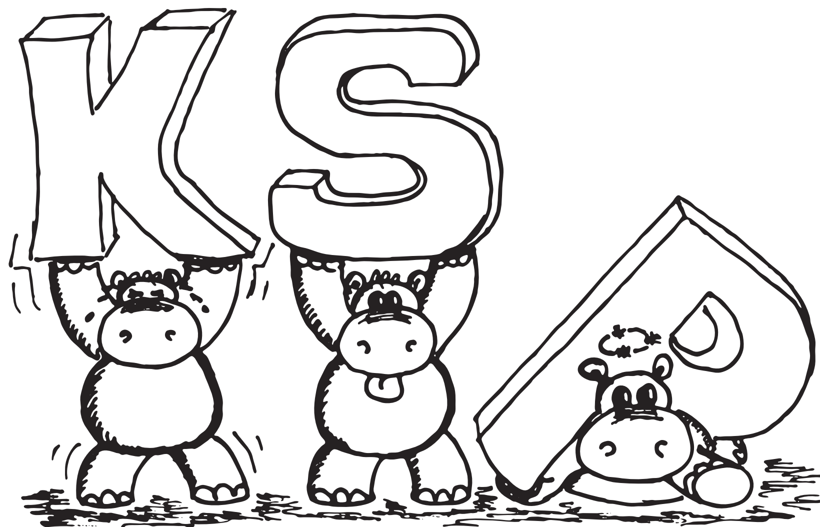
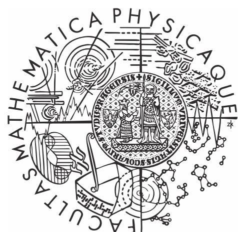
MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVI. ročník – 2003/2004





Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Copyright © 2004 Milan Straka
© Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MILAN STRAKA A KOLEKTIV

Korespondenční seminář z programování

XVI. ročník – 2003/2004

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Úvod

Korespondenční seminář z programování (dále jen *KSP*), jehož šestnáctý ročník se vám dostává do rukou, patří k nejznámějším aktivitám pořádaným MFF pro zájemce o informatiku a programování z řad studentů středních škol. Řešíce úlohy našeho semináře, středoškoláci získávají praxi ve zdolávání nejrůznějších algoritmických problémů, jakož i hlubší náhled na mnohé discipliny informatiky. Proto některé úlohy *KSP* svou obtížností vysoko přesahují rámec běžného středoškolského vzdělání, a tudíž i požadavky při přijímacím řízení na vysoké školy, MFF z toho nevyjímaje. To ovšem vůbec neznamená, že nemá smysl takové problémy řešit – při troše přemýšlení není příliš obtížné nějaké (i když někdy ne to nejlepší) řešení nalézt. Nakonec – posuďte sami.

KSP probíhá tak, že student od nás jednou za čas dostane poštou zadání několika (čtyř či pěti) úloh, v klidu domácího krbu je (ne nutně všechny) vyřeší, svá řešení v přiměřeně vzhledné podobě sepíše a do určeného termínu zašle na níže uvedenou adresu (ať už fyzickou či elektronickou). My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme při vhodné příležitosti zpět na adresu studenta. Tento cyklus se nazývá *série,* resp. kolo.

Za jeden školní rok obvykle proběhnou čtyři série, v letech hojnějších pak pět. Závěrečným bonbónkem je pak pravidelné *soustředění* nejlepších řešitelů semináře, konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čítající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě).

Náš korespondenční seminář není ojedinělou aktivitou svého druhu – existují korespondenční semináře z fyziky a matematiky při MFF, jakož i jiné programátorské semináře (kupříkladu bratislavský). Rozhodně si však nekonkurujeme, ba právě naopak – každý seminář nabízí něco trochu jiného, řešitelé si mohou vybrat z bohaté nabídky úloh a najdou se i takoví nadšenci, kteří úspěšně řeší několik seminářů najednou.

Velice rádi vám odpovíme na libovolné dotazy jak ohledně studia informatiky na naší fakultě, tak i stran jakýchkoliv informatických či programátorských problémů. Jakýkoliv problém, jakákoliv iniciativa či nabídka ke spolupráci je vítána na adrese:

Korespondenční seminář z programování
KSVI MFF
Malostranské náměstí 25
118 00 Praha 1

e-mail: *ksp@mff.cuni.cz*
www: *http://ksp.mff.cuni.cz/*

Zadání úloh

16-1-1 Telefonní seznam 10 bodů

Vytrvalejším řešitelům našeho semináře již 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.

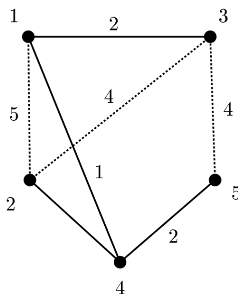
16-1-3 Důl 11 bodů

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.

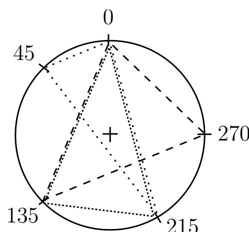
Příklad: Pro situaci jako na obrázku by měl váš program nalézt cestu $1, 2, 3, 5$, po které může projet auto s váhou 4.



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 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 objevovat v celém našem seriálu, měli bychom si nejdříve ujasnit, co si pod tímto pojmem představujeme. Obvykle budeme potřebovat generovat náhodná celá čísla z nějakého intervalu $0 \ldots N-1$ 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 $0 \ldots N-1$ (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*.

16-1-K Recepty z programátorské kuchařky — Dan Kráľ

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 porov-

návat). 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 čas konstantní, 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 $\lfloor (k-1)/2 \rfloor$ 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 $v_1v_2\ldots v_d$ 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 $v_0v_1\ldots v_kv$ takové, že všechny vrcholy $v_0, v_1, \ldots, v_k$ 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 ještě není trvale ohodnocený. Nechť $v_0v_1\ldots v_kv$ je nejkratší cesta z v_0 do v taková, že všechny vrcholy $v_0, v_1, \ldots, v_k$ 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 $v_0v_1, \ldots, v_k$ je nejkratší cesta z v_0 do v_k přes vrcholy z množiny A, a tedy můžeme předpokládat, že žádný z vrcholů $v_1, \ldots, v_k$ 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.

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,j: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
   j:=(i-1) div 2;
   x:=halda[j];halda[j]:=halda[i];halda[i]:=x;
   i:=j;
  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;
```

16-2-1 Král Eeek — 10 bodů

Byl-nebyl jednou jeden král, který se jmenoval Eeek a měl tuze rád knížky. Jeho palác se už dávno stal jednou velikánskou knihovnou a král Eeek trávil celé dny vysedáváním u krbu ve své soukromé čítárně. Až jednoho dne za ním přišel lord nejvyšší knihovník a svěřil se králi, že objevil knihu, které ani za mák nerozumí.

Celý text onoho masivního fasciklu byl tvořen jen a pouze číslicemi, uspořádanými zdánlivě bez jakéhokoliv řádu, a spolehlivě činil zmatek jak v pomazané hlavě královské, tak v poněkud méně vznešené, leč vzdělanější hlavě knihovníkově. Dlouho o tom při dobrém víně přemýšleli, až dospěli k následující teorii:

Kdysi dávno žila civilizace, jejíž filosofové rádi zapisovali své myšlenky jako dlouhatánská čísla v roztodivných číselných soustavách. Ovšem základy těchto soustav byly natolik velké, že jim brzy došly symboly pro číslice, a tak jednotlivé číslice začali zapisovat v desítkové soustavě. Navíc ještě za číslo připojovali základ soustavy, rovněž zapsaný desítkově. Čísla tedy vypadala například takto:

$$(1)(6)[10] = 1 \cdot 10^1 + 6 \cdot 10^0 = 16$$
$$(10)(10)[11] = 10 \cdot 11^1 + 10 \cdot 11^0 = 120$$
$$(14)(10)(5)[16] = 14 \cdot 16^2 + 10 \cdot 16^1 + 5 \cdot 16^0 = 3749$$
$$(1)(4)(1)(0)(5)(1)[6] = 1 \cdot 6^5 + \ldots + 1 \cdot 6^0 = 13207$$

Přitom zápisy číslic nikdy nezačínaly nulou, pokud nešlo o nulu samotnou, a byla to vždy celá nezáporná čísla menší než základ. Základ byl celé číslo větší než 1 a také nikdy nezačínal nulou. Formálně řečeno, hodnota čísla se stanovovala podle následujícího pravidla:

$$(a_n)(a_{n-1})\ldots(a_1)(a_0)[z] = \sum_{i=0}^{n} a_i \cdot z^i.$$

„Léty ovšem závorky v zápisu vybledly a zbyly jen číslice, takže původní číslo už stěží kdo rozpozná,“ povzdechl si král Eeek. Knihovník na to ale opáčil, že zatímco se Jeho veličenstvo ráčilo věnovati hodnotné literatuře, poddaní mezitím objevili počítače a dokonce si už založili i programátorský korespondenční seminář, takže jistě dokáží napsat program, který o daném řetězci čísel rozhodne, kolik existuje způsobů, jak doplnit závorky tak, aby vznikl korektní zápis nějakého čísla.

Co říkáte, dokážete to?

Příklad: Posloupnost `1410516` odpovídá zápisům:

(1)(4)(1)(0)(5)(1)[6]	(1)(4)(1)(0)[516]
(1)(4)(1)(0)(5)[16]	(14)(1)(0)[516]
(14)(1)(0)(5)[16]	(1)(41)(0)[516]
(1)(4)(10)(5)[16]	(141)(0)[516]
(14)(10)(5)[16]	(1)(4)(10)[516]
(1)(4)[10516]	(14)(10)[516]
(14)[10516]	(1)(410)[516]
(1)[410516]	

Naproti tomu posloupnost `100` žádnému zápisu neodpovídá.

16-2-2 Král Ovopole — 10 bodů

Nebyl-byl jednou jiný král, kterému říkali Ovopole, neboť měl zvláštní zálibu ve vajíčkách – mimo to, že si na nich rád pochutnával, je také vědecky zkoumal. Jednoho dne ho napadlo, že zjistí, ze kterého nejnižšího patra jeho vysokánského (takto N-patrového) paláce se vajíčko puštěné z okna na nádvoří rozbije.

To je samozřejmě snadné zjistit pokusy: V každém pokusu král vajíčko pustí z nějakého patra a když se vajíčko nerozbije, nechá si ho přinést a může s ním podniknout další pokus; pokud se rozbije, musí král sáhnout po dalším vajíčku. To je snadné, ale ouha, Ovopole právě s ú(div|děs|žas)em zjistil, že v celém paláci se nachází jen k vajíček, která doposud nepodlehla předchozím experimentům. Navíc král je poněkud netrpělivý, takže by s těmito k vajíčky chtěl získat správnou odpověď na co nejméně pokusů.

Napište proto našemu králi program, který jeho problém vyřeší (jistě se vám za to dostane královské odměny). Takový program dostane na vstupu počet pater N a počet vajíček k a postupně bude navrhovat jednotlivé pokusy a přijímat odpovědi, jak právě vypsaný pokus dopadl. Nakonec program odpoví číslem hledaného nejnižšího patra.

16-2-3 Král Potvorník 10 bodů

Král Potvorník (matematiky většinou zvaný „Ten, který zadává ε“ nebo prostě Nepřítel) si u svých zeměměřičů objednal přeměření své královské potvorologické zahrady za účelem stavby nového plotu. Zahrada má, jak je známo, tvar nepravidelného konvexního n-úhelníku (to, že je konvexní, znamená, že všechny vnitřní úhly jsou menší než 180°) a Potvorník potřebuje zjistit její obvod, aby věděl, kolik pletiva a ostnatého drátu musí objednat.

Zeměměřiči vyměřili souřadnice všech sloupků budoucího plotu ležících přesně ve vrcholech n-úhelníka a když se užuž chystali dát se do počítání, přiběhla jedna z obyvatelek zahrady a jako na potvoru do hromádky lístků s napsanými souřadnicemi strčila a beznadějně je pomíchala. Nedosti na tom, zamíchala mezi ně i jiné lístky, na nichž byly souřadnice různých objektů ležících uvnitř zahrady.

Na vás je, abyste zeměměřiče zachránili před královskou odměnou (která by je jistě neminula, kdyby nespočítali včas) tím, že napíšete program, který dostane na vstupu všechny nasbírané souřadnice (tedy souřadnice vrcholů a nějakých bodů uvnitř, to vše v libovolně zpotvořeném pořadí) a odpoví co možná nejrychleji, jaký je obvod zahrady (jak již asi tušíte, ani tento král není zrovna vzorem trpělivosti).

16-2-4 Křížový král 10 bodů

Při putování křížem kráž světem ve službách Křížového (nebo Krážového?) krále jste se dostali až do jeskyně obývané ohnivým drakem. Drak vás vřele přivítal a rovnou vás pozval k obědu. Brzy jste bohužel zjistili, že se také můžete stát jeho podstatnou součástí. Nicméně draci jsou čestní a navíc se tento už dlouho nudil, takže jste dostali šanci zachránit si život. Stačí porazit draka ve hře.

Hra je velmi jednoduchá: drak si zvolí nějaké přirozené číslo mezi 1 a N (kteréžto N je předem známo) a na vás je, abyste ho uhodli. Vy si v každém tahu zvolíte množinu čísel a drak vám řekne, zda se v ní jeho číslo nachází či nikoliv. Když jste si jistí, řeknete číslo, o kterém si myslíte, že si ho drak myslí, a drak zřejmým postupem rozhodne, zda si pochutnáte na pečínce nebo skončíte na pekáči.

Nicméně drak je velmi starý a buď se definice čestnosti od dob jeho mládí trochu změnila, nebo začíná být poněkud sklerotický. Drak považuje za zcela čestné podvádět, pokud to ovšem neudělá častěji než jednou za hru. Čili ve vaší hře jedna z jeho odpovědí může (ale také nemusí) být chybná.

Pokud ovšem s odpovědí příliš otálíte a ptáte se příliš dlouho, drak začne být z nutnosti provádět komplikované operace s množinami hladový a samozřejmě v takové situaci netoužíte po tom, aby mu došla trpělivost.

Je tu ještě jeden drobný háček – v dračí řeči si nejste zrovna nejjistější a drak sice češtinu ovládá bez problémů, nicméně nedávno si nechal nabrousit zuby a jeho výslovnost od té doby není příliš dobrá. Abyste předešli nedorozumněním, dohodli jste se, že budete komunikovat v jazycích zvaných Pascal nebo C a místo toho, abyste hráli přímo, popíšete svou strategii jako program.

P.S.: Zaručená informace od předkrmu říká, že při $N = 1\,000\,000$ drakovi trpělivost vydrží alespoň 26 kol.

16-2-5 Král Popleta — 10 bodů

Panovník Popleta XI. (nebo že by už XII., kdo ví?) jistého nejmenovaného království se již chystá na odpočinek (moderně bychom řekli do důchodu) a je potřeba, aby své království přenechal svým potomkům. A to je právě ten problém. Král má dva syny, Petra a Pavla, a shodou okolností jsou to dvojčata. Jak tak léta běžela, dvořané už dávno zapomněli, který ze synů je vlastně ten prvorozený, a bude proto potřeba království mezi ně spravedlivě rozdělit.

Království je tvořeno n provinciemi, které jsou očíslovány čísly od 1 do n. Každá připadne právě jednomu ze synů. Synové sepsali své požadavky na to, jak by takové rozdělení mělo vypadat. Petrovy podmínky jsou následujícího tvaru:

- Dostanu provincii s číslem i.
- Dostane-li provincii s číslem i druhý syn, pak já dostanu provincii s číslem j.
- Dostanu provincii s číslem i nebo s číslem j (nebo ještě lépe obě).

Stejné tři typy podmínek (až na to, že provincie chce pro sebe) má i Pavel.

Popletovi XI. jde z toho hlava kolem, a tak povolal své moudré rádce, aby království rozdělili. Rádcové podmínky synů podrobně zkoumali a brzy zjistili, že neexistuje žádné rozdělení provincií, které by všechny sepsané podmínky splnilo naráz. Na druhou stranu také zjistili, že libovolné tři podmínky naráz splnit lze, tj., např. se mezi podmínkami nevyskytuje následující trojice:

- Petr požaduje pro sebe provincii s číslem 2.
- Petr požaduje pro sebe provincii s číslem 4.
- Pavel požaduje pro sebe provincii s číslem 2 nebo s číslem 4.

I přes tato objevná pozorování, nakonec rádci králi Popletovi XI. poradili, ať si u každé provincie hodí korunou a podle výsledku hodu pak provincii přenechá buď Petrovi nebo Pavlovi. Že prý takto (v průměrném případě), splní alespoň polovinu všech podmínek, které si oba synové dohromady vymysleli. Všimněte si, že podmínky synů by byly splněny s pravděpodobností 1/2 a 3/4 (dle jejich typu) a tedy v průměrném případě je splněna alespoň polovina všech podmínek. Popletovi XI. se to však nějak nezdálo (přece nebude házet svou zlatou královskou korunou, co nosí na hlavě, to dá rozum) a obrátil se na vás.

Vaším úkolem je vymyslet pravděpodobnostní algoritmus, který v průměrném případě splní co největší množství podmínek na rozdělení království. Tedy, nalezněte co největší číslo α, $0 \leq \alpha \leq 1$, a k němu příslušný pravděpodobnostní algoritmus takový, že střední hodnota počtu splněných podmínek na rozdělení království je αm, tj. rozdělení provincií nalezené algoritmem průměrně splní alespoň αm podmínek ze všech m podmínek. Váš algoritmus by měl pracovat v polynomiálním čase (král chce předat království svým synům ještě za svého života).

16-2-K Recepty z programátorské kuchařky — Tomás Valla

Jestli programátor ve své praxi bude dělat něco velmi často, tak to bude dozajista třídění nejrůznějších dat. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale chceme stanovit jejich správné pořadí. Se setříděnými údaji se mnohem lépe pracuje, usnadníme si zejména pozdější vyhledávání uložených dat. Třídící algoritmy jsou jedny z nejstudovanějších algoritmů, my však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

V programech obvykle třídíme jednotlivé exempláře nějaké datové struktury typu pascalského záznamu. V takové struktuře bývá obsažena jedna význačná položka označovaná většinou jako *klíč*, podle které se záznamy řadí. Abychom si zjednodušili výklad, budeme nadále předpokládat, že třídíme záznamy obsahující pouze klíč, a to dejme tomu celočíselný. Čísla budeme chtít seřadit od nejmenšího k největšímu. Pomocí počtu tříděných čísel N budeme vyjadřovat časovou složitost jednotlivých algoritmů. Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na tzv. *vnitřní třídění,* kdy si můžeme dovolit všechna data načíst do paměti počítače, a *vnější třídění,* kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. My se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 20;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod.* Všechny mají několik společných rysů: jsou krátké a jednoduché a třídí přímo v poli (nepotřebujeme pomocné). Tyto algoritmy mají časovou složitost $O(N^2)$. Z toho vyplývá, že jsou použitelné, jen když tříděných dat není příliš mnoho. Stručně si přiblížíme tři nejznámější přímé metody:

Třídění přímým výběrem (SelectSort) spočívá v opakovaném vybírání nejmenšího čísla m z dosud nesetříděných čísel. Nalezené číslo m si šikovně prohodíme se začátkem pole a postup opakujeme, tentokrát na indexech $2, \ldots, N$.

Nalezený nejmenší prvek se tak dostane těsně za číslo m. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole.

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
   for i:=1 to N-1 do
   begin
      k:=i;
      for j:=i+1 to N do
         if A[j] < A[k] then k:=j;
      if k > i then
         begin x:=A[k]; A[k]:=A[i]; A[i]:=x end
   end
end;
```

Třídění přímým vkládáním (InsertSort) funguje na podobném principu, vlevo na začátku pole si střádáme již správně utříděnou posloupnost. Čísla z pravého úseku se berou jedno po druhém a do levého úseku se vkládáním zatřiďují podle velikosti, kam patří.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
   for i:=2 to N do
   begin
      x:=A[i];
      j:=i-1;
      while (j>0) and (x < A[j]) do
      begin
         A[j+1]:=A[j];
         j:=j-1;
      end;
      A[j+1]:=x;
   end
end;
```

(Upozornění: ve našich příkladech předpokládáme, že máme v překladači zapnuto zkrácené vyhodnocování logických výrazu, třeba v předchozím while cyklu se při `j=0` již s prvkem `A[0]` neporovnává. Zdrojáky pro úplné vyhodnocování si jistě každý dokáže sám upravit.)

Bublinkové třídění (BubbleSort) pracuje maličko jinak. Postupně se systematicky porovnávají dvojice sousedních prvků a vyměňují se spolu vždy, když menší číslo následuje po větším. Na konci výpočtu jsou všechny dvojice sousedních prvků uspořádané, pole je tedy setříděné. Algoritmu se říká „bublinkový“ proto, že podobně jako bublinky v limonádě stoupají vysoká čísla v poli vzhůru.

```
procedure BubbleSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    for j:=N downto i do
      if A[j-1] > A[j] then
        begin
          x:=A[j-1]; A[j-1]:=A[j]; A[j]:=x
        end
end;
```

Lepší třídící algoritmy běží v čase $O(N \log N)$. V minulé Kuchařce jsme si ukázali datovou strukturu zvanou halda. Do haldy umíme vkládat prvek v čase $O(\log N)$ a odebírat z haldy nejmenší prvek taktéž v čase $O(\log N)$. Což tedy si nejdříve všechna tříděná čísla postupně vložit do haldy a následně z haldy N-krát odebrat minimum? A ejhle, vymysleli jsme algoritmus, který na první, vkládací fázi potřebuje $N \log N$ kroků, na fázi vybírací taktéž $N \log N$ kroků.

Třídění haldou (HeapSort) tedy běží celé v čase $O(N \log N)$ a lze ho výhodně naprogramovat tak, aby potřeboval pouze jediné pole.

```
procedure HeapSort(var A: Pole);
var i,x: integer;
  { "zabublání" prvku v haldě }
  procedure bubbledown(n,i: integer);
  var j,x: integer;
  begin
    while 2*i <= n do begin
      j := 2*i;
      if (j<n) and (A[j+1] > A[j]) then j:=j+1;
      if A[i] >= A[j] then break;
      x := A[i];
      A[i] := A[j];
      A[j] := x;
      i := j;
   end;
  { postav haldu }
  for i:=N div 2 downto 1 do bubbledown(N,i);
```

```
    { vybírej nejmenší prvek }
    for i:=N downto 2 do begin
      x := A[1];
      A[1] := A[i];
      A[i] := x;
      bubbledown(i-1,1);
    end;
  end;
```

Třídění sléváním (MergeSort) je založené na principu slévání již setříděných posloupností dohromady v jedinou setříděnou. Dvě posloupnosti jednoduše v lineárním čase slijeme tak, že se díváme na nejmenší prvky obou posloupností a na výstup vydáme menší z nich, který z jeho posloupnosti poté odmažeme. Algoritmus, který umí slévat dvě posloupnosti uvnitř jednoho pole, je dosti složitý, my budeme proto výslednou slitou posloupnost ukládat do pomocného pole.

Na počátku je každý prvek jednoprvkovou setříděnou posloupností. V další fázi se ze všech sousedních jednoprvkových slitím vytvoří dvouprvkové posloupnosti, poté ze všech sousedních dvouprvkových posloupností čtyřprvková, a tak dále, obecně v i-té fázi ze dvou 2^{i-1}-prvkových posloupností vytvoříme 2^i-prvkovou. Takto pokračujeme, dokud nezbude jediná setříděná posloupnost – celé pole. Zjevně v každé fázi vykonáme $O(N)$ kroků, dvě posloupnosti totiž umíme slévat v lineárním čase vzhledem k jejich délce, a naše posloupnosti pokrývají celé tříděné pole. Počet fází bude $O(\log N)$, neboť v každé fázi pracujeme s dvojnásobně velkými posloupnostmi a tedy nejpozději v $(\log_2 N)$-té fázi již celé pole bude jedinou setříděnou posloupností. Dohromady proto MergeSort spotřebuje čas $O(N \log N)$.

V samotném programu ovšem s výhodou použijeme metodu Rozděl & Panuj: necháme si zvlášť rekurzivně setřídit levou i pravou polovinu pole a výsledky slijeme. Nevýhodou algoritmu MergeSort je, že na slévání potřebuje ještě jedno pomocné pole.

```
procedure MergeSort(var A,P:Pole; l,r:integer);
var i,j,k,s: integer;
begin
  s:=(l+r) div 2;
  if l < s then MergeSort(A, P, l, s);
  if s+1 < r then MergeSort(A, P, s+1, r);
  i:=l;
  j:=s+1;
  k:=l;
  while (i <= s) and (j <= r) do
  begin
```

```
      if A[i] <= A[j] then
        begin P[k]:=A[i]; i:=i+1 end
      else
        begin P[k]:=A[j]; j:=j+1 end;
      k:=k+1
    end;
    while i <= s do
      begin P[k]:=A[i]; i:=i+1; k:=k+1 end;
    while j <= r do
      begin P[k]:=A[j]; j:=j+1; k:=k+1 end;
    for k:=l to r do
      A[k]:=P[k]
end;
```

Jako poslední rychlý algoritmus si předvedeme *QuickSort*. Podobně jako MergeSort, i on je postaven na jednoduché myšlence a metodě Rozděl & Panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat pivot. Více o jeho volbě později. Poté prvky v poli takto přeuspořádáme: v levé části pole budou pouze prvky menší než pivot, v prostřední části prvky rovné pivotu a v pravé části prvky větší než pivot. A poté si prvky levé i pravé části rekurzivním voláním setřídíme. Jistě tak správně setřídíme celé pole.

Velká zrada ovšem spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl tzv. *medián* tříděného úseku, tj. prvek takový, jenž by se po setřídění vyskytoval uprostřed úseku. Přeuspořádání jistě zvládneme v lineárním čase a stejným argumentem jako při analýze MergeSortu (v i-té úrovni rekurze třídíme úseky dlouhé $N/2^i$) dostáváme časovou složitost $O(N \log N)$. Ačkoli existuje algoritmus, který medián pole nalezne v čase $O(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je dost velká. Namísto toho se většinou pivot volí libovolně z našeho dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že algoritmus s velmi vysokou pravděpodobností poběží v čase $O(N \log N)$. Důkaz je netriviální a my ho zde nebudeme předvádět.

Je však potřeba si uvědomit, že QuickSort může v ojedinělých případech nepříjemně zpomalit. Představme si, že pivot v každém rekurzivním volání neustále nešťastně volíme jako největší prvek z tříděného úseku. V takovém případě bude pravá část pole prázdná a levá bude mít velikost $N - 1$. Rekurze dosáhne hloubky N a čas tak vyjde $O(N^2)$.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
```

```
    k:= A[(i+j) div 2];
    repeat
      while A[i] < k do i:=i+1;
      while A[j] > k do j:=j-1;
      if i<=j then
      begin
        x:=A[i]; A[i]:=A[j]; A[j]:=x;
        i:=i+1;
        j:=j-1;
      end
    until i >= j;
    if j>l then QuickSort(A, l, j);
    if i<r then QuickSort(A, i, r);
  end;
```

Trošičku stranou všech zmíněných algoritmů stojí *přihrádkové třídění (RadixSort)*. RadixSort se vyznačuje tím, že tříděná čísla sice musí ležet v jistém nevelkém intervalu, nicméně s touto podmínkou je zvládne setřídit v lineárním čase. Předpokládejme tedy, že všechna tříděná čísla jsou z intervalu $[D, H]$. Připravíme si $H - D + 1$ přihrádek indexovaných čísly $D, D+1, \ldots, H-1, H$. Postupně čteme tříděné záznamy a „sypeme" je do té přihrádky, jejíž index se shoduje s klíčem záznamu. Po přečtení vstupu projdeme přihrádky od nejmenší k největší a vypíšeme obsažené záznamy. Zjevně jsme data setřídili v čase $O(N + (H - D))$, tedy $O(N)$ pokud jsou D a H konstantní.

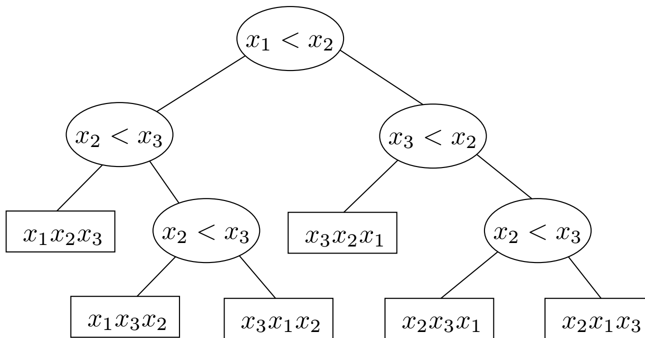
Pokud by bylo třeba přihrádek příliš mnoho, můžeme použít tzv. *víceprůchodový* RadixSort. V první fázi čísla rozdělíme do přihrádek podle nejméně významné cifry. Je důležité, aby se uvnitř přihrádky zachovalo pořadí, v jakém byla čísla vložena. Poté postupně vybereme prvky od nejnižší po nejvyšší přihrádku. V druhé fázi čísla opětovně rozdělíme do přihrádek, tentokrát podle druhé cifry, a opět je z nich vybereme. A tak dále, dokud nezpracujeme všechny cifry (jejichž počet je podle zadání úlohy nějaký malý). Zbývá si jen rozmyslet, že po i-té fázi jsou čísla správně utříděná podle i-té a nižších cifer.

```
const
  D = 1;
  H = 100;
procedure RadixSort(var A: Pole);
var C: array[D..H] of integer;
   i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]] := C[A[i]] + 1;
  k:=1;
```

```
    for i:=D to H do
      for j:=1 to C[i] do
      begin
        A[k]:=i;
        k:=k+1;
      end
  end;
```

Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídit obecné údaje, se kterými neumíme provádět nic jiného, než je navzájem porovnávat, rychleji než $O(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou popřehází. Předpokládejme pro jednoduchost, že všechny tříděné údaje jsou navzájem různé. Porovnávací činnost algoritmu si můžeme popsat tzv. *rozhodovacím stromem.* Zde je příklad rozhodovacího stromu pro tříprvkové pole:



Každý vrchol obsahuje porovnání dvou prvků x a y, v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.

Kolik maximálně hladin h a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím odpovídají různé listy. V i-té hladině se počet vrcholů oproti $(i-1)$-ní hladině nejvýše zdvojnásobí, platí tedy

$$2^h \geq N!,$$

neboli

$$h \geq \log_2(N!).$$

Existuje několik způsobů, jak si h zdola odhadnout, my použijeme skutečnost, že

$$n^n \geq n! \geq n^{n/2}.$$

Rozepsáno

$$h \geq \log_2(N!) \geq \log_2(N^{N/2}) \geq \frac{N}{2} \log N.$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $N \log N$ kroků. Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrný* čas třídění nemůže být rychlejší než $N \log N$.

16-3-1 Fyzikova blecha — 10 bodů

Newtoon trénuje svoji blechu na bleší turnaj. Ten probíhá na svislé stěně, na které jsou vodorovné plošinky. Cílem blechy je slézt ze startovací polohy co nejdříve na podlahu. Pohyb blechy závisí na tom, zda je blecha na nějaké plošince, či padá. Pokud padá, klesne za jednu blechovteřinu o jeden blechometr. Pokud je na plošince, posune se za jednu blechovteřinu o jeden blechometr vlevo či vpravo.

Závod tedy probíhá tak, že blecha padá, padá, až dopadne na plošinku. Pak se rozhodne (nebo jí její majitel přikáže), zda půjde doleva nebo doprava, a jde, dokud nedojde na konec plošinky. Z ní pak seskočí a zase padá, dokud se nedostane na podlahu. Vítězí blecha, která přistane na podlaze jako první. Ovšem je nutné, aby žádná blecha nespadla z větší výšky než v blechometrů, jinak se totiž po dopadu urazí a odmítne pokračovat v závodě.

Newtoon již vytrénoval svou blechu tak, že ho poslouchá na slovo. Problém je ten, že sám neví, jak blechu navigovat, aby prošla bludištěm nejkratší možnou cestou. Protože Vás ale zajímá, jak vypadá blecha, která poslouchá, rozhodli jste se Newtoonovi pomoci.

Na vstupu dostanete jednak počáteční souřadnice blechy (v celých blechometrech), v, což je největší výška, ze které může blecha spadnout, aby se neurazila, a N, což je počet plošinek na stěně. Dále dostanete popis N plošinek, u každé plošinky souřadnice jejího levého horního rohu a její šířku (vše opět

v celých blechometrech). Všechny plošinky jsou vysoké jeden blechometr a žádné dvě se nedotýkají. Blecha je na podlaze, pokud se nachází na souřadnicích $[x; 0]$, kde x je libovolné celé číslo.

Výstupem Vašeho programu je nejmenší počet blechovteřin, které bude blecha potřebovat, aby se dostala na podlahu. Kromě tohoto počtu vypište i počet plošinek, na které blecha dopadne, a u každé plošinky (v pořadí, jak na ně blecha dopadá) rozhodněte, zda má blecha jít vlevo či vpravo. Pokud úloha nemá řešení (moc malé v), vypište odpovídající zprávu.

Příklad: Blecha se nachází na souřadnicích $[5; 12]$, $v = 4$, $N = 3$. Plošinky jsou $[3; 8]; 5$, $[3; 4]; 5$ a $[7; 6]; 3$ ([souřadnice levého horního rohu]; délka). Nejkratší cesta trvá 17 blechovteřin, blecha navštíví všechny tři plošinky a půjde vpravo na první, vlevo na druhé a vpravo na třetí navštívené plošince.

16-3-2 Historikova past — 10 bodů

Váš přítel historik Vykopávka si na Vás přichystal, aby vylepšil Vaše (podle něj poněkud zanedbané) historické vzdělání, následující hru. Hraje se ve dvourozměrném bludišti $N \times M$ polí. V bludišti se (samozřejmě kromě zdí, ty se vyskytují v každém pořádném bludišti) nachází Theseus a Minotaurus. Abyste uspokojili svého přítele Vykopávku, musíte si tato jména nejprve zapamatovat. Theseus a Minotaurus. Theseus a Minotaurus...

Vy budete ovládat Thesea a budete se snažit dostat z bludiště ven, čili dostat se na hranici bludiště a následujícím krokem z bludiště utéct. Ovšem nikdy nesmíte narazit na Minotaura, sic bídně zhynete. Na Minotaura narazíte, pokud s ním sdílíte políčko.

Jeden tah probíhá následovně: nejprve se hýbe Minotaurus a táhne k-krát následujícím způsobem. *Pokud nejsou postavy Minotaura a Thesea ve stejném sloupci, chce se Minotaurus pohnout o jedno políčko vlevo nebo vpravo, aby se Theseovi přiblížil. Pokud není na políčku, kam se Minotaurus chce posunout, zeď, skutečně se tam posune. Pokud nejsou obě postavy na stejném řádku bludiště, chce se Minotaurus pohnout nahoru či dolů opět směrem k Theseovi. Opět se na zvolené políčko Minotaurus přesune jen tehdy, není-li tam zeď. V jednom kroku provádí Minotaurus tyto pohyby v zadaném pořadí a může provést oba dva, čili se může dostat na jedno z okolních osmi políček.*

Po k takovýchto krocích Minotaura se hýbe Theseus, a to na jedno ze čtyř okolních volných polí. Takto se oba střídají na tahu, dokud buď Theseus neuteče z bludiště, nebo dokud Minotaurus nedohoní Thesea.

Vaším úkolem bude najít pro Thesea nejkratší posloupnost pohybů vlevo, vpravo, nahoru, dolů, aby se dostal bezpečně z bludiště, případně říci, že to není možné.

Příklad: Pro $k = 2$ a bludiště o rozměrech 8×10

```
XXXXXXXXXX
X.M......X
X........X
X........X
X...X....X   . - volné políčko
X.XXX....X   X - zeď
X....T....   M - Minotaurus
XXXXXXXXXX   T - Theseus
```

je nejkratší cesta z bludiště ←←→→→→→→→.

16-3-3 Genetikova evoluce **10 bodů**

Když si Blátošlap bral ponožky ze svého prádelníku, zjistil, že se mu v něm vyvinula celá kolonie neznámých živočichů. Protože to byl genetik, jal se hned tyto živočichy zkoumat a zjistil, že i když se jedná o jediný druh, jeho zástupci jsou velmi rozmanití. Každý jedinec má totiž 30 znaků, které ho charakterizují a které se mohou měnit pouze mutací. Dalším výzkumem zjistil, že v prádelníku má N různých poddruhů (poddruhy se navzájem liší alespoň v jednom znaku), i když jeden z nich převažuje.

Ihned se dovtípil, že v jeho prádelníku došlo k evoluci. A protože ho zajímá její průběh, byli jste požádáni o vyřešení tohoto problému.

Blátošlap Vám zadá N a dále popis jednotlivých poddruhů. Každý poddruh je charakterizovaný číslem $0..2^{30} - 1$, přičemž i-tý bit tohoto čísla vyjadřuje, zda je i-tý z 30 znaků, které poddruh charakterizují, přítomen. Pokud bychom si číslo poddruhu napsali ve dvojkové soustavě, i-tý bit tohoto čísla je i-tá cifra (počítáno zprava od nuly) v tomto zápisu. Jinak řečeno, pokud číslo i-krát vydělíme dvěma a spočteme zbytek po dělení tohoto čísla dvěma, výsledek je i-tý bit původního čísla.

Vaším úkolem je zjistit, jaký poddruh se vyvinul z jakého, a počet mutací, ke kterým muselo při tomto vývoji dojít. Předpokládejte, že vývoj probíhal tak, že každý poddruh kromě toho, který Vám Blátošlap zadal jako první (to je ten nejpočetnější), se vyvinul právě z jednoho jiného poddruhu. Navíc první zadaný poddruh je původním prapředkem všech poddruhů v prádelníku. Dále předpokládejte, že evoluce probíhala nejjednodušší možnou cestou, a tedy počet mutací v evoluci je nejmenší možný. Počet mutací je součet všech rozdílných znaků mezi každým poddruhem (kromě prvního zadaného) a jeho předkem.

Navíc žádný poddruh v Blátošlapově prádelníku nevymřel, všechny evolucí vzniklé poddruhy přežily až do dnešní doby.

Příklad: Pro $N = 4$ a poddruhy 0,3,7,12 je hledaná evoluce tato: poddruhy 3 a 12 se vyvinuly z poddruhu 0, poddruh 7 se vyvinul z poddruhu 3. Počet mutací, ke kterým muselo v evoluci dojít, je roven 5.

16-3-4 Ekonomova odměna 10 bodů

Dlouhoprst byl velmi bohatý ekonom. Jednoho dne si umínil, že by zase mohl zvýšit své jmění. Ale jak? Vždyť všichni lidé už zjistili, že jeho finanční poradenství je spíše finančním proradenstvím. A tak se rozhodl dojít za ďáblem a upsat mu svou duši za pár stříbrňáků.

Po cestě do pekla potkal skupinku pirátů, kteří se handrkovali nad truhlou plnou stříbrňáků. Dlouhoprstovi to nedalo, dal se s nimi do řeči a zjistil, že se piráti domlouvají, jak si poklad rozdělí. Již se shodli na následujícím způsobu dělení.

Řekněme, že v truhle je S stříbrňáků a pirátů je P. Piráti se očíslují, čili každý dostane právě jedno číslo od 1 do P. Pirát s pořadovým číslem P navrhne způsob rozdělení všech stříbrňáků mezi piráty (každému pirátovi včetně sebe přidělí nezáporný počet stříbrňáků). Ostatní o tomto rozdělení hlasují a pokud je jich alespoň polovina pro, návrh je přijat. V opačném případě je pirát P zabit a navrhuje pirát s číslem $P - 1$.

Při hlasování se piráti řídí těmito pravidly: nejdůležitější je přežít. Pokud pirát ví, že v budoucnu určitě zemře, hlasuje vždy pro, nezávisle na výši svého podílu. Druhé pravidlo je zisk. Pokud má pirát v budoucnu šanci, že získá víc, než kolik je mu právě nabídnuto, hlasuje proti (ovšem jen pokud ví, že přežije). Třetí pravidlo je touha popravit co nejvíc pirátů, čili pokud pirát ví, že v budoucnu bude moci určitě získat stejný obnos, jaký je mu nabídnut, hlasuje proti. Důležité je, že se všichni piráti řídí stejnými pravidly a navzájem to o sobě vědí.

Dlouhoprst vycítil příležitost, a protože piráti se doteď nedohodli na tom, jak se očíslují, nabídl jim své finanční proradenství. Očísluje je, ale za odměnu se bude moci do rozdělování zapojit, a to s pořadovým číslem $P + 1$. A piráti souhlasili. Dlouhoprst ovšem zjistil, že je to i nad jeho síly, a poprosil Vás o pomoc.

Váš program dostane na vstupu čísla S (počet stříbrňáků v pokladu) a P (počet pirátů). Musíte zjistit, zda může Dlouhoprst vůbec přežít a pokud ano, poraďte mu návrh rozdělení stříbrňáků mezi P pirátů takový, aby byl jeho zisk (počet stříbrňáků, které nemusí mezi piráty rozdělit) co největší.

Příklad: Pro $S = 100$ a $P = 2$ Dlouhoprst přežít může a jeho zisk je 100 stříbrňáků. Jeho návrh je dát oběma pirátům nula stříbrňáků. Pirát číslo 2 pro jeho návrh hlasuje, protože jinak (kdyby byl Dlouhoprst popraven) by pirát 1 hlasoval proti libovolnému jeho návrhu a on sám by byl popraven.

16-3-5 Botanikova setba 10 bodů

Známý botanik Konopník se rozhodl, samozřejmě z čistě vědeckých důvodů, zasít svojí nejoblíbenější rostlinu. Bohužel je pronásledován davy laiků, kteří

by se rádi podíleli na sklizni, a tak je nucen zasít na poli, kde už je zaseta kukuřice, aby jeho úroda nebyla lidem na očích.

Protože ale Konopník chce, aby úroda byla co největší, rozdělil kukuřičné pole na pravidelnou síť oblastí a u každé oblasti zjistil její předpokládanou úrodnost. Ta může být i záporná, pokud daná oblast brání v pěstování na okolních oblastech. Konopník by chtěl osít jediné pravoúhlé území takové, aby součet úrodností na tomto území byl největší možný. Ovšem protože je biolog, obrátil se s tímto problémem na Vás.

Na vstupu dostanete N a M, čili šířku a délku kukuřičného pole. Dále dostanete matici $N \times M$, jejíž hodnoty jsou úrodnosti jednotlivých oblastí kukuřičného pole. Vaším úkolem je najít podmatici (souvislou pravoúhlou oblast) takovou, že součet jejích prvků je největší možný. Pokud je takových podmatic víc, najděte takovou, která má nejmenší obsah.

Příklad: Pro pole o rozměrech 4×5 a úrodnostech

$$\begin{pmatrix} 1 & -1 & 4 & 5 & 3 \\ -2 & 3 & -2 & -5 & 8 \\ 2 & -4 & 2 & 1 & -7 \\ 4 & -3 & 2 & 1 & 1 \end{pmatrix}$$

má hledaná podmatice levý roh v prvním řádku a druhém sloupci, pravý dolní roh v druhém řádku a pátém sloupci.

16-3-6 Matematikova sonda — 10 bodů

Matematici jsou zvláštní lidé a jako takovým se jim občas v hlavě zhmotní zvláštní nápady. Jako třeba vyslat sondu na Mars.

K tomuto problému je samozřejmě potřeba přistupovat systematicky. Náš matematik si nejprve studiem odborné literatury a předchozích publikovaných prací v tomto oboru zjistil, že existují dva postupy.

- Může si za v zásadě nepatrnou částku (kterou si označíme jako jednu jednotku) nechat sondu vyrobit a vyslat. Je ovšem možné, že se něco nezdaří a pokus bude potřeba opakovat.
- Může si ji vlastnoručně sestrojit za $k > 1$ jednotek. Samozřejmě si je naprosto jistý, že v něčem tak triviálním by přece on nemohl udělat chybu.

Chtěl by samozřejmě zvolit levnější variantu, nicméně která to je, záleží na tom, kolik koupených sond se rozbije. Proto vymyslel následující algoritmus:

Prvních $k - 1$ sond si koupí. Pokud žádná z nich neuspěje, sondu si sám vyrobí.

Tento algoritmus mu zajišťuje, že nikdy nezaplatí víc než $(2 - 1/k)$ krát tolik, kolik by musel: Nechť uspěje n-tá sonda. Pak buď $n \leq k - 1$, pak zaplatí

n jednotek, což je optimální. Nebo $n \geq k$, pak zaplatí $2k - 1$ jednotek, zatímco optimální řešení by bylo vyrobit si sondu rovnou a zaplatit pouze k jednotek.

A také si okamžitě dokázal, že tento postup je nejlepší možný. Zjevně jediná možnost, jak si volit strategii, je nějakým způsobem určit počet sond d, po jejichž zničení si sondu vyrobí. Nechť uspěje $(d + 1)$-ní sonda, pak zaplatí $k + d$ jednotek. Pokud by d bylo větší nebo rovno k, optimum je k, tedy by zaplatil alespoň dvakrát tolik, kolik musel. Naopak kdyby d bylo menší než $k - 1$, optimum je $d + 1$ a $\frac{k+d}{d+1} \geq \frac{(d+2)+d}{d+1} \geq 2$.

Po chvíli si však uvědomil, že by možná mohl ušetřit alespoň v průměrném případě, kdyby jednu z mincí našetřených na pořízení sondy použil jako generátor náhodných čísel. Nicméně protože je to starý zkušený profesor, přenechal vám vyřešení této úložky jako jednoduché domácí cvičení.

Vaším úkolem je tedy nalézt pravděpodobnostní strategii, pro kterou by existovala nějaká konstanta c (pokud možno co nejmenší) taková, že ať už se porouchá libovolný počet sond, profesor zaplatí v průměru nanejvýš c-krát tolik, než kolik musí.

16-3-K Recepty z programátorské kuchařky Dan Král', MM a MS

V dnešním, v pořadí již třetím, dílu kuchařky popíšeme problém minimální kostry a ukážeme si, jak ho řešit. Také popíšeme datovou strukturu *Disjoint-Find-Union* (její název je často zkracován na DFU), kterou šikovně použijeme právě na hledání minimální kostry grafu.

Co je to *graf* jsme si vysvětlili již dříve. *Podgraf* nějakého grafu vznikne z původního grafu odebráním libovolných vrcholů a libovolných hran. *Cesta* v grafu je posloupnost vrcholů taková, že každý vrchol je s následujícím spojen hranou a vrcholy se na cestě neopakují (často za cestu považujeme i tyto hrany). *Kružnice* v grafu je podobná posloupnost vrcholů, pouze první vrchol je shodný s posledním a nikde jinde se opět vrcholy neopakují. (Pro úplnost: *tah* bychom tomu říkali tehdy, kdyby se vrcholy mohly opakovat, ale hrany nikoliv, a *sled* tehdy, pokud by se mohlo opakovat cokoliv.)

Pokud v grafu mezi každými dvěma vrcholy existuje cesta (což je totéž, jako když mezi nimi existuje sled, rozmyslete si, proč), říkáme takovému grafu *souvislý*. Pokud graf souvislý není, můžeme ho rozdělit na podgrafy, které již souvislé jsou a nevedou mezi nimi žádné hrany. Takovým podgrafům se pak říká *komponenty souvislosti*. U souvislého grafu za komponentu souvislosti považujeme celý graf.

Strom je graf, který je souvislý a zároveň *acyklický* (čili neexistuje v něm žádná kružnice). *List* stromu je takový vrchol, ze kterého vede jen jedna hrana. Každý konečný strom o dvou či více vrcholech má vždy alespoň dva listy. Můžeme je nalézt třeba tak, že sestrojíme ve stromě nejdelší cestu (rozmyslete si, proč bude mít tato cesta konečnou délku) a všimneme si, že koncové vrcholy

této cesty jsou hledané listy (kdyby z koncových vrcholů cesty vedla více než jedna hrana, pak bychom ji buďto o ni buďto mohli cestu prodloužit nebo by tato hrana vedla do vrcholu, který na cestě již leží, jenže tím by tvořila kružnici).

Zajímavé je to, že strom s N vrcholy má právě $N - 1$ hran. To můžeme dokázat indukcí podle počtu vrcholů stromu: Strom s jedním vrcholem neobsahuje hranu žádnou. Pokud máme strom s $N > 1$ vrcholy, víme, že má alespoň dva listy. Vezmeme tedy libovolný list a ze stromu ho odebereme. Tím získáme opět strom (souvislost jsme porušit nemohli a kružnici jsme také nevytvořili) a jeho počet vrcholů je o 1 menší, čili podle indukčního předpokladu má o 1 méně hran než vrcholů. Nyní list „přilepíme“ zpět, čímž zvýšíme počet vrcholů i hran o 1 a tvrzení stále platí.

A nyní k slibovaným kostrám. *Kostra* grafu budeme říkat podgrafu, který obsahuje všechny vrcholy a nejmenší počet hran takový, aby cesta mezi dvěma různými vrcholy existovala v kostře grafu právě tehdy, existuje-li v grafu původním. Všimněte si, že kostra souvislého grafu je sama souvislá a navíc neobsahuje žádnou kružnici (jinak bychom mohli libovolnou hranu ležící na kružnici z kostry beze škody odebrat, čímž bychom získali menší kostru, a to nám definice zakazuje.) Čili každá kostra souvislého grafu je strom a jelikož všechny stromy na N vrcholech mají $N - 1$ hran, všechny kostry také. Pokud je graf nesouvislý, stačí tuto úvahu provést pro každou komponentu zvlášť a zjistíme, že kostra je *les* (to je graf, jehož každá komponenta je strom) s $N - k$ hranami, kde k je počet komponent souvislosti.

Pokud každé hraně grafu přiřadíme nějaké *ohodnocení,* což bude nějaké číslo (pro naše potřeby vždy kladné), dostaneme *ohodnocený graf.* V takových grafech pak obvykle hledáme mezi všemi kostrami *kostru minimální,* a to je taková, pro kterou je součet ohodnocení jejích hran nejmenší možný.

Algoritmus hledání minimální kostry

Náš algoritmus na hledání minimální kostry (tzv. hladový algoritmus) bude velmi jednoduchý. Nejprve setřídíme hrany vzestupně podle jejich váhy. Kostru budeme postupně vytvářet přidáváním hran od té s nejmenší vahou tak, že hranu do kostry přidáme právě tehdy, pokud spojuje dvě (prozatím) různé komponenty souvislosti vytvořeného podgrafu. Jinak řečeno, hranu do vytvářené kostry přidáme, pokud v ní zatím neexistuje cesta mezi vrcholy, které zkoumaná hrana spojuje.

Je zřejmé, že tímto postupem získáme kostru, tj. acyklický podgraf grafu, který je souvislý (pokud vstupní graf je souvislý, ale toto mlčky předpokládáme). To, že nalezená kostra je minimální, si ukážeme později. Nyní se podívejme na časovou složitost našeho algoritmu: Pokud vstupní graf má N vrcholů a M hran, pak úvodní setřídění hran vyžaduje čas $O(M \log M)$ (použijeme nějaký

z rychlých třídících algoritmů popsaných v minulém díle) a poté se pokusíme přidat každou z M hran. V druhé části kuchařky si ukážeme datovou strukturu, s jejíž pomocí bude M testů toho, zda mezi dvěma vrcholy vede hrana, trvat méně než $O(M \log M)$. Celková časová složitost našeho algoritmu je tedy $O(M \log N)$ (všimněte si, že $\log M \leq \log N^2 = 2 \log N$). Paměťová složitost je lineární vzhledem k počtu hran, tj. $O(M)$.

Důkaz správnosti hladového algoritmu

Zbývá dokázat, že nalezená kostra vstupního grafu je minimální. Bez újmy na obecnosti můžeme předpokládat, že váhy všech hran grafu jsou vesměs různé: Pokud tomu tak není již na začátku, přičteme k některým z hran, jejichž váhy jsou duplicitní, velmi malá kladná celá čísla tak, aby pořadí hran nalezené naším třídícím algoritmem zůstalo zachováno. Tím se kostra nalezená naším algoritmem nezmění a pokud bude tato kostra minimální s modifikovanými váhami, bude minimální i pro původní zadání.

Označme T_{alg} kostru nalezenou naším algoritmem a T_{min} minimální kostru. Nechť $e_1, \ldots, e_{N-1}$ jsou hrany kostry T_{alg} v pořadí, v jakém byly přidány naším algoritmem do kostry. Pokud jsou kostry T_{alg} a T_{min} různé, pak existuje hrana e_i, která není obsažena v kostře T_{alg}. Zvolme nejmenší takové i, tzn., že kostra T_{min} obsahuje všechny hrany $e_1, \ldots, e_{i-1}$. Váha každé hrany kostry T_{min} kromě $i-1$ hran $e_1, \ldots, e_{i-1}$ je ostře větší než váha hrany e_i: Kdyby tomu tak nebylo, pak by kostra T_{min} obsahovala hranu f s menší vahou než e_i a náš algoritmus by hranu f přidal k hranám $e_1, \ldots, e_{i-1}$ při vytváření kostry T_{alg}. Hrana f pak by musela být mezi hranami $e_1, \ldots, e_{i-1}$, protože její váha je menší než váha hrany e_i, ale tomu tak není. Tedy taková hrana f neexistuje.

Přidejme nyní hranu e_i ke kostře T_{min}. Takto vzniklý podgraf vstupního grafu zjevně obsahuje kružnici (už před přidáním hrany e_i existovala v kostře T_{min} cesta mezi koncovými vrcholy hrany e_i, protože kostra T_{min} je souvislá). Tuto kružnici si označme C. Protože T_{alg} neobsahuje žádnou kružnici, kružnice C obsahuje alespoň jednu hranu e', která není v kostře T_{alg}. Protože hrana e' není žádnou z hran $e_1, \ldots, e_{i-1}$, je její váha ostře větší než váha hrany e_i. Odstraňme nyní hranu e'. Označme T' výsledný podgraf vstupního grafu, tj. graf získaný z kostry T_{min} záměnou hrany e' za hranu e_i. Protože e' a e_i ležely ve společné kružnici C, je T' souvislý podgraf, a tedy kostra vstupního grafu. Součet vah hran kostry T' je však ostře menší než součet vah hran kostry T_{min}, což není možné, neboť T_{min} je minimální kostra vstupního grafu. Tedy neexistuje žádné i takové, že hrana e_i není obsažena v kostře T_{min} a kostry T_{alg} a T_{min} jsou shodné.

Disjoint-Find-Union

Datová struktura DFU slouží k udržování rozkladu množiny na několik disjunktních podmnožin (čili takových, že žádné dvě nemají společný prvek). To

znamená, že pomocí této struktury můžeme pro každé dva z uložených prvků říci, zda patří či nepatří do stejné podmnožiny rozkladu.

V algoritmu hledání minimální kostry budou prvky v DFU vrcholy zadaného grafu a budou náležet do jedné podmnožiny rozkladu, pokud mezi nimi již ve vytvářené kostře existuje cesta, čili podmnožiny v DFU budou odpovídat komponentám souvislosti vytvářené kostry.

S reprezentovaným rozkladem umožňuje datová struktura DFU provádět následující dvě operace:

- **find**: Test, zda dva prvky leží ve stejné podmnožině rozkladu. Tato operace bude v našem případě odpovídat testu, zda dva vrcholy leží ve stejné komponentě souvislosti. Tato operace bývá často implementována tak, že vrací nějakého pevně zvoleného reprezentanta podmnožiny. Test, zda jsou dva prvky ve stejné podmnožině se pak provede porovnáním příslušných reprezentantů. V této podobě by tato operace odpovídala níže popsané funkci `root`.
- **union**: Sloučení dvou podmnožin do jedné. Tato operace bude v naší aplikaci odpovídat přidání hrany mezi dvě různé komponenty grafu.

Povězme si nejprve, jak budeme jednotlivé podmnožiny reprezentovat. Prvky obsažené v jedné podmnožině budou tvořit zakořeněný strom. V tomto stromě však povedou ukazatele, trochu nezvykle, od listů ke kořenu. Operaci `find` lze pak jednoduše implementovat tak, že pro oba zadané prvky nejprve nalezneme kořeny jejich stromů. Jsou-li tyto kořeny stejné, jsou prvky ve stejném stromě, a tedy i stejné podmnožině rozkladu. Naopak, jsou-li různé, jsou zadané prvky v různých stromech, a tedy jsou i v různých podmnožinách reprezentovaného rozkladu. Operaci `union` provedeme tak, že mezi kořeny stromů reprezentujících slučované podmnožiny přidáme ukazatel a tím tyto dva stromy spojíme dohromady.

Implementace dvou výše popsaných operací, jak jsme si ji právě popsali, následuje. Pro jednoduchost množina, jejíž rozklad reprezentujeme, bude množina čísel od 1 do N. Ukazatele ve stromě si pak pamatujeme v poli `parent`, kde 0 znamená, že prvek nemá rodiče, tj. je kořenem svého stromu. Funkce `root(v)` vrací kořen stromu, který obsahuje prvek v.

```
var parent:array[1..N] of integer;

procedure init;
 var i:integer;
 begin
  for i:=1 to N do parent[i]:=0;
 end;
```

```
function root(v: integer):integer;
 begin
  if parent[v]=0 then root:=v
  else root:=root(parent[v]);
 end;

function find(v,w:integer):boolean;
 begin
  find:=(root(v)=root(w));
 end;

procedure union(v,w:integer);
 begin
  v:=root(v);
  w:=root(w);
  if v<>w then parent[v]:=w;
 end;
```

S právě předvedenou implementací operací `find` a `union` by se nám mohlo stát, že stromy odpovídající podmnožinám budou vypadat jako „hadi“ a pokud budou obsahovat N prvků, nalezení kořene zabere čas $O(N)$. Tedy operace `find` a `union` vyžadují čas až $O(N)$.

Ke zrychlení práce DFU se používají dvě jednoduchá vylepšení:

- **union by rank**: Každý prvek má přiřazen `rank`. Na začátku jsou ranky všech prvků rovny nule. Při provádění operace `union`, připojíme strom s kořenem menšího ranku ke kořeni stromu s větším rankem. Ranky kořenů stromů se v tomto případě nemění. Pokud kořeny obou stromů mají stejný rank, připojíme je libovolně, ale rank kořenu výsledného stromu zvětšíme o jedna.
- **path compression**: Ve funkci `root(v)` přepojíme všechny prvky na cestě od prvku v ke kořeni rovnou na kořen, tj. změníme jejich rodiče na kořen daného stromu.

Než si obě metody blíže rozebereme, podívejme se, jak se změní funkce `root` a `union`:

```
var parent:array[1..N] of integer;
  rank:array[1..N] of integer;

procedure init;
 var i:integer;
 begin
  for i:=1 to N do
```

```
   begin
    parent[i]:=0;
    rank[i]:=0;
   end;
 end;

{zmena kvuli path compression}
function root(v: integer):integer;
 begin
  if parent[v]=0 then root:=v
  else begin
   parent[v]=root(parent[v]);
   root:=parent[v];
  end;
 end;

{stejna jako minule}
function find(v,w:integer):boolean;
 begin
  find:=(root(v)=root(w));
 end;

{zmena kvuli union by rank}
procedure union(v,w:integer);
 begin
  v:=root(v);
  w:=root(w);
  if v=w then exit;
  if rank[v]=rank[w] then
   begin
    parent[v]:=w;
    rank[w]:=rank[w]+1;
   end
  else
   if rank[v]<rank[w] then parent[v]:=w
   else parent[w]:=v;
 end;
```

Zaměřme se nyní blíže na metodu „union by rank“. Nejprve učiníme následující pozorování: Pokud je prvek v s rankem r kořenem stromu v datové struktuře DFU, pak tento strom obsahuje alespoň 2^r prvků. Naše pozorování dokážeme indukcí podle r. Pro $r = 0$ tvrzení zřejmě platí. Nechť tedy $r > 0$.

V okamžiku, kdy se rank prvku v mění z $r-1$ na r, slučujeme dva stromy, jejichž kořeny mají rank $r-1$. Každý z těchto dvou stromů má dle indukčního předpokladu alespoň 2^{r-1} prvků a tedy výsledný strom má alespoň 2^r prvků, jak jsme požadovali. Z našeho pozorování ihned plyne, že rank každého prvku je nejvýše $\log_2 N$ a prvků s rankem r je nejvýše $N/2^r$ (všimněme si, že rank prvku v DFU se nemění po okamžiku, kdy daný prvek přestane být kořen nějakého stromu).

Když tedy provádíme jen „union by rank“, je hloubka každého stromu v DFU rovna ranku jeho kořene, protože rank kořene se mění právě tehdy, když zvětšujeme hloubku stromu o jedna. A protože rank každého prvku je nanejvýš $\log_2 N$, hloubka každého stromu v DFU je také nanejvýš $\log_2 N$. Tím pádem ale nalezení reprezentanta (čili kořene) libovolné skupiny trvá $O(\log N)$ a tedy operace `find` a `union` budou vyžadovat čas $O(\log N)$.

Amortizovaná časová složitost

Abychom mohli pokračovat dále, musíme si vysvětlit, co znamená *amortizovaná* časová složitost. Pokud řekneme, že nějaká operace vyžaduje amortizovaně čas $O(t)$, pak provedení k takových operací vyžaduje čas nejvýše $O(kt)$, ale provedení jedné konkrétní z nich může vyžadovat čas větší. Tento větší čas je pak ve výsledném odhadu kompenzován kratším časem, který spotřebovaly některé předchozí operace.

Předveďme si to na příkladě. Řekněme, že máme číslo zapsané ve dvojkové soustavě. Přičíst k tomuto číslu jedničku jistě netrvá konstantní čas, záleží na tom, kolik jedniček se vyskytuje na konci zadaného čísla. Pokud se nám ale povede ukázat, že přičíst jedničku k tomuto číslu N-krát nám zabere čas $O(N)$, pak můžeme říci, že přičíst k číslu jedničku trvá amortizovaně $O(1)$.

Jak tedy ukážeme, že N přičtení jedničky k číslu zabere čas $O(N)$? Použijeme k tomu „penízkovou metodu“. Každá operace nás bude stát jeden penízek a pokud jich na N operací použijeme jen $O(N)$, uspějeme. Předpokládejme tedy, že každá jednička v dvojkovém zápisu daného čísla už má jeden svůj penízek (když začneme jedničky přičítat k nule, tuto podmínku splníme). Každé jedničce, kterou chceme přičíst, dáme ze začátku dva penízky. Přičítání bude probíhat tak, že přičítaná jednička se „podívá“ na poslední bit (tj. ve dvojkovém zápise na poslední cifru) zadaného čísla (to ji stojí jeden penízek). Pokud je to nula, změní ji na jedničku a dá jí svůj zbylý penízek. Pokud to je jednička, vezme si přičítaná jednička její penízek (čili už má zase dva), změní zkoumanou jedničku na nulu a pokračuje u dalšího bitu… dokud nenajde nulu. Takto zachová podmínku, že každá jednička v dvojkovém zápisu čísla má jeden penízek.

Tedy N přičítání jednotky nás stojí $2N$ penízků a tím pádem čas $O(N)$. Ačkoli jedno konkrétní přičtení jedničky k danému číslu může trvat déle než konstantní čas, amortizovaná složitost přičtení jedničky k číslu je konstantní.

Dokončení analýzy DFU

Pokud bychom prováděli pouze „path compression“ a nikoliv „union by rank“, dalo by se dokázat, že každá z operací `find` a `union` vyžaduje amortizovaně čas $O(\log N)$. To ale nebudeme dokazovat, protože tím bychom si nijak oproti samotnému „union by rank“ nepomohli. Proč tedy vlastně hovoříme o obou vylepšeních? Inu proto, že při použití obou současně dosáhneme mnohem lepšího amortizovaného času $O(\alpha(N))$ na jednu operaci `find` nebo `union`, kde $\alpha(N)$ je inverzní Ackermannova funkce. Její definici můžete nalézt na konci kuchařky, zde jen poznamenejme, že hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pro všechny praktické hodnoty N nejvýše čtyři. Čili dosáhneme v podstatě amortizovaně konstantní časovou složitost na jednu (libovolnou) operaci DFU.

My si zde předvedeme poněkud horší, ale technicky jednodušší časový odhad $O((N+L)\log^* N)$, kde L je počet provedených operací `find` nebo `union` a $\log^* N$ je tzv. iterovaný logaritmus, jehož definice následuje. Nejprve si definujeme funkci $2 \uparrow k$ rekurzivním předpisem:

$$2 \uparrow 0 = 1, \quad 2 \uparrow k = 2^{2\uparrow(k-1)}.$$

Máme tedy $2 \uparrow 1 = 2$, $2 \uparrow 2 = 2^2 = 4$, $2 \uparrow 3 = 2^4 = 16$, $2 \uparrow 4 = 2^{16} = 65536$, $2 \uparrow 5 = 2^{65536}$, atd. A konečně, iterovaný logaritmus $\log^* N$ čísla N je nejmenší přirozené číslo k takové, že $N \leq 2 \uparrow k$. Jiná (ale ekvivalentní) definice iterovaného logaritmu je ta, že $\log^* N$ je nejmenší počet, kolikrát musíme číslo N opakovaně logaritmovat, než dostaneme hodnotu menší nebo rovnu jedné.

Zbývá provést slíbenou analýzu struktury DFU s vylepšujícími metodami „union by rank“ a „path compression“. Prvky si rozdělíme do skupin podle jejich ranku: K-tá skupina prvků bude tvořena těmi prvky, jejichž rank je mezi $(2 \uparrow (K-1)) + 1$ a $2 \uparrow K$. Např. třetí skupina obsahuje ty prvky, jejichž rank je mezi 5 a 16. Prvky jsou tedy rozděleny do $1 + \log^* \log N = O(\log^* N)$ skupin. Odhadněme shora počet prvků v K-té skupině:

$$\frac{N}{2^{(2\uparrow(K-1))+1}} + \cdots + \frac{N}{2^{2\uparrow K}} = \frac{N}{2^{2\uparrow(K-1)}} \cdot \left(\sum_{i=1}^{2\uparrow K - 2\uparrow(K-1)} \frac{1}{2^i} \right) \leq$$

$$\leq \frac{N}{2^{2\uparrow(K-1)}} \cdot 1 = \frac{N}{2 \uparrow K}.$$

Teď můžeme provést časovou analýzu funkce `root(v)`. Čas, který spotřebuje funkce `root(v)`, je úměrný délce cesty od prvku v ke kořeni stromu. Tato cesta je pak následně rozpojena a všechny prvky na ní jsou přepojeny přímo na kořen stromu. Rozdělíme rozpojené hrany této cesty na ty, které „naúčtujeme“ tomuto volání funkce `root(v)`, a ty, které zahrneme do faktoru $O(N \log^* N)$ v dokazovaném časovém odhadu. Do volání funkce `root(v)` započítáme ty

hrany cesty, které spojují dva prvky, které jsou v různých skupinách. Takových hran je zřejmě nejvýše $O(\log^* n)$ (všimněte si, že ranky prvků na cestě z listu do kořene tvoří rostoucí posloupnost).

Uvažme prvek v v K-té skupině, který již není kořenem stromu. Při každém přepojení rank rodiče prvku v vzroste. Tedy po $2 \uparrow K$ přepojeních je rodič prvku v v $(K+1)$-ní nebo vyšší skupině. Pokud v je prvek v K-té skupině, pak hrana z něj na cestě do kořene nebude účtována volání funkce `root(v)` nejvýše $(2 \uparrow K)$-krát. Protože K-tá skupina obsahuje nejvýše $N/(2 \uparrow K)$ prvků, je počet takových hran pro všechny prvky této skupiny nejvýše N. A protože počet skupin je nejvýše $O(\log^* N)$, je celkový počet hran, které nejsou započítány voláním funkce `root(v)`, nejvýše $O(N \log^* N)$. Protože funkce `root(v)` je volána $2L$-krát, plyne časový odhad $O((N+L)\log^* N)$ z právě dokázaných tvrzení.

Inverzní Ackermannova funkce $\alpha(N)$

Ackermannovu funkci lze definovat následující konstrukcí:

$$A_0(i) = i + 1, \quad A_{k+1}(i) = A_k^i(i) \text{ pro } k \geq 0,$$

kde výraz A_k^i zastupuje složení i funkcí A_k, např. $A_1(3) = A_0(A_0(A_0(3)))$. Platí tedy následující rovnosti:

$$A_0(i) = i + 1, \quad A_1(i) = 2i, \quad A_2(i) = 2^i \cdot i.$$

Ackermannova funkce $A(k)$ je pak rovna hodnotě $A_k(2)$, čili $A(2) = A_2(2) = 8$, $A(3) = A_3(2) = 2^{11}$, $A(4) = A_4(2) \approx 2 \uparrow 2048$ atd… Hodnota inverzní Ackermannovy funkce $\alpha(N)$ je pak nejmenší přirozené číslo k takové, že $N \leq A(k) = A_k(2)$. Jak je vidět, ve všech reálných aplikacích platí, že $\alpha(N) \leq 4$.

16-4-1 Mnichova posedlost aneb Hanoi strikes back — 10 bodů

Všichni jistě víte, co jsou to Hanojské věže. Jedná se o tři kolíky, na kterých je nasazeno celkem N disků různých velikostí. Na začátku jsou všechny disky nasazeny na krajním kolíku a uspořádány podle velikosti (dole leží největší disk). Vaším úkolem je přemístit všechny disky na jiný kolík a přitom dodržet pravidlo, že vždy musí menší disk ležet na větším.

Mnichům v Tibetu se v časech dávno minulých zjevil sám Bůhdha a slíbil jim, že až přesunou v Hanojských věžích 64 disků z jednoho kolíku na jiný, svět bude u konce. A mnichové se snaží jím zadaný úkol splnit již po tisíciletí, protože k tomu potřebují (jak jistě víte) $2^{64} - 1$ přesunů.

Vzhledem k tomu, že je to práce velmi jednotvárná, stalo se jednou, že do mnicha Askejťáka vstoupil démon Kazisvět, který nechce, aby svět skončil (jinak by ho už nemohl dál kazit). Mnich pod vlivem Kazisvěta přesouval disky sice podle pravidla, že menší disk musí ležet na větším, avšak vůbec nedodržoval

již tisíce let stanovený postup. Poté, co Kazisvět odešel, mniši zjistili, že vůbec nevědí, jak dál. Ale Vy jistě vědět budete.

Na vstupu dostanete počet disků N. Předpokládejte, že disky jsou očíslovány podle velikosti od největšího (1) k nejmenšímu (N). Dále dostanete popis situace na Hanojských věžích (jaké disky jsou na kterých kolících; uvědomte si, že pořadí disků na kolíku je určeno vždy jednoznačně). Vaším úkolem je říci, na jaký kolík bude pro mnichy nejvýhodnější všechny disky nakonec nasadit a kolik k tomu budou potřebovat kroků. Můžete pro jednoduchost předpokládat, že počet *potřebných* kroků bude menší než $2^{64} - 1$.

Příklad: Pokud $N = 7$ a na prvním kolíku jsou disky $6, 7$, na druhém kolíku disk 5 a na třetím kolíku disky $1, 2, 3, 4$, je pro mnichy nejvýhodnější přemístit všechny disky na kolík číslo tři. Dokážou to pomocí čtyř přesunů.

16-4-2 Technologické trable — 8 bodů

Jedna nejmenovaná počítačová společnost nedávno zveřejnila, že hodlá vrhnout na trh naprosto originální novinku mezi úložnými zařízeními. Její naprostá nepřekonatelná moderní nedostižná a prostě skvělá přednost tkví v tom, že bude mít nekonečnou kapacitu.

Toto úložné zařízení slouží k uchování *nekonečně* mnoha celých čísel. Jednotlivá čísla jsou uložena v buňkách, které jsou očíslovány přirozenými čísly, a navíc platí, že obsahy paměťových buněk jsou uloženy ve vzestupném pořadí.

Společnost se ovšem natolik vyčerpala několikatýdenní letákovou, novinovou, televizní a billboardovou propagandou, že již není schopná napsat program, který bude s propagovaným zařízením zacházet.

Vaším úkolem je napsat program, který najde zadanou hodnotu v nekonečně velkém vzestupně uspořádaném poli se začátkem. Pozor na to, že jednotlivé hodnoty se mohou opakovat. Protože toto pole je nekonečně velké, nemůžete ho dostat na vstupu. Hodnoty toho pole budete zjišťovat tak, že se budete zařízení ptát na hodnotu v konkrétní paměťové buňce (pro nás to znamená, že se budete ptát uživatele). Výsledkem programu má být číslo paměťové buňky, kde se hledaná hodnota vyskytuje, případně -1, pokud se hledaná hodnota v zařízení nevyskytuje vůbec.

Důležité na Vašem programu je to, kolik hodnot si z popisovaného zařízení vyžádá. Zkuste také dokázat, že Vaše řešení je optimální (pokud opravdu je).

Příklad: Hledaná hodnota je 15, odpovědi na dotazy jsou tyto: v buňce 1 je uložena hodnota -8, v buňce 3 je 14 a v buňce 4 je hodnota 16. Výsledek programu je -1.

16-4-3 Stávka programátorů 10 bodů

Pondělí ráno. Ústředí televize TeleNovela. Telefon zvoní. „Prosím?“ „Pane řediteli, hrozná věc! Programátoři stávkují, prý že i programátor má právo na lidská práva!“ „Ale my jsme televize TeleNovela. Tohle nás vůbec nezajímá.“

To samé pondělí ráno. Ústředí televize CNN. Telefon zvoní. „Prosím?“ „Pane řediteli, hrozná věc! Programátoři stávkují, prý že i programátor má právo na lidská práva!“ „To je ale hrůza! O tom musíme natočit reportáž!“

Programátoři ve známém městě *Jsi-li con, válej* se rozhodli uspořádat stávku. Chtějí mít všechna lidská práva, hlavně právo dostat za práci přiměřenou odměnu. Většina z nich je totiž neúnosně přeplacena. A oni chtějí spravedlnost.

Město si můžeme představit jako pravoúhlou síť ulic, kterých je ve vodorovném i svislém směru N. Ulice si očíslujeme, ty vodorovné odspoda, ty svislé zleva. Každou křižovatku můžeme označit dvojicí (x, y), kde x je číslo ulice svislé a y je číslo ulice vodorovné.

Programátoři budou postupně stávkovat na M různých křižovatkách. A televize CNN chce o všech stávkách natočit reportáž. K tomu, aby natočili reportáž o stávce na křižovatce (x, y) však nemusí být přímo na této křižovatce, ale stačí, když jsou ve svislé ulici x nebo ve vodorovné ulici y (mají totiž velmi dobré objektivy).

Televize CNN má ústředí na křižovatce (cnn_x, cnn_y). Má k dispozici bohužel jediný vůz, který na začátku vyjede z ústředí, nafilmuje všech M stávek v pořadí, v jakém se budou konat, a vrátí se zpět do ústředí. Vaším úkolem je naplánovat jeho cestu tak, aby nafilmoval všechny stávky (jak už bylo řečeno, nemusí být přímo u stávky, stačí, když bude na ulici procházející křižovatkou stávky) a jeho cesta byla nejkratší možná.

Příklad: Pro $N = 4$, $M = 3$, stávky na křižovatkách v pořadí $(1, 4)$, $(4, 1)$, $(3, 4)$ a pro centrum CNN $(2, 2)$ je nejkratší možná cesta filmového vozu $\leftarrow\downarrow\rightarrow\rightarrow\uparrow\leftarrow$.

16-4-4 Dlouhoprstova zapeklitá hra 10 bodů

Dlouhoprstova hrabivost byla sice jeho vítězstvím nad piráty na chvilku utlumena, ale jeho sebevědomí poněkud stouplo. A tak si řekl, že když už je jednou na cestě do pekla, aby upsal ďáblu duši za pár stříbrňáků, tak tam dojde.

V pekle byl Dlouhoprst Satanem pěkně uvítán, ale návrh, že upíše svou duši, mu neprošel. Satan tvrdil, že Dlouhoprstova duše skončí tak jako tak v pekle, takže proč by mu za to měl ještě platit. Nicméně souhlasil, že si s Dlouhoprstem zahraje karetní hru.

Hraje se na jedné straně o Dlouhoprstův život, o truhlu stříbrňáků na straně druhé. Hry prší se samozřejmě v horoucím pekle bojí jako kříže a ani s mariášem Satan nesouhlasil. Viděl totiž Dlouhoprstovy rukávy nadité falešnými kartami. A tak Satan navrhl hru vlastní.

Hraje se se speciálními pekelnými kartami, každá karta je označená buď malým nebo velkým písmenem nebo číslicí. Satan před sebe položí dvě řady karet, obě v nějakém konkrétním pořadí. V jedné řadě je karet N, v druhé M. V každé řadě otočí nějaké karty vzhůru rubem a nějaké lícem. Každou řadu tedy můžeme zaznamenat jako posloupnost malých a velkých písmen a číslic (to jsou konkrétní karty), otazníků (práve jedna libovolná karta) a hvězdiček (0 a více libovolných karet).

Protože Satan ví o Dlouhoprstových falešných kartách, dostal Dlouhoprst za úkol vytáhnout z útrob svého obleku takovou *nejkratší* posloupnost karet, že může být stejná jako obě Satanovy řady karet. Případně má říci, že to není možné. Pokud je nejkratších posloupností více, stačí libovolná z nich.

Příklad: Pokud jsou Satanovy řady karet `t?2*f?` a `t*fg`, Dlouhoprst má vytáhnout například karty `tX2fg`.

16-4-5 Obchodníci s deštěm — 10 bodů

Pro svou práci na novém operačním systému potřebujete spolehlivý generátor pseudonáhodných čísel. A to takový, který bude generovat nelokální posloupnosti náhodných čísel. To jsou takové posloupnosti, jejichž členy jsou rozptýlené na celém používaném intervalu. Jinak řečeno, nejmenší vzdálenost mezi dvěma libovolnými prvky je pokud možno co největší.

Bohužel jediná firma, která generátory tohoto typu nabízí, je O. Š. Kubal a synové. Tu musíte určitě znát. Je to jediná společnost, která dokázala prodat tisíc kusů ledniček značky „Mrazík" Eskymákům, patentovala si v USA perpetuum mobile a dokázala si prodat vlastní nos mezi svýma očima.

Ale protože znáte O. Š. Kubala i jeho syny a chcete, aby generátor doopravdy fungoval, rozhodli jste se ho nejprve vyzkoušet. A k tomuto účelu si potřebujete napsat následující program.

Na vstupu dostanete N a K. Pak budete postupně načítat N různých náhodných čísel. *Hned* po načtení jednoho náhodného čísla (kromě prvního) vypíšete, jaký je nejmenší rozdíl mezi libovolnými různými dvěma z posledních K načtených náhodných čísel.

Příklad: Pro $N = 6$, $K = 3$ má vypadat vstup a výstup programu následovně:

náhodné číslo	*aktuální nejmenší rozdíl*
5	
7	2
4	1
15	3
6	2
20	5

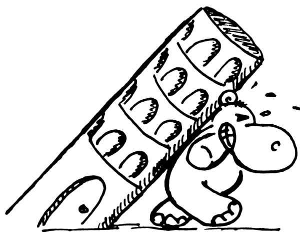
16-4-6 Šikmá věž v Kocourkově — 10 bodů

V Kocourkově se radní buřtipáni rozhodli, že si postaví šikmou věž. Všichni s jejich návrhem souhlasili a všichni chtěli hned pomoci. Silák se hned nabídl, že vykopá do země díru, aby byla věž šikmá, Bohemína tvrdila, že z hromady vajec dokáže udělat skvělou maltu, Zpívarotti slíbil, že za trochu piva zazpívá a všem půjde práce pěkně od ruky.

Brzy (po několika postavených věžích) ale zjistili, že to nebude tak jednoduché. Některé úkoly bylo totiž nutné vyřešit před tím, než jiné začnou. Silák bohužel nedokázal pod již postavenou věží vykopat dost velkou díru, aby věž byla šikmá. A i když Lamželezo byl silák a s velkým rozběhem zvládl vybourat dveře v již postavené zdi, postavit zeď i s dveřmi bylo přeci jen lepší.

Dále zjistili, že nějaké práce jsou *klíčové*. Zpoždění takových klíčových prací totiž vede ke zpoždění celé stavby. A protože chtějí mít Kocourkovští věž postavenou co nejdříve, poslali si pro Vás, abyste jim pomohli.

Váš program dostane na vstupu N, což je počet pracovních úkonů, které je třeba k postavení věže vykonat. U každého úkonu víte dobu, jakou trvá, a dále seznam jiných úkonů, které musí být dokončeny před tím, než začne práce na tomto úkonu. Výstupem programu by mělo být buď slovo „nelze“, pokud věž postavit vůbec nejde (například Zpívarotti zpívá jen za pivo a hospodský chce dát pivo jen za doušek pěkného zpěvu). Pokud věž postavit jde, měl by Váš program vypsat nejmenší dobu, za jakou mohou Kocourkovští věž postavit, a ty úkony, které jsou klíčové.



Příklad: $N = 5$, úkon 1 trvá 4 časové jednotky, úkon 2 trvá 2 časové jednotky, úkon 3 trvá 5 časových jednotek, úkon 4 trvá 10 časových jednotek a úkon 5 trvá 2 časové jednotky. Úkon 2 je možné začít, až když jsou hotové úkony 1 a 3. Úkon 4 může začít až po skončení úkonů 5 a 2.

Tehdy lze věž postavit za 17 časových jednotek, klíčové jsou úkony 2, 3 a 4.

16-4-K Recepty z programátorské kuchařky — MM a Tomáš Valla

V nedávném vydání programátorské kuchařky jsme se zabývali tříděním dat, tentokráte si povíme, jak v uspořádaných datech něco efektivně najít a jak si data udržovat stále uspořádaná. K tomu se nám bude hodit zejména binární vyhledávání a různé druhy vyhledávacích stromů.

Binární vyhledávání

Představte si, že jste k narozeninám dostali obrovské pole setříděných záznamů (to je, pravda, trochu netradiční dárek, ale proč ne – může to být třeba telefonní seznam). Záznamy mohou vypadat libovolně a to, že jsou setříděné, znamená jen a pouze, že $x_1 < x_2 < \ldots < x_N$, kde $<$ je nějaká relace, která nám řekne, který ze dvou záznamů je menší (pro jednoduchost předpokládáme, že žádné dva záznamy nejsou stejné).

Co si ale s takovým polem počneme? Zkusíme si v něm najít nějaký konkrétní záznam z. To můžeme udělat třeba tak, že si nalistujeme prostřední záznam (označíme si ho x_m) a porovnáme s ním naše z. Pokud $z < x_m$, víme, že se z nemůže vyskytovat „napravo" od x_m, protože tam jsou všechny záznamy větší než x_m a tím spíše než z. Analogicky pokud $z > x_m$, z se nemůže vyskytovat v první polovině pole. V obou případech nám zbude jedna polovina a v ní budeme pokračovat stejným způsobem. Tak budeme postupně zmenšovat interval, ve kterém se z může nacházet, až buďto z najdeme nebo vyloučíme všechny prvky, kde by mohlo být.

Tomuto principu se obvykle říká *binární vyhledávání* nebo také *hledání půlením intervalu* a snadno ho naprogramujeme buďto rekursivně nebo pomocí cyklu, v němž si budeme udržovat interval $\langle l, r\rangle$, ve kterém se hledaný prvek může nacházet:

```
function BinSearch(z : integer):integer;
var l,r,m : integer;
begin
  l := 1;  { interval, ve kterém hledáme }
  r := N;
  while l <= r do begin { ještě není prázdný }
    m := (l+r) div 2;  { střed intervalu }
    if z < x[m] then
      r := m-1       { je vlevo }
    else if z > x[m] then
      l := m+1       { je vpravo }
    else begin       { Bingo! }
      hledej := m;
      exit;
    end;
  end;
  hledej := -1;      { nebyl nikde }
end;
```

Všimněte si, že průchodů cyklem `while` může být nejvýše $\lceil \log_2 N \rceil$, protože interval $\langle l, r\rangle$ na počátku obsahuje N prvků a v každém průchodu jej zmenší-

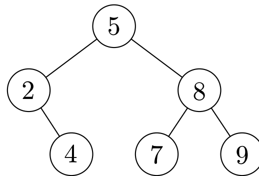
me na polovinu (ve skutečnosti ještě o jedničku, ale tím lépe pro nás), takže po k průchodech bude interval obsahovat nejvýše $N/2^k$ prvků. Jenže pokud by k bylo větší než $\log_2 N$, bude $N/2^k < 1$ a tehdy se algoritmus určitě musí zastavit. Časová složitost binárního vyhledávání tedy bude $O(\log N)$. [Základ logaritmu nemusíme psát, protože $\log_a b = \log_c b / \log_c a$, čili logaritmy o různých základech se liší jen konstantou, která se „schová do O-čka."]

Hledání půlením intervalu je tedy velmi rychlé, pokud máme možnost si data předem setřídit. Jakmile ale potřebujeme za běhu programu přidávat a odebírat záznamy, potážeme se se zlou: buďto budeme mít záznamy uložené v poli, a pak nezbývá než při zatřiďování nového prvku ostatní „rozhrnout", což může trvat až N kroků, a nebo si je budeme udržovat v nějakém seznamu, do kterého dokážeme přidávat v konstantním čase, jenže pak pro změnu nebudeme při vyhledávání schopni najít tolik potřebnou polovinu.

Zkusme ale provést jednoduchý myšlenkový pokus:

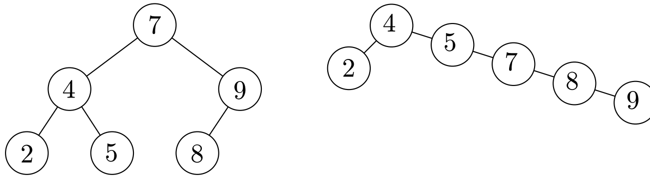
Vyhledávací stromy

Představme si, jakými všemi možnými cestami se může v našem poli binární vyhledávání ubírat. Na začátku porovnáváme s prostředním prvkem a podle výsledku se vydáme jednou ze dvou možných cest (nebo rovnou zjistíme, že se jedná o hledaný prvek, ale to není moc zajímavý případ). Na každé cestě nás zase čeká porovnání se středem příslušného intervalu a to nás opět pošle jednou ze dvou dalších cest atd. To si můžeme přehledně popsat pomocí stromu:



Jeden vrchol stromu prohlásíme za kořen a ten bude odpovídat celému poli (a jeho prostřednímu prvku). K němu budou připojené vrcholy obou polovin pole (opět obsahující příslušné prostřední prvky) a tak dále. Ovšem jakmile známe tento strom, můžeme náš půlící algoritmus provádět přímo podle stromu (ani k tomu nepotřebujeme vidět původní pole a umět v něm hledat poloviny): začneme v kořeni, porovnáme a podle výsledku se buďto přesuneme do levého nebo pravého podstromu a tak dále. Každý průběh algoritmu bude tedy odpovídat nějaké cestě z kořene stromu do hledaného vrcholu z.

Teď si ale všimněte, že aby hledání hodnoty podle stromu fungovalo, strom vůbec nemusel vzniknout přesně půlením intervalu – stačilo, aby v každém vrcholu platilo, že všechny hodnoty v levém podstromu jsou menší než tento vrchol a naopak hodnoty v pravém podstromu větší. Hledání v témže poli by tedy popisovaly následující stromy (např.):



Hledací algoritmus podle jiných stromů samozřejmě už nemusí mít pěknou logaritmickou složitost (když bychom hledali podle „degenerovaného“ stromu z pravého obrázku, trvalo by to dokonce lineárně). Důležité ale je, že takovéto stromy se dají poměrně snadno modifikovat a že je při troše šikovnosti můžeme udržet dostatečně podobné ideálnímu půlení intervalu. Pak bude hloubka stromu stále $O(\log N)$ (a tedy i časová složitost hledání a, jak za chvilku uvidíme, i dalších operací).

Definice

Zkusme si tedy pořádně nadefinovat to, co jsme právě vymysleli:

Binární vyhledávací strom (po domácku BVS) je buďto prázdná množina nebo *kořen* obsahující jednu hodnotu a mající dva *podstromy* (levý a pravý), což jsou opět BVS, ovšem takové, že všechny hodnoty uložené v levém podstromu jsou menší než hodnota v kořeni, a ta je naopak menší než všechny hodnoty uložené v pravém podstromu.

Úmluva: Pokud x je kořen a L_x a R_x jeho levý a pravý podstrom, pak kořenům těchto podstromů (pokud nejsou prázdné) budeme říkat *levý a pravý syn* vrcholu x a naopak vrcholu x budeme říkat *otec* těchto synů. Pokud je některý z podstromů prázdný, pak vrchol x příslušného syna nemá. Vrcholu, který nemá žádné syny, budeme říkat *list* vyhledávacího stromu. Všimněte si, že pokud x má jen jediného syna, musíme stále rozlišovat, je-li to syn levý nebo pravý, protože potřebujeme udržet správné uspořádání hodnot. Také si všimněte, že pokud známe syny každého vrcholu, můžeme již rekonstruovat všechny podstromy.

Každý BVS také můžeme popsat velmi jednoduchou strukturou v paměti:

```
type pvrchol = ^vrchol;
        vrchol = record
           l, r : pvrchol;  { levý a pravý syn }
           x    : integer;  { hodnota }
        end;
```

Pokud některý ze synů neexistuje, zapíšeme do příslušné položky hodnotu *nil*.

Find

V řeči BVS můžeme přeformulovat náš vyhledávací algoritmus takto:

```
function TreeFind(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu. Vrátí vrchol,
  kde se hodnota nachází, nebo nil, není-li. }
begin
  while (v<>nil) and (v^.x<>x) do begin
    if x<v^.x then
      v := v^.l
    else
      v := v^.r
  end;
  TreeFind := v;
end;
```

Funkce `TreeFind` bude pracovat v čase $O(h)$, kde h je hloubka stromu, protože začíná v kořeni a v každém průchodu cyklem postoupí o jednu hladinu níže.

Insert

Co kdybychom chtěli do stromu vložit novou hodnotu (aniž bychom se teď starali o to, zda tím strom nemůže degenerovat)? Stačí zkusit hodnotu najít a pokud tam ještě nebyla, určitě při hledání narazíme na odbočku, která je *nil*. A přesně na toto místo připojíme nově vytvořený vrchol, aby byl správně uspořádán vzhledem k ostatním vrcholům (že tomu tak je, plyne z toho, že při hledání jsme postupně vyloučili všechna ostatní místa, kde nová hodnota být nemohla). Naprogramujeme opět snadno, tentokráte si ukážeme rekurzivní zacházení se stromy:

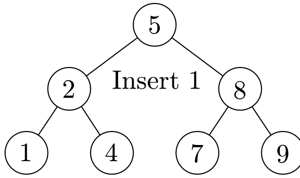
```
function TreeIns(v:pvrchol; x:integer):pvrchol;
{ Dostane kořen stromu a hodnotu ke vložení,
  vrátí nový kořen. }
begin
  if v=nil then begin
  { prázdný strom => založíme nový kořen }
    new(v);
    v^.l := nil;
    v^.r := nil;
    v^.x := x;
  end else if x<v^.x then  { vkládáme vlevo }
    v^.l := TreeIns(v^.l, x)
  else if x>v^.x then     { vkládáme vpravo }
    v^.r := TreeIns(v^.r, x);
  TreeIns := v;
end;
```

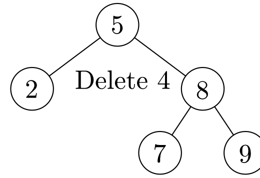
Delete

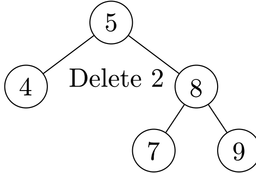
Mazání bude o kousíček pracnější, musíme totiž rozlišit tři případy: Pokud je mazaný vrchol list, stačí ho vyměnit za *nil*. Pokud má právě jednoho syna, stačí náš vrchol v ze stromu odstranit a syna přepojit k otci v. A pokud má syny dva, najdeme největší hodnotu v levém podstromu (tu najdeme tak, že půjdeme jednou doleva a pak pořád doprava), umístíme ji do stromu namísto mazaného vrcholu a v levém podstromu ji pak smažeme (což už umíme, protože má 1 nebo 0 synů). Program následuje:

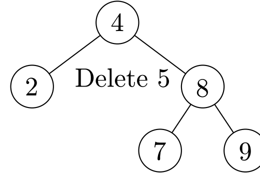
```
function TreeDel(v:pvrchol; x:integer):pvrchol;
{ Parametry stejně jako TreeIns }
var w:pvrchol;
begin
  TreeDel := v;
  if v=nil then exit        { prázdný strom }
  else if x<v^.x then
    v^.l := TreeDel(v^.l, x) { ještě hledáme x }
  else if x>v^.x then
    v^.r := TreeDel(v^.r, x)
  else begin             { našli jsme }
    if (v^.l=nil) and (v^.r=nil) then begin
      TreeDel := nil;      { mažeme list }
      dispose(v);
    end else if v^.l=nil then begin
      TreeDel := v^.r;     { jen pravý syn }
      dispose(v);
    end else if v^.r=nil then begin
      TreeDel := v^.l;     { jen levý }
      dispose(v);
    end else begin         { má oba syny }
      w := v^.l;           { hledáme max(L) }
      while w^.r<>nil do w := w^.r;
      v^.x := w^.x;         { prohazujeme }
   { a mažeme původní max(L) }
      v^.l := TreeDel(v^.l, w^.x);
    end;
  end;
end;
```

Když do stromu z našeho prvního obrázku zkusíme přidávat nebo z něj odebírat prvky, dopadne to takto:









Jak vkládání, tak mazání opět budou trvat $O(h)$. Ale pozor, jejich používáním může h nekontrolovatelně růst – sami zkuste najít nějaký příklad, kdy h dosáhne až N.

Procházení stromu

Pokud bychom chtěli všechny hodnoty ve stromu vypsat, stačí strom rekursivně projít a sama definice uspořádání hodnot ve stromu nám zajistí, že hodnoty vypíšeme ve vzestupném pořadí: nejdříve levý podstrom, pak kořen a pak podstrom pravý. Časová složitost je, jak se snadno nahlédne, lineární, protože strávíme konstantní čas vypisováním každého prvku a těch je právě N. Program opět snadný:

```
procedure TreeShow(v:pvrchol);
begin
   if v=nil then exit;  { není co dělat }
   TreeShow(v^.l);
   writeln(v^.x);
   TreeShow(v^.r);
end;
```

Vyvážené stromy

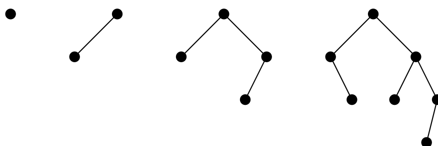
S binárními stromy lze dělat všelijaká kouzla. Ale prakticky všechny stromové algoritmy mají společné to, že jejich časová složitost je lineární v hloubce stromu. (Pravda, právě ten poslední byl výjimka, ale všechny prvky opravdu rychleji než lineárně s N nevypíšeme.) Jenže jak jsme viděli, neopatrným insertováním a deletováním prvků mohou snadno vznikat všelijaké degenerované stromy, které mají lineární hloubku. Abychom tomu zabránili, musíme stromy *vyvažovat*. To znamená definovat si nějaké šikovné omezení na tvar stromu, aby

hloubka byla vždy $O(\log N)$. Možností je mnoho, my uvedeme jen ty nejdůležitější:

Dokonale vyvážený budeme říkat takovému stromu, ve kterém pro každý vrchol platí, že počet vrcholů v jeho levém a pravém podstromu se liší nejvýše o jedničku. Takové stromy kopírují dělení na poloviny při binárním vyhledávání, a proto (jak jsme již dokázali) mají vždy logaritmickou hloubku. Jediné, čím se liší, je, že mohou zaokrouhlovat na obě strany, zatímco náš půlící algoritmus zaokrouhloval polovinu vždy dolů, takže levý podstrom nemohl být nikdy větší než pravý. Z toho také plyne, že se snadnou modifikací půlícího algoritmu dá dokonale vyvážený BVS v lineárním čase vytvořit ze setříděného pole. Bohužel se ale při Insertu a Deletu nedá v logaritmickém čase strom znovu vyvážit.

AVL stromy

Zkusíme tedy vyvažovací podmínku trochu uvolnit a vyžadovat pouze, aby se u každého vrcholu lišily o jedničku nikoliv velikosti podstromů, nýbrž jejich hloubky. Takovým stromům se říká *AVL stromy* a mohou vypadat třeba takto:



Každý dokonale vyvážený strom je také AVL stromem, ale jak je vidět na předchozím obrázku, opačně to platit nemusí. To, že hloubka AVL stromů je také logaritmická, proto není úplně zřejmé a zaslouží si to trochu dokazování:

Věta: AVL strom o N vrcholech má hloubku $O(\log N)$.

Důkaz: Označme A_d nejmenší možný počet vrcholů, jaký může být v AVL stromu hloubky d. Snadno vyzkoušíme, že $A_1 = 1$, $A_2 = 2$, $A_3 = 4$ a $A_4 = 7$ (příslušné minimální stromy najdete na předchozím obrázku). Navíc platí, že $A_d = 1 + A_{d-1} + A_{d-2}$, protože každý minimální strom hloubky d musí mít kořen a 2 podstromy, které budou opět minimální, protože jinak bychom je mohli vyměnit za minimální a tím snížit počet vrcholů celého stromu. Navíc alespoň jeden z podstromů musí mít hloubku $d-1$ (protože jinak by hloubka celého stromu nebyla d) a druhý hloubku $d-2$ (podle definice AVL stromu může mít $d-1$ nebo $d-2$, ale s menší hloubkou bude mít evidentně méně vrcholů).

Spočítat, kolik přesně je A_d, není úplně snadné, ale nám bude stačit dokázat, že $A_d \geq 2^{n/2}$. To provedeme indukcí: Pro $d < 4$ to plyne z ručně spočítaných hodnot. Pro $d \geq 4$ je pak $A_d = 1 + A_{d-1} + A_{d-2} > 2^{(d-1)/2} + 2^{(d-2)/2} = 2^{d/2} \cdot (2^{-1/2} + 2^{-1}) > 2^{d/2}$ (součet čísel v závorce je ≈ 1.207).

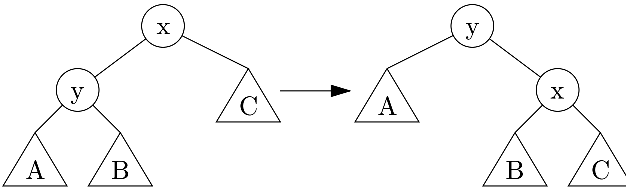
Jakmile ale už víme, že A_d roste s d alespoň exponenciálně, tedy že $\exists c : A_d \geq c^d$, důkaz je u konce: Máme-li AVL strom T na N vrcholech, najdeme

si nejmenší d takové, že $A_d \leq N$. Hloubka stromu T může být maximálně d, protože jinak by T musel mít alespoň A_{d+1} vrcholů, ale to je více než N. A jelikož A_d rostou exponenciálně, je $d \leq \log_c N$, čili $d = O(\log N)$. *Q.E.D.*

AVL stromy tedy vypadají nadějně, ale ještě stále nevíme, jak provádět Insert a Delete tak, aby AVL vyváženost zachovávaly. Nemůžeme si totiž dovolit strukturu stromu měnit libovolně – stále musíme dodržovat správné uspořádání hodnot. K tomu se nám bude hodit zavést si nějakou množinu operací, o kterých dokážeme, že jsou korektní, a pak strukturu stromu měnit vždy jen pomocí těchto operací. Budou to:

Rotace

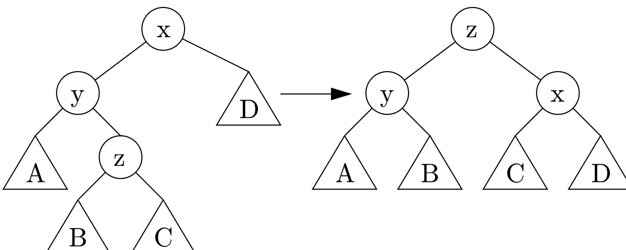
Rotací binárního stromu (respektive nějakého podstromu) nazveme jeho „překořenění“ za některého ze synů kořene. Místo formální definice ukažme raději obrázek:



Strom jsme překořenili za vrchol y a přepojili jednotlivé podstromy tak, aby byly vzhledem k x a y opět uspořádané správně (všimněte si, že je jen jediný způsob, jak to udělat). Jelikož se tím okolí vrcholu y „otočilo“ po směru hodinových ručiček, říká se takové operaci *rotace doprava*. Inverzní operaci (tj. překořenění za pravého syna kořene) se říká *rotace doleva* a na našem obrázku odpovídá přechodu zprava doleva.

Dvojrotace

Také si nakreslíme, jak to dopadne, když provedeme dvě rotace nad sebou lišící se směrem (tj. jednu levou a jednu pravou nebo opačně). Tomu se říká *dvojrotace* a jejím výsledkem je překořenění podstromu za vnuka kořene připojeného „cikcak“. Raději opět předvedeme na obrázku:



Znaménka

Při vyvažování se nám bude hodit pamatovat si u každého vrcholu, v jakém vztahu jsou hloubky jeho podstromů. Tomu budeme říkat *znaménko* vrcholu a bude buďto 0, jsou-li oba stejně hluboké, $-$ pro levý podstrom hlubší a $+$ pro pravý hlubší. V textu budeme znaménka, respektive vrcholy se znaménky značit $\oplus$, $\ominus$ a $\odot$.

Pokud celý strom zrcadlově obrátíme (prohodíme levou a pravou stranu), znaménka se změní na opačná ($\oplus$ a $\ominus$ se prohodí, 0 zůstane). Toho budeme často využívat a ze dvou zrcadlově symetrických situací popíšeme jenom jednu s tím, že druhá se v algoritmu zpracuje symetricky.

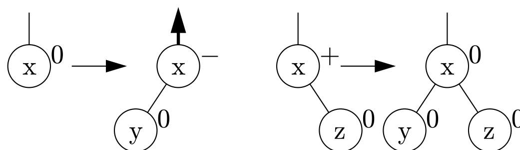
Často také budeme potřebovat nalézt otce nějakého vrcholu. To můžeme zařídit buďto tak, že si do záznamů popisujících vrcholy stromu přidáme ještě ukazatele na otce a budeme ho ve všech operacích poctivě aktualizovat, a nebo využijeme toho, že jsme do daného vrcholu museli někudy přijít z kořene, celou cestu z kořene si zapamatujeme v nějakém zásobníku a postupně se budeme vracet.

Tím jsme si připravili všechny potřebné ingredience, tož s chutí do toho:

Vyvažování po Insertu

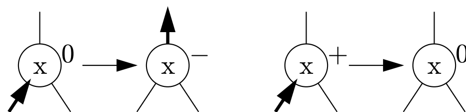
Když provedeme Insert tak, jak jsme ho popisovali u obecných vyhledávacích stromů, přibude nám ve stromu list. Pokud se tím AVL vyváženost neporušila, stačí pouze opravit znaménka na cestě z nového listu do kořene (všude jinde zůstala zachována). Pakliže porušila, musíme s tím něco provést, konkrétně ji šikovně zvolenými rotacemi opět opravit. Popíšeme algoritmus, který bude postupovat od listu ke kořeni a vše potřebné zařídí:

Nejprve přidání listu samotné:

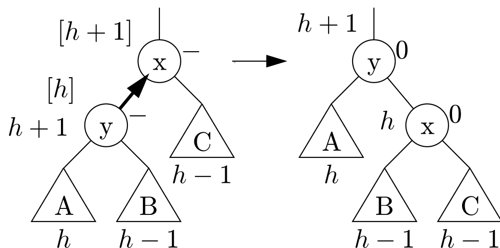


Pokud jsme přidali list (bez újmy na obecnosti levý, jinak vyřešíme zrcadlově) vrcholu se znaménkem $\odot$, změníme znaménko na $\ominus$ a pošleme o patro výš informaci o tom, že hloubka podstromu se zvýšila (to budeme značit šipkou). Přidali-li jsme list k $\oplus$, změní se na $\odot$ a hloubka podstromu se nemění, takže můžeme skončit.

Nyní rozebereme případy, které mohou nastat na vyšších hladinách, když nám z nějakého podstromu přijde šipka. Opět budeme předpokládat, že přišla zleva; pokud zprava, vyřešíme zrcadlově. Pokud přišla do $\oplus$ nebo $\odot$, ošetříme to stejně jako při přidání listu:

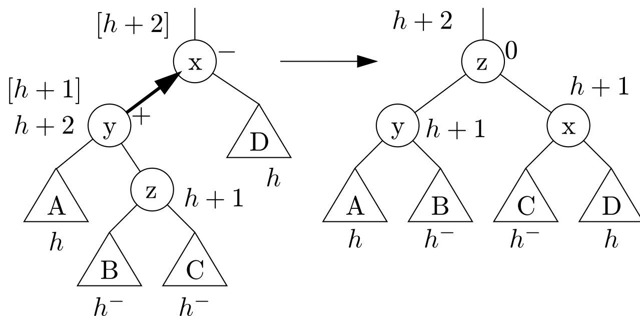


Pokud ale vrchol x má znaménko $\ominus$, nastanou potíže: levý podstrom má teď hloubku o 2 vyšší než pravý, takže musíme rotovat. Proto se podíváme o patro níž, jaké je znaménko vrcholu y pod šipkou, abychom věděli, jakou rotaci provést. Jedna možnost je tato (y je $\ominus$):



Tehdy provedeme jednoduchou rotaci vpravo. Jak to dopadne s hloubkami jsme přikreslili do obrázku – pokud si hloubku podstromu A označíme jako h, B musí mít hloubku $h-1$, protože y je $\ominus$, atd. Jen nesmíme zapomenout, že v x jsme ještě $\ominus$ nepřepočítali (vede tam přeci šipka), takže ve skutečnosti je jeho levý podstrom o 2 hladiny hlubší než pravý (původní hloubky jsme na obrázku naznačili v [závorkách]). Po zrotování vyjdou u x i y znaménka $\odot$ a celková hloubka se nezmění, takže jsme hotovi.

Další možnost je y jako $\oplus$:

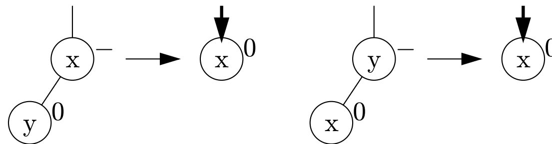


Tehdy se podíváme ještě o hladinu níž a provedeme dvojrotaci. (Nemůže se nám stát, že by z neexistovalo, protože jinak by v y nebylo $\oplus$.) Hloubky opět najdete na obrázku. Jelikož z může mít libovolné znaménko, jsou hloubky podstromů B a C buďto h nebo $h-1$, což značíme h^-. Podle toho pak vyjdou znaménka vrcholů x a y po rotaci. Každopádně vrchol z vždy obdrží $\odot$ a celková hloubka se nemění, takže končíme.

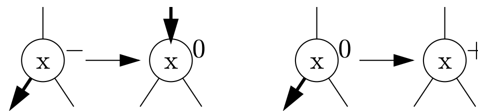
Poslední možnost je, že by y byl ⊙, ale tu vyřešíme velmi snadno: všimneme si, že nemůže nastat. Kdykoliv totiž posíláme šipku nahoru, není pod ní ⊙. (Kontrolní otázka: jak to, že ⊕ může nastat?)

Vyvažování po Deletu

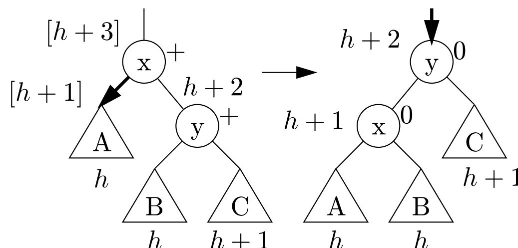
Vyvažování po Deletu je trochu obtížnější, ale také se dá popsat pár obrázky. Nejdříve opět rozebereme základní situace: odebíráme list (BÚNO levý) nebo vnitřní vrchol stupně 2 (tehdy ale musí být jeho jediný syn listem, jinak by to nebyl AVL strom):



Šipkou dolů značíme, že o patro výš posíláme informaci o tom, že se hloubka podstromu snížila o 1. Pokud šipku dostane vrchol typu ⊖ nebo ⊙, vyřešíme to snadno:

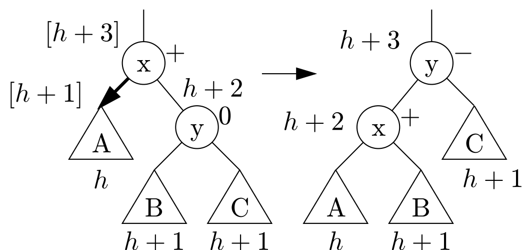


Problematické jsou tentokráte ty případy, kdy šipku dostane ⊕. Tehdy se musíme podívat na znaménko *opačného* syna a podle toho rotovat. První možnost je, že opačný syn má ⊕:



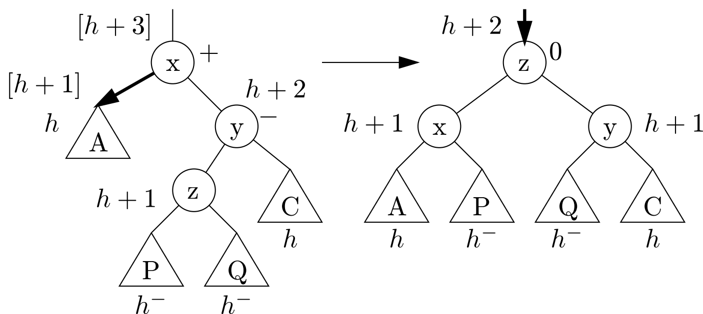
Tehdy provedeme rotaci vlevo, x i y získají nuly, ale celková hloubka stromu se sníží o hladinu, takže nezbývá, než poslat šipku o patro výš.

Pokud by y byl $\odot$:



Opět rotace vlevo, ale tentokráte se zastavíme, protože celková hloubka se nezměnila.

Poslední, nejkomplikovanější možnost je, že by y byl $\ominus$:



V tomto případě provedeme dvojrotaci (z určitě existuje, jelikož y je typu $\ominus$), vrcholy x a y obdrží znaménka v závislosti na původním znaménku vrcholu z a celý strom se snížil, takže pokračujeme o patro výš.

Happy end

Jak při Insertu, tak při Deletu se nám podařilo strom upravit tak, aby byl opět AVL stromem, a trvalo nám to lineárně s hloubkou stromu (konáme konstantní práci na každé hladině), čili stejně jako trvá Insert a Delete samotný. Jenže o AVL stromech jsme již dokázali, že mají hloubku vždy logaritmickou, takže jak hledání, tak Insert a Delete zvládneme v logaritmickém čase (vzhledem k aktuálnímu počtu prvků ve stromu).

Další typy stromů

AVL stromy samozřejmě nejsou jediný způsob, jak zavést stromovou datovou strukturu s logaritmicky rychlými operacemi. Další jsou třeba:

- *Červeno-Černé stromy* – ty si místo znamének vrcholy barví, každý je buďto červený nebo černý a platí, že nikdy nejsou dva červené vrcholy

pod sebou a že na každé cestě z kořene do listu je stejný počet černých vrcholů. Opět je hloubka stromu logaritmická, po Insertu a Deletu barvy opravujeme přebarvováním na cestě do kořene a rotováním, ovšem je potřeba rozebrat podstatně více případů než u AVL stromů. (Za to jsme ale odměněni tím, že nikdy neděláme více než 2 rotace.)

- *2-3-stromy* – v jednom vrcholu nemáme uloženu jednu hodnotu, nýbrž jednu nebo dvě (a synové jsou pak 2 nebo 3, odtud název). Hloubka opět logaritmická, vyvažování řešíme pomocí spojování a rozdělování vrcholů.
- *Splay stromy* – nezavádíme žádnou vyvažovací podmínku, nýbrž definujeme, že kdykoliv pracujeme s nějakým vrcholem, vždy si jej vyrotujeme do kořene a pokud to jde, preferujeme dvojrotace. Takové operaci se říká Splay a dají se pomocí ní definovat operace ostatní: Find hodnotu najde a poté na ni zavolá Splay. Insert si nechá vysplayovat předchůdce nové hodnoty a vloží nový vrchol mezi předchůdce a jeho pravého syna. Delete vysplayuje mazaný prvek, pak uvnitř pravého podstromu vysplayuje minimum, takže bude mít jen jednoho syna a můžeme jím tedy nahradit mazaný prvek v kořeni. Jednotlivé operace samozřejmě mohou trvat až lineárně dlouho, ale dá se o nich dokázat, že jejich *amortizovaná* složitost (co to je, najdete v kuchařce k předchozí sérii) je vždy $O(\log N)$. To u většiny použití stačí (datovou strukturu obvykle používáte uvnitř nějakého algoritmu a zajímá vás, jak dlouho běží všechny operace dohromady) a navíc je Splay stromy daleko snazší naprogramovat než nějaké vyvažované stromy. [Mimo to mají i jiné krásné vlastnosti: přizpůsobují svůj tvar četnostem hledání, takže často hledané prvky jsou pak blíž ke kořeni, snadno se dají rozdělovat a spojovat atd.]
- *Treapy* – randomizovaně vyvažované stromy: něco mezi stromem (tree) a haldou (heap). Každému prvku přiřadíme *váhu,* což je náhodné číslo z intervalu $\langle 0, 1\rangle$. Strom pak udržujeme uspořádaný stromově podle hodnot a haldově podle vah (všimněte si, že tím je jeho tvar určen jednoznačně). Insert a Delete opravují haldové uspořádání velmi jednoduše pomocí rotací. Časová složitost v průměrném případě je $O(\log N)$.
- *BB-α stromy* – zobecnění dokonalé vyváženosti jiným směrem: zvolíme si vhodné číslo α a vyžadujeme, aby se velikost podstromů každého vrcholu lišila maximálně α-krát (prázdné podstromy nějak ošetříme, abychom nedělili nulou; dokonalá vyváženost odpovídá $\alpha = 1$ [až na zaokrouhlování]). V každém vrcholu si budeme pamatovat, kolik vrcholů obsahuje podstrom, jehož je kořenem, a po Insertu a Deletu přepočítáme tyto hodnoty na cestě zpět do kořene a zkontrolujeme,

jestli je strom ještě stále α-vyvážený. Pokud ne, najdeme nejvyšší místo, ve kterém se velikosti podstromů příliš liší, a vše od tohoto místa dolů znovu vytvoříme algoritmem na výrobu dokonale vyvážených stromů. Ten, pravda, běží v lineárním čase, ale čím větší podstrom přebudováváme, tím to děláme méně často, takže vyjde opět amortizovaně $O(\log N)$ na operaci.

Cvičení

Několik věcí, které se do kuchařky už nevešly, ale můžete si je zkusit vymyslet:

1. jak konstruovat dokonale vyvážené stromy
2. jak pomocí toho naprogramovat BB-α stromy
3. algoritmus, který k prvku ve stromu najde jeho následníka, což je prvek s nejbližší vyšší hodnotou (zde předpokládejte, že ke každému prvku máte uložený ukazatel na jeho otce)
4. jak vypsat celý strom tak, že začnete v minimu a budete postupně hledat následníky (i když nalezení následníka může trvat až $O(h)$, všimněte si, že projití celého stromu přes následníky bude lineární)
5. jak do vrcholů stromu ukládat různé pomocné informace, jako třeba počet vrcholů v podstromu každého vrcholu, a jak tyto informace při operacích se stromem udržovat (při Insertu, Deletu, rotaci)
6. že libovolný interval $\langle a, b\rangle$ lze rozložit na logaritmicky mnoho intervalů odpovídajících podstromům
7. a že zkombinováním předchozích dvou cvičení lze odpovídat i na otázky typu „kolik si strom pamatuje hodnot ze zadaného intervalu" v logaritmickém čase...

Několik poznámek na závěr

- Pokud záznamy můžeme jenom porovnávat, je binární vyhledávání nejlepší možné. Je totiž vždy popsáno nějakým binárním stromem a binární strom s N vrcholy musí mít vždy hloubku alespoň $\lfloor \log N \rfloor$.
- Pokud bychom ale předpokládali, že se záznamy můžeme zacházet i jinak, dají se některé operace provádět i v konstantním čase (alespoň průměrně). K tomu se hodí například *hashování,* a to si popíšeme v některé z kuchařek v příštím ročníku KSP. Jeho nevýhodou ovšem je, že udržuje jenom množinu prvků, nikoliv uspořádání na ní, takže například nelze najít k zadanému prvku nejbližší vyšší.
- Pokud bychom připustili, že se mohou vyskytnout dva stejné záznamy, bude vše také fungovat, jen si musíme dát o něco větší pozor na to, co všechno při operacích se stromem může nastat.

- Jakpak přišly AVL stromy ke svému jménu? Inu, podle svých objevitelů pánů Adeľsona-Velského a Landise.
- Rekurenci $A_d = 1 + A_{d-1} + A_{d-2}$, $A_1 = 1$, $A_2 = 2$ pro velikosti minimálních AVL stromů je samozřejmě možné vyřešit i přesně. Žádné překvapení se nekoná, objeví se totiž stará známá Fibonacciho čísla: $A_n = F_{n+2} - 1$.

Vzorová řešení

16-1-1 Telefonní seznam — Zdeněk Dvořák

Jak většina řešitelů správně poznala, tato úložka se dá řešit použitím haldy. Snadno si rozmyslíme, že když „přefiltrujeme“ zadanou posloupnost slov haldou velikosti k – tj. do haldy vložíme prvních k prvků a pak střídavě odebíráme minimum a přidáváme další prvky – je výsledná posloupnost setříděná. Přidání do haldy i odebrání z ní vyžaduje $O(\log k)$ porovnání a těchto operací provedeme $O(n)$. Je ovšem nutné vzít v úvahu také to, že porovnání řetězců nelze provést v konstantním čase. V nejhorším případě se může stát, že se dva řetězce liší až na některé z posledních pozic, pak jejich porovnání zabere čas $\Theta(L)$, kde L je jejich délka. Výsledná časová složitost tohoto řešení je tedy $O(n \cdot L \cdot \log k)$. V paměti si stačí udržovat pouze haldu, prvky se načítají postupně a lze je rovnou vypisovat, tedy paměťová složitost je $O(k \cdot L)$.

Existují ovšem nejméně dvě řešení dosahující stejné nebo lepší časové složitosti:

Je možné prostě zapomenout na k a setřídit slova pomocí přihrádkového třídění. Toto řešení pracuje s časovou složitostí $O((n + p) \cdot L)$, kde p je počet písmen v abecedě, a paměťovou složitostí $O(n \cdot L)$.

Další řešení pracuje na podobném principu jako první řešení, nicméně vyhýbá se použití haldy. Funguje takto: z posloupnosti slov si vezmeme prvních $2k$ a setřídíme je, to vyžaduje $O(k \cdot \log k)$ porovnání. k nejmenších z nich vypíšeme, zbytek přidáme na začátek posloupnosti a opakujeme. Pro důkaz správnosti je potřeba uvědomit si dvě věci:

1) Vzhledem k tomu, že pozice každého prvku v posloupnosti se od své správné pozice v setříděné posloupnosti liší nejvýše o k, k nejmenších musí být mezi prvními $2k$ prvky posloupnosti, a tedy po setřídění tohoto úseku budou na správných pozicích.
2) Přerovnáním zbývajících k prvků nepokazíme to, že prvky jsou ve vzdálenosti nejvýše k od správné pozice. Ukážeme to sporem. Nechť tedy takový „pokažený“ prvek existuje. Buď x největší takový. Pokud je jeho správná pozice (v původní nezkrácené posloupnosti) nejvýše $2k$, nemohli jsme nic pokazit. Tedy tato pozice je nějaké q, $q > 2k$. Označme t novou pozici slova x, víme že $q - t > k$. Jestliže $t = 2k$, je x vůbec největší z prvních $2k$, a mohli jsme ho posunout pouze směrem ke q. Tedy $t < 2k$. Slovo y na pozici $t + 1$ je větší než x, jeho správná pozice je tedy alespoň $q + 1$. Nicméně $(q + 1) - (t + 1) > k$ a to je spor s tím, že x je největší s touto vlastností.

Časová složitost tohoto řešení je opět $O(n \cdot L \cdot \log k)$, protože tříděných úseků je $O(n/k)$. Paměťová je $O(k \cdot L)$, protože si stačí pamatovat aktuálně tříděný úsek. Program implementuje toto řešení, protože se nejsnáze naprogramuje a já jsem líný.

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

#define MAX_K 100
#define L 100

static char *words[2 * MAX_K + 1];

static void read_words (int *n, int k, char **tgt)
{
   char buffer[L + 1];

   while (*n > 0 && k-- > 0)
      {
         scanf ("%s", buffer);
         tgt[k] = strdup (buffer);
         (*n)--;
      }
   while (k-- > 0) tgt[k] = NULL;
}

static int cmps (const void *a, const void *b)
{
   const char *sa = * (const char **) a;
   const char *sb = * (const char **) b;

   if (!sa) return 1;
   if (!sb) return -1;
   return strcmp (sa, sb);
}

int main (void)
{
   int n, k, i;

   scanf ("%d%d", &n, &k);

   read_words (&n, k, words + k);
   while (1)
      {
         read_words (&n, k, words);
         qsort (words, 2 * k, sizeof (char *), cmps);
         if (!n)
            break;
         for (i = 0; i < k; i++)
            {
               puts (words[i]);
               free (words[i]);
            }
      }
```

```
  for (i = 0; words[i]; i++)
    {
      puts (words[i]);
      free (words[i]);
    }
  return 0;
}
```

16-1-2 Lokální minimum — Pavel Šanda

Je podivu hodné, kolik lidí se upokojilo s triviálním algoritmem o lineární časové složitosti – i přes explicitní upozornění, že to jde lépe. Pravda, nechalo se najít pár vylepšení typu číst každý druhý prvek, používat pouze konstatní paměťovou složitost, leč k nalezení logaritmické složitosti (co asi tak může být mezi $O(n)$ a nesmyslem $O(1)$?) je tato cesta neperspektivní. Jestliže má něco logaritmickou složitost, obvykle se tam nějakým způsobem rovnoměrně rozdělují vstupní data – v tomto případě se bude půlit celý interval a po výběru správné poloviny se úloha bude rekurzivně řešit pouze uvnitř této.

V intervalu $[x_0 \ldots x_{n/2} \ldots x_{n+1}]$ existuje Lokální Minimum (sporem). Vezmeme prostřední prvek. Pokud je LM, úloha je vyřešena. Jinak se LM nutně nalézá v polovině (či v obou), jejíž krajní prvek $x_{n/2\pm1}$ je menší než $x_{n/2}$.

Proč? Nechť bez újmy na obecnosti $x_{n/2} > x_{n/2+1}$. Pro spor předpokládejme, že v pravé polovině není LM. Proto nutně $x_{n/2+1} > x_{n/2+2}$ a stejným argumentem pokračuji až dostávám $x_n > x_{n+1}$, což je hledaný spor, neboť $x_{n+1} = \infty$.

Úlohu proto stačí rekurzivně spustit na vybranou polovinu.

I Plovací sval nahlédne, že to lze provést nejvýše logaritmus-krát, z čehož plyne i celková složitost algoritmu $O(\log n)$. Všechna čísla jsou uložena v poli, a tedy paměťová složitost je $O(n)$.

```
program minimum;
const MAXP = 100;
var
 l, r, m : Integer; {Levý, pravý konec a střed zkoumaného intervalu}
 n : Integer;       {Délka posloupnosti}
 A : Array[0..MAXP+1] of Integer; {Pole s prvky}
begin
 {Inicializace - vlastne ještě není součástí hledáni minima}
 Read(n);
 for l := 1 to n do
  Read(A[l]);
 A[0] := MAXINT;
 A[n+1] := MAXINT;
```

```
 {Začínáme hledat lokální minimum}
 l := 0; r := n+1;
 while l <> r do begin
  m := (l+r) div 2;
  if A[m] < A[m+1] then
   if A[m-1] < A[m] then
    r := m-1
   else begin
    l := m;
    r := m;
   end
  else
   l := m+1;
 end;
 WriteLn('Minimum je na pozici ', (l+r) div 2, '.');
end.
```

A ještě lahůdka pro céčkaře:

for (l=0, r=N+1, m= (l+r)/2; l!=r; m= (l+r)/2, $A[m]<A[m+1]$? ($A[m-1]<A[m]$?r=m−1:l=r=m): (l=m+1));

16-1-3 Důl — Dan Kráľ

Většina z vás správně rozpoznala, že tento příklad je jedním z oněch dvou, na které se má použít recept z naší programátorské kuchařky. Úlohu lze vcelku jednoduše vyřešit modifikovaným Dijkstrovým algoritmem, kde místo délky cesty uvažujeme nosnost nejhorší silnice, kterou cesta obsahuje.

Zmiňme tedy jen stručně rozdíly oproti algoritmu z naší kuchařky. Ohodnocení měst není tvořeno délkami nejkratších cest, ale maximálními nosnostmi dosud nalezených cest, kde nosnost cesty je minimum nosností všech silnic, které obsahuje. V každém kroku vybereme dočasně ohodnocené město M s největším ohodnocením, a jeho ohodnocení prohlásíme za trvalé. Poté zkusíme nalézt výhodnější cesty přes město M do zbylých (dočasně ohodnocených) měst. Poznamenejme, že nosnost cesty přes město M je minimum z nosnosti cesty do města M a silnice z města M do sousedního města. Důkaz správnosti se provede podobně jako důkaz správnosti původního Dijkstrova algoritmu.

Přímočará implementace právě popsaného algoritmu bez použití haldy má časovou složitost $O(n^2)$ a paměťovou složitost $O(n+m)$, kde n je počet měst a m je počet silnic. Řešení bez použití haldy, lze nalézt ve vzorovém programu. Za pomoci haldy, lze dosáhnout časové složitosti $O(m \log n)$.

Existuje i alternativní řešení pomocí datové struktury zvané *disjoint find union* (DFU), kterou si podrobně popíšeme v některém z následujících dílů naší

kuchařky. Datová struktura DFU nám umožňuje udržovat si rozklad množiny na disjunktní podmnožiny. Na začátku je množina rozložena na jednobodové podmnožiny a v průběhu práce s ní můžeme podmnožiny sjednocovat do větších. Tato datová struktura nám pak umožňuje odpovídat na dotazy, zda dva prvky jsou ve stejné podmnožině či nikoliv. Provedeme-li celkem K operací na n-prvkové množině, pak čas spotřebovaný touto datovou strukturou je nejvýše $O((K+n)\cdot\alpha(n))$, kde $\alpha(n)$ je inverzní funkce k tzv. Ackermannově funkci. Poznamenejme, že funkce $\alpha(n)$ roste s n velmi velmi pomalu a pro počet atomů ve vesmíru je tato tato funkce rovna 4.

Jak tedy takovéto řešení bude fungovat? Nejdříve si silnice setřídíme od té s největší nosností po tu s nejmenší. Následně budeme postupně povolovat silnice, které můžeme použít, tak dlouho dokud nebude existovat cesta z dolu do továrny jen po povolených silnicích. Silnice povolujeme od té s největší nosností. Na začátku není žádná silnice povolena. Množiny měst, mezi kterými existuje cesta jen po povolených silnicích, tvoří právě rozklad množiny všech měst, který udržujeme pomocí DFU. Právě popsané řešení má časovou složitost $O(m\log m)+O((m+n)\cdot\alpha(n)) = O(m\log n)$ a paměťovou složitost $O(n+m)$.

```
program dul;
const MAX=100;
var N: word; { počet měst }
   nosnosti: array[1..MAX,1..MAX] of integer; { nosnosti silnic }
  ohodnoceni: array[1..MAX] of integer;      { nosnosti nal. cest }
   odkud: array[1..MAX] of integer;           { předchozí město }
   trvaly: array[1..MAX] of boolean;          { trvale ohodnocen? }
   i,j,w:word;
   max_nosnost:integer;
procedure vypis(konec: word);
begin
  if odkud[konec]<>-1 then vypis(odkud[konec])
  else write('Nalezená cesta:');
  write(' ',konec);
end;
begin
  { Nejdříve načteme popis silniční situace Kaputánii }
  readln(N);
  max_nosnost:=0;
  for i:=1 to N do
    for j:=i+1 to N do
      begin
        readln(nosnosti[i][j]);
```

```
        nosnosti[j][i]:=nosnosti[i][j];
        if max_nosnost<nosnosti[i][j] then
          max_nosnost:=nosnosti[i][j];
      end;
  for i:=1 to N do
    begin
      trvaly[i]:=false;
      ohodnoceni[i]:=-1;
    end;
  odkud[1]:=-1;
  ohodnoceni[1]:=max_nosnost;
  repeat
    w:=0;
    for i:=1 to N do
      if not(trvaly[i]) then
        if w=0 then
          w:=i
        else
          if ohodnoceni[i]>ohodnoceni[w] then w:=i;
    if w<>0 then
      begin
        trvaly[w]:=true;
        for i:=1 to N do
          if (nosnosti[w][i]<>-1) and not(trvaly[i]) and
              { podmínku "not(trvaly[i])" lze vypustit }
             (ohodnoceni[i]<ohodnoceni[w]) and
              { podmínka ohodnoceni[i]<ohodnoceni[w] je vždy splněna !}
             (ohodnoceni[i]<nosnosti[w][i]) then
            begin
              if (nosnosti[w][i]<ohodnoceni[w]) then
                ohodnoceni[i]:=nosnosti[w][i]
              else
                ohodnoceni[i]:=ohodnoceni[w];
              odkud[i]:=w;
            end
      end
  until trvaly[N] or (w=0);
  if trvaly[N] then vypis(N)
  else write('Cesta z dolu do továrny neexistuje.');
  writeln;
end.
```

16-1-4 Neruš mé trojúhelníky! Tomáš Vyskočil

Představitele Bosstánu by jistě potěšilo, že se našlo tolik řešení jejich šifrovacího problému. Avšak ne všechna řešení byla dost rychlá, aby se dala používat. Spousta z vás řešila problém naprosto přímočarým způsobem, který ovšem běží v čase $O(n^3)$, a tedy velice pomalu už pro malé n. S malým pozorováním lze tento algoritmus vylepšit až na složitost $O(n^2)$. Algoritmus vlastně funguje stejně jako přímočaré řešení jen s tím rozdílem, že si pro každou dvojici bodů pamatujeme interval, ve kterém může ležet třetí bod (a výsledný trojúhelník přitom bude obsahovat střed) a ten pouze updatujeme. Avšak ani toto řešení, jak mnozí poznali, není nejlepší. Vzorové řešení pracuje v čase $O(n \log n)$ následujícím způsobem:

Nejprve si vstupní data, body udané úhly, setřídíme. Pak si problém převedeme na opačnou úlohu, tedy kolik trojúhelníku neobsahuje střed. Tato úloha se řeší velice jednoduše, přesněji v lineárním čase. Nejprve si určíme, kdy trojúhelník neobsahuje střed. To nastává právě tehdy, když jsou body trojúhelníka v intervalu dlouhém < 180 stupňů, tedy jednodušeji, jsou pouze na jedné půlce kružnice. A tyto body se již určí velice jednoduše. Postupně volíme body na kružnici a pro ně určujeme nejvzdálenější bod po směru kružnice takový, že jejich úhlová vzdálenost je < 180 stupňů. Protože konec intervalu pro bod y následující na kružnici po x stačí hledat za koncem intervalu pro bod x, celou kružnici objedeme maximálně 2-krát. Tedy algoritmus má lineární složitost. Zbývá snad už jen dodat, že je nutné si dát pozor na to, abychom započítali každý trojúhelník právě jednou.

Složitost třídění je $O(n \log n)$ a algoritmus na hledání počtu trojúhelníků neobsahujících střed má složitost $O(n)$. Tedy celková časová složitost algoritmu je $O(n \log n)$ a paměťová složitost je $O(n)$.

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

#define MAX 1024

float p[MAX];
int n;

int cmp (const void *a, const void *b)
{
        return * (float *)a - * (float *)b;
}

int test (int i, int j)
{
        int h = (j+1)/n;
        if (p[i] + 180 > p[ (j+1) % n] + 360 * h)
                return 1;
        return 0;
}
```

```
int main (void)
{
        int i, k, sum;

        scanf ("%d", &n);
        for (i=0; i<n; i++){
                scanf ("%f", &p[i]);
        }

        qsort (p, n, sizeof (float), cmp);

        k = 0;
        while (test (0, k)) k++;

        sum = (k - 1) * k / 2;
        for (i=1; i<n; i++){
                while (test (i, k)) k++;
                sum += (k - i) * (k - i - 1) / 2;
        }

        printf ("%d\n", n* (n-1)* (n-2)/6-sum);

        return 0;
}
```

16-1-5 Pravděpodobnostní algoritmy — Martin Mareš

Jak se ukázalo (a jak ostatně napoví i pohled na výsledkovou listinu), generování náhodných čísel s rovnoměrným rozdělením pravděpodobností je problém trochu potvornější, než jak na první pohled vypadá. Nejdříve si ho proto vyřešme pro případ, kdy je N mocninou dvojky, tedy rovné nějakému 2^k. Tehdy nám stačí k-krát si „hodit korunou“ (tj. zavolat funkci *randbit*) a výsledek si vyložit jako dvojkový zápis nějakého čísla mezi 0 a $N-1$:

```
int random2 (int N)
{
   int z = 0;
   N--;
   while (N)
      {
         z = 2*z + randbit ();
         N /= 2;
      }
   return z;
}
```

(všimněte si, že k ani nemusíme počítat, že stačí $N-1$ postupně dělit dvojkou tak dlouho, než vyjde 0).

Všechny výsledky jsou určitě stejně pravděpodobné, protože každé číslo je určeno právě jedním dvojkovým zápisem a pravděpodobnost každého dvojkového zápisu je $(1/2) \cdot (1/2) \cdot \ldots \cdot (1/2) = 2^{-k} = 1/N$.

Ale co když N nebude mocninou dvojky? Mohli bychom například zvolit nějaké $N' > N$, které mocninou dvojky bude (a takové určitě najdeme mezi N a $2N$), vygenerovat x v rozsahu 0 až $N' - 1$ a výsledek nějak „upravit“, aby byl vždy menší než N. Jak to ale přesně provést?

Pokus č. 1: Spočíst x mod N: to nebude fungovat, protože číslo 0 můžeme vygenerovat dvojím způsobem (pro $x = 0$ i $x = N$), zatímco třeba číslo $N - 1$ pouze jedním způsobem ($x = N - 1$), a proto má 0 dvojnásobnou pravděpodobnost než $N - 1$.

Pokus č. 2: Postupné generování dvojkového čísla po bitech si můžeme také vyložit jako hledání nějakého čísla v poli $(0, 1, \ldots, N - 1)$ půlením intervalů, přičemž v každém kroku se nerozhodujeme podle nerovnosti s hledaným číslem, nýbrž náhodně. To je bezpochyby pro $N = 2^k$ totéž, ale v ostatních případech musíme dříve nebo později narazit na interval, který se na dva stejně dlouhé rozdělit nedá. Ať už to ošetříme jakkoliv, opět dostaneme nestejné pravděpodobnosti vygenerovaných čísel.

A není náhodou, že se nám nedaří rovnoměrného rozdělení dosáhnout – ono to totiž na žádný konečný počet volání funkce *randbit* nelze. Poslyšte důkaz: Kdyby stačilo h hodů mincí, můžeme si program upravit tak, aby *randbit* volal vždy právě h-krát (počítali bychom si, kolikrát ho zavolal, a nakonec bychom doplnili příslušný počet volání, která by nijak neovlivnila výsledek programu). Program pak může probíhat 2^h různými způsoby podle toho, jakou posloupnost náhodných bitů od funkce *randbit* dostal, přičemž všechny tyto průběhy jsou stejně pravděpodobné. Každé číslo x, které program může vygenerovat, pak odpovídá některým z těchto průběhů (může jich být více, protože lze dospět různými cestami k témuž výsledku) a označíme-li si počet takových průběhů c_x, bude pravděpodobnost vygenerování čísla x rovna přesně $c_x/2^h$. Jenže pokud není N mocnina dvojky, nemůžeme žádanou pravděpodobnost $1/N$ vyjádřit jako $c_x/2^h$ pro žádné c_x. [Pokud je $a/b = c/d$, pak $ad = bc$, čili v našem případě $2^h = N \cdot c_x$. Jenže v rozkladu levé strany na součin prvočísel vystupují jen dvojky, v rozkladu pravé i jiná prvočísla.] Takže náš program opravdu nemůže fungovat.

Zkusme tedy na konečnost na chvíli zapomenout a navrhnout naprosto přímočaré řešení: Vygenerujeme náhodné číslo mezi 0 a $N' - 1$. Pokud bude menší jak N, prohlásíme ho za výsledek, jinak ho zahodíme a generujeme znovu:

```
int random (int N)
{
  int x;
  do
    x = random2 (N);
  while (x >= N);
  return x;
}
```

(zde zneužíváme toho, že výpočet k ve funkci *random2* zaokrouhluje nahoru, takže místo do $N-1$ generujeme až do $N'-1$, jak potřebujeme).

Jelikož výsledky funkce *random2* jsou, jak už víme, všechny stejně pravděpodobné a my si z nich pouze vybíráme a v případě, že si číslo nevybereme, zapomeneme veškerý dosavadní průběh výpočtu, musí být i výsledky naší nové funkce všechny stejně pravděpodobné.

Háček ovšem může být v tom, že nedokážeme předpovědět, kolik iterací bude naše funkce na vygenerování jednoho čísla potřebovat. Může se to povést už napoprvé, ale také se to nemusí povést vůbec – třeba tehdy, bude-li nám funkce *randbit* vracet stále samé jedničky. Nicméně pravděpodobnost toho, že se nám první číslo nebude hodit, je $p=1-N/N'<1/2$, pravděpodobnost toho, že ani to druhé ne, je $p^2<1/4$, …, l neúspěšných pokusů nastane s pravděpodobností $p^l<2^{-l}$ a nekonečná posloupnost neúspěchů má pravděpodobnost 0. [Což neznamená, že by nemohla nastat.]

Zkusme tedy spočítat, kolik pokusů budeme potřebovat *v průměru.* Nejprve si ale nadefinujme, co takový průměr přesně je: Mějme nějakou množinu náhodných jevů $\Omega=\{\omega_1,\omega_2,\ldots\}$, přičemž jev $\omega\in\Omega$ nastává s pravděpodobností $p(\omega)$. Pak pro každou funkci F, která přiřazuje těmto jevům nějaká reálná čísla, můžeme nadefinovat *střední hodnotu* $\mathrm{E}F$ takto:

$$\mathrm{E}F=\sum_{\omega\in\Omega}F(\omega)\cdot p(\omega).$$

Pokud mají všechny jevy stejnou pravděpodobnost, tedy $\forall\omega\in\Omega:p(\omega)=1/|\Omega|$, splývá tato definice s obyčejným (aritmetickým) průměrem všech funkčních hodnot; pokud ale jsou některé jevy pravděpodobnější, dáváme jim větší váhu, a tak průměr ovlivňují více.

Příklad 1: Pokud budeme házet kostkou a budeme chtít spočítat, kolik nám v průměru padne, bude naše množina $\Omega=\{1,2,3,4,5,6\}$ a všechná $p(\omega)$ budou rovna 1/6. Pak si stačí zvolit funkci $I(x)=x$ a spočítat její střední hodnotu:

$$\mathrm{E}I=\sum_{x=1}^{6}I(x)\cdot p(x)=\sum_{x=1}^{6}x\cdot\frac{1}{6}=\frac{1}{6}\cdot\sum_{x=1}^{6}x=\frac{21}{6}=\frac{7}{2}.$$

Příklad 2: Házíme dvěma kostkami a chceme spočíst průměrný součet. Množina Ω bude tentokrát zahrnovat všechny uspořádané dvojice (x,y), kde $1\le x,y\le 6$, každá dvojice bude mít pravděpodobnost 1/36 a střední hodnotu budeme počítat z funkce $S((x,y))=x+y$ [dvojích závorek se nelekejte, pouze vyjadřují, že parametrem funkce nejsou dvě čísla, nýbrž jedna uspořádaná dvojice čísel]. Mohli bychom počítat otrocky podle definice, ale raději si všimneme jedné zajímavé vlastnosti středních hodnot:

Odbočka: Střední hodnota je *lineární,* to znamená, že platí:

$$\begin{aligned} \mathrm{E}(F+G) &= \sum_{\omega\in\Omega}(F+G)(\omega)\cdot p(\omega) = \\ &= \sum_{\omega\in\Omega}(F(\omega)+G(\omega))\cdot p(\omega) = \\ &= \left(\sum_{\omega\in\Omega}F(\omega)\cdot p(\omega)\right)+\left(\sum_{\omega\in\Omega}G(\omega)\cdot p(\omega)\right) = \\ &= (\mathrm{E}F)+(\mathrm{E}G). \end{aligned}$$

a také (což se dokáže obdobně):

$$\mathrm{E}(\alpha\cdot F) = \ldots = \alpha\cdot(\mathrm{E}F) \qquad \text{pro každé } \alpha\in\mathbf{R}.$$

Intuitivně řečeno: E lze „přetáhnout“ jak přes +, tak přes násobení konstantou.

Také si všimněme, že střední hodnotu lze ekvivalentně definovat jako:

$$\mathrm{E}F = \sum_x x\cdot \mathrm{Pr}_\omega[F(\omega)=x], \tag{$*$}$$

kde sčítáme přes všechna x z oboru hodnot funkce F a $\mathrm{Pr}_\omega[\textit{podmínka}]$ značíme pravděpodobnost, že náhodně vybrané $\omega\in\Omega$ splňuje *podmínku*, jinými slovy součet všech $p(\omega)$ vyhovujících prvků ω. Pokud je jasné, co je náhodná proměnná, místo Pr_ω často píšeme jenom Pr. Stejně tak závorky okolo $(\mathrm{E}F)$ budeme vynechávat.

Zpět k příkladu 2: Funkci S můžeme vyjádřit jako součet dvou jednodušších funkcí $S_1((x,y))=x$ a $S_2((x,y))=y$, jejichž střední hodnota je $\mathrm{E}S_1=\mathrm{E}S_2=\frac{7}{2}$ (stačí si všimnout, že každou hodnotu x dosáhneme pro 6 různých y, takže jsme ve stejné situaci jako v předchozím příkladu). Proto

$$\mathrm{E}S = \mathrm{E}(S_1+S_2) = (\mathrm{E}S_1)+(\mathrm{E}S_2) = \frac{7}{2}+\frac{7}{2} = 7.$$

Zpět k analýze našeho algoritmu: Zkusme spočítat průměrný počet volání funkce *random2* během výpočtu, čili střední hodnotu funkce R, která každému možnému průběhu výpočtu (tj. tomu, jaké jsme dostali náhodné bity – zbytek je již jednoznačně určen) přiřadí, kolikrát byla v tomto případě *random2* zavolána. To můžeme elegantně spočítat třeba tak, že R vyjádříme jako součet funkcí $R_1, R_2, \ldots$, kde $R_i=1$, pokud i-té volání nastalo, jinak $R_i=0$. Pak dostaneme:

$$\mathrm{E}R = \mathrm{E}\left(\sum_{i=1}^{\infty}R_i\right) = \sum_{i=1}^{\infty}\mathrm{E}R_i.$$

Pravděpodobnost toho, že $R_i = 1$, jsme již spočítali a je to $p_{i-1} < 2^{1-i}$, takže podle $(*)$ dostaneme:

$$\mathrm{E}R_i = 0 \cdot \Pr[R_i = 0] + 1 \cdot \Pr[R_i = 1] = 0 + p_{i-1} < 2^{1-i}.$$

Z toho:

$$\mathrm{E}R < \sum_{i=1}^{\infty} 2^{1-i} = 2 \cdot \sum_{i=1}^{\infty} 2^{-i} = 2 \cdot 1 = 2$$

[to není nic jiného než součet nekonečné geometrické řady].

Spočítat z toho průměrnou časovou složitost našeho algoritmu je už hračka: nechť $T(\omega)$ značí časovou složitost výpočtu ω, čili $O(R(\omega) \cdot \log N)$. Pak

$$\mathrm{E}T = \mathrm{E}O(R \cdot \log N) = O((\mathrm{E}R) \cdot \log N) = O(\log N).$$

[Cvičeníčko: dokažte, že střední hodnotu lze „přetahovat přes O“.]

Takže i přesto, že náš algoritmus na první pohled vypadal velmi neefektivně, v průměrném případě běží jen logaritmicky dlouho (a tím pádem také funkci *randbit* volá jen logaritmicky-krát).

Jiný, možná elegantnější, způsob odvození hodnoty $\mathrm{E}R$ můžeme získat takto: Algoritmus v každém případě zavolá funkci *random2* jednou a pokud získá moc velké číslo (to nastává s pravděpodobností $p < 1/2$), dostane se opět na začátek a nic si z předchozího nepamatuje. Proto platí:

$$\mathrm{E}R = 1 + p \cdot \mathrm{E}R,$$

kterážto rovnice má řešení

$$\mathrm{E}R = 1/(1-p) < 2.$$

[Cvičeníčko: chvíli přemítejte o tom, proč je tato úvaha opravdu korektní.]

Poznámka: Někteří řešitelé vymýšleli různé důmyslné (a někdy i korektní) způsoby, jak část náhodných bitů z neúspěšných pokusů „recyklovat“ v pokusech dalších. To samozřejmě lze, ale jediné, co tím dokážeme, je pro některá N zmenšit multiplikativní konstantu schovanou v O-čku, což nestojí za tu námahu. Kdybychom ovšem řešili obecnější úlohu a bylo naším cílem generovat n náhodných čísel namísto jednoho, začalo by se recyklování náhodnosti vyplácet a došli bychom (po úctyhodném množství počítání) k tomu, že čím větší je n, tím více se průměrný počet náhodných bitů blíží k $n \cdot \log N$ (přesně, tedy bez O-čka!). To si ale třeba nechme na jindy, pro zvídavé napovím jen, že jeden ze způsobů, jak toho dosáhnout, je vzít populární kompresní algoritmus řečený aritmetické kódování, inicializovat mu pravděpodobnosti všech znaků na $1/N$ a poté nechat dekomprimovat posloupnost náhodných bitů.

Ještě jedna poznámka: V mnohých řešeních se vyskytlo počítání dvojkových logaritmů pomocí vzorců typu

$$\lfloor \log_2 N \rfloor = \text{trunc}(\log(N)/\log(2)).$$

To by bylo správně, nebýt jednoho zádrhele: počítač nepočítá s opravdovými reálnými čísly, nýbrž pouze s jejich aproximacemi. Proto všechny výpočty v typech jako je *real*, *float* apod. jsou zatíženy chybami, které i po zaokrouhlení mohou být podstatné. Často se samozřejmě dá dokázat, že v daném případě je výsledek správný, ale nebývá to jednoduché, takže pokud se chcete takovým problémům vyhnout, je praktické při počítání s celými čísly používat jen celočíselné operace.

Tak, to už je pravděpodobně všechno.

16-2-1 Král Eeek — Pavel Šanda

Jednociferný základ vyřešíme zvlášť: jestliže všechny cifry jsou menší než základ, máme jednu jedinou možnost, jinak nemáme žádnou.

Nechť n je počet cifer na vstupu, $s[i]$ jsou vstupní cifry.

Řešení v čase $O(n^3)$ je jednoduché: V poli $p[i]$ si budeme udržovat počet možností, které by existovaly, kdyby na vstupu chybělo prvních i čísel. Pro všechny délky základu z provedeme následující výpočet:

- $p[n-z-1] = 1$, protože jednociferné číslo je určité menší než alespoň 2-ciferný základ (tohle platí samozřejmě pouze tehdy, nezačíná-li základ cifrou nula).
- Nechť máme všechna $p[i+1]$, $p[i+2]$, ... a chceme spočítat $p[i]$. Zjistíme, kolikaciferné číslo může být na této pozici (dle konkrétních cifer to může být z nebo $z-1$, nebo pouze 1-ciferné, pokud ta cifra je 0; toto řeší funkce `kolik_cifer`), a počet možností pro $p[i]$ bude $p[i+1]$ (použijeme jednociferné číslo) + $p[i+2]$ (nebo dvouciferné) + $p[i+3]$ (nebo trojciferné) + ... + $p[i+z-1]$ (a možná + $p[i+z]$).

Pokud základ nezačínal nulou (viz. dříve), můžeme k celkovému řešení přičíst $p[0]$. Na každé cifře provedeme z kroků, cifer je $n-z$, všech základů je n, čili algoritmus pracuje v čase $O(n^3)$.

Časovou složitost lze vylepšit:

a) potřebujeme rychle zjišťovat, jestli na daném místě smí být z nebo $z-1$ cifer

b) potřebujeme počítat součet $p[i+1]+p[i+2]+\ldots+p[i+z-1]$ nějak inteligentně (tj. v konstantním čase)

a) jde vyřešit vyhledávacím automatem, ale to je tak trochu kanón na vrabce. Lepší je jít s délkou základu postupně od dvojky nahoru

a pamatovat si, jestli číslo bylo v minulém kroku dlouhé z či $z-1$. Jestliže nejlevější cifra zkoumaného čísla je menší než nejlevější číslice základu, není co řešit a zkoumané číslo určitě může být plné délky. Jestliže nejlevější cifra je větší, také není co řešit – číslo musí být kratší. Jestliže se nejlevější cifry shodují, stačí se podívat, jak to dopadlo v minulém kroku; v tomto kroce dopadne porovnání stejně.

b) zavedeme pomocné pole $c[i]$ takové, že bude platit $c[i] = \sum_{j=i}^{n-z-1} p[j]$. Když budeme potřebovat zjistit součet $p[a] + \ldots + p[b]$, spočítáme ho jako $c[a] - c[b+1]$.

Tímto algoritmus urychlíme na $O(n^2)$, paměťová složitost zůstane $O(n)$.

```
#include <stdio.h>
#define MAXN 10240
#define min (a, b) ( ( (a) < (b)) ? (a) : (b))

char s[MAXN];
int mensi[MAXN], nove_mensi[MAXN], n;

int kolik_cifer (int i, int z)
{
        int m = −1, res;

        if (s[i] < s[n−z])
                m = 1;
        else if (s[i] == s[n−z]) m = mensi[i+1];
             else m = 0;

        nove_mensi[i] = m;
        if (s[i] == '0')                    /* připadá v úvahu jediná cifra - nula */
          return 1;
        res = min (z−1+m, n−z−i);           /* jinak máme nanejvýš res cifer */
        return res;
}

int main (void)
{
        int c[MAXN], p[MAXN];
        int i, z, result;

        gets (s);
        n = strlen (s);

        for (i=0; i<n−1; i++) {
                mensi[i] = 0;
        }

        result = 1;                         /* Vyřešíme jednociferný základ */
        for (i=0; i<n−1; i++) {
                if (s[i] >= s[n−1])
                        result = 0;
                if (s[i] < s[n−1])
                        mensi[i] = 1;
        }
```

```
        for (z=2; z<n; z++) {                   /* A ještě víceciferné základy */
                c[n-z] = 1; c[n-z+1] = 0; p[n-z] = 1;

                for (i=n-z-1; i>=0; i--) {
                        int h = kolik_cifer (i, z);
                        p[i] = c[i+1] - c[i+h+1];
                        c[i] = c[i+1] + p[i];
                }
                if (s[n-z] != '0')              /* Pokud základ nezačíná nulou, OK */
                        result += p[0];

                for (i=0; i<n; i++)
                        mensi[i] = nove_mensi[i];
        }
        printf ("%d\n", result);
        return 0;
}
```

16-2-2 Král Ovopole — Honza Kára

Většina řešitelů této úlohy nevýslovně zkoušela královu trpělivost, když jejich řešení vyžadovala více pokusů, než bylo nezbytně nutné. Své počáteční rozhodnutí, že všichni švindlíři budou setnuti nebo přinejmenším vsazeni do věže, musel Ovopole brzy poněkud pozměnit pro nedostatek popravčích mistrů a věží. I bombardování pukavci krále brzy přestalo bavit, a tak většina špatných řešitelů byla potrestána jen nízkým bodovým ohodnocením. Abychom ale nebyli jednostranní, je třeba uvést, že se našlo i několik pěkných řešení, z nichž jedno dokonce předčilo očekávání organizátorů. Na snad všechna špatná řešení fungoval následující protipříklad (proto ho uvádím zde a neopisoval jsem ho do každého špatného řešení): Mějme dvě vejce a 28 pater. Pro tuto konfiguraci je správné (viz algoritmus níže) hodit nejdříve vejce ze šestého patra (patra číslujeme od nuly), pokud se nerozbije, tak ze dvanáctého, pokud se nerozbije, tak ze sedmnáctého, pak z dvacátého prvního, dvacátého čtvrtého, dvacátého šestého a nakonec z dvacátého sedmého patra (předpokládáme, že v nějakém patře se vejce rozbít musí – diskuse tohoto předpokladu je též uvedena níže). Pokud se vejce při některém pokusu rozbije, tak prostě budeme procházet patra od posledního, kde se vejce nerozbilo, směrem vzhůru, až najdeme hledané patro. V nejhorších případech potřebuje tento algoritmus pouze 7 pokusů na nalezení hledaného patra.

A nyní již jak má vypadat správné řešení: Protože v zadání nebylo zcela jasně řečeno, jestli se vejce musí rozbít při pádu z nejvyššího patra a jestli se nerozbije při pádu z nejnižšího patra, budeme předpokládat, že oba tyto případy mohou nastat. Abychom je nemuseli speciálně ošetřovat, přidáme si nad poslední patro ještě jedno s tím, že při pádu z tohoto patra se vejce zaručeně

rozbije. Nyní už můžeme začít hledat nejnižší patro, při pádu z nějž se vejce rozbije, protože víme, že takové patro zaručeně existuje.

Budeme zkoumat, mezi kolika nejvýše patry je možné rozeznat naše hledané patro, když máme k dispozici k vajec a l pokusů – tento počet pater si označíme $P(k,l)$. Je zřejmé, že $P(0,l)=1$ – víme, že z nějakého patra se vejce rozbije, nemáme žádný pokus, takže naše patro lze rozeznat pouze pokud není na výběr. Dále víme, že $P(k,k)=2^k$, protože můžeme použít půlení intervalu a lépe to nejde. Interval pater, ve kterém hledáme, si vždy udržujeme tak, aby zaručeně obsahoval naše hledané patro. Tedy pokud se vejce nerozbije z patra i, tak počátek intervalu nastavíme na $i+1$, pokud se vejce rozbije, tak konec nastavíme na i. Pro $k>l$ je zřejmě $P(k,l)=P(l,l)$. Dalším důležitým pozorováním je, že $P(k,l)=P(k-1,l-1)+P(k,l-1)$. Tato rovnost plyne z toho, že pokud vejce pustíme z nějakého patra a ono se rozbije, tak lze naše hledané patro nalézt mezi $P(k-1,l-1)$ patry (rozbili jsme jedno vejce a udělali jeden pokus), pokud se vejce nerozbije, tak lze naše patro nalézt mezi $P(k,l-1)$ patry. A nyní přijde kouzlo (pro matematiky znalejší to asi nebude až takové kouzlo, ale dlužno poznamenat, že mě též nenapadlo, ale uviděl jsem ho v jednom řešení): Tvrdíme, že $P(k,l)=\sum_{i=0}^{k}\binom{l}{i}$. Ověření této rovnosti je snadná aplikace matematické indukce (využije se, že $\binom{l}{i}+\binom{l}{i+1}=\binom{l+1}{i+1}$), a proto ho vynecháme. Pro ty, kdo neví, co znamená $\binom{n}{k}$: je to tzv. *kombinační číslo* a říká, kolik různých k-prvkových podmnožin lze vybrat z množiny velikosti n. Je určeno následujícím vzorečkem:

$$\binom{n}{k}=\frac{n!}{k!\cdot(n-k)!},$$

kde $n!=n\cdot(n-1)\cdot(n-2)\cdot\ \cdots\ \cdot 2\cdot 1$.

Dále se nám bude hodit, že z $P(k,l)$ umíme rychle spočítat:

$$P(k-1,l)=P(k,l)-\binom{l}{k}$$

$$P(k,l+1)=P(k,l)+P(k-1,l)=2\cdot P(k,l)-\binom{l}{k}$$

$$P(k-1,l-1)=\left(P(k-1,l)+\binom{l-1}{k-1}\right)/2=$$

$$=\left(P(k,l)-\binom{l}{k}+\binom{l-1}{k-1}\right)/2$$

Pozn.: Jak zajistíme, že máme vždy po ruce spočítané potřebné kombinační číslo, si můžete nalézt v programu.

Matematickou přípravu již máme za sebou a nyní je čas ukázat, jak to vše využijeme v našem algoritmu. Nejdříve musíme nalézt nejmenší l takové, že $P(k,l) \geq n+1$ (n je počet pater, máme jedno patro přidané nahoře). To snadno uděláme tak, že budeme zkoušet l od 0 a průběžně počítat $P(k,l)$ – využíváme, že umíme spočítat $P(k,l+1)$ z $P(k,l)$. Když toto nejmenší l nalezneme, začnou skutečné pokusy. V proměnné `nejmensi` si budeme uchovávat nejnižší číslo patra, ze kterého se vejce může rozbít (na počátku `nejmensi`$= 0$). Vejce vždy pustíme z patra $nejmensi+P(k-1,l-1)-1$. Pokud se vejce rozbilo, tak snížíme k i l o jedna, přepočítáme P a pokračujeme dalším pokusem. Pokud se vejce nerozbilo, zvýšíme `nejmensi` o $P(k-1,l-1)$, snížíme l o jedna, přepočítáme P (na přepočítávání P používáme rovnosti uvedené výše) a opět pokračujeme dalším pokusem. Z definice P takto zjevně po l krocích nalezneme hledané patro a počet kroků byl nejmenší možný.

Paměťová složitost algoritmu je $O(1)$, časová $O(l)$, kde l je nejmenší možný počet pokusů potřebný k nalezení patra.

Pro zájemce: proč je $P(k,k) = 2^k$? Tento zápis znamená, že na k pokusů umíme najít hledané patro, jen pokud ho hledáme nanejvýš mezi 2^k patry (víme, že jedno z nich je ono hledané). Pokud použijeme půlení intervalu, opravdu tohoto výsledku dosáhneme (při každém pokusu se můžeme zbavit poloviny pater). Ale nejde to lépe?

Jeden pokus skončí jednou ze dvou různých možností (vajíčko se rozbije/nerozbije). Dva pokusy skončí jednou ze čtyř různých možností (první vajíčko se rozbije/nerozbije a druhé se rozbije/nerozbije). Obecně k pokusů skončí jednou z 2^k možností. Hledané patro určíme podle toho, jakou možností našich k provedených pokusů skončilo. Jedné takové možnosti (posloupnosti výsledků) může odpovídat nanejvýš jedno patro, protože jinak bychom po provedení k pokusů stále nebyli rozhodnuti, které patro je to hledané. Protože ale možností je 2^k, není možno najít hledané patro mezi více než 2^k patry.

```
program KingsField;
var
 n, k, l : Integer;   {Počet pater; počet vajec; počet pokusů}
 P, C : Integer;      {Spočítaný max. počet pater;Predpočítané k.č.}
 nejmensi : Integer; {Nejnižší patro, odkud se vejce může rozbít}
 rozbilo : Integer;  {Rozbilo se hodem vejce?}
begin
 Write('Pocet pater:');
 Read(n);
 Write('Pocet vajec:');
 Read(k);
 l := 0;
```

```
  P := 1;
  C := 1;                {v C bude uloženo (l nad k), pokud l>=k}

  while P < n+1 do begin {Najdeme minimální nutný počet pokusů}
    if l < k then        {Dokud je dost vajec, půlíme}
      P := 2*P
    else begin
      P := 2*P - C;
      C := C*(l+1) div (l+1-k); {(l nad k) -> (l+1 nad k)}
    end;
    Inc(l);
  end;

  nejmensi := 0;          {Jdeme zkoušet hody}
  while l > 0 do begin
    P := (P-C+C*k div l) div 2;   {Přepočteme P(k,l) na P(k-1,l-1)}
    Write('Hod vejce z patra ',nejmensi+P-1,'. Rozbilo se?');
    Read(rozbilo);
    if rozbilo = 1 then begin
      C := C*k div l;      {Prepočteme (l nad k) na (l-1 nad k-1)}
      Dec(k);              {P už je správně přepočteno...}
    end
    else begin
      nejmensi := nejmensi + P;
      C := C*(l-k) div l;{Přepočteme (l nad k) na (l-1 nad k)}
      P := P + C;          {Přepočteme P(k-1,l-1) na P(k,l-1)}
    end;
    Dec(l);
  end;
  if nejmensi = n then
    WriteLn('Vejce se nerozbije z zadneho patra.')
  else
    WriteLn('Vejce se rozbije z patra ',nejmensi,'.');
end.
```

16-2-3 Král Potvorník — Pavel Čížek

Idea algoritmu je jednoduchá. Vezmeme body, setřídíme je dle X, vezmeme provázek, přivážeme ho k nejlevějšímu bodu a budeme jím naší množinu sloupů obtahovat po směru hodinových ručiček. Konvexní obal tedy bude po celou dobu „zatáčet“ doprava. Nyní nechť máme zpracované body $1..N$ zleva a známe jejich konvexní obal. Chceme přidat další bod. Připojíme ho tedy k seznamu konvexního obalu a „zatáhneme“ za provázek. Ten se vzdálí ode všech sloupů u nichž „zatáčel vlevo“. Za povšimnutí stojí, že tyto sloupy tvoří v konvexním obalu souvislý úsek, který končí před právě přidaným bodem (úsek může být i prázdný). Takže v programu se stačí kouknout na 3 poslední body nynějšího obalu a dokud tam obal „zatáčí“ vlevo (čili je nekonvexní), předposlední bod vyhazovat. Protože každý bod můžeme vložit, resp. vyhodit nejvýše jednou, N kroků trvá $O(N)$.

Pokud započítáme i třídění, složitost celého algoritmu je $O(N \log N)$. Ale má to drobný háček. Takhle bychom dopadli jako v jistém večerníčku, protože bychom vytvořili jen horní část konvexního obalu. Nicméně není těžké si rozmyslet, že stejně lze zkonstruovat i dolní polovinu (nebo k „vstupní“ posloupnosti bodů připojit na konec posloupnost stejných bodů, které budou setříděné obráceně, a pustit výše popsaný algoritmus na tuto prodlouženou posloupnost. Zřejmě pak bude provázek stále „zatáčet doprava“, ale jelikož jsme se „otočili“ o 180 stupňů, vytvoří i dolní část obalu).

Program je jen implementací výše uvedeného algoritmu, časová náročnost je $O(N \log N)$ a paměťová $O(N)$.

Na závěr bych přidal jednu poznámku k došlým řešením. Relativně hodně jich běhalo v čase $O(N \log N)$, ale používalo při tom funkce typu `ArcTan` apod... Proč? Tyto funkce nepatří, přiznejme si, zrovna k nejrychlejším a program lze napsat bez nich stejně jednoduše. Ideální je použít vektorový součin.

Vektorový součin třísložkových vektorů A a B je takový třísložkový vektor C, pro který platí:

1) Velikost C je rovna ploše rovnoběžníku definovaného vektory A a B. Matematicky: $|C| = |A| \cdot |B| \cdot sin(\varphi)$, kde φ je úhel, který svírají vektory A a B.
2) Vektor C je kolmý na vektory A a B (zřejmě je-li B násobkem vektoru A, vektor C nemůže být určen jednoznačně (vektorů kolmých na A a B je mnoho). V tomto případě je naštěstí – dle bodu 1 – vektor C nulový).
3) Vektor C má (není-li nulový) směr, který odpovídá směru palce pravé ruky,pokud jí přiložíme k vektoru A a prsty „stočíme“ po menším úhlu k vektoru B. Tato podmínka vlastně pouze určuje znaménko vektoru C.

Tohle stačí k tomu, abychom výsledný vektor jednoznačně definovali. Jak se tedy vektorový součin třísložkových vektorů A a B počítá?

$$A \times B = (A_y \cdot B_z - A_z \cdot B_y, A_z \cdot B_x - A_x \cdot B_z, A_x \cdot B_y - A_y \cdot B_x)$$

My použijeme vektorový součin takto: pro tři body A,B,C, které leží v tomto pořadí na obvodu zahrady, nám vektorový součet řekne, zda $|\angle ABC| < 180°$, měřeno od polopřímky BA po směru hodinových ručiček. Vlastně nám poví, zda bude provázek v bodě B „zatáčet“ doleva ($|\angle ABC| > 180°$) či doprava ($|\angle ABC| < 180°$), tedy zda bod B patří nebo nepatří do konvexního obalu.

Výpočet provedeme takto: vektor U bude vektor od bodu B k bodu A, vektor V bude vektor od bodu B k bodu C. Spočteme z-ovou souřadnici součinu $U \times V$ (všimněte si, že při tomto výpočtu nepotřebujeme z-ové složky vektorů U a V, takže tyto vektory mohou být pouze dvousložkové – to ale chceme, neboť zadané body leží v rovině a tedy mají pouze dvě souřadnice). Bude-li z-ová souřadnice součinu kladná, $|\angle ABC| < 180°$, bude-li záporná, $|\angle ABC| > 180°$ (rozmyslete si, jak to plyne z bodu 3).

```
program KralPotvornik;
const Presnost = 1E-10; {U reálných čísel musíme počítat s chybou}
type PSloup = ^TSloup;
   TSloup = record
         X,Y:real;
         Dalsi:PSloup;
         end;
var Sloupy:PSloup;

function Nacti:PSloup;
var NovySloup:PSloup;
   Vystup:PSloup;
   PocetSloupu:longint;
begin
  Vystup:=nil;
  Readln(PocetSloupu);
  if PocetSloupu<3 then begin
    writeln('A z tohohle chces vytvorit konvexni obal?!');
    halt;
  end;
  while PocetSloupu>0 do begin
    new(NovySloup);
    readln(NovySloup^.X,NovySloup^.Y);
    NovySloup^.Dalsi:=Vystup;   {Přidáme do seznamu sloupů}
```

```
    Vystup:=NovySloup;
    dec(PocetSloupu);
  end;
  Nacti:=Vystup;
end;

procedure MergeSort(var Co:PSloup);
var Seznam:array[boolean] of PSloup;
   DalsiPrvek,PosledniPrvek:PSloup;
   KterySeznam:boolean;
begin
  if Co^.Dalsi=nil then exit;      {Není co řešit}
  KterySeznam:=false;
  Seznam[false]:=nil;Seznam[true]:=nil;
  while Co<>nil do begin
    DalsiPrvek:=Co^.Dalsi;
    Co^.Dalsi:=Seznam[KterySeznam];
    Seznam[KterySeznam]:=Co;
    KterySeznam:=not(KterySeznam);
    Co:=DalsiPrvek;
  end;
  MergeSort(Seznam[false]);
  MergeSort(Seznam[true]);
  PosledniPrvek:=nil;
  while (Seznam[false]<>nil) and (Seznam[true]<>nil) do begin
    KterySeznam:=Seznam[true]^.X<Seznam[false]^.X;
    if abs(Seznam[true]^.X-Seznam[false]^.X)<Presnost then
      KterySeznam:=Seznam[true]^.Y>Seznam[false]^.Y;
    DalsiPrvek:=Seznam[KterySeznam];
    Seznam[KterySeznam]:=Seznam[KterySeznam]^.Dalsi;
    if PosledniPrvek=nil then
      Co:=DalsiPrvek             {Výstup je zatím prázdný}
    else
      PosledniPrvek^.Dalsi:=DalsiPrvek;     {Přidáme na konec}
    DalsiPrvek^.Dalsi:=nil;         {Opravíme konec}
    PosledniPrvek:=DalsiPrvek;
  end;
  PosledniPrvek^.Dalsi:=Seznam[Seznam[false]=nil];
end;         {...a je to setříděné}
```

```
procedure PridejKopii(Sloupy:PSloup);
var Kopie,Nova:PSloup;
begin
  Kopie:=nil;
  while Sloupy^.Dalsi<>nil do begin
    new(Nova);
    Nova^:=Sloupy^;
    Nova^.Dalsi:=Kopie;
    Kopie:=Nova;
    Sloupy:=Sloupy^.Dalsi;
  end;
  Sloupy^.Dalsi:=Kopie;          {...a přidáme kopii na konec}
end;

function KonvexniObal(ZCeho:PSloup):PSloup;
var Obal,DalsiSloup:PSloup;
   Vektor1,Vektor2:record X,Y:real; end;
begin
  DalsiSloup:=ZCeho;
  ZCeho:=ZCeho^.Dalsi;
  DalsiSloup^.Dalsi:=nil;
  Obal:=ZCeho;
  ZCeho:=ZCeho^.Dalsi;
  Obal^.Dalsi:=DalsiSloup;   {Vložíme první 2 body do posloupnosti}
  while ZCeho<>nil do begin
    DalsiSloup:=ZCeho;
    ZCeho:=ZCeho^.Dalsi;
    DalsiSloup^.Dalsi:=Obal;
    Obal:=DalsiSloup;
    while Obal^.Dalsi^.Dalsi<>nil do begin   {Máme alespoň 3 body?}
      {Vektor2 určuje směr predposl. a posl. sloupu}
      Vektor2.X:=Obal^.X-Obal^.Dalsi^.X;
      Vektor2.Y:=Obal^.Y-Obal^.Dalsi^.Y;
      {Vektor1 je totéž vzhledem k předchozím dvěma}
      Vektor1.X:=Obal^.Dalsi^.X-Obal^.Dalsi^.Dalsi^.X;
      Vektor1.Y:=Obal^.Dalsi^.Y-Obal^.Dalsi^.Dalsi^.Y;

      if (Vektor1.X*Vektor2.Y-Vektor1.Y*Vektor2.X)>Presnost then
        break;
      DalsiSloup:=Obal^.Dalsi; {Bod je třeba odebrat}
      Obal^.Dalsi:=Obal^.Dalsi^.Dalsi;
```

```
      dispose(DalsiSloup);
    end;
  end;
  KonvexniObal:=Obal;
end;

function Delka(Obal:PSloup):real;
var Vystup:real;
begin
  Vystup:=0;
  while Obal^.Dalsi<>nil do begin
    Vystup:=Vystup+sqrt(sqr(Obal^.X-Obal^.Dalsi^.X)+
        sqr(Obal^.Y-Obal^.Dalsi^.Y));
    Obal:=Obal^.Dalsi;
  end;
  Delka:=Vystup;
end;

begin
  Sloupy:=Nacti;
  MergeSort(Sloupy);
  PridejKopii(Sloupy);
  Sloupy:=KonvexniObal(Sloupy);
  writeln('Delka plotu je:',Delka(Sloupy):10:3);
end.
```

16-2-4 Křížový král — Martin Mareš

Hojnost neoptimálních řešení, která jsme dostali, zdá se nasvědčovat tomu, že drak je v posledních letech výjimečně sytý a dobře naladěný, či že mu možná posledních pár set let nejde počítání pokusů tak bystře, jako ve starých dobrých časech udatných a hlavně křupavých těch, no, rytířů. Zde budeme ctěnému publiku demonstrovati zaručeně spolehlivý návod, kterak draka důmyslem svým přechytračiti (čili tzv. návod na draka).

Abychom si zjednodušili práci, předpokládejme bez újmy na obecnosti, že nehádáme čísla od 1 do N, nýbrž od 0 do $N-1$ a že N je mocnina dvojky (kdyby nebylo, stačí N zvýšit, čímž si, jak uvidíme, pohoršíme jen nepatrně).

Kdyby drak nelhal (což je podobně pravděpodobné, jako kdyby peklo zamrzlo a turecký med byl zadarmo… ale potěšme se tou představou aspoň chviličku), bylo by všechno jednodušší: stačilo by se postupně vyptat na hodnoty jednotlivých bitů čísla, tj. v i-tém kroku hry nabídnout drakovi ta čísla, jejichž i-tý bit je jedničkový. Takto bychom číslo uhodli po $\log N$ krocích (log

bude vždy značit dvojkový logaritmus). Podobnost s půlením intervalů není nikterak náhodná. Také si všimněte, že naše strategie nijak nespoléhá na to, že hádáme číslo z intervalu – kdybychom volili mezi nějakými obecnými možnostmi $a_0, \ldots, a_N$, postupovali bychom úplně stejně, až na to, že bychom se nerozhodovali podle bitů čísel a_i, nýbrž podle bitů jejich indexů i.

Jenže drak samozřejmě lže (a peklo není studené a turecký med je nekřesťansky drahý...), takže když se zeptáme stejně jako předtím, nedostaneme jednoznačný výsledek, nýbrž více možností: jednak tu dřívější, odpovídající tomu, že drak nelhal, a dále $\log N$ dalších, odpovídajících všem možným otázkám, na které drak mohl odpovědět lživě. Ale jaképak s tím círáty, použijeme ještě jednou stejnou strategii, tentokráte ovšem pouze na zbylých $\log N + 1$ možností (zde použijeme trik s indexy). Na to budeme potřebovat $\lceil \log(1 + \log N) \rceil$ otázek.

Leč drak nám také mohl lhát během této druhé fáze, proto mezi první a druhou fázi přidáme ještě jeden dotaz, kterým si ověříme, zda výsledek první fáze byl správný a pokud byl, druhou fázi ani nespustíme.

Nyní si rozmysleme, jak by nám drak mohl lhát: Pokud lže v první fázi, ověření neprojde a druhá fáze (určitě bez lži) najde správný výsledek. Pokud lže v ověření, jsou všechny ostatní odpovědi správně a druhá fáze opět vydá správný výsledek. Kdyby se vyskytovala lež v druhé fázi, znamenalo by to, že odpovědi jak v první fázi, tak během ověření byly pravdivé, což by ovšem znamenalo, že jsme se do druhé fáze ani nedostali.

Dohromady jsme potřebovali (včetně zaokrouhlení na mocninu dvojky) $\lceil \log N \rceil + \lceil \log(1 + \lceil \log N \rceil) \rceil + 1$ otázek, což pro $N = 1\,000\,000$ opravdu dá kýžených 26. Otázky a odpovědi zvládneme zpracovat jednoduchým programem s časovou složitostí $O(N \cdot \log N)$ a paměťovou $O(N)$.

Pozor, matematika! Pro odolnější povahy připojíme navíc důkaz (skoro)optimality našeho řešení. Představme si, že máme program hádající číslo pomocí binárních otázek, přičemž každá otázka je jednoznačně určena odpověďmi na předcházející otázky. Předpokládejme navíc, že program pokaždé použije stejný počet r otázek (pokud nepoužije, můžeme vždy na konec doplnit zbytečné otázky, které nijak neovlivní výsledek). Přiřaďme každému číslu x posloupnost $p_1(x), \ldots, p_r(x)$ odpovědí (kódovaných 0=ne, 1=ano), které vedou k tomuto číslu bez lhaní, a dále posloupnosti $p_1^i(x), \ldots, p_r^i(x)$ pro $i = 1, \ldots, r$ popisující odpovědi v případě, že drak zalhal při i-té otázce.

Snadno nahlédneme, že všech $r + 1$ posloupností odpovídajících témuž číslu musí být navzájem různých (dvěma různými lžemi není možné dojít k témuž výsledku) a že libovolná posloupnost pro číslo x musí být různá od libovolné posloupnosti pro každé jiné číslo y (protože jinak bychom nebyli schopni z posloupnosti odpovědí jednoznačně zrekonstruovat drakovo číslo).

Existuje tedy $N \cdot (r+1)$ různých posloupností odpovědí, ty se ovšem musí vejít do množiny všech možných binárních posloupností délky r, kterých je 2^r, a proto musí platit $N \cdot (r+1) \leq 2^r$, čili $N \leq 2^r/(r+1)$, což nám pro každý počet dotazů r říká, kolik nejvýše čísel jsme schopni rozlišit. Pokud zkusíme nerovnost „obrátit“, vyjde nám opravdu přibližně $r \geq \log N + \log\log N$, čili náš algoritmus je velice blízko k optimu.

Pro ty, jimž by nedalo spát, že jsme jim nabídli řešení jenom *skoro* optimální, přidáváme konstrukci fungující pro libovolné $r = 2^z - 1$, která opravdu dosáhne maximálního možného N, které nám z naši nerovnosti vyšlo, tedy $N = 2^r/(r+1) = 2^{r-z}$. Nejdříve si sestrojíme *binární kód se vzdáleností 3*, čili množinu N dvojkových posloupností délky r (těm budeme říkat slova) takových, že každé dvě slova se liší v alespoň třech prvcích. K tomu stačí vzít všechny možné $(r-z)$-bitové posloupnosti (těch je N) a za každou z nich přidat z *kontrolních bitů,* abychom dosáhli požadované vzdálenosti. Původním bitům budeme říkat *datové* a oindexujeme si je z-cifernými dvojkovými čísly, přičemž vynecháme nulu a všechna čísla obsahující právě jednu jedničku (takže nám zbude $2^z - z - 1 = r - z$ indexů, jak potřebujeme). Kontrolní bity pak spočítáme takto: i-tý kontrolní bit bude určovat paritu datových bitů, jejichž index má i-tý bit jedničkový (parita je 0, pokud je mezi vybranými datovými bity sudý počet jedniček, jinak je 1). Příklad pro $N = 11$ a $z = 4$ následuje:

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	c_0	c_1	c_2	c_3
1	1	0	1	1	0	1	0	1	0	1	←			
1	0	1	1	0	1	1	0	0	1	1		←		
0	1	1	1	0	0	0	1	1	1	1			←	
0	0	0	0	1	1	1	1	1	1	1				←

Každá dvě sestrojená slova se budou lišit alespoň jedním bitem v datové části. Pokud se liší právě jedním, budou se lišit i alespoň dvěma kontrolními bity (protože v indexu změněného datového bitu jsou alespoň dvě jedničky). Kdyby se datové části lišily dvěma bity, musí mít jejich indexy alespoň jeden bit rozdílný, takže alespoň jeden kontrolní bit se také liší. Čili každá dvě slova se liší v alespoň třech bitech, jak jsme slíbili.

Naši množinu slov lze použít jako samoopravný kód na opravu jedné chyby: pokud nám někdo pošle libovolné ze slov a při přenosu nastane nejvýše jedna chyba, jsme schopni jednoznačně určit, které slovo bylo vysláno. (Pokud by se slovo x změněné v jednom bitu shodovalo se slovem y změněným rovněž nejvýše v jednom bitu, znamenalo by to, že slova x a y se liší v max. dvou bitech, což víme, že není možné.)

Stejným způsobem ovšem můžeme komunikovat s drakem: číslům 0 až $N-1$ přiřadíme našich N slov a v i-tém kole se draka zeptáme na i-tý bit slova (vyjmenujeme čísla, jejichž slova mají tento bit jedničkový), čímž dostaneme

hledané slovo s maximálně jedním bitem změněným a ten je možno opravit. Máme tedy strategii, která je optimální pro vybraná r (zkuste nahlédnout, že funguje a je optimální i pro ostatní r), je ji možno naprogramovat v čase $O(N \cdot \log N)$ a paměti $O(N)$ a dokonce nám i spolehlivě odhalí, zda drak zalhal. Náš program se nicméně bude držet mnohem jednoduššího prvního řešení, které je vždy o nejvýše 2 dotazy horší než optimální.

```
#include <stdio.h>

#define MAX 1048576                  /* Do kolika umí draci počítat */
int query[MAX];                      /* Na co se ptáme (my počítáme od 0, drak
                                        od 1) */

int qlen;

int ask_dragon (void)                /* Zeptáme se draka, copak nám řekne? */
{
        int i;
        for (i=0; i<qlen; i++)
                printf ("%d ", query[i]+1);
        printf (" ?");
        scanf ("%d", &i);
        return i;
}

int main (void)
{
        int N;                       /* Hledané číslo je mezi 0 až N-1 */
        int guess;                   /* Jaké jsme zatím uhádli */
        int bit;                     /* Který bit zkoumáme */
        int c;                       /* Číslo, které zrovna zkoumáme */

        printf ("Velevazeny draku, rci N! ");
        scanf ("%d", &N);

        guess = 0;                   /* 1. fáze: Půlení intervalů na 0..N-1 */
        for (bit=1; bit<N; bit += bit) {
                qlen = 0;
                for (c=0; c<N; c++)
                        if (c & bit)
                                query[qlen++] = c;
                if (ask_dragon ())
                        guess |= bit;
        }

        query[0] = guess;            /* Kontrola (pro N=1 ji neděláme) */
        qlen = 1;
        if (N != 1 && !ask_dragon ()) {      /* Někdy lhal, ve 2. fázi najdeme, kdy */
                int rest[8*sizeof (int)+1];  /* Jaké jsou možnosti? */
                int nrest;
                nrest = 0;           /* Špatně byla kontrola */
                rest[nrest++] = guess;
                for (bit=1; bit<N; bit += bit)   /* Nebo jednotlivé bity */
                        rest[nrest++] = guess ^ bit;
```

```
            guess = 0;                      /* Půlení intervalů pro zbylé možnosti */
            for (bit=1; bit<nrest; bit += bit) {
                    qlen = 0;
                    for (c=0; c<nrest; c++)
                            if (c & bit)
                                    query[qlen++] = rest[c];
                    if (ask_dragon ())
                            guess |= bit;
            }
            guess = rest[guess];            /* Jdeme na jisto */
    }
    printf ("Ha, Tve cislo je %d!\n", guess+1);
    return 0;
}
```

16-2-5 Král Popleta — Zdeněk Dvořák a Kráľ Dan

Nejprve si úlohu převedeme do řeči logických výrazů, čili výrazů nad proměnnými, které mohou nabývat pouze hodnot `true` a `false`. S těmito proměnnými můžeme provádět různé logické operace. Pro náš příklad budou důležité pouze dvě – negace, definovaná vztahy `not false` = `true` a `not true` = `false`, a disjunkce, definovaná vztahy `false or false` = `false`, `false or true` = `true`, `true or false` = `true` a `true or true` = `true`.

Pro každou provincii si zavedeme jednu proměnnou. Ta bude nabývat hodnoty `true`, pokud provincii získá Petr, a `false`, pokud ji získá Pavel.

Jak snadno nahlédneme, požadavky druhého a třetího typu jsou ve skutečnosti stejné – oba jsou splněny, pokud Petr (resp. Pavel) dostane alespoň jednu ze zmiňovaných provincií. Všechny požadavky tedy jdou vyjádřit jedním z těchto způsobů:

- Jestliže Petr požaduje provincii x, odpovídá tomu výraz (x).
- Jestliže Pavel požaduje provincii x, odpovídá tomu výraz (`not` x).
- Jestliže Petr požaduje provincii x nebo provincii y, odpovídá tomu výraz (x `or` y).
- Jestliže Pavel požaduje provincii x nebo provincii y, odpovídá tomu výraz (`not` x `or` `not` y).

Úkolem této úlohy je nalézt takové přiřazení hodnot `true` a `false` proměnným, aby co nejvíce z těchto výrazů mělo hodnotu `true`.

Protože každé tři požadavky lze splnit, platí:

- Pro žádnou proměnnou x se mezi požadavky nevyskytuje jak (x), tak (`not` x). Můžeme tedy bez újmy na obecnosti předpokládat, že všechny požadavky, které obsahují pouze jednu proměnnou, jsou ve

tvaru (x) – pokud je požadavek ve tvaru (`not` x), ve všech požadavcích prohodíme x za `not` x a naopak. Tím zjevně nezměníme počet splnitelných požadavků – pokud by řešení nově vzniklé úlohy dávalo x hodnotu `false`, dáme mu v řešení původní úlohy hodnotu `true` a naopak.
- Nemůže se stát, že by (`not` x `or` `not` y), (x) a (y) byly požadavky. Tedy platí, že pro každý požadavek tvaru (`not` x `or` `not` y) se buď (x) nebo (y) nevyskytují jako požadavek.

Nyní se vraťme k řešení úlohy. Řešení, které se králi nelíbilo, bylo dát každé proměnné hodnotu `true` s pravděpodobností 1/2. Tím by splnil požadavky zmiňující dvě provincie s pravděpodobností 3/4, nicméně kazí mu to požadavky na jednotlivé provincie, které splní s pravděpodobností pouze 1/2. Zkusme tedy pomoci těmto požadavkům – dejme proměnné x, která se vyskytuje sama jako požadavek, hodnotu `true` s pravděpodobností $p > 1/2$. Ostatní proměnné, které se nevyskytují jako samostatné požadavky, dále volme s pravděpodobností 1/2.

Požadavky typu (x) na jednotlivé provincie splníme s pravděpodobností p. Požadavky typu (`not` x `or` `not` y) splníme s pravděpodobností $1-p/2$ pokud se právě jedno z x, y vyskytuje jako požadavek, nebo s pravděpodobností 3/4, pokud se tak nevyskytuje ani jedno z nich (3/4 je zřejmě větší než $1-p/2$). Snadno si rozmyslíme, že ve všech ostatních případech bude pravděpodobnost splnění požadavku také alespoň $1-p/2$. Dohromady tedy v průměru splníme alespoň $\min(p, 1-p/2)$ z požadavků. Toto číslo bude největší, pokud $p = 1-p/2$, čili když $p = 2/3$. V tomto případě jsme schopni splnit průměrně alespoň 2/3 požadavků.

Navíc se králi doneslo, že v sousedním království, kde měli podobný problém, si královští synové dokázali naklást takové požadavky, že z nich opravdu více než 2/3 splnit nešly. Tajná služba uvedla, že tyto požadavky byly natolik složitě formulované, že je není vhodné králi detailně prezentovat, a on se s tím spokojil – tedy i vy budete muset.

Zbývá vyřešit to, že ani k tomuto řešení král nehodlá propůjčit svou korunu. Chtěli bychom se tedy obejít bez generátoru náhodných čísel.

Nejprve si rozmysleme, jak přesně spočítáme, kolik požadavků v průměru splníme: Označme si p_x pravděpodobnost, že x bude `true`. Pak pravděpodobnost, že splníme požadavek

- (x) je p_x.
- (x `or` y) je $1-p_x p_y$.
- (x `or` `not` y) je $1-p_x(1-p_y)$.
- (`not` x `or` y) je $1-(1-p_x)p_y$.
- (`not` x `or` `not` y) je $1-(1-p_x)(1-p_y)$.

Když všechny tyto pravděpodobnosti sečteme, dostaneme hledaný průměrný počet splněných požadavků, označme si ho P.

Nyní řekněme, že bychom si napevno zvolili x jako `true` (tedy položili $p_x = 1$) a ostatní proměnné volili stále náhodně se stejnými pravděpodobnostmi. Stejným postupem si spočítáme průměrný počet splněných požadavků v tomto případě a označíme si ho $P_{\texttt{true}}$. Analogicky, pokud si napevno zvolíme x jako `false`, v průměru splníme $P_{\texttt{false}}$. Nyní nahlédneme, že $P = p_x \cdot P_{\texttt{true}} + (1 - p_x) \cdot P_{\texttt{false}}$. Řekněme, že volbu, které provincie dostane který syn, provádíme postupně počínajíce provincií x. S pravděpodobností p_x se rozhodneme dát provincii x Petrovi, v tom případě splníme průměrně $P_{\texttt{true}}$ požadavků. S pravděpodobností $(1 - p_x)$ ji naopak dáme Pavlovi a splníme průměrně $P_{\texttt{false}}$ požadavků. Dohromady tedy průměrně splníme $p_x \cdot P_{\texttt{true}} + (1 - p_x) \cdot P_{\texttt{false}}$ požadavků, což musí být rovno P.

P je tedy vážený průměr $P_{\texttt{true}}$ a $P_{\texttt{false}}$, jedno z $P_{\texttt{true}}$ a $P_{\texttt{false}}$ tedy musí být alespoň tak velké jako P, protože kdyby byly obě menší než P, i jejich průměr by musel být menší než P. To nám říká, komu přidělit provincii x – pokud je $P_{\texttt{true}} \geq P$, dáme ji Petrovi, jinak Pavlovi. Nyní si za x do všech požadavků dosadíme zvolenou hodnotu a tento postup opakujeme s takto modifikovanými požadavky pro další proměnnou.

Algoritmus bude tedy pracovat takto: Na začátku nastavíme p_x pro všechny proměnné podle popsaného postupu (buď 1/2 nebo 2/3). Poté spočteme P (součet pravděpodobností splnění všech požadavků). Nyní vybereme jednu proměnnou x, které jsme ještě nedali žádnou hodnotu, a spočteme $P_{\texttt{true}}$ a $P_{\texttt{false}}$. Tyto hodnoty se počítají jako P, jen jednou použijeme $p_x = \texttt{true}$ a jednou $p_x = \texttt{false}$. Víme, že buď $P_{\texttt{true}}$ nebo $P_{\texttt{false}}$ je větší než P. Pokud tedy $P_{\texttt{true}} > P_x$, dáme proměnné x hodnotu `true` a $p_x = 1$. Jinak položme $x = \texttt{false}$ a $p_x = 0$. Se změněnou hodnotou p_x přepočítáme P, vybereme jinou proměnnou, které jsme ještě nedali hodnotu, a tu zpracujeme stejným způsobem. Algoritmus skončí, pokud jsme již všem proměnným přidělili nějaké hodnoty, čili jsme rozdělili všechny provincie.

V celém tomto postupu si nikde nepotřebujeme volit žádná náhodná čísla (pouze provádíme výpočty s pravděpodobnostmi), čímž jsme se vyhnuli nutnosti otloukat královskou korunu.

Časová složitost přímočaré implementace tohoto algoritmu je $O(N^2)$, kde N je počet požadavků, protože spočtení P nám trvá $O(N)$ a počítáme ho třikrát pro každou proměnnou (kterých je nanejvýš dvakrát tolik, co požadavků). Ovšem pokud si všimneme toho, že při změně pouze jedné p_x se P „moc“ nezmění, a trochu se zamyslíme, můžeme časovou složitost snížit na $O(N)$. Paměťovou složitost dokážeme také, při troše snahy, udržet na $O(N)$.

16-3-1 Fyzikova blecha — Tomáš Vyskočil a MS

Tak jak to všechno dopadlo... U některých řešení by se blecha urazila nebo dokonce přímo unudila, než by dostala řešení. Tím myslím především algoritmy s exponenciální časovou složitostí, už daleko lepší byly kvadratické a nejlepší algoritmy byly se složitostí $O(N \log N)$. A takový si zde ukážeme.

Jak tento bleší problém vyřešíme? Začneme tím, že si plošinky utřídíme podle y-ové souřadnice. Předpokládejme, že u konců každé plošinky víme, na jakou jinou plošinku z tohoto konce blecha spadne. Budeme probírat plošinky podle stoupající y-ové souřadnice a u každé plošinky si budeme u obou konců počítat nejkratší cestu na podlahu. To provedeme tak, že zkusíme ze zpracovávaného konce plošinky spadnout na nižší plošinku (víme na kterou). Protože plošinka, na kterou dopadneme, je níž než zpracovávaná, už u ní známe nejkratší cestu z obou konců – vybereme si, zda jít doleva nebo doprava, aby byla cesta co nejkratší.

Celý tento postup zvládneme v čase $O(N)$, protože u každé plošinky uděláme jen konstantně mnoho operací (zjistíme na kterou plošinku spadneme, jak bude dlouhá cesta když po dopadu zahneme nalevo, jak bude dlouhá cesta když po dopadu zahneme napravo, vybereme minimum).

Jak tedy budeme u plošinky určovat, na jakou nižší blecha z jejího konce spadne? Použijeme k tomu *statický intervalový strom*. To je struktura, která si pro každý prvek s indexem 1 až P pamatuje nějaké číslo, přičemž P musí být pevné po celou dobu běhu programu. Intervalový strom umí dvě operace: *zjisti hodnotu prvku* i a *nastav hodnotu prvků* v intervalu $i \dots j$ na *co*, obě v čase $O(\log N)$.

Předpokládejme, že už takovou strukturu známe. Použijeme ji tímto způsobem: Jednotlivé prvky intervalového stromu budou použité x-ové souřadnice plošinek (je jich nanejvýš $2N$) a hodnota prvku i (i-tá nejmenší x-ová souřadnice) je číslo nejvýše umístěné plošinky, která se na této x-ové souřadnici vyskytuje. Abychom mohli x-ové souřadnice očíslovat, musíme si je za začátku opět setřídit.

Na začátku dáme do intervalového stromu jen podlahu. Probereme si plošinky opět podle vzrůstající y-ové souřadnice a u každého konce (souřadnice $left_x, right_x$; předpokládejme, že po očíslování mají indexy $\mathit{left}_i, right_i$) se intervalového stromu zeptáme, jaká je hodnota prvku left_i a $right_i$ (0 znamená podlaha, jiné číslo je pořadové číslo plošinky). Tím jsme zjistili, na kterou plošinku spadne blecha z levého a pravého konce plošinky. Poté zpracovávanou plošinku „přidáme" do intervalového stromu, čili (pokud zpracováváme i-ou odspoda) do intervalového stromu zapíšeme hodnotu i do prvků v intervalu $\mathit{left}_i \dots right_i$.

Pokud tedy zvládneme naimplementovat popsaný intervalový strom, máme řešení s časovou složitostí $O(N \log N)$, protože třídění nás stojí $O(N \log N)$

a dále zpracováváme N plošinek a každou v čase $O(\log N)$. Paměťová složitost je jako obvykle $O(N)$.

Statický intervalový strom si můžeme představit jako dokonale vyvážený binární strom. Jednotlivé vrcholy odpovídají intervalům z rozmezí 1 až P tak, že listy tohoto stromu jsou jednotlivé prvky (odpovídají intervalům $i \ldots i$) a každý vnitřní vrchol odpovídá intervalu, který je roven sjednocení intervalů synů tohoto vrcholu. Čili vrchol celého stromu odpovídá intervalu $1 \ldots P$, jeho levý syn intervalu $1 \ldots \lfloor P/2 \rfloor$ a pravý syn intervalu $(\lfloor P/2 \rfloor + 1) \ldots P$.

U každého vrcholu si budeme pamatovat jednak hodnotu h_i a jednak informaci p_i, zda hodnota h_i odpovídá všem prvkům na intervalu, který tento vrchol reprezentuje (u listů je to vždy *true*). Zjištění hodnoty nějakého prvku potom provedeme následovně: začneme ve vrcholu. Pokud je p_v *true*, vrátíme hodnotu h_v. Jinak si vybereme levého nebo pravého syna (podle indexu prvku, jehož hodnotu zjišťujeme), a rekurzíme (určitě se zastavíme, listy mají p_i na *true*).

Jak dopadne nastavení hodnoty prvků na intervalu $i \ldots j$? Opět začneme ve vrcholu. Pokud interval, který zkoumaný vrchol v pokrývá, je podinterval $i \ldots j$, nastavíme p_v na *true* a h_v na nastavovanou hodnotu. Jinak se spustíme na toho syna (případně na oba), jehož interval má neprázdný průnik s intervalem $i \ldots j$. (*Pozor:* Bylo-li p_v *true*, je třeba nejprve rozdělit vrcholem reprezentovaný interval synům.)

Protože strom je dokonale vyvážený, má logaritmickou výšku. Obě operace závisí na výšce stromu (u nastavování intervalu je si to třeba rozmyslet - někdy se sice spustíme na pravého i levého syna, ale když to nastane, jednoho syna pokryjeme celého – nebudeme se z něj spouštět níže), mají tedy logaritmickou složitost.

Zbývá vyřešit jedinou drobnost – jak si rozumně vytvořit takový binární strom? Použijeme pro to pole o velikosti $2N$. Řekneme, že prvek s indexem 1 odpovídá intervalu $1 \ldots P$. Dále řekneme, že synové prvku i pokrývajícího interval $\mathit{left}_i \ldots \mathit{right}_i$ jsou $2i$ levý a $2i+1$ pravý. Levý syn bude pokrývat interval $\mathit{left}_i \ldots \lfloor \mathit{left}_i + \mathit{right}_i)/2 \rfloor$, pravý syn bude pokrývat interval $(\lfloor \mathit{left}_i + \mathit{right}_i)/2 \rfloor + 1) \ldots \mathit{right}_i$. Tímto trikem můžeme chápat pole jako strom, přičemž jednotlivé intervaly, které vrcholy pokrývají, si počítáme až při průchodu tímto stromem/polem.

```
#include <stdio.h>

#define MAX 1024
#define INF 0xffffff

int n, xb, yb, v, o[2*MAX], value[4*MAX];    /* value - hodnoty int. stromu */
int p[4*MAX],                                /* p - vrchol int. stromu je pokryt */

int cmp (const void *a, const void *b)       /* porovnání dle y */
{ return ( (int *)a)[2] - ( (int *)b)[2]; }
```

```
int cmp2 (const void *a, const void *b)            /* a dle x */
{ return * (int *)a − * (int *)b; }

int find (int node, int lnode, int rnode, int x)   /* najdi nejbližší nižší plošinku */
{
    int m = (lnode + rnode) / 2;

    if (p[node])                                   /* máme koncový vrchol */
        return value[node];
    if (x <= o[m])                                 /* jinak jdeme vlevo nebo vpravo */
        return find (2 * node, lnode, m, x);
    return find (2 * node + 1, m+1, rnode, x);
}

void add (int node, int lnode, int rnode, int from, int to, int index)
                                                   /* přidej plošinku do int. stromu */
{
    int m;

    if (o[lnode] >= from && o[rnode] <= to){
        p[node] = 1;
        value[node] = index;
    } else {
        if (p[node] && lnode < rnode){             /* rozdělíme interval na dva menší */
            p[2 * node] = p[2 * node + 1] = 1, p[node] = 0;
            value[2 * node] = value[2 * node + 1] = value[node];
        }
        m = (lnode + rnode) / 2;                   /* a pokryjeme zbytek */
        if (from <= o[m]) add (2 * node, lnode, m, from, to, index);
        if (o[m] < to) add (2 * node, m + 1, rnode, from, to, index);
    }
}

int main (void)
{
    int i, on, j, k, d[MAX][3], lpred[MAX], rpred[MAX], lsum[MAX], rsum[MAX];
    int left, right, maxi, max, result[MAX];
    bzero (o, MAX*sizeof (int));

    scanf ("%d", &n);                              /* načítáme */
    scanf ("%d %d %d", &xb, &yb, &v);
    for (i=0; i<n; i++){
        scanf ("%d %d %d", &d[i][0], &d[i][2], &d[i][1]);
        d[i][1] += d[i][0];
    }
    d[n][0] = −INF, d[n][1] = INF, d[n++][2] = 0;

    qsort (d, n, sizeof (int)*3, cmp);             /* třídíme podle výšky */
    for (i=0; i<2*n; i++) o[2*i] = d[i][0], o[2*i+1] = d[i][1];
    qsort (o, 2*n, sizeof (int), cmp2);            /* třídíme si x-ovou osu */
    i=0, j=0;
```

```
while (i < 2*n){
   while (i < 2*n && o[i] == o[i+1])
      i++;
   o[j++] = o[i++];
}
on = j;                                   /* počet různých x-ových hodnot */

for (k=1; on<k; k*=2);
for (i=1; i<k; i++) p[i] = 0;             /* p-čka vnitřních vrcholů 0 */
for (i=k; i<2*k; i++) { p[i] = 1; value[i] = 0; }   /* p-čka listů 1 */

add (1, 1, on, d[0][0], d[0][1], 0);
for (i=1; i<n; i++){                      /* na které plošinky dopadneme? */
   lpred[i] = find (1, 0, on, d[i][0]);
   if (d[i][2] - d[lpred[i]][2] > v) lpred[i] = n;
   rpred[i] = find (1, 0, n, d[i][1]);
   if (d[i][2] - d[rpred[i]][2] > v) lpred[i] = n;
   add (1, 0, on, d[i][0], d[i][1], i);
}
lsum[n] = rsum[n] = INF;
lsum[0] = rsum[0] = 0;
for (i=1; i<n; i++){                      /* hledáme nejkratší cestu */
   left = lpred[i], right = rpred[i];
   if (left && lsum[left] + abs (d[i][0] - d[left][0]) < rsum[left] + abs (d[i][0] -
         d[left][1]))
      lsum[i] = lsum[left] + abs (d[i][0] - d[left][0]);
   else if (left)
      lsum[i] = rsum[left] + abs (d[i][0] - d[left][1]);
   else lsum[i] = 0;

   right = lpred[i], right = rpred[i];
   if (right && lsum[right] + abs (d[i][0] - d[right][0]) > rsum[right] + abs (d[i][0] -
         d[right][1]))
      rsum[i] = lsum[right] + abs (d[i][0] - d[right][0]);
   else if (right)
      rsum[i] = rsum[right] + abs (d[i][0] - d[right][1]);
   else rsum[i] = 0;
}
for (max=0, maxi=0, i=0; i<n; i++){   /* najdeme první plošinku */
   if (d[i][2] > max && yb >= d[i][2] && d[i][0] < xb && d[i][1] > xb){
      max = d[i][2];
      maxi = i;
   }
}
i = maxi, j = 0;                          /* a vysledujeme zpáteční cestu */
while (!lpred[i] && !rpred[i] && i != n){
   if (xb - lsum[maxi] < rsum[maxi] - xb){
      i = lpred[maxi];
      result[j] = 'l';
   } else {
```

```
      i = rpred[maxi];
      result[j] = 'r';
    }
    j++;
  }
  if (i == n) printf ("Chudak blecha asi se nam urazi!\n");
  else {
    for (i=0; i<j; i++){
      printf ("%c\n", result[j]);
    }
    if (xb − lsum[maxi] < rsum[maxi] − xb)
      printf ("%d\n", xb+lsum[maxi]);
    else
      printf ("%d\n", xb+rsum[maxi]);
  }

  return 0;
}
```

16-3-2 Historikova past — Tomáš Valla

Nejprve se zamysleme nad tím, jak bychom úlohu řešili, kdyby v bludišti nepřekážel Minotauros. V tomto jednoduchém případě potřebujeme pouze najít nejkratší cestu ze startu k cíli.

Využijeme při tom algoritmus *vlny.* Představme si, že v prvním kroku vylijeme vodu na startovní políčko, což si na něm poznačíme například číslem 1. V druhém kroku nám voda přeteče do políček sousedících se startovním (samozřejmě těch, kde není zeď) a to si na nich poznačíme číslem 2. A tak dále, obecně v i-tém kroku označíme číslem i všechny dosud nezaplavené sousedy políček s číslem $i-1$. Zjevně číslo přiřazené políčku udává délku nejkratší cesty do něj. Samotnou cestu vypíšeme zpětným průchodem od cíle tak, že z políčka s číslem j popojdeme do libovolného souseda s číslem $j-1$, případně si můžeme pamatovat matici předchůdců, odkud do políčka voda natekla.

Vlastně se probíráme grafem, kde vrcholy (stavy bloudění) jsou všechny možné přípustné polohy Thesea, hrana mezi dvěma vrcholy vede pokud pozice v bludišti sousedí a hledáme v něm nejkratší cestu ze startu do cíle. Jak podobný přístup použít i za přítomnosti žravého Minotaura?

Stav bloudění je tentokrát určen jak polohou Thesea, tak polohou Minotaura. Všechny přípustné stavy jsou tedy dvojice (poloha Thesea, poloha Minotaura), kde ani jeden zrovna nestojí ve zdi a Minotaurus nežere Thesea. Jakmile se Theseus pohne, umíme jednoznačně určit v čase $O(k)$, jak na to zareaguje Minotaurus. Opět tedy můžeme hledat nejkratší cestu v grafu, jehož vrcholy budou všechny stavy bloudění a hrana povede z (T, M) do (T', M'), pokud T' je sousedem T a Minotaurus na přesun Thesea do polohy T' zareaguje

posunem z M na M'. Na tento graf (stavový prostor), kde cílové stavy jsou takové, ve kterých Theseus stojí v cíli, stačí pouze vypustit algoritmus vlny. (Graf si samozřejmě nemusíme pamatovat žádným z populárních způsobů uchovávání grafu v paměti, přecházet mezi vrcholy umíme jednoduše i bez konstrukce hran.)

Zbývá si pouze rozmyslet, jak vlnu efektivně naprogramovat. Zavedeme si frontu na vrcholy, ze který se bude rozlévat voda. Vždy vyzvedneme prvek ze začátku fronty, zaplavíme jeho dosud nezaplavené sousedy a zařadíme je na konec fronty. Tím si zaručíme, že na každý stav (kterých je nyní nejvýše $(NM)^2$) při vlně sáhneme jen konstantněkrát. Přechod mezi dvěma stavy umíme vypočítat v čase $O(k)$, zpětný výpis zvládneme v čase lineárním k počtu stavů, celkový čas výpočtu tudíž bude $O(N^2M^2k)$. Potřebujeme si zapamatovat matici $N \times M$ s bludištěm, stavový prostor velikosti $(NM)^2$) reprezentovaný čtyřrozměrným polem, kam si ukládáme čas zaplavení, stejně mnoho předchůdců pro výpis cesty a opět stejně velkou frontu. Celková spotřebovaná paměť tedy bude $O(N^2M^2)$.

```
#include <stdio.h>
#include <ctype.h>

#define INFTY 999999
#define MAX 20

struct state {
        int tx, ty;                          /* Theseus */
        int mx, my;                          /* Minotaurus */
};

struct state queue[MAX*MAX*MAX*MAX];
int queue_head, queue_tail;
int N, M, K;
int Ex, Ey;                                  /* cíl */
int B[MAX][MAX];                             /* bludiště */
int D[MAX][MAX][MAX][MAX];                   /* stavový prostor */
struct state P[MAX][MAX][MAX][MAX];          /* předchůdci */

struct state m_move (struct state s)         /* pohni s Minotaurem a vrat nový stav */
{
        int i;
        for (i=0; i<K; i++) {
                if (s.my > s.ty && B[s.mx][s.my-1] == '.') s.my--;
                if (s.my < s.ty && B[s.mx][s.my+1] == '.') s.my++;
                if (s.mx > s.tx && B[s.mx-1][s.my] == '.') s.mx--;
                if (s.mx < s.tx && B[s.mx+1][s.my] == '.') s.mx++;
        }
        return s;
}

void add (struct state s, int step, int dx, int dy)   /* zařaď "rozlévací" stav do fronty */
{
        struct state t = s;
```

```
        t.tx += dx;
        t.ty += dy;
        if (t.tx>=0 && t.tx<N && t.ty>=0 && t.ty<M && B[t.tx][t.ty] == '.'&&
            D[t.tx][t.ty][t.mx][t.my] > step) {
                queue[queue_tail++] = t;
                D[t.tx][t.ty][t.mx][t.my] = step;
                P[t.tx][t.ty][t.mx][t.my] = s;
        }
}
int main (void)
{
        int i, j, x, y, c;
        struct state s;
        scanf ("%d %d %d", &N, &M, &K);
        scanf ("%d %d %d %d %d %d",     /* souřadnice Thesea, Minotaura a východu
                                           */
          &s.tx, &s.ty, &s.mx, &s.my, &Ex, &Ey);
        for (i=0; i<N; i++) for (j=0; j<M; j++) {
                while (isspace (c=getchar ()));
                B[i][j] = c;
                for (x=0; x<N; x++) for (y=0; y<M; y++)
                        D[i][j][x][y] = INFTY;
        }
        queue[0] = s;                   /* počáteční stav zařaď do fronty */
        queue_tail = 1;
        D[s.tx][s.ty][s.mx][s.my] = 0;

        while (queue_head < queue_tail) {
                s = queue[queue_head++];
                j = D[s.tx][s.ty][s.mx][s.my] + 1;
                s = m_move (s);
                if (s.mx == s.tx && s.my == s.ty)   /* unhappy end */
                        continue;
                if (s.tx == Ex && s.ty == Ey)   /* happy end */
                        break;
                add (s, j, -1, 0);
                add (s, j, 1, 0);
                add (s, j, 0, -1);
                add (s, j, 0, 1);
        }
        if (queue_head < queue_tail) {        /* happy end */
                i = D[s.tx][s.ty][s.mx][s.my] + 1;
                printf ("Cesta nalezena (v opacnem poradi):\n");
                while (i--) {
                        printf ("(%d,%d)\n", s.tx, s.ty);
                        s = P[s.tx][s.ty][s.mx][s.my];
                }
        } else printf ("Cesta neexistuje\n");
        return 0;
}
```

16-3-3 Genetikova evoluce — Pavel Šanda

Většina z vás správně uhodla, že hledaná evoluce je vlastně minimální kostrou grafu, jehož vrcholy jsou jednotlivé druhy, hrany mezi nimi možné evoluční skoky a ohodnocení hran (vzdálenost mezi vrcholy) odpovídá počtu mutací mezi jednotlivými druhy.

Stačilo pak opsat kuchařku a řešení mžouralo na svět. Žel, jak při rodinné večeři u Blátošlapů vyšlo najevo, nikoliv optimální. Hned poté, co se Blátošlapovi podařilo umlčet nejmladšího Blátomarcka, (jsa genetikou dosud neinfikován, neustále cosi drmolil o povstávání z bláta), musel zažehnávat hádku s pubertálním vzdorem Blátoťapky - jeho jedinou zábavou toho roku bylo neustále rozvracet genetikův svatý svět (jde přeci o vývoj toho, kdo text čte, nikoliv jen o vývoj textu samotného. . .). Nejstarší Blátotlačka byla jediná, s kým byla ten den rozumná řeč - a jaká ! Ihned si všimla dvou detailů.

Časová složitost v kuchařce je tak veliká, hlavně kvůli setřídění všech hran a nenápadná zmínka o maximálním počtu sledovaných znaků, ji přivedla na myšlenku třídících algoritmů, které za určitých podmínek pracují rychleji (RadixSort, CountedSort, atd). Při troše péče se pak nechá časová složitost zmenšit až na kvadratickou vzhledem k počtu vrcholů.

Druhý podstatný detail - nepracujeme s obecným grafem, ale jen se speciální podtřídou grafů, kde je každý vrchol spojen s každým, čili počet hran je vždy $O(n^2)$. Díky tomu si při použití DFU můžeme dovolit investovat do slití dvou tříd čas $O(n)$ (je to strom, slévat budu n-1 krát), tak abychom byli schopni zjistit reprezentanta třídy vždy v $O(1)$ - nepotřebujeme pak žádné zrychlovací finty, které zachrání amortizovaný čas. Tím se můžeme vyhnout nabobtnalým zdrojákům, ve kterých se vrší chyba na chybě.

A jaké bude naše řešení: víme, že hran je $O(n^2)$, takže je nepravděpodobné, že se někdy dostaneme pod tuto složitost. Nenecháme se zmást narafičenou kuchařkou a vzpomeneme/vyhledáme Prim-Jarníkův algoritmus, který lze implementovat v $O(n^2)$.

Jaká je jeho myšlenka: Budeme kostru budovat postupně, podle vrcholů (nikoliv hran). V každém kroku bude budovaný podgraf G souvislý a nebude obsahovat kružnice, tj. bude to strom. Z toho plyne, že po přidání posledního vrcholu budeme mít v rukou kostru původního grafu.

Indukční krok: Další vrchol vybírám z množiny sousedů budovaného podgrafu (tj. takové vrcholy, které sousedí v původním grafu s G, ale dosud do G nebyly zařazeny). Z této množiny vyberu ten, který je „nejbližší“ ke G, tj. vyberu vrchol, který v daném kroku zvýší celkový součet mutací v G minimálním možným způsobem.

V našem případě začneme budovat G vždy od vrcholu, který reprezentuje prvotního předka v evoluci, čímž zaručíme správné pořadí otec $\rightarrow$ syn při generování evoluce.

To, že je nalezená kostra vskutku minimální by se dokazovalo sporem, a zvídavější povahy nechť hledají např. letošní MOP, kde je důkaz proveden se vší parádou.

Implementační detaily: budeme zařazovat celkově n vrcholů, tedy pro každé zařazení smím spotřebovat maximálně $O(n)$ času. Zavedu si pole min, které pro každý nezařazený vrchol, uchovává nejbližší zařazený(tj.již v podgrafu G). Vyhledání minima v tomto poli jistě v $O(n)$ zvládnu. Aktualizaci pole po zařazení, znamená zkontrolovat, nemá-li nějaký nezařazený vrchol blíže k právě zařazenému - a to v $O(n)$ zvládnu také.

Zjištění velikosti mutace mezi dvěma druhy ($bits$), je ekvivalentní zjišťování počtu jedniček v XORu znaků jednotlivých druhů. V našem řešení lineární vzhledem k počtu znaků - těch je ovšem konstantně mnoho, i ono proto trvá konstantní čas (existuje však také algoritmus logaritmický vzhledem k počtu znaků). Zjišťování vzdálenosti mezi dvěma vrcholy se bude vícekrát opakovat, takže by bylo možné výpočet urychlit zavedením pomocného pole pro spočtené výsledky. Odnesli bychom to však zhoršením paměťové složitosti ze současných $O(n)$ na $O(n^2)$.

```
#include <values.h>
#include <stdio.h>
#define M 30                                /* Max. vrcholu */

int used[M], min[M], gens[M];               /* v kostře?;nejbližší vrch. v kostře;vstup */
int bits (int x, int y){
  int b=0, i, c=gens[x]^gens[y];
  for (i=0; i<30; i++) b+=c%2, c/=2;
  return b;
}                                           /* bits */

int main (){
  int j, i, b, N, mutat=0;                  /* N=vrcholy; # mutací */
  int minI, minV=MAXINT;                    /* index nejbližšího vrch. min edge val */
  used[0]=1;                                /* root of tree */
  scanf ("%d", &N); for (i=0; i<N; i++) scanf ("%d", &gens[i]);

  for (j=0; j<N-1; j++) {                   /* kostra má n − 1 hran */
   for (i=0, minV=MAXINT; i<N; i++)         /* nejbližší vrchol */
      if (!used[i]) b=bits (i, min[i]),  minV= b<minV ?minI=i, b:minV;

   used[minI]=1; mutat+=bits (minI, min[minI]);    /* přidej ho */
   printf ("%d->%d;", gens[min[minI]], gens[minI]);

   for (i=0; i<N; i++)                      /* uprav seznam nejbližších vrch. */
      if (!used[i]) if (bits (i, minI)<bits (i, min[i])) min[i]=minI;
   }                                        /* for j */

  printf ("\nCelkove %d mutaci\n", mutat);
}                                           /* main */
```

16-3-4 Ekonomova odměna — Milan Straka

Většina z Vás sice neodsoudila Dlouhoprsta k smrti, ale zato ho odsoudila k chudobě. A ta je pro zloděje snad ještě horší než smrt. Avšak ne nadarmo se Dlouhoprst zove Dlouhoprstem – brzy přišel na to, jak piráty ožebračit.

Nejprve předpokládejme, že máme stříbrňáků nepočítaně, a zkusme zjistit, kolik nejméně jich potřebujeme k tomu, aby byl Dlouhoprstův návrh přijat. První důležité pozorování je to, že je-li $P \neq 1$, tak Dlouhoprst přežije. Pokud Dlouhoprst zaplatí dost, může na svou stranu získat všechny piráty kromě toho s číslem P.

Nejprve si ukažme, jak bude situace vypadat pro malá P:

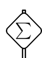
P							
0	S						
1	S	-1					
2	0	0	S				
3	1	1	0	$S-2$			
4	0	2	1	0	$S-3$	nebo	
4	2	0	1	0	$S-3$		
5	0	2	2	1	0	$S-5$	nebo
5	2	0	2	1	0	$S-5$	nebo
5	2	2	0	1	0	$S-5$	

Pozn.: Poslední číslo na řádku je Dlouhoprstův zisk, –1 = †.

Důležité je to, proč pirátům $1, 2$ stačí v případě $P = 5$ pouze 2 stříbrňáky. Vzhledem k tomu, že situace může v případě $P = 4$ dopadnout dvěma způsoby (buď dostane zaplaceno 1. nebo 2. pirát), nemají tito piráti **jistotu**, že ony 2 stříbrňáky získají. Proto se při $P = 5$ spokojí s dvěma stříbrňáky (ovšem jeden by jim nestačil - mají **naději** na dva).

Pokud budeme pokračovat v rozboru dále, zjistíme, že kromě počátačních případů ($P \leq 3$) je optimální řešení takovéhle: chceme, aby pro Dlouhoprsta hlasovalo $\lceil P/2 \rceil$ pirátů. Pirátovi P nedáme nic, pirátovi $P-1$ dáme 1 stříbrňák a ze zbylých pirátů $1 \ldots P-2$ vybereme libovolných $\lceil P/2 \rceil - 1$ a dáme každému z nich 2 stříbrňáky (pro jednoduchost vybereme např. prvních $\lceil P/2 \rceil - 1$ pirátů). Celkem použijeme $2\,(\lceil P/2 \rceil - 1) + 1 = 2\lceil P/2 \rceil - 1$ stříbrňáků.

Pokud je tedy $S \geq 2\lceil P/2 \rceil - 1$, umíme úlohu vyřešit dokonce v konstantním čase a prostoru. Případ $P \leq 3$ vyřešíme tabulkou a pro ostatní hodnoty P použijeme popsané rozdělení (pozor na to, že nesmíme vypsat počet stříbrňáků pro každého piráta – to by mělo složitost $O(P)$).



Situace začne být složitější, je-li S menší než $2\lceil P/2 \rceil - 1$. Není totiž pravda, že pak nikdy neexistuje řešení. Např:

S = 1 P									
0	1								
1	1	−1							
2	0	0	1						
3	0	0	1	−1					
4	0	1	0	0	0				
5	0	1	0	0	0	−1			
6	0	1	0	0	0	−1	−1		
7	0	1	0	0	0	−1	−1	−1	
8	0	1	0	0	0	0	0	0	0

Ačkoli se piráti dělí o jediný stříbrňák, pokud je jich 8, Dlouhoprst může přežít, protože pro něj zadarmo budou hlasovat piráti $6, 7, 8$, kteří jinak zemřou.

Řešení situace, kdy je S malé, založíme na dynamickém programování. Budeme postupně řešit situaci pro rostoucí P až do zadaného. U každého piráta si budeme pamatovat, jaký počet stříbrňáků má „jistý" a zda má šanci získat víc než svou „jistotu". Všimněte si, že dále popsaný postup funguje pro libovolná S, mohli bychom tak řešit (byť pomaleji) i případ dostatečně velkého S.

Řekněme, že známe řešení pro $P-1$ pirátů a zajímá nás řešení situace s P piráty. Hledáme $\lceil P/2 \rceil$ pirátů, kterým zaplatíme nejméně. Pokud budeme chtít, aby pro Dlouhoprsta hlasovali všichni piráti, kteří mají jistých R stříbrňáků, nabídneme jim $R+1$ stříbrňáků a upravíme jejich „jistotu". Pokud budeme chtít, aby hlasovali jenom někteří piráti z těch s „jistotou" R stříbrňáků, nabídneme jim taktéž $R+1$ stříbrňáků. Nezměníme ovšem jejich „jistotu", jenom si u nich poznamenáme, že mají šanci získat více než svou „jistotu".

Nyní si všimneme důležité věci: žádný pirát (kromě toho, který právě navrhuje rozdělení) nemůže dostat více než dva stříbrňáky, neboli „jistota" každého piráta je menší než dva. To platí proto, že aby měl nějaký pirát jisté dva stříbrňáky, musel by někdy Dlouhoprst nabídnout dva stříbrňáky všem pirátům, kteří měli jistý jeden. Uvažujme nyní situaci, kdy Dlouhoprst poprvé nabízí dva stříbrňáky všem pirátům, kteří mají jistý jeden. V tuto chvíli jsou ještě „jistoty" všech pirátů menší než dva. Když už by ovšem Dlouhoprst uplácel piráty s jedním jistým stříbrňákem, piráty s nižší „jistotou" by už museli být uplaceni (jsou totiž levnější). Tedy piráti s „jistotou" menší než jedna jsou již uplaceni, všichni piráti s „jistotou" rovnou jedné jsou upláceni a žádní jiní piráti neexistují. Čili Dlouhoprst by uplácel **všechny** piráty! To je ovšem spor s tím, že Dlouhoprst chce uplatit jen $\lceil P/2 \rceil$ pirátů.

Pokud tedy zjišťujeme řešení situace s P piráty ze situace s $P-1$ piráty, nejprve zjistíme, kterým pirátům bude Dlouhoprst platit. Vzhledem k tomu, že „jistoty" všech pirátů jsou menší než dva, dokážeme to v konstantním čase (pokud si pamatujeme, kolik pirátů má „jistotu" rovnu -1 (smrt), kolik jí má

nulovou a kolik pirátů má jistý 1 stříbrňák). Poté můžeme jedním průchodem nad piráty $1 \ldots P$ upravit jejich „jistoty“ a šance na získání většího počtu stříbrňáků. To celé zvládneme v čase $O(P)$.

Celý algoritmus bude fungovat tak, že bude zjišťovat, jak dopadnou návrhy pro $1, 2, \ldots, P$ pirátů a poslední z těchto návrhů vypíše. Při výpisu všem pirátům, kteří nemají šanci na nic víc než jejich „jistotu“, zaplatíme. Ovšem pirátům, kteří mají naději na víc, platíme o jeden stříbrňák víc a nemusíme jim platit všem, jenom tolika, abychom uplatili celkem $\lceil P/2 \rceil$ pirátů.

Celkem P-krát opakujeme postup o složitosti $O(P)$ a poté výsledky v čase $O(P)$ vypíšeme, čili celková časová složitost popsaného algoritmu je $O(P^2)$. Paměťová složitost je $O(P)$. A ačkoliv popsaný algoritmus používáme jen v případě, kdy $S < 2\lceil P/2 \rceil - 1$, funguje korektně pro všechna S.

A pokud si všimnete chování tohoto algoritmu, může Vám běžet až v $O(P)$.

```
#include <stdio.h>
#define MAX_P 100

int coins[MAX_P];                       /* toto jsou "jistoty" pirátů */
int unsure_thing[MAX_P];                /* má pirát naději na víc než "jistotu"? */
int freqs[3];                           /* kolik pirátů s jistotami -1,0,1 existuje */
int S, P;

int main (void) {
  int i, j, paid;                       /* paid – tolik musí Dlouhoprst platit */
  int needed, have;                     /* kolik potřebuji a mám hlasů */
  int whom_to_pay;                      /* kterým pirátům platím */
  int not_paid_to;                      /* kolika pirátům chtějícím whom_to_pay
                                           jsem neplatil */

  printf ("Zadejte počet Pirátů a Stříbrňáků:");
  scanf ("%d %d", &P, &S);

  if (S >= (2* ( (P+1)/2) - 1)) {       /* S je dostatečné – známe řešení */
    switch (P) {
    case 0:printf ("Nesmysl, žádný pirát neexistuje.\n"); break;
    case 1:printf ("Dlouhoprst zemře.\n"); break;
    case 2:printf ("Návrh je nedat nikomu nic, zisk je %d.\n", S); break;
    case 3:printf ("Návrh je dát prvním dvěma pirátům 1 stříbrňák, zisk je %d.\n",
          S-2); break;
    default:printf ("Návrh je dát prvním %d pirátům dva stříbrňáky,"
"pirátovi %d jeden a ostatním nic. Zisk je %d.\n", (P+1)/2-1, P-1, S-2* ( (P+1)/2)+1);
    }
    return 0;
  }

  for (i=0; i<=P; i++) {                /* S je malé – simuluj */
    needed= (i+1)/2;
    have=paid=0;
    whom_to_pay=-1;
    while (whom_to_pay<2) {
      if (have+freqs[whom_to_pay+1]>=needed) {
        paid+= (needed-have)* (whom_to_pay+1);
```

```
            not_paid_to=have+freqs[whom_to_pay+1]−needed;
            break;
         }
         have+=freqs[whom_to_pay+1];
         paid+=freqs[whom_to_pay+1]* (whom_to_pay+1);
         whom_to_pay++;
      }
      if (paid > S) whom_to_pay=2;              /* nemám na to dost peněz */
      if (whom_to_pay<2) {                      /* šlo to uplatit */
         for (j=0; j<i; j++) {
            if (coins[j]<2) freqs[coins[j]+1]−−;
            if (coins[j]>whom_to_pay) coins[j]=unsure_thing[j]=0;
            else if (! (coins[j]==whom_to_pay && not_paid_to)) coins[j]++,
                  unsure_thing[j]=0;
            else unsure_thing[j]=1;
            if (coins[j]<2) freqs[coins[j]+1]++;
         }
         coins[i]=S−paid;
      } else {                                  /* nešlo to uplatit */
         coins[i]=−1;
      }
      if (coins[i]<2) freqs[coins[i]+1]++;
   }
   if (coins[P]==−1) puts ("Dlouhoprst přežít nemůže.");
   else {
      printf ("Dlouhoprst přežít může a vydělá si %d stříbrňáků.\nNávr je tento:", S−paid);
      not_paid_to=needed−have;
      for (i=0; i<P; i++) printf ("%d␣", (unsure_thing[i] && not_paid_to>0) ?
                        (not_paid_to−−, coins[i]+1) : (unsure_thing[i]) ?0:coins[i]);
   }
   return 0;
}
```

16-3-5 Botanikova setba — Pavel Machek

V jednorozměrném případě je úloha poměrně jednoduchá:

Udržujeme si dvě proměnné (a_x, b_x), které nám říkají odkud a kam sahá zkoumaný úsek. Pokud úrodnost ve zkoumaném úseku je větší než prozatímní největší, upravíme ji. Pokud je naopak úrodnost menší nebo rovna nule, nevyplatí se nám tento úsek použít, a je lepší na něj zapomenout (tj. $a_x = b_x + 1$). Pokud je součet prvků aspoň jedna, vždy se nám vyplatí a_x ponechat a tyto prvky přidat k naší zkoumané oblasti. Časová složitost tohoto je $O(n)$.

Nyní převedeme dvojrozměrný případ na jednorozměrný:

Projdeme všechny horní řádky a_y. Nyní budeme procházet všechny spodní řádky b_y, a zároveň si udržovat součet všech prvků ve sloupečcích v poli c. Časová složitost této části je $O(n^2)$. Na pomocné pole c je nyní možné aplikovat jednorozměrné řešení. Celková složitost tak bude $O(n^3)$.

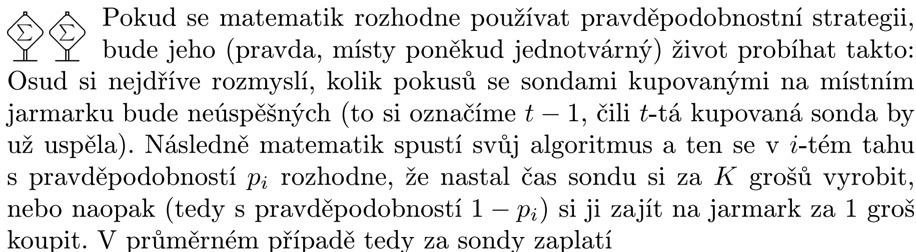
Při implementaci je trochu potřeba dát pozor na matici plnou záporných čísel, ale v této implementaci jsme to zvládli bez speciálního kódu. (Takový případ jde řešit v $O(n^2)$, ale kdo by sázel rostlinky se záporným ziskem?)

Vhodným předotočením matice je možné složitost vylepšit pro nečtvercové matice na $O(m * n * \min(m, n))$ (s díky Martinu Dobrouckému).

```
#include <stdio.h>

#define N 4
#define M 5
int p[N][M];                          /* p[y][x] */
int c[M];                             /* Součty sloupečků */
int ay, by;
int max=-100000, maxax, maxay, maxbx, maxby, maxobsah;

void
na_primce (void)
{
        int ax=0, bx, h=0;
        for (bx=0; bx<M; bx++) {
                int obsah = (bx-ax)* (by-ay);
                h+=c[bx];
                if ( (h > max) || ( (h == max) && (obsah<maxobsah))) {
                        maxax=ax; maxay=ay; maxbx=bx; maxby=by; max=h;
                        maxobsah = obsah;
                }
                if (h<=0)
                        h=0, ax=bx+1;
        }
}

int
main (void)
{
        int i;
        for (i=0; i<N*M; i++)
                scanf ("%d ", &p[0][i]);

        for (ay=0; ay<N; ay++) {
                for (i=0; i<M; i++) c[i] = 0;
                for (by=ay; by<N; by++) {
                        for (i=0; i<M; i++) c[i] += p[by][i];
                        na_primce ();
                }
        }
        printf ("Hledaná podmatice má levý horní roh v %d. sloupci, %d. řádku "
"a pravý dolní v %d. sloupci a %d. řádku.\n", maxax+1, maxay+1, maxbx+1, maxby+1);
        return 0;
}
```

16-3-6 Matematikova sonda — Martin Mareš

Kdyby náš matematik byl věděl, jaké nesnáze řešitelům KSP způsobí, asi by kosmický výzkum pověsil na hřebík a raději by pořádal přednášky o pravděpodobnosti nebo okopával na zahrádce integrály. Přišla nám totiž necelá dvě řešení. My si ukážeme celé třetí. Držte se, jedeme s kopce!



Pokud se matematik rozhodne používat pravděpodobnostní strategii, bude jeho (pravda, místy poněkud jednotvárný) život probíhat takto: Osud si nejdříve rozmyslí, kolik pokusů se sondami kupovanými na místním jarmarku bude neúspěšných (to si označíme $t-1$, čili t-tá kupovaná sonda by už uspěla). Následně matematik spustí svůj algoritmus a ten se v i-tém tahu s pravděpodobností p_i rozhodne, že nastal čas sondu si za K grošů vyrobit, nebo naopak (tedy s pravděpodobností $1-p_i$) si ji zajít na jarmark za 1 groš koupit. V průměrném případě tedy za sondy zaplatí

$$A_t = \sum_{i=1}^{t} p_i \cdot (K+i-1) + \left(1 - \sum_{i=1}^{t} p_i\right) \cdot t \qquad [\spadesuit]$$

$$= \sum_{i=1}^{t} p_i \cdot (K+i-1) + \sum_{i>t} p_i \cdot t \qquad [\heartsuit]$$

(první suma odpovídá případům, kdy si matematik koupí sondu v i-tém kroku, čili si ji $(i-1)$-krát koupí za 1, pak zaplatí K za výrobu svépomocí a pak už nic; druhá suma popisuje případ, kdy si ji vždy koupil). Optimální strategie (kterou by použil každý správný věštec) by stála $O_t = \min(t, K)$ (pokud je $t < K$, vyplatí se sondy kupovat, jinak si ji hned na začátku za K vyrobit).

Všimněte si, že každá volba čísel $p_1, p_2, p_3, \ldots$, kde $\forall i\ p_i \in \langle 0, 1\rangle$ a $\sum_i p_i = 1$, odpovídá nějaké pravděpodobnostní strategii, a naopak každá p. s. se dá nějakou takovou posloupností popsat. My tedy chceme najít taková p_i, aby relativní cena naší strategie vůči optimální (ať už je na nás Osud sebevíc zlý) – tedy $C = \max_t A_t/O_t$ – byla nejmenší možná.

Nejdříve dokážeme, že pro $i > K$ se vyplatí p_i ponechat nulové: Pokud $t > t' \geq K$, je také $A_t \geq A_{t'}$, protože při přechodu od A_t k $A_{t'}$ přejdou v $\heartsuit$ některá p_i (pro $t \geq i > t'$) z první sumy do druhé, takže se místo $K+i-1$ budou násobit t', což je číslo menší. Pokud bychom ale libovolné $p_{i>K}$ vynulovali a jeho původní hodnotu přičetli k p_K (tím zůstane $\sum_i p_i$ zachována), všechna $A_{j<K}$ zůstanou nezměněna, $A_{j\geq i}$ se sníží a A_j pro $K \leq j < i$ se sice zvýší, ale jelikož $A_j/O_j = A_j/K \leq A_i/K = A_i/O_i$, určitě si tím nepohoršíme. Takto můžeme postupně vynulovat všechna $p_{i>K}$ a získat stejně dobrou nebo dokonce lepší strategii.

Situace se tím nekonečněkrát zjednoduší – místo nekonečně mnoha p_i teď stačí najít pouze $p_1, \ldots, p_K$ a také není třeba uvažovat $t > K$, jelikož pro

taková t se jak náš, tak optimální algoritmus chovají stejně jako pro $t = K$.

Pokud má naše strategie mít relativní cenu C, musí tedy pro všechna $t \leq K$ být $A_t \leq C \cdot t$, což si rozepíšeme pomocí ♠ a místo C použijeme $c = C - 1$, aby se nám lépe počítalo:

$$A_t - t = \sum_{i=1}^{t} p_i \cdot (K + i - 1) - t \cdot \sum_{i=1}^{t} p_i \leq c \cdot t.$$

Bez újmy na obecnosti budeme předpokládat, že pro všechna t dokonce nastává rovnost (vyzkoušejte si, že kdyby nenastávala, mohli bychom p_i trochu pozměnit a buďto zvětšit počet rovností, nebo dokonce snížit c). Pokud si pak do této rovnosti dosadíte $t = 1$, vyjde, že:

$$p_1 = c/(K - 1).$$

A pokud místo toho rovnost pro $t = j + 1$ odečtete od rovnosti pro $t = j$, dozvíte se, že:

$$p_{j+1} \cdot (K + j) - (j + 1) \cdot p_{j+1} - \sum_{i=1}^{j} p_i = c,$$

čili

$$p_{j+1} \cdot (K - 1) = c + \sum_{i=1}^{j} p_i.$$

Tuto rovnost opět odečteme od sebe samé pro j-čka lišící se o jedničku a konečně dostaneme rozumný rekurentní vztah:

$$p_{j+1} = p_j \cdot \frac{K}{K - 1},$$

z nějž plyne, že:

$$p_j = p_1 \cdot \left(\frac{K}{K - 1}\right)^{j-1}.$$

Tím jsou pro každé c všechna p_i jednoznačně určena, ovšem obvykle nebudou dávat korektní rozdělení pravděpodobností. Musíme proto c nastavit tak, aby platilo:

$$1 = \sum_{i=1}^{K} p_i = \frac{c}{K - 1} \cdot \sum_{i=1}^{K} \left(\frac{K}{K - 1}\right)^{i-1} =$$

$$= \frac{c}{K - 1} \cdot \frac{\left(\frac{K}{K-1}\right)^{K} - 1}{\frac{K}{K-1} - 1} = c \cdot \left(\left(1 + \frac{1}{K - 1}\right)^{K} - 1\right).$$

Jenže jak každý správný matematik ví už od kolébky, výraz $(1+1/K)^K$ se pro rostoucí K blíží k Eulerovu číslu $e \approx 2.718281828459045$ a stejně tak $(1+1/(K-1))^K$, kterýžto výraz je dokonce i pro malá K zaručeně $\geq e$. Platí tedy, že $c \leq 1/(e-1)$, a tudíž $C = 1 + c \leq e/(e-1) \approx 1.5819767068693264$. Takže jsme našli pravděpodobnostní strategii pro pořizování kosmických sond, podstatně lepší, než je ta deterministická, a dokonce jsme dokázali, že taková strategie je nejlepší možná. (Tedy alespoň za předpokladu, že Osud a bohyně Fortuna stanovují své úradky „offline“, čili nezávisle na našich experimentech. Zkuste si rozmyslet, že nad online Osudem již neuhrajeme více než deterministický algoritmus.)

Tak končí naše malá sonda do života matematiků. A promiňte, už musím letět, právě mi přistává létající talíř výhodně pořízený od místního hokynáře…

16-4-1 Mnichova posedlost aneb Hanoi strikes back — Milan Straka

Řádění Askejťáka mnichy zmátlo natolik, že nechali nápravu pouze na Vás. Ovšem často jenom čekali a čekali a spali a čekali…Nakonec se na nekonečné posouvání disků nemohl dívat ani sám Bůhdha a rozhodl se přispět svou troškou k nápravě toho, co jeho příbuzný Kazisvět napáchal.

Začněme jednoduchým pozorováním: věž je vždy nejlepší postavit na tom kolíku, kde leží největší disk. Kdybychom ji chtěli postavit na jiném kolíku, museli bychom všech $N-1$ disků přesunout na nějaký volný kolík, pak přesunout největší disk na cílový kolík a pak na něj přesunout zbytek disků. Nicméně v tom stavu, kdy bylo všech $N-1$ disků na nějakém volném kolíku a největší disk nebyl dosud přesunutý, jsme mohli rovnou přesunout $N-1$ disků na ten největší a ušetřit tak jedno přesunutí (největšího disku).

Dále bychom potřebovali vědět, na kolik přesunutí dokážeme přemístit věž o m discích na jiný kolík. Označme tento počet $T(m)$ (evidentně $T(1)=1$). Když tedy chceme přesunout věž o m discích na jiný kolík, musíme nejprve $m-1$ disků přesunout na volný kolík, největší disk přesunout na cílový a zbylých $m-1$ disků vrátit zpátky na největší disk. Toto jednoduché pozorování se dá vyjádřit takto: $T(m) = T(m-1) + 1 + T(m-1) = 2T(m-1) + 1$. Po chvilce počítání zjistíme, že $T(m) = 2^m - 1$.

Vyzbrojení (Bůhdhou) těmito dvěma pozorováními můžeme se směle pustiti do našeho původního problému. Budeme si pro každý disk i počítat, kolik přesunů by nám zabralo postavit věž s disky $i, \ldots, N$ na každý z kolíků $1, 2, 3$. Tuto hodnotu můžeme značit $\mathit{Tahy}(i,k)$, kde k je číslo kolíku. Ještě si $K(i)$ označme, na jakém kolíku je i-tý disk.

Je jasné, že hledaná hodnota je $\mathit{Tahy}(1, K(1))$. Jak ale tyto hodnoty počítat? Pro jediný disk určitě platí, že na nějakém kolíku už je, nebo ho tam

mohu přesunout jediným tahem, tedy $Tahy(N, k = K(N)) = 1$ a $Tahy(N, k \neq K(N)) = 0$.

Pokud chceme přesunout disky od i-tého až po nejmenší (N-tý) na kolík k, můžou nastat dva případy. Buď je i-tý disk na kolíku k a pak je nutné přesunout zbylé disky na kolík k, čili $Tahy(i, k = K(i)) = Tahy(i + 1, k)$. Nebo disk i na tomto kolíku neleží a pak je nutné přesunout disky od $(i + 1)$-ního na volný kolík pom (ani k ani $K(i)$), disk i přesunout na kolík k a pak všechny disky od $(i + 1)$-ního vrátit na kolík k, čili $Tahy(i, k \neq K(i)) = Tahy(i + 1, pom) + 1 + T(N - i) = Tahy(i + 1, pom) + 2^{N-i}$.

Z toho plyne řešení s lineární časovou složitostí. Budeme počítat jednotlivé hodnoty $Tahy(i, k)$ pro $i{=}N, \ldots, 1$ a nakonec vypíšeme hodnotu $Tahy(1, K(1))$. Takto by byla paměťová složitost také lineární. Nicméně můžeme si všimnout toho, že hodnotu $Tahy(i, k)$ potřebujeme jen k výpočtu hodnot $Tahy(i - 1, k)$, takže není nutné si pole *Tahy* pamatovat celé, ale stačí nám aktuální dva řádky. Pokud by vstupní data byla čísla kolíků, na kterém leží disky $N, \ldots, 1$, bude paměťová složitost konstantní.

Ještě poznámka k došlým řešením – málokdo vzal v úvahu, že počet přesunů může být až $2^{64} - 1$ a že se tedy nejspíš nevejde do běžného 32-bitového čísla. Nicméně většina dnešních jazyků 64-bitové čísla podporuje a tak stačilo použít příslušný 64-bitový typ.

Navíc mnozí řešitelé používali při výpočtu mocniny dvojky a tak si je předpočítali do pole. To ale není vůbec třeba, protože jak v Pascalu tak v Céčku existují operace bitového posunu, který se dá použít přesně k tomuto výpočtu: $2^i = 1$ `<<` $i = 1$ `shl` i.

Ve zkratce: víme, kde má skončit největší disk. Končíme tedy tím, že přesuneme druhý největší disk (evidentně také z místa, kde byl na začátku) na první a ze zbylého kolíku pak přestěhujeme všechny zbylé disky nad druhý disk. Jinými slovy potřebujeme 2^{N-2} tahů + tahy na složení zbylých disků na zbylý kolík, což je ale ta samá úloha (oba velké disky nám v tom nemohou nijak překážet). Takže můžeme všechny disky zpracovat pozpátku v konstantním prostoru, lineárním čase a jednom řádku programu (viz dál). –M.M.

```
#include <stdio.h>
#define MAX_N 1000

int N, kolik[MAX_N];                          /* na jakém kolíku je i-tý disk */
unsigned long long min_tahu[MAX_N+1][3];      /* jak přendat disky od i-tého na lib. kol. */

int main (void) {
    int k, i, disk;

    printf ("Zadejte počet disků:");
    scanf ("%d", &N);
    for (k=0; k<3; k++) {
        printf ("Zadejte disky na kolíku %d:", k+1);
        while (scanf ("%d", &i), i) kolik[i-1]=k; }
```

```
for (disk=N−1; disk>=0; disk−−)
  for (k=0; k<3; k++)                    /* disky od „disk"-tého na k-tý kolík */
    if (kolik[disk]==k) {                /* disk je už na místě */
      min_tahu[disk][k]=min_tahu[disk+1][k];
    } else {                             /* disk není na místě */
      i=3−k−kolik[disk];                 /* pomocný kolík */
      min_tahu[disk][k]=min_tahu[disk+1][i] + (1≪ (N−disk−1));
    }
 printf ("Nejlepší to bude na kolík %d, celkem %llu přesunů.\n", kolik[0]+1,
    min_tahu[0][kolik[0]]);
 return 0;
}
```

A ještě jedna lahůdka v céčku (pouze do $2^{32} - 1$ přesunů, na vstupu jsou čísla kolíků $(0, 1, 2)$ a čtou se ze vstupu všechna):

```
int n,a,k,_;main(){scanf("%d",&a);while(scanf("%d",&k)>0)
_+=_,k!=a?_++,a=3-a-k:0;printf("%d\n",_);}
```

16-4-2 Technologické trable — Pavel Šanda

Kuchařka a počet bodů svádí k tomu, abychom na naše pole prostě napasovali binární vyhledávání a hotovo. Jenže základní problém se světem je ten, že krokodýla štěkat nenaučíš a půlením nekonečna se ke konstantě nedobereš (a tedy ani k hledané hodnotě).

Proto budeme muset nejprve vyřešit tři problémy:

a) Vzhledem k čemu měřit časovou složitost, jestliže pole ve kterém hledáme je nekonečné.

   1. Když se standardně uvažuje o časové složitosti operací na poli (orání), bere se vzhledem k jeho velikosti – to v našem případě zřejmě není použitelná cesta, navíc přímo v zadání je dáno, že pole nemáme považovat za vstup.
   2. Další možnost by byla měřit ji vzhledem k hodnotě hledaného čísla, které je skutečným vstupem, o což se někteří z vás pokusili. Žel bohu, čísla se mohou v poli opakovat, takže pro libovolně rychlý algoritmus hledání stejného nebo vyššího čísla vymyslím takové pole, že místo výsledku dostanete kopřivku.
   3. Zbývá tedy použít index N hledaného čísla x v poli, byť ze zadání nejsme schopní o jeho velikosti nic říci.

b) Žádný algoritmus v principu nemůže být konečný, neboť u nekonečného pole plného jedniček nemáme šanci zjistit, že se v něm hledaná dvojka nevyskytuje. Proto naše odhady budou platit pouze pro případ, kdy algoritmus skončí.

c) Jak naučit krokodýla štěkat. (Haf!)

A jak tedy hledat?

Jediné, co o posloupnosti čísel v poli vím, je, že je neklesající. Dále mohu dotazem zjistit konkrétní hodnotu v určité buňce. Z toho však nemohu nic usoudit o rychlosti růstu v neprobádaných oblastech – z tohoto důvodu byly všechny vaše snahy o nástřel pozice hledaného čísla na základě hodnot buněk marné (vzhledem k nejhoršímu možnému odhadu, a o ten nám jde). Zbývá nám využít monotonii – každý dotaz na buňku nám dává pouze informaci, ve které polovině pole vůči buňce se hodnota nalézá – zabývat se druhou částí pole je čiré plýtvání, neb se nic nového nedovíme.

Podtrženo sečteno, jde o to, jak se pomocí půlení intervalu co nejrychleji dostat k hledané hodnotě. Každý dotaz nám řekne, v které polovině pole hledat, což nás dovede k tomu, že první fáze algoritmu se týká nalezení stejné nebo vyšší hodnoty než je hledaná. Všimněte si, že pokud by se jednalo pouze o tuto úlohu, je otázka optimality neřešitelná – ke každému algoritmu najdu ještě rychlejší. Jenomže nám se bude stávat, že najdeme číslo, které je vyšší, a tak nastane druhá fáze – klasické binární vyhledávání v intervalu I ohraničeném posledními dvěma dotazy. Následuje další pozorování – čím rychlejší bude první fáze, tím větší budou intervaly mezi jednotlivými dotazy, a tedy i poslední interval pro druhou fázi (BÚNO předpokládám, že poslední interval není kratší než intervaly předchozí). S pomocí půlení intervalů – a nic víc dotazem na buňku nezjistím – jsem v i-tém kroku schopen zúžit prohledávaný interval I na $I/2^i$, což dává složitost $\log_2 I$ pro druhou fázi.

Nyní rozeberme složitosti pro vybrané strategie v první fázi:

1. Pokud budu hledat pravý konec intervalu přičítáním konstanty k k indexu, bude nám první fáze trvat $O(N)$, druhá fáze $O(logk) = O(1)$; celkově $O(N)$.
2. Pokud budu hledat pravý konec násobením indexu konstantou k, bude první fáze trvat $\lceil \log_k N \rceil$ a druhá fáze dostane interval délky $\leq (k-1) \cdot N$, na kterém bude hledání trvat $O(\log_2(k-1) + \log_2 N)$; celkově tedy $O(\log N)$.
3. Pokud budu hledat pravý konec v čase $O(F(N))$, kde F je asymptoticky pomalejší než $\log N$, dostane druhá fáze interval I_F a bude trvat $O(\log I_F)$, což se bude rovnat $O(\log N)$ pouze v případě, že I_F není větší než mocnina N. Příkladem strategie, kdy se interval ještě vejde do mocniny, je násobení indexu sebou samým místo konstanty $(O(\log N^2) = O(\log N))$. Příkladem strategie, kdy se do mocniny nevejdeme, budiž výpočet dalšího indexu pomocí Ackermannovy funkce – v takovém případě se nám druhá fáze obecně dostane nad logaritmus.

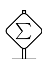
Z hlediska asymptotického chování je optimum na složitosti $O(\log N)$, šťourové zajímající se o optimalitu počtu dotazů do pole bez zanedbávání multipli-

kativních konstant mohou začít zkoumat vzájemnou závislost F a I_F.

Někteří z vás řešili, jak je to s paměťovou složitostí, jestliže budu chtít ukládat velké indexy nekonečného pole. Uvědomte si, že tento problém nastává i za normální situace, kdy je velikost pole n – zjevně s velikostí n bude muset růst i rozsah hodnot, které jsme schopni uložit v registru. Na to jsou dvě možné odpovědi:

1. Standardní: předpokládáme nezáludnost programátora a do každého registru mu dovolíme uložit libovolně velké číslo. Nezáludnost spočívá v tom, že ho nenapadne do tohoto registru začít nějak kódovat další informaci – každý program bychom pak mohli spočítat pomocí konstantní paměťové složitosti.
2. Pro záludné přestaneme počítat paměťové buňky na čísla a začneme počítat bity. Každý registr pak dostane paměťovou složitost $\log n$; zvedne se také dosud konstantní složitost na atomické operace s registry atd. Celkově se proto zvýší odhady složitostí jednotlivých algoritmů, nicméně stále nám zůstane schopnost porovnávat rychlosti jednotlivých algoritmů mezi sebou.

V implementaci je použito přímo pole; považuji-li přístupy do něj za dotaz na uživatele, je paměťová složitost konstantní; nenechte se mýlit céčkem, pole začíná od indexu 1.



Šťoura se hlásí o slovo. Nejprve si uvědomme, že Šandíkův dvojfázový algoritmus je vlastně jediný možný: položíme-li první dotaz a zjistíme, že číslo v poli je menší než hledaná hodnota, nemá se smysl dále ptát na cokoliv před ní, analogicky pro další dotazy, takže musíme jít doprava, dokud nedostaneme větší hodnotu. Ale jakmile ji dostaneme, zase víme, že hledané číslo je mezi touto hodnotou a předchozím dotazem, a tady už je optimální binární vyhledávání.

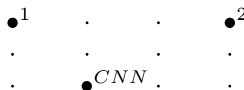
Co se optimálního počtu dotazů týče: co to vlastně znamená optimální? Libovolný algoritmus můžeme přeci pro prvních n hodnot zlepšit až na $\lfloor \log_2(n-1) \rfloor + 2$ tím, že první dotaz bude na n-tý prvek a pokud je hledaná hodnota menší, spustíme půlení intervalu, jinak přepneme na původní algoritmus, který jsme tím na ostatních prvcích o konstantu zlepšili. Naopak pokud v uvedeném algoritmu s k-násobením zvolíme větší k, bude nám první fáze trvat $\log_2 k$-krát rychleji a druhou si tím zpomalíme jen o konstantu $\log_2(k-1)$, čili jsme algoritmus zrychlili všude až na prvních několik hodnot. Analogicky místo libovolné funkce F můžeme použít funkci o konstantu pomalejší. Docházíme tak k překvapivému závěru, že ke každému algoritmu můžeme najít takový, který pro vybrané hodnoty bude o něco rychlejší. Ale na druhou stranu, alespoň $\log_2 n$ je určitě potřeba, takže dokud nás nezajímají multiplikativní konstanty, je $O(\log n)$ dozajista optimální. Haf! –M.M.

```
int main (void) {
    int l=1,  r=1,  i;                    /* left, right, index */
    int w[4]={0, 1, 2, 3};                /* nekonečné pole :-) */
    int x=2;                              /* hodnota, kterou hledáme */

    while (w[r]<x) l=r+1,  r*=2;          /* pravá zarážka */

    while (l<=r) {                        /* binární vyhledávání */
      i = (l+r)/2;
      if (w[i]==x) return i;
      if (w[i]>x) r=i-1;
      if (w[i]<x) l=i+1;
    }
    return -1;
}
```

16-4-3 Stávka programátorů — Milan Straka

Stávková reportáž televize CNN nedopadla úplně tak, jak její majitel čekal. Byla to vlastně docela katastrofa. Jedni dali řidičovi filmového vozu radu, ať chvíli počká, že to hned v momentě vyřeší (a jestli neumřeli, řeší dodnes). Jiní ho instruovali, aby si vybral vždy cestu, která je nejkratší, takže ho všichni ostatní (lépe instruovaní) řidiči předjeli a reportáž televize CNN byla odvysílání s velkým zpožděním. Je tedy nutno říci, že stávka programátorů neměla náležitý dopad (když ji televize CNN nestihla během týdne okomentovat) a tak budou muset brát programátoři dál obrovské sumy peněz za tak lehkou práci.

Většina řešení radila řidiči, aby si mezi dvěma stávkami vybral cestu buď přímo vodorovně nebo přímo svisle. A z těchto dvou tu, která je kratší. To je bohužel řešení chybné a dostalo nula bodů. Můžeme si to ukázat na následujícím protipříkladě:



Pokud si vyberete nejkratší cestu na stávku 1, pojedete o jednu ulici doleva. Pak na stávku druhou pojedete o dvě nahoru a pak o tři zpět do CNN, čili celkem šest. Ovšem pokud byste se ovšem vydali na začátku nahoru o dvě ulice, nafilmovali byste obě stávky a stačilo by se vrátit – celkem tedy pouze 4 ulice.

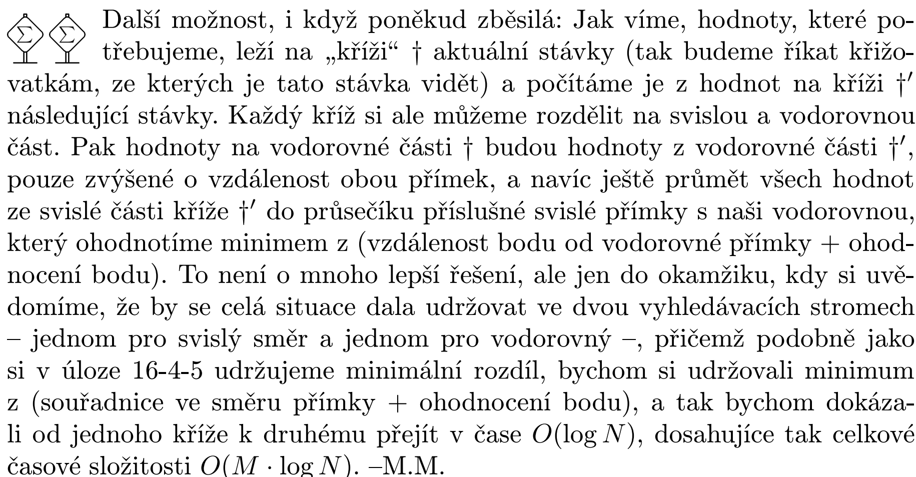
Úloha se dala řešit několika způsoby. Jeden z nich je prohledávání do šířky. Představme si ne jedno, ale M měst přesně nad sebou, přičemž v prvním je jen CNN a první stávka, v druhém je jen druhá stávka, ..., v M-tém městě je M-tá stávka a opět CNN. Pokud si představíme, že z města i se do $(i+1)$-ního dá dostat jen z ulic v městě i, ze kterých je vidět i-tá stávka, tak hledáme nejkratší cestu z CNN v prvním městě do CNN v městě M-tém. Tato nalezená cesta bude nejkratší a vzhledem k tomu, jak přecházíme mezi městy, bude u každé stávky. Toto řešení má časovou i paměťovou složitost úměrnou velikosti prohledávaného prostoru, čili $O(M \cdot N^2)$.

Jiný pohled na řešení může být tento: pro města i od M-tého k prvnímu si spočteme matici C_i, ve které budeme mít pro každou křižovatku v daném městě nejkratší vzdálenost cesty, která nafilmuje stávky $i, \ldots, M$ a vrátí se do CNN. Pokud tyto matice vzdáleností budeme počítat v popsaném pořadí od M-té k první, můžeme spočítat délku jedné nejkratší cestu (jeden prvek matice) v konstantním čase: v i-tém městě na křižovatce (x, y) vede hledaná nejkratší cesta buď horizontálně nebo vertikálně na ulici, ze které je možné nafilmovat i-tou stávku a a pak pokračuje dále (přičemž délku tohoto pokračování už známe). Čili délka této cesty (pokud i-tá stávka je na křižovatce (i, j)) je $min(abs(x - i) + C_{i+1}(i, y), abs(y - j) + C_{i+1}(x, j))$.

Tímto máme další řešení, které musí počítat $M \cdot N^2$ čísel, každé v konstantním čase. To není o nic lepší. Ale můžeme si všimnout jedné věci – hodnoty matice C_{i+1}, které při výpočtu skutečně použijeme, leží vždy jen na ulicích, ze kterých je vidět i-tá stávka. Takže vlastně nepotřebujeme počítat hodnoty v celé matici C_{i+1}, stačí pouze ty, ze kterých je vidět i-tou stávku. Těch je ale docela málo – přesněji $2N - 1$.

Z toho plyne následující řešení: pro každou stávku od M-té k první si spočteme délku nejkratší cesty, která nafilmuje stávky $i, \ldots, M$ a skončí v CNN. Tyto délky budeme ale počítat jen ve křižovatkách, ze kterých můžeme i-tou stávku nafilmovat. K výpočtu těchto délek nám poslouží již popsaný vzoreček.

Navíc pokud si u každé křižovatky budeme pamatovat, kudy z ní nejkratší cesta vede, můžeme nakonec i vypsat cestu vozu (a popředu – proto jsme hledali nejkratší cesty od poslední stávky). Celkem má naše řešení časovou i paměťovou složitost $O(M \cdot N)$. Paměťová by šla snížit na $O(N)$, ovšem jen pokud bychom nechtěli znát cestu, ale jen její délku.



Další možnost, i když poněkud zběsilá: Jak víme, hodnoty, které potřebujeme, leží na „kříži“ $\dagger$ aktuální stávky (tak budeme říkat křižovatkám, ze kterých je tato stávka vidět) a počítáme je z hodnot na kříži $\dagger'$ následující stávky. Každý kříž si ale můžeme rozdělit na svislou a vodorovnou část. Pak hodnoty na vodorovné části $\dagger$ budou hodnoty z vodorovné části $\dagger'$, pouze zvýšené o vzdálenost obou přímek, a navíc ještě průmět všech hodnot ze svislé části kříže $\dagger'$ do průsečíku příslušné svislé přímky s naši vodorovnou, který ohodnotíme minimem z (vzdálenost bodu od vodorovné přímky + ohodnocení bodu). To není o mnoho lepší řešení, ale jen do okamžiku, kdy si uvědomíme, že by se celá situace dala udržovat ve dvou vyhledávacích stromech – jednom pro svislý směr a jednom pro vodorovný –, přičemž podobně jako si v úloze 16-4-5 udržujeme minimální rozdíl, bychom si udržovali minimum z (souřadnice ve směru přímky + ohodnocení bodu), a tak bychom dokázali od jednoho kříže k druhému přejít v čase $O(\log N)$, dosahujíce tak celkové časové složitosti $O(M \cdot \log N)$. –M.M.

```
#include <stdio.h>
#define MAX_N 100
#define MAX_M (MAX_N*MAX_N)
#define min(a, b) (((a) < (b)) ? (a) : (b))
#define abs(a) (((a) > 0) ? (a) : -(a))
enum {ROW, COL};

struct krizovatka {int x, y; };
struct krizovatka cnn;
struct krizovatka stavky[MAX_M];            /* pole se stávkami */
int delky[MAX_M][2][MAX_N+1];               /* délky do řádku a sloupce na stávce */
int M, N;

int main (void) {
  int i, stavka;
  struct krizovatka m;
  int m_delka;

  printf ("Zadejte N a M:");
  scanf ("%d %d", &N, &M);
  printf ("Zadejte souřadnice CNN:");
  scanf ("%d %d", &cnn.x, &cnn.y);

  for (i=0; i<M; i++) {
    printf ("Zadejte souřadnice %d. stávky:", i+1);
    scanf ("%d %d", &stavky[i].x, &stavky[i].y);
  }

  for (i=1; i<=N; i++) {                   /* počáteční nastavení cest do CNN */
    delky[M-1][ROW][i]=abs(i-cnn.x)+abs(stavky[M-1].y-cnn.y);
    delky[M-1][COL][i]=abs(stavky[M-1].x-cnn.x)+abs(i-cnn.y);
  }

  for (stavka=M-2; stavka>=0; stavka--) {  /* pro všechny stávky */
    for (i=1; i<=N; i++) {
      delky[stavka][ROW][i]=min(abs(stavky[stavka].y-stavky[stavka+1].y)+
                                    delky[stavka+1][ROW][i],
            abs(i-stavky[stavka+1].x)+delky[stavka+1][COL][stavky[stavka].y]);
      delky[stavka][COL][i]=min(abs(stavky[stavka].x-stavky[stavka+1].x)+
                                    delky[stavka+1][COL][i],
            abs(i-stavky[stavka+1].y)+delky[stavka+1][ROW][stavky[stavka].x]);
    }
  }
  printf ("CNN->");                         /* výpis trasy */
  m=cnn;
  m_delka=min(abs(cnn.y-stavky[0].y)+delky[0][ROW][cnn.x],
              abs(cnn.x-stavky[0].x)+delky[0][COL][cnn.y]);
  for (stavka=0; stavka<M; stavka++) {
    if (delky[stavka][ROW][m.x]+abs(m.y-stavky[stavka].y) == m_delka)
      m.y=stavky[stavka].y;
    else m.x=stavky[stavka].x;
    printf ("(%d,%d)->", m.x, m.y);
  }
  printf ("CNN\n"); return 0; }
```

16-4-4 Dlouhoprstova zapeklitá hra — Martin Mareš

Většina z vás, zdá se, přála Dlouhoprstovi buďto předlouhý život nebo z pekla štěstí, jelikož většina řešení byla buďto ďábelsky pomalá (pracovala v exponenciálním nebo dokonce faktoriálovém čase) nebo to byly po čertech podivné heuristiky fungující pouze pro některé vstupy a pro jiné se chovající značně démonicky. Ale přeci jen někteří dokázali našemu hrdinovi (neříkám, že kladnému) s dlouhými prsty a krátkýma nohama pomoci. Jde to třeba takto:

Označme si $D(\alpha,\beta)$ nejkratší možnou posloupnost Dlouhoprstových karet (Dlouhoprstova posloupnost, zkráceně DP), která odpovídá Satanovým posloupnostem (SP) α a β. Podle toho, jak α a β začínají, můžeme rozlišit následující případy:

- $D(x\alpha,x\beta) = xD(\alpha,\beta)$ – pokud začínají obě SP stejným písmenem, musí DP začínat také tímto písmenem a za ním bude následovat DP pro původní SP bez tohoto písmena.
- $D(x\alpha,y\beta) = \Psi$ – pokud SP začínají různými písmeny, evidentně příslušná DP neexistuje, takže vrátíme chybu, a tu budeme značit Ψ.
- $D(\varepsilon,\varepsilon) = \varepsilon$ – pokud jsou obě SP prázdné, je DP také (ε budeme značit prázdný řetězec).
- $D(x\alpha,*\beta) = \min(D(x\alpha,\beta),D(\alpha,*\beta))$ – pokud se objeví hvězdička, můžeme ji buďto splnit prázdným řetězcem a nebo ji nechat „spolknout“ první písmenko druhého řetězce (případně i nějaká další, protože * v řetězci ponecháme). Vybereme si samozřejmě kratší z obou variant (jako min značíme minimum řetězcové, které z dvou řetězců vrátí ten kratší a pokud mají stejnou délku, tak libovolný z nich; navíc Ψ je delší než všechny řetězce).
- $D(\varepsilon,*\beta) = D(\varepsilon,\beta)$ – pokud je levá strana prázdná, musí hvězdičce vyhovovat prázdný řetězec, ale ještě musíme pokračovat, protože napravo může být hvězdiček více.
- $D(*\alpha,x\beta)$, $D(*\alpha,\varepsilon)$ – analogicky.
- $D(*\alpha,*\beta) = \min(D(\alpha,*\beta),D(*\alpha,\beta))$ – pokud obě SP začínají na *, nemá smysl do DP přidávat znaky, které by měly vyhovovat oběma hvězdičkám – ty by byly zbytečné. Takže chceme jednu z hvězdiček vypustit a druhou ponechat, jen si musíme vybrat tu správnou, pročež zkusíme obě.
- pokud se někde vyskytne otazník, můžeme ho chápat jako písmeno, které se rovná libovolnému jinému písmenu a pokud bychom ho chtěli vypsat do výstupu, vypíšeme místo něj libovolné jedno písmeno.
- $D(x\alpha,\varepsilon)$ a ostatní zbylé případy (jedna SP došla a druhá ještě ne) jsou neřešitelné, a tak odpovíme Ψ.

Rekursivním použitím těchto pravidel už můžeme spočítat D pro libovolnou dvojici Satanových posloupností, ale má to jeden háček: rekurse se nám na hvězdičkách větví, takže může trvat exponenciálně dlouho. Jak z toho ven?

Všimneme si, že všechny řetězce, pro které D počítáme, jsou vždy suffixy původních SP (suffix je část řetězce od nějakého místa až do konce) a rekurse je exponenciální jen proto, že se pro mnohé dvojice suffixů počítá totéž vícekrát. Budeme si proto již spočtené hodnoty pamatovat v pomocném poli a kdykoliv by po nás někdo chtěl hodnotu spočítat znovu, prostě ji jen vytáhneme z pole jako králíka z klobouku a hned se vrátíme bez dalšího rekursivního volání. Možných dvojic suffixů je jenom $(M+1)\cdot(N+1)$, takže netriviálních volání funkce D může být jen $O(M \cdot N)$.

Již z toho by plynul pěkný algoritmus se složitostí $(M \cdot N \cdot (M+N))$, ale ten by většinu času trávil předáváním řetězců mezi funkcemi. Tak se ho ještě zkusíme zbavit: Všimneme si, že každé $D(\alpha, \beta)$ vždy získáme z nějakého $D(\alpha', \beta')$ (kde α' je nějaký suffix řetězce α a obdobně β') přidáním jednoho nebo žádného znaku na začátek. Naše funkce tedy místo toho, aby vrátila řetězec, jen poznamená do nějakého pomocného pole, jak má hodnota vzniknout a jak bude dlouhá (to zvládneme na konstantní počet operací), a po ukončení výpočtu ten správný řetězec podle těchto poznámek zrekonstruujeme (v lineárním čase).

Celkově má tedy náš algoritmus časovou i paměťovou složitost $O(M \cdot N)$.

```
#include <stdio.h>

#define MAX 100                              /* sizeof(Hell) */
#define INF 100000                           /* Nekonečno */
char a[MAX], b[MAX];                         /* Satanovy řetězce, ukončené kódem 0 */
int opt[MAX][MAX];                           /* Optimální délka řetězce pro danou
                                                dvojici suffixů SP */
int optx[MAX][MAX], opty[MAX][MAX];          /* Poznámky ke konstrukci řetězců */
char optc[MAX][MAX];

int D (int x, int y)                         /* Spočte D pro zadané suffixy SP */
{
   int ret;                                  /* Budoucí výsledek */
   int bx=0, by=0;                           /* Odkud jsme ho vzali */
   int bc=0;                                 /* Co musíme připsat, aby vznikl */
   if (opt[x][y] >= 0) return opt[x][y];     /* Králík z klobouku? */
   /* Dvě pomocná makra: nastavení výsledku a pokus o jeho vylepšení */
#define SET (xx, yy, cc) do { ret=D (xx, yy); bx=xx; by=yy; bc=cc; } while (0)
#define UPDATE (xx, yy, cc) do { int r2=D (xx, yy); \
                        if (r2 < ret) { ret=r2; bx=xx; by=yy; bc=cc; } } while (0)
   if (!a[x] && !b[y]) ret = 0;              /* Oba řetězce skončily */
   else if (a[x] == '*'&& b[y] == '*') {     /* Dvě hvězdičky */
         SET (x+1, y, 0);
         UPDATE (x, y+1, 0);
   } else if (a[x] == '*') {                 /* Hvězdička vlevo ... */
         SET (x+1, y, 0);
         if (b[y]) UPDATE (x, y+1, b[y]);
```

```
    } else if (b[y] == '*') {                    /* ... nebo vpravo */
        SET (x, y+1, 0);
        if (a[x]) UPDATE (x+1, y, a[x]);
    } else if (a[x] == b[y]) SET (x+1, y+1, a[x]);   /* 2 stejné znaky */
    else if (a[x] == '?'&& b[y]) SET (x+1, y+1, b[y]);   /* Otazníky */
    else if (a[x] && b[y] == '?') SET (x+1, y+1, a[x]);
    else ret = INF;                              /* Jinak nemožno */
    optx[x][y] = bx;
    opty[x][y] = by;
    optc[x][y] = bc;
    return opt[x][y] = ret;
}

int main (void)
{
    int i, j, l;
    scanf ("%s%s", a, b);
    for (i=0; i<MAX; i++)
      for (j=0; j<MAX; j++)
        opt[i][j] = -1;
    if (D (0, 0) >= INF) {
        printf ("Pomoooc! Prohraavaaaaam!\n");
        return 0;
    }
    int x=0, y=0, nx;                            /* Rekonstrukce výsledného řetězce */
    while (a[x] && b[y]) {
        if (optc[x][y])
          putchar (optc[x][y] == '?' ? 'x': optc[x][y]);
        nx = optx[x][y];
        y = opty[x][y];
        x = nx;
    }
    putchar ('\n');
    return 0;
}
```

16-4-5 Obchodníci s deštěm — Milan Straka

Velkou část našich řešitelů O. Š .Kubal, věren svému jménu, žel Bůhdhovi, o.š.kubal. Svými pomalými programy totiž nedokázali zjistit, jaký zmetek jim chtějí Kubalovi prodat.

První věci, které si všimneme, je to, že čas potřebný na jednu odpověď (vypsání aktuálního rozdílu po přečtení jednoho čísla) by neměl být závislý na N, ale jenom na K.

Nejjednodušší řešení je, po načtení další hodnoty, spočítat všechny vzdálenosti dvojic posledních K vrcholů a z nich si vybrat tu nejmenší. To určitě zvládneme v $O(N \cdot K^2)$.

Vylepšit to můžeme například tak, že si všimneme, že pokud bychom měli posledních K hodnot setříděných, nemusíme zkoumat $O(K^2)$ vzdáleností, stačí

nám spočítat vzdálenosti mezi dvěma sousedními prvky (sousedí v *setříděném* poli). Těch už je jenom $K - 1$, nicméně třídění nás stojí zase $O(K \log K)$. Celkem vylepšení na $O(N \cdot K \log K)$.

Další pozorování je, že po načtení jednoho čísla se pole posledních K čísel moc nezmění – určitě nemá cenu ho třídit vždy znova. Pokud máme setříděné pole posledních K čísel a načítáme další, stačí to nejstarší z pole vyhodit ($O(K)$) a nové přidat ($O(K)$) tak, aby pole zůstalo uspořádané. Pak stačí v jednom průchodu nad polem spočítat vzdálenosti sousedních prvků a vypsat nejmenší. Tím jsme na $O(N \cdot K)$.

Vylepšovat ale jde ještě dále. Ukážeme si dvě možná řešení se složitostí $O(N \cdot \log K)$. První z nich je založeno na tomto pozorování: pokud uvažujeme o postupu s lineárním časem, tak počet dvojic, jejichž vzdálenost počítáme, se při načtení jednoho čísla mění velmi málo. Můžeme tedy mít všechny vzdálenosti sousedních prvků (sousedních v setříděném poli) v haldě. Při načtení nového čísla ho zatřídíme do nějaké struktury (použijeme např. AVL stromy z minulé kuchařky), která nám řekne jeho sousedy (většího a menšího) v setříděném poli. Pokud už je máme, z haldy odebereme vzdálenost těchto dvou sousedů a naopak do ní vložíme vzdálenost aktuálního prvku od menšího a vzdálenost aktuálního prvku od většího souseda. Při mazání čísla uděláme podobnou úpravu (zase si najdeme sousedy mazaného prvku, z haldy odebereme dvě hodnoty a dáme tam místo nich jednu [vzdálenost sousedů mazaného prvku]).

Pokud použijeme ke zjišťování sousedů nějaký druh vyvážených stromů (třeba AVL :–), můžeme hledání sousedů, vkládání a mazání provádět v čase $O(\log K)$. Stejnou složitost mají i operace s haldou – a protože všeho tohoto děláme konstantní počet, máme řešení se složitostí $O(N \cdot \log K)$.

To bylo jedno řešení, slíbili jsme ještě druhé: opět použijeme nějaký vyvážený binární strom. Každý jeho vrchol bude odpovídat jednomu z posledních K čísel, nicméně ve vrcholu si kromě hodnoty budeme pamatovat ještě tyto údaje:

- *min* – minimum hodnot v tomto podstromě.
- *max* – maximum hodnot v tomto podstromě.
- *delta* – nejmenší vzdálenost hodnot v tomto podstromě.

Pokud máme vrchol a známe tyto hodnoty u obou jeho synů, můžeme si spočítat i jeho hodnoty v konstantním čase:

- *min* – vezmeme minimum od levého syna.
- *max* – vezmeme maximum od pravého syna.
- *delta* – vezmeme minimum z delt levého a pravého syna, dále ze vzdálenosti hodnoty aktuálního vrcholu od maxima levého syna a ještě rozdíl hodnoty aktuálního vrcholu a minima pravého syna.

Můžeme tedy načtené hodnoty vložit do stromu, přepočítat popsané hodnoty a vypsat deltu kořene. Přepočítání hodnot můžeme provádět tak, že po vložení/smazání prvku budeme stromem procházet od vloženého/smazaného prvku směrem ke kořeni a po cestě upravovat popsané hodnoty. Pokud bude strom opravdu vyvážený, bude mít logaritmickou hloubku a tedy popsané operace budou mít složitost $O(\log K)$ a celé řešení tedy $O(N \cdot \log K)$.

Ve vzorovém řešení jsme schválně nepoužili AVL stromy, ty už znáte. Použili jsme tzv. BB-α stromy, které mají logaritmickou složitost pouze amortizovaně. To nám ale vůbec nevadí, protože nás zajímá složitost N operací a ne jedné.

BB-α strom je normální binární vyhledávací strom takový, že v každém vrcholu platí podmínka, že počet vrcholů v levém a pravém podstromě se liší nanejvíc α-krát. Takový strom má vždy logaritmickou hloubku, protože podstrom nějakého stromu má nanejvýš $\alpha/(\alpha+1)$ vrcholů – počet vrcholů v podstromu tak klesá geometrickou řadou a maximální možná výška stromu je tak $\log_{(\alpha+1)/\alpha} N$.

A jak takovou podmínku dodržet? U každého vrcholu si budeme udržovat počet vrcholů v levém a pravém podstromu. Pokud kdykoliv zjistíme, že se liší více než α-krát, celý podstrom odpojíme, vytvoříme z něj vyvážený strom a vrátíme zpátky. Takové „vybalancování" určitě trvá lineárně vzhledem k počtu vrcholů ve vybalancovávaném stromečku.

Předpokládejme nyní, že $\alpha = 2$, stejně jako ve vzorovém programu. Kolik stojí jedno vkládání či mazání? Na to, aby se nějaké vybalancování spustilo, se musí lišit hodnoty v levém a pravém podstromu dvakrát, čili od minulého rebalancování muselo dojít k řádově tolika vkládáním a mazáním, kolik je vrcholů ve zkoumaném stromečku. Čili stačilo, aby každé vkládání a mazání přispělo aktuálnímu vrcholu konstantním časem (jedním penízkem), ze kterého se pak vybalancování „uplatí". Každé vkládání a mazání musí přispět na rebalancování všem vrcholům, přes které projde. Těch je ale nanejvíc tolik, jaká je výška stromu – a ta je logaritmická. Čili amortizovaná složitost vkládání nebo mazání prvku je $O(\log K)$ (amortizovaná znamená, že i když nevíme, jak dlouho bude jedna operace doopravdy trvat, N operací bude trvat nejvýš $O(N \cdot K)$).

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_K 100

struct node {
    int value;
    int min, max, delta;
    struct node *left, *right, *parent;
    int left_c, right_c;
};

struct node *root=NULL;                          /* kořen stromu */
```

```
int hodnoty[MAX_K];                          /* posledních K načtených hodnot */
int N, K;

void bbtree_oprav (struct node *);           /* nadeklarujeme až později */

struct node *bbtree_find (struct node *ptr, int value) {
   if (value < ptr->value && ptr->left) return bbtree_find (ptr->left, value);
   if (value > ptr->value && ptr->right) return bbtree_find (ptr->right, value);
   return ptr;
}

void bbtree_insert (int value) {             /* vloží vrchol do stromu */
   struct node *new=malloc (sizeof (struct node)), *ptr;
   new->value=new->min=new->max=value;
   new->left=new->right=new->parent=NULL;
   new->delta=new->left_c=new->right_c=0;

   if (!root) root=new;                      /* ještě nemám root */
   else {
      ptr=bbtree_find (root, value);
      new->parent=ptr;                       /* napojíme na rodiče */
      if (value < ptr->value) ptr->left=new; else ptr->right=new;
                                                /* a rodiče na mě */
      bbtree_oprav (new);
   }
}

void bbtree_remove (int value) {             /* odebere vrchol */
   struct node *ptr=bbtree_find (root, value), *son;

   if (!ptr || ptr->value != value) return;  /* nenašli jsme – to se nám nestane */
   if (ptr->left && ptr->right) {            /* mám 2 syny – najdi za sebe náhradu */
      struct node *nahrada= (ptr->left_c>ptr->right_c) ? ptr->left : ptr->right;
      while ( (ptr->left_c>ptr->right_c) ? nahrada->right : nahrada->left)
         nahrada= (ptr->left_c>ptr->right_c) ? nahrada->right : nahrada->left;
      ptr->value=nahrada->value;
      ptr=nahrada;
   }

   son= (ptr->left) ? ptr->left : ptr->right;
   if (son) son->parent=ptr->parent;         /* úprava syna */
   if (!ptr->parent) root=son;               /* úprava rodiče */
   else if (ptr->parent->left==ptr) ptr->parent->left=son;
      else ptr->parent->right=son;
   if (ptr->parent) bbtree_oprav (ptr->parent);
   free (ptr);
}

void bbtree_update_vrcholu (struct node *ptr) {   /* přepočítá min, max, delta, left_c a
                                                 right_c aktuálního vrcholu */
#define upr_delta (a) if ( (a)&& (ptr->delta==-1|| (a)<ptr->delta)) ptr->delta= (a);
   ptr->min= (ptr->left) ? ptr->left->min : ptr->value;
   ptr->max= (ptr->right) ? ptr->right->max : ptr->value;
   ptr->delta=-1;                            /* delta neinicializovaná */
   if (ptr->left) {                          /* pomůže mi levý syn? */
```

```
    upr_delta (ptr->left->delta);
    upr_delta (ptr->value - ptr->left->max);
  }
  if (ptr->right) {                          /* a co pravý syn? */
    upr_delta (ptr->right->delta);
    upr_delta (ptr->right->min - ptr->value);
  }
  if (ptr->delta<0) ptr->delta=0;
  ptr->left_c= (ptr->left) ? ptr->left->left_c + 1 + ptr->left->right_c: 0;
  ptr->right_c= (ptr->right) ? ptr->right->left_c + 1 + ptr->right->right_c: 0;
}

  /* tohle vytvoří spoják z vrcholů v stromu s vrcholem ptr a vrátí to první prvek p */
  /* spoják je svázaný ->left pointerama, jenom p->right je konec spojáčku */
  /* a p->right_c je počet prvků ve spojáčku */
struct node *bbtree_collect (struct node *ptr) {
  struct node *begin;
  if (ptr->left) {                           /* někdo vlevo? */
    begin=bbtree_collect (ptr->left);
    begin->left->right=ptr;
  } else {begin=ptr; begin->right_c=0; }
  begin->left=ptr;
  begin->right_c++;
  if (ptr->right) {                          /* a co vpravo? */
    begin->left->right=bbtree_collect (ptr->right);
    begin->right_c+=begin->left->right->right_c;
    begin->left=begin->left->right->left;
  }
  return begin;
}

  /* tohle z výše popsaného spojáčku udělá vyvážený strom */
  /* parametr next ukazuje na první dosud nepoužitý prvek z popsaného seznamu */
struct node *bbtree_rebuild (struct node *spojak, struct node **next) {
  int l=spojak->right_c/2;                   /* počet vrcholů vlevo */
  int r=spojak->right_c-l-1;                 /* a vpravo */
  struct node *left=NULL, *right=NULL;

  if (l) {spojak->right_c=l; left=bbtree_rebuild (spojak, next); spojak=*next; }
  *next=spojak->right;                       /* toto je momentálně první nepoužitý
                                                prvek */
  if (r) {spojak->right->right_c=r; right=bbtree_rebuild (spojak->right, next); }

  spojak->left=left;                         /* spoják bude kořen */
  if (left) left->parent=spojak;             /* upravit pointery na syny a rodiče */
  spojak->right=right;
  if (right) right->parent=spojak;
  bbtree_update_vrcholu (spojak);
  return spojak;
}

  /* tato funkce se stará o opravu hodnot min,max a delta ve stromě */
  /* podle potřeby přestavuje strom funkcemi bbtree_collect a bbtree_rebuild */
void bbtree_oprav (struct node *ptr) {
```

```
    struct node *parent=ptr->parent;
    bbtree_update_vrcholu (ptr);
    if (ptr->left_c/2 > ptr->right_c || ptr->right_c/2 > ptr->left_c) {
                                                 /* přestavba stromu? */
        struct node *tmp;
        tmp=bbtree_rebuild (bbtree_collect (ptr), &tmp);

        if (parent && parent->value > ptr->value) parent->left=tmp;
        else if (parent && parent->value < ptr->value) parent->right=tmp;
        else root=tmp;
        tmp->parent=parent;
    }
    if (parent) bbtree_oprav (ptr->parent);      /* propagace výš */
}

int main (void) {
    int i;

    printf ("Zadejte N a K:");
    scanf ("%d %d", &N, &K);

    for (i=0; i<N; i++) {
        int v=i%K;

        if (i >= K) bbtree_remove (hodnoty[v]);     /* budeme mazat? */
        scanf ("%d", hodnoty+v);
        bbtree_insert (hodnoty[v]);

        if (i) printf ("Aktuální nejmenší rozdíl je %d.\n", root->delta);
    }
    return 0;
}
```

16-4-6 Šikmá věž v Kocourkově — Martin Mareš

Řešení potíží našich věžechtivých Kocourkovanů v kostce: Klíčové úkony leží na nejdelší cestě v závislostním grafu (acyklickém!) a všechny takové cesty můžeme najít projitím grafu v topologickém pořadí.

Připusťme ale poněkud pomalejší ch(r)ápání kocourkovských buřtipánů a zkusme tento nápad poněkud rozvést:

Nejdříve si sestrojíme *závislostní graf:* to bude orientovaný graf, jehož vrcholy budou odpovídat úkonům a z u do v povede hrana právě tehdy, když je nutné úkon u dokončit před započetím úkonu v; navíc si každý vrchol ohodnotíme číslem, které bude udávat, jak dlouho příslušný úkon trvá. Ještě přidáme počáteční úkon α odpovídající položení základního kamene stavby (ohodnocena nulou a vedou z ní hrany do všech ostatních úkonů) a úkon ω odpovídající kolaudaci (opět nula [naivní, vím], hrany ze všech ostatních úkonů). *Délkou cesty* budeme nazývat součet ohodnocení vrcholů, které na ni leží, *bez počátečního a koncového vrcholu,* což je sice trochu nesystematické, ale usnadní nám to počítání. *Odlehlostí* z vrcholu (úkonu) u do v pak nazveme délku *nejdelší*

cesty z u do v (to je svým způsobem opak vzdálenosti, která je délkou cesty nejkratší).

Pokud je v závislostním grafu cyklus, věž evidentně postavit nelze. Budeme tedy předpokládat, že graf je acyklický a ukážeme, že věž postavit půjde, a to v čase rovném odlehlosti L z α do ω. Ještě trocha značení: pro každý úkon u bude $l(u)$ délka tohoto úkonu, $a(u)$ odlehlost z α do u a $z(u)$ odlehlost z u do ω.

Teď ale na chvíli odbočíme od tématu a zavedeme si *topologické pořadí* vrcholů grafu. Tak budeme říkat takovému pořadí $v_1, v_2, \ldots, v_n$ všech vrcholů, pro které platí, že pokud vede hrana z v_i do v_j, je vždy $i < j$. V libovolném acyklickém grafu takové pořadí existuje, což si dokážeme tak, že si rovnou předvedeme lineární algoritmus, který ho najde.

Pokud je graf acyklický, musí v něm existovat vrchol v_1, kam nevede žádná hrana: stačí vyjít z libovolného vrcholu a stále jít proti směru hran – jelikož graf je konečný, nemůžeme přicházet do stále nových vrcholů, takže časem narazíme buďto na vrchol, do kterého nic nevede, nebo na vrchol, ve kterém jsme už byli, což ovšem není možné, protože bychom tím uzavřeli cyklus. Nalezený vrchol v_1 očíslujeme jedničkou (to je určitě správně, nevede do něj přeci žádná hrana) a z grafu ho odebereme včetně všech hran, které z něj vedou. Tím získáme opět acyklický graf a v něm pokračujeme stejně od dvojky. Až nezbude žádný vrchol, budeme mít hotové topologické pořadí; pokud by nějaký zbyl, znamená to, že se v grafu nacházely cykly. Abychom ale dosáhli lineární časové složitosti, potřebujeme umět takové vrcholy nacházet v konstantním čase. K tomu stačí zapamatovat si pro každý vrchol počet hran, které do něj vedou, při odebírání hran tyto počty aktualizovat a udržovat si frontu vrcholů, které už mají nulu, ale ještě jsme je neodebrali.

[Ve vzorovém programu nepřiřazujeme čísla explicitně, ale využíváme toho, že ve frontě jsou vrcholy uloženy také v topologickém pořadí. Ještě jiný způsob, jak topologické pořadí najít, by byl prohledat graf do hloubky a všimnout si, že pořadí, ve kterém se z vrcholů vracíme, je obrácené topologické pořadí. Zkuste si rozmyslet, proč to platí, v některé z kuchařek v příštím ročníku se na to podíváme podrobněji.]

Hodnoty $a(u)$ pak můžeme spočítat velice snadno indukcí: bereme vrcholy v topologickém pořadí, začínáme s $a(\alpha) = 0$. Pro každý další vrchol využijeme toho, že již známe $a(v)$ pro všechny předchůdce v vrcholu u, čili vrcholy, z nichž vede do u hrana, a položíme $a(u) = \max_v(a(v) + l(v))$ (nejdelší cesta z α do u musí být nutně nejdelší cestou do některého z předchůdců u prodloužená o hranu do u). To zvládneme v lineárním čase a stejně tak můžeme spočítat i $b(u)$ pomocí opačného topologického pořadí a L jako $a(\omega)$ nebo $z(\alpha)$.

(Konec odbočky.) Nyní si všimněme, že žádný úkon nelze začít provádět dříve než v čase $a(u)$: víme totiž, že existuje posloupnost úkonů, které musí být všechny provedeny po sobě a před u a dohromady trvají $a(u)$. Z toho

také plyne, že celou věž nemůžeme dostavět dříve než za $L = a(\omega)$. Teď už stačí ukázat, že si můžeme úkony rozvrhnout tak, abychom u dokončili v čase $a(u) + l(u)$, a tedy celou stavbu v čase L. To je snadné: každý úkon u začneme provádět v čase $a(u)$ a vzorec pro $a(u)$ z minulého odstavce vlastně říká přesně to, že všechny předchozí úkoly jsou nejpozději v tomto okamžiku hotové.

Zbývá ještě zjistit, které úkoly jsou klíčové: jsou to ty, které leží na některé z nejdelších cest (tedy ty, pro které je $L_u = L$, kde $L_u = a(u) + b(u) + l(u)$ je délka nejdelší cesty z α do ω, která obsahuje u). Pokud úkon u na nejdelší cestě leží, je nutně klíčový, protože na úkonech na nejdelší cestě pracujeme po celou dobu L bez přestávek, takže pokud libovolný úkon prodloužíme, prodloužíme i celou dobu stavby. Naopak pokud úkon u neleží na nejdelší cestě, můžeme ho prodloužit až o $L - L_u$ a celkovou dobu tím nezměníme.

Program je toliko formálním zápisem našich úvah a má lineární časovou složitost a kvadratickou paměťovou (ale jen proto, že jsme si ušetřili práci s načítáním hran; jinak by byla také lineární).

[Všchn thle b s smzřjm dlo smrsknt d jdnh prchdu grfm, le bl b t čtlné as jko thl vta. –M.M.]

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

#define MAXN 100

int N;                                   /* Počet úkonů: úkon 0 je α, N je ω */
int l[MAXN];                             /* Délky úkonů */
int a[MAXN], z[MAXN];                    /* Délky maximálních cest (viz popis) */
int edges[MAXN][MAXN];                   /* Hrany vedoucí z jednotlivých vrcholů */
int outdeg[MAXN], indeg[MAXN];           /* Vstupní a výstupní stupeň vrcholů */
int top[MAXN];                           /* Topologické pořadí vrcholů */

void add_edge (int v, int w)             /* Přidá hranu do závislostního grafu */
{
    edges[v][outdeg[v]] = w;
    outdeg[v]++;
    indeg[w]++;
}

void read (void)                         /* Přečte vstup a vytvoří graf */
{
    int v, w;
    scanf ("%d", &N); N++;               /* Počet úkonů a jejich ceny */
    for (v=1; v<N; v++)
        scanf ("%d", &l[v]);
    while (scanf ("%d%d", &v, &w) == 2 && v > 0)   /* Závislosti, ukončeny (0,0) */
        add_edge (v, w);
    for (v=1; v<N; v++) {                /* Závislosti pro vrcholy α a ω */
            add_edge (0, v);
            add_edge (v, N);
    }
}
```

```
void topsort (void)                               /* Topologicky setřídí graf */
{
   int r = 0, w = 1, i, u, v;
   top[0] = 0;                                    /* První je vrchol α */
   while (r < w) {                                /* Probíráme frontu vrcholů (v)stupně 0 */
         u = top[r++];
         for (i=0; i<outdeg[u]; i++) {            /* Projdeme všechny hrany */
               v = edges[u][i];
               if (!--indeg[v])                   /* Snížime stupeň cílového vrcholu */
                  top[w++] = v;                   /* a pokud je 0, přidáme do fronty */
         }
   }
   if (w != N+1) {                                /* Objevili jsme cyklus */
         printf ("Nelze.\n");
         exit (0);
   }
}
void paths (void)                                 /* Spočte délky nejdelších cest */
{
   int i, j, v, w;
   for (i=0; i<=N; i++) {                         /* a[v] počítáme popředu */
         v = top[i];
         for (j=0; j<outdeg[v]; j++) {
               w = edges[v][j];                   /* hrana (v, w) */
               if (a[w] < a[v]+l[v]) a[w] = a[v]+l[v];
         }
   }
   for (i=N; i>0; i--) {                          /* z[v] zase pozpátku */
         v = top[i];
         for (j=0; j<outdeg[v]; j++) {
               w = edges[v][j];                   /* hrana (v, w) */
               if (z[v] < z[w]+l[w]) z[v] = z[w]+l[w];
         }
   }
}
void answer (void)                                /* Vypíše odpověď */
{
   int i;
   printf ("Věž lze postavit za %d TUK.\n", a[N]);   /* Time Unit of Kocourkov */
   for (i=1; i<N; i++)
      if (a[i] + z[i] + l[i] == a[N]) printf ("Úkon %d je klíčový.\n", i);
}
int main (void)
{
   read ();
   topsort ();
   paths ();
   answer ();
   return 0;
}
```

Pořadí řešitelů

Pořadí	*Jméno*	*Škola*	*Ročník*	*Úloh*	*Bodů*
			max.	22	219
1.	Peter Perešíni	GJGTajov	2	20	180
2.	Miroslav Cicko	GJGTajov	3	21	177
3.	Petr Škoda	GÚstavníPH	4	21	175
4.	Jan Bulánek	G Klatovy	3	17	147
5.	Kryštof Hoder	GKptJaroBO	4	13	120
6.	Jana Kravalová	G VKlobou	4	18	113
7. – 8.	Ondřej Bílka	G Zlín	2	15	103
	Miroslav Klimoš	G Lanškr	0	18	103
9.	David Matoušek	GZborovPH	4	12	93
10.	Pavel Klavík	G Chrudim	1	21	92
11.	Marek Jančuška	G Nitra	4	9	88
12.	Jan Hrnčíř	GFXŠaldy	2	19	86
13.	Ondřej Garncarz	G Příbor	3	22	84
14.	Michal Repovský	G Trebišov	4	12	83
15.	Pavel Motloch	GPBezruče	1	12	77
16.	Zbyněk Falt	GNeumZdar	3	12	74
17.	Marek Blahuš	G UHradi	3	10	72
18.	Jana Fabriková	GKptJaroBO	4	14	66
19.	Martin Koníček	G UBrod	3	11	62
20.	Martin Podloucký	G Strážnic	3	11	60
21.	Eva Schlosáriková	G Piešťany	3	14	58
22. – 24.	Martin Čech	G UBrod	3	13	56
	Petr Kratochvíl	G Světlá	1	12	56
	Filip Šauer	G Klatovy	3	13	56
25.	Peter Šufliarsky	G NZámky	4	11	55
26. – 27.	Stanislav Basovník	G Kroměříž	3	7	51
	Martin Dobroucký	G MTřebová	3	8	51
28.	Stanislav Haviar	G Klatovy	3	8	50
29.	Michal Bečka	G MTřebová	4	7	48
30.	Peter Černo	GĽŠtúra	3	6	46
31. – 32.	Petr Kortánek	G Sedlča	2	11	45
	Martina Tomisová	GZborovPH	4	13	45
33. – 34.	Jindřich Flídr	G Lanškr	4	7	43
	Tomáš Gavenčiak	G Bílovec	4	5	43
35.	Daniel Marek	GZborovPH	2	5	42
36.	Petr Soběslavský	GJHeyrovPH	3	14	41

37. – 38.	Cyril Hrubiš	G Bílovec	2	11	37
	Martin Křivánek	GKptJaroBO	2	5	37
39.	Ján Zahornadský	GZborovPH	3	8	36
40.	Petr Švec	G Beroun	4	8	35
41.	Jaroslav Havlín	G Sedlča	4	5	32
42.	Adam Přenosil	GSladkovPH	2	7	31
43.	Marek Ludha	GJGTajov	4	5	30
44. – 45.	Jan Křetínský	GMLerchaBO	4	3	24
	Martin Kupec	GMendel	2	8	24
46.	Benjamin Vejnar	G Nymburk	4	3	23
47. – 48.	Miroslav Kratochvíl	G Čáslav	2	6	17
	Petr Musil	G MBuděj	2	6	17
49.	Milan Dvořák	G NMnMor	1	3	16
50.	Zbyněk Konečný	GKptJaroBO	1	2	15
51. – 52.	Jiří Bělohradský	SŠAK HrKr	2	2	13
	Jan Richter	G Příbor	3	5	13
53. – 54.	Jiří Cabal ml.	SPŠDvKrál	1	5	12
	Kristýna Knapová	G Jičín	4	2	12
55.	Michal Potfaj	G NMnVáh	4	4	10
56.	Martin Schmid	G ČTřebová	0	3	9
57.	Jindřich Pergler	G Klatovy	3	2	8
58. – 60.	David Irschik	G Ledeč	3	2	7
	Petr Paščenko	GDašická	4	2	7
	Jaromír Vojíř	G Ledeč	3	3	7
61.	Radoslav Sopoliga	G Svidník	4	4	6
62.	Tomáš Herceg	G Třebíč	1	1	4
63. – 64.	Aleš Razým	SpGTáborPL	3	1	2
	Pavel Vlašánek	SPŠ Brunt	3	1	2

Obsah

Úvod . . . 5
Zadání úloh . . . 7
První série . . . 7
Kuchařka první série . . . 9
Druhá série . . . 15
Kuchařka druhé série . . . 19
Třetí série . . . 26
Kuchařka třetí série . . . 31
Čtvrtá série . . . 39
Kuchařka čtvrté série . . . 43
Vzorová řešení . . . 59
První série . . . 59
Druhá série . . . 71
Třetí série . . . 87
Čtvrtá série . . . 104
Pořadí řešitelů . . . 123
Obsah . . . 125

Milan Straka a kolektiv
Korespondenční seminář z programování
XVI. ročník

Autoři a opravující úloh:
Pavel Čížek, Zdeněk Dvořák, Jan Kára,
Daniel Kráľ, Pavel Machek, Martin Mareš,
Marek Sulovský, Milan Straka, Pavel Šanda,
Tomáš Valla, Tomáš Vyskočil

Vydala Univerzita Karlova v Praze, Matematicko-fyzikální fakulta
Oddělení vnějších vztahů a propagace
Ke Karlovu 3, 121 16 Praha 2
Praha 2004

TEX-ová makra pro sazbu ročenky vytvořil Martin Mareš.

S jejich pomocí ročenku vysázel Milan Straka.

Sazba byla provedena písmem Computer Modern v progamu TEX.

Ilustraci na titulní straně vytvořil Martin Kruliš.

Vytiskla Tiskárna Graphis – P. Flodr.

128 stran
Vydání první
Náklad 300 výtisků

Jen pro potřebu fakulty