16-1-1 Telefonní seznam — 10 bodů

Vytrvalejším řešitelům našeho semináře již dobře známá společnost Shumm & Brumm rozšiřuje své podnikání v oblasti telekomunikací a jejím nejnovějším počinem má být všeobsahující telefonní seznam. Oddělení pro shromažďování údajů již získalo všechna jména a telefonní čísla a předalo je Oddělení pro třídění údajů, které začalo jména třídit. Třídění je nicméně práce značně vyčerpávající a pracovníci oddělení si to postupně srovnávali v hlavě a odcházeli za lepším výdělkem. A tak se jednoho dne stalo, že nezbyl nikdo, kdo by třídil. Vedení společnosti se v zoufalství obrátilo na vás, abyste pomohli dotřídit onen seznam.

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu počet jmen v seznamu N a dále předtříděný seznam jmen (háčky a čárky ve jménech či českou specialitu s tříděním „ch" nebudeme uvažovat). Seznam je předtříděný tak, že libovolné jméno se v něm nachází nejvýše ve vzdálenosti k od místa, kde bude v setříděném seznamu. Toto číslo k též dostane váš program na vstupu. Na výstup má váš program vypsat setříděný seznam jmen.

Příklad: Pro seznam sedmi jmen *Pycha, Kopyto, Pytel, Netopyr, Pysk, Spytihnev, Slepys* je setříděný seznam *Kopyto, Netopyr, Pycha, Pysk, Pytel, Slepys, Spytihnev*. Jako k by mohl váš program dostat 2, protože nejvzdálenější od svých správných pozic byla slova *Netopyr, Pycha* a *Pytel* a ta se posunula o dvě místa.

16-1-2 Lokální minimum — 10 bodů

Mějme pole A s N celými čísly. Řekneme, že na pozici i je *lokální minimum* pokud $A[i-1] \geq A[i] \leq A[i+1]$ (pro jednoduchost předpokládáme, že $A[0] = A[N+1] = \infty$). Vaším úkolem v této úloze nebude nic těžšího, než nalézt v daném poli pozici libovolného lokálního minima. Problém ale spočívá v tom, že byste to měli udělat skutečně rychle – asymptotická časová složitost vašeho algoritmu (nepočítáme načítání pole) by měla být menší než $O(N)$.

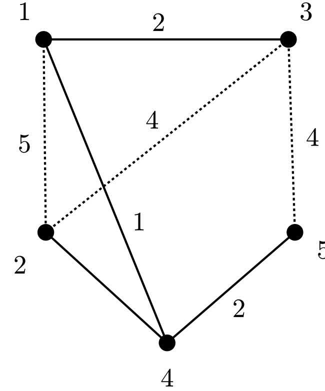
Příklad: V poli $1, 3, 2, 1, 4, 2, 5$ se lokální minima nachází na pozicích 1, 4 a 6 a váš program tedy může vrátit libovolnou z těchto pozic.

16-1-3 Důl — 11 bodů

Těžařská společnost Coppermine získala povolení k těžbě mědi v Kaputánii. Patřičný geologický průzkum již byl proveden, stroje nakoupeny, jediný problém, který ještě zbývá vyřešit, je doprava vytěžené měděné rudy ke zpracování. Společnost sice disponuje velkými nákladními auty, nicméně silnice v Kaputánii jsou poměrně nízké kvality a plně naložené nákladní auto by nemusely unést. Plánovače společnosti Coppermine by přirozeně zajímalo, jaké nejtěžší auto může projet z dolu do továrny na zpracování rudy, a proto se obrátili na vás.

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu počet měst v Kaputánii N, počet silnic M a dále popis oněch silnic. Každá silnice vede mezi dvěma městy a předpokládáme, že mimo města se silnice nekříží. Silnici proto popisují jednoznačně čísla dvou měst (města si očíslujeme od jedné do N), mezi kterými vede. Dále je u každé silnice udána její nosnost. Na výstup má váš program vypsat cestu z dolu do továrny (důl je ve městě s číslem 1 a továrna ve městě s číslem N), po které může projet co nejtěžší auto – mezi všemi cestami z dolu do továrny to je taková cesta, na které je minimum z nosností jednotlivých úseků co největší. Můžete předpokládat, že mezi dolem a továrnou vždy vede nějaká cesta. Pokud je cest se stejnou nosností více, můžete vrátit libovolnou z nich.

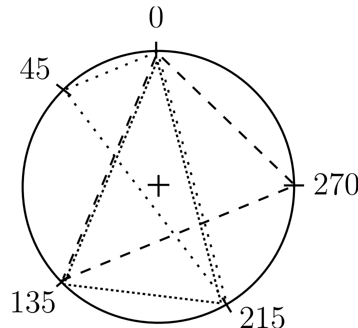
Příklad: Pro situaci jako na obrázku by měl váš program nalézt cestu $1, 2, 3, 5$, po které může projet auto s váhou 4.



16-1-4 Neruš mé trojúhelníky! — 10 bodů

V průběhu věků lidé vymysleli mnoho různých více či méně šílených šifrovacích metod. Jednu z nich vymysleli kryptografičtí odborníci z Artustánu. Zašifrovaná zpráva vypadala jako několik nevinně vyhlížejících kružnic s vyznačenými některými body na obvodu. Výzvědná služba ze sousedního Bosstánu dlouho nemohla šifru rozluštit, až se jednomu z jejích špiónů podařilo vypátrat, že pro šifru je podstatný počet trojúhelníků s vrcholy ve vyznačených bodech, které obsahují střed kružnice. Zjistit toto číslo ručně je ovšem pracné, a tak jste byli požádáni o pomoc.

Vaším úkolem je navrhnout algoritmus a napsat program, který dostane na vstupu počet vyznačených bodů na obvodu kružnice N a dále N reálných čísel udávajících úhly, pod kterými jednotlivé body leží ($0°$ je bod „na sever" od středu, úhel roste proti směru hodinových ručiček). Na výstup pak vypíše počet trojúhelníků s vrcholy v zadaných bodech, které obsahují střed kružnice.



Příklad: Pro kružnici s pěti vyznačenými body na pozicích $0°, 45°, 135°, 215°, 270°$ (viz obrázek) existuje pět trojúhelníků obsahujících střed.

16-1-5 Pravděpodobnostní algoritmy — 10 bodů

V letošním ročníku jsme se rozhodli, že v rámci obvyklého seriálu uvedeme několik úloh zaměřených na pravděpodobnostní algoritmy. Co to takový pravděpodobnostní algoritmus je? Inu je to vlastně obyčejný algoritmus, který ale navíc při svém běhu využívá náhodná čísla. Asi si řeknete:

„K čemu nám jsou náhodná čísla dobrá?“ Ač to je možná na první pohled překvapivé, náhodná čísla mohou pomoci výrazně urychlit běh našeho algoritmu.

Protože náhodná čísla se budou táhnout celým naším seriálem, měli bychom si nejdříve ujasnit, co si pod tímto pojmem představujeme. Obvykle budeme potřebovat generovat náhodná celá čísla z nějakého intervalu $0 \dots N-1$ tak, aby všechna čísla měla stejnou pravděpodobnost (tedy abychom každé číslo vygenerovali s pravděpodobností $1/N$). Získat takové náhodné číslo ale není tak jednoduché, jak by se mohlo zdát a my toho využijeme v naší první úloze.

Představte si, že máte k dispozici generátor náhodných bitů – tedy nějakou funkci *randbit*, která vám s pravděpodobností 1/2 vrátí 0 a s pravděpodobností 1/2 vrátí 1 – a chcete s její pomocí vytvořit funkci *random*(N), která vrací náhodné číslo z intervalu $0 \dots N-1$ (takové, jaké jsme popsali v předchozím odstavci). Navíc bychom chtěli, aby vaše funkce *random* potřebovala na vygenerování jednoho náhodného čísla co nejméně volání funkce *randbit*. Součástí vašeho řešení by mělo být jednak zdůvodnění, proč váš generátor vygeneruje každé číslo z daného intervalu se stejnou pravděpodobností a jednak odhad počtu volání funkce *randbit*.

Recepty z programátorské kuchařky

Naše povídání o algoritmech a datových strukturách začneme u jednoho z nejznámějších algoritmů, Dijkstrova algoritmu pro hledání nejkratších cest v grafech. A protože se nám k tomu bude hodit šikovná datová strukturka zvaná halda, tak si popíšeme nejdříve ji.

Halda je datová struktura pro uchovávání množiny čísel (či jakýchkoliv jiných objektů, na kterých máme definováno uspořádání, tj. umíme je porovnávat). Tato datová struktura obvykle podporuje následující operace: Přidání nového prvku do haldy, odebrání nejmenšího prvku a dotaz na nejmenší prvek. My si ukážeme takovou implementaci haldy, že pokud halda obsahuje N čísel (prvků), tak na přidání či odebrání jednoho prvku potřebujeme čas $O(\log N)$ a na zjištění hodnoty nejmenšího prvku v haldě nám stačí konstantní čas, tj. $O(1)$.

Jedna taková implementace haldy je následující: Pokud halda obsahuje N prvků, pak její prvky máme uloženy v poli na pozicích 0 až $N-1$. Prvek na pozici s indexem k má dva *následníky,* a to prvky na pozicích $2k+1$ a $2k+2$; samozřejmě, pokud je k velké, a tedy např. $2k+2 > N-1$, pak má takový prvek jednoho či dokonce žádného následníka. Prvek na pozici $\lceil (k-1)/2 \rceil$ pak nazýváme *předchůdcem* prvku na pozici k. Ti z vás, kteří znají binární stromy, jistě rozpoznali ve výše uvedeném možnost, jak v poli uchovávat vyvážené binární stromy (následníci jsou synové a předchůdci otcové v obvyklé stromové terminologii).

My však v poli neuchováváme prvky haldy úplně v libovolném pořadí. Chceme, aby platilo, že každý prvek je menší než kterýkoliv z jeho následníků. Takže naše halda může vypadat např. následovně:

0	1	2	3	4	5	6	7	8
5	6	20	25	7	21	22	26	27

Z toho, co jsme si právě popsali je jasné, že nejmenší prvek je uložen na pozici s indexem 0 a tedy můžeme snadno v konstantním čase zjistit jeho hodnotu. V následujícím odstavci prozradíme, jak lze prvky do haldy rychle přidávat a odebírat.

Popišme si nejprve, jak lze prvek do haldy přidat. Jestliže halda obsahuje N prvků, pak nový prvek, říkejme mu třeba X, nejprve umístíme na konec pole, tj. na pozici s indexem N. Nyní X porovnáme s jeho předchůdcem. Pokud je jeho předchůdce menší, pak je vše v pořádku a jsme hotovi. V opačném případě X prohodíme s jeho předchůdcem. Zřejmě je X nyní menší než kterýkoliv z jeho následníků, ale stále by mohl být menší než jeho nový předchůdce. Takže X porovnáme s jeho současným předchůdcem a pokud je X menší, tak tyto dva prvky opět prohodíme. A takto pokračujeme, dokud současný předchůdce X není menší než X nebo X nemá žádného předchůdce (tj. X je na pozici 0). Protože se v každém kroku index pozice, kde se prvek X právě nachází, zmenší zhruba na polovinu, tak celkově provedeme nejvýše $O(\log N)$ výměn, a tedy spotřebujeme čas $O(\log N)$. Odebírání prvků probíhá podobně: Prvek z poslední pozice (tj. z pozice $N-1$) přesuneme na pozici 0. Místo s předchůdci jej však porovnáme s jeho následníky a v případě, že je větší než některý z jeho následníků, tak je prohodíme (pokud je větší než oba jeho následníci, pak ho prohodíme s menším z nich). A protože se nám v každém kroku index „bublajícího“ prvku v poli zhruba zdvojnásobí, opět spotřebujeme čas $O(\log N)$.

Jako cvičení si rozmyslete, že když si pamatujeme pro každý prvek, kde se v haldě nachází, pak lze z haldy libovolný prvek odstranit (nebo změnit jeho hodnotu) v čase $O(\log N)$.

Ukázkovou implementaci haldy si můžete prohlédnout na konci naší kuchařky.

Dijkstrův algoritmus se používá pro hledání nejkratších cest v grafu. Graf si můžeme představovat jako nějaké body, kterým říkáme *vrcholy,* spojené navzájem čarami, kterým pro změnu říkáme *hrany.* V našem případě budou mít hrany přiřazeny *váhy,* tedy něco jako délku čáry. *Cestou* pak nazveme posloupnost vrcholů $v_1 v_2 \dots v_d$ takovou, že každé dva po sobě jdoucí vrcholy jsou spojeny hranou, a *délkou cesty* součet vah hran spojujících takové dvojice vrcholů. Graf si můžeme představit jako např. města spojená silnicemi, kde váha je délka silnice a délka cesty je pak vzdálenost, kterou ujedeme mezi městy. Někdy se také setkáme s *orientovanými* grafy, ve kterých mají hrany přiřazenu orientaci, tj. směr z jednoho z krajních vrcholů do druhého, a je po nich možné cestovat pouze v tomto daném směru.

Dijkstrův algoritmus nalezne v grafu nejkratší cestu mezi dvěma zadanými vrcholy za předpokladu, že váhy všech hran jsou nezáporné. Ve skutečnosti tento algoritmus dělá o malinko více: Najde totiž nejkratší cesty z jednoho zadaného vrcholu do všech ostatních.

Nechť v_0 je vrchol grafu, ze kterého chceme určit délky nejkratších cest. Budeme si udržovat pole délek zatím nalezených cest z vrcholu v_0 do všech ostatních vrcholů grafu. Navíc u některých vrcholů budeme mít poznamenáno, že cesta nalezená do nich je už ta nejkratší možná. Takovým vrcholům budeme říkat *trvale ohodnocené.* Na začátku inicializujeme v poli všechny hodnoty na ∞ kromě hodnoty odpovídající vrcholu v_0, kterou inicializujeme na 0 (délka nejkratší cesty z v_0 do v_0 je 0). V každém *kroku* algoritmu pak provedeme následující: Vybereme vrchol w, který není trvale ohodnocený, a mezi všemi takovými vrcholy je délka zatím nalezené cesty do něj nejkratší možná. Vrchol w prohlásíme za trvale ohodnocený. Dále otestujeme, zda pro nějaký vrchol v cesta z vrcholu v_0 do w a pak po hraně z w

do v není kratší, než zatím nalezená cesta z v_0 do v a je-li tomu tak, pak změníme délku zatím nalezené cesty do v. Toto provedeme pro všechny takové vrcholy v.

Celý algoritmus skončí, pokud jsou už všechny vrcholy trvale ohodnocené nebo všechny vrcholy, co nejsou trvale ohodnocené, mají délku cesty do nich rovnou ∞ (v takovém případě se graf skládá z více nesouvislých částí). Před tím, než dokážeme, že právě představený algoritmus opravdu nalezne délky nejkratších cest z vrcholu v_0, se zamysleme nad jeho časovou složitostí: K uchovávání délek dosud nalezených cest použijeme haldu. Zřejmě halda bude obsahovat na začátku N prvků a v každém kroku se počet jejích prvků sníží o jeden. Celý algoritmus má nejvýše N kroků, kde N je počet vrcholů vstupního grafu. V každému kroku musíme zkontrolovat tolik vrcholů v, kolik hran vede z vrcholu w. Každá taková kontrola může vyústit ve vyjmutí a přidání prvku v haldě, tj. můžeme na ni potřebovat čas $O(\log N)$. Počet takových změn pro všechny kroky dohromady je pak nejvýše $O(M)$, kde M je počet hran vstupního grafu. Celková časová složitost našeho algoritmu je $O((N+M)\log N)$.

Může se samozřejmě stát, že náš graf má hodně hran, až kvadraticky mnoho v počtu vrcholů N. V takovém případě je lepší haldu vůbec nepoužít a v každém kroku určit w v čase $O(N)$ prostým výběrem nejmenší hodnoty z trvale neohodnocených vrcholů (a tyto hodnoty si uchovávat v normálním poli). Časová složitost této implementace Dijkstrova algoritmu, jejíž kód lze najít na konci naší kuchařky, je $O(N^2)$.

Poznámka pro zvídavé: použitím jiného druhu haldy, tzv. Fibonacciho haldy, lze zlepšit časovou složitost Dijkstrova algoritmu až na $O(M + N\log N)$.

Vraťme se nyní k důkazu správnosti Dijkstrova algoritmu. Ukážeme, že po každém kroku algoritmu platí následující tvrzení: Nechť A je množina trvale ohodnocených vrcholů. Pak délka dosud nalezené cesty z v_0 do v (v je libovolný vrchol grafu) je délka nejkratší cesty $v_0v_1\dots v_kv$ takové, že všechny vrcholy $v_0, v_1, \dots, v_k$ jsou v množině A. Tvrzení dokážeme indukcí, dle počtu kroků algoritmu, které již proběhly. Tvrzení zřejmě platí před a po prvním kroku algoritmu. Nechť w je vrchol, který byl v předchozím kroku prohlášen za trvale ohodnocený. Uvažme nejprve nějaký vrchol v, který je trvale ohodnocený. Pokud $v = w$, tvrzení je triviální. V opačném případě ukážeme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w. Označme D délku cesty z v_0 do v přes vrcholy $A \setminus \{w\}$. Protože v každém kroku vybíráme vrchol s nejmenším ohodnocením a ohodnocení vybraných vrcholů v jednotlivých krocích tvoří neklesající posloupnost (váhy hran jsou nezáporné!), tak délka cesty z v_0 do w přes vrcholy z A je alespoň D. Ale potom délka libovolné cesty z v_0 do v přes w používající vrcholy z A je alespoň D. Z volby D pak víme, že existuje nejkratší cesta z v_0 do v přes vrcholy z A, která nepoužívá vrchol w.

Nyní uvažme takový vrchol v, který není trvale ohodnocený. Nechť $v_0v_1\dots v_kv$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \dots, v_k$ jsou v množině A. Pokud $v_k = w$, pak jsme ohodnocení v změnili na délku této cesty v právě proběhlém kroku. Pokud $v_k \neq w$, pak $v_0v_1, \dots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A, a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \dots, v_k$ není w (podle toho, co jsme si rozmysleli na konci minulého odstavce). Potom se ale délka cesty do v rovnala správné hodnotě už před právě proběhlým krokem.

Vzhledem k tomu, že po posledním kroku množina A obsahuje právě ty vrcholy, do kterých existuje cesta z vrcholu v_0, dokázali jsme, že náš algoritmus funguje správně.

Na závěr ještě poznamenejme, že Dijkstrův algoritmus funguje i pro orientované grafy a že jej lze snadno upravit tak, aby nám kromě délky nejkratší cesty i takovou cestu našel: U každého vrcholu si v okamžiku, kdy mu měníme ohodnocení, poznamenáme, ze kterého vrcholu do něj přicházíme. Nejkratší cestu do nějakého vrcholu pak zrekonstruujeme tak, že u posledního vrcholu této cesty zjistíme, který vrchol je předposlední, u předposledního, který je předpředposlední, atd.

Svá řešení první série nám zasílejte do 20. října 2003 na výše uvedenou adresu.

Ukázková implementace haldy

```
var halda:array[0..MAX] of integer;
    N: word; { počet prvků v haldě }

procedure chyba; { něco je špatně }

function nejmensi:integer;
begin
  if N=0 then chyba;
  nejmensi:=halda[0]
end;

procedure vloz(prvek: integer);
var i:word;
    x:integer;
begin
  if N=MAX then chyba;
  i:=N; N:=N+1;
  halda[i]:=prvek;
  while (i>0) and (halda[(i-1) div 2]>halda[i]) do
    begin
      x:=halda[(i-1) div 2]; halda[(i-1) div 2]:=halda[i]; halda[i]:=x;
      i:=(i-1) div 2
```

```
    end
end;

procedure zrus_nejmensi;
var i,j:word;
    x:integer;
begin
  if N=0 then chyba;
  halda[0]:=halda[N-1];
  N:=N-1; i:=0;
  while 2*i+1<=N-1 do
    begin
      j:=i;
      if (2*i+1<=N-1) and (halda[j]>halda[2*i+1]) then j:=2*i+1;
      if (2*i+2<=N-1) and (halda[j]>halda[2*i+2]) then j:=2*i+2;
      if i=j then break;
      x:=halda[i]; halda[i]:=halda[j]; halda[j]:=x;
      i:=j
    end
end;
```

Implementace Dijkstrova algoritmu

```
var N: word; { počet vrcholů }
    vahy: array[1..MAX,1..MAX] of integer;
             { váhy hran, -1 = hrana neexistuje }
    delky: array[1..MAX] of integer;
             { délky zatím nalezených cest, -1 = nekonečno }
    trvaly: array[1..MAX] of boolean;
             { trvale ohodnocen? }

procedure Dijkstra(odkud: word);
var i: word;
    w,v: word;
begin
  for i:=1 to N do
    begin
      trvaly[i]:=false;
      delky[i]:=-1;
    end;
  trvaly[odkud]:=true;
  delky[odkud]:=0;
  repeat
    w:=0;
    for i:=1 to N do
      if not(trvaly[i]) then
        if w=0 then
          w:=i
        else
          if delky[i]<delky[w] then
            w:=i;
    if w<>0 then
      begin
        trvaly[w]:=true;
        for i:=1 to N do
          if (vahy[w][i]<>-1) and not(trvaly[i]) and
               { podmínku "not(trvaly[i])" lze vypustit }
             (delky[w]+vahy[w][i]<delky[i]) then
            delky[i]:=delky[w]+vahy[w][i]
      end
  until w=0;
end;
```

Tips & Tricks: Z letáků KSP si můžete složit knížečku