B-II-1 Výstavba v Egypte pokračuje

Na zisk troch bodov stačilo implementovať postup uvedený v zadaní a po jednom posielať lode do príslušných miest. Takéto riešenie je však pomalé, keďže zadané hodnoty sú pomerne veľké.

Pre jednoduchosť najprv vyriešme úlohu, v ktorej nám stačí odpovedať na jedinú otázku, ktorú si označíme k. Vyššie popísané riešenie by sme zlepšili, ak by sme vedeli lode odosielať po väčších počtoch ako po jednej, aby sme sa ku k dopočítali rýchlejšie. K tomu si potrebujeme všimnúť niekoľko užitočných vlastností.
Zoberme si mesto i, ktoré potrebuje x_i dodávok kameňa. Kedy vyšleme do tohto mesta prvú loď? Vtedy, keď dostatočne vyhovieme všetkým mestám, ktoré mali väčšie požiadavky. Ak totiž akékoľvek iné mesto vyžaduje viac ako x_i dodávok, bude uprednostnené pred mestom i. A zároveň, do žiadneho mesta s menšou požiadavkou ako x_i loď určite nepôjde.
Z tohto vyplýva, že dodávky do miest sa snažia postupne vyrovnať. Na začiatku pôjdu všetky dodávky do mesta s najväčšou hodnotou x_i až kým sa táto hodnota nevyrovná druhému najväčšiemu číslu. Od tohto momentu už musia dodávky ísť do oboch miest – vždy keď pošleme dodávku do jedného, to druhé bude mať najvyššiu aktuálnu potrebu. Takto sa ich hodnoty x_i dostanú na úroveň mesta s treťou najväčšou hodnotou atď.
Postupne nám teda pribúdajú mestá, do ktorých je nutné posielať zásielky kameňa a to v poradí na základe ich hodnôt x_i. Pre jednoduchosť si predstavme, že hodnoty x_i sú zadané v zostupnom poradí, teda mesto 1 potrebuje najviac kameňa, mesto n najmenej. Ak začneme postupne posielať lode, budú sa nám postupne znižovať prvé z týchto hodnôt. V nejakom momente sa prvých i hodnôt bude rovnať číslu x_i.
Vieme, že od tohto momentu sa budú zásielky posielať do týchto miest, až kým ich hodnota nedosiahne hodnotu x_{i+1}. Takýchto zásielok bude presne $(x_i - x_{i+1}) \cdot i$, čo môže byť potenciálne veľké číslo a ak je medzi nimi naša k-ta loď môžeme mať problém. V skutočnosti si však môžeme všimnúť, že tieto lode sú vysielané vo veľmi pravidelnom poradí. Keďže všetky potrebujú aktuálne rovnako veľa zásielok (x_i), zvolí sa to z nich, ktoré malo pôvodne najvyššiu potrebu (a v prípade rovnosti to s najmenším číslom), čo je v našom zjednodušenom prípade určite mesto 1. Jeho hodnota však následne klesne, pri ďalšom náklade ho teda neuvažujeme a vyhrá mesto 2. A po ňom $3, 4 \ldots i$.
V tomto momente sú hodnoty všetkých prvých i miest rovné $x_i - 1$, sme preto v podstate v úplne rovnakej situácii a posielanie bude opäť v poradí $1, 2, 3 \ldots i$. Vďaka tomu, že sa posielanie takto systematicky opakuje vieme jednoducho zistiť, do ktorého mesta ide požadovaná loď. Ak by sme napríklad chceli vedieť, do ktorého mesta pôjde p-ta takto vyslaná loď (teda nie od začiatku, ale od momentu, keď bolo prvých i hodnôt rovných x_i), stačí jednoducho vypísať hodnotu $((p-1) \bmod i) + 1$, kde mod označuje zvyšok po delení.
V našom zadaní je to o čosi komplikovanejšie – hodnoty x_i nemusia byť usporiadané. Stačí však, ak si ich po načítaní usporiadame my sami: primárne podľa väčšieho x_i, sekundárne podľa menšieho i. Pre správny výpis odpovede si ešte budeme musieť pamätať, ktorá hodnota v usporiadanom poradí patrila ku ktorému mestu, inak je však riešenie totožné.
Na otázku, do ktorého mesta bude vyslaná k-ta loď, teraz vieme odpovedať nasledovne. Postupne sa budeme pozerať na to, koľko lodí musíme vyslať, kým sa potreby prvých $i + 1$ miest budú rovnať hodnote x_{i+1} (predpokladajúc, že hodnoty x_i sú usporiadané), ak vieme, že prvých i hodnôt je rovných x_i. Začíname hodnotou $i = 1$. Vieme, že tento počet je $p = (x_i - x_{i+1}) \cdot i$. Ak platí, že $k \leq p$ hľadaná loď je medzi nimi a pôjde do mesta $((k-1) \bmod i) + 1$. V opačnom prípade môžeme týchto p lodí ignorovať a pokračovať s ďalšou hodnotou i. Samozrejme však tento počet odčítame od k, keďže v ďalšej várke lodí už nehľadáme tú k-tu, ale $(k-p)$-tu.
V okamihu ako máme hodnoty x usporiadané, tak časová zložitosť takéhoto riešenia je $O(n)$ – pre každé mesto spravíme zopár počtových operácií.
Rozšírenie o odpovedanie na viacero otázok samozrejme nemôžeme spraviť tak, že tento postup budeme opakovať, pretože by sme dostali zložitosť $O(n \cdot q)$. Namiesto toho si vzostupne usporiadame aj všetky položené otázky a ako prechádzame jednotlivými hodnotami i snažíme sa najprv odpovedať na prvú ešte neodpovedanú otázku. Ak je jej hodnota väčšia, musia byť väčšie aj všetky ďalšie a môžeme sa posunúť ďalej. V opačnom prípade odpovieme na túto otázku a skontrolujeme aj ďalšiu v poradí. Dokopy odpovieme na každú otázku práve raz a časová zložitosť takéhoto riešenia bude $O(n + q)$.
Samozrejme, pri odhade zložitosti nemôžeme zabudnúť na to, že hodnoty x a k sme museli usporiadať. Výsledná časová zložitosť je teda $O(n \log n + q \log q)$.

Listing programu (Python)

```
n = int(input())
X = list(map(int, input().split()))
potrebny_kamen = [ (X[i], i + 1) for i in range(n) ]
# pridaj umele mesto na zastavenie prehladavania
potrebny_kamen.append((0, n+47))
# usporiadaj prvky klesajuc podla poctu (preto minus), stupajuc podla poradia
potrebny_kamen.sort(key=lambda mesto: (-mesto[0], mesto[1]))

q = int(input())
K = list(map(int, input().split()))
otazky = [ (K[i], i) for i in range(q) ]
# usporiadaj otazky
otazky.sort()

print(potrebny_kamen)
print(otazky)

posledna_lod = 0
posledna_otazka = 0
vysledok = [0] * q
for i in range(n):
    zmensenie = potrebny_kamen[i][0] - potrebny_kamen[i + 1][0]
    lode_v_iteracii = (i + 1) * zmensenie
    print(i, posledna_lod, lode_v_iteracii)
    while posledna_otazka != q and otazky[posledna_otazka][0] <= posledna_lod + lode_v_iteracii:
        otazka = otazky[posledna_otazka]
        vysledok[otazka[1]] = potrebny_kamen[(otazka[0] - posledna_lod - 1) % (i + 1)][1]
        posledna_otazka += 1
    posledna_lod += lode_v_iteracii

print(*vysledok, sep='\n')
```

B-II-2 Generátor príkladov

Riešenia hrubou silou

Prvým pozorovaním je, že pokiaľ dovolíme iba guľaté zátvorky, tak všetky správne uzátvorkované výrazy dĺžky n majú $n/2$ otváracích a $n/2$ zatváracích zátvoriek. Nasledujúci dobre uzátvorkovaný reťazec vieme potom hrubou silou nájsť napríklad tak, že vygenerujeme všetky permutácie zátvoriek, usporiadame ich, nájdeme tú zo vstupu a pozrieme sa, aká nasleduje.

Pamäťová zložitosť tohto riešenia je $O(n)$. Časová zložitosť tohto riešenia je rovná počtu permutácií, ktoré otestujeme. Mohlo by sa tak zdať, že tak dostávame riešenie s časovou zložitosťou $O(n!)$, nie je tomu ale tak. Dobrým argumentom je, že na počet všetkých permutácií, ak máme iba dva rôzne prvky sa môžeme pozrieť tak, že sa na každom mieste rozhodujeme, či naň dáme otváraciu alebo zatváraciu zátvorku a teda počet permutácií je nanajvýš 2^n. Potrebujeme ich ešte všetky usporiadať. Každé porovnanie dvoch permutácií trvá $O(n)$ času, a porovnaní pri triedení spravíme rádovo $2^n \log(2^n) = n2^n$. Dostávame teda časovú zložitosť $O(n^2 2^n)$.

O čosi lepšie použitie hrubej sily je že postupne generujeme nasledujúce permutácie zadaného reťazca, až kým nevygenerujeme takú, ktorá je dobre uzátvorkovaná. (Napr. v jazyku C++ máme k dispozícii knižničnú funkciu `next_permutation`, ktorá z danej permutácie vyrobí lexikograficky nasledujúcu. To nám implementáciu riešenia hrubou silou ešte viac zjednoduší.)

Ostáva vedieť overiť, či je výraz správne uzátvorkovaný. To vieme napríklad tak, že výraz prechádzame zľava doprava a pamätáme si počet ešte neuzavretých otváracích zátvoriek naľavo od aktuálnej pozície. V prípade, že narazíme na otváraciu zátvorku, počítadlo zvýšime, pri zatváracej ho zase znížime. Chyba nastane, ak niekedy chce počítadlo ísť pod nulu (zatváracia zátvorka ktorá nemá zodpovedajúcu otváraciu) a tiež ak úplne na konci nie je počítadlo na nule (prevyšujú otvorené zátvorky, ktoré sme neuzatvorili). Tým máme všetko čo potrebujeme na riešenie za 3 body.

Listing programu (C++)

```
#include <iostream>
#include <stack>
#include <algorithm>
```

```
using namespace std;

bool dobre_uzatvorkovane(const string& input)
{
  int otvaracie = 0;
  for (char x : input)
  {
    if (x == '(')
      ++otvaracie;
    else
    {
      if (otvaracie == 0)
        return false;
      --otvaracie;
    }
  }
  return (otvaracie==0);
}

string next(string vstup)
{
  while (next_permutation(vstup.begin(), vstup.end()))
    if (dobre_uzatvorkovane(vstup))
      {
        return vstup;
      }
  return "-1";
}

int main()
{
  string vstup;
  cin >> vstup;
  cout << next(vstup) << "\n";
  return 0;
}
```

Predchádzajúce riešenie stačilo na vstupy, kde je vstupný reťazec s krátky a zložený iba z guľatých zátvoriek. Pre riešenie malých vstupov, ktoré obsahujú aj hranaté zátvorky musíme jednak vymyslieť iné generovanie všetkých možností a tiež overovanie, či je postupnosť dobre uzátvorkovaná. Problémom v tomto prípade je, že všetky možné správne uzátvorkované reťazce už nie sú iba permutácie, nakoľko môžu obsahovať aj hranaté zátvorky. Vhodným prístupom recyklujúcim časť kódu a myšlienku predchádzajúceho riešenia je vyskúšať všetky možné počty hranatých zátvoriek a overiť všetky permutácie pre daný počet hranatých zátvoriek. Treba ešte domyslieť, ako overovať či je reťazec správne uzátvorkovaný. Riešenie sa nebude moc líšiť od predchádzajúceho. Okrem počtu otváracích zátvoriek si počas prechodu overovaným reťazcom budeme pamätať aj ich vzájomnú polohu. To vieme robiť efektívne pomocou dátovej štruktúry zásobník tak, že si do zásobníka hádžeme otváracie zátvorky. Zatváracie zátvorky porovnávame s posledným prvkom, ktorý sme vložili do zásobníka a ak zátvorky typovo sedia, odstránime otváraciu zátvorku zo zásobníka a považujeme ju za uzavretú.

Listing programu (C++)

```
bool dobre_uzatvorkovane(const string& input)
{
  stack<char> s;
  for (char x : input)
  {
    if (x == '(' || x == '[') // otvaracie zatvorky hadzeme na zasobnik
      s.push(x);
    else
    {
      if (s.empty())
        return false;
      if(x == ')' && s.top() == '[' || (x == '(' && s.top() == ']'))
        return false;
      s.pop();
    }
  }
  return s.empty(); // nezabudneme skontrolovat ci je zasobnik prazdny
}

string next(string vstup)
{
  int n = vstup.size();

  string zly_najvacsi(n, ']'); // pomocny retazec, ktory je vacsi ako vsetko
  string minimalny = zly_najvacsi;

  for(int hranate=0; hranate<=n; hranate += 2)
  {
```

```
    string curr = string((n-hranate)/2, '(') + string((n-hranate)/2, ')');
    curr += string(hranate/2, '[') + string(hranate/2, ']');
    do
    {
      if (dobre_uzatvorkovane(curr) && curr > vstup && curr < minimalny)
      {
        minimalny = curr;
      }
    } while (next_permutation(curr.begin(), curr.end()));
  }

  if(minimalny == zly_najvacsi) //nenasli sme ziadny vyhovujuci vyraz
    return "-1";
  else
    return minimalny;
}
```

Ešte šikovnejší spôsob generovania všetkých dobre uzátvorkovaných reťazcov dĺžky n, každého práve raz, vyzerá tak, že vyskúšame obe možnosti pre prvú otváraciu zátvorku, všetky možnosti pre dĺžku výrazu, ktorý je v tejto zátvorke, všetky možnosti pre tento výraz (rekurzívne volanie pre zvolenú menšiu dĺžku) a všetky možnosti pre výraz nasledujúci za týmto celým (druhé rekurzívne volanie pre takú dĺžku, aby sme dokopy dostali n).

Listing programu (Python)

```
def generuj(n):
    # vráti všetky možné výrazy z n párov zátvoriek
    if n == 0:
        yield ''
    else:
        for prva in [ '()', '[]' ]:
            for dnu in range(n):
                for vdnu in generuj(dnu):
                    for vvon in generuj(n-1-dnu):
                        yield prva[0] + vdnu + prva[1] + vvon

n = int(input())
for kombo in sorted(generuj(n)):
    print(kombo)
```

Efektívne riešenie pre guľaté zátvorky

Najskôr sa pozrieme na riešenie jednoduchšej podúlohy, teda budeme predpokladať iba použitie guľatých zátvoriek. Naším cieľom je vstupný reťazec prerobiť tak, aby sme dostali prvý lexikograficky väčší dobre uzátvorkovaný reťazec. Predstavme si, že ako vstup dostaneme napr. reťazec $s =$ `(((())))`. Je zjavné, že musíme nejakú zátvorku zmeniť aby sme dostali iný reťazec. Predstavme si, že máme dve slová a a b, ktoré sú v lexikografickom poradí neskôr ako s. Označme si prvé políčko, na ktorom sa líši a a s ako k_a. Podobne si označíme ako k_b prvé miesto kde sa líši b a s. Uvedomme si, že na pozícií k_a resp. k_b muselo prísť ako v a tak aj v b oproti s k zmene otváracej zátvorky na zatváraciu. Inak by platilo, že a a b sú v lexikografickom usporiadaní skôr ako s. Predpokladajme, že $k_a < k_b$. V tomto prípade si ale rozmyslite, že potom b je v lexikografickom poradí bližšie ku s ako a. Toto pozorovanie nám hovorí, že chceme pri prerábaní reťazca zachovať čo najväčší začiatok slova s nezmenený.

Ukážeme si to teraz na príklade. V prípade $s =$ `(((())))` je nemožné otočiť prvú zátvorku. Druhú sa dá vymeniť za zatváraciu, čím dostaneme reťazec začínajúci `()....`. Chceme to ale spraviť? Ak by sme ju nechali na pokoji, mali by sme reťazec začínajúci `((....`, a zjavne hocijaký takýto reťazec je v lexikografickom poradí skôr ako `()....`. Otázka teda je, či ešte existuje nejaký dobre uzátvorkovaný reťazec začínajúci `((....` okrem nášho aktuálneho. A skutočne: môžeme v aktuálnom reťazci zmeniť tretiu zátvorku na zatváraciu. Tým dostaneme reťazec začínajúci `(()...`. Takéto dobre uzátvorkované reťazce skutočne existujú. Ten, čo hľadáme, je lexikograficky najmenší spomedzi nich.

Prišli sme teda na to, že ďalší reťazec v lexikografickom poradí chceme generovať tak, že sa budeme snažiť nezmeniť čo najdlhší začiatok reťazca na vstupe. Pozrime sa teraz na to, ako nájsť prvú zátvorku, ktorú môžeme zmeniť z otváracej na zatváraciu. To najjednoduchšie čo vieme robiť je ísť od konca a pre každú otváraciu zátvorku sa opýtať, či je možné ju zmeniť na zatváraciu. Naskytá sa teraz otázka, čo ovplyvňuje to, či môžeme zmeniť zátvorku na pozícií i z otváracej na zatváraciu. Všetko napravo od tejto zátvorky môžeme zmeniť, takže to nijak neovplyvňuje naše rozhodnutie, či môžeme zátvorku vymeniť. To čo nás ovplyvňuje je to, ako vyzerá náš

reťazec naľavo od zátvorky na pozícii i. Kedže chceme zmeniť otváraciu zátvorku na indexe i na zatváraciu, musí platiť, že naľavo je aspoň jedna nespárovaná otváracia zátvorka. Čo teraz ďalej urobiť? Ak sme nespárovanú zátvorku otočili, máme už garantované, že náš reťazec bude v lexikografickom poradí neskôr ako pôvodný reťazec. To znamená, že už chceme len dogenerovať zvyšok nového reťazca tak, aby bol v lexikografickom poradí čo najmenší. To urobíme tak, že za pozíciu i naukladáme najskôr všetky zostávajúce otváracie a až potom všetky zostávajúce zatváracie zátvorky. Toto je zjavne lexikograficky najmenšia zo všetkých možností, ako doplniť zvyšok, a navyše vieme, že je korektná, lebo keďže najskôr dávame otváracie zátvorky, nemôže sa nám stať, že by niektorá zatváracia zátvorka nemala naľavo od seba svoj pár.
Vo vyššie uvedenom príklade by sme teda začiatok `(()...` následne doplnili na `(()())`.
Časová zložitosť tohto riešenia je $O(n)$ pretože prejdeme vstupný reťazec práve raz a na každom políčku robíme iba konštantný počet operácií. Pamäťová zložitosť ostáva $O(n)$.

Listing programu (C++)

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

using namespace std;

string next(string x)
{
  int n = x.size(),
      nezatvorene = 0; // dlzka vstupu a pocet neuzatvorenych zatvoriek nalavo
  for (int i = n - 1; i >= 0; --i)
  {
    if (x[i] == ')') // jednoduchy pripad, ideme dalej
    {
      ++nezatvorene;
      continue;
    }
    else
    {
      --nezatvorene;
    }

    // zlozitejsi pripad, skusime otocit '(' na ')'
    int volnych = n - i - 1; // kolko miesta mame napravo
    int volitelne = volnych - (nezatvorene - 1); // kolko vieme vyuzit na ((...()...))

    if (nezatvorene != 0 &&
        volitelne % 2 == 0) // druha podmienka v skutocnosti nie je potrebna
    {
      int otvaracich = volitelne / 2;
      string out;
      out = x.substr(0, i);
      out += ')';
      out += string(otvaracich, '(') + string(n-i-1-otvaracich, ')');
      return out;
    }
  }

  return "-1";
}
```

Vzorové riešenie

Myšlienka vzorového riešenia do veľkej miery využíva pozorovania z predchádzajúceho riešenia. Základná pointa ostáva zachovaná: čím dlhší začiatok reťazca zvládneme nechať bez zmeny, tým lexikograficky menší výsledok dostaneme. Rovnako ako v predchádzajúcom riešení preto pôjdeme od konca, kým nenájdeme prvé miesto, kde reťazec vieme zväčšiť. Teraz ešte potrebujeme domyslieť tri detaily:

- Ako spoznať, kedy môžeme jednu zátvorku nahradiť inou? (Toto bude trochu komplikovanejšie, ale bude to našťastie závisieť len od situácie, ktorú vidíme na danom mieste reťazca. Detailne si to popíšeme nižšie.)
- Ak máme viac možností, čím zátvorku nahradiť, ktorú si vybrať? (Toto je jednoduché, preto to vybavíme rovno tu: ak máme viac vyhovujúcich možností, vyberieme si najskoršiu z nich, tá zjavne vyrobí lexikograficky menší reťazec ako tie ostatné.)
- Keď už sme zátvorku nahradili inou, ako najlepšie doplniť zvyšok reťazca?

Pozrime sa na to, v čom je problém pri zmene zátvorky a dopĺňaní zvyšku reťazca. Keď sme mali len guľaté zátvorky, stačilo nám vedieť, koľko otváracích a koľko zatváracích zátvoriek máme doplniť a mohli sme to spraviť pažravo. Aj tu by sme chceli najradšej dopĺňať to isté, keďže guľaté zátvorky sú „v abecede“ skôr, nie vždy to však smieme urobiť. To, čo môžeme a nemôžeme robiť, totiž závisí od toho, ako vyzerá nezmenená časť reťazca – presnejšie, aké otváracie zátvorky v nej nie sú uzavreté. Tie totiž niekedy treba uzavrieť, a to v správnom poradí.

Toto nám zároveň hovorí jednu ľahkú kontrolu toho, ktoré zmeny určite nemôžeme robiť. Určite nemôžeme zmeniť `)` na `]`, lebo tá nová zátvorka nebude pasovať k tej otváracej, ktorú zatvárala tá stará. No a podobne, ak chceme meniť otváraciu zátvorku na zatváraciu, potrebujeme vedieť, akého typu bola *naposledy otvorená* otváracia zátvorka, lebo práve tú musíme zavrieť. Nižšie už budeme uvažovať len takéto zmeny (a ešte nižšie si ukážeme, ako túto podmienku efektívne vyhodnotiť).

Uvažujme teraz ľubovoľnú situáciu, v ktorej sme našli zátvorku, ktorú sme vedeli zväčšiť a vybrali sme si nejakú korektnú možnosť ako ju zväčšiť. Teraz teda potrebujeme dve veci: v prvom rade vedieť povedať, či sa vôbec zvyšok reťazca dá nejak korektne doplniť, a ak áno, tak potrebujeme vedieť, ktorý zo všetkých spôsobov doplnenia je lexikograficky najmenší.

Predpokladajme, že v časti, ktorú už máme, je x neuzavretých otváracích zátvoriek. Celý reťazec má párnu dĺžku, uzatvorené dvojice zátvoriek v hotovej časti reťazca majú párnu dĺžku, nedoplnený zvyšok reťazca má teda rovnakú paritu ako x. Ak nám ostalo menej ako x znakov, reťazec sa doplniť nedá – nevieme zavrieť všetky otvorené zátvorky. Ak nám ostalo x a viac znakov, reťazec sa doplniť dá – napr. v správnom poradí zavrieme všetky otvorené zátvorky a potom pridáme toľko `()` koľko treba.

Ktoré doplnenie je ale lexikograficky najmenšie? Zátvorky, ktoré doplniť musíme, majú presný počet aj poradie. Všetky sú ale zatváracie, a teda horšie ako otváracia zátvorka `(`. Chceme preto dať *najskôr* čo najviac `(`: ak nám ostáva $x + 2k$ znakov, tak si môžeme dovoliť k takýchto zátvoriek. No a zvyšok reťazca už je teraz vynútený: najskôr príde k kusov `)` a na záver x kusov zátvoriek, ktoré zavrú všetko čo ešte ostalo otvorené.

Ostáva už len jediné: ako vedieť povedať, aké je x a v akom poradí máme zatvárať otvorené zátvorky? Toto bude jednoduchejšie, ako by sme možno čakali. Ako ideme sprava doľava a postupne „gumujeme“ zátvorky, tak okrem iného odstraňujeme presne tie zatváracie zátvorky, ktoré tam potom budeme musieť naspäť pridať. Presnejšie, pridať budeme musieť tie, ku ktorým sme nevygumovali aj zodpovedajúcu otváraciu zátvorku.

Poradie, v ktorom treba zatvárať zátvorky naľavo, si vieme pekne pamätať v dátovej štruktúre zásobník. Ako prechádzame vstupný reťazec odzadu a hľadáme prvú vymeniteľnú zátvorku, do zásobníka si hádžeme zatváracie zátvorky, ktoré sme stretli. Pokiaľ natrafíme na nejakú otváraciu, môžeme uzatváraciu zátvorku, ktorú sme naposledy vložili do zásobníka, z neho vyhodiť. (Táto dvojica musela tvoriť pár.) V každom momente každej uzatváracej zátvorke na zásobníku zodpovedá práve jedna otváracia naľavo od našej aktuálnej pozície, a to v tom poradí, v akom ich máme na zásobníku.

Napríklad prechádzajme sprava reťazec `(()[()[()]])`. V prvých štyroch krokoch stretneme zatváraciu zátvorku a dáme ju na zásobník. V piatom kroku stretneme otváraciu zátvorku `(`, odoberieme teda z vrchu zásobníka jej pár. Na zásobníku nám ostali, zhora dole, `]`, `]` a `)`. V šiestom kroku by sme spracovali `[` a z vrchu zásobníka odobrali `]`. V siedmom kroku by sme na vrch zásobníka pridali `)`. V tejto chvíli nespracovaný začiatok reťazca vyzerá nasledovne: `(()[(`. No a na zásobníku máme práve tie pravé zátvorky, ktoré majú v nespracovanej časti svoju ľavú. Zhora dole sú to `)`, `]` a `)`.

Riešenie môžeme teda zhrnúť tak, že prechádzame vstupný reťazec od konca a snažíme sa nájsť prvú zátvorku, ktorá je vymeniteľná za lexikograficky väčšiu tak, aby bolo možné reťazec korektne douzatvárať. Narozdiel od predchádzajúceho riešenia si nepamätáme počet neuzatvorených zátvoriek naľavo ale si v pomocnej štruktúre pamätáme ako ich douzatvárať tak aby to viedlo ku korektnému výrazu. Časová a pamäťová zložitosť tohto riešenia ostáva $O(n)$ tak ako v predchádzajúcom riešení.

Listing programu (C++)

```
#include <iostream>
#include <stack>
#include <string>

using namespace std;

inline bool zatvara(char c) { return c == ')' || c == ']'; }
```

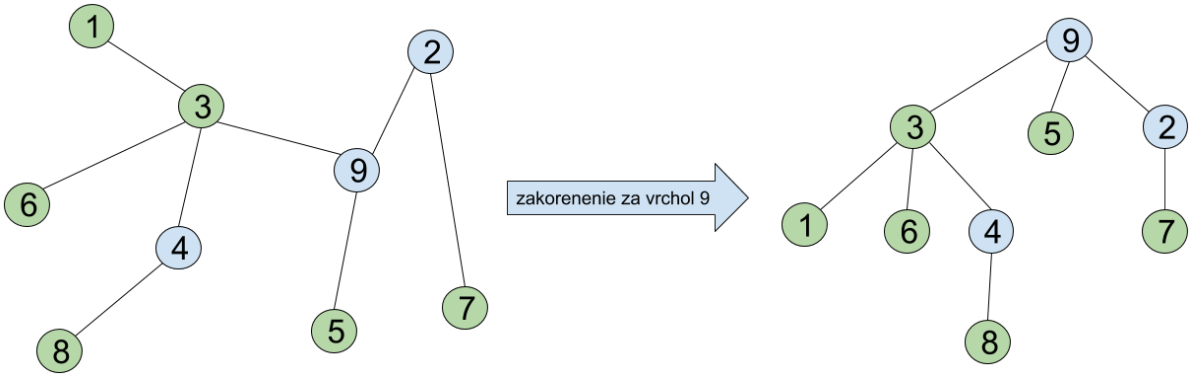
```
string next(string x)
{
  string porad = "()[]"; // lexikograficke poradie

  int n = x.size();
  stack<char> S;
  string out;
  for (int i = n - 1; i >= 0; --i)
  {
    if (zatvara(x[i])) // upravíme zásobník
      S.push(x[i]);
    else
      S.pop();
    for (int j = 0; j < 4; ++j)
      if (porad[j] > x[i])
      { // chceme väčší znak
        if (zatvara(porad[j]))
        {
          if (S.empty() || S.top() != porad[j])
            continue; // uzatvárajúca zátvorka nepasuje
          S.pop();
        }
        else
          S.push(porad[j + 1]);           // uzatvárajúca k Z[j]
        int k = n - 1 - i - S.size(); // veľkosť rezervy
        if (k >= 0)
        {
          out += x.substr(0, i); // zachovame prefix
          out += porad[j]; // prvy zmeneny znak
          out += string(k / 2, '(') + string(k / 2, ')'); // (((...()...))),
          while (!S.empty())
          {
            out += S.top();
            S.pop();
          }
          return out;
        }
        if (zatvara(porad[j]))
          S.push(porad[j]);
        else
          S.pop();
      }
  }
  return "-1";
}

int main()
{
    string vstup;
    cin >> vstup;
    cout << next(vstup) << "\n";
}
```

B-II-3 Sociálne dištančné preteky

V tejto úlohe ste pracovali so špecifickým grafom, ktorý sa nazýva *strom*. Práve stromami totiž nazývame súvislé grafy, ktoré neobsahujú žiadne cykly. Ľahko sa dá dokázať, že každý n-vrcholový strom má $n - 1$ hrán a že medzi každými dvoma jeho vrcholmi vedie práve jedna jednoduchá cesta. Stromy sa často zobrazujú v takzvanej zakorenenej verzii. Predstaviť si to môžete tak, že strom zavesíme za jeden vrchol a zvyšné popadajú pod neho.



Takéto zobrazenie použijeme aj my. Samotný strom sa síce nijakým spôsobom nezmení, takáto reprezentácia nám však pomôže uvažovať o úlohe z iného uhla a uľahčí nám riešenie. A aby sme si navzájom rozumeli, zaveďme si nasledovné označenie (zodpovedajúce teórii grafov) – námestia voláme *vrcholy*, uličky sú *hrany*.
Na vstupe zadaný strom sme teda zavesili za jeden z jeho vrcholov (je jedno za ktorý). Pozrime sa teraz na jeho samý spodok. Všimnime si, že sa tam nachádza niekoľko vrcholov, z ktorých sa už nedá dostať nižšie – vedie z nich jediná hrana, a to k vyššiemu vrcholu. Takéto vrcholy voláme listy. Ak v niektorom liste nemôže začínať alebo končiť bežecká trasa, nemusíme tento vrchol vôbec uvažovať. Žiadna bežecká trasa cezeň totiž nemôže prechádzať, lebo by nemala ako pokračovať ďalej.
Zaujímavé sú teda iba listy, ktoré sú vhodné ako jeden z koncov hľadanej trasy. Samozrejme, môže sa stať, že tento list vo finálnom riešení nebudeme chcieť použiť, to však zatiaľ nevieme, a preto môžeme uvažovať, čo sa stane ak by v riešení predsa len bol. Kadiaľ v takom prípade vedie bežecká trasa? Ako sme si povedali, z listu vedie práve jedna hrana (dohora), musí ísť teda po tejto hrane, ktorú si môžeme označiť.
Všetky listy sme teda vyriešili jednoznačne, pozrime sa teda o jednu úroveň vyššie. Tu nastáva niekoľko rôznych prípadov.

Žiadna z dodola vedúcich hrán nie je označená. V tomto prípade žiaden z pod ním ležiacich listov nemohol byť koncom bežeckej trasy a ísť do nich je úplne zbytočné. Pri tomto vrchole teda môžeme postupovať rovnako, ako keby bol sám listom. Ak v ňom môže končiť bežecká trasa, jediná možnosť ako vedie je dohora a preto si túto hranu označíme. A ak ani v ňom cesta končiť nemôžeme, môžeme ho ignorovať.

Práve jedna dodola vedúca hrana je označená a v tomto vrchole bežecká cesta končiť nemôže. V jednom liste mohla začínať bežecká trasa a nás teraz zaujíma, že ak by sme ju chceli použiť, kadiaľ by viedla. Zatiaľ sa dostala do tohto vrcholu a musíme sa rozhodnúť kam ju poslať ďalej. Všimnite si však, že v skutočnosti nemáme na výber. Vrátiť sa nemôžeme a ísť do iného listu je zbytočné, lebo v nich bežecká trasa končiť nemôže (inak aj ich hrana by bola označená) a iba by sme sa zasekli. Môžeme teda použiť iba jedinú hranu dohora, preto si ju označíme.

Označené sú aspoň dve dodola vedúce hrany, poprípade je označená práve jedna, ale v tomto vrchole môže končiť bežecká cesta. Táto možnosť sa vyznačuje tým, že ak by sme chceli, mohli by sme vytvoriť práve jednu celistvú bežeckú trasu. Ak dodola vedú aspoň dve označené hrany, môžeme si vybrať ľubovoľné dve a spojiť ich. Dostaneme tak bežeckú trasu z dvoch hrán, ktorá prechádza týmto vrcholom. Tak isto vieme uzavrieť trasu, ak je označená iba jedna hrana dodola ale samotný náš vrchol môžeme použiť ako jej druhý koniec.
Zároveň platí, z toho, čo vidíme v podstrome aktuálneho vrcholu, viac ako jednu trasu nevytvoríme. Všetky možné trasy totiž prechádzajú cez náš aktuálny vrchol a teda ak by sme ľubovoľne vyrobili dve, križovali by sa. Poslednou otázkou je, či sa nám oplatí túto jednu trasu vytvoriť. Čo ak by sme namiesto toho viedli bežeckú trasu ďalej dohora?
Uvedomme si však, že aj takto vieme získať len najviac jednu bežeckú trasu. Dohora vedie len jedna hrana a tou môže pokračovať len jedna trasa. Ak teda nevytvoríme **jednu** krátku bežeckú trasu hneď, najlepšie, čo sa nám môže stať, je, že niekedy v budúcnosti vytvoríme trasu dlhšiu, ale tiež iba **jednu**. Celkový počet teda bude rovnaký. Navyše, dlhšia trasa nám potenciálne môže zasiahnuť do iných trás a obmedziť ich celkový počet. Oplatí sa preto vytvoriť bežeckú trasu rovno teraz a tým pádom nechať hranu vedúcu dohora neoznačenú.

V tomto momente sme jednoznačne spracovali ďalšiu úroveň vrcholov a môžeme sa posunúť vyššie. Tam si však môžeme uvedomiť, že vyššie uvedené podmienky naďalej platia. V skutočnosti je totiž jedno, či dodola označená hrana vedie priamo do listu alebo len do nejakého nižšieho vrchola. Rovnakým spôsobom teda postupujeme až na úplný vrchol a pritom iba počítame, koľko bežeckých trás sme vytvorili.
Počas tohto riešenia raz spracujeme každý vrchol a raz každú hranu dodola z neho. Keďže hrán je dokopy $n-1$, je celková časová zložitosť takéhoto riešenia $O(n)$.

Spôsobov implementácie je viacero, ja som si vybral rekurzívny, keďže sa veľmi ľahko implementuje, lebo pomocou jedného prehľadávania zakorení strom a zároveň spočíta hľadané hodnoty. Funkcia `ries()` dostane ako parameter vrchol v a pre tento vrchol vypočíta dve hodnoty – koľko najviac bežeckých trás viem vytvoriť v podstrome pod vrcholom v a či je hrana vedúca dohora z vrcholu v označená.

Na to, aby som vedel tieto hodnoty vypočítať, potrebujem vedieť tieto isté odpovede pre všetky vrcholy priamo pod nimi. Následne spočítam všetky bežecké trasy a počet dohora označených hrán, ku ktorému pridám 1 ak vo vrchole v môžeme ukončiť trasu. Ak je tento počet 2 a viac, pridám do celkového výsledku pre podstrom v ďalšiu trasu a hranu nahor neoznačím, v opačnom prípade počet bežeckých trias nezvýšim a hranu označím podľa toho, či mám 0 alebo 1 označené hrany.

Listing programu (Python)

```
n = int(input())
graf = [[] for i in range(n)]
for _ in range(n-1):
    x, y = map(int, input().split())
    x, y = x - 1, y - 1
    graf[x].append(y)
    graf[y].append(x)

k = int(input())
vhodne = [False] * n
for x in input().split():
    vhodne[int(x) - 1] = True

navstivene = [False] * n

def ries(vrchol):
    navstivene[vrchol] = True
    dokopy = 0
    dohora = 0
    for sused in graf[vrchol]:
        if navstivene[sused]:
            continue
        podstrom = ries(sused)
        dokopy += podstrom[0]
        dohora += podstrom[1]
    if vhodne[vrchol]:
        dohora += 1
    if dohora > 1:
        dokopy += 1
        dohora = 0
    return (dokopy, dohora)

print(ries(0)[0])
```

B-II-4 Bitové operácie

Podúloha A: rotácia o tri doprava

Operácia `x shr 3` spraví skoro presne to, čo chceme: posunie všetky bity o 3 pozície doprava. Len teda najpravejšie tri bity stratíme a na najľavejšie tri pozície namiesto nich dostaneme nuly. Aby sme toto napravili, stačí si tieto tri bity zapamätať, posunúť ich úplne doľava a tam ich zase pridať.
Rotácia o tri teda môže vyzerať napr. nasledovne: `(x shr 3) or ( (x and 7) shl (B-3) )`

Podúloha B: test pre rozdiel dvoch mocnín dvojky

Ako vyzerá rozdiel dvoch mocnín dvojky zapísaný v dvojkovej sústave? Napríklad:

```
    000100000000
  - 000000010000
================
    000011110000
```

Ľahko nahliadneme, že takýto vzor tam budeme mať vždy: čísla, ktoré nás zaujímajú, sú práve tie čísla, v ktorých existuje práve jeden neprázdny úsek jednotkových bitov.
Formálne, ľahko overíme, že $2^x - 2^y = 2^y + 2^{y+1} + \cdots + 2^{x-1}$.
(Túto rovnosť môžeme čítať oboma smermi: rozdiel ľubovoľných dvoch mocnín dvojky je rovný súčtu nejakých po sebe idúcich mocnín dvojky a naopak, každý takýto súčet je rovný rozdielu nejakých dvoch mocnín dvojky.)

Kontrolu, či máme číslo takéhoto tvaru, môžeme spraviť napríklad nasledovne:

- Nula nemá požadovaný tvar, ak máme nulu, rovno odpovieme, že je zlá.
- Nášmu číslu zmeníme mu všetky nuly za poslednou jednotkou na jednotky. Naďalej platí, že dobré čísla sú práve tie, v ktorých všetky jednotky tvoria jeden súvislý úsek.

  (Ak sme začínali s dobrým číslom s hodnotou $2^y - 2^z$, máme teraz číslo s hodnotou $2^y - 1$.)
- Pripočítame 1. Tým zmeníme na konci čísla úsek tvaru „01111“ na úsek tvaru „10000“.

  Z dobrého čísla sme práve dostali mocninu dvoch (alebo nulu ak pripočítanie jednotky spôsobilo pretečenie). Zo zlého čísla sme dostali číslo, ktoré má naďalej aspoň dve jednotky.

Ukážková implementácia v Pythonu-podobnom pseudokóde:

```
def je_rozdiel_mocnin_dvojky(x):
    if x == 0:
        return False
    x = x or (x-1)
    x += 1
    if x == 0: return True
    return (x and (x-1)) == 0
```

Podúloha C: určenie exponentov

Ak sme začali z čísla x, ktoré zaručene bolo tvaru $2^y - 2^z$, tak sme na konci riešenia predchádzajúcej podúlohy v premennej x dostali hodnotu 2^y. Podobne vieme izolovať aj hodnotu 2^z, stačí si zapamätať pôvodnú hodnotu x a odčítať ju od 2^y. Jediný krok, ktorý nám chýba, je dvojkový logaritmus: z čísla tvaru 2^k zistiť hodnotu k. Implementácia tejto časti riešenia:

```
def zisti_exponenty(x):
    dva_na_y = (x or (x-1)) + 1
    dva_na_z = dva_na_y - x
    y = log2(dva_na_y)
    z = log2(dva_na_z)
    return (y, z)
```

No a myšlienku toho, ako implementovať logaritmus, sme už stretli v riešeniach domáceho kola: použijeme binárne vyhľadávanie. Jednu stručnú implementáciu uvádzame nižšie:

```
def log2(k):
    # predpokladame ze k > 0 je nejaka mocnina dvoch
    lo, hi = 0, B
    # plati invariant, ze 2^lo <= k < 2^hi
    while hi - lo > 1:
        med = (lo + hi) // 2
        if (1 shl med) <= k:
            lo = med
        else:
            hi = med
    return lo
```

Takéto riešenie má zjavne časovú zložitosť $\Theta(\log b)$, čiže logaritmickú od počtu bitov v registri.

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Andrej Korman
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2021