B-II-1 Šachoví králi

Ak by mal Šachistan len dva riadky, bolo by rozmiestňovanie kráľov ľahké. Stačilo by ísť zľava doprava a vždy, keď stretneme stĺpec, kam môžeme dať kráľa, doň jedného umiestnime. Je jedno, do ktorého riadku ho dáme, lebo tak či tak nám zablokuje celý nasledujúci stĺpec.
Toto už nie je pravda, ak máme tri riadky. Kráľ v hornom riadku zablokuje len horné dve políčka v nasledujúcom stĺpci. A tiež zablokuje v svojom stĺpci len stredné políčko, takže teoreticky je možné dať ďalšieho kráľa už do toho istého stĺpca, alebo do nasledujúceho.
K optimálnemu riešeniu sa ešte stále dá dopracovať *pažravými* úvahami, je to ale výrazne komplikovanejšie. Nie vždy sa napríklad oplatí dávať kráľa čo najviac doľava. Uvažujme napríklad takúto situáciu:

```
XOX...
OOX...
XOX...
```

Ak dáme kráľa hneď do prvého stĺpca, už nemôžeme dať žiadnych kráľov do druhého. Lepšie je nechať prvý stĺpec prázdny a dať dvoch kráľov do druhého stĺpca.
Rozboru prípadov sa však vieme šikovne vyhnúť tak, že pomocou techniky *dynamického programovania* postupne prezrieme všetky možnosti a vyberieme si najlepšiu z nich.
Začnime najľavejším stĺpcom. Existuje nanajvýš 5 spôsobov, ako doň umiestniť kráľov: nedáme tam žiadneho, dáme tam práve jedného do niektorého riadku, alebo tam dáme dvoch: do prvého a tretieho riadku. Pre každý z týchto piatich spôsobov nás zaujíma, koľko najviac kráľov vieme umiestniť do zvyšku krajiny. Ak by sme vedeli zodpovedať týchto päť otázok, vedeli by sme si z týchto piatich možností vybrať najlepšiu a tak vyriešiť našu úlohu.
Pozrime sa teda na konkrétnu otázku tohto typu. Pre nejaký stĺpec krajiny máme povedané, ako sme doň umiestnili kráľov, a pýtame sa, koľko najviac ďalších kráľov vieme umiestniť napravo od tohto stĺpca.
Ako takúto otázku zodpovieme? Zopakujeme rovnakú úvahu. Ak už vieme, kde sú králi v stĺpci s, pozrieme sa na stĺpec $s+1$. Niektoré jeho políčka sú zakázané – buď priamo vstupom, alebo kráľom v stĺpci s. Vyskúšame postupne všetky možnosti, ako rozmiestniť kráľov na tie políčka v stĺpci $s+1$, ktoré ostali nezakázané. Ktorá z týchto možností je najlepšia? Na to potrebujeme zakaždým vedieť, ako najlepšie vieme kráľmi vyplniť zvyšné stĺpce. Opäť sa teda budeme pýtať otázky toho istého typu, len tentokrát má oblasť, na ktorú sa pýtame, o stĺpec menej.
Dokopy existuje len nanajvýš $5n$ otázok tohto typu – každá otázka je jednoznačne určená počtom stĺpcov, ktoré ešte ostávajú nespracované, a rozmiestnením kráľov v poslednom spracovanom stĺpci.
No a hlavná pointa celej tejto techniky je uvedomiť si, že tieto otázky vieme postupne všetky zodpovedať *ak to budeme robiť v správnom poradí*. Na začiatku vieme zodpovedať otázky o poslednom stĺpci šachovnice – napravo od neho nič nie je, takže máme vždy len tých kráľov, ktorých dáme doň. Keď už vieme odpovede na tieto otázky, vieme zodpovedať otázky o predposlednom stĺpci. A tak ďalej. Postupne prejdeme vstupom sprava doľava, až sa úplne na koniec dostaneme k tým piatim otázkam, ktoré nás zaujímali na začiatku. Ich zodpovedaním vyriešime našu úlohu.
Máme lineárny počet otázok. Každú z nich vieme (pomocou už zistených odpovedí na otázky o nasledujúcom stĺpci) zodpovedať v konštantnom čase. Vyššie popísané riešenie má preto časovú zložitosť lineárnu od n.

Listing programu (Python)

```
N = int( input() )
MAPA = [ input() for _ in range(3) ]
TYPY = [ [], [0], [1], [2], [0,2] ] # do ktorych riadkov dat kralov v konkretnom stlpci?

def pasuju(n, t, tleft):
    # pasuju do stlpca n krali typu t, ak je vo stlpci vlavo
    # rozmiestnenie kralov typu tleft?
    #
    # vsimnite si, ze ak zavolame tuto funkciu ako pasuju(n, t, 0),
    # tak iba otestuje, ci do stlpca n pasuju krali typu t

    # nesmieme pouzit zakazane policka v tomto stlpci
    if any( MAPA[r][n] == 'X' for r in TYPY[t] ): return False
```

```
    # nasi krali nesmu susedit s tymi vedla
    if any( abs(r1-r2) <= 1 for r1 in TYPY[t] for r2 in TYPY[tleft] ): return False

    return True

# odpovede[n][t] = maximalny pocet kralov, ktorych vieme umiestnit
# do stlpcov n .. N-1, ak je v stlpci n rozmiestnenie kralov typu t
odpovede = [ [ -987654321 for t in range(5) ] for n in range(N) ]

# pre posledny stlpec len prezrieme platne umiestnenia kralov
for t in range(5):
    if pasuju(N-1, t, 0):
        odpovede[N-1][t] = len( TYPY[t] )

# a teraz ideme dolava a postupne zodpovedame coraz zlozitejsie otazky
for n in reversed(range(N-1)):
    for t in range(5):
        if pasuju(n, t, 0):
            for tright in range(5):
                if pasuju(n+1, tright, t):
                    odpovede[n][t] = max( odpovede[n][t], len(TYPY[t]) + odpovede[n+1][tright] )

print( max( odpovede[0] ) )
```

B-II-2 Nočná mora

Ako zistiť, či je bod napravo od úsečky?

Najprv sa pozrieme, ako zistiť, či je jedna konkrétna Jožkova pozícia napravo od jedného konkrétneho lasera. Ak toto už viete, kľudne preskočte môžete preskočiť na ďalšiu časť vzorového riešenia.
Jeden priamočiary spôsob je predstaviť si každý laser ako lineárnu funkciu $f(y) = ay + b$. $f(y)$ nám hovorí, akú x-ovú súradnicu má bod lasera, ktorý má y-ovú súradnicu rovnú y. Ak sa Jožko nachádza na pozícii (x_i, y_i) stačí nám porovnať x_i a $f(y_i)$.

- $x_i > f(y_i)$ Jožko je napravo od lasera (lebo jeho x-ová pozícia je väčšia ako x-ová pozícia lasera vo výške y_i).
- $x_i < f(y_i)$ Jožko je naľavo od lasera.
- $x_i = f(y_i)$ laser prechádza cez Jožka, ale so zadanie vieme, že na vstupe nebudú takéto pozície.

Ostáva nám, ešte pre každý laser spočítať parametre a, b jeho zodpovedajúcej lineárnej funkcie. To vieme spočítať zo sústavy dvoch rovníc pre koncové body. Majme laser vedúci z bodu $(x_1, 0)$ do (x_2, h). (Jeden koniec je na spodnom konci miestnosti a jeden na hornom).
Platí, že

$$x_1 = f(0) = 0a + b = b$$

$$b = x_1$$

Ďalej platí, že

$$x_2 = f(h) = ah + b$$

$$ah = x_2 - b = x_2 - x_1$$

$$a = (x_2 - x_1)/h$$

Bod (x, y) je vpravo od úsečky $(x_1, 0) \to (x_2, h)$ pokiaľ

$$x > y \cdot (x_2 - x_1)/h + x_1$$

$$hx > (x_2 - x_1) \cdot y + x_1 \cdot h$$

Druhý riadok vznikne prenásobením prvého premenou h. Takáto podmienka sa lepšie overuje, lebo sa nemusíme trápiť nepresnosťami vznikajúcimi v programoch pri operácii delenia.

Existuje aj lepší spôsob, ako zisťovať, či je niečo napravo alebo naľavo od niečoho iného. V predošlom príklade sme mali šťastie, že laser ide z výšky 0 až do výšky h, ale ak by sme mali zistiť, či je daný bod napravo od všeobecnej úsečky $(x_1, y_1) \to (x_2, y_2)$, bol by vzorec o dosť komplikovanejší a nefungoval by pre situácie keď $y_1 = y_2$.
V geometrii existuje veľmi užitočný nástroj, ktorý dokáže tento problém vyriešiť jednoducho a pre akékoľvek úsečky. Tento nástroj sa volá **vektorový súčin**.
Nebudeme zachádzať do detailov, lebo tento nástroj je podrobnejšie rozpísaný napríklad v našej kuchárke https://ksp.sk/kucharka/skalarny_a_vektorovy_sucin/, povieme len, že vektorový súčin v 2D je operácia na dvoch vektoroch $v_1 = (x_1, y_1)$ a $v_2 = (x_2, y_2)$, ktorá vráti jedno číslo

$$v_1 \times v_2 = x_1 y_2 - y_2 x_1$$

Zároveň vieme (opäť odporúčame prečítať kuchárku, alebo za skúsiť zamyslieť s papierom a ceruzkou), že $v_1 \times v_2$ je kladný práve vtedy, keď v_1 je napravo od v_2. Toto je jeden z najužitočnejších základných stavebných kameňov geometrie a odporúčame si toto vštepiť do mysle a rovnako zapamätať aj vzorec $x_1 y_2 - y_2 x_1$, alebo vedieť tento vzorec odvodiť.
Ak už poznáme vektorový súčin, tak overiť, či sa Jožko (na pozícii (x, y)) nachádza napravo od lasera $(x_1, 0) \to (x_2, h)$, je triviálne, je to vtedy, ak vektorový súčin vektoru z $(x_1, 0)$ do (x, y) a vektoru z $(x_1, 0)$ do (x_2, h) je kladný.

$$(x - x_1)h - (x_2 - x_1)y > 0$$

Všimnime si, že nám koniec koncov vyjde ekvivalentný vzorec. Avšak, ak už dobre poznáme vektorový súčin, tak vieme tento vzorec odvodiť rýchlejšie, s menej chybami a bez strachu z delenia nulou.

Ako vyriešiť úlohu v kvadratickom čase?

Porovnáme každú Jožkovu pozíciu s každým laserom a spočítame, pre koľko laserov nám vyšlo, že Jožko sa nachádza napravo od lasera.
Časová zložitosť je $O(nk)$ a pamäťová $O(n)$

Listing programu (Python)

```
def precitaj_riadok_cisel() -> list:
    return [int(x) for x in input().split()]

def vektorovy_sucin(x1: int, y1: int, x2: int, y2: int) -> int:
    return x1 * y2 - x2 * y1

# Je (x,y) vpravo od usecky (x1,y1 - x2,y2)?
def vpravo(x: int, y: int, x1: int, y1: int, x2: int, y2: int) -> bool:
    return vektorovy_sucin(x - x1, y - y1, x2 - x1, y2 - y1) > 0


h, w, n, k = precitaj_riadok_cisel()
X1 = precitaj_riadok_cisel()
X2 = precitaj_riadok_cisel()
pocty = []

for i in range(k):
    x, y = precitaj_riadok_cisel()
    pocet = 0
    for j in range(n):
        if vpravo(x, y, X1[j], 0, X2[j], h):
            pocet += 1
    pocty.append(pocet)

print("␣".join(str(p) for p in pocty))
```

Vzorové riešenie

Aby sme zrýchlili algoritmus, použijeme techniku binárneho vyhľadávania. Nebudeme každú Jožkovu pozíciu porovnávať s každým laserom, ale využijeme, že lasery sú usporiadané zľava doprava.

Ak máme 100 000 laserov, môžeme porovnať, či sa Jožko nachádza napravo od lasera číslo 50 000. Ak áno, už vieme, že je napravo od všetkých laserov 1 až 50 000. Ak nie, tak je naľavo od všetkých laserov 50 000 až 100 000. Každopádne sme si ušetrili polovicu porovnaní.
Tento postup môžeme iterovať, až kým nám neostane jeden laser, v každom kroku „zahodíme“ ľavú alebo pravú polovicu zostávajúcich laserov. Dokopy spravíme len $\lceil \log_2 n \rceil$ porovnaní, ak n je 100 000, je to 17 porovnaní – obrovské ušetrenie času.
Pri implementácii binárneho vyhľadávania si treba dávať pozor na ± 1 chyby, najlepší spôsob je udržovať si polouzavretý interval – zostávajúce lasery budú vrátane `zac` ale bez `kon`. (Napr. počet laserov je `kon - zac`, bez ± 1 kdekoľvek.)
Časová zložitosť je $O(n + k \log n)$ pretože pre každú Jožkovu pozíciu musíme spraviť iba $O(\log n)$ porovnaní a pamäťová $O(n)$.

Listing programu (Python)

```
def precitaj_riadok_cisel() -> list:
    return [int(x) for x in input().split()]

def vektorovy_sucin(x1: int, y1: int, x2: int, y2: int) -> int:
    return x1 * y2 - x2 * y1

# Je (x,y) vpravo od usecky (x1,y1 - x2,y2)?
def vpravo(x: int, y: int, x1: int, y1: int, x2: int, y2: int) -> bool:
    return vektorovy_sucin(x - x1, y - y1, x2 - x1, y2 - y1) > 0


h, w, n, k = precitaj_riadok_cisel()
# pridame na zaciatok nulty laser, ktory je urcite nalavo od jozka
X1 = [-1] + precitaj_riadok_cisel()
X2 = [-1] + precitaj_riadok_cisel()
pocty = []

for i in range(k):
    x, y = precitaj_riadok_cisel()
    zac, kon = 0, len(X1)
    while kon - zac > 1:
        mid = (zac+kon)//2
        if vpravo(x, y, X1[mid], 0, X2[mid], h):
            zac = mid
        else:
            kon = mid
    pocty.append(zac)

print("␣".join(str(p) for p in pocty))
```

B-II-3 Hady a rebríky

Ak nevieme kde začať, je výhodné si úlohu najprv nejakým spôsobom zjednodušiť. Povedzme, že by nás zaujímala iba áno-nie odpoveď na otázku, či sa zo štartovného políčka dá dostať do cieľa. Ak nevieme na otázku jednoducho odpovedať, môžeme zistiť, kam sa vieme dostať napr. prvým hodom zo štartovného políčka. Nájsť dosiahnuteľné políčka je pre konkrétne políčko už celkom priamočiare. Postupne skúšame všetky možné výsledky hodu kockou a podľa výsledku hodu vždy odsimulujeme jeden posun (teda posun o počet krokov, ktoré padli na kocke a potom ešte možný následný pohyb po rebríku alebo hadovi). Podstatné však je, že políčka dosiahnuteľné na jeden krok z konkrétneho políčka vieme nájsť v konštantnom čase.
Teraz už vieme celkom jednoducho zistiť, na ktoré políčka sa vieme dostať na najviac 2 hody. Pozrieme sa na všetky políčka, ktoré boli dosiahnuteľné jedným hodom a z nich aplikujeme postup, ktorý sme už použili pri označovaní políčok dosiahnuteľných prvým hodom. Niektoré políčka už ako dosiahnuteľné označené boli, určite však máme teraz označené všetky dosiahnuteľné na jeden a dva hody. Tento postup môžeme opakovať k krát, čím označíme všetky políčka, ktoré sú dosiahnuteľné do k hodov kockou. Nastáva už iba otázka, dokedy tento proces opakovať aby sme si boli istý, že ak je cieľové poličko dosiahnuteľné, tak ho aj raz označíme. Nemáme totiž zaručené, že sa do cieľa dá dostať. Dôležité pozorovanie je, že nemusíme čakať donekonečna. Stačí, ak sa pri nejakom takomto kole simulovania hodov nenájde žiadne novo označené políčko. Potom už vieme, že sa nenájde nikdy. Akú zložitosť bude mať toto riešenie? Koľko môže byť takýchto kôl? Tvrdíme, že nanajvýš $n - 1$, nakoľko pri každom takomto kole sa pridá aspoň jedno novo označené políčko (až na posledné kolo) a

pridať sa ich môže nanajvýš $n - 1$. Implementovať jednu takúto fázu vieme celkom priamočiaro v časovej a pamäťovej zložitosti $O(n)$ napríklad tak, že si v poli veľkosti n pamätáme na indexe i, či je i-te políčko už označené. Výsledná časová zložitosť je tak $O(n^2)$ a pamäťová $O(n)$.

Listing programu (C++)

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

using namespace std;

int main()
{
  int n;
  cin >> n;
  vector<int> plan(n);
  for (int i = 0; i < n; ++i)
  {
    cin >> plan[i];
  }
  vector<bool> kde_viem_byt(n, false);
  kde_viem_byt[0] = true; // prve policko je oznacene

  for (int krokov = 0;; ++krokov)
  {
    if (kde_viem_byt[n - 1])
    { // ak je oznacene policko n-1, koncime, vieme na kolko hodov je
      // dosiahnutelne
      std::cout << krokov << "\n";
      return 0;
    }
    // skopirujeme obsah pola staro oznacenych policok
    vector<bool> nove_kde_viem_byt = kde_viem_byt;

    for (int i = 0; i < n; ++i)
      if (kde_viem_byt[i])
      {
        for (int kocka = 1; kocka < 7; ++kocka)
          if (i + kocka < n)
          {
            // ciel ratame ako:
            // aktualne_policko + co padlo na kocke + mozny presun
            // hadom/rebrikom
            int ciel = i + kocka + plan[i + kocka];
            nove_kde_viem_byt[ciel] = true;
          }
      }

    // ak sme neoznacili ziadne nove policka, koncime
    if (kde_viem_byt == nove_kde_viem_byt)
    {
      std::cout << "-1\n";
      return 0;
    }

    kde_viem_byt = nove_kde_viem_byt;
  }

  return 0;
}
```

Ide tento postup nejak vylepšiť? Uvedomme si, že v každej fáze sa pokúšame nové políčka dosiahnuť z každého možného doteraz dosiahnuteľného políčka. To je ale zbytočná práca navyše. V každej jednej fázi totiž napríklad opakujeme výpočet, v ktorom sa pozrieme na štartovné políčko a zistíme, kam všade sa dá z neho dostať. To je zjavne nepotrebné. Pozorovanie je, že v k-tej fáze objavíme nové políčko len cez políčka, ktoré boli novo objavené v predchádzajúcej $(k - 1)$-vej fáze. Prečo to tak je? Všetky políčka dosiahnuteľné z novo objaveného políčka sú vždy pridané v nasledujúcej fáze. Aktualizované riešenie znovu pozostáva z postupného objavovania dosiahnuteľných políčok. Tento krát ale kandidátov na novo objavené políčka budeme hľadať iba z novo objavených políčok z posledného kola. Počet kôl tento krát môže byť zas rádovo n. Dôležité však je, že v predchádzajúcom riešení sme v každej fáze robili až $O(n)$ práce. Teraz však v každom kole prejdeme iba políčka novo objavené v predchádzajúcom kole, ktorých môže byť dokopy za všetky kolá najviac n. Dokopy teda za všetkých najviac n kôl môžeme urobiť iba $O(n)$ práce a tak bude výsledná časová aj pamäťová zložitosť lineárna.

Listing programu (Python)

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

spracovane = set()

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
            ciel += plan[ciel]
            if ciel in spracovane: continue
            kde_budem_po_tahu.add(ciel)

    spracovane |= kde_viem_byt
    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
```

Listing programu (C++)

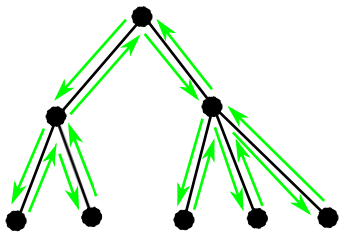
```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  int n;
  cin >> n;
  vector<int> plan(n);
  for (int i = 0; i < n; ++i)
  {
    cin >> plan[i];
  }
  vector<bool> kde_viem_byt(n, false);
  // naposledy_objavil sluzi na ulozenie
  // policok objavenych v poslednej "faze"
  vector<int> naposledy_objavil;

  // prve policko je dosiahnutelne
  kde_viem_byt[0] = true;
  naposledy_objavil.push_back(0);

  for (int krokov = 0;; ++krokov)
  {
    if (kde_viem_byt[n - 1])
    {
      std::cout << krokov << "\n";
      return 0;
    }

    vector<int> teraz_objavene;
    for (int x : naposledy_objavil)
    {
      for (int kocka = 1; kocka < 7; ++kocka)
        if (x + kocka < n)
        {
          int ciel = x + kocka + plan[x + kocka];
          kde_viem_byt[ciel] = true;
          teraz_objavene.push_back(ciel);
        }
    }

    if (teraz_objavene.empty())
    {
      std::cout << "-1\n";
      return 0;
    }

    naposledy_objavil = teraz_objavene;
  }

  return 0;
}
```

B-II-4 Školský výlet II

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*. (Hamiltonovské cesty môžu začínať aj končiť hocikde v grafe, nemajú predpísaný začiatočný ani koncový vrchol.)

Podúloha A: výlet po strome

Sada $n - 1$ hrán, ktoré vybrala Majka, dokopy tvorí strom – presnejšie, jednu možnú kostru nášho grafu. Po hranách tohto stromu chceme prejsť tak, aby sme navštívili všetky jeho vrcholy a pri tom práve dvakrát (raz tam, raz späť) prešli po každej hrane.
Priamočiarym spôsobom, ako toto dosiahnuť, je prehľadávanie do hĺbky. Detailný popis tohto algoritmu nájdete tu: https://www.ksp.sk/kucharka/dfs/.

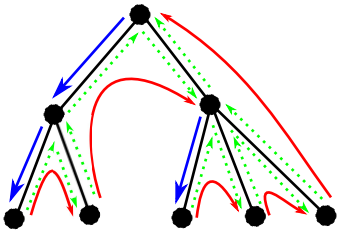


Príklad okružnej cesty po strome pomocou prehľadávania do hĺbky.

```
def dfs(graf, vystup, vrchol, rodic = None):
    for sused in graf[vrchol]:
        if sused != rodic:
            print('cestujeme z', vrchol, 'do', sused)
            vystup.append(sused)

            dfs(graf, vystup, sused, vrchol)

            print('cestujeme z', sused, 'do', vrchol)
            vystup.append(vrchol)

graf = [ [] for n in range(N) ]
for x, y in hrany:
    graf[x].append(y)
    graf[y].append(x)

vystup = [0]
dfs(graf, vystup, 0)
```

Podúloha B: vylepši výlet po strome

Hlavná myšlienka riešenia tejto podúlohy je nasledovná: Predstavme si, že sme postupne navštívili vrcholy $x_1, x_2, \ldots, x_k$. Za tento úsek cesty sme zaplatili dokopy $C[x_1, x_2] + \cdots + C[x_{k-1}, x_k]$ peňazí. Ako by sa zmenila cena, ak by sme išli z x_1 priamo do x_k?
Pripomeňme si, že naše ceny spĺňajú trojuholníkovú nerovnosť. Ak teda z cesty vynecháme nejaký vrchol, jej cena zostane rovnaká alebo sa zmenší. Túto úvahu môžeme opakovať, až kým nedostaneme cestu tvorenú len prvým a posledným vrcholom. (Formálne, matematickou indukciou od k sa dá dokázať, že $C[x_1, x_2] + \cdots + C[x_{k-1}, x_k] \geq C[x_1, x_k]$.)
Vyzbrojení týmto pozorovaním vieme teraz podúlohu B vyriešiť nasledovne: spustíme riešenie podúlohy A, len navyše vždy, keď by chcelo navštíviť vrchol, kde už sme predtým boli, tento vrchol jednoducho preskočíme.
Malo by byť zjavné, že takto dostaneme plán cesty, ktorý každý vrchol navštívi práve raz. A vyššie sme si zdôvodnili, že tým, že nejaké vrcholy preskočíme, sa celková dĺžka cesty nemôže predĺžiť. Splnili sme teda obe požiadavky.



Príklad prerobenia okružnej cesty po strome na platný výlet.
(Výlet tvoria modré a červené hrany. Modré sú pôvodné úseky, červené sú „skratky“.)

Samotná implementácia je veľmi jednoduchá. Stačí robiť presne to isté, čo v podúlohe A, len tentokrát vypísať každý vrchol len raz – vtedy, keď doň prvýkrát prídeme.

Podúloha C

Nech k je celková cena najlacnejšej kostry.
Algoritmus z podúlohy A nájde plán okružnej cesty s cenou presne $2k$. Algoritmus z podúlohy B z neho následne vyrobí platný plán výletu, pričom vieme, že cena určite nestúpne. Výlet, na ktorý pôjde Kleofášova trieda, teda bude stáť nanajvýš $2k$.

Tvrdenie 1 (o vzťahu medzi Hamiltonovskou kružnicou a Hamiltonovskou cestou): Najlacnejšia Hamiltonovská cesta je lacnejšia ako najlacnejšia Hamiltonovská kružnica.

Dôkaz: Zoberme konkrétnu najlacnejšiu Hamiltonovskú kružnicu. Vynechajme ľubovoľnú jej hranu. Dostaneme nejakú Hamiltonovskú cestu. Keďže ceny hrán sú kladné, naša Hamiltonovská cesta je lacnejšia od najlacnejšej Hamiltonovskej kružnice. No a najlacnejšia Hamiltonovská cesta je nanajvýš taká drahá ako tá naša.

Tvrdenie 2 (o vzťahu medzi kostrou a Hamiltonovskou cestou): Najlacnejšia kostra je nanajvýš taká drahá ako najlacnejšia Hamiltonovská cesta.

Dôkaz: Každá Hamiltonovská cesta je aj kostrou. Sú teda len dve možnosti: buď je najlacnejšia Hamiltonovská cesta zároveň aj najlacnejšou kostrou, alebo existuje iná kostra, ktorá je od nej ešte lacnejšia.

Z vyššie dokázaných tvrdení vieme teraz vyvodiť nasledovný záver:
Nech h je cena najlacnejšej Hamiltonovskej kružnice, čiže cena najlacnejšieho možného okružného výletu.
Z tvrdení 1 a 2 dokopy dostávame, že nutne platí $k < h$. No a keďže Kleofášov výlet stojí nanajvýš $2k$, tak stojí menej ako $2h$, čiže menej ako dvojnásobok optimálnej ceny. Kleofáš teda hovoril pravdu.

TRIDSIATY SIEDMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Farnbauer, Michal Forišek, Ján Hozza, Andrej Korman
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2022