Riešenia kategórie A

A-II-1 Hotel 2

Základom asi všetkých dobrých riešení je pozorovanie, že úlohu je vhodné riešiť začínajúc na najvyššom poschodí hotelu. Optimálne riešenie si môžeme popísať nasledovne: Predstavme si, že sa na začiatku každý hosť vyvezie výťahom na najvyššie poschodie, na ktorom môže bývať. Následne príde menežer hotela a postupne pre každé poschodie, *idúc zhora nadol*, zopakuje nasledovnú procedúru: Ak tu čaká nanajvýš n hostí, ubytuje ich všetkých. Ak nie, ubytuje tých n, ktorí mu zaplatia najviac, a ostatných pošle o poschodie nižšie.

Dokážme si teraz, že vyššie uvedený postup naozaj vždy vyrobí optimálne riešenie.

Zamyslime sa najskôr, prečo naše riešenie funguje pre najvyššie poschodie hotela. Nech C je množina tých ľudí, ktorých by tam ubytoval náš algoritmus. Tvrdíme, že *existuje optimálne riešenie, v ktorom na najvyššom poschodí ubytujeme práve ľudí z množiny C.*

Prečo? Ukážeme, že ľubovoľné *optimálne* riešenie vieme bez zmeny zisku prerobiť na také, v ktorom platí naše tvrdenie. Majme teda ľubovoľné optimálne riešenie. Ak sú v ňom všetci ľudia z C ubytovaní na najvyššom poschodí, sme hotoví.[1] V opačnom prípade musí existovať nejaký človek c z C, ktorý tam ubytovaný nie je. Tu rozoberieme tri prípady:

- Človek c v našom optimálnom riešení býva na nejakom nižšom poschodí a na najvyššom poschodí máme voľnú izbu. V tomto prípade človeka c do takej izby presunieme, čím sa zisk nezmení.
- Človek c v našom optimálnom riešení býva na nejakom nižšom poschodí, ale najvyššie poschodie je plné. V tomto prípade máme na najvyššom poschodí človeka d, ktorý do C nepatrí. Ľudí c a d vymeníme. Tým sa zjavne zisk nezmení a opäť dostaneme platné riešenie: c môže bývať na najvyššom poschodí, lebo patrí do C, no a d sme presunuli na nižšie poschodie ako pôvodne obýval.
- Človeka c sme v našom optimálnom riešení neubytovali. Najvyššie poschodie musí byť v našom riešení zjavne plné, inak by sme naň mohli človeka c pridať a mať lepšie riešenie. Preto opäť existuje človek d ako v predchádzajúcom bode. A keďže (vďaka výberu množiny C) človek c platí aspoň toľko ako človek d, dostaneme aspoň tak dobré riešenie, ak namiesto d ubytujeme c.[2]

Opakovaním vyššie uvedeného postupu teda zjavne vieme ľubovoľné optimálne riešenie prerobiť na také optimálne riešenie, v ktorom na najvyššom poschodí bývajú práve ľudia z množiny C. Tým sme náš dôkaz úspešne ukončili. Vieme teda, že náš algoritmus na najvyššom poschodí hotela nič nepokazí.

Tým je ale dokázaná aj celková správnosť nášho algoritmu! Akonáhle sme totiž ubytovali vybraných hostí na najvyššom poschodí, môžeme jednoducho zabudnúť na to, že toto poschodie a títo hostia existujú. Tým dostávame pôvodný problém, ale už len pre zvyšok hotela a zvyšok hostí. A teda môžeme dokola pre každé poschodie opakovať tú istú úvahu.

Poslednou otázkou je efektívna implementácia vyššie uvedeného postupu. Aby sme nemuseli na každom poschodí odznova zisťovať, ktorí hostia sú ochotní zaplatiť najviac, použijeme na uloženie aktuálnej množiny hostí haldu, v ktorej budú usporiadaní podľa sumy, ktorú sú ochotní zaplatiť. Algoritmus teda začne s prázdnou haldou a následne pre každé poschodie zhora dole vykoná nasledovné kroky:

1. Postupne vloží do haldy všetkých ľudí, ktorí majú svoju maximálnu výšku na aktuálnom poschodí.
2. Postupne vyberie z haldy (nanajvýš) n ľudí, ktorí budú ubytovaní na tomto poschodí.

Tento algoritmus vieme implementovať s časovou zložitosťou $O(p + h \log h)$.

[1] V našom tvrdení sme uviedli, že ubytujeme *práve* ľudí z C. Ľahko však nahliadneme, že to musí byť v danej chvíli pravda. Ak $|C| < n$, aj tak tam nemá kto ďalší byť, a ak $|C| = n$, už tam nie je miesto pre nikoho ďalšieho.

[2] Takáto situácia nastane len vtedy, ak c aj d platili presne rovnako, len d mal smolu, že sa do C napriek tomu nezmestil.

Listing programu (C++)

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

// na ulozenie hosti si spravime zaznam; "vacsi" host ma vacsiu prioritu na ubytovanie
struct host { int id, platba; };
bool operator< (const host &A, const host &B) { return A.platba < B.platba; }

int main() {
    // nacitame vstup
    int P, N, H;
    cin >> P >> N >> H;
    vector<int> vysky(H), platby(H);
    for (int h=0; h<H; ++h) cin >> vysky[h];
    for (int h=0; h<H; ++h) cin >> platby[h];

    // inicializujeme vystupne premenne
    long long zarobok = 0;
    vector<int> ubytovanie(H,-1);

    // umiestnime kazdeho hosta na najvyssie poschodie kde moze byvat
    vector< vector<host> > hotel(P+1);
    for (int h=0; h<H; ++h) hotel[ vysky[h] ].push_back( { h, platby[h] } );

    // postupne pre kazde poschodie zhora dole ubytuvame hosti
    priority_queue<host> Q;
    for (int p=P; p>=1; --p) {
        for (const auto &h : hotel[p]) {
            Q.push(h);
        }
        for (int n=0; n<N; ++n) if (!Q.empty()) {
            host h = Q.top();
            Q.pop();
            zarobok += h.platba;
            ubytovanie[h.id] = p;
        }
    }

    // vypiseme vystup
    cout << zarobok << "\n";
    for (int h=0; h<H; ++h) cout << ubytovanie[h] << (h+1==H ? "\n" : " ");
}
```

A-II-2 Lyžovačka

Podúlohu A ľahko vyriešime obyčajným prehľadávaním grafu. V podúlohe B k optimálnemu riešeniu povedie jednoduché použitie dynamického programovania.

Podúloha A

V tejto podúlohe vlastne potrebujeme nájsť všetky bufety, ktoré majú dve dobré vlastnosti: sú dosiahnuteľné z lokality 1 (z hornej stanice lanovky) a zároveň z nich je dosiahnuteľná lokalita 0 (dolná stanica lanovky).

Všetky lokality dosiahnuteľné z lokality 1 nájdeme tak, že z nej spustíme prehľadávanie (napr. do hĺbky alebo do šírky). Rovnakým postupom vieme nájsť aj všetky lokality, z ktorých je dosiahnuteľná lokalita 0, len tentokrát zadaný graf prehľadávame proti smeru hrán – teda ako keby sme začali na dolnej stanici lanovky a postupne prezerali, kam všade sa vieme dostať tak, že ideme hore zjazdovkami. Pri vhodnej reprezentácii grafu v pamäti má takéto riešenie časovú zložitosť $\Theta(n + z)$, teda lineárnu od veľkosti vstupu.

Listing programu (Python)

```
from queue import Queue

N, Z, B = [ int(x) for x in input().split() ]
bufety  = [ int(x) for x in input().split() ]

zjazdovky_dodola = [ [] for n in range(N) ]
zjazdovky_dohora = [ [] for n in range(N) ]
for z in range(Z):
    odkial, kam = [ int(x) for x in input().split() ]
    zjazdovky_dodola[odkial].append( kam )
    zjazdovky_dohora[kam].append( odkial )
```

```
def prehladaj(graf, start):
    # prehlada do sirky dany graf z daneho vrcholu
    # vrati pole booleanov: kam sa vieme dostat a kam nie
    navstivil = [ False for _ in range(len(graf)) ]
    navstivil[start] = True
    Q = Queue()
    Q.put(start)
    while not Q.empty():
        kde = Q.get()
        for kam in graf[kde]:
            if not navstivil[kam]:
                navstivil[kam] = True
                Q.put(kam)
    return navstivil

viem_z_1  = prehladaj( zjazdovky_dodola, 1 )
viem_do_0 = prehladaj( zjazdovky_dohora, 0 )

dobre_bufety = [ b for b in bufety if viem_z_1[b] and viem_do_0[b] ]
print( len( dobre_bufety ) )
```

Úlohu sa dá riešiť aj pomocou jediného prehľadávania do hĺbky. Keďže ide vlastne o jednoduchšiu verziu riešenia podúlohy B, odložíme si jeho vysvetlenie až k nej.

Podúloha B

Pre každú lokalitu x si môžeme položiť nasledovnú otázku: „Ak chcem začať lyžovať v lokalite x a spustiť sa odtiaľ do lokality 0, koľko najviac bufetov môžem cestou navštíviť?“ Odpoveď na túto otázku si označíme $M(x)$, pričom sa dohodneme, že ak sa z lokality x do lokality 0 vôbec nedá dostať, bude $M(x) = -\infty$.

Riešením zadanej podúlohy je hodnota $M(1)$, tú teda chceme vypočítať. Aby sme to dosiahli, vypočítame postupne vo vhodnom poradí hodnoty $M(x)$ pre všetky lokality dosiahnuteľné z lokality 1.

Ľahko zistíme $M(0)$: keďže v lokalite 0 nie je bufet a ani už nikam nelyžujeme, zjavne platí $M(0) = 0$.

Predstavme si teraz, že sme v nejakej inej lokalite x. Ako vyzerá optimálne riešenie? V prvom rade sa pozriem, či je v lokalite x bufet, a ak áno, navštívim ho. Následne sa pozriem na zjazdovky, ktoré vedú z lokality x. Lokality, kam vedú, označíme $y_1, \ldots, y_k$. My si teraz musíme vybrať jednu zo zjazdoviek a spustiť sa ňou. Ale ktorú? Ktorá nám dá najlepšie riešenie? Na to, aby sme sa správne rozhodli, potrebujeme poznať hodnoty $M(y_1), \ldots, M(y_k)$. Tie nám pre každú lokalitu y_i hovoria, koľko bufetov ešte zvládneme navštíviť, ak sa teraz spustíme na ňu. A keďže chceme bufetov navštíviť čo najviac, vyberieme si samozrejme tú nasledujúcu lokalitu, ktorej hodnota M je najväčšia. Platí teda nasledovný vzťah:

$$M(x) \;=\; [1 \text{ ak je v lokalite } x \text{ bufet}] \;+\; \max\{M(y_i) : \text{existuje zjazdovka z } x \text{ do } y_i\}$$

Tento vzťah vieme ľahko zapísať ako rekurzívnu funkciu. Aby sme dostali efektívny algoritmus, pridáme ešte *memoizáciu*: pre každé x budeme hodnotu $M(x)$ počítať len raz. Akonáhle ju zistíme, zapíšeme si ju do poľa a následne už len budeme v ďalšom výpočte používať takto uloženú hodnotu.

Po tejto úprave teda dostávame program, ktorý nanajvýš raz spracuje každú lokalitu, a takisto nanajvýš raz spracuje každú zjazdovku – totiž vždy pri spracúvaní lokality postupne prezrieme všetky zjazdovky, ktoré z nej vedú. Časová zložitosť je teda $\Theta(n + z)$. Toto riešenie je zjavne optimálne, keďže už len na načítanie vstupu potrebujeme rádovo rovnako veľa krokov výpočtu.

Listing programu (Python)

```
N, Z, B = [ int(x) for x in input().split() ]

bufety  = [ int(x) for x in input().split() ]
pocet_bufetov = [ 0 for n in range(N) ]
for b in bufety: pocet_bufetov[b] = 1

zjazdovky = [ [] for n in range(N) ]
for z in range(Z):
    odkial, kam = [ int(x) for x in input().split() ]
    zjazdovky[odkial].append( kam )

# pomocne pole v ktorom si pamatame uz vypocitane hodnoty M()
pamat = [ None for n in range(N) ]

def M(x):
    if pamat[x] is not None: return pamat[x]  # hodnotu M(x) uz pozname
```

```
    if x == 0: return 0                          # vieme ze M(0) je 0
    odpoved = float('-inf')
    for kam in zjazdovky[x]: odpoved = max( odpoved, pocet_bufetov[x] + M(kam) )
    pamat[x] = odpoved
    return odpoved

print( M(1) )
```

Alternatívne, rovnako dobré riešenie vieme implementovať aj bez použitia rekurzie. Začneme tým, že nájdeme tzv. *topologické usporiadanie* vrcholov nášho grafu – teda jedno prípustné usporiadanie lokalít do poradia podľa ich nadmorskej výšky. V tomto poradí (začínajúc od najnižšie položených) postupne každú lokalitu raz spracujeme pomocou vyššie uvedeného vzorca a vypočítanú odpoveď si zapamätáme.

A-II-3 Miešanie kariet

Pohyb jednej karty. Vezmime si jedno konkrétne eso a pozrime sa, čo sa s ním bude diať. Keď budeme balíček kariet miešať na Kleofášovom stroji stále dookola, eso v balíčku bude meniť svoju pozíciu. Možných pozícií esa v balíčku je n, teda najneskôr po n zamiešaniach sa nám musí stať, že eso sa ocitlo na pozícii, na ktorej už niekedy pred tým bolo. Keďže pozíciu esa po premiešaní ovplyvňuje iba jeho pozícia pred premiešaním, od tohto okamihu sa eso bude hýbať úplne rovnako, ako sa hýbalo, keď bolo na rovnakej pozícii naposledy. Inými slovami, jeho pozície sa budú periodicky opakovať.

Ďalším zaujímavým pozorovaním je, že keď sa eso prvý raz dostane do pozície, v ktorej už bolo, bude to určite pozícia, v ktorej bolo eso úplne na začiatku. Ak by to totiž bola nejaká iná pozícia, znamenalo by to, že sa do tejto pozície mohlo eso dostať z dvoch rôznych miest (z jedného miesta sa do nej dostalo prvý raz a z iného druhý raz). To ale nie je možné, keďže na každú pozíciu v balíčku sa vždy dostane práve jedna karta. Eso teda mení pozície periodicky už od začiatku.

Ak by nás zaujímalo, v ktorých časoch bude naše eso na nejakej konkrétnej pozícii X, stačí nám simulovať jeho pohyb krok po kroku, až kým sa nedostane naspäť do pozície, v ktorej začínalo. To sa stane určite najneskôr po n krokoch, teda celá simulácia nám bude trvať v najhoršom prípade lineárny čas (jeden krok vieme simulovať v konštantnom čase, keďže nám stačí pamätať si iba pozíciu esa a tú aktualizovať). Počet krokov potrebných na to, aby sa eso dostalo naspäť na svoju začiatočnú pozíciu (teda periódu, s ktorou sa pozície esa opakujú) označme p. Ak sa počas simulácie eso niekedy k krokov od začiatku dostalo na pozíciu X, potom na tejto pozícii bude práve vtedy, keď počet premiešaní od začiatku miešania dáva zvyšok k po delení p. Ak sa počas simulácie eso na pozíciu X, nedostalo, potom sa na túto pozíciu nedostane nikdy.

Pohyb dvoch kariet. Vezmime si teraz nejaké dve konkrétne esá (napríklad srdcové a pikové) a nejaké dve konkrétne pozície X a Y. V ktorých časoch bude srdcové eso na pozícii X a súčasne pikové eso na pozícii Y? Periódu, s ktorou sa opakuje pozícia srdcového esa, označme p_1 a periódu pikového označme p_2. Počet krokov, po ktorých sa srdcové eso prvý raz dostane na pozíciu X označme k_1 a počet krokov, po ktorých sa pikové eso prvý raz dostane na pozíciu Y označme k_2. Periódy p_1, p_2 aj čísla k_1, k_2 vieme nájsť v lineárnom čase simulovaním pohybu es (každého zvlášť). Ak niektoré z čísel k_1, k_2 neexistuje (teda niektoré z es sa nikdy nedostane na požadovanú pozíciu), potom vieme, že situácia so srdcovým esom na pozícii X a pikovým na pozícii Y nenastane nikdy. V opačnom prípade nastane práve vtedy, keď počet premiešaní od začiatku dáva zvyšok k_1 po delení p_1 a zvyšok k_2 po delení p_2.

Najmenší spoločný násobok čísel p_1 a p_2 označme m. Všimnime si, že po m krokoch od začiatku budú obe esá na rovnakej pozícii, ako boli na začiatku. Od tohoto okamihu sa budú ich pozície meniť rovnako, ako sa menili od začiatku, teda ak doteraz nenastala situácia, v ktorej je srdcové eso na pozícii X a pikové na pozícii Y, potom takáto situácia nenastane nikdy. Ak niekedy po l krokoch od začiatku takáto situácia nastane, potom sa bude opakovať každých m krokov. Skôr, ako po m krokoch sa zopakovať nemôže, lebo potom by p_1 a p_2 mali spoločného deliteľa menšieho ako m. Preto situácia, kde je srdcové eso na pozícii X a pikové na pozícii Y, nastane práve vtedy, keď počet krokov od začiatku dáva zvyšok l po delení m.

Číslo l by sme mohli nájsť tak, že pre každé číslo od 0 do $m-1$ overíme, či dáva zvyšok k_1 po delení p_1 a zvyšok k_2 po delení p_2. Efektívnejšie riešenie je overovať to len pre čísla, ktoré dávajú zvyšok k_1 po delení p_1:

postupne prejdeme čísla $k_1, p_1 + k_1, 2p_1 + k_1, \ldots, \left(\frac{m}{p_1} - 1\right) p_1 + k_1$ a pre každé z nich overíme, či dáva zvyšok k_2 po delení p_2. Keďže $\frac{m}{p_1} \leq p_2$, prejdeme najviac p_2 čísel, teda celé to bude mať časovú zložitosť $O(p_2)$, čo je v najhoršom prípade $O(n)$.

Pohyb štyroch kariet. Pozrime sa teraz na všetky štyri esá a na nejaké štyri pozície X_1, X_2, X_3, X_4. Kedy budú súčasne prvé eso na pozícii X_1, druhé na pozícii X_2, tretie na pozícii X_3 a štvrté na pozícii X_4? Periódy es si označme postupne p_1, p_2, p_3, p_4 a počty krokov, po ktorých budú prvý raz na svojej pozícii X_i označme k_1, k_2, k_3, k_4 (všetky tieto čísla vieme nájsť v lineárnom čase). Ak niektoré z čísel k_i neexistuje, potom naša situácia nenastane nikdy. Najmenší spoločný násobok p_1 a p_2 označme m_1. Už vieme v čase $O(p_2)$ zistiť, kedy bude prvé eso na pozícii X_1 a zároveň druhé na pozícii X_2 – buď to nenastane nikdy (a sme hotoví), alebo vždy, keď počet krokov dáva zvyšok l_1 po delení m_1, kde l_1 je nejaké číslo.

Kedy budú prvé tri esá na svojich pozíciách? Vždy, keď počet krokov dáva zvyšok l_1 po delení m_1 a zvyšok k_3 po delení p_3. Podobný problém sme už riešili a podobne ako predtým vieme v čase $O(p_3)$ nájsť také číslo l_2, že prvé tri esá sú na správnych pozíciach vždy vtedy, keď počet krokov dáva zvyšok l_2 po delení m_2, kde m_2 je najmenší spoločný násobok m_1 a p_3. Alebo môžeme zistiť, že také číslo l_2 neexistuje a prvé tri esá nikdy nebudú súčasne na správnych pozíciách.

Nakoniec túto úvahu ešte raz zopakujeme pre štvrté eso a v čase $O(p_4)$ nájdeme čísla l_3 a m_3 také, že všetky štyri esá sú na správnych pozíciách vtedy, keď počet krokov dáva zvyšok l_3 po delení m_3 (alebo zistíme, že esá nikdy nebudú všetky súčasne na správnych pozíciách). Celé toto zisťovanie nám trvalo lineárny čas, keďže sme urobili iba konštante veľa krát lineárne veľa práce.

V našej úlohe síce nemáme presne určené, ktoré eso má prísť na ktorú pozíciu (a v optimálnom riešení možno niektoré esá nebudú na žiadnej z vrchných piatich pozícií), je však iba konštantne veľa možností ako vybrať, ktoré esá budú medzi vrchnými piatimi kartami a pre každé z nich na ktorej konkrétnej pozícii má byť. Pre každú z týchto konštantne veľa možností vieme v lineárnom čase zistiť, či a kedy nastane a potom už iba vyberieme tú najlepšiu. Celý algoritmus má teda časovú zložitosť $O(n)$.

Možností, ako esám priradiť pozície, je síce konštantne, ale dosť veľa. Ich počet však vieme značne orezať napríklad tým, že neberieme do úvahy možnosti, kde by sa nejaké eso malo dostať na pozíciu, na ktorú sa nikdy nedostane. Tiež môžeme využiť, že ak sa dve esá vedia dostať na rovnakú pozíciu, potom musia mať rovnakú periódu, teda niektoré ich vzájomné pozície vieme vylúčiť v konštantnom čase.

Čínska zvyšková veta. Lineárne riešenie, ktoré sme si predstavili v predchádzajúcej časti, využívalo na nájdenie spoločnej periódy a offsetu trik: bolo si treba všimnúť, že každé eso má cyklus dĺžky najviac n a následne menšie cykly kombinovať do jedného veľkého vo vhodne zvolenom poradí. Za zmienku stojí, že v teórii čísel poznáme nástroj, ktorý nám umožňuje robiť takéto „spájanie cyklov“ ešte efektívnejšie. Tzv. čínska zvyšková veta (viď napr. heslo „Chinese remainder theorem“ na anglickej Wikipédii) udáva efektívny predpis ako nájsť hľadanú spoločnú periódu a offset. Najzložitejším krokom, ktorý potrebujeme pri jej použití spraviť, je nájdenie vhodných inverzných prvkov pri modulárnom násobení. Tie vieme efektívne nájsť pomocou rozšíreného Euklidovho algoritmu na hľadanie najväčšieho spoločného deliteľa. Pomocou tohto nástroja by sa dalo jednotlivé periódy našich štyroch es skombinovať do jednej dokonca v čase $O(\log n)$. (Toto ale v našej úlohe nebolo potrebné, keďže sme predtým aj tak museli stráviť $\Theta(n)$ času nájdením cyklov v permutácii kariet.)

A-II-4 Sufixové stromy

V oboch podúlohách sa stačilo „správne pozrieť“ na vhodný sufixový strom. Ukážeme si, ako na to.

Podúloha A: najdlhší spoločný podreťazec

V domácom kole sme zistili, že keď máme písmenkový strom obsahujúci všetky sufixy reťazca S, tak každý jeho vrchol zodpovedá jednému z reťazcov, ktoré sa v S aspoň raz vyskytujú ako podreťazce. (Odborne hovoríme, že ide o bijekciu medzi podreťazcami a vrcholmi stromu.)

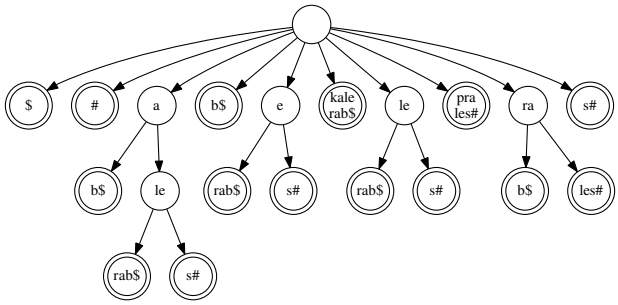
Ekvivalentne, podreťazce reťazca S sú presne všetky tie reťazce, ktoré si vieme prečítať na nejakej ceste z koreňa stromu dodola.

V sufixovom strome (ktorý vznikne skomprimovaním hrán vo vyššie spomínanom písmenkovom strome) teda každý podreťazec reťazca S tiež zodpovedá nejakej ceste z koreňa stromu dodola, len tentokrát pre niektoré podreťazce môže táto cesta končiť niekde na jednej z hrán.

Podúlohu A by sme vedeli vyriešiť tak, že si postavíme dva samostatné sufixové stromy (jeden pre S a druhý pre T) a následne šikovným spôsobom oba naraz prehľadáme. Omnoho ľahšie implementovateľné riešenie však dostaneme použitím triku spomínaného v študijnom texte: rozšíreného sufixového stromu.

Naše riešenie začneme tým, že si postavíme sufixový strom pre reťazec $S\texttt{\$}T\texttt{\#}$. Podotýkame, že znaky `$` a `#` sa nevyskytujú v reťazcoch S a T. Slúžia nám ako značky ich koncov. Navyše zabezpečujú, že každému sufixu reťazcov $S\texttt{\$}T$ a T zodpovedá v našom sufixovom strome nejaký list – pozri v študijnom texte časť „Sufixový strom so zarážkou“.

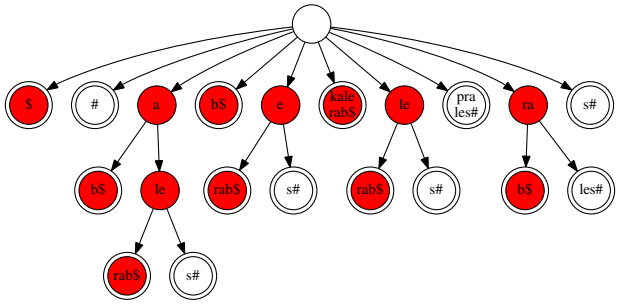
Z tohto stromu navyše odignorujeme všetko, čo sa nachádza pod výskytmi zarážky `$`. Teda ak sme mali hranu na ktorej bolo napísané „…T`#`“, budeme sa tváriť, že je na nej napísané len „…`$`“. Takto dostaneme sufixový strom, ktorý je akoby zjednotením dvoch sufixových stromov: jedného pre reťazec S`$` a druhého pre reťazec T`#`. Takto by tento strom vyzeral pre reťazce $S =$ `kalerab` a $T =$ `prales` z príkladu v zadaní:



Obr. 1: Zovšeobecnený sufixový strom pre reťazce `kalerab` a `prales`.

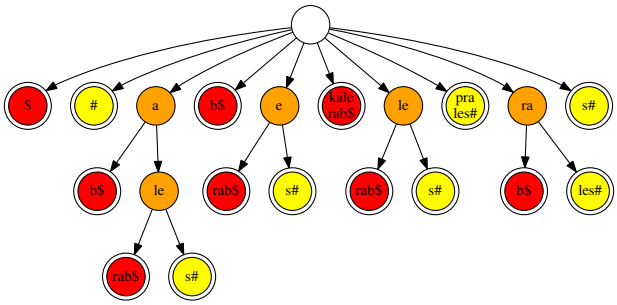
V obrázku sme spravili zmenu pre lepšiu čitateľnosť: namiesto toho, aby sme reťazce písali na hrany, ktorým zodpovedajú, píšeme každý reťazec do toho vrchola, do ktorého zodpovedajúca hrana vedie.

Ako teraz nájsť najdlhší spoločný podreťazec? Začneme tým, že si napríklad na červeno ofarbíme všetky vrcholy, ktoré predstavujú podreťazce reťazca S`$`. Toto vieme spraviť jedným prehľadaním sufixového stromu do hĺbky: list ofarbíme ak reťazec na hrane doň vedúcej končí `$`, vnútorný vrchol stromu ofarbíme ak sme ofarbili aspoň jedného z jeho synov.



Obr. 2: Na červeno ofarbené vrcholy predstavujú podreťazce reťazca `kalerab$`.

Následne napríklad na žlto ofarbíme všetky vrcholy predstavujúce podreťazce reťazca T`#`. No a čo takto dostaneme? Vrcholy, ktoré sme ofarbili aj na červeno aj na žlto (volajme ich oranžové) nám predstavujú všetky podreťazce, ktoré sa vyskytujú aj v S`$` aj v T`#` – inými slovami, spoločné podreťazce reťazcov S a T.



Obr. 3: Žlté vrcholy sú podreťazce ležiace len v `prales#`, oranžové sú spoločné.

No a na vyriešenie našej úlohy nám už len stačí nájsť oranžový vrchol ležiaci najďalej od koreňa. To vieme spraviť buď tretím prehľadávaním, alebo priamo počas druhého prehľadávania: vždy keď ideme zafarbiť na žlto doteraz červený vrchol, pozrieme sa, či sme nenašli nové najlepšie riešenie.

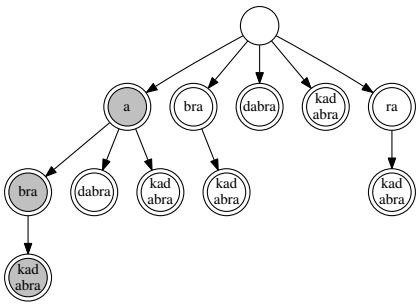
Podúloha B: najkratšia perióda

V tejto podúlohe hľadáme najmenšie p také, že reťazec S presne pasuje na reťazec „S posunutý o p pozícií doprava“. Pozrime sa na písmená, ktoré majú pôvodné S a posunuté S spoločné. Ak má pôvodné S dĺžku n, spoločný podreťazec má dĺžku $n-p$. Keďže leží na konci pôvodného S, ide o niektorý zo sufixov S. A keďže leží na začiatku posunutého S, ide zároveň aj o prefix S.

A platí to aj naopak: vždy, keď pre nejaké p platí, že prvých $n-p$ písmen reťazca S je rovnakých ako posledných $n-p$ písmen, je p periódou daného reťazca. Ak teda chceme nájsť najkratšiu periódu daného reťazca, hľadáme vlastne najdlhší jeho sufix ktorý je zároveň jeho prefixom.

Ako to spravíme, keď máme sufixový strom pre reťazec S?

Začneme tým, že nájdeme najhlbší vrchol v celom strome – inými slovami, cestu z koreňa do listu, ktorá zodpovedá celému reťazcu S. Prefixom tejto cesty zodpovedajú práve všetky prefixy reťazca S. Ktoré z nich sú zároveň sufixami? V našom strome máme predsa konce všetkých sufixov označené. Stačí teda na našej ceste nájsť najhlbší označený vrchol (iný ako list, ktorým cesta končí).



Obr. 4: Sufixový strom pre reťazec `abrakadabra`. Vyfarbené vrcholy zodpovedajú ceste na ktorej leží celé toto slovo. Najhlbší nie-list na nej v ktorom končí nejaký sufix je vrchol zodpovedajúci podreťazcu `abra`. Keďže celý reťazec má dĺžku 11 a reťazec `abra` dĺžku 4, má najkratšia perióda dĺžku $11-4=7$.

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Eduard Batmendijn, Michal Forišek, Vladimír Macko
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2016