B-I-1 Jamy a rebríky

Začneme jednoduchým pozorovaním: Každá hra Jám a rebríkov má nanajvýš $n-1$ hodov kockou. Po každom hode kockou totiž hráč buď spadne do jamy, alebo sa posunie dopredu. Tu je dôležité, že každý rebrík má koniec na väčšom čísle ako začiatok, takže aj ak prejdeme po rebríku, číslo políčka, kde skončíme, je určite väčšie ako to, kde sme začínali.

Teraz sa pozrieme na riešenie, ktoré nie je najefektívnejšie možné, ale zato je zjavne správne. V ňom budeme akoby postupne simulovať naraz všetky priebehy hry.
Na začiatku (po nula hodoch kockou) sa môžeme nachádzať len na políčku číslo 1.
Teraz budeme postupne simulovať jednotlivé hody kockou. Povedzme, že už sme spracovali i hodov kockou a vieme, že naša figúrka sa nachádza na jednom z políčok v množine M_i. (Na začiatku teda máme $M_0 = \{1\}$.)
Čo sa stane v nasledujúcom hode kockou? Postupne sa pozrieme na všetky možnosti, kde sa figúrka nachádzala a čo padlo na kocke. Pre každú možnosť zistíme, kde figúrka skončí (po tom, ako prípadne prejde po rebríku), a takto dostaneme všetky možnosti, kde sa môžeme nachádzať po $i+1$ hodoch kockou.
Akonáhle sa prvýkrát dostaneme na políčko n, vieme odpoveď a môžeme skončiť. A naopak, akonáhle pre nejaké i dostaneme, že $M_i = \emptyset$ (čiže už nemôžeme byť nikde), vieme, že kvôli jamám sa na políčko n nijako nedá dostať.
Pre každý hod kockou prejdeme nanajvýš $6n$ možností – mohli sme byť na nanajvýš n rôznych políčkach a pre každé z nich máme šesť možností, čo padlo na kocke. No a už vieme, že každá hra má nanajvýš $n-1$ hodov kockou, takže po odsimulovaní $n-1$ hodov už musel nastať jeden alebo druhý možný koniec.
Takéto riešenie má teda v najhoršom prípade časovú zložitosť kvadratickú od počtu políčok, teda $\Theta(n^2)$.

Listing programu (Python)

```
N = int(input())
plan = [ int(_) for _ in input().split() ]

kde_viem_byt = set([0])
for krokov in range(1,N):
    kde_budem_po_tahu = set()

    for kde_som in kde_viem_byt:
        for kocka in [1,2,3,4,5,6]:
            ciel = kde_som + kocka
            if ciel > N-1: continue
            if plan[ciel] == -1: continue
            ciel += plan[ciel]
            if plan[ciel] == -1: continue
            kde_budem_po_tahu.add(ciel)

    kde_viem_byt = kde_budem_po_tahu

    if N-1 in kde_viem_byt:
        print(krokov)
        break
    if len(kde_viem_byt) == 0:
        print(-1)
        break
```

Vyššie popísané riešenie vieme vylepšiť na riešenie v lineárnom čase pomocou jedného dodatočného pozorovania: každé políčko má zmysel spracúvať len raz. Ak už sme vedeli byť na políčku 17 po štyroch hodoch kockou a neskôr zistíme, že na políčku 17 vieme byť aj po šiestich hodoch kockou, nemá zmysel znova pozerať, kam sa odtiaľ vieme pohnúť ďalej. Všetky tieto možnosti sme už predsa v minulosti prezreli.
Pre tých, ktorí už poznáte teóriu grafov, doplníme, že takto vylepšené riešenie vlastne predstavuje *prehľadávanie do šírky* v grafe, ktorého vrcholy predstavujú, kde sa práve figúrka nachádza, a ktorého orientované hrany zodpovedajú tomu, ako sa figúrka môže pohnúť po jednom hode kockou.

Jednoduchšie lineárne riešenie

Existuje však aj riešenie s lineárnou časovou zložitosťou, ktoré je ešte jednoduchšie na implementáciu.
Toto riešenie bude založené na jednoduchej myšlienke: postupne pôjdeme hracím plánom „sprava doľava" (t.j. začneme na políčku n a postupne pôjdeme na políčka s čoraz menším číslom) a postupne pre každé políčko zistíme, na najmenej koľko hodov kockou sa vieme z neho dostať do cieľa.

Ako a prečo toto môžeme spraviť?
Majme konkrétne políčko $x < n$. Ak sme na tomto políčku, ešte nie sme v cieli. Musíme teda hodiť kockou. Na kocke padne nejaká hodnota od 1 po 6. Ak padne hodnota i, znamená to, že sa posunieme na políčko $x + i$, možno padneme do jamy, možno odtiaľ vylezieme po rebríku, a tam možno padneme do jamy. Tak či onak vieme vypočítať, či a kde skončíme po tomto hode kockou.
Takto teda dostaneme nanajvýš šesť možností, kam sa vieme dostať prvým hodom kockou. Všetky tieto políčka majú čísla väčšie ako x, čo znamená, že už sme ich skôr spracovali. A teda pre každé z nich vieme, na koľko hodov kockou sa z neho dá dostať do cieľa hry.
Ako teda vyzerá najlepšie možné riešenie pre políčko x? Spomedzi možností, ktoré máme, si pochopiteľne vyberieme tú najlepšiu – teda tú, kde nám na dokončenie hry stačí najmenší počet hodov.
V nižšie uvedenom programe si vypočítané hodnoty pamätáme v poli D. Špeciálna hodnota INF (nekonečno) sa použije vtedy, ak sa z nejakého políčka kvôli dieram vôbec nevieme dostať do cieľa. Takáto hodnota je potom vždy horšia ako ľubovoľná možnosť ako sa konečným počtom hodov kockou dostať do cieľa.

Listing programu (Python)

```
INF = 1023456789

n = int(input())
plan = [int(x) for x in input().split()]

D = [INF for i in range(n)]
if plan[-1] >= 0: D[-1] = 0

for i in reversed(range(n-1)):
    if plan[i] < 0:
        continue
    for j in range(i+1, i+7):
        if j < n:
            D[i] = min(D[i], D[j+plan[j] if plan[j]>0 else j] + 1)


print(D[0] if D[0] < INF else -1)
```

(Technický detail: V oboch programoch políčka plánu číslujeme od 0 po $n - 1$ namiesto od 1 po n.)

B-I-2 Zaujímavé faktoriály

Súťažnú úlohu sa dalo riešiť postupným zlepšovaním základného riešenia. Na dva body stačí vedieť faktoriály počítať, na druhé dva body potrebujeme vedieť efektívne povedať, koľkými nulami končí konkrétne $x!$. Nasledujúce tri body sú za podúlohu, ktorá je zároveň pomocným krokom ku kompletnému riešeniu. V takomto poradí si tieto riešenia teraz predstavíme.

Na riešenie prvých dvoch sád a získanie dvoch bodov stačilo riešenie priamočiaro naprogramovať. Pokiaľ váš jazyk podporuje násobenie veľkých čísel (napr. Python) stačilo prejsť daný rozsah od začiatku po koniec, vypočítať faktoriál a potom napríklad dané číslo premeniť na reťazec, z ktorého už ľahko vieme zistiť počet núl na jeho konci. Ak násobenie veľkých čísel váš jazyk nepodporuje, je možné aj toto riešenie zreplikovať za cenu nepríjemností s implementáciou veľkých čísel a násobenia.

Listing programu (Python)

```
import math

def pocetNul(x):
    xStr = str(x)[::-1]
    for i in range(len(xStr)):
        if xStr[i] != '0':
            return i

a, b = int(input()), int(input())
sum = sum(pocetNul(math.factorial(i)) for i in range(a, b+1))
print(sum)
```

Ak chceme vyriešiť túto úlohu na plný počet bodov, musíme si uvedomiť, že zatiaľ máme dve pomalé časti algoritmu a to zisťovanie počtu núl v čísle $x!$ a prechádzanie intervalu, ktorý môže byť obrovský.

Vo vstupoch 3 a 4 je však samotný interval malý, takže sa najskôr zameriame na zrýchlenie zisťovania počtu núl na konci čísla $x!$.
Je zjavné, že keď číslo vynásobíme desiatimi, pribudne mu nula na konci. Z toho ľahko odvodíme, že počet núl na konci čísla je rovný najväčšej mocnine čísla 10, ktorou je bezo zvyšku deliteľné.
Zaujímavé môže byť napr. odsimulovať si, ako sa vyvíja počet núl na konci prvých faktoriálov: 1!, 2!, 3!, a tak ďalej. Jednoduché pozorovanie je, že počet núl nebude klesať. Ak je totiž číslo $x!$ deliteľné 10^k, číslo $(x+1)! = (x+1) \cdot x!$ je určite tiež deliteľné aspoň 10^k.
Pozorovaním prvých faktoriálov vidíme, že počet núl narastie vždy práve pri faktoriáloch, ktorých základ je násobkom piatich: 5! je prvý faktoriál končiaci nulou, 10! je prvý končiaci dvoma nulami.
Prečo sa to deje práve vtedy? Desať je dvakrát päť, najväčšia mocnina 10 ktorá delí $x!$ teda závisí od toho, koľko dvojok a koľko pätiek máme v prvočíselnom rozklade $x!$.
Ľahké pozorovanie v prípade faktoriálov nás dovedie k tomu, že počet pätiek bude vo všetkých rozkladoch menšie alebo rovné ako počet dvojok. (V súčine $1 \cdot 2 \cdot 3 \cdot 4 \cdot$ je každé druhé číslo deliteľné dvoma, každé štvrté štyrmi, atď., zatiaľ čo len každé piate je deliteľné piatimi, každé dvadsiate piate deliteľné 5^2, atď.)
Preto platí, že počet núl na konci $x!$ je rovný najvyššej mocnine 5, ktorá toto číslo delí. Každý násobok 5 pridá do prvočíselného rozkladu $x!$ jednu päťku. Každý násobok 25 pridá aj druhú. Každý násobok 125 aj tretiu. A tak ďalej. Toto nám dáva efektívny spôsob ako zistiť, koľko núl má na konci konkrétne $x!$. Pomocou neho vieme napísať riešenie, ktoré získa 4 body:

Listing programu (C++)

```
#include <iostream>

using namespace std;

long long spocitaj_nuly(int x)
{
  // m obsahuje aktualnu mocninu 5
  long long m = 5, pocet_patiek = 0;
  // pripocitava 5ky pre kazdy nasobok m
  while (m <= x)
  {
    pocet_patiek += x / m;
    m *= 5;
  }
  return pocet_patiek;
}

int main()
{
  int a, b;
  long long suma = 0;
  cin >> a >> b;
  for (int i = a; i <= b; ++i)
  {
    suma += spocitaj_nuly(i);
  }
  cout << suma << "\n";
  return 0;
}
```

V nasledujúcich vstupoch je začiatok intervalu na nule, čiže hľadáme odpoveď na otázku, aký je celkový počet núl na konci číslach $0!, 1!, 2!, \dots, b!$.
Kľúčové je spozorovať, ako sa zmení počet núl medzi $(x-1)!$ a $x!$. Už vieme, že narastie práve vtedy, ak je x násobkom piatich. A presnejšie, narastie o toľko, akou najvyššou mocninou piatich je x deliteľné.
Pozerajme sa postupne na jednotlivé mocniny 5. Ak by každý násobok 5 prispieval len jednou päťkou, videli by sme nasledujúce počty núl: nula núl medzi 0! až 4!, jednu nulu vo faktoriáloch 5! až 9!, dve nuly od 10! až po 14!, a tak ďalej. Odpoveďou by teda bol nasledujúci súčet: $0+0+0+0+0 \;+\; 1+1+1+1+1 \;+\; 2+2+\cdots$.
Teraz sa pozrime na násobky 25. Každý z nich prispieva aj druhou päťkou. Dostávame teda ďalšiu jednu nulu v každom z faktoriálov 25! až 49!, dve nuly v 50! až 74!, atď.
Postupne pre každú mocninu $m = 5^i$ takto dostaneme súčet, v ktorom je najskôr m núl, potom m jednotiek, potom m dvojek, a tak ďalej. Keď vyberieme m pred zátvorku, v zátvorke ostane súčet $(0+1+2+3+\cdots)$. No a to je dobre známa aritmetická postupnosť, ktorú vieme v konštantnom čase sčítať pomocou známeho vzorca: ak je posledný sčítanec p, súčet je $p(p+1)/2$.
(Vo všeobecnosti bude platiť, že každá hodnota *okrem poslednej* sa v našom súčte vyskytuje presne m-krát.

Poslednú hodnotu preto ošetríme samostatne a až na tie od nej menšie použijeme práve odvodený vzorec.)

No a k plnému 10-bodovému riešeniu už chýba len jeden krok: ak máme úlohu riešiť pre faktoriály od $a!$ po $b!$, stačí dvakrát použiť vyššie popísaný postup. Najskôr zistíme, koľko núl majú na konci faktoriály od 0! po $b!$ a potom zistíme, koľko z nich nechceme – teda koľko majú na konci núl faktoriály od 0! po $(a-1)!$.

Listing programu (C++)

```
#include <iostream>

using namespace std;

long long helper(int x)
{
  long long total = 0;

  for (long long d = 5; d <= x; d *= 5)
  {
    int n = (x / d);
    long long to_add = d * n * (n + 1) / 2;
    to_add -= (d - 1 - x % d) * n;
    total += to_add;
  }
  return total;
}

int main()
{
  int a, b;
  cin >> a >> b;
  long long res = helper(b);
  if (a != 0)
    res -= helper(a - 1);
  cout << res << "\n";
  return 0;
}
```

Listing programu (Python)

```
def solve(a):
    res = 0
    power = 5
    while power <= a:
        top = (a - power + 1) // power
        res += power * top * (top + 1) // 2
        res += (top + 1) * (a - (top + 1) * power + 1)
        power *= 5
    return res

a = int(input())
b = int(input())
print(solve(b) - solve(a - 1))
```

B-I-3 Domáca úloha z elektrotechniky

Našou úlohou bolo zistiť, či vieme usporiadať zadanú postupnosť čísel od najmenšieho po najväčšie v prípade, že môžeme vymieňať iba prvky, ktoré sú na pozíciách k a $k+2$ (pre $1 \leq k \leq n-2$). Keď si vyskúšame na papieri spraviť niekoľko výmien, veľmi rýchlo prídeme k najdôležitejšiemu pozorovaniu celej úlohy. Bez ohľadu na výmeny, ktoré spravíme **nevieme zmeniť paritu pozície** prvku. Pozície k a $k+2$ majú totiž rovnakú paritu – buď sú obe párne alebo obe nepárne.
Tým pádom môžeme vstupnú postupnosť rozdeliť na dve časti – prvky na párnych pozíciách a prvky na nepárnych pozíciách. Dostaneme tak dve postupnosti polovičnej dĺžky, ktoré sú pri výmenách nezávislé jedna od druhej. V týchto skrátených postupnostiach následne môžeme vymieňať ľubovoľné **susedné** prvky. Prvok medzi nimi je totiž v druhej postupnosti.
A aj keď v týchto kratších postupnostiach spravíme nejaké výmeny, vieme ich spätne spojiť a zistiť, ako by po týchto výmenách vyzerala pôvodná postupnosť. Stačí, že prvky z týchto dvoch postupností dáme na striedačku. Otázkou teda je, či je možné spraviť výmeny v týchto postupnostiach tak, aby bol výsledok ich spojenia usporiadaný. Uvedomme si však, že ak máme **usporiadanú postupnosť, usporiadaná je aj každá jej pod-**

postupnosť. To znamená, že jediná možnosť ako získať usporiadanú postupnosť je usporiadať dve kratšie podpostupnosti.

Zhrňme si teraz naše riešenie. Najprv rozdelíme vstupnú postupnosť na dve časti – čísla na párnych a nepárnych pozíciách. Tieto dve postupnosti usporiadame a na striedačku pospájame späť. Nakoniec už len jedným prechodom overíme, či je výsledná postupnosť správne usporiadaná. Vieme totiž, že usporiadaná postupnosť vie vzniknúť iba z dvoch usporiadaných postupností, neplatí však, že dve usporiadané postupnosti musia vytvoriť usporiadanú postupnosť.

Na koniec nám už len ostáva dokázať, že pomocou výmen dvoch susedných prvkov vieme usporiadať ľubovoľnú postupnosť. Môžeme sa odvolať na zadanie, ktoré to explicitne spomína, samotný dôkaz je však pomerne triviálny. Keď si totiž nájdeme najmenší prvok, vieme ho poposúvať na úplný začiatok. Následne nájdeme druhý najmenší prvok a poposúvame ho na druhú pozíciu, čím si zároveň určite nepokazíme prvú pozíciu. A toto opakujeme.

Na záver ešte dodajme, že naše riešenie vôbec nepotrebuje robiť takéto pomalé triedenie, ale môže využiť ľubovoľný algoritmus na to určený, keďže nás nezaujíma spôsob ako postupnosť usporiadať, ale iba to, či je to možné. Ak teda použijeme nejaké knižničné riešenie, dostaneme riešenie so zložitosťou $O(n \log n)$.

Listing programu (Python)

```python
n = int(input())
prvky = list(map(int, input().split()))

# rozdelime postupnost na dve mensie
parne_pozicie = []
neparne_pozicie = []
for i in range(n):
    if i % 2 == 0:
        parne_pozicie.append(prvky[i])
    else:
        neparne_pozicie.append(prvky[i])

# usporiadame kratsie postupnosti
parne_pozicie.sort()
neparne_pozicie.sort()

# spojime kratsie postupnosti dokopy
vysledok = []
for i in range(n):
    if i % 2 == 0:
        vysledok.append(parne_pozicie[i // 2])
    else:
        vysledok.append(neparne_pozicie[i // 2])

# skontroluj usporiadanost vyslednej postupnosti
usporiadane = True
for i in range(n-1):
    if vysledok[i] > vysledok[i+1]:
        usporiadane = False

print('ano' if usporiadane else 'nie')
```

B-I-4 Školský výlet

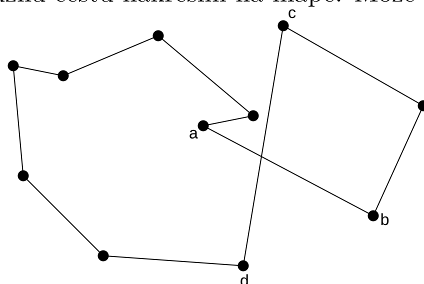
Súťažnú úlohu v odborných textoch nájdete pod menom *Problém obchodného cestujúceho*. Ide o tzv. NP-ťažký optimalizačný problém. To znamená, že ani pre jednoduchšiu verziu tejto úlohy (pre dané s rozhodnúť, či sa dá celý školský výlet zrealizovať za nanajvýš s peňazí) nie je známy žiadny algoritmus s polynomiálnou časovou zložitosťou. A naopak, poznáme viacero dôvodov, kvôli ktorým sa domnievame, že žiaden takýto algoritmus ani neexistuje. Bolo si treba poradiť ináč.

V riešení budeme používať terminológiu z teórie grafov. V zadaní sme dostali graf, ktorého vrcholmi sú jednotlivé slovenské okresné mestá a hranami priame cesty medzi nimi. V tomto grafe hľadáme najlacnejšiu *kružnicu*, ktorá navštívi každý vrchol práve raz. Takúto kružnicu voláme *Hamiltonovská kružnica*. (Formálne, kružnica je cyklická postupnosť vrcholov, v ktorej sú každé dva po sebe idúce vrcholy prepojené hranou.)

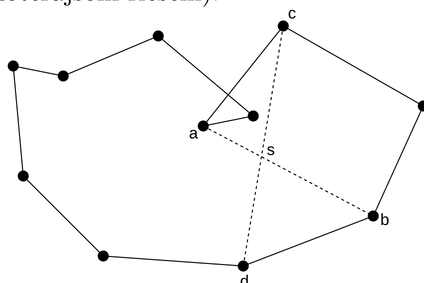
Ak stačí navštíviť práve raz každý vrchol grafu a už sa netreba z posledného vrcholu vrátiť späť na štart, hovoríme o *Hamiltonovskej ceste*.

Heuristické prístupy

Nájsť *nejaké dobré* riešenie našej úlohy je výrazne ľahšie ako nájsť to *úplne optimálne*.
Pri hľadaní dobrého riešenia môžeme k nášmu problému pristupovať geometricky. Reálne ceny cestovného sú približne priamo úmerné vzdušnej vzdialenosti medzi lokalitami. Zamyslime sa preto nad tým, ako by sa naša úloha dobre riešila v ideálnom svete, v ktorom platí presná rovnosť – teda ak za cenu cesty z a do b môžeme priamo považovať ich Euklidovskú vzdialenosť.
Predstavme si, že sme si nejakú okružnú cestu nakreslili na mape. Môže to celé vyzerať napríklad takto:



Už z pohľadu na tento obrázok by nám malo byť jasné, že nakreslená cesta nemôže mať optimálnu cenu. Zjavným problémom sú dva úseky, ktoré sa križujú.
Križovanie úsekov vieme vždy odstrániť: namiesto presunov ab a cd použijeme ac a bd (a úsek medzi b a c prejdeme v opačnom smere ako pri doterajšom riešení).



Vo všeobecnosti nám takáto zmena môže vyrobiť nové križovania, to ale nevadí (keďže celý postup vieme opakovať a zbaviť sa tak aj tých). Dôležité je, že každá takáto zmena skráti celkovú dĺžku cesty.
Pozrime sa na štvoruholník $acbd$. Namiesto oboch jeho uhlopriečok (prvé riešenie) používame dve jeho protiľahlé strany (druhé riešenie). Súčet dĺžok dvoch strán je vždy menší ako súčet dĺžok uhlopriečok. (Dôkaz: trojuholníková nerovnosť: pozrite sa zvlášť na trojuholník acs a zvlášť na bds.)

V tejto chvíli môžeme zase zabudnúť na celú geometriu a uvedomiť si, že sme vlastne práve objavili všeobecnejšie pravidlo, ktoré môžeme používať úplne vždy, dokonca bez ohľadu na to, či naše ceny hrán spĺňajú trojuholníkovú nerovnosť:

Zlepšovacie pravidlo. Majme nejakú Hamiltonovskú kružnicu. Ak na nej existujú dve disjunktné hrany ab a cd také, že súčet cien ab a cd je väčší ako súčet cien ac a bd, vieme nájsť lacnejšiu Hamiltonovskú kružnicu tak, že pôvodné dve hrany nahradíme novými a úsek medzi b a c prejdeme v opačnom smere.

No a keď máme zlepšovacie pravidlo, ľahko pomocou neho navrhneme aj pomerne priamočiary algoritmus: začneme z ľubovoľnej Hamiltonovskej kružnice a dokola na ňu budeme používať naše pravidlo, až kým nedostaneme takú, ktorá sa už zlepšiť nedá.
Pozor, **neplatí**, že keď dostaneme riešenie, ktoré už naše pravidlo nevie zlepšiť, tak ide o najlepšie možné riešenie. Opak je pravdou – v praxi bude existovať množstvo navzájom rôzne dobrých riešení, ktoré toto pravidlo ale nevie zlepšiť. (Na zlepšenie takýchto riešení treba väčšie štrukturálne zmeny.)
Ešte stále však takýmto postupom vieme nájsť veľmi dobré riešenie – obzvlášť ak celý postup niekoľkokrát zopakujeme, pričom zakaždým začneme z iného náhodne zvoleného riešenia. Spomedzi všetkých riešení si potom samozrejme vyberieme to najlepšie, ku ktorému sa nám podarilo dostať.

Oplatí sa ešte pridať jednu dobrú radu: vždy, keď vieme zlepšovacie pravidlo použiť rôznymi spôsobmi, vyberieme si ten, ktorý riešenie zlepší najviac.
Takejto technike hľadania približných dobrých riešení hovoríme *technika lezenia na kopec* (anglicky *hill climbing*). Názov pochádza z nasledujúcej metafory: každú malú zmenu riešenia si predstavíme ako krok, ktorý môžeme spraviť po krajine. Každému riešeniu zodpovedá nejaká nadmorská výška, ktorá predstavuje sumu ušetrenú oproti riešeniu, z ktorého sme začínali. Naše pravidlo nám hovorí, ako robiť kroky hore kopcom, a dobrá rada nám hovorí, že z krokov hore kopcom si vyberáme vždy ten najstrmší. Takto postupne dokráčame na nejaký vrch kopca, teda na riešenie, ktoré je *lokálnym (aj keď nie nutne globálnym) optimom*.
Nižšie uvádzame príklad implementácie takéhoto riešenia v jazyku Python.
Uvedieme ešte, že existujú aj pokročilejšie verzie tejto optimalizačnej techniky, ktoré v praxi zvyknú rýchlejšie dosahovať lepšie výsledky. Príkladom takejto pokročilejšej techniky je tzv. *simulované žíhanie* (anglicky *simulated annealing*).

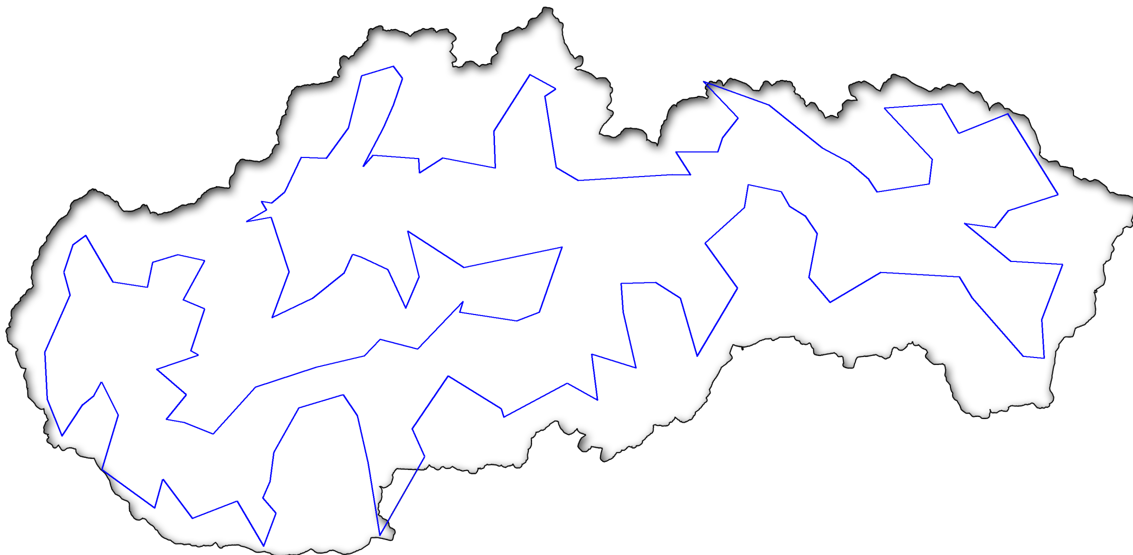
Listing programu (Python)

```python
from random import shuffle

RESTARTS = 10 # kolkokrat chceme zacat uplne odznova?

# pomocna funkcia: pre dane pole cien a plan vyletu spocitaj celkovu cenu

def total(ceny, plan):
    return sum( ceny[i][j] for i, j in zip( plan, plan[1:]+[plan[0]] ) )

# nacitame mena miest a ceny

f = open('vzdialenosti-final.csv')
nazvy_miest = f.readline().strip().split(',')[1:]
N = len(nazvy_miest)
ceny = [ [ 0 for b in range(N) ] for a in range(N) ]
for n in range(N):
    ceny_z_n = [ int(x) for x in f.readline().split(',')[1:] ]
    for idx, dist in enumerate(ceny_z_n):
        ceny[n][idx] = ceny[idx][n] = dist

# zoberieme postupnost miest s Lucencom na zaciatku ako zakladne riesenie

lucenec = nazvy_miest.index('Lucenec')
nie_lucenec = [ x for x in range(N) if nazvy_miest[x] != 'Lucenec' ]

najlepsi_vylet = [lucenec] + nie_lucenec
najlepsia_cena = total(ceny, najlepsi_vylet)

for restart in range(RESTARTS):
    # vyrobime novu nahodnu postupnost miest
    shuffle(nie_lucenec)
    aktualny_vylet = [lucenec] + nie_lucenec
    aktualna_cena = total(ceny, aktualny_vylet)

    # dokola ju zlepsujeme, kym to ide
    while True:
        # prejdeme vsetky moznosti ako vymenit dve hrany, zapamatame si najlepsiu
        novy_vylet, nova_cena = None, None
        for i in range(1, len(aktualny_vylet)):
            for j in range(i+2, len(aktualny_vylet)+1):
                skusam_vylet  = aktualny_vylet[:i]
                skusam_vylet += list(reversed( aktualny_vylet[i:j] ))
                skusam_vylet += aktualny_vylet[j:]
                skusam_cena = total(ceny, skusam_vylet)
                if nova_cena is None or skusam_cena < nova_cena:
                    novy_vylet, nova_cena = skusam_vylet, skusam_cena

        #ak sa uz nevieme pohnut "hore kopcom", koncime
        if nova_cena >= aktualna_cena: break

        # posunieme sa hore kopcom
        # ak sme nasli doteraz najlepsie riesenie, zapamatame si ho
        aktualny_vylet, aktualna_cena = novy_vylet, nova_cena
        if aktualna_cena < najlepsia_cena:
            najlepsi_vylet, najlepsia_cena = aktualny_vylet, aktualna_cena
            print(f'beh {restart}: nova najlepsia cena {najlepsia_cena}')

# vypiseme najlepsi najdeny vylet
for x in najlepsi_vylet: print( nazvy_miest[x] )
```

Exaktné riešenie

Dáta vo veľkosti, ktorá bola použitá v súťažnej úlohe, sú ešte dostatočne malé na to, aby sme pokročilými technikami vedeli nájsť aj skutočne optimálne riešenie. Záujemcov o problematiku odkazujeme na archív Korešpondenčného seminára z programovania: ročník 31, kolo 4, úloha 5.

Nami nájdená optimálna cesta po Slovensku je schematicky znázornená na obrázku nižšie. (Polohy miest sú len približne správne a namiesto ciest sú jednotlivé presuny kreslené len ako priame úsečky v 2D.)



TRIDSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

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