i)%MOD);
    For(i, F.size()) I.push_back(umocni(F[i],MOD-2));
}

void rek(int maxi) {
    if(maxi <= 4 && P[2] >= 2) {
        P[2]-=2; P[4]++;
        rek(4);
        P[2]+=2; P[4]--;
    }
    if(maxi <= 6 && P[2] >= 1 && P[3] >= 1) {
        P[2]--; P[3]--; P[6]++;
        rek(6);
        P[2]++; P[3]++; P[6]--;
    }
    if(maxi <= 8 && P[2] >= 3) {
        P[2]-=3; P[8]++;
        rek(8);
        P[2]+=3; P[8]--;
    }
    if(maxi <= 9 && P[3] >= 2) {
        P[3]-=2; P[9]++;
        rek(9);
        P[3]+=2; P[9]--;
    }
    P[1]=n;
    for(int i=2; i<=9; i++) P[1]-=i*P[i];
    if(P[1] < 0) return;
    int suc=0;
    For(i,10) suc+=P[i];
    ll pocet = F[suc];
    For(i,10) {
        pocet = (pocet * I[P[i]]) % MOD;
    }
    res = (res+pocet)%MOD;
}

int main() {
    init();
    scanf(" %d",&n);
    P.clear();
    P.resize(10,0);
    int p=n;
    for(int i=2; i<=9; i++) {
        while(p%i == 0) {
            P[i]++;
            p/=i;
        }
    }
    if(p != 1) {
        printf("0\n");
        return 0;
    }
    res=0;
    rek(0);
    printf("%lld\n",res);
}
```

8. Obchod mení cenu kryptomeny!

buj (bui@ksp.sk)
(max. 12 b za popis, 8 b za program)

V odhadoch časovej zložitosti budeme označovať ako P horný odhad množstva peňazí, ktoré má Jemko pri sebe. (V testovacích prípadoch $P = 50$.)

Hrubá sila

Tak ako v pôvodnej úlohe, aj v tejto vieme použiť štandardný algoritmus na riešenie problému batoha (tzv. *knapsack algoritmus*). O kryptomenách si neudržujeme žiadnu ďalšiu informáciu, iba ich ceny a hodnoty. Zmenu ceny potom vieme vykonať v konštantnom čase.

Listing programu (C++)

```
// knapsack.cpp

#include <vector>
using namespace std;

#define NEVIEM -1

int knapsack(vector<pair<int, int> >& K, int n, int p, vector<vector<int> >& memo) {
  // K := ceny a hodnoty kryptomien
  // n := prvych kolko kryptomien uvazujeme?
  // p := mnozstvo penazi v penazenke
  // memo := struktura, v ktorej si ukladame uz vypocitane hodnoty (memoizacia)
```

```
  // zakladny pripad o-ifujeme
  if (n == 0) {
    return 0;
  }
  // ak este nepozname odpoved, tak pocitame
  if (memo[n][p] == NEVIEM) {
    int naj = knapsack(K, n-1, p, memo); // ak nekupim i-tu kryptomenu
    int p1 = p - K[n-1].first;
    if (p1 >= 0) {
      int kupim = K[n-1].second + knapsack(K, n-1, p1, memo); // ak ju kupim
      if (kupim > naj) {
        naj = kupim;
      }
    }
    memo[n][p] = naj;
  }
  return memo[n][p];
}

int knapsack(vector<pair<int, int> >& K, int p) {
  // pomocna metoda, vytvori nam memo aby sme to nemuseli vytvarat explicitne
  int n = K.size();
  vector<vector<int> > memo(n+1, vector<int>(p+1, NEVIEM));
  return knapsack(K, n, p, memo);
}
```

Zvyšok riešenia vyzerá nasledovne:

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include "knapsack.cpp"
using namespace std;

int main() {
  int n, q;
  cin >> n >> q;

  // nacitame pociatocne data o kryptomenach
  vector<pair<int, int> > K(n); // kryptomeny: first := cena, second := hodnota
  for (int i = 0; i < n; i++) {
    int c, h;
    cin >> c >> h;
    K[i] = make_pair(c, h);
  }

  // spracujeme kazdy z q nasledujucich dni
  for (int qi = 0; qi < q; qi++) {

    // zmena ceny kryptomeny
    int k, b;
    cin >> k >> b;
    k--;
    K[k].first = b;

    // Jemkova navsteva
    int l, r, p;
    cin >> l >> r >> p;
    l--;
    vector<pair<int, int> > podzoznam(K.begin() + l, K.begin() + r);
    int vysl = knapsack(podzoznam, p);
    cout << vysl << "\n";
  }

  return 0;
}
```

Na druhej strane, nájdenie optimálneho nákupu bude pomalé: robíme knapsack na n kryptomenách, a máme vo vrecku najviac P peňazí. Toto potrvá $O(n \cdot P)$, a keďže máme q otázok, celková časová zložitosť bude až $O(n \cdot P \cdot q)$.

Za toto riešenie sa dalo získať 2 body. Veď hrubú silu ste si mohli naprogramovať už v pôvodnej úlohe[8]...

Rozšírenie pôvodnej úlohy? Nie...

Kto ale skúšal riešenie v duchu "upravím vzorák pôvodnej úlohy", zistil, že to nie je také ľahké[9]. Problém robia zmeny hodnôt kryptomien, ktoré nevieme vykonať efektívne. V štruktúrach, ktoré využívame na riešenie pôvodnej úlohy, by sa toho menilo príliš veľa.

Nie všetko je ale stratené—netreba zabudnúť na to, že obmedzenia tejto úlohy sú iné. Možno teda existuje úplne iné riešenie, ktoré nebuduje na riešení pôvodnej úlohy.

Jemko má nejako málo peňazí... náhoda?

Všimnime si, že Jemko má pri sebe vždy najviac 50 peňazí, čo je výrazne menej ako limit v pôdovnej úlohe

[8] https://www.ksp.sk/ulohy/zadania/1479/

[9] Nie je nám známe žiadne riešenie na takéto motívy.

(kde bol 2000). Nevedeli by sme to nejako využiť?

Predstavme si, že Jemko by mal na výber iba z kryptomien ceny 1. Určite sa mu oplatí kupovať od najhodnotnejších kryptomien. Navyše, keďže každá kryptomena stojí 1 peniaz a Jemko má pri sebe nanajvýš P peňazí, kúpi nanajvýš P z týchto kryptomien.

Toto pozorovanie vieme zovšeobecniť: spomedzi kryptomien ceny c stačí Jemkovi pri nákupe uvažovať iba najhodnotnejších $\lfloor \frac{P}{c} \rfloor$ [10] z nich. Viac ich určite nekúpi, lebo potom by minul viac peňazí, ako má pri sebe.

To nám ale výrazne zužuje výber. Aj keby Jemko dovidel na 300 000 kryptomien, stačí mu pri nákupe uvažovať dokopy iba

$$\lfloor \frac{P}{1} \rfloor + \lfloor \frac{P}{2} \rfloor + \ldots + \lfloor \frac{P}{P} \rfloor$$

kryptomien, čo je asymptoticky $O(P \log P)$. Dôkaz tohto odhadu presahuje rámec tohto vzoráku, záujemcom ale odporučím túto stránku[11].

Nemusíme byť ale matematický mágovia na to, aby sme zistili, že onen súčet je dosť malý. Pre $P = 50$ ho vieme zrátať jednoduchým skriptom:[12]

Listing programu (Python)

```python
def sucet(p):
  """Vypocita nasledovny sucet: p/1 + p/2 + ... + p/2, pricom zlomky
  zaokruhli nadol."""
  vysl = 0
  for i in range(1, p+1):
    vysl += p // i
  return vysl

print(sucet(50))
```

Z čoho dostaneme, že sa Jemkovi stačí v našich testovacích prípadoch pozerať vždy na najviac 207 kryptomien. To je výrazne menej, ako keby sa pozeral na všetkých potenciálne až 300 000 kryptomien.

Lepšie riešenie

Ako pomocou vyššie uvedeného tvrdenia možno urýchliť riešenie hrubou silou? Namiesto toho, aby sme pri knapsacku uvažovali všetky kryptomeny, na ktoré Jemko dovidí, stačí uvažovať len niekoľko málo z nich. Konkrétne, každej cenovej kategórie c uvažujeme len $\lfloor \frac{P}{c} \rfloor$ najhodnotnejších kryptomien. Pre každú cenu vieme tento zoznam zostrojiť jednoducho pomocou minimovej haldy: vložíme kryptomenu do haldy, a ak je v halde priveľa prvkov, vyhodíme z nej najmenej hodnotnú kryptomenu.

Listing programu (C++)

```cpp
// vyber.cpp

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

void vyber (int l, int r, int p, vector<pair<int, int> >& K, vector<pair<int, int> >& vysl) {
  /** Vyberie zo zoznamu kryptomien 'K' tie, ktore su relevantne pre otazku:
   * "Ako mam nakupit v intervale [l, r) za p penazi?" Tieto relevantne kryptomeny
   * ulozi do 'vysl'. */
  vector<priority_queue<int, vector<int>, greater<int> > > haldy(p);
  for (int i = l; i < r; i++) {
    int c = K[i].first;    // cena kryptomeny
    if (c > p) {
      continue;
    }
    int h = K[i].second;  // hodnota kryptomeny
    int ci = c - 1;       // index do zoznamu hald
    haldy[ci].push(h);
    if (haldy[ci].size() * c > p) {
      haldy[ci].pop();
    }
  }
  for (int ci = 0; ci < p; ci++) {
    while (!haldy[ci].empty()) {
      int h = haldy[ci].top();
      haldy[ci].pop();
      vysl.push_back(make_pair(ci + 1, h));
    }
  }
}
```

[10] Symbol $\lfloor x \rfloor$ označuje dolnú celú časť reálneho čísla x, teda x zaokrúhlené nadol.
[11] https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)#Integral_test
[12] alebo aj pomocou WolframAlpha[13]

Zlúčením všetkých P háld dostaneme zoznam všetkých kryptomien, na ktoré má zmysel sa pozerať. Potom spustíme knapsack algoritmus, ale iba na kryptomenách v tomto zozname.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include "knapsack.cpp"
#include "vyber.cpp"
using namespace std;

int main() {
  int n, q;
  cin >> n >> q;

  // nacitame pociatocne data o kryptomenach
  vector<pair<int, int> > K(n); // kryptomeny: first := cena, second := hodnota
  for (int i = 0; i < n; i++) {
    int c, h;
    cin >> c >> h;
    K[i] = make_pair(c, h);
  }

  // spracujeme kazdy z q nasledujucich dni
  for (int qi = 0; qi < q; qi++) {

    // zmena ceny kryptomeny
    int k, b;
    cin >> k >> b;
    k--;
    K[k].first = b;

    // Jemkova navsteva
    int l, r, p;
    cin >> l >> r >> p;
    l--;
    vector<pair<int, int> > podzoznam;
    vyber(l, r, p, K, podzoznam);
    int vysl = knapsack(podzoznam, p);
    cout << vysl << "\n";
  }

  return 0;
}
```

Časová zložitosť na jeden nákup sa nám zmenšila na $O(n \log P + P^2 \log P)$: na zostrojenie zoznamu relevantných kryptomien musíme prejsť $O(n)$ kryptomien, a pri každej strávime $O(\log P)$ času na haldu[14]. Následný knapsack trvá $O(P \log P \cdot P)$, pretože máme $O(P \log P)$ vecí a P peňazí.

Toto riešenie si vyslúžilo 6 bodov. Jeho najpomalšou časťou je konštrukcia zoznamu relevantných kryptomien. To robíme jednoducho (ale pomaly) tak, že prejdeme všetky kryptomeny, na ktoré Jemko dovidí.

Vzorové riešenie

V úlohe sa často pýtame na nejaké intervaly, a tak by nás hneď mohol napadnúť intervalový strom. Nevedeli by sme ho tu nejako využiť?

Avšak, priamočiare využitie, kde by sme si v každom intervale pamätali knapsack na tom intervale, nestačí. Problém je so spájaním viacerých podknapsackov do jedného—nejde to robiť efektívne.[15]

Avšak my máme niečo lepšie. Vieme, že nám stačí uvažovať iba niekoľko málo kryptomien z každej ceny. Možno by sme si vedeli pamätať túto informáciu vo vrcholoch stromu.

V každom vrchole si budeme pre každú cenu pamätať zoznam relevantných kryptomien. Navyše budeme mať tieto zoznamy utriedené podľa hodnôt.

Teraz nás zaujímajú dve veci. Po prvé, čo má byť v listoch? To je ľahké: list obsahuje iba jedinú kryptomenu. Nech jej cena je c, potom zoznam pre cenu c obsahuje túto kryptomenu, a ostatný zoznamy sú prázdne.

Ďalej, dajú sa tieto zoznamy efektívne spočítať v nejakom vrchole, ak ich už máme pre jeho synov? Dajú. Postupujeme podobne ako pri *mergesorte*. Pri zostrojovaní zoznamu pre cenu c opakujeme nasledovnú úvahu:

- Ktorý prvok ceny c môže byť najväčší? Zrejme to bude buď najväčší prvok ceny c v ľavom synovi, alebo v pravom synovi. Pre oboch synov ale máme usporiadaný zoznam prvkov s touto cenou. Bude to teda prvý prvok niektorého z týchto zoznamov.
- Porovnáme tieto prvé prvky, zoberieme väčší z nich a umiestnime ho na prvé voľné miesto v našom zozname.

[14] Šikovné oko si všimne, že v skutočnosti zoznam vieme zostrojiť rýchlejšie, ako pomocou haldy.

[15] Možno argumentujete, že v pôvodnej úlohe sme predsa použili intervalový strom. Zamyslite sa ale, či nebol nejaký divný. Napríklad, keď sme dostali nejaký interval, na koľko najviac vrcholov v strome sme ho rozdelili? Bolo to $O(\log n)$, tak ako v obyčajných intervaláčoch?

- Opakujeme so zvyškom, pričom vybratý prvok ďalej neberieme v úvahu. Za najväčší prvok v tom synovi teda budeme považovať druhý prvok, keď sa vyberie ten tak potom tretí, …

Listing programu (C++)

```
// zluc.cpp

#pragma once // aby sa to pri kompilacii include-lo len raz

#include <vector>
using namespace std;

#define MAX_P 50
#define ZLA_HODNOTA -1

struct Vyber {
  // Struktura odrzujuca najlepsich 50/50 kariet ceny 50,
  // najlepsich 50/49 kariet ceny 49,
  // ...,
  // najlepsich 50/2 kariet ceny 2,
  // najlepsich 50 kariet ceny 1.

  vector<vector<int> > V; // V[i] obsahuje zoznam pre cenu <i>, V[0] sa nepouziva

  Vyber() : V(MAX_P + 1) {}

  Vyber(int c, int h) : V(MAX_P + 1) {
    // zakladny Vyber, obsahuje len 1 kartu s cenou <c> a hodnotou <h>.
    V[c].push_back(h);
  }

  Vyber(Vyber& a, Vyber& b) : V(MAX_P + 1) {
    // zluci <a> a <b> do jedneho
    for (int ci = 1; ci <= MAX_P; ci++) {
      int ai = 0, bi = 0;

      // zoberieme najhodnotnejsich MAX_P/ci z <a> a z <b> (s cenou <ci>)
      for (int j = 0; j < MAX_P/ci; j++) {
        int ah = (ai < (int)a.V[ci].size() ? a.V[ci][ai] : ZLA_HODNOTA);
        int bh = (bi < (int)b.V[ci].size() ? b.V[ci][bi] : ZLA_HODNOTA);

        // ale ak uz nie su ani v <a> ani v <b>, tak skonci
        if (ah == ZLA_HODNOTA && bh == ZLA_HODNOTA) {
          break;
        }

        int h;
        if (ah > bh) {
          h = ah;
          ai++;
        }
        else {
          h = bh;
          bi++;
        }
        V[ci].push_back(h);
      }
    }
  }
};

Vyber zluc(vector<Vyber>& A) {
  // Zluci vsetky Vybery vo <V> do jedneho, v ktorom su z kazdej ceny najhodnotnejsie meny.
  // Predpokladame, ze sme dostali aspon 1 Vyber vo <V>...
  if (A.size() == 1) {
    return A[0];
  }
  Vyber vysl(A[0], A[1]);
  for (int i = 2; i < (int)A.size(); i++) {
    vysl = Vyber(vysl, A[i]);
  }
  return vysl;
}

void do_pola(Vyber& a, vector<pair<int, int> >& vysl) {
  // Do vektora <vysl> ulozi obsah Vyberu <v> (vo forme dvojic <cena kryptomeny, jej hodnota>).
  // Na toto pole potom staci zavolat knapsack funkciu.
  vysl.clear();
  for (int ci = 1; ci <= MAX_P; ci++) {
    for (auto h : a.V[ci]) {
      vysl.push_back({ci, h});
    }
  }
}
```

Aká je časová zložitosť tohto predpočítania? Zlučovanie dvoch intervalov do jedného trvá $O(P \log P)$, a vrcholov v strome je $O(n)$. Bude teda trvať $O(n \cdot P \log P)$.

Listing programu (C++)

```
// intervalac.cpp

#pragma once // aby sa to pri kompilacii include-lo len raz
```

```
#include <vector>
#include "zluc.cpp"
using namespace std;

struct Intervalac {
  vector<Vyber> A;
  int m;

  Intervalac(vector<pair<int, int> >& V) {
    // inicializujeme intervalac: kryptomeny s povodnymi cenami a hodnotami <V>
    int n = V.size();
    m = 1;
    while (m < n) m *= 2;

    A.resize(2*m);
    // A[0] sa nepouziva
    // A[1...m-1] su intervaly dlzok 2 a viac
    // A[m...2m-1] su intervaly dlzky 1
    for (int i = 0; i < n; i++) {
      A[m+i] = Vyber(V[i].first, V[i].second);
    }
    for (int i = m-1; i > 0; i--) {
      A[i] = Vyber(A[2*i], A[2*i+1]);
    }
  }

  void zmen(int k, int b, int h) {
    // zmen cenu <k>-tej kryptomeny na <b> a hodnotu na <h>, pricom <k> cislujeme od 0
    int kde = m+k;
    A[kde] = Vyber(b, h);
    kde /= 2;
    while (kde > 0) {
      A[kde] = Vyber(A[2*kde], A[2*kde+1]);
      kde /= 2;
    }
  }

  int L, R;
  void pomocny_rozloz(int l, int r, int i, vector<Vyber>& vysl) {
    // Rekurzivne rozlozi interval <l; r) na podintervaly a ich Vybery
    // prida do <vysl>. <l>, <r> a <i> su viazane: posledne menovane
    // udava, v ktorom vrchole intervalacu sme, a prve dve definuju
    // interval. Iba intervaly, ktore zodpovedaju nejakemu vrcholu,
    // su povolene.
    if (l >= L && r <= R) {
      vysl.push_back(A[i]);
      return;
    }
    if (l >= R || r <= L) {
      return;
    }
    int s = (l+r)/2;
    pomocny_rozloz(l, s, 2*i, vysl);
    pomocny_rozloz(s, r, 2*i+1, vysl);
  }

  void rozloz(int l, int r, vector<Vyber>& vysl) {
    // Rozlozi interval <l; r) na podintervaly, a ich Vybery da do <vysl>.
    // Z toho potom vieme vycitat, ktore kryptomeny v useku <l; r) su tie
    // podstatne (dostatocne hodnotne).
    L = l;
    R = r;
    pomocny_rozloz(0, m, 1, vysl);
  }
};
```

Ako zistiť zoznam relevantných kryptomien pri nákupe? Interval rozbijeme na $O(\log n)$ intervalov zo stromu, a tie zlúčime. To potrvá $O(\log n \cdot P \log P)$, a následné vyriešenie knapsacku bude trvať $O(P \cdot P \log P)$. Časová zložitosť jedného nákupu je teda $O(P \log P \cdot (P + \log n))$.

Ako prepočítať intervalový strom, keď sa zmení hodnota kryptomeny? Keď vieme zlučovať intervaly, nie je problém: zmeníme príslušný vrchol-list[16] a všetkých jeho predkov [17] prepočítame. Tých je iba $O(\log n)$, časová zložitosť jednej úpravy je teda $O(\log n \cdot P \log P)$.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include "knapsack.cpp"
#include "zluc.cpp"
#include "intervalac.cpp"
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);
  cout.tie(0);

  int n, q;
  cin >> n >> q;
```

[16] reprezentujúci interval dĺžky 1

[17] reprezentujúce všetky ďalšie intervaly, ktoré kryptomenu obsahujú

```
  // nacitame pociatocne data o kryptomenach
  vector<pair<int, int> > K(n); // kryptomeny: first := cena, second := hodnota
  for (int i = 0; i < n; i++) {
    int c, h;
    cin >> c >> h;
    K[i] = make_pair(c, h);
  }
  Intervalac I(K);

  // spracujeme kazdy z q nasledujucich dni
  for (int qi = 0; qi < q; qi++) {

    // zmena ceny kryptomeny
    int k, b;
    cin >> k >> b;
    k--;
    if (b != K[k].first) {
      K[k].first = b;
      I.zmen(k, b, K[k].second);
    }

    // Jemkova navsteva
    int l, r, p;
    cin >> l >> r >> p;
    l--;

    // najdeme iba tie podstatne kryptomeny
    vector<Vyber> vybery;
    I.rozloz(l, r, vybery);
    Vyber podvyber = zluc(vybery);
    vector<pair<int, int> > podzoznam;
    do_pola(podvyber, podzoznam);

    int vysl = knapsack(podzoznam, p);
    cout << vysl << "\n";
  }

  return 0;
}
```

Celková časová zložitosť je

$$O(P \log P \cdot (n + (P + \log n) \cdot q)),$$

čo síce nevyzerá pekne, ale aspoň je to dosť málo. Toto riešenie si vyslúžilo plných 8 bodov.

Pomocou *lazy-loadingu* by sme sa vedeli vyhnúť zdĺhavému predpočítaniu a dosiahnuť tak o chlp lepšiu časovú zložitosť. Na plný počet bodov sme to ale nevyžadovali.