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

Vzorové riešenia 2. kola zimnej časti

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

1. Zwarte Doos

Level 1

Na rozcvičku si Doos pripravila opäť niečo jednoduché. Vaše n zmenila na $2n+3$. Na vyriešenie stačilo zadať $\frac{47-3}{2} = 22$.

Level 2

Ak ste išli postupne od nuly, nebolo ťažké si všimnúť, že Doos vždy pripočíta vstup k predchádzajúcemu výsledku. Vracala teda súčet prvých n čísel. 11628 je súčtom prvých 152 čísel, čo ste mohli zistiť výpočtom alebo binárnym vyhľadávaním. (Ak neviete, čo je binárne vyhľadávanie, pozrite si vzorové riešenie k levelu 4 z predchádzajúceho kola.)

Level 3

Začiatok mohol byť trochu mätúci, ale väčšie čísla odhalili, že sa jedná o ciferný súčin vstupu. Chceme získať číslo 529200, ktoré má nasledovný rozklad na prvočísla – $2^4 \cdot 3^3 \cdot 5^2 \cdot 7^2$. To znamená, že číslo 22223335577 by malo dať správne riešenie. Je však pridlhé a Doos ho neakceptuje. Potrebujeme teda zoskupiť niektoré cifry dokopy. Zafungovalo napríklad číslo 8695577.

Level 4

Ak ste chápali výstup ako súradnice nejakého bodu, nebolo ťažké si všimnúť, že sa pomocou nich postupne kreslí štvorcová špirála. Štvorcová špirála rozmerov 5×5 vyzerá takto (číslo x predstavuje x-tý bod špirály):

```
16 15 14 13 12 29
17  4  3  2 11 28
18  5  0  1 10 27
19  6  7  8  9 26
20 21 22 23 24 25
```

Bod $(37, 73)$ bude na obvode nejakého štvorca so stranou dlhou zhruba $73 \cdot 2 = 146$, teda pred ním sa vyskytlo približne $146^2 = 21316$ iných bodov. Týmto postupom nedostaneme presný výsledok, ale súradnicu $(-73, 73)$. Takže sa musíme vrátiť o 110 bodov späť. 21206 je teda správny výsledok.

Samozrejme, bolo možné zrátať správny výsledok aj presne (a nezaškodí vám si to vyskúšať), ale uvedený postup je rýchlejší.

Level 5

Ak ste skúšali náhodné čísla, dostali ste náhodné slová. Ak ste však šli postupne, slová začali tvoriť vety. Tá prvá znela *Niet domu kde sa nedymí v komíne*. A múdry Google vám hneď poradí, že je to nadpis prvej kapitoly diela Dom v Stráni od Martina Kukučína. Elektronická verzia knihy je voľne prístupná na internete, takže vám už len stačí zistiť, koľké v poradí je v nej slovo *drúže*.

Toto slovo sa v texte nachádzalo práve raz, stačilo si ho nechať vyhľadať. Potom si stačí skopírovať celý text pred ním a použiť nejaký nástroj na spočítanie slov (online alebo textový editor). Dostaneme počet slov pred slovom *drúže* a teda aj jeho poradie.

Keďže rôzne počítadlá slov majú rôzne názory na to, či sa napríklad slová oddelené pomlčkou počítajú za jedno alebo za dve, nemuseli ste sa na prvý pokus trafiť úplne presne. Z kontextu však viete zistiť, kde v texte sa nachádzate a o koľko ste sa pomýlili. Takto sa určite nakoniec dopracujete k správnej pozícii 93226.

Nakoniec by som ešte rád spomenul, že poznámky pod čiarou sa do textu, samozrejme, nepočítali.

Level 6

Zo začiatku ste si iste všimli, že odpovede postupne stúpali a blížili sa k vysnívanej 1. Ak ste ju však preskočili, začali zákerne klesať.

Nie príliš komplikované riešenie bolo postupne odhaľovať dĺžku a cifry správneho výsledku. Stačilo sa vždy opýtať na dve po sebe idúce čísla a podľa toho, či postupnosť stúpa alebo klesá, ste hneď zistili, či ste príliš vpredu alebo príliš vzadu.

Ak by ste mali záujem o rýchlejší spôsob, opäť pomôže binárne vyhľadávanie (kde sa tiež pýtate na 2 po sebe idúce čísla).

Existuje však ešte efektívnejší spôsob ako v postupnosti, ktorá najskôr stúpa a potom klesá, nájsť maximum. Hovorí sa mu ternárne vyhľadávanie. Podobne ako pri binárnom, aj tu si udržujete interval, v ktorom by mohlo byť maximum. V každom kroku ho však rozdelíte na 3 časti, porovnáte hraničné prvky medzi nimi, a vylúčite tú z krajných častí, ktorá je pri nižšom hraničnom prvku. Ak sú oba prvky rovnaké, vylúčite obe krajné časti. Takto pokračujete, až kým nemáte dostatočne malý interval (napríklad dĺžky 5), v ktorom maximum nájdete vyskúšaním všetkých možností. Skúste si rozmyslieť, prečo to funguje.

Správny výsledok bol 7182818. Nie je vám povedomý?

Level 7

Tento level bol jeden z najťažších, lebo bolo ťažké zistiť, čo vlastne Doos vracia. Všimnúť ste si mohli, že rovnaké cifry prispievajú do celkového súčtu rovnakou hodnotou. A tou je počet paličiek, ktoré treba rozsvieť na klasickom 7 segmentovom digitálnom displeji, aby sme cifru zobrazili.

Na získanie čísla 42, teda môžeme použiť napríklad 888888.

Level 8

Prvá vec, ktorú ste si mohli všimnúť, bola, že pre prvočísla vracia Doos vždy číslo 2. To celkom naznačuje spojitosť s deliteľmi daného čísla – Doos totiž vracia počet deliteľov vstupu.

Ale ktoré číslo má 728 deliteľov? $728 = 2^3 \cdot 7 \cdot 13$. Deliteľa nejakého čísla vytvoríte tak, že z jeho prvočíselného rozkladu niečo poškrtáte. Ak bude v rozklade 2^{12}, môžete poškrtať od 0 po 12 dvojok, čo je 13 možností. Ak pridáme 3^6 vieme nezávisle na tom škrtnúť trojky až 7 možnosťami. Nakoniec do rozkladu na prvočísla pridáme aj $5 \cdot 7 \cdot 11$ z ktorých môžeme každé buď škrtnúť, alebo neškrtnúť, čo je 8 možností.

Stačilo teda zadať hodnotu $2^{12} \cdot 3^6 \cdot 5 \cdot 7 \cdot 11 = 1149603840$.

Level 9

Nepárne čísla zostali sami sebou. 2, 6, 10 sa zdvojnásobili, 4 a 12 zoštvornásobili a 8 dokonca zosemnásobila. Každé číslo sa teda vynásobilo najväčšou mocninou dvojky, ktorá ho delila.

Keďže $1263872 = 4937 \cdot 2^8 = 4937 \cdot 2^4 \cdot 2^4$, správny vstup bude $4937 \cdot 2^4 = 78992$. Takto bude totiž 2^4 najväčšia mocnina dvojky, ktorá ho delí, a teda sa ňou ešte prenásobí.

Level 10

Na začiatku ste zrejme mali zopár krátkych pokusov, *7 SPRÁVNE, 0 NA ZLEJ POZÍCII*. To vám hádam našepkalo, že správny počet cifier je 7.

Zostáva už iba uhádnuť, ktoré cifry to sú. Zakaždým ste sa dozvedeli, koľko cifier ste trafili a koľko ste uviedli správne, avšak na nesprávnej pozícii. Presne ako v staršej stolovej hre Logik.

Bežne sa hra rieši tak, že najskôr spravíte zopár náhodných tipov a potom sa z nich snažíte čo najviac dedukovať. Ak sa vám do niečoho takého nechcelo, mohli ste najskôr zistiť, koľko ktorých cifier sa vo výsledku nachádza (skúšaním siedmych rovankých cifier) a potom zistením, na ktorú pozíciu, ktorá cifra patrí (umiestnením tejto cifry medzi šesť deviatok, ktoré sa vo výsledku nenachádzali). V oboch prípadoch by ste sa však mali dopracovať k číslu 2637033.

2. Zobudená Programátorka

vzorák napísal Zygro
(max. 6 b za popis, 4 b za program)

Úloha sa dá riešiť dvoma spôbmi, buď vrecúška prechádzame od najmenších po najväčšie alebo naopak. Formát výstupu (tam vypisujeme čísla od najmenších) nám napovedá, ktorý spôsob bude asi jednoduchší.

Celý algoritmus je veľmi jednoduchý – ak je celková hmotnosť nepárna, musíme použiť vrecúško s hmotnosťou 1, ak je hmotnosť párna tak vrecúško použiť nesmieme. Po prípadnom odčítaní najmenšieho vrecúška nám ostalo párne číslo a tiež hmotnosti všetkých ostatných vrecúšok sú párne. Preto môžeme imaginárne zmenšiť všetky hmotnosti na polovicu a zrazu riešime tú istú úlohu len s menšími číslami.

Algoritmus bude teda opakovane kontrolovať paritu čísla zo vstupu. Keď je číslo v i-tom kole nepárne, vypíše hmotnosť i-teho vrecúška a po každej kontrole vydelí číslo zo vstupu dvoma.

Koľkokrát môžeme vydeliť číslo n dvoma, kým dostaneme jednotku? Správna odpoveď je $\log_2(n)$ (dvojkový logaritmus z n), pretože $\log_a(a^x) = x$[1]. Preto algoritmus skončí po $O(\log n)$ krokoch. Časová zložitosť je teda $O(\log n)$ a pamäťová $O(1)$, pretože si stačí pamätať zopár čísel.

Všimnime si, že každé prirodzené číslo sa dá zapísať ako súčet rôznych mocnín dvojek. Tento rozklad je skoro to isté ako prevod čísla do dvojkovej (binárej) sústavy.

Listing programu (Pascal)

```
var hmotnost,mocnina : int64;
    treba_medzeru : boolean;
begin
    treba_medzeru := false;
    mocnina := 1;
    readln(hmotnost);
    while hmotnost > 0 do begin
        if hmotnost mod 2 = 1 then begin
            if treba_medzeru then write('␣')
            else treba_medzeru := true;
            write(mocnina);
        end;
        hmotnost := hmotnost div 2;
        mocnina := mocnina * 2;
    end;
    writeln;
end.
```

Listing programu (C++)

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

int main(){
    bool treba_medzeru = false;
    long long mocnina = 1;
    long long hmotnost;
    cin >> hmotnost;

    while(hmotnost > 0) {
        if (hmotnost % 2) {
            if (treba_medzeru) cout << "␣";
            else treba_medzeru = true;
            cout << mocnina;
        }
        hmotnost /= 2;
        mocnina *= 2;
    }

    cout << endl;
}
```

Listing programu (Python)

```
n = int(input())
x = 1
ans = [] # pre jednoduchší kód sa uspokojíme s horšou pamäťovou zložitosťou
while n > 0:
    if n%2: ans.append(str(x))
    x *= 2
    n //= 2
print('␣'.join(ans))
```

3. Zved Tigrík

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

Na začiatok jednoduchší prípad

Uvažujme na začiatok jednoduchšiu úlohu: čo ak by k bolo rovné n? Vtedy vieme, že Tigrík stihne navštíviť všetky svoje zdroje a treba len spočítať, koľko za tento deň zarobí.

Na to potrebujeme pre každý typ informácie zistiť, koľko zdrojov ju má. Inými slovami, potrebujeme spočítať počet výskytov každého písmena. Toto najľahšie spravíme tak, že pre každé písmeno budeme mať jedno počítadlo. Najlepšie je uložiť si tieto počítadlá do poľa veľkosti 26 – toľko je rôznych písmen v anglickej abecede.

[1] Logaritmus n pri základe a zapisovaný ako $\log_a(n)$ má nasledovný význam: číslo, na ktoré musíme umocniť a, aby sme dostali hodnotu n. Preto $\log_2(8) = 3$ a $\log_{10}(100) = 2$. No a to, čo používame, je vzťah $a^{\log_a(n)} = n$. Inými slovami, ak umocníme číslo a na **číslo, na ktoré musíme umocniť a aby sme dostali n**, dostaneme n.

Keď to už máme spočítané, tak si stačí uvedomiť, že ak máme x informácií rovnakého typu (teda x výskytov konkrétneho písmena), tak každé z nich má cenu x, a dokopy za toto písmeno dostaneme $x \times x = x^2$ peňazí.

Takže ak sme si spočítali počty výskytov jednotlivých písmen, stačí nám sčítať ich druhé mocniny a máme Tigríkov zárobok.

Listing programu (Pascal)

```
var n,k,sucet,i:int64;
        c : char;
        a : array[1..26] of int64;
begin
        readln(n,k);
        for i:=1 to 26 do a[i] := 0;
        for i:=1 to n do begin
                read(c);
                inc(a[ord(c)-ord('A')+1]);
        end;
        sucet := 0;
        for i:=1 to 26 do begin
                sucet := sucet + a[i]*a[i];
        end;
        writeln(sucet);
end.
```

Pôvodný problém

Teraz sa vráťme k všeobecnému zadaniu, kedy môže byť $k < n$. Vtedy musíme zistiť, ktoré zdroje sa Tigríkovi oplatí navštíviť – inými slovami, ktorých k písmen máme zobrať, aby sme dokopy dostali čo najväčší zárobok.

Myslím, že každý z vás si všimol, že čím viac máme rovnakých písmen, tým viac Tigrík zarobí. Intuícia nám teda napovedá, že Tigrík by mal preferovať tie písmená, ktorých je na vstupe veľa.

Nádejne teda vyzerá nasledovný postup: Tigrík si nájde to písmeno, ktorého je na vstupe najviac, a zoberie čo najviac jeho výskytov. Potom spraví to isté s druhým najčastejším, tretím najčastejším, a tak ďalej, až kým už nemôže zobrať žiadne ďalšie písmeno.

Zamyslime sa najskôr, ako by sme takéto riešenie implementovali. Rovnako ako v prvej časti začneme tým, že si spočítame počty výskytov jednotlivých písmen. Teraz by sme ich potrebovali usporiadať podľa počtu výskytov. Keďže máme len 26 rôznych písmen, je úplne jedno, aký algoritmus na to použijeme. Môžeme použiť štandardné knižničné triedenie (napr. `sort()` v C++) alebo implementovať nejaké vlastné.

Po usporiadaní počtov výskytov už odpoveď vypočítame jednoduchým cyklom (všimnite si, že sa musíme zakaždým pozrieť, či môžeme zobrať všetky písmená daného typu, alebo ich je priveľa a len naplníme našu kapacitu):

Listing programu (Pascal)

```
{ pred týmto kusom programu usporiadame pole 'a' od najväčšieho po najmenšie }
sucet := 0;
i := 1;
while (k > 0) do begin
        if (k < a[i]) then kolko := k
        else kolko := a[i];
        sucet := sucet + kolko*kolko;
        k := k - kolko;
        inc(i);
end;
writeln(sucet);
```

Asi najjednoduchšie na implementáciu je triedenie max-sort. Priamo počas neho vieme aj počítať riešenie. Jednoducho postupne 26-krát zopakujeme nasledovný postup: "Nájdi najčastejšie sa vyskytujúce ešte nespracované písmeno a zober z neho najviac ako sa dá."

Dôkaz správnosti

Teraz nás ale čaká tá ťažšia časť tohto vzorového riešenia. Je síce pekné, že sme si vymysleli jednoduché *pažravé*[2] riešenie, ktoré sa nám ľahko implementovalo, čo nám ale zaručí, že toto riešenie je naozaj optimálne? Nemôže sa niekedy stať, že by iný spôsob výberu písmen viedol k väčšiemu celkovému zárobku?

Aby bolo naše riešenie úplné, musíme na tieto otázky vedieť odpovedať. Inými slovami, potrebujeme ešte dokázať správnosť nášho algoritmu.

Pomocné tvrdenie

Dokážeme si nasledovné tvrdenie: *"Nič nepokazíme, keď z najčastejšieho písmena zoberieme najviac kusov, ktoré môžeme zobrať."* Inými slovami, tvrdíme, že pre ľubovoľný vstup existuje medzi optimálnymi riešeniami

[2] Pažravé sa nazýva preto, lebo používa nejakú podmienku, ktorá je založená na maximalite (minimalite) nejakého prvku.

také riešenie, v ktorom Tigrík zoberie najväčší možný počet výskytov najčastejšieho písmena.

Dôkaz: Nech v je počet výskytov najčastejšieho písmena a nech z je celkový počet písmen, ktoré ešte môžeme zobrať. Dôkaz nášho tvrdenia si rozdelíme na dva prípady.

Prípad prvý: $v > z$, teda nemôžeme zobrať všetky výskyty najčastejšieho písmena.
V ľubovoľnom riešení by sme zobrali nejakých z písmen. Keďže každého z nich by sme zobrali najviac z kusov, bol by celkový zisk najviac z^2. Viac ako z^2 sa zjavne dosiahnuť nedá. No a pri našej stratégii zoberieme z kusov najčastejšieho písmena a dostaneme tak zisk presne z^2, čo je zjavne optimálne.

Prípad druhý: $v \leq z$, teda môžeme zobrať všetky výskyty nášho písmena a možno ešte aj niečo navyše.
Chceme teraz dokázať, že existuje optimálne riešenie, v ktorom Tigrík vezme všetkých v kusov tohto písmena. Toto dokážeme tak, že ukážeme, že k ľubovoľnému riešeniu, v ktorom zoberie *menej ako* v kusov, existuje *aspoň tak isto dobré* riešenie, v ktorom ich vezme *viac*.
Uvažujme teda ľubovoľné riešenie, v ktorom Tigrík nezobral v kusov nášho písmena, ale len w, pričom $w < v$.

- Podprípad 2a: Nejakého iného písmena zobral Tigrík $x > w$ kusov.

  V tomto prípade vieme dostať *rovnako dobré* riešenie tak, že zoberieme x kusov nášho a w kusov toho druhého písmena. (Touto zmenou sme *zväčšili* počet zobratých kusov nášho písmena.)

- Podprípad 2b: Každého iného písmena zobral Tigrík w alebo menej kusov.

  Za každé písmeno teraz Tigrík dostane w alebo menej bodov. V tejto situácii vieme vyrobiť *ostro lepšie* riešenie tak, že zahodíme jedno iné písmeno a namiesto neho zoberieme jedno ďalšie naše. Ak tých iných písmen bolo doteraz x (pričom $x \leq w$), tak zisk klesol o $2x - 1$ za odstránené písmeno, ale stúpol o $2w + 1$ za písmeno pridané, a teda v súčte aspoň o 2 stúpol. Oplatí sa nám teda vymieňať písmená za tie, ktoré sa tam vyskytujú častejšie.

A tým je dôkaz hotový.

Záver dôkazu správnosti

Z práve dokázaného pomocného tvrdenia už priamočiaro vyplýva dôkaz správnosti nášho algoritmu. Na začiatku vieme, že existuje optimálne riešenie, v ktorom zoberieme najväčší možný počet výskytov najčastejšieho písmena, tak to urobíme. Na toto písmeno môžeme teraz šťastne zabudnúť. Ostala nám opäť taká istá situácia: máme na výber nejaké písmená a máme nejakú voľnú kapacitu. Teraz môžeme znovu použiť tú istú argumentáciu – z toho písmena, ktoré je *teraz* najčastejšie, určite môžeme zobrať najviac ako sa dá. A tak ďalej.

Zložitosti

Načítanie vstupu má časovú zložitosť lineárnu od n, čo zapisujeme $O(n)$. Počas toho vieme aj postupne zvyšovať počítadlá písmen. Celý zvyšok riešenia už má konštantnú časovú zložitosť – rôznych písmen je len 26, a teda na ich usporiadanie podľa počtu výskytov aj na ich následné spracovanie nám stačí konštantný počet krokov, nezávislý od veľkosti vstupu. Výsledná časová zložitosť je teda $O(n)$.

Pamäťová zložitosť je $O(1)$, teda konštantná. Vôbec si totiž nemusíme držať v pamäti celý vstup, stačí si pamätať iba 26 počítadiel a nejaké pomocné premenné.

Listing programu (Pascal)

```
var n,k,sucet,i,j,kolko,pozMax,pomocna:int64;
        c : char;
        a : array[1..26] of int64;
begin
        readln(n,k);
        for i:=1 to 26 do a[i] := 0;
        for i:=1 to n do begin
                read(c);
                inc(a[ord(c)-ord('A')+1]);
        end;

        //triedenie max-sort, najde maximum a da ho na zaciatok, potom najde
        //maximum zo zvysnych a da ho na druhe miesto ...
        for j := 1 to 25 do begin
                pozMax := j;
            for i := j+1 to 26 do begin
                        if a[i] > a[pozMax] then pozMax := i;
            end;
            if pozMax <> j then begin
                        pomocna := a[pozMax];
                        a[pozMax] := a[j];
                        a[j] := pomocna;
            end;
```

```
        end;

        sucet := 0;
        i := 1;
        while (k > 0) do begin
                if (k < a[i]) then kolko := k
                else kolko := a[i];
                sucet := sucet + kolko*kolko;
                k := k - kolko;
                inc(i);
        end;
        writeln(sucet);
end.
```

vzorák napísal mišof

4. Záhradka

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

Keďže v každom rohu obvod Kleofášovej záhradky zatáča o 90 stupňov, musia sa na obvode striedať vodorovné a zvislé strany. Vodorovných strán je teda presne toľko isto ako zvislých. Inými slovami, počet strán záhradky musí byť párny. Pre všetky vstupy s nepárnym s teda môžeme smelo odpovedať, že nemajú riešenie.

Následne ošetríme okrajové prípady: $t = 1$ aj $t = 2$ sú možné len ak $s = 4$. (V prvom prípade tvorí záhradku jeden štvorec, v druhom prípade sú to nutne dva stranou susediace štvorce. Tvar obvodu je v oboch prípadoch jednoznačne určený.) Odteraz teda môžeme predpokladať, že $t \geq 3$.

Ako ďalší krok nášho riešenia sa pozrieme na najmenší povolený prípad: situáciu kedy $s = 2t$. Chceme teda nakresliť útvar s $2t$ stranami a obsahom t.

Ako to bude vyzerať pre $t = 3$? Ak má mať záhradka tri štvorce, môžu byť buď v rade za sebou (vtedy má ale obvod len štyri strany), alebo môžu byť zatočené "do L" (kedy je strán 6, čo je presne to, čo sme chceli).

```
+---+
|   |
|   +---+   <-- 6 strán, obsah 3
|       |
+-------+
```

Ďalej máme $t = 4$. Tu chceme útvar, ktorý bude mať 8 strán. Ideálne by bolo neriešiť úlohu odznova, ale skúsiť upraviť predchádzajúce riešenie – pridať nový štvorec tak, aby nám tým pribudli dve strany. A skutočne to ide spraviť. Jedna dobrá možnosť vyzerá nasledovne:

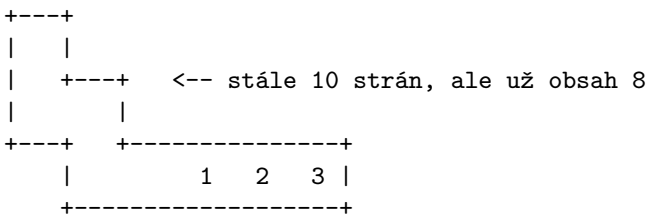
```
+---+
|   |
|   +---+
|       |   <-- 8 strán, obsah 4
+---+   +
    |   |
    +---+
```

No a už by malo byť aj jasné, že tento postup vieme ďalej "do nekonečna" opakovať. Pre názornosť sa ešte pozrime na $t = 5$. Tam opäť pridáme jeden štvorec "na chvost hada", čím sa nám obsah zväčší o 1 a počet strán o 2.

```
+---+
|   |
|   +---+   <-- 10 strán, obsah 5
|       |
+---+   +---+
    |       |
    +-------+
```

Takže už vieme riešiť všetky prípady kedy $t \geq 3$ a $s = 2t$: jednoducho vyrobíme t-políčkového hada. Čo teraz so všeobecným prípadom? Predpokladajme, že s je párne a že $t = s/2 + x$. Máme teda mať ten istý počet strán, ale o x väčší obsah. Ako to dosiahnuť? Jednoducho – stačí napríklad nášmu hadovi "natiahnuť chvost".

Riešenie si opäť ukážeme na jednom príklade, z ktorého bude jasné, ako to funguje vo všeobecnosti. Majme opäť $s = 10$, ale tentokrát namiesto $t = 5$ budeme mať $t = 8$, teda o tri viac. Spravíme teda nášmu hadovi o tri dlhší chvost:



Tu je implementácia tejto myšlienky:

Listing programu (Python)

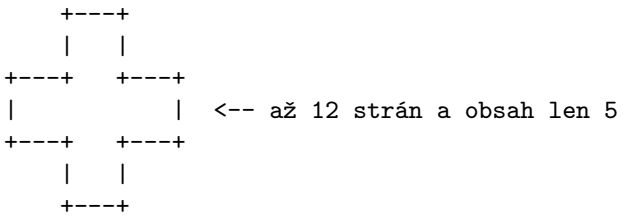
```
import sys

S, O = [ int(x) for x in input().split() ]

if S%2 != 0 or S == 2:
    print("Neda sa")
    sys.exit(0)
if S == 4:
    print("S", 1)
    print("V", O)
    print("J", 1)
    print("Z", O)
else:
    chvost = O - S//2
    inner = S - 6
    Sd = inner

    print("Z", chvost+1)
    print("S", 1)
    print("V", chvost+2)
    dole = True
    while Sd > inner//2:
        print("J" if dole else "V", 1); Sd -= 1
        dole = not dole
    print("J" if dole else "V", 2 if dole else 1)
    print("Z" if dole else "J", 1)
    print("S" if dole else "Z", 1 if dole else 2)
    while Sd > 0:
        print("Z" if dole else "S", 1); Sd -= 1
        dole = not dole
```

Ako vyzerajú nejaké situácie s malým obsahom?

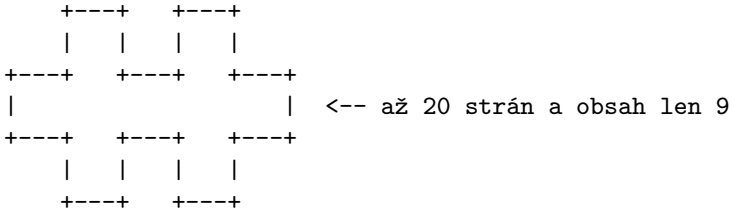
Okrem naprogramovania vyššie popísaného riešenia ste mali ešte jednu úlohu: ukázať, že existuje nekonečne veľa dvojíc (s, t), pre ktoré $t < s/2$ a napriek tomu hľadaná záhradka existuje. Teraz sa pozrieme na to, ako takéto prípady vyzerajú.

Jeden už vlastne poznáme: $t = 1$ a $s = 4$, teda jediný štvorec. To je však patologický prípad, ktorý nevieme zovšeobecniť.

Po troche skúšania však ľahko odhalíme druhý najmenší prípad s obsahom menším ako $s/2$. Vyzerá nasledovne:



A odtiaľ už ľahko pokračujeme k ďalším, väčším záhradkám s touto vlastnosťou. Tu je nasledujúca:



a už je asi jasné, ako to bude pokračovať ďalej. Tým sme splnili zadanú úlohu – našli sme nekonečne veľa rôznych záhradiek, ktorých obsah t je menší ako polovica počtu ich strán.

Hlbšie zamyslenie na záver

Na záver tohto vzorového riešenia si ukážeme dôslednejšiu úvahu, ktorá nás privedie k tomu, ako musia vyzerať záhradky s veľkým počtom strán a malým obsahom.

Začneme tým, že si predstavíme ľubovoľnú záhradku s obsahom t. Namiesto počtu strán obvodu sa ale sústreďme na jeho dĺžku (pričom strana štvorca má dĺžku 1). Aký najdlhší obvod vieme dosiahnuť?

Máme t štvorcov a každý z nich má 4 strany, čiže určite to nebude viac ako $4t$. V skutočnosti to ale musí byť ešte výrazne menej. Celá záhradka musí držať pokope, takže niektoré dvojice štvorcov musia susediť. No a vždy, keď dáme dva štvorce jeden vedľa druhého, zmenšíme tým obvod o 2 – ani z jedného ani z druhého už nie je na obvode ich spoločná strana. (Napr. dva izolované štvorce majú celkový obvod dĺžky 8, zatiaľ čo obdĺžnik 1×2 má obvod dĺžky 6.)

Koľko dvojíc susediacich štvorcov musí byť v záhradke tvorenej t štvorcami? Ľahko nahliadneme, že aspoň $t - 1$. (Predstavme si, že záhradku ideme pozliepať dokopy z t izolovaných štvorcov. Každým zlepením pozdĺž nejakej strany zmenšíme počet kusov o 1. Aby teda celá záhradka držala pokope, musíme lepiť aspoň $t - 1$ ráz.)

Najväčší možný obvod záhradky je teda $4t - 2(t - 1) = 2t + 2$. (Tento obvod majú všetky záhradky, ktoré majú stromovú topológiu.)

No a počet strán záhradky je zjavne nanajvýš rovný jej obvodu – pričom rovnosť nastáva práve vtedy, keď má každá strana dĺžku 1. Takže najlepšie, čo teoreticky môžeme dosiahnuť, je $s = 2t + 2$. Inými slovami, $t = (s/2) - 1$.

Ak nám teda niekto povie počet strán s a chce od nás, aby sme mu navrhli záhradku s obsahom menším ako $s/2$, pôjde to len vo veľmi špecifickom prípade: budeme musieť nakresliť záhradku, ktorá má stromovú topológiu a ktorá má všetky strany dĺžky 1. Vedeli by ste teraz v tejto úvahe pokračovať a dokázať, ako vyzerajú *úplne všetky* záhradky pre ktoré $s/2 > t$?

5. Odpad spoločnosti

vzorák napísal Mário
(max. 7 b za popis, 8 b za program)

Skúšanie všetkých dvojíc

Ako sa dá najjednoduchšie zistiť, kto s kým sedí? Spýtame sa na všetky dvojice študentov. Stačí klásť otázky typu 1 2, 1 3, ... a vždy keď dostaneme odpoveď 1, čo znamená, že títo dvaja sedia v jednej lavici, zapamätáme si túto dvojicu. Ak sa na každú dvojicu spýtame len raz, položíme $\binom{2n}{2} = \frac{2n(2n-1)}{2}$ otázok. Časová zložitosť takéhoto riešenia je $O(n^2)$ a mohli ste zaň dostať 2 body.

Pokiaľ neviete s úlohou pohnúť, je dobré skúsiť aspoň to prvé (správne), čo vás napadne. Navyše, na jednoduchom programe si viete vyskúšať spôsob komunikácie s testovačom – "flushovanie" výstupu, spracúvanie viacerých sád a formát otázok a odpovedí.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#define For(i,N) for(int i=0; i<N; i++)
#define FOR(i,z,k) for(int i=z; i<k; i++)

using namespace std;

int N, T;
vector<pair<int,int> > pary;

int main(){
    scanf("%d", &T);
    For(t,T){
        pary.clear();
        scanf(" %d", &N);
        For(i,2*N)
            FOR(j,i+1,2*N){
                printf("2 %d %d\n",i+1,j+1); fflush(stdout);
                int odpoved;
                scanf("%d", &odpoved);
                if(odpoved==1) pary.push_back( make_pair(i+1, j+1) );
            }

        //vypis riesenia
        printf("0\n");
        For(i,N)
            printf("%d %d\n", pary[i].first, pary[i].second);
        fflush(stdout);
```

```
    }
    return 0;
}
```

Skúšanie všetkých rozumných dvojíc

Môžeme si všimnúť, že ak sedia spolu študenti 1 a 2, v predošlom programe sa dozvieme o tejto dvojici hneď v prvej otázke. Ďalších $2n-2$ otázok ohľadom študenta 1 je teda úplne zbytočných. Preto sa dá vyššie popísaný program v praxi mierne zlepšiť, ak si budeme o každom študentovi pamätať, či už vieme, s kým sedí. Pred vypísaním otázky potom zvážime, či je zbytočná, alebo či sa ňou niečo nové dozvieme.

V najhoršom prípade môžu spolu sedieť žiaci $1, 2n$; $2, 2n-1$; $\ldots$ $n, n+1$. Pre prvého sa tak spýtame $2n-1$ otázok, pre druhého $2n-3$, atď. Teda $2(n)-1+2(n-1)-1+2(n-2)-1+\cdots+2(n-(n-1))-1 = 2\frac{n(n-1)}{2}-n = n(n-2)$ otázok. A hoci časová zložitosť zostáva stále $O(n^2)$, aj v najhoršom prípade sa spýtame približne o polovicu otázok menej. Za takéto zlepšenie ste mohli dostať až 4 body.

Binárne vyhľadávnie

Viete, kadiaľ vedie cesta k lepšiemu riešniu[3]? Ak ste bezradní, prečítajte si zadanie a zistite, čo z toho, čo ste mali k dispozícii, ste zatiaľ nevyužili.

V predošlých riešeniach sme totiž vôbec nepoužili to, že sa môžeme pýtať na *ľubovoľnú podmnožinu študentov*. Sústreďme sa na jedného študenta, napr. s číslom 1 a pokúsime sa zistiť jeho spolusediaceho. Ak sa spýtame, v koľkých laviciach sedí celá trieda, dostaneme neprekvapivú odpoveď n. Spýtajme sa radšej:

Koľko lavíc bolo obsadených, ak bola v škole polovica žiakov spolu so žiakom 1?

Dostaneme odpoveď d a toto číslo môže byť v rozmedzí $\langle n/2, n\rangle$. Skúsme sa ešte raz spýtať tú istú otázku bez študenta 1.

- Ak je odpoveď $d-1$, študent 1 sedel v tento deň sám – v tejto polovici triedy jeho spolusediaci nie je, teda je v druhej polovici.
- Ak je odpoveď d, v tejto polovici triedy musí byť aj spolusediaci študenta 1, lebo aj po odchode žiaka 1 zostala jeho lavica obsadená.

Čo sme dosiahli? Pomocou dvoch otázok sme rozhodli, či môže byť polovica študentov spolusediaca s 1. Ďalej nás bude zaujímať už len tá polovica študentov, medzi ktorými je aj jeho spolusediaci. Ďalšími dvoma otázkami vylúčime štvrtinu, potom osminu, atď. Po spýtaní sa $2\log 2n$ otázok bude možný spolusediaci len jeden. Ak sa pre každého študenta spýtame $2\log 2n$ otázok, pre $n = 1\,000$ sa spýtame približne $20\,000$ otázok a za takéto riešnie ste mohli dostať 8 bodov. Spýtame sa teda $O(n\log n)$ otázok.

Ak by vás zaujímala aj časová zložitosť, musíte zvážiť to, že pýtanie sa rôznych otázok vyžaduje rôzny čas. Ak sa napríklad pýtate na polovicu triedy, položenie jednej otázky si vyžiada čas $O(n)$. Pri kladení otázok o jednom študentovi vypisuje program $n, n/2, n/4\ldots$ čísel, čo sa nasčítava na približne $2n$. Časová zložitosť zostáva stále $O(n^2)$, no počet otázok sa nám podarilo efektívne znížiť, čo bolo, koniec koncov, to podstatné v tejto úlohe.

Toto riešenie sa dá tiež mierne zlepšiť, ak sa budeme pýtať len na študentov, o ktorých nevieme, s kým sedia. Na polovice, štvrtiny… nemusíme teda deliť celú triedu, stačí deliť len množinu neznámych študentov.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#define For(i,N) for(int i=0; i<N; i++)

using namespace std;

int N, T;
vector<bool> neznamy;
vector<pair<int,int> > pary;

bool je_v_skupine(int x, int l, int r, vector<int> &V ){
    //ziak x, v intervale <l,r), v casti triedy V
    int bezx, sx;

    if(r-l == 1) bezx = 1;
    else{
        //vypis prvej otazky
        printf("%d", r-l);
        for(int i=l; i<r; i++)
```

[3] Cez binárne vyhľadávnie predsa, veď je to tu napísané!

```
            printf("␣%d", V[i]+1);
        printf("\n");
        fflush(stdout);

        //nacitanie odpovede testovaca
        scanf("%d", &bezx);
    }

    //vypis druhej otazky
    printf("%d␣", r-l+1);
    for(int i=l; i<r; i++)
        printf("%d␣", V[i]+1);
    printf("%d\n", x+1);
    fflush(stdout);

    //nacitanie 2. odpovede testovaca
    scanf("%d", &sx);

    return (sx==bezx);
}

int main(){
    scanf("%d", &T);
    For(t,T){
        neznamy.clear();
        pary.clear();
        scanf("%d", &N);
        neznamy.resize(2*N,1);

        For(i,2*N)
            if(neznamy[i]){
                //chceme zistit, s kym je i v pare
                //pripravime si pole
                vector<int> mozno;
                For(j,2*N) if(neznamy[j] && j != i) mozno.push_back(j);

                //binarne vyhladame, v ktorej skupine je druhy
                int l=0, r=int(mozno.size());
                while(r-l > 1){
                    int piv = (l+r)/2;
                    if( je_v_skupine( i, l, piv, mozno ) )
                        r = piv;
                    else
                        l = piv;
                }

                //ulozime si 1 par do riesenia
                neznamy[i] = neznamy[mozno[l]] = 0;
                pary.push_back( make_pair(i, mozno[l]) );
        }

        //vypis riesenia
        printf("0\n");
        For(i,N)
            printf("%d␣%d\n", pary[i].first+1, pary[i].second+1);
        fflush(stdout);
    }
    return 0;
}
```

vzorák napísal Žaba

6. Opatera zvieratiek

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

Našou úlohou je vybrať takú podpostupnosť vstupného reťazca, že tvorí palindróm. Táto podpostupnosť navyše nemusí byť súvislá a dokonca ani najdlhšia možná, stačí ak bude obsahovať 100 znakov (poprípade, ak tam nie je takých 100 znakov, tak musíme nájsť tú najdlhšiu).

Pokúsme sa zamyslieť nad tým, ako môže vyzerať nejaký najjednoduchší palindróm. Napríklad môže obsahovať len písmená jedného druhu, ako je to v palindrómoch *aaaa* alebo *ddddd*. To ale znamená, že ak náš vstupný reťazec obsahuje 100 výskytov toho istého písmena, vieme nájsť riešenie veľmi jednoducho. Proste vyberieme 100 takýchto písmen.

Náš program teda môže začať nasledovne: Prejde celým vstupných reťazcom a pre každé písmeno si spočíta počet jeho výskytov. Ak sa niektoré písmeno nachádza v reťazci aspoň 100 krát, vypíšeme výsledok skladajúci sa iba z tohoto písmena a program ukončíme.

Pomohli sme si? Veru áno. Aký dlhý môže byť reťazec, v ktorom sa niečo takéto nestane? Každé písmeno sa tam môže vyskytovať najviac 99 krát, teda bude mať dĺžku $99 \cdot 26 = 2574$. Keď pridáme ďalší znak, túto vlastnosť pokazíme. Inak povedané[4], ľubovoľný reťazec dĺžky aspoň 2575 obsahuje aspoň 100 rovnakých znakov. Výrazne sme teda zmenšili obmedzenie na dĺžku vstupu, pre ktorú je naša úloha netriviálna.

Hľadanie najdlhšieho palindrómu

Zostáva nám teda vyriešiť tú zložitejšiu úlohu – ako nájsť najdlhší podpalindróm vo vstupnom reťazci?

[4] Použitím Dirichletovho princípu.

Takáto úloha je pomerne klasická a preto sa oplatí poznať jej riešenie. Dokonca sa ani nebudeme zaoberať tým, či je ten palindróm dlhý najviac 100, lebo už nám to viac pri riešení nepomôže.

Poďme sa vrhnúť rovno na popis riešenia. Nebude to nič iné ako jednoduchá rekurzia s memoizáciou, ktorú si hneď vysvetlíme. To čo hľadáme je **najdlhší palindróm od pozície** 0 **po pozíciu** n.

Jedna možnosť je, že písmená na pozíciách 0 a $n-1$ sú rôzne. V tom prípade aspoň jedno z týchto písmen nebude vo výsledku. Ak by totiž boli obe, boli by na kraji našej podpostupnosti a tým pádom by sme nemali palindróm. To znamená, že odpoveď bude buď **najdlhší palindróm od pozície** 0 **po pozícii** $n-1$ alebo **najdlhší palindróm od pozície** 1 **po pozíciu** n. A toto sú nejaké menšie podproblémy, ktoré môžeme riešiť rekurzívne.

Druhá možnosť je, že obe písmená na krajných pozíciách sú rovnaké. V tom prípade ich môžeme zobrať do výsledku (ktorý je teraz o jedna dlhší) a hľadáme **najdlhší palindróm od pozície** 1 **po pozíciu** $n-1$.

Náš problém sme si teda zapísali ako jednoduchú rekurzívnu funkciu, ktorá bude počítať **aký je najdlhší palindróm od pozície** ℓ **po pozíciu** r, a ktorú by sme teraz mali vedieť bez problémov implementovať. Nezabudnite však na to, že ak už vyrátate odpoveď pre nejakú dvojicu (ℓ, r), treba si túto odpoveď zapamätať, aby sme ju nemuseli pri ďalšom opýtaní opäť rátať celú. Na zapamätanie nám poslúži jednoduché dvojrozmerné pole. Bez tohto zapamätávania (memoizácie) by bola časová zložitosť riešenia exponenciálna.

Takisto nemôžete zabudnúť na to, že na konci musíme vypísať reťazec a teda musíme vedieť spätne zistiť, ktoré písmená patria do najdlhšieho palindrómu. Toto si môžete buď pamätať v ďalšom dvojrozmernom poli, kde si poznačíte, ktorým spôsobom dostanete najdlhší palindróm z daného úseku alebo si to budete značiť priamo pri simulácii. Dajte si však pozor, že keď bude najdlhší palindróm dlhší ako 100 znakov, musíte ho na túto dĺžku skrátiť.

Na záver sa už len pozrime na časovú a pamäťovú zložitosť. Každý stav rekurzie vieme vyrátať v konštantnom čase, stačí keď sa pozrieme na tri iné čísla. Stavov je ale n^2, takže časová zložitosť nášho riešenia je $O(n^2)$. A rovnako to bude aj s pamäťovou zložitosťou, lebo si musím zapamätať výsledok pre každý možný stav.

Keď máme nejako popísať časovú zložitosť celého riešenia, označme si d maximálnu dĺžku palindrómu a a počet písmen abecedy. Dĺžka vstupu je n. Potom časová zložitosť je $O(n + (\min(n, d \cdot a))^2)$. Keď považujeme d a a za konštanty, tak je to jednoducho $O(n)$.

Listing programu (C++)

```
#include <cstdio>
#include <string>
#include <cstring>
#include <algorithm>
#include <vector>
#define For(i,N) for(int i=0; i<N; i++)
#define Fors(i,s) for(int i=0; s[i]; i++)
using namespace std;

vector<int> P(26,0);
char buf[2000000];
vector<vector<int> > T(3000, vector<int>(3000,-1));

int maxd( int l, int r){
    if(l > r) return 0;
    if( T[l][r] != -1 ) return T[l][r];
    if( buf[l] == buf[r] ) return T[l][r] =1+ (l!=r) + maxd( l+1, r-1 );
    return T[l][r] = max( maxd(l+1, r), maxd(l, r-1) );
}

int main(){
    scanf("␣%s", buf);
    Fors(i, buf) P[buf[i]-'a']++;
    For(i,26) if(P[i] >= 100){
        For(j,100) printf("%c", 'a'+i);
        printf("\n");
        return 0;
    }

    int i=0, j=strlen(buf)-1;
    int maxi = maxd(i,j);
    string zac="", kon="";
    while(i<=j){
        while(i<=j && maxd(i,j) == maxi ) i++;
        if(i-1<=j && maxd(i,j) != maxi) zac += buf[i-1];
        while(j >= i && buf[j] != buf[i-1] ) j--;
        if(j > i-1) kon.push_back( buf[j] ); j--;
        maxi = maxd(i,j);
    }
    reverse(kon.begin(), kon.end() );
    if(zac.length()+kon.length()>100) {
        zac="";
        For(i,50) zac+=kon[i];
        For(i,50) zac+=kon[49-i];
    }
    else zac += kon;
    printf("%s\n", zac.c_str());
```

```
    return 0;
}
```

vzorák napísal mišof

7. Osobná doprava

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

Našou úlohou bolo zistiť, či sa dá zo zastávky A dostať na zastávku B, a ak áno, ako najrýchlejšie to vieme spraviť. Toto je zjavne úloha grafová a zrejme bude nejak súvisieť s najkratšími cestami. Ale ako presne?

Základné pozorovanie, ktoré nám pomôže úlohu vyriešiť, je nasledovné: Predstavme si, že sme objavili dva rôzne spôsoby, ako sa zo zastávky A dá dostať na nejakú zastávku C. Prvý z nich tam príde v čase t_1, druhý v čase t_2 a platí $t_1 < t_2$. Pri hľadaní optimálneho presunu z A do B nám stačí zapamätať si len jeden z týchto dvoch spôsobov – ten prvý. Totiž každý spôsob, akým vieme pokračovať zo zastávky C ďalej v druhom spôsobe máme k dispozícii aj pri prvom spôsobe príchodu, a možno máme ešte aj nejaké iné možnosti navyše, ktoré by sme pri príchode v čase t_2 nestihli.

Myšlienka všetkých nižšie popísaných algoritmov bude teda veľmi jednoduchá: V priebehu nášho algoritmu sa budeme snažiť zostrojiť množinu *všetkých* zastávok, ktoré sú dosiahnuteľné zo zastávky A. Pre každú takú zastávku si navyše vypočítame aj **najskorší** čas, v ktorom môžeme na danej zastávke byť.

Predspracovanie vstupu

Už vieme, že vrcholmi nášho grafu budú jednotlivé zastávky. Na vstupe však namiesto hrán máme zadanú množinu spojení, ktoré našimi zastávkami prechádzajú. Priamo pri načítavaní vstupu však vieme každé spojenie prekonvertovať na sadu hrán – ako keby sme ho nahradili viacerými linkami, z ktorých každá premáva len z jednej zastávky spojenia na druhú, bezprostredne nasledujúcu. Všetky hrany pochádzajúce z toho istého spojenia budú mať tú istú periódu. Ich offsety vypočítame jednoducho tak, že simulujeme jednu jazdu dotyčného spojenia a zapisujeme si, kedy sme navštívili ktorú zastávku.

Pre algoritmus Bellmana a Forda (prvé z riešení uvedených nižšie) by nám stačilo uložiť si všetky hrany grafu v jednom dlhočiznom zozname. Pre Dijkstrov algoritmus (druhé, lepšie riešenie) však budeme potrebovať hrany roztriediť: v každom vrchole grafu si budeme pamätať zoznam hrán, ktoré vedú z neho von. Takéto predspracovanie uvádzame aj v nasledovnom listingu.

Listing programu (Python)

```
from collections import defaultdict
from queue import PriorityQueue

def nacitaj_vzdialenosti():
    vzdialenost = {}
    d = int( input() )
    for riadok in range(d):
        x, y, dist = input().split()
        vzdialenost[ (x,y) ] = vzdialenost[ (y,x) ] = int(dist)
    return vzdialenost

class Segment:
    def __init__(self,kam,offset,perioda,cas):
        self.kam     = kam
        self.offset  = offset
        self.perioda = perioda
        self.cas     = cas

def nacitaj_linky(vzdialenost):
    graf = defaultdict(list)

    s = int( input() )
    for riadok in range(s):
        tokeny = input().split()
        v, p, o, z = [ int(x) for x in tokeny[:4] ]
        zastavky = tokeny[4:]
        for i in range(z-1):
            dist = vzdialenost[ (zastavky[i],zastavky[i+1]) ]
            time = (dist + v - 1) // v
            graf[ zastavky[i] ].append( Segment( zastavky[i+1], o, p, time ) )
            o = (o+time) % p
    return graf
```

V ďalšom texte budeme počet zastávok označovať n a celkový počet hrán v našom grafe (teda súčet počtov úsekov jednotlivých spojení) budeme označovať m.

Algoritmus Bellmana a Forda

Tento algoritmus je veľmi jednoduchý. Na začiatku vieme, že na zastávke A vieme byť v čase 0, a na žiadnej inej zastávke ešte nevieme byť. Toto si zapamätáme tak, že si pre ostatné zastávky zapíšeme, že tam vieme byť v čase "nekonečno", pričom ako nekonečno použijeme nejakú dostatočne veľkú hodnotu.

Teraz ideme tieto hodnoty postupne zlepšovať. Zlepšovanie bude prebiehať v kolách. V každom kole sa postupne (je jedno, v akom poradí) raz pozrieme na každú hranu. Predstavme si, že sa práve pozeráme na hranu zo zastávky C na zastávku D. Momentálne sa vieme na zastávku C dostať v čase t_c a na zastávku D v čase t_d. Pomôže nám hrana, ktorú práve skúmame, niečo zlepšiť? Jediné, v čom nám môže táto hrana pomôcť, je, že sa pomocou nej možno vieme skôr dostať na zastávku D. Zoberime teda čas t_c, počkáme na zastávke C na najbližší spoj idúci práve skúmanou hranou a zistíme, kedy tento spoj dorazí na zastávku D. Ak sme práve dostali hodnotu menšiu ako t_d, zmenšíme t_d na práve vypočítanú hodnotu – práve sme objavili lepší spôsob ako docestovať z A na D.

V programe spracovanie jednej hrany vyzerá nasledovne:

Listing programu (Python)

```
def cestuj(odkial,kedy,hrana):
    # sme na zastavke "odkial" v case "kedy"
    # chceme ist hranou "hrana" do jej cielovej zastavky
    # kedy najskor tam vieme byt?

    # pockame na najblizsi spoj iduci po linke "hrana"
    if startt % hrana.perioda != hrana.offset:
        cakaj = hrana.offset - (startt % hrana.perioda)
        if cakaj < 0: cakaj += hrana.perioda
    else:
        cakaj = 0

    # odvezieme sa linkou a vysledok vratime
    return kedy + cakaj + hrana.cas
```

Keď už v nejakom kole nenastanú vôbec žiadne zmeny, celý proces končí. Tvrdíme, že časy, ktoré sme vypočítali, sú najlepšími možnými časmi. Teda ak si pre nejakú zastávku ešte stále pamätáme hodnotu nekonečno, nedá sa na ňu vôbec dostať, a v opačnom prípade čas, ktorý sme vypočítali, je najlepší možný.

Teraz potrebujeme spraviť dve veci: dokázať, že vyššie uvedené tvrdenie platí, a zistiť, akú má tento algoritmus časovú zložitosť.

Všimnime si ľubovoľnú zastávku X, na ktorú sa dá zo zastávky A dostať. Nech je optimálna cesta z A na X tvorená k hranami a postupne prechádza zastávkami X_1 až X_{k-1}. Čo sa deje v našom algoritme? V prvom kole sa postupne pozeráme na všetky hrany, teda aj na hranu z A do X_1. Po prvom kole sa teda už určite vieme dostať do X_1 v potrebnom čase. V druhom kole sa *opäť* postupne pozeráme na všetky hrany, a medzi nimi aj na hranu z X_1 do X_2. Po druhom kole sa teda už vieme v potrebnom čase dostať aj z A do X_2. A postupne ďalej – po k-tom kole už musí byť vypočítaný čas pre zastávku X rovný optimálnemu.

Náš algoritmus je teda korektný. Navyše si uvedomme, že pre ľubovoľnú zastávku existuje optimálna cesta, pre ktorú platí $k < n$. To je jednoduché – nikdy sa neoplatí vracať na zastávku, na ktorej sme už boli, takže existuje optimálna cesta, na ktorej sa zastávky neopakujú. Algoritmu teda bude stačiť nanajvýš $n - 1$ kôl. A keďže v každom kole sa pozeráme na m hrán, je časová zložitosť tohto algoritmu $O(nm)$.

Implementácia spracovania jednej otázky:

Listing programu (Python)

```
def spracuj_otazku(graf,odkial,kam):

    # pre zastavku kde zaciname je vzdialenost 0
    # pre vsetky ostatne zastavky je to 2 na 60 == nekonecno
    najskorsi_prichod = defaultdict(lambda:2**60)
    najskorsi_prichod[odkial] = 0

    zmena = True
    while zmena:
        # zacina nove kolo, postupne sa pozerame na vsetky hrany
        zmena = False
        for odkial in graf:
            for seg in graf[odkial]:
                # zistime, ci hranou "seg" vieme nieco zlepsit

                if najskorsi_prichod[ odkial ] == 2**60: continue

                cas_v_cieli = cestuj( odkial, najskorsi_prichod[odkial], seg )
                if cas_v_cieli < najskorsi_prichod[ seg.kam ]:
                    najskorsi_prichod[ seg.kam ] = cas_v_cieli
                    zmena = True

    # a na zaver uz len vypiseme spravnu odpoved

    if najskorsi_prichod[kam] == 2**60:
        print('neda sa')
    else:
        s = najskorsi_prichod[kam]
        d = s // 86400
        s %= 86400
        h = s // 3600
        s %= 3600
```

```
    m = s // 60
    s %= 60
    print('{}d␣{}h␣{}m␣{}s'.format(d,h,m,s))
```

Dijkstrov algoritmus

Dijkstrov algoritmus počíta presne tie isté hodnoty ako algoritmus Bellmana a Forda, robí to ale šikovnejším spôsobom, a teda dosiahneme lepšiu časovú zložitosť. Existuje viacero rôznych implementácií Dijkstrovho algoritmu. Tá, ktorú si ukážeme my, bude mať časovú zložitosť $O(m \log n)$.

Zlepšenie dosiahneme nasledovne: Namiesto toho, aby sme znova a znova prechádzali celý zoznam hrán, budeme každú hranu spracúvať len raz, "v správnej chvíli". Takisto práve raz budeme spracúvať každú zastávku. Zastávky budeme spracúvať *usporiadané podľa optimálneho času, kedy sa na ne vieme dostať*. Tieto časy síce na začiatku nepoznáme, ale postupne ich budeme zisťovať a vždy budeme vedieť, ktorú zastávku spracovať ako nasledujúcu v poradí.

Na začiatku zjavne spracujeme zastávku A: pozrieme sa na všetky hrany vedúce z nej a tak zistíme nové spôsoby ako sa dostať na nejaké ďalšie zastávky. Zastávku A si následne označíme ako spracovanú.

Ako bude vo všeobecnosti vyzerať kolo tohto algoritmu? Nejaké zastávky sme už spracovali, tie nás už nezaujímajú. Ak sa už na žiadnu ďalšiu zastávku nevieme dostať, algoritmus končí. Inak sa pozrime na tie, ktoré ešte spracované nie sú, a vyberme spomedzi nich zastávku X, na ktorú sa momentálne vieme dostať (spomedzi všetkých nespracovaných) najskôr. Túto zastávku spracujeme ako nasledujúcu v poradí.

Prečo tento algoritmus funguje? Preto, že v okamihu, keď zastávku spracúvame, tak platí, že čas, ktorý sme pre ňu vypočítali, je už optimálny.

(Formálne by sme napríklad matematickou indukciou dokázali, že platí nasledovné tvrdenie: keď sme už spracovali prvých k zastávok, tak pre každú nespracovanú zastávku máme nájdený najlepší čas takej cesty na ňu, počas ktorej môžeme prestupovať len na už spracovaných zastávkach. No a následne si už len stačí všimnúť, že pre nami vybranú zastávku X už nemôže existovať ani rýchlejšia cesta, pri ktorej by sme prestupovali aj na nejakých nespracovaných zastávkach.)

Ako to celé dobre implementovať? Kľúčová operácia, ktorú potrebujeme robiť efektívne, je nájdenie zastávky X, ktorú ideme spracovať ako ďalšiu v poradí. Aby sme toto vedeli robiť rýchlo, budeme si zatiaľ nespracované zastávky *udržiavať usporiadané* podľa doteraz najlepšej vzdialenosti do nich.

V C++ by sme napríklad mohli ako dátovú štruktúru použiť `set< pair<int,int> > Q`, v ktorom by sme mali záznamy tvaru (t, z) s významom "najlepší známy čas, v ktorom už vieme byť na zastávke z, je t". Nájsť nasledovnú zastávku na spracovanie vieme v čase $O(\log n)$ tak, že sa pozrieme na `Q.begin()` (teda zastávku s najmenšou vzdialenosťou). Všetky výbery dokopy počas celého algoritmu nám teda budú trvať len $O(n \log n)$.

Spracovať konkrétnu zastávku vieme v čase $O(d \log n)$, kde d je počet hrán vedúcich z nej. Pre každú hranu skúsime zlepšiť čas pre zastávku, kam vedie. A ak sa nám to podarí, tak z `Q` zmažeme starý záznam pre dotyčnú zastávku a nahradíme ho novým s menšou vzdialenosťou. No a keďže každú zastávku spracujeme počas behu algoritmu najviac raz, nasčítajú sa nám časy ich spracovania na sľubovaných $O(m \log n)$.

V našej implementácii sme použili trochu lenivejší prístup. Ako dátovú štruktúru použijeme obyčajnú prioritnú frontu (implementovanú ako haldu). Z nej síce nevieme priebežne vymazávať, ale to nám vôbec vadiť nebude – jednoducho to nebudeme robiť a keď zlepšíme čas pre nejakú zastávku, do prioritnej fronty pridáme nový záznam s novým lepším časom. A potom len pri výbere ďalšej zastávky na spracovanie dáme pozor, či nejde o zastávku, ktorú sme už skôr spracovali. Takéto záznamy jednoducho preskočíme (a teda ich vlastne až vtedy zmažeme).

V najhoršom prípade bude naša prioritná fronta obsahovať až $O(m)$ záznamov naraz. (Totiž každú hranu v grafe spracúvame najviac raz a každé také spracovanie nám pridá najviac jeden záznam.) Všetky operácie s prioritnou frontou budú teda v čase $O(\log m)$ a teda výsledná časová zložitosť našej implementácie bude $O(m \log m)$.

Listing programu (Python)

```
def spracuj_otazku(graf,odkial,kam):

    najskorsi_prichod = defaultdict(lambda:2**60) # ak este nepriradil hodnotu, je to 2^60
    najskorsi_prichod[odkial] = 0

    Q = PriorityQueue()
    Q.put( (0,odkial) )
    while not Q.empty():
        kedy, kde = Q.get()
        if najskorsi_prichod[kde] < kedy: continue # uz sme ho spracovali
```

```
        for seg in graf[kde]:
            cas_v_cieli = cestuj( kde, kedy, seg )
            if cas_v_cieli < najskorsi_prichod[ seg.kam ]:
                najskorsi_prichod[ seg.kam ] = cas_v_cieli
                Q.put( (cas_v_cieli, seg.kam) )

    # vypis riesenia je rovnaky ako v algoritme Bellmana a Forda
```

vzorák napísal Žaba

8. Okná sa vymieňajú

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

Opäť úloha podľa môjho gusta. Dôvod, prečo sa mi páči je ten, že na jej riešenie nie je potrebné dostať správny nápad, ale postupne zlepšovať riešenie v sérii malých krokoch. Nuž a tieto kroky si v tomto vzorovom riešení ukážeme.

Pomalý začiatok

Keď si nevieme poradiť s nejakou úlohou, vždy je dobré začať pomalým riešením, poprípade sa pozrieť na malé príklady a zistiť, ako sa to správa na nich. Začneme teda tak, že si naprogramujeme jednu funkciu, ktorá bude simulovať náš komprimátor – bude zisťovať počet jednotiek v binárnom zápise čísla a, a druhú funkciu, ktorá bude počítať, koľkokrát musíme použiť tento komprimátor na to, aby sme z čísla a dostali číslo 1. Takéto niečo by malo byť veľmi jednoduché na naprogramovanie a nemali by ste s tým mať problém.

Následne spustíme náš program na prvých 100 číslach a pozeráme, aké výsledky dostávame. Prekvapivo, sú to všetko veľmi malé čísla, najväčšie, ktoré nájdeme je 3. Skúsime to pre prvých milión čísiel a zistíme, že najväčší počet komprimácií, ktorý vieme dosiahnuť je 4. To vyzerá nanajvýš podozrivo, skúsme sa teda zamyslieť, prečo sú tieto čísla také malé.

Keď sa pozrieme na formát vstupu, zistíme, že najväčšie číslo, ktoré môžeme dostať je 10^{18} – čo je číslo, ktoré sa pohodlne zmestí do 64 bitovej premennej. To ale znamená, že najviac 64 bitov môže byť rovných 1.[5] Teda po prvej komprimácii ľubovoľného čísla v našom rozmedzí l až h dostaneme číslo menšie ako 64. A pre takto malé čísla už vieme, že na komprimáciu na 1 potrebujeme najviac 3 komprimácie. To znamená, že najväčšie k, pre ktoré existuje nenulová odpoveď je 4.[6]

Toto pozorovanie bude veľmi kľúčové pre ďalší postup. Aj bez neho však máme veľmi jednoduché riešenie, ktoré pre každé číslo l až h zistí, koľko potrebuje komprimácií aby z neho vzniklo číslo 1 a ak sa toto číslo rovná k, zväčší o jedna svoju odpoveď.

Od stredu k riešeniu

Interval l až h je pomerne veľký, môže obsahovať veľké množstvo čísiel. Predchádzajúce pozorovanie nám však vraví, že po prvom kroku budú všetky čísla komprimované na niečo menšie ako 64. Skúsime to teda riešiť od tohoto zlomového okamihu, po prvej komprimácií, a skúsime spočítať všetky vhodné čísla. Z hodnoty čísla k vieme, že musíme spraviť ešte $k-1$ komprimácií. Pozrieme sa teda na prvých 63 čísiel a zistíme, ktoré z nich potrebujú ešte $k-1$ komprimácií na to, aby sme dostali 1. Do nášho riešenia teda chceme zarátať všetky čísla, ktoré sa po prvej komprimácii zmenia na niektoré z týchto čísiel. Dokopy totiž budú potrebovať práve k komprimácií.

Nuž a čo vieme povedať o čísle, ktoré sa skomprimuje na číslo a? Že v jeho binárnom zápise sa nachádza práve a jednotiek. Uvedomme si teraz, že ak by sme vedeli odpovedať na nasledovnú otázku, máme vyhrané: "Koľko je takých čísiel medzi l a h, že majú práve a jednotiek v binárnom zápise?"

Náš program by teraz vyzeral nasledovne. Predrátali by sme si riešenie pre prvých 63 čísiel. Následne by sme cez ne prešli a vždy, keď by nejaké číslo a potrebovalo $k-1$ komprimácií, k výsledku by sme pripočítali počet čísiel medzi l a h, ktoré majú a jednotiek v binárnom zápise. Každé také a nám dá vlastnú nezávislú množinu čísiel, všetky dokopy dávajú odpoveď. Ostáva už len zistiť, ako odpovedať na našu otázku.

Jedno ohraničenie namiesto dvoch

Tento trik by mal byť pomerne známy a dá sa veľmi pekne využiť aj v našej úlohe. Otázka, ktorú sa pýtame má dve ohraničenia l a h. Veľmi ľahko to však vieme zmeniť len na to horné. Presnejšie, budeme chcieť vedieť, koľko je takých čísiel menších alebo rovných ako h, že potrebujú k komprimácií. Toto nám samozrejme dá nejaké čísla navyše. To sú ale tie čísla, ktoré potrebujú k komprimácií a sú menšie alebo rovné ako $l-1$. Ak teda máme

[5] Dokonca je to len 63, keďže prvý bit sa používa ako znamienko.

[6] Je zaujímavé si všimnúť, že na to, aby sme dostali odpoveď 5 potrebujeme číslo 2^{127}. A na odpoveď 6 až číslo $2^{2^{127}}$, čo je oveľa viac ako počet atómov vo vesmíre. Táto pomerne pomaly rastúca funkcia sa volá hviezdičkový logaritmus.

funkciu, ktorá odpovedá na našu otázku pre horné ohraničenie, na interval to vieme zmeniť odčítaním riešení pre h a $l-1$.

Takéto zjednodušenie nám môže výrazne pomôcť pri programovaní aj rozmýšľaní, lebo nám stačí dodržiavať len jednu podmienku. V tomto prípade to bude značná pomoc.

Posledný kúsok

Postupne sme si teda našu úlohu značne zmenili a teraz sa pýtame, koľko existuje čísiel menších ako h, ktoré majú práve a jednotiek v binárnom zápise. Poďme si tieto čísla postupne vytvárať, jednu cifru za druhou. A, samozrejme, myslíme binárne cifry.

Predstavme si, že číslo h máme zapísané po binárnych cifrách v poli $H[]$, ktoré má n políčok a na 0-tej pozícii je najmenej významná cifra. Budeme postupne vytvárať všetky čísla, ktoré sú nanajvýš takto veľké a obsahujú a cifier 1. Všetky tieto čísla naviac budú mať n cifier aj keby mali začínať prebytočnými nulami.

Zjavne $H[n-1]=1$, lebo najdôležitejšia cifra musí byť 1. A ako môže vyzerať naše vytvárané číslo? To bude mať na pozícii $n-1$ buď cifru 1 alebo 0. Ak tam dáme 0, tak bez ohľadu na to ako bude číslo pokračovať, naše hľadané číslo bude menšie ako h. Takže chceme splniť už len to, aby malo aj správny počet jednotiek. Koľkými spôsobmi vieme uložiť a jednotiek na $n-1$ pozícii? To je jednoduché kombinačné číslo $n-1$ nad a. Všetky tieto čísla teda môžeme pričítať do výsledku. V prípade, že na pozíciu $n-1$ dáme 1, musíme pokračovať v skúšaní ďalších cifier. Minuli sme si jednu jednotku, takže ich chceme rozdať už len $a-1$.

Pokračujeme teda v skúšaní možností pre pozíciu $n-2$, $n-3$, ... Pokiaľ narazíme na cifru 1 vyriešime to ako v predošlom prípade, ak nájdeme v H cifru 0, tak naše hľadané čísla (tie ktoré sme ešte nezapočítali) musia mať na rovnakej pozícii tiež nulu, inak by boli väčšie ako h.

Takto pokračujeme až kým neprídeme na začiatok poľa H. No a medzičasom sme zrátali všetky možné čísla, ktoré môžu slúžiť ako výsledok.

Rekapitulácia

Ako teda vyzerá celé riešenie pokope? Na začiatku načítame vstup a spravíme nejaké predrátania. Presnejšie, zrátame odpoveď pre prvých 63 čísiel a predpočítame si Pascalov trojuholník do hĺbky 63. Ten nám bude slúžiť na to, aby sme vedeli rýchlo povedať hodnotu jednotlivých kombinačných čísiel. Stačí nám to počítať do dvojrozmerného poľa postupne od malých hodnôt k väčším, klasickým prístupom – sčítame dve susedné z predošlého riadku.

Pokračujeme tým, že vyrátame výsledok pre horné ohraničenie h. Toto číslo si zmeníme na binárne. Potom prechádzame prvých 63 čísiel a vždy, keď vidíme, že nejaké číslo a má riešenie $k-1$ komprimácií, zistíme počet čísiel menších ako h, ktoré obsahujú a jednotiek v binárnom zápise. Toto zistíme vyššie popísaným dynamickým programovaním cez jednotlivé cifry.

Od výsledku odrátame výsledok pre horné ohraničenie $l-1$ a máme výsledok pôvodnej úlohy. Zostáva už len odhadnúť časovú a pamäťovú zložitosť. Nebudeme sa tváriť, že 63 je konštanta, lebo to vzniklo ako logaritmus čísla h a tak to aj budeme označovať. Pamätať si musíme binárny zápis čísla h a Pascalov trojuholník, ktorý je kvadratický, takže pamäťová zložitosť bude $O(\log^2 h)$. A časová zložitosť bude úplne rovnaká, keďže pre najviac $\log h$ čísiel pustíme dynamiku so zložitosťou $O(\log h)$.

Listing programu (C++)

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

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

int A[74];
ll F[63][63];

void predrataj() {
    // počet jednotiek v bin. zápise
    A[1]=0;
    for(int i=2; i<70; i++) {
        int x=i,p=0;
        For(j,10) if(x&(1<<j)) p++;
        A[i]=1+A[p];
    }
    // kombinačné čísla
    F[0][0]=1;
    for(int i=1; i<63; i++) {
        F[i][0]=1;
        for(int j=1; j<=i; j++) F[i][j]=F[i-1][j]+F[i-1][j-1];
    }
}
```

```
// koľko je čísel menších ako 'h', ktoré majú v bin. zápise 'jed' jednotiek
ll pocet(ll h, int jed) {
    vector<int> P;
    ll res=0;
    if(jed==1) res=-1;
    ll x=h;
    while(x>0) {P.push_back(x%2); x/=2;}
    reverse(P.begin(),P.end());
    int bity=P.size();
    For(i,P.size()) {
        if(jed<0) break;
        if(P[i]==0) continue;
        res+=F[bity-i-1][jed];
        jed--;
    }
    if(jed==0) res++;
    return res;
}

// koľko je čísel menších ako 'h', ktoré sa skomprimujú na 'kolko' krokov
ll zrataj(ll h, int kolko) {
    if(h==0) return 0;
    if(kolko==0) return 1;
    ll res=0;
    for(int i=1; i<63; i++) {
        if(A[i]!=kolko-1) continue;
        res+=pocet(h,i);
    }
    return res;
}

int main() {
    predrataj();
    int t;
    scanf("%d",&t);
    while(t--) {
        ll l,r;
        int x;
        cin >> l >> r >> x;
        cout << zrataj(r,x)-zrataj(l-1,x) << endl;
    }
}
```