# 60. ročník Matematické olympiády – 2010/2011

*Řešení úloh domácího kola kategorie P*

---

## P-I-1 Indiana a poklad

Nejjednoduší řešení je následující – budeme si udržovat setříděnou posloupnost posledních $K$ prvků a vždy vypíšeme prostřední. To uděláme snadno tak, že si budeme navíc pamatovat $K$ posledních načtených prvků. Potom můžeme ze setříděné množiny vypustit $K$-tý nejstarší prvek, který máme uložený, a zároveň do ní přidáme nově načtený prvek. Pokud budeme setříděnou množinu udržovat například pomocí pole a posloupnost posledních $K$ načtených prvků pomocí fronty, tak zjevně dosáhneme časové složitosti $\mathcal{O}(NK)$, protože zatřídění případně odebrání prvku ze setříděného pole zvládneme v čase $\mathcal{O}(K)$.

S tím bychom se ovšem neměli spokojit. Podívejme se na problém z trochu jiné strany. Zkusme posledních $K$ čísel setřídit a rozdělit na $\lfloor K/2 \rfloor + 1$ menších čísel a $\lfloor K/2 \rfloor$ čísel větších. Zjevně medián potom získáme jako největší z menších čísel. Přidání dalšího čísla pak vyřešíme tak, že jím nahradíme odebírané číslo a porovnáme maximum z menších (označme jej jako $M$ a minimum z větších čísel $V$). Mohou nastat dvě možnosti – $M \leq V$. Pak je vše v pořádku a posledních $K$ čísel máme korektně rozdělených na dvě části nebo $M > V$ a pak máme problém. Pokusme se jej vyřešit tak, že $M$ a $V$ odebereme ze své množiny a vložíme do té druhé. Získáme tak korektní rozdělení? Je velmi důležité si uvědomit, co se přidáním dalšího prvku mohlo pokazit – do množiny větších čísel jsme přidali číslo příliš malé nebo analogicky pro menší čísla. Podstatné je, že takové číslo bylo pouze jedno. Pokud jej přemístíme do správné množiny, tak všechna ostatní čísla již jsou ve správných množinách a proto výše uvedená operace skutečně postačuje.

Trochu jiná otázka je, zda jsme si pomohli. Pokud množiny implementujeme přímočaře, tak nalezení a nahrazení starého prvku potrvá $\mathcal{O}(K)$ a stejně tak prohození minima a maxima obou množin. Celková složitost bude tedy $\mathcal{O}(NK)$ a jsme zdánlivě tam, kde o odstavec dřív. Nicméně zdání klame a od vítězství nás dělí jen několik technických triků. Tím prvním je, že hlavní operace, které od námi udržovaných množin čísel požadujeme je přidání prvku a nalezení minima a maxima. Na to se skvěle hodí haldy (viz např. na *http://ksp.mff.cuni.cz/tasks/18/cook3.html*). Bohužel odebírat z haldy jiný prvek než z jejího vrcholu není zcela přímočaré, protože bychom museli odebíraný prvek nejprve najít, a tak musíme trochu upravit algoritmus – nebudeme prvky z haldy odebírat hned. Všimneme si, že prvek, který v haldě už nemá být, nám nevadí pokud není v jejím vrcholu. Pokud se ale takový prvek do vrcholu dostane, tak už jej umíme snadno z haldy odstranit. Takže algoritmus ještě upravíme tak, že před porovnáním hodnot ve vrcholu hald zkontrolujeme, zda už nejsou příliš staré. Pokud ano tak je zahodíme a zkusíme to znovu. Abychom mohli zjišťovat, zda prvek je ještě aktuální, budeme si u něj muset pamatovat jeho index. Se složitostí to teď bude trochu složitější. První pohled opět klame – po přidání nového prvku můžeme až řádově $N$-krát odebrat vrchol haldy, protože bude příliš zastaralý

a celková složitost se tedy zdá býti $\mathcal{O}(N^2 \log N)$ (odebrání vrcholku haldy trvá čas $\mathcal{O}$(log velikosti haldy) a v haldě může být až $\mathcal{O}(N)$ prvků). Při bližším zkoumání ale zjistíme, že něco takového se nemůže stávat příliš často. Uvědomíme-li si, že každý prvek je do haldy přidán jen jednou a vyřazen kvůli stáří také jen jednou, zjistíme, že celkový počet takových vyřazení je $N$ a tedy i celková složitost je $\mathcal{O}(N \log N)$. Ti z vás co se nyní s jásotem vrhají ke klávesnicím, aby si to zkusili naprogramovat požádám o chvilku strpení, protože si ukážeme ještě jedno vylepšení.

Asi každého z vás napadlo, že problém je v haldách obsahujících mnoho nezajímavých prvků. Pokud bychom se jich uměli rozumně zbavit, dokázali bychom omezit velikost hald na $K$. Jak na to? Pokud si někde bokem budem pro každý prvek pamatovat jeho pozici v haldě, není problém ho smazat. Toto je spíše pracné než zajímavé, protože při libovolném pohybu prvku v haldě je potřeba tyto pozice aktualizovat. Pokud známe pozici vypouštěného prvku v haldě, můžeme ho nahradit přidávaným prvkem. Nakonec je potřeba opravit jeho pozici, což znamená ho zabublat hlouběji nebo naopak vybublat výše dle potřeby – to ovšem zvládneme v $\mathcal{O}$(log velikosti haldy). No a protože touto úpravou jsme velikost haldy omezili na maximálně $K$, tak výsledná složitost bude $\mathcal{O}(N \log K)$. Paměť potřebujeme pouze na udržení hald což je $\mathcal{O}(K)$, protože zapamatování si maximálního mediánu jakož i další pomocné proměnné nám spotřebuje pouze konstantní množství paměti.

```
program indiana;

const
   MAXK = 100000;

type

   { Datový typ pro haldu }
   THalda = record
      pole  : array[ 1..MAXK div 2 + 1 ] of record
         hodnota : longint;
         id : longint; { Index do odpovídající položky v poli TIndexy }
      end;
      pocet : longint;
   end;

   { Každý prvek si pamatuje index ve své haldě }
   TIndexy = array [ 1..MAXK ] of record
      index : longint;
      halda : (horni, dolni);
   end;

procedure Vymen(var a,b : longint);
var
   x : longint;
begin
   x:=a;
   a:=b;
   b:=x;
end;

{ Opraví podstrom v haldě }
procedure Dolu(var halda : THalda; pozice : longint; var indexy : TIndexy);
```

```
var
   potomek         : longint;
   pokracovat      : boolean;
begin
   with halda do begin
      pokracovat:= (2<=pocet) and (2*pozice<=pocet);

      while Pokracovat do begin
         potomek:=2*pozice;

         if potomek < pocet then
            if pole[potomek+1].hodnota < pole[potomek].hodnota then
               potomek:=potomek+1;

         if pole[pozice].hodnota > pole[potomek].hodnota then begin
            Vymen(pole[pozice].hodnota, pole[potomek].hodnota);
            Vymen(pole[pozice].id, pole[potomek].id);
            Vymen(indexy[pole[pozice].id].index, indexy[pole[potomek].id].index);

            pozice:=potomek;
            pokracovat:=2*pozice<=pocet
         end else
            pokracovat:=false
      end;
   end;
end;

{ Opraví cestu ke kořenu haldy }
procedure Nahoru(var halda : THalda; pozice : longint; var indexy : TIndexy);
var
   otec        : longint;
   pokracovat  : boolean;
begin
   with halda do begin
      pokracovat:=pozice>1;

      while Pokracovat do begin
         otec:=pozice div 2;
         if pole[pozice].hodnota < pole[otec].hodnota then begin
            Vymen(pole[pozice].hodnota, pole[otec].hodnota);
            Vymen(pole[pozice].id, pole[otec].id);
            Vymen(indexy[pole[pozice].id].index, indexy[pole[otec].id].index);
            pozice:=otec;
            pokracovat:=pozice>1
         end else
            pokracovat:=false;
      end;
   end;
end;

{ Zajistí, aby maximum v dolní haldě bylo menší nebo rovno minimu v horní haldě }
procedure Vyvaz(var dHalda, hHalda : THalda; var indexy : TIndexy);
var
   pomoc : longint;
begin
   if (hHalda.pocet = 0) then exit;
   while -dHalda.pole[1].hodnota > hHalda.pole[1].hodnota do begin
      indexy[dHalda.pole[1].id].halda := horni;
```

```
      indexy[hHalda.pole[1].id].halda := dolni;

      Vymen(dHalda.pole[1].id, hHalda.pole[1].id);

      pomoc:=-dHalda.pole[1].hodnota;
      dHalda.pole[1].hodnota := -hHalda.pole[1].hodnota;
      hHalda.pole[1].hodnota := pomoc;

      Dolu(dHalda, 1, indexy);
      Dolu(hHalda, 1, indexy);
   end;
end;

{ Vrátí aktuální medián }
function Median(var dHalda, hHalda : THalda; var indexy : TIndexy) : longint;
begin
   { Protože nový prvek mohl prijít do jiné haldy, než do té, do které má patřit,
     je nutné haldy vyvážit a teprve poté spočítat medián }
   Vyvaz(dHalda, hHalda, indexy);
   Median := -dHalda.pole[1].hodnota;
end;

var
   N, K, dPocet, hPocet : longint;
   i, j : longint;

   { Dolní halda obsahuje všechna čísla negovaná, takže se chová jako maximální,
     horní halda se chová jako maximální }
   dHalda, hHalda : THalda;

   { Seznam indexu jednotlivých prvku v haldě }
   indexy  : TIndexy;
   zacatek : longint;
   pomoc   : longint;
   maximum : longint;

begin
   readln(N,K);
   dPocet:=K div 2 + 1; { Dolní halda obsahuje vždy o jeden prvek víc }
   hPocet:=K div 2;

   dHalda.pocet := dPocet;
   for i:=1 to dPocet do begin
      read(j);
      dHalda.pole[i].hodnota := -j;
      dHalda.pole[i].id:=i;
      indexy[i].index := i;
      indexy[i].halda := dolni;
   end;

   hHalda.pocet := hPocet;
   for i:=1 to hPocet do begin
      read(hHalda.pole[i].hodnota);
      hHalda.pole[i].id:=i+dPocet;
      indexy[i+dPocet].index := i;
      indexy[i+dPocet].halda := horni;
   end;

   { Vystavení obou hald }
   for i:=dPocet div 2 downto 1 do
```

```
      Dolu(dHalda, i, indexy);

   for i:=hPocet div 2 downto 1 do
      Dolu(hHalda, i, indexy);

   zacatek := 1;

   { První medián poslouží jako dočasné maximum }
   maximum := Median(dHalda, hHalda, indexy);

   for i:=K+1 to N do begin
      read(j);

      { Nahrazení nejstaršího prvku z jeho haldy nejnovějším prvkem }
      if indexy[zacatek].halda = dolni then begin
         dHalda.pole[indexy[zacatek].index].hodnota:=-j;
         Nahoru(dHalda, indexy[zacatek].index, indexy);
         Dolu(dHalda, indexy[zacatek].index, indexy);
      end else begin
         hHalda.pole[indexy[zacatek].index].hodnota:=j;
         Nahoru(hHalda, indexy[zacatek].index, indexy);
         Dolu(hHalda, indexy[zacatek].index, indexy);
      end;

      pomoc := Median(dHalda, hHalda, indexy);
      if pomoc > maximum then
         maximum := pomoc;

      inc(zacatek);
      if zacatek > K then
         zacatek := 1;
   end;

   writeln(maximum);
end.
```

## P-I-2 Poklad podruhé

Nejdříve trochu přeformulujme zadání úlohy, aby byl další výklad srozumitelnější. Dlaždice chodby budeme dále nazývat políčka a cílem bude spočítat, kolika způsoby lze pokrýt chodbu dlaždicemi velikosti $1 \times 2$ tak, aby žádná dlaždice neležela na zakázaném políčku. Dále budeme předpokládat, že chodba má vždy tři řádky délky $N$ políček a každý sloupeček má právě 3 políčka.

První myšlenka, která každého jistě napadne, je jednoduše vygenerovat všechny možnosti jak chodbu vydláždit a tyto možnosti jednoduše spočítat. Takový algoritmus by vypadal např. tak, že v jednom kroku vyzkoušíme všechny možnosti, jak zcela pokrýt např. nejpravější nepokrytý sloupeček. Pro každou takovou možnost rekurzivně spočítáme, kolika způsoby lze pokrýt zbývající část chodby. Tato část je už ale o jeden sloupeček kratší, takže se postupně dostaneme k chodbě s nulovou délkou, u které je výsledek triviální. Výsledky rekurzivních volání funkce sečteme a součet pak tvoří výslednou hodnotu.

Během pokrývaní sloupečku si jenom musíme dát pozor na to, abychom nepokryli i zakázané políčko ve vedlejším sloupečku. To je ale v tuto chvíli jenom implementační detail.

Protože výška sloupečku je vždy 3 políčka, je počet možností, jak jej zcela pokrýt konstantně mnoho. Rozborem případů zjistíme, že pro libovolnou konfiguraci sloupečku dokonce existují nejvýše 3 možnosti. Každý krok tedy může vygenerovat nejvýše tři možnosti a časová složitost tak bude nejvýše $\mathcal{O}(3^N)$. Složitost je tedy exponenciální. Snadno si ale všimneme, že během výpočtu mnohokrát voláme rekurzivní funkci se stejnými parametry. Právě tato skutečnost způsobuje neefektivitu algoritmu, neboť opakujeme stejný výpočet. V takových situacích je vhodné pamatovat si v nějaké tabulce již získané výsledky volání funkce a při jejím zavolání se nejdříve podívat do ní, jestli požadovaný výsledek již není spočítaný. Výpočet pustíme jenom tehdy, pokud výsledek spočítaný ještě není. V opačném případě vrátíme jako výsledek zapamatovanou hodnotu.

Protože postupujeme systematicky a v jednom kroku vždy pokryjeme jeden celý sloupeček, je snadné takovou tabulku sestrojit. Každá chodba, kterou se snažíme pokrýt má totiž specifický tvar. První část je nepokrytá, následuje sloupeček, který pokrýváme a za ní je již chodba zcela pokrytá. Počet možností nevydlážděného sloupečku je $2^3$, takže potřebujeme tabulku $N \times 8$ hodnot.

Na výpočet jedné hodnoty tabulky je potřeba konstantní čas, pokud máme spočítané všechny hodnoty, na kterých aktuální hodnota závisí (zkoušíme maximálně 3 možnosti). Na výpočet těchto hodnot ale za stejných podmínek opět potřebujeme konstantní čas. Spočítat, kolika způsoby lze vydláždit chodba, jejíž délka je nulová, je možné opět v konstantním čase, takže celková časová složitost bude lineární vzhledem k počtu hodnot v tabulce a tento počet je lineární vzhledem k $N$.

Tento algoritmus je téměř učebnicovou ukázkou algoritmu, který lze řešit pomocí dynamického programování. Pokud totiž máme rekurzivní algoritmus, u kterého si lze pamatovat mezivýsledky dílčích výpočtů, tak je možné směr výpočtu obrátit. V tomto případě tak nebudeme postupně zprava zmenšovat vydlážděnou chodbu, ale zleva zvětšovat vydlážděnou chodbu.

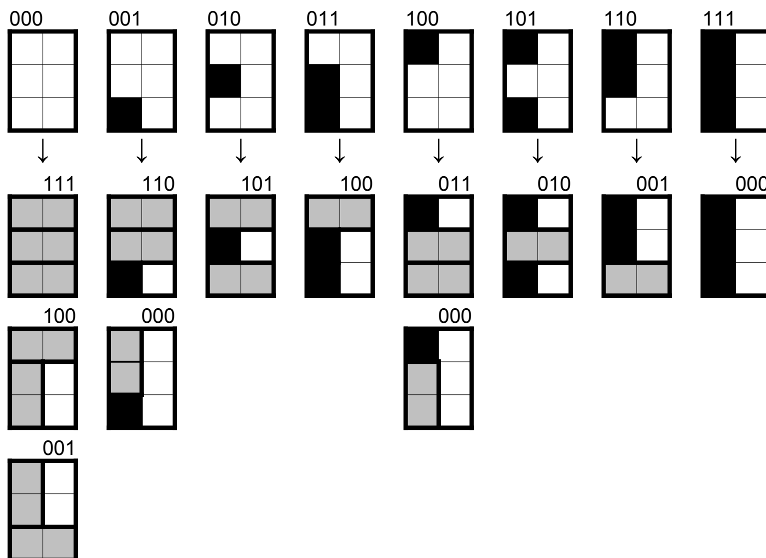
V tabulce si tedy nebudeme pamatovat, kolika způsoby lze vydláždit zbytek chodby, ale kolika způsoby jsme dosáhli takto vydlážděné chodby (tato čísla jsou hodnotově stejná, jen se na ně díváme z jiné strany).

Toto obrácení výpočtu má velkou výhodu v tom, že není třeba žádná rekurzivní funkce. Obecně je volání funkce časově náročná operace. Navíc každé volání spotřebuje určité množství paměti na zásobníku programu a pokud je hloubka rekurze vysoká, může potřebná paměť překročit velikost zásobníku a program skončí se známou chybou „Stack overflow“. Navíc i samotná implementace algoritmu bývá jednodušší.

Algoritmus je pak jednoduchý. Postupujeme zleva a v každém kroku zvětšíme velikost vydlážděné plochy o jeden sloupeček. Do tabulky si průběžně zapisujeme počet způsobů, kterými lze vydláždit chodbu, které končí aktuálním sloupečkem (počítáme samozřejmě pro všechny možné konfigurace tohoto sloupečku).

Abychom nemuseli pracně rozebírat všechny konfigurace sloupečku, které mohou nastat, přímo v kódu programu, využijeme vlastností bitových operátorů AND a OR. Sloupeček zakódujeme pomocí tří bitů. Číslice 1 bude odpovídat políčku, které

pokrýt nesmíme, neboť je již pokryté dlaždicí z vedlejšího sloupečku nebo se jedná o políčko zakázané. Číslice 0 pak odpovídá políčku, které je třeba pokrýt. Těchto kódů může být maximálně $2^3 = 8$. Pro každý kód si ručně předpočítáme všechny způsoby, kterými lze zcela zaplnit sloupeček a pro každý způsob si zapamatujeme kód, na kterých políčkách budou dlaždice přečnívat do vedlejšího sloupečku. Např. kód 000 lze pokrýt třemi způsoby. První možnosti odpovídá kód 111 (tři dlaždice leží pod sebou naležato a všechny přečnívají), 001 (první dlaždice je na stojato, poslední je naležato a přečnívá) a 100 (přesně opačná situace). Na následujícím obrázku jsou vidět možnosti i pro ostatní kódy. První řádek obsahuje všechny možné konfigurace sloupečku, řádky pod ním pak možnosti, kterými jej můžeme pokrýt.




Při výpočtu si spočítáme kód sloupečku $K_1$, který chceme zaplnit, a kód vedlejšího sloupečku $K_2$. Nyní projdeme všechny možnosti $M_i$ odpovídající kódu $K_1$ a pokud $M_i$ AND $K_2 = 0$, pak nenastala kolize přečnívající dlaždice se zakázaným políčkem ve vedlejším sloupečku, a výraz $M_i$ OR $K_2$ je roven stavu vedlejšího sloupečku v následujícím kroku.

Dílčí výsledky si postupně pamatujeme v pomocné tabulce. Během výpočtu si ale stačí pamatovat pouze aktuální sloupec tabulky, ze kterého spočítáme sloupec pro příští krok výpočtu. Proto je možné uchovávat si pouze dva sloupce tabulky, které postupně střídáme.

Protože jsou vstupní souřadnice zakázaných políček setříděné, není nutné je načíst rovnou všechny, ale je možné je načítat během výpočtu tak, jak je zrovna potřebujeme. Výsledná paměťová složitost je tedy konstantní a časová je lineární vzhledem k $N$.

```
program P_I_2;

const
   { protože možností je různě mnoho a pole může obsahovat
     pouze stejně dlouhé řádky, je za poslední možností
     číslo 8, které slouží jako zarážka při jejich procházení }
   moznosti : array [0..7, 1..4] of integer = (
      ( 4, 1, 7, 8), { 000 -> 100, 001, 111 }
      ( 6, 0, 8, 8), { 001 -> 110, 000 }
      ( 5, 8, 8, 8), { 010 -> 101 }
      ( 4, 8, 8, 8), { 011 -> 100 }
      ( 3, 0, 8, 8), { 100 -> 011, 000 }
      ( 2, 8, 8, 8), { 101 -> 010 }
      ( 1, 8, 8, 8), { 110 -> 001 }
      ( 0, 8, 8, 8)  { 111 -> 000 }
   );

   MAXN = 10000000;

type TSloupecek = array [ 1..3 ] of boolean;

{ zakóduje sloupeček do binární soustavy }
function Zakoduj(var sloupecek : TSloupecek) : integer;
var
   kod : integer;
begin
   kod := 0;
   if sloupecek[1] then
      kod := kod + 1;
   if sloupecek[2] then
      kod := kod + 2;
   if sloupecek[3] then
      kod := kod + 4;
   Zakoduj := kod;
end;

var
   sloupecek : TSloupecek;    { hodnota true je na zakázaném políčku }
   N, K, L : longint;
   X, Y : longint;
   i, j : longint;

   pocty : array [ 1..2 ] of array [ 0..7 ] of longint;
   aktualni, pristi : integer;
   K1, K2, Mi : integer;

begin
   { načtení vstupu }
   readln(N, K, L);

   { zarážka, abychom zajistili, že žádná dlaždice nebude přečnívat ven z chodby }
   for j:=0 to 7 do begin
      pocty[1][j] := 0;
      pocty[2][j] := 0;
   end;

   aktualni:=1;
   pristi:=2;

   pocty[aktualni][7]:=1;
```

```
   if K > 0 then
      readln(X, Y);

   for j:=1 to N+1 do begin
      { za konec chodby umístíme zarážku, abychom zajistili, že žádná dlaždice
        nebude přečnívat ven z chodby }
      sloupecek[1]:=(j=N+1);
      sloupecek[2]:=(j=N+1);
      sloupecek[3]:=(j=N+1);

      while (K > 0) and (Y = j) do begin
         sloupecek[X] := true;
         dec(K);
         if K > 0 then
            readln(X, Y);
      end;

      K2:=Zakoduj(sloupecek);
      for K1:=0 to 7 do begin { pro všechny možné konfigurace sloupečku }
         i:=1;
         while moznosti[K1][i] < 8 do begin { pro všechny možnosti, jak jej pokrýt }
            Mi := moznosti[K1][i];
            if K2 and Mi = 0 then { pokud nenastane kolize se zakázaným políčkem }
               { aktualizujeme počet možností pro další sloupeček }
               pocty[pristi][K2 or Mi] :=
                  (pocty[pristi][K2 or Mi] + pocty[aktualni][K1]) mod L;
            inc(i);
         end;
      end;

      { příští sloupeček tabulky se stane aktuálním }
      aktualni:=pristi;
      pristi:=3-aktualni;

      { příští sloupeček vynulujeme }
      for i:=0 to 7 do
         pocty[pristi][i] := 0;
   end;

   { vypíšeme počet možností zcela pokryté chodby }
   writeln(pocty[aktualni][7]);
end.
```

## P-I-3 Řeka

Jedním možným řešením je postavit si nad pilami binární vyhledávací strom (detailní popis najdete např. na *http://ksp.mff.cuni.cz/tasks/20/cook5.html*). Každý vrchol stromu odpovídá jedné pile, jejíž pozice bude větší než pozice pil v podstromu levého syna a menší než pozice pil v podstromu pravého syna. Aby se nám snáze odpovídalo na dotazy, kromě pozice této pily si v každém vrcholu navíc budeme pamatovat maximum z pozic pil, nacházejících se v jeho podstromu. Nechť je z pozice $x$ vyslán vor. Vyhledávání začneme v kořeni stromu, který obsahuje pilu na pozici $y$. Je-li $y > x$ a maximum z levého syna je menší než $x$, pak vor dorazí do pily v kořeni a můžeme na dotaz rovnou odpovědět. Je-li $y < x$, pak se pila, do níž vor dorazí, nachází dále po proudu od pily v kořeni, tj. v podstromu pravého syna. Jinak dorazí do některé z pil v podstromu levého syna. V těchto dvou případech se přesuneme do

příslušného syna a postup opakujeme. V každém kroku se posuneme u úroveň níž ve stromu, takže časová složitost bude omezená hloubkou stromu.

Jak implementovat další operace? Zahajujeme-li údržbu pily, mohli bychom ji ze stromu odebrat a při ukončení údržby ji naopak vložit. Nevýhodou tohoto řešení je nutnost udržovat hloubku takto se měnícího stromu. To vyžaduje jeho vyvažování, což bývá poměrně pracné. Jiná možnost je nechat strom pevný a u pil si pamatovat, zda jsou aktivní či nikoliv. Ve vrcholech si pak budeme pamatovat nikoliv maximum z pozic všech pil v podstromu, ale pouze maximum z pozic aktivních pil. Jestliže v podstromu není žádná aktivní pila, budeme místo maxima mít uloženou nějakou speciální hodnotu, například $-\infty$. Jestliže pila začne či přestane být aktivní, musíme upravit informace ve vrcholech na cestě od ní ke kořeni. Pro každý takový vrchol, obsahuje-li pravý syn aktivní pilu, pak převezmeme maximum pravého syna. Neobsahuje-li pravý syn aktivní pilu, ale pila ve vrcholu je aktivní, pak její pozice je nové maximum. Jinak převezmeme maximum z levého syna. Musíme také upravit vyhledávání v případě, že pila v uvažovaném vrcholu není aktivní. V tomto případě budeme dále hledat v levém synovi, jestliže jeho maximum je větší nebo rovno $x$, a jinak v pravém synovi.

Jelikož pily máme zadány setříděné dle vzdálenosti od pramene, můžeme z nich postavit perfektně vyvážený strom (takový, že počet pil v levém a v pravém podstromu se vždy liší nanejvýš o jedna) v čase $\mathcal{O}(P)$. Hloubka takového stromu je $\mathcal{O}(\log P)$ a taková bude i časová složitost každé operace. Celková časová složitost bude $\mathcal{O}(P + M \log P)$ a paměťová $\mathcal{O}(P)$.

Výše popsané řešení však nijak nevyužívá omezené délky řeky. Místo binárního stromu, který rozděluje pily tak, aby jejich počet v levém i pravém synovi byl přibližně stejný, bychom ale mohli na poloviny dělit řeku. Např. v levém synovi kořene by byly reprezentovány všechny pily ve vzdálenosti mezi 1 a $\lfloor N/2 \rfloor$ od pramene a v pravém synovi pily ve vzdálenosti $\lfloor N/2 \rfloor + 1$ až $N$. V každém synovi by se pak obdobně příslušný interval dělil na poloviny. Na rozdíl od prvního řešení neukládáme pily do vnitřních vrcholů stromu, ale jen do jeho listů. Ve vnitřních vrcholech si ukládáme pouze pomocné informace: zda se v příslušném podstromu nachází alespoň jedna pila, a jestliže ano, pak minimum a maximum ze vzdáleností takových pil od pramene. Řekněme, že vyhledáváme, kam dorazí vor z pozice $x$, a aktuální vrchol stromu odpovídá intervalu $\langle a, b \rangle$ (tedy levý syn odpovídá intervalu $\langle a, m \rangle$ a pravý intervalu $\langle m+1, b \rangle$, kde $m = \lfloor (a+b)/2 \rfloor$). Je-li $x > m$, pak pokračujeme v hledání v pravém synovi. Jestliže $x \leq m$, ale $x$ je větší než maximum z levého syna, pak vor dorazí do pily rovné minimu z pravého syna. Jinak pokračujeme v hledání v levém synovi. Takto dostaneme řešení s časovou složitostí $\mathcal{O}(N + M \log N)$ a paměťovou složitostí $\mathcal{O}(N)$. Tyto složitosti můžeme vylepšit na $\mathcal{O}(\min(N, P \log N) + M \log N)$, resp. $\mathcal{O}(\min(N, P \log N))$, jestliže nebudeme stavět stromy pro intervaly, které neobsahují žádnou pilu, ale i to je zjevně horší, než naše první řešení.

Je snad toto druhé řešení slepou uličkou? Všimněme si, že není důvod intervaly dělit pouze na dva díly. Zkusme každý interval dělit na $d$ dílů zhruba stejné délky – např. interval $\langle a, m \rangle$ bude rozdělen na intervaly $\langle a, m_1 \rangle$, $\langle m_1 + 1, m_2 \rangle$, ...,

$\langle m_{d-1}+1, b\rangle$, kde $m_i = a + \lfloor (b-a)i/d \rfloor$. Výsledný strom už nebude binární, ale každý jeho vnitřní vrchol bude mít $d$ synů. Při dotazu na vor z $x$ nejprve najdeme interval, do kterého $x$ patří (na to stačí jedno celočíselné dělení). Nechť tento interval odpovídá $i$-tému synovi $s_i$. Jestliže $x$ je menší než maximum z pozic pil v intervalu vrcholu $s_i$, pak budeme pilu, do níž $x$ dorazí, hledat dále v $s_i$. Jinak postupně projdeme syny $s_{i+1}$, $s_{i+2}$, …, a z prvního z nich, který obsahuje alespoň jednu pilu, vezmeme tu s minimální vzdáleností od pramene. Tedy, v každém vrcholu strávíme buď konstantní čas a zanoříme se o úroveň níž, nebo čas $\mathcal{O}(d)$, ale tím vyhledávání skončí. Hloubka stromu je $D = \mathcal{O}(\log_d(N)) = \mathcal{O}(\log N/\log d)$, a časová složitost vyhledávání tedy bude $\mathcal{O}(\log N/\log d + d)$. Zvolíme-li $d = \log N/\log\log N$, dostáváme časovou složitost na dotaz $\mathcal{O}(\log N/\log\log N)$.

Trochu problém ale je s touto časovou složitostí pily také přidávat či odebírat. Přidání pily může v nejhorším případě způsobit vytvoření cesty délky $D$ z kořene až do listu. Jestliže vrchol reprezentujeme jako pole s $d$ položkami, pak inicializace vrcholů na této cestě zabere čas $\mathcal{O}(dD) = \mathcal{O}((\log N/\log\log N)^2)$. To se dá obejít několika způsoby. Jedním by bylo udržovat si „sklad“ prázdných vrcholů: vždy, když vrchol smažeme, převedeme ho do skladu. Když pak vrchol tvoříme, vyzvedneme ho ze skladu, a víme, že je správně nainicializován. Jiná možnost je místo pole použít pro reprezentaci vrcholu hešovací tabulku (*http://ksp.mff.cuni.cz/tasks/21/cook4.html*) proměnlivé velikosti: pak velikost vrcholu bude úměrná počtu synů, a čas na inicializaci tedy bude pouze $\mathcal{O}(D)$. Mnohem jednodušší varianta je reprezentovat speciálně pouze vrcholy s právě jedním synem. Pak při přidání pily se nanejvýš jeden takový speciální vrchol změní na vrchol se dvěma syny, a budeme ho proto muset nahradit polem délky $d$; dále pak bude moci vzniknout $D$ vrcholů s jedním synem, které však reprezentujeme speciálními vrcholy konstantní velikosti; to dává časovou složitost $\mathcal{O}(d + D)$. Další možnost je vrcholy s jedním synem vůbec nevytvářet. V každém vrcholu si navíc k již popsaným informacím můžeme pamatovat, na jaké je úrovni a jakému intervalu odpovídá, a místo odkazů na syny si můžeme ukládat odkazy na nejbližšího potomka s alespoň dvěma syny či list v daném podintervalu. Tím pak při přidání nové pily vzniknou nanejvýš dva nové vrcholy (list obsahující přidávanou pilu a jeho otec), a časová složitost inicializace bude $\mathcal{O}(d)$. Pro optimální složitost lze tuto variantu zkombinovat s reprezentací vrcholů hešovacími tabulkami, čímž se složitost sníží na $\mathcal{O}(1)$. Tato složitost se ale týká pouze inicializace samotné, neboť nalezení místa, kam je třeba vrchol přidat, může trvat až $\mathcal{O}(D)$. Nicméně paměťová složitost bude úměrná časové složitosti inicializace, a proto bude pro celou datové strukturu nanejvýš $\mathcal{O}(P)$.

Dalším problémem je udržování minim a maxim z poloh pil v intervalu každého vrcholu. Při přidání tyto údaje jednoduše nahradíme minimem, resp. maximem, z původní hodnoty a pozice přidávané pily. Stačí změnit hodnoty na cestě z kořene do přidaného listu, a na to stačí čas $\mathcal{O}(D)$. Při odebírání však takto jednoduše postupovat nelze. Místo toho bychom potřebovali pro každý vrchol na této cestě nalézt prvního, resp. posledního syna, který obsahuje alespoň jednu pilu, a převzít jeho minimum, resp. maximum. Pak by časová složitost byla $\mathcal{O}(dD)$. Nicméně i toto lze opravit. V každém vrcholu si budeme udržovat navíc spojový seznam jeho synů, které obsahují alespoň jednu pilu. Pak minimum z pozic pil v daném vrcholu je rovno

minimu z pozic v prvním prvku tohoto seznamu, a podobně pro maximum, a proto je dokážeme najít v konstantním čase. Při odebrání pily mohou být smazány některé vrcholy stromu, které jednoduše odebereme ze seznamů jejich otců. Při přidání pily se zvýší stupeň jednoho vrcholu $v$, vznikne jeden list a případně několik vrcholů s pouze jedním synem. V každém z nově vzniklých vrcholech zjevně strávíme pouze konstantní čas. Ve vrcholu $v$ musíme najít místo v seznamu, kam přidat jeho nového syna, což zabere čas $\mathcal{O}(d)$. Celková složitost udržování těchto seznamů proto bude nejvýše $\mathcal{O}(d + D)$ na operaci.

S těmito vylepšeními je časová složitost přidání i odebrání pily $\mathcal{O}(d + D) = \mathcal{O}(\log N / \log\log N)$. Snadno také tuto datovou strukturu postavíme čase lineárním v počtu vložených prvků, jsou-li setříděné. Dostáváme řešení s časovou složitostí $\mathcal{O}(P + M \log N / \log\log N)$. Jestliže $N = \mathcal{O}(P)$, pak je toto řešení o trochu lepší než první, zato ale podstatně složitější.

Zkusme ho ještě vylepšit. Tím, že jsme zvýšili $d$, jsme snížili hloubku stromu. Za to ale platíme tím, že hledání nejbližšího následujícího syna obsahujícího pilu zabere čas $\mathcal{O}(d)$. Nedalo by se toto hledání nějak urychlit? Například bychom si nad neprázdnými syny mohli postavit vyhledávací strom. Tím by se složitost snížila na $\mathcal{O}(\log d)$, a snadno nahlédneme, že tím zlepšíme složitost vyhledání v celé datové struktuře na $\mathcal{O}(\sqrt{\log N})$ (pro $d = 2^{\sqrt{\log N}}$). Nešlo by to ještě lépe? Úloha, kterou řešíme při hledání nejbližšího následujícího syna, je úplně stejná, jako úloha, se kterou jsme začali, pouze na kratší „řece“ délky $d$. Můžeme proto použít rekurzivně stejnou datovou strukturu! S tímto vylepšením dosáhneme časové složitosti $\mathcal{O}(\log\log N)$ na dotaz (a dá se dokázat, že lépe to nejde).

Popišme nyní výslednou datovou strukturu podrobněji. Pro jednoduchost vynecháme některá dříve zmiňovaná vylepšení: nebudeme vynechávat prázdné podstromy, používat hešovací tabulky, či odstraňovat vrcholy s jedním synem. To trochu zhorší paměťovou složitost, ale značně zjednoduší výsledný algoritmus. Budeme ale potřebovat trik, že vrcholy obsahující právě jeden prvek budou reprezentovány speciálně.

Datová struktura $F(a, n)$ bude reprezentovat množinu přirozených čísel $S \subseteq \{a, a + 1, \ldots, a + n - 1\}$. Bude podporovat přidávání a odebírání prvků z $S$, a pro zadané $x$, nalezení nejmenšího prvku v $S$ většího než $x$, v čase $\mathcal{O}(\log\log n)$. Dále budeme umět rozhodnout, zda $S = \emptyset$, a jestliže ne, pak nalézt minimum a maximum z $S$ v čase $\mathcal{O}(1)$.

Pro $n \leq 2$ bude struktura $F(a, n)$ obsahovat seznam prvků množiny $S$. Uvažujme $n \geq 3$, a buď $s = \lfloor \sqrt{n + 1} \rfloor$ a $t = \lfloor (n - 1)/s \rfloor + 1$. Povšimněte si, že $s, t < n$. Struktura $F(a, n)$ se skládá z následujících položek:

- Pole $A$ indexované čísly 0, 1, ..., $t - 1$. Jestliže $|S| > 1$, pak položka tohoto pole na pozici $i$ bude obsahovat datovou strukturu $F(a + is, s)$ reprezentující množinu $S_i = S \cap \{a + is, a + is + 1, \ldots, a + is + s - 1\}$. Jestliže $|S| \leq 1$, pak všechny struktury $A[i]$ budou reprezentovat prázdnou množinu.
- Minimum $m_1$ a maximum $m_2$ z $S$ ($\infty$ a $-\infty$, jestliže $S$ je prázdná).

- Strukturu $B$ typu $F(0,t)$, reprezentující množinu $\{i : 0 \leq i < t, S_i \neq \emptyset\}$ jestliže $|S| > 1$ a jinak prázdnou množinu.

Nechť chceme nalézt následníka $x$ v $S$. Jestliže $x \geq m_2$, pak tento následník neexistuje. Jestliže $|S| = 1$, tj. $m_1 = m_2$, pak je $m_1$ následník $x$. Jinak položme $i = \lfloor (x-a)/s \rfloor$. Nejprve se podívejme, zda maximum z $S_i$ je menší nebo rovno $x$, dotazem na strukturu $A[i]$ v čase $\mathcal{O}(1)$. Je-li tomu tak, nalezneme pomocí struktury $B$ nejbližší vyšší index $j$ takový, že $S_j$ je neprázdná, a vrátíme minimum z $A[j]$. Jinak nalezneme následníka $x$ v $A[i]$. V obou případech provedeme pouze jeden rekurzivní dotaz na strukturu velikosti $\mathcal{O}(t) = \mathcal{O}(\sqrt{n})$. Po $\mathcal{O}(\log\log n)$ rekurzivních zanořeních dospějeme ke struktuře velikosti nanejvýš 2, a zodpovíme dotaz v konstantním čase. Časová složitost dotazu tedy skutečně je $\mathcal{O}(\log\log n)$.

Nyní přidávejme prvek $x$ do $S$. Jestliže $S = \emptyset$, pak pouze pouze položme $m_1 = m_2 = x$. Jestliže $|S| = 1$, pak nejprve vložme $m_1$ do $A[j]$ a $j$ do $B$, kde $j = \lfloor (m_1 - a)/s \rfloor$, a pokračujme jako v případě $|S| > 1$. Obě tyto operace proběhnou v konstantním čase, jelikož tyto datové struktury jsou prázdné. Jestliže $|S| > 1$, pak nejprve upravme $m_1$ a $m_2$. Poté položme $i = \lfloor (x - a)/s \rfloor$. Vložme $x$ do $A[i]$. Bylo-li $A[i]$ je prázdné, pak navíc vložme $i$ do $B$. Povšimněte si, že bylo-li $A[i]$ prázdné, pak vložení do $A[i]$ zabralo pouze konstantní čas, proto provedeme pouze jedno rekurzivní vnoření do struktury $B$, a tedy stejně jako v předchozím případě je časová složitost vnoření $\mathcal{O}(\log\log n)$.

Konečně, odebírejme $x$ z $S \neq \emptyset$. Je-li $|S| = 1$, pak pouze nastavíme $m_1 = \infty$ a $m_2 = -\infty$. Jinak smažme $x$ z $A[i]$. Je-li poté $A[i]$ prázdná (což nastane pouze tehdy, pokud předtím měla pouze jeden prvek, a jeho smazání proběhlo v konstantním čase), smažme $i$ z $B$. Je-li nyní $m_1'$ minimum z $B$, nastavme $m_1$ na minimum z $A[m_1']$. Obdobně upravíme $m_2$. Je-li $m_1 = m_2$, smažme ještě $m_1$ z $A[j]$ a $j$ z $B$, kde $j = \lfloor (m_1 - a)/s \rfloor$ (obě tyto struktury obsahují jen jeden prvek, takže jeho smazání proběhne v konstantním čase). Časová složitost je opět $\mathcal{O}(\log\log n)$.

Jak jsme na tom s paměťovou složitostí? Předpokládejme, že nám prázdnou datovou strukturu již někdo naalokooval, a postupně do ní přidávejme všechna čísla od $a$ do $a + n - 1$. Tím navštívíme všechna místa ve struktuře a dle výše popsané analýzy nám to zabere čas $\mathcal{O}(n \log\log n)$, proto taková bude v nejhorším případě i paměťová složitost.

Aplikujeme-li tuto datovou strukturu na náš problém, dostáváme časovou složitost $\mathcal{O}((N+M)\log\log N)$ a paměťovou složitost $\mathcal{O}(N\log\log N)$. Jak jsme naznačili, jestliže místo polí použijeme hešovací tabulky a budeme přeskakovat cesty z vrcholů, majících pouze jednoho syna, vylepší se časová složitost na $\mathcal{O}(P + M\log\log N)$ a paměťová na $\mathcal{O}(P)$. Zmiňme ještě, že popsané datové struktuře se říká van Emde Boasova fronta (případně strom).

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))
#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))
```

```
struct emdeboas
{
  int a, n, s;           /* Parametry struktury; s je velikost podstruktur. */
  int min, max;          /* Minimum a maximum z fronty.  Pro n <= 2 určuje prvky. */
  struct emdeboas **A;   /* Podstromy dle vzdálenosti. */
  struct emdeboas *B;    /* Fronta neprázdných podstromů. */
};

static void pridej (struct emdeboas *kam, int co);
static void odeber (struct emdeboas *odkud, int co);

/* Celočíselná druhá odmocnina z X. */
static int
int_sqrt (int x)
{
  int f = 1, t = x + 1, m;

  while (t - f > 1)
    {
      m = (f + t) / 2;
      if (m * m > x)
        t = m;
      else
        f = m;
    }
  return f;
}

/* Vytvoří prázdnou frontu pro rozsah A .. A + N - 1. */
static struct emdeboas *
nova_fronta (int a, int n)
{
  struct emdeboas *f = malloc (sizeof (struct emdeboas));
  int t, i;

  f->a = a;
  f->n = n;
  f->min = INT_MAX;
  f->max = INT_MIN;

  if (n <= 2)
    {
      f->s = 0;
      f->B = NULL;
      f->A = NULL;
    }
  else
    {
      f->s = int_sqrt (n + 1);
      t = (n - 1) / f->s + 1;
      f->B = nova_fronta (0, t);
      f->A = malloc (t * sizeof (struct emdeboas *));
      for (i = 0; i < t - 1; i++)
        f->A[i] = nova_fronta (a + i * f->s, f->s);
      f->A[t - 1] = nova_fronta (a + (t - 1) * f->s, n - (t - 1) * f->s);
    }
  return f;
}
```

```
/* Přidá prvek CO do podfront fronty KAM. */
static void
pridej_do_podfront (struct emdeboas *kam, int co)
{
  int i = (co - kam->a) / kam->s;

  if (kam->A[i]->min == INT_MAX)
    pridej (kam->B, i);
  pridej (kam->A[i], co);
}

/* Přidá prvek CO do fronty KAM. */
static void
pridej (struct emdeboas *kam, int co)
{
  int jednoprvkova = (kam->min == kam->max);
  int prvek = kam->min;

  kam->min = MIN (kam->min, co);
  kam->max = MAX (kam->max, co);

  if (kam->n <= 2 || kam->min == kam->max)
    return;

  if (jednoprvkova)
    pridej_do_podfront (kam, prvek);
  pridej_do_podfront (kam, co);
}

/* Odebere prvek CO z podfront fronty ODKUD. */
static void
odeber_z_podfront (struct emdeboas *odkud, int co)
{
  int i = (co - odkud->a) / odkud->s;

  odeber (odkud->A[i], co);
  if (odkud->A[i]->min == INT_MAX)
    odeber (odkud->B, i);
}

/* Odebere prvek CO z fronty ODKUD. */
static void
odeber (struct emdeboas *odkud, int co)
{
  if (co < odkud->min || co > odkud->max)
    return;

  if (odkud->min == odkud->max)
    {
      odkud->min = INT_MAX;
      odkud->max = INT_MIN;
      return;
    }

  if (odkud->n <= 2)
    {
      if (odkud->min == co)
        odkud->min = odkud->max;
      else if (odkud->max == co)
        odkud->max = odkud->min;
```

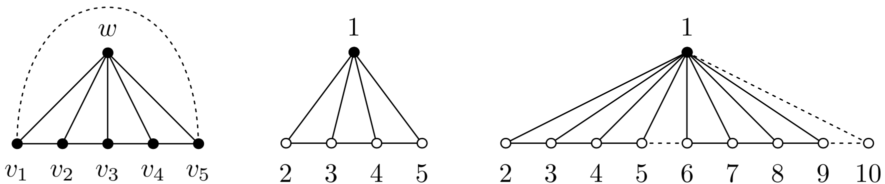
```
      return;
    }

  odeber_z_podfront (odkud, co);
  odkud->min = odkud->A[odkud->B->min]->min;
  odkud->max = odkud->A[odkud->B->max]->max;

  if (odkud->min == odkud->max)
    odeber_z_podfront (odkud, odkud->min);
}

/* Nalezne nejmenší prvek větší než CEHO ve frontě ODKUD.
 * Jestliže neexistuje, vrátí INT_MAX. */
static int
najdi_naslednika (struct emdeboas *kde, int ceho)
{
  int i;

  if (ceho >= kde->max)
    return INT_MAX;

  if (ceho < kde->min)
    return kde->min;

  if (kde->n <= 2)
    return kde->max;

  i = (ceho - kde->a) / kde->s;
  if (kde->A[i]->max > ceho)
    return najdi_naslednika (kde->A[i], ceho);
  else
    return kde->A[najdi_naslednika (kde->B, i)]->min;
}

int main (void)
{
  int n, p, m, i, x;
  char udalost[2];
  struct emdeboas *fronta;

  /* Načtení vstupů. */
  scanf ("%d%d%d", &n, &p, &m);
  fronta = nova_fronta (1, n);

  for (i = 0; i < p; i++)
    {
      scanf ("%d", &x);
      pridej (fronta, x);
    }

  /* Provádění operací. */
  for (i = 0; i < m; i++)
    {
      scanf ("%s %d", udalost, &x);
      switch (udalost[0])
        {
        case 'U':
          odeber (fronta, x);
          break;
        case 'K':
```

```
      pridej (fronta, x);
      break;
    case 'V':
      x = najdi_naslednika (fronta, x);
      printf ("%d\n", x == INT_MAX ? 0 : x);
      break;
    }
  }
  return 0;
}
```

## P-I-4 Grafový počítač

**a)** *Loukoťová kola.* Podobně jako při konstrukci cesty v příkladech v zadání se i tady vyplatí použít rekurzi: pokud dokážeme z grafu velikosti $n/2$ vytvořit pomocí konstantního počtu operací graf velikosti $n$, dosáhneme logaritmické časové složitosti. Bude ovšem snazší nekonstruovat přímo kola, nýbrž jiné, podobné grafy.

*Vějíř* velikosti $n$ budeme říkat grafu, který se skládá z cesty $v_1, \ldots, v_n$ a vrcholu $w$ (středu) spojeného hranou se všemi $v_i$ (viz první obrázek). Takový vějíř má tedy $n+1$ vrcholů a $2n-1$ hran a když do něj přidáme hranu $v_1, v_n$ (nakreslena čárkovaně), vznikne kolo velikosti $n$.




Náš program bude vytvářet vějíře očíslované podle druhého obrázku: střed dostane číslo 1 a značku 1, vrcholy cesty čísla $2, \ldots, n+1$ a značky `undef`.

Sledujme na třetím obrázku, co se stane, když dva vějíře $V$ velikosti 4 spojíme pomocí operace `Join(V, V, value_defined, none)`. Středy vějířů se slijí do společného vrcholu 1, vrcholy cest zůstanou samostatné a dostanou čísla 2 až 9. Přidáme-li hranu $\{5, 6\}$, vznikne vějíř velikosti 8. Pokud bychom chtěli vytvořit o 1 větší vějíř, doplníme ještě vrchol 10 a hrany $\{9, 10\}$ a $\{1, 10\}$.

Stejným způsobem můžeme z vějíře velikosti $n$ vyrobit vějíře velikostí $2n$ a $2n+1$.

Toho využije náš algoritmus: pokud máme vytvořit vějíř velikosti $n$, vytvoříme rekurzivním zavoláním vějíř poloviční velikosti $p = \lfloor n/2 \rfloor$, pak z něj pomocí `Join` vyrobíme vějíř velikosti $2p$ a pokud bylo $n$ liché, přidáme vrchol a spojíme ho hranami se zbytkem grafu. Nakonec vyrobíme z vějíře kolo přidáním hrany.

Jelikož se při každém rekurzivním zavolání velikosti vějíře zmenší alespoň dvakrát, je hloubka rekurze logaritmická a stejně tak časová složitost celého algoritmu.

Program může vypadat třeba následovně:

```
function vejir(n: Integer): Graph;
var g: Graph;
    p: Integer;
begin
   if n=1 then begin          { Triviální vějíř velikosti 1 }
      g := EmptyG;
      AddV(g, 1);
      AddV(g, undef);
      AddE(g, 1, 2, undef);
   end else begin             { Jinak použijeme rekurzi }
      p := n div 2;
      g := vejir(p);          { Vějíř poloviční velikosti }
      g := Join(g, g, value_defined, none);
      AddE(g, p+1, p+2, undef);  { 2 kopie spojíme hranou }
      if n mod 2 = 1 then begin  { Ošetříme liché n }
         AddV(g, undef);
         AddE(g, n, n+1, undef);
         AddE(g, 1, n+1, undef);
      end;
   end;
   vejir := g;
end;

function kolo(n: Integer): Graph;
var g: Graph;
begin                         { kolo = vějíř + hrana }
   g := vejir(n);
   AddE(g, 2, n+1, undef);
   kolo := g;
end;
```

**b)** *Vzdálenost* by bylo velmi snadné spočítat, kdybychom předem věděli, že *nejkratší cesta* mezi vrcholy $v$ a $w$ (připomínáme, že tak se říká cestě s nejmenším možným součtem vah hran) bude složena z právě $p$ hran. Tehdy bychom již známým způsobem vytvořili cestu délky $p$, jejímu počátečnímu vrcholu nastavili značku 1, koncovému 2 a ostatním `undef`. Pak bychom v zadaném grafu nastavili vrcholu $v$ značku 1, vrcholu $w$ značku 2 a všude jinde `undef`. Nakonec bychom pomocí funkce `Find` nalezli v zadaném grafu nejlehčí kopii cesty (přičemž bychom vyžadovali, aby značky vrcholů souhlasily) a pomocí `SumE` spočítali součet vah hran v této kopii – to je přesně vzdálenost mezi $v$ a $w$. To vše bychom stihli spočítat v logaritmickém čase (jediná část algoritmu, která nepracuje v konstantním čase, je konstrukce cesty, a o té už víme z příkladu v zadání, že se dá stihnout logaritmicky).

Obvykle ale nevíme, z kolika hran je nejkratší cesta složena – může to být cokoliv mezi 0 a $n-1$ (kde $n$ značí počet všech vrcholů). Samozřejmě bychom mohli vyzkoušet všechny možné počty hran, což by vedlo na algoritmus o složitosti $\mathcal{O}(n \log n)$, ale půjdeme na to trochu chytřeji. Upravíme totiž graf přilepením vějíře velikosti $n$ (tvořeného hranami nulové váhy), přesně podle obrázku na následující straně.

Nový graf $G'$ bude mít tyto příjemné vlastnosti:

- Kdykoliv existuje v $G$ nějaká cesta z $v$ do $w$ tvořená $p$ hranami, můžeme ji použitím $n - p$ hran vějíře rozšířit na cestu v $G'$ z $v'$ do $w'$, která bude mít stejnou váhu a právě $n$ hran.
- Naopak kdykoliv existuje v $G'$ cesta z $v'$ do $w'$, můžeme ji zkrátit na cestu v $G$ z $v$ do $w$ o přesně stejné váze.

Nyní tedy stačí hledat v grafu $G'$ nejkratší z těch cest mezi $v'$ a $w'$, které jsou tvořeny právě $n$ hranami.

Jakou časovou složitost toto řešení má? Spotřebujeme čas $\mathcal{O}(\log n)$ na konstrukci vějíře, načež ho konstantním počtem grafových operací přilepíme. Pak nás stojí dalších $\mathcal{O}(\log n)$ operací sestrojení cesty o $n$ hranách a $\mathcal{O}(1)$ nalezení její nejlehčí kopie. Celkově tedy naše řešení dosahuje složitosti $\mathcal{O}(\log n)$.

Program následuje:

```
function vzdalenost(g: Graph; v, w: Integer): Value;
var h, k, l, m: Graph;
    n: Integer;
begin
   { Značky v zadaném grafu: v -> 3, w -> 2, ostatní undef }
   SetAllV(g, undef);
   SetV(g, v, 3);
   SetV(g, w, 2);

   { Sestrojíme vějíř: střed -> 1, jeden z okrajů -> 3, jinde undef }
   n := CountV(g);
   h := vejir(n);
   SetV(h, n+1, 3);

   { Slepíme G s vějířem a odstraníme značku 3 }
   k := Join(g, h, value_defined, any);
   ReplaceV(k, 3, undef);

   { Vytvoříme cestu s okraji označenými 1, 2 }
   l := cesta(n);
   SetAllV(l, undef);
   SetV(l, 1, 1);
   SetV(l, n+1, 2);
   WriteG(l);

   { Najdeme nejlehčí kopii cesty a spočítáme její váhu }
   m := Find(k, l, value_strict, any);
   vzdalenost := SumE(m);
end;
```