Korešpondenčný seminár z programovania
42. ročník, 2024/25
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

Vladaso

1. Hlúpa intergalatická pošta!

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

Ako najjednoduchšie riešenie je si sitúaciu odsimulovať. Spraviť si pole `[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]` a pamätať si pozíciu, na ktorej sa nachádzame. Túto pozíciu meníme a pre každé číslo v pin-e si pomocou cyklu cez pole zistíme, ktorá strana je bližšie a na tú stranu sa vydáme. Pri takomto počítaní nemôžeme zabudnúť, že pole je ako keby tvaru kruhu a preto musíme myslieť na pozície na krajoch (1 -> 0 a 0 -> 1). Toto však nie je úplne najlepšie riešenie.

Problém si chceme simulovať, ale nechceme manuálne pohybovať ukazovátkom. Vlastne, z každého bodu máme dve cesty kadiaľ sa vieme vybrať. Buď doľava alebo doprava. Minimum pohybov potrebných na napísanie pinu dostaneme vtedy, keď si vždy vyberieme tú kratšiu cestu. Ešte však treba vymyslieť logiku za tým, koľko krokov potrebujeme na to, aby sme sa dostali na číslo z nejakej pozície.

Stále si potrebujeme pamätať našu pozíciu, no nejdeme krok po kroku, ale vypočítame si počet krokov, ktoré potrebujeme. V našom cykle pozíciu potom vždy updatujeme iba na miesto, kam sa chceme presunúť.

Treba si uvedomiť že k počtu krokov môžeme ihneď pripočítať dĺžku pin kódu, keďže určite budeme musieť každé číslo stlačiť.

Ak sa chceme dostať z čísla 5 na 6 alebo 6 na 5 je to vlastne $\text{abs}(6-5)$ alebo $\text{abs}(5-6)$, čo je rovnaké číslo. Musíme však rátať aj s tým, že cesta druhou stranou vie niekedy byť rýchlejšia. Cestu druhou stranou dostaneme keď od celkového počtu políčok odčítame vzdialenosť, ktorú by sme prešli ak by sme nepoužili cestu z 1 na 0. Teda pre nás je to $10 - \text{abs}(5-6)$, čo nám dá výsledok 9, čo je naozaj počet krokov, keby putujeme druhou stranou. Vzorec na výpočet najkratšej vzdialenosti z pozície i na pozíciu x bude teda vyzerať takto: $\min(\text{abs}(x-i), 10 - \text{abs}(x-i))$

Menšiu z týchto dvoch možností (ísť doľava alebo doprava) pripočítame k počtu krokov a opakujeme pre všetky čísla v pin-e. Keď prejdeme všetkými číslami v pin-e, vypíšeme celkový počet krokov.

Gardener

2. Vladkova hra

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

Simulácia

Priamočiarym riešením by mohlo byť si Vladkove hranie odsimulovať. Prechádzať po políčkach zľava doprava, vždy zmeniť stav plošinky a po postavení sa na políčko bez plošinky, vrátiť sa na začiatok. Počas tohto si budeme počítať, koľko krokov sme spravili. Toto budeme opakovať dovtedy, kým sa Vladko nedostane na koniec.

Toto riešenie funguje, avšak je pomerne pomalé.

Potrebujeme to simulovať?

Skúsme sa pozrieť na niektoré hry. Ak Vladkov level začína v stave `0 0 0`, po prvom prejdení skončí v stave `1 0 0`, potom `0 1 0`, `1 1 0`, `0 0 1`, `1 0 1`, `0 1 1` a skončí v `1 1 1`. Vladko tento level dokončí po 7-ich pokusoch.

Môžeme si všimnúť, že ak by sme level mali nakreslený opačne, tak postupnosť `0 0 0`, `0 0 1`, `0 1 0`... vyzerá presne ako postupnosť binárnych čísiel od 0 do 7. Teraz si potrebujeme uvedomiť, že každý level, ktorý sa skladá iba z políčok bez plošiniek sa správa rovnako, a teda ho vieme reprezentovať ako postupnosť binárnych čísiel od 0 po x. Následne potrebujeme zistiť, aké je to x pre daný počet políčok. Ak máme v leveli n políčok, aké najväčšie binárne číslo v ňom vieme zobraziť? To bude n jednotiek. Ak n binárnych jednotiek chceme premeniť na číslo v desiatkovej sústave, dostaneme $2^0 + 2^1 + \cdots + 2^{n-1}$. Alternatívne, vieme, že ak máme n binárnych jednotiek, tak číslo o jedna väčšie bude jednotka a n núl, čiže 2^n v desiatkovej sústave. Potom naše pôvodné číslo bude $2^n - 1$.

Čo ale, ak nezačíname so samými nulami? Vtedy môžeme počiatočný stav vyjadrený ako binárne číslo odpočítať od celkového počtu pokusov, keby všetky políčka boli nulové.

Alternatívne môžeme počiatočný stav ako binárne číslo znegovať (`0 1 0 -> 1 0 1`) a premeniť do desiatkovej sústavy, čím dostaneme počet pokusov.

Časová aj pamäťová zložitosť takéhoto riešenia je $O(n)$. Technicky si nemusíme pamätať celý vstup, ale spracovávať ho postupne, vtedy by sme vedeli dostať pamäťovú zložitosť aj $O(1)$.

Maťo

3. Intraplanetárne čížmy

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

Ak sa nachádzame na pozícii x a vieme prejsť ešte d, vieme sa dostať na všetky pozície až do $x+d$. Spomedzi čižiem na týchto pozíciách si tak chceme vybrať nejaké čižmy, ktoré nás dovedú k optimálnej odpovedi.

Intuitívne dáva zmysel, že to budú také čižmy i, že ich $A_i + B_i$ je najväčšie spomedzi všetkých dostupných čižiem. Uvažujme ale, že by sme tieto čižmy nezobrali a zobrali nejaké iné čižmy j. V následujúcom kroku by sme mali na výber z čižiem z nejakej podmnožiny tých, ku ktorým sme sa vedeli dostať pomocou čižiem i, pretože by sme sa mohli dostať na všetky čižmy, okrem tých, ktoré sú na ceste na pozíciách $x + A_j + B_j + 1$ až $x + A_i + B_i$. (pre každé j musí platiť, že $A_j + B_j \leq A_i + B_i$, keďže $A_i + B_i$ je maximálne).

Keďže teraz vieme, že máme na výber stále iba z podmnožiny čižiem dosiahnuteľných z i, ak zoberieme čižmy i, vieme, že výber čižiem v ďalšom kroku bude najväčší možný. Z toho vyplíva, že po k krokoch budeme vždy najďalej ako sa dá, pretože stále robíme najväčší krok, ktorý nám aj ako sme si ukázali, najviac zväčší výber. To nám zaručí, že dôjdeme na koniec cesty v najmenšom počte krokov, ak je to možné. Ak to možné nie je, zistíme to tak, že z aktuálnych čižiem sa už nie je možné dostať do iných čižiem, ktorými vieme dôjsť ďalej a nie je možné sa s nimi dostať ani do cieľa.

Prvé riešenie

Popísaný postup môžeme simulovať a dostaneme sa k nejakému riešeniu. Z každej pozície, kde sme si obuli nové čižmy sa pozrieme na čižmy v dosiahnuteľnej vzdialenosti vpravo. Spomedzi týchto čižiem následne vyberieme také, ktorými sa dostaneme najďalej a tento proces opakujeme, až dokým nedôjdeme na koniec cesty.

Časová zložitosť

Aj keď tento algoritmus možno na prvý pohľad vyzerá, ako $O(n^2)$, vieme dokázať, že na žiadne čižmy sa nepozrieme viac ako dvakrát. Ak sa pozeráme na čižmy i, prezrieme všetky čižmy na úseku A_i až po $A_i + B_i$. Z týchto vyberieme najlepšie čižmy j. Následne sa pozrieme na všetky čižmy na úseku od A_j až po $A_j + B_j$. Druhýkrát sa počas toho pozrieme na čižmy na úseku od A_j po $A_i + B_i$. Nikdy sa ale nepozrieme na tieto čižmy tretíkrát, pretože na tomto úsek už nemôžu byť žiadne lepšie čižmy (s ktorými by sme sa dostali ďalej ako $A_j + B_j$), pretože inak by sme ich zobrali namiesto j. Teda nutne vyberieme nejaké, čo sa nachádzajú až za $A_i + B_i$. Toto platí pre každú dvojicu po sebe idúcich vybratých čižiem a teda na všetky úseky na ktoré sa pozrieme druhýkrát sa nepozrieme už tretíkrát.

Z toho už vyplýva, že časová zložitosť je $O(2n) = O(n)$, keďže čižmy sú už na vstupe zoradené. Pamäťová zložitosť je tiež $O(n)$.

Druhé riešenie

Existuje aj iné, rovnako efektívne riešenie s ľahšou analýzou časovej zložitosti. Môžeme si všimnúť, že zaujímajú nás iba také čižmy i, že $A_i + B_i > A_j + B_j$ pre všetky $i > j$. Čižmy tak stačí postupne prejsť a zapamätať si iba tie pre ktoré platí spomenutá podmienka. Overovať všetky j nie je treba, pretože posledné zapamätané čižmy budú stále tie s najväčším dosahom a tak stačí podmienku skontrolovať pre nich.

Následne postupne prejdeme novo vytvorený zoznam a stále keď máme čižmy i, zoberieme posledné zapamätané čižmy j, ktore sú za i, a ktorých $A_j < A_i + B_i$. Tými sa už dostaneme najďalej ako je to možné zo všetkých dosiahnuteľných čižiem od čižiem i. Toto opakujeme podobne ako v prvom riešení. Druhé riešenie je teda veľmi podobné tomu prvému, akurát odstránime nejaké čižmy, aby sme sa v druhom cykle pozreli na všetky ostávajúce čižmy iba raz. Časová aj pamäťová zložitosť je rovnaká ako pri prvom riešení.

Andrej Lackovič

4. Existenčná kríza

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

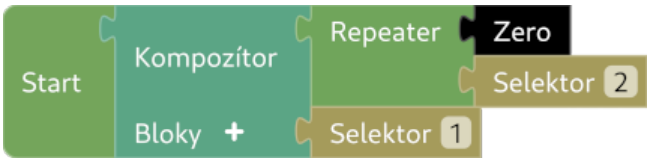
Pokiaľ pri podúlohe nie je uvedené inak, tak si vstupy označíme postupne a, b, c, …

a. nula

Na to, aby sme zavolali Zero bez vstupov, využijeme Kompozítor a Repeater.

Kompozítor vyžaduje aspoň dva bloky, z čoho vyplýva, že `final` blok dostane vždy aspoň jeden vstup. Kompozítor tak vieme využiť na eliminovanie počtu vstupov na jeden.

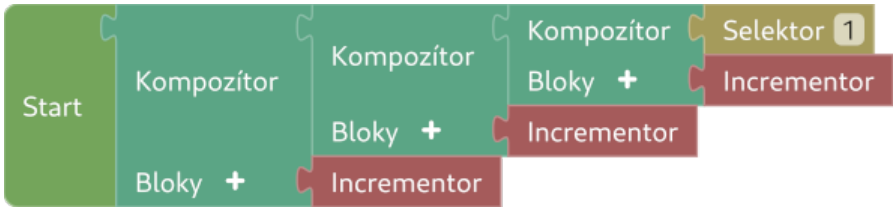
Teraz sa pozrieme na Repeater - ako prvý vstup berie počet opakovaní a ostatné vstupy sú voliteľné. Keď Repeater zavoláme s jedným vstupom, tak `init` nedostane žiadny vstup, čiže ako `init` vieme dať Zero. Už len si potrebujeme túto hodnotu udržať, a tak `step` musí vždy vrátiť priebežnú hodnotu. Na to už len použijeme Selektor.



b. +3

Použijeme vnorené Kompozítory, aby sme postupne 3× zavolali Increment na vstupe.

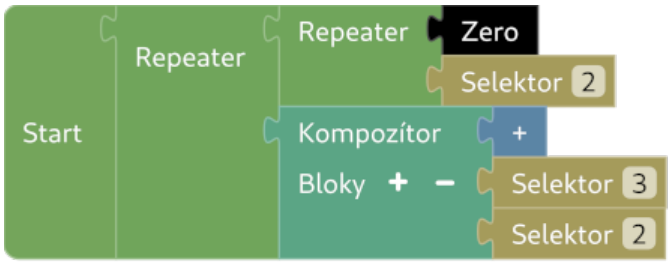
Skúste si uvedomiť, že pomocou podúlohy a. takto vieme vyrobiť ľubovoľnú nezápornú konštantu s ľubovoľným počtom vstupov.



c. násobenie

Násobenie spravíme pomocou postupného sčítania (to je vysvetlené v tutoriáli). Pomocou Repeatera budeme volať sčítavanie, ktoré zoberie predchádzajúcu hodnotu a druhý vstup Repeatera.

A čo bude počiatočná hodnota Repeatera? Takýmto spôsobom a krát pripočítame b, čiže začiatočná hodnota musí byť 0. Na to môžeme využiť nulu z podúlohy a.

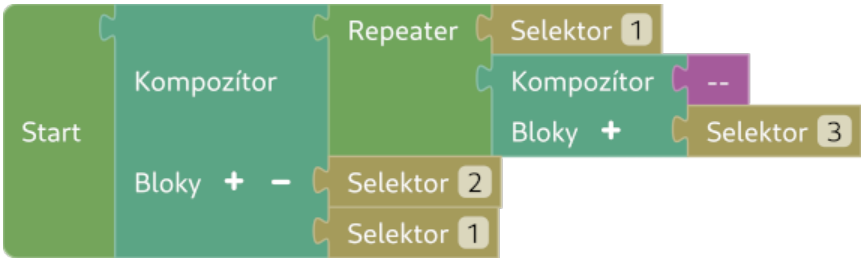


d. decrement

Trik spočíva v tom, že Repeater počíta iterácie od 0 po $n-1$. Použijeme Repeater s počiatočnou hodnotou 0 (špeciálny prípad) a `step` vráti číslo iterácie.

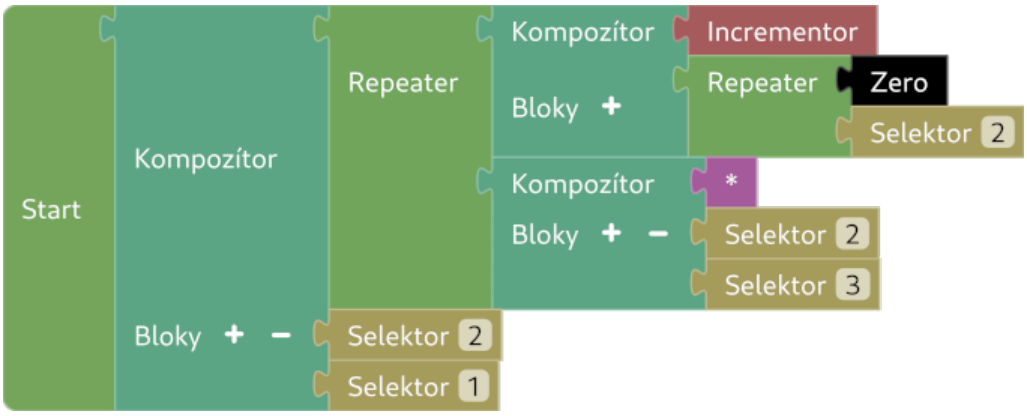
e. odčítanie

Odčítanie spravíme podobne ako sčítanie, ale použijeme decrement namiesto Incrementora a vymeníme poradie vstupov pomocou Kompozítora, aby sme sme odčítali b číslo od a.



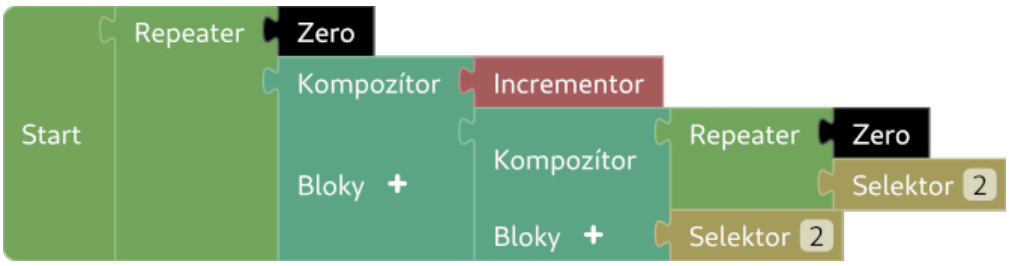
f. umocňovanie

Obdobne ako sme pri násobení použili sčítanie, teraz použijeme násobenie. Avšak tento krát počiatočná hodnota bude 1 (keby sme nechali 0, tak výsledok bude vždy 0), čiže použijeme pozorovanie z podúlohy b. Nakoniec ešte musíme vymeniť poradie vstupov, aby sme b krát vynásobili a a nie naopak.



g. sign

Potrebujeme vrátiť 0, ak je vstup 0, inak vrátiť 1. Na to použijeme Repeater - ak $a = 0$, tak Repeater vráti počiatočnú hodnotu cyklu, ktorú nastavíme na 0. Ak $a > 0$, tak Repeater by mal Repeater vrátiť 1 - na to vieme opäť použiť pozorovanie z podúlohy b.

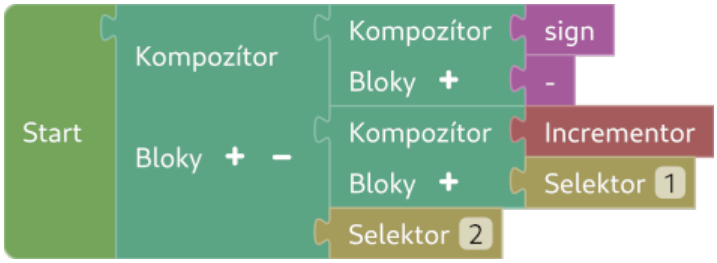


h. >=

$a - b$ nám vráti 0 ak $a \leq b$, v opačnom prípade vráti kladné číslo. My ale chceme vrátiť iba 0 alebo 1 a na to vieme použiť `sign` z podúlohy g. Takto dostaneme 0 ak $a \leq b$ a v opačnom prípade 1.

Ak chceme dostať 0 ak $a < b$ a v opačnom prípade 1, musíme rovnosť posunúť. Zväčšením a o jedna, dostaneme $a + 1 - b$, ktoré vráti 0 ak $a < b$ a inak vráti 1 (po použití `sign`).

Skúste si uvedomiť, že podobným spôsobom vieme vytvoriť aj funkcie `<`, `<=` a `>`.



i. if

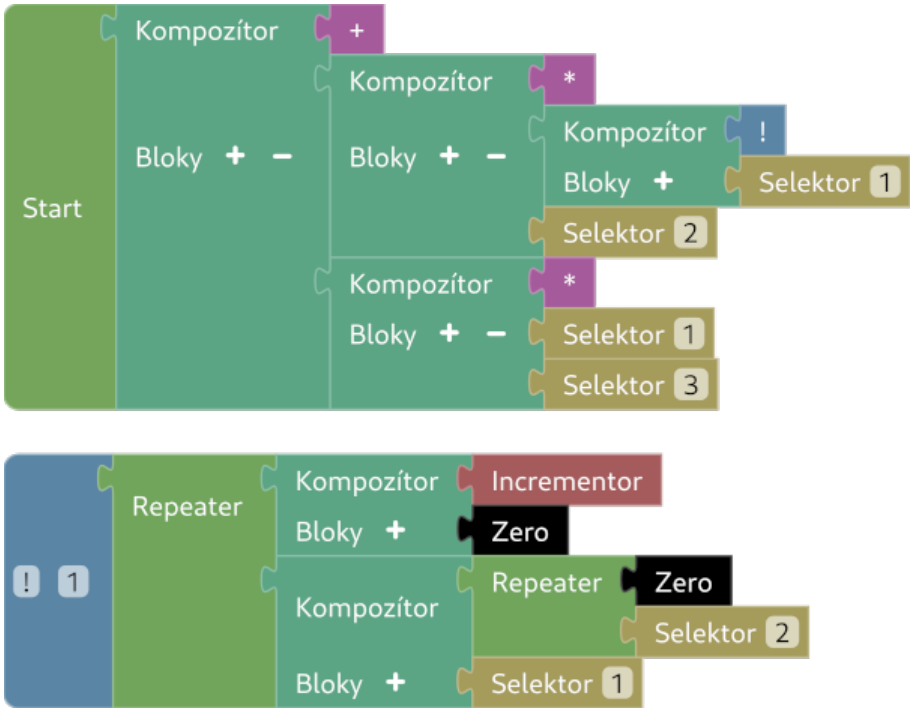
Označme si vstupy postupne c (condition), i (if) a e (else).

Začnime tým, že si definujeme funkciu `!` , ktorá zneguje svoj jediný vstup - ak je 0, tak vráti 1, inak vráti 0. Môžeme si všimnúť, že je to vlastne `sign`, len s vymenenými konštantami - `init` bude 1 a `step` bude 0.

Teraz, keď už vieme znegovať c, môžeme si skúsiť úlohu vyjadriť matematicky:

$$v = !a * c + b * c$$

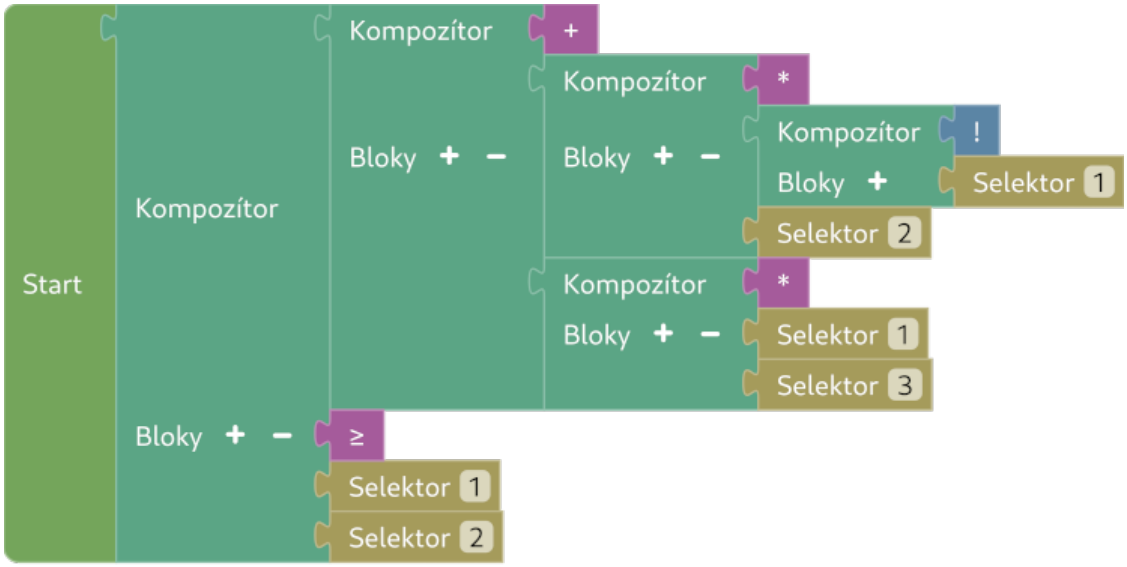
Ak $c = 0$, tak $v = b$, inak $v = a$, čo je to, čo máme urobiť. A to už vieme urobiť pomocou predchádzajúcich podúloh a vnorených Kompozítorov.

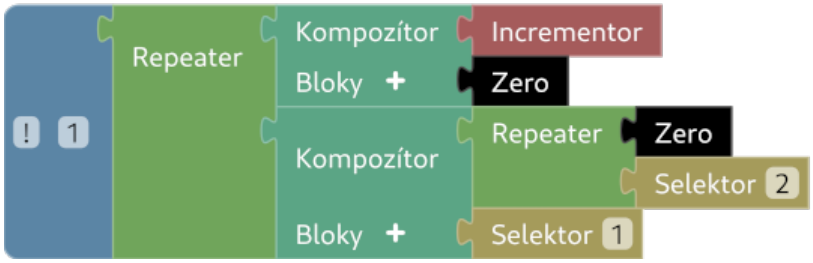


j. min

Táto podúloha sa veľmi podobá na podúlohu i, až na to, že nemáme c na vstupe. c si vieme získať pomocou podúlohy h. Čiže nám stačí pomocou Kompozítora a pretransformovať vstup pre program z podúlohy i.

Skúste si uvedomiť, že `max` vieme vytvoriť použitím `<=` z podúlohy h.

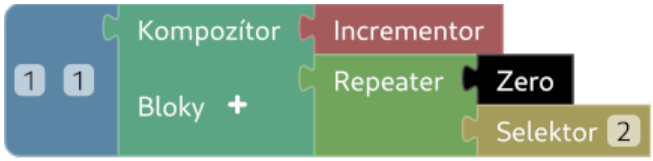


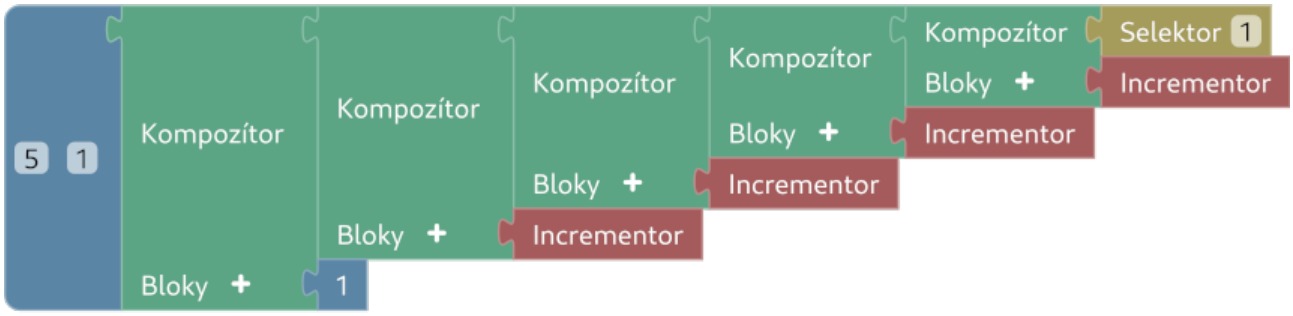


k. bonus

Pre detaily ohľadom riešenie pôvodnej úlohy si pozrite vzorák prvej úlohy.

Najskôr si definujme konštanty `10` a `1` s jedným vstupom:



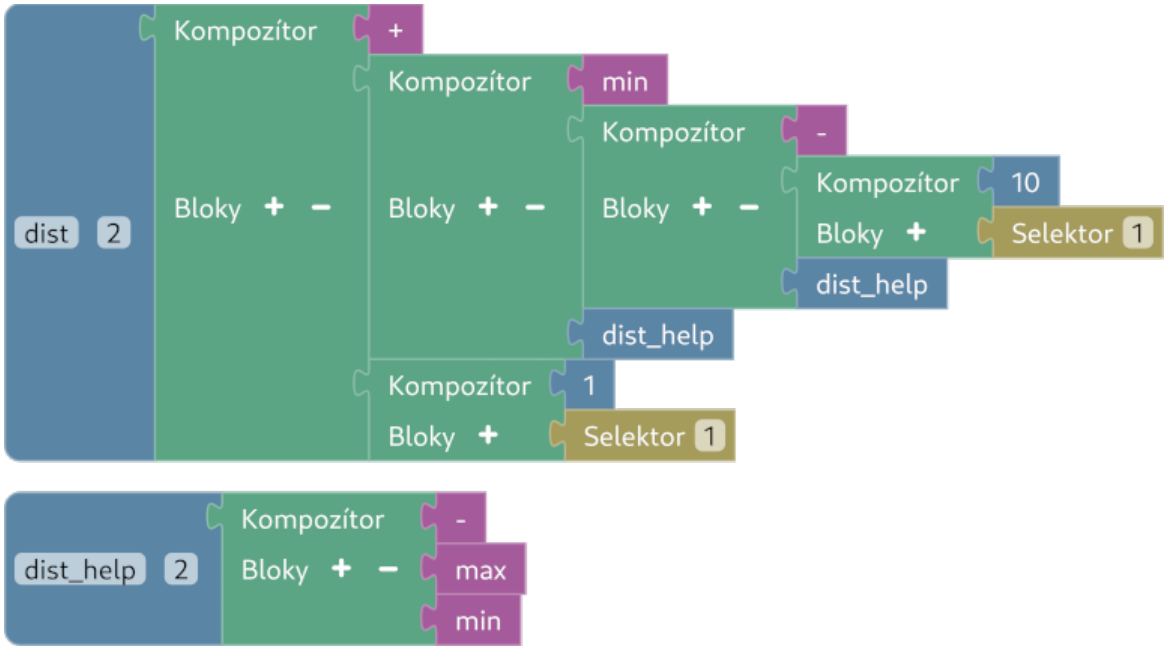


Ďalej si potrebujeme vyrobiť funkciu `dist` s dvoma vstupmi a a b, ktorá vypočíta vzdialenosť dvoch pozícií (vstupov) + 1 (kliknutie).

Vzdialenosť dvoch pozícií je $\min(10 - \text{abs}(a - b), \text{abs}(a - b))$

Vieme spraviť všetko, okrem `abs`. My ale `abs` nepotrebujeme, stačí nám vždy odčítavať menšie číslo od väčšieho, čo vieme spraviť ako: $\max(a, b) - \min(a, b)$. Túto funkciu si pomenujme `dist_help`.

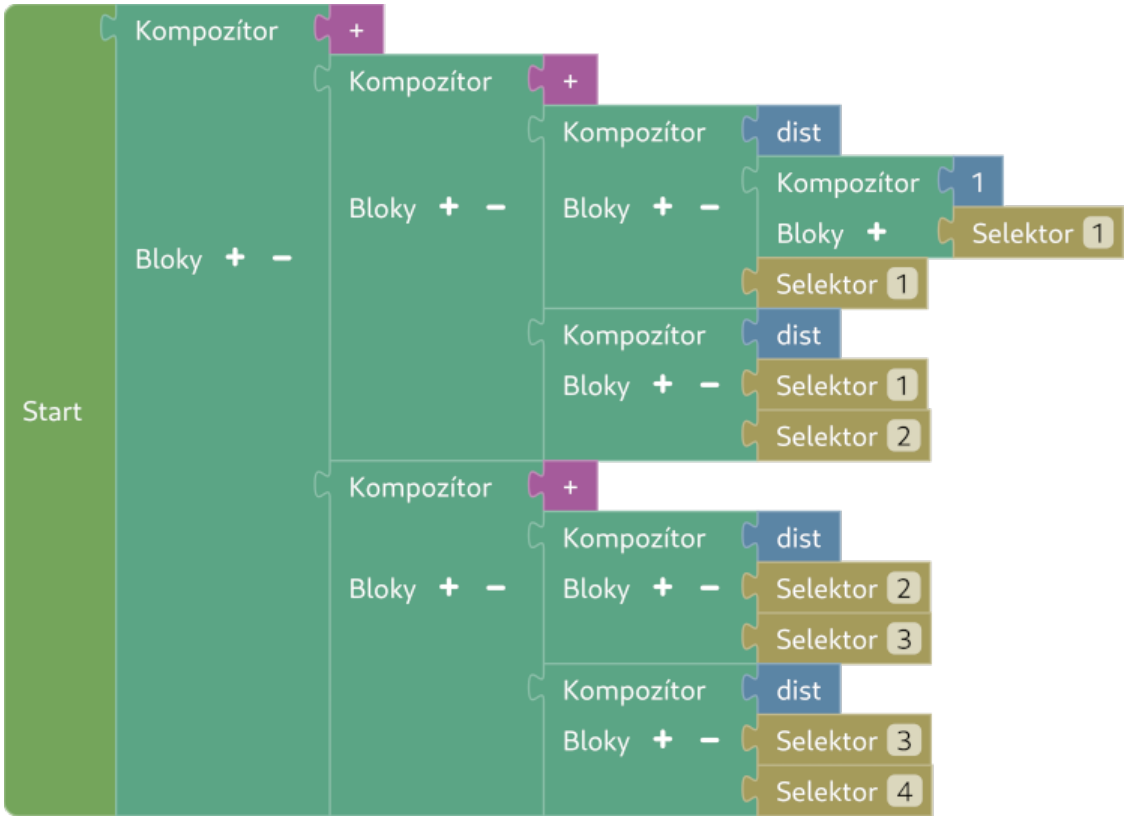
Počítanie vzdialenosti tak vyzerá nasledovne:



Keď už vieme vypočítať vzdialenosť dvoch pozícií, už len potrebujeme dostať finálny výsledok. To vieme tak, že postupne sčítame medzivýsledky:

$$\text{dist}(1, a) + \text{dist}(a, b) + \text{dist}(b, c) + \text{dist}(c, d)$$

Takto dostaneme konečný výsledok (1 je počiatočná pozícia, ako je definované v zadaní):



Strizo

5. Zašifrovávať budeme

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

Bruteforce

Prvé čo nám napadne je, že však poďme si to odsimulovať a tým pádom na konci len vypíšeme k-te pismenko z nášho výsledneho stringu. Toto riešenia má časovú zložitosť $O(D^N)$ kde D je, že koľko najviac písmenok sa vie stať z jedného písmenka, a N je počet "zámien" ktoré máme spraviť. Pamäťová zložitosť je $O(D^N)$, keďže až po takto veľký náš výsledok vie narásť.

Niečo lepšie

Ďalej si môžeme všimnúť, že my si nemusíme odsimulovávať vždy celý string, ale môžme ísť postupne po písmenkách. Čiže namiesto toho, aby sme z "abc" si vyvorili "aakfhjioufdljfskmdklmdscdk", sa môžme pozrieť iba na prvé písmenko, a na prvé písmenko čo nám vznikne z tohto písmenka. Týmto pádom nám vznikne niečo ako strom kde každý vrchol je písmenko a z neho vedú hrany k písmenkam ktoré z neho môžu vzniknúť. Tento spôsob nám ale dovoľuje aby sme zastali v generovaní vtedy keď sa dostaneme na k-tu pozíciu a tým pádom nemusíme generovať celý výsledný string. A akú to má časovú zložitosť? Takže je nám jasné, že ak by sme sa pýtali na posledné písmenko zo stringu, tak by sme ho museli vygenerovať celý, a teda worst-case by bola $O(D^N)$, ale keďže vieme prerušiť generovanie keď dôjdeme k nášmu k-temu prvku, tak časová zložitosť bude $O(min(K, D^N))$, čo keď si všimneme obmedzenia by nám malo vedieť prejsť prvými troma sadami. A pamäťová zložitosť sa dá optimalizovať na $O(KD)$, keďže si musíme pamätať ku každému písmenku za čo ho zamieňame a potom už len nejaký zásobník veľkosti &n& aby sme sa vedeli hýbať po našom strome.

Zaujímavá myšlienka

My ale v skutočnosti nepotrebujeme vedieť, aké boli tie písmenká pred našim k-čkom. To znamená že ak nájdeme rozumný spôsob ako povedať, že koľko písmenok vznikne z tohto nášho za J vnorení, potom vieme povedať, či sa nám ho oplatí úplne preskočiť, alebo je naše riešenie v ňom a teda ho pôjdeme generovať

Optimálne riešenie

Čiže čo sa snažíme zistiť je, že pre každé písmenko, koľko sa z neho stane po J vnoreniach. Znie povedome? Áno, vytvorili sme si podproblémy, ktoré na seba rozumne nadväzujú. Je nám totiž jasné, že keď viem pre každé písmenko koľko z neho vznikne iných po J opakovaniach, tak potom keď chcem to isté vedieť pre niektoré moje písmenko po $J+1$ opakovaniach, tak sa mi len stačí pozrieť na aké všetky sa zmení po jednej premene a spočítať hodnoty tých, na ktoré by sa premenila, ktoré majú po J opakovaniach. A keď už všetky tieto informácie mám, tak potom viem postupne prechádzať cez úvodný string, a ak moje K-te pismenko sa vytvori z toho písmenka, na ktoré sa pozerám, tak si ho rozložím a znova sa pozriem, či sa moje k-te písmenko nachádza pod prvým alebo druhým atď. To, pod ktorým písmenkom sa nachádza moje, viem povvedať tak, že ak mám spočítané, koľko mi dajú všetky písmenká pred ním, tak ak to písmenko, na ktorom som + tie, ktoré som už prešiel, je väčšia alebo rovná ako K tak vtedy sa k vytvorí z tohto môjho písmenka, keďže je nám jasné, že suma tých pred týmto mojim je menej ako K, inak by sme sa zastavili na nejakom predchádzajúcom písmenku. Toto riešenie má časovú zložitosť $O(D_N)$, keďže generujeme tak veľkú súčtovú tabuľku a nájdenie výsledku je taktiež najhoršie $O(D_N)$, keďže by sme N krát mohli prejsť cez celý string čo vytvárame z daného písmenka ktorý je maximálne D. A pamäťová zložitosť je taktiež $O(DN)$, keďže túto súčtovú tabuľku si musíme pamätať.

Viktor

6. Diskrétny Banket

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

Zadrátované riešenie

Najprv sa pozrime na veličiny, ktoré v úlohe vystupujú. Všimnime si, že otázok je síce veľa, ale čísla v nich budú pomerne malé. Vyzerá to teda, že optimálne bude výsledky pre čísla počítať dopredu, respektíve ich vypočítať, keď ich potrebujeme, a odvtedy si ich už pamätať. Možných otázok je totiž pomerne málo… V prvej sade dokonca platí, že žiadne čísla nie sú zakázané - teda v tejto sade bude odpoveď na dané číslo zakaždým rovnaká. Prvú sadu teda vieme jednoducho zriešiť tak, že najprv si nejakým ľubovoľne pomalým riešením vygenerujeme odpovede pre prvých 42 čísel, a potom len napíšeme program, ktorý podľa čísla na vstupe zvolí správnu odpoveď. Problémom bude, keď sa v neskorších sadách objavia aj zakázané čísla. Vtedy totiž taká istá otázka môže mať rôzne odpovede, v závislosti od toho, ktoré čísla sú zakázané… Časová zložitosť je $O(1)$, a pamäťová zložitosť je $O(a)$.

Bruteforce

Mohli by sme pre každé číslo na vstupe vygenerovať všetky možné podpostupnosti $2, 3, 4...a$ a pre každú otestovať, či spĺňa všetky zadané podmienky. Týchto podpostupností je rádovo 2^a. Každú z nich musíme otestovať - ako to spraviť pomerne rýchlo? Najprv overíme, či naozaj skákaním z posledného čísla skončíme na 1. To vieme overiť jediným prechodom - prejdeme postupnosť od konca, a vždy, keď narazíme na číslo, na ktorom teraz sme, pozrieme sa na jeho index a ten si odteraz pamätáme. Pokiaľ pred dorazením na 1 nenájdeme v zozname nejaké číslo, ktoré sme dostali, vieme, že postupnosť nevyhovuje. Popri tom ešte musíme kontrolovať, či neprechádzame cez zakázané číslo. Prvotný nápad je zakaždým prezrieť celý zoznam zakázaných čísel, či sa v ňom to naše nenachádza. Stačí nám ale použiť techniku dvoch bežcov: keďže zakázaný zoznam je zoradený, a čísla, ktorých prítomnosť v ňom kontrolujeme, sa vždy znižujú, stačí nám pamätať si pozíciu medzi zakázanými číslami, teda "medzeru", do ktorej by momentálne číslo patrilo. Na začiatku bude na konci, a posunieme ju bližšie k začiatku vždy, keď kontrolujeme číslo, ktoré je pod spodným okrajom tejto medzery. Takto toto miesto posúvame iba na začiatok, a teda počas celého kontrolovania postupnosti prejdeme jeho r prvkov najviac raz. Teda okontrolovať postupnosť má zložitosť $O(r + a)$. V každej z k otázok ale testujeme 2^a postupností, teda celková zložitosť algoritmu je $O(k * 2^a * (r + a))$. Pamäťová zložitosť je $O(r + a)$, lebo si pamätáme postupnosť dĺžky až $a - 1$, a r zakázaných čísel.

Optimálne riešenie

Pomôže nám dynamické programovanie. Budeme si pamätať pre všetky dvojice a, n počet postupností končiacich na a, ktoré majú práve n prvkov ($n >= 2$). Zakaždým, keď príde otázka, sčítame počty pre dané a pre všetky možné dĺžky n, ak ich už poznáme. Ak ešte nie, postupne budeme počítať tieto hodnoty pre najnižšie nevypočítané a, až kým sa k otázke nedostaneme.

Dobre teda, ale ako vypočítame odpoveď pre dvojicu a, n? Využijeme pri tom, samozrejme, odpovede pre nižšie a. Majme teda postupnosť končiacu na a dĺžky \$n%. Hneď vieme, aký bude prvý krok pri skákaní po tejto postupnosti. Keďže a je na pozícií n, skočíme na číslo n. Na akej pozícií môže byť toto n? Na akejkoľvek od 1 po $n - 1$. Prejdime všetkými a pre každú vypočítajme, koľko takých postupností existuje. Označme pozíciu, na ktorej je n, ako s. Uvedomme si, že prvých s čísel, až po číslo n, bude tiež vyhovujúcou postupnosťou (začneme na jej konci a doskáčeme na jednotku). Teda takýchto postupností je už vypočítané množstvo - dvojica n, s. Ale čo pozície po tej s-tej? Po žiadnom z jej čísel nikdy nebudeme skákať, teda môžu to byť úplne ľubovoľné čísla medzi n a a. Teda predpočítanú dvojicu n, s vynásobíme kombinačným číslom $a - n - 1$ nad $n - s - 1$. Kombinačné čísla si tiež predpočítame - zakaždým, keď potrebujeme nové, sčítavame tie pred ním, využívajúc vzťah $(a - 1nadb) + (a - 1nadb - 1) = (anadb)$.

Ešte jeden detail nám chýba - zakázané čísla. Jednoducho, vždy keď počítame počet postupností pre dvojicu a, n, tak ak náhodou a je zakázané, namiesto počítania hneď vrátime nulu. Na rozdiel od minulého riešenia, tentokrát sa čísla, na ktorých zakázanie sa pýtame, zvyšujú, tak použijeme rovnakú techniku, ako minule, len sa poľom zakázaných čísel hýbeme od začiatku po koniec.

Aká je teda zložitosť tohto riešenia? Potrebujeme najviac raz prejsť pole zakázaných čísel, teda $O(r)$. Musíme vypočítať odpovede pre všetky dvojice posledného čísla a dĺžky postupnosti, čo je $O(a^2)$ - pretože pre každú dvojicu stačí vynásobiť premennú kombinačným číslom v konštantnom čase. No a samozrejme, predpočítavame kombinačné čísla. Teda musíme vypočítať všetky kombinačné čísla až po $(anad?)$, a tých je $O(a^2)$. Teda celková časová zložitosť je $O(a^2 + r)$. Pamäťová zložitosť je tiež $O(a^2 + r)$, pretože toľko výsledkov si predpočítavame, a k tomu máme pole zakázaných čísel.

fezjo

7. Nevítaný hosť s chlpatým kožuchom

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

Vzorák hádam niekedy bude.

Eliška

8. Aquamarínové guličky

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

Prerekvizity: Kruskalov algoritmus (vedieť, ako a prečo funguje), union-find, binary-lifting (budeme používať takú tú tabuľku čo staviame keď chceme hladať LCA) a DFS.

Implementácia tejto úlohy je pomerne náročná a je ľahké sa v nej zamotať, ak si po prečítaní vzoráku stále nie si istý, ako na to, odporúčam začať preriešením si nasledujúcich úloh z testovača KSP: Sneh, Kostry, Počet komponentov, Najnižší spoločný predok, Vzdialenosť v strome.

A už dosť rečí, ide sa na vec!

Dolný odhad na odpoveď

Najprv sa zamyslíme, že koľko hrán musíme pridať po odobratí vrchola i a všetkých hrán, ktoré s ním susedia.

Nech jeho stupeň (teda počet hrán, ktoré s ním susedia) je d_i. To znamená, že máme d_i komponentov, ktoré musíme pospájať. Ľahko si rozmyslíme, že vždy vieme pridať hranu ktorá spojí dva komponenty, a zároveň nikdy nevieme hranou naraz spojiť viac ako dva komponenty, preto v optimálnom riešení pridáme $d_i - 1$ hrán.

Teraz musíme určiť, aký najmenší súčet hrán novo pridaných hrán vieme dosiahnuť. V prvom rade, keďže každá hrana spájajúca dva rôzne vrcholy má cenu aspoň 1, celkový súčet musí byť aspoň $d_i - 1$. V prípade, že i nie je 1 ani n, vieme urobiť ďalšie jednoduché pozorovanie: ak neexistuje hrana, ktorá spája vrchol s číslom menším ako i s vrcholom s číslom väčším ako i, tak jedna novo pridaná hrana bude mať určite cenu aspoň dva. Prečo? Môžeme si uvedomiť, že každá takáto hrana má cenu aspoň dva, no a zároveň určite aspoň jednu takúto hranu potrebujeme, ináč by nový graf nebol súvislý (vieme ho rozdeliť na tie malé a na tie veľké vrcholy). To znamená, že v tomto prípade celkový súčet musí byť aspoň d_i.

Vieme nájsť aj nejaké ďalšie dolné ohraničenia na to, že aká veľká musí byť tá výsledná cena? Nie? Ako uvidíme neskôr, tento odhad je skutočne optimálny. Zároveň existuje niekoľko spôsobov ako implementovať vyššie opísané počítanie hrán a cien, podľa presnej časovej zložitosti môžeme za toto riešenie získať 2 až 4 body.

Ako skonštruovať hľadanú množinu hrán?

Najprv taká úvaha na úvod, aký algoritmus by sme asi mohli použiť na postavenie toho nového grafu? Pridávame hrany tak aby sme mali súvislý graf a chceme čo najmenšiu cenu... To znie presne ako Kruskalov algoritmus. Čo by sme teda mohli urobiť je, že nejakým spôsobom nájdeme množinu hrán a potom na nich pustíme Kruskalov algoritmus aby našiel najlacnejší spôsob, ako vybrať podmnožinu hrán tak, aby bol výsledný graf súvislý.

Jeden jednoduchý príklad na takúto množinu sú hrany $(1,2),(2,3),...,(i-1,i+1),...,(n-1,n)$. Ľahko si môžeme rozmyslieť, že keďže sami o sebe tvoria súvislý graf (až na odstránený vrchol i), tak ak na rozpadnutom grafe + tejto množine spustíme Kruskalov algoritmus, dostaneme tiež súvislý graf. Dôležité pozorovanie: Iné hrany ako tie v tejto množine sa nám ani neoplatí používať. Skutočne, ak by sme v optimálnom riešení mali hranu medzi $(a,b), a+1<b$ (pre jednoduchosť uvažujme prípad $b<i$, ostatné sa dokazujú podobne), mohli by sme ju nahradiť hranami $(a,a+1),(a+1,a+2),...,(b-1,b)$, cena sa nezmení a ak boli predtým dva vrcholy spojené, určite budú spojené aj teraz. No lenže graf bol súvislý už aj predtým, a my sme teraz nejaké hrany pridali, to znamená, že sa vytvoril aspoň jeden nový cyklus. To znamená, že nejaké z nových hrán môžeme odstrániť, graf bude naďalej súvislý a dokonca ešte lacnejší a to je spor s tým, že sme predpokladali, že riešenie bolo optimálne.

Teraz už vieme urobiť $O(n^2 \log n)$ riešenie za 4 body - postupne skúsime odobrať každý vrchol i a hrany s ním susediace, ostatným pôvodným hranám priradíme cenu 0, následne pridáme hrany $(1,2),(2,3),...,(i-1,i+1),...,(n-1,n)$ (každej priradíme cenu rovnú jej kontrastu) a nájdeme minimálnu kostru, ako sme ukázali, to je optimálna množina hrán ktorú chceme v grafe mať po tomto odobraní. Ak ešte tieto sady vyriešené nemáš, odporúčam si ísť toto riešenie ihneď naprogramovať skôr než sa pustíš do čítania zvyšku vzoráku. Môže sa ti neskôr hodiť aj ako bruteforce, proti ktorému možno testovať nefungujúce rýchlejšie riešenia.

Vzorové riešenie

Ako uvidíme zachvíľu, riešenie na plný počet bodov má v princípe rovnakú myšlienku, ale nemôžeme si dovoliť zakaždým staviať také veľké kostry. Označme si množiny vrcholov patriacich do toho istého komponentu po odobratí vrcholu i ako $C_i(1), C_i(2), ..., C_i(d_i)$. Prichádza asi najtrikovejšie pozorovanie celej úlohy: Existuje optimálne riešenie, ktoré pridáva len hrany spomedzi $(min(C_i(1))-1, min(C_i(1)), ..., (min(C_i(d_i))-1, min(C_i(d_i)), (max(C_i(1))-1, max(C_i(1)), ..., (max(C_i(d_i))-1, max(C_i(d_i)), (i-1, i+1)$.

Odporúčam sa v tomto momente zastaviť, nakresliť si niekoľko príkladov, že ako táto množina vyzerá pre konkrétne grafy a pokúsiť sa najprv samostatne dokázať, prečo je optimálna.

Dôkaz trikového pozorovania: Najprv ukážeme, že po pridaní všetkých navrhovaných hrán bude graf opäť súvislý. Ak by to nebola pravda, ozančme si a najmenší vrchol taký, že vrcholy a a $a+1$ sú v rôznych komponentoch. Ak $a \neq i, i-1$ tak potom $a+1$ je zjavne najmenší vrchol vo svojom komponente (lebo zjavne všetky vrcholy $1,...,a$ sú v rovnakom komponente) a preto existuje hrana z $a+1$ do a ktorú sme mohli pridať a zvýšiť tým počet súvislých komponentov. Ak $a = i$ alebo $a = i+1$, spor sa ukáže veľmi podobne, stačí sa pozrieť buď na dvojicu $(a, a+2)$, alebo na dvojicu $b, b+1$, kde b je druhý najmenší vrchol taký, že b a $b+1$ nie sú v rovnakom komponente. Teraz už len stačí ukázať, že kostra ktorú nájdeme bude skutočne najlacnejšia. To si vieme rozmyslieť tak, že sa pozrieme na to, čo spoja hrany dĺžky 1 - uvidíme, že dva komponenty, a teda potreba pridať hranu dĺžky 2, vznikne len a len vtedy, ak v pôvodonom grafe neexistovala hrana spájajúca

vrchol s menším číslom ako i a vrchol s väčším číslom ako i, teda cena nájdená týmto riešením sa zhoduje s dolným odhadom na cenu nájdeným na začiatku vzoráku.

Na to, aby sme efektívnejšie vedeli simulovať hladanie kostry ju nebudeme staviať na celom pôvodnom grafe, ale len na susedoch vrcholu i. Keď rozmýšlame, či pridať nejakú hranu, nahradíme čísla vrcholov jej koncov číslami susedov vrcholu i, do ktorých komponentov tieto vrcholy patria. Toto vieme robiť pomocou techniky známej ako binary lifting.

Pre samotný Kruskalov algoritmus si postavíme union find, ale aby sme nemuseli to pole predkov a veľkostí vždy vytvoriť nanovo, použijeme zakaždým to isté pole, ale medzi jednotlivými iteráciami mazania vrcholov vždy napravíme tie hodnoty, ktoré sme práve menili (čiže políčka zodpovedajúce susedom vrcholu i).

Tiež si na samotnom začiatku pre každú hranu spočítame minimum a maximum v dvoch komponentoch ktoré dostaneme po jej odobratí, toto sa dá napríklad pomocou dvoch DFS: zakoreníme si strom v ľubovoľnom vrchole a najprv spočítame minimum a maximum pre podstrom každého vrcholu a potom pomocou nich spočítame tieto hodnoty aj pre tie "opačné" podstromy.

Vďaka tomuto predpočítavaniu vieme nájsť riešenie po zmazaní vrcholu i v časovej zložitosti $O(d_i \log n)$, čo sa sčíta na celkovú časovú zložitosť $O(n \log n)$.

Pamäťová zložitosť je tiež $O(n \log n)$, bottle-neck je pole predpočítaných hodnôt pre binary lifting.