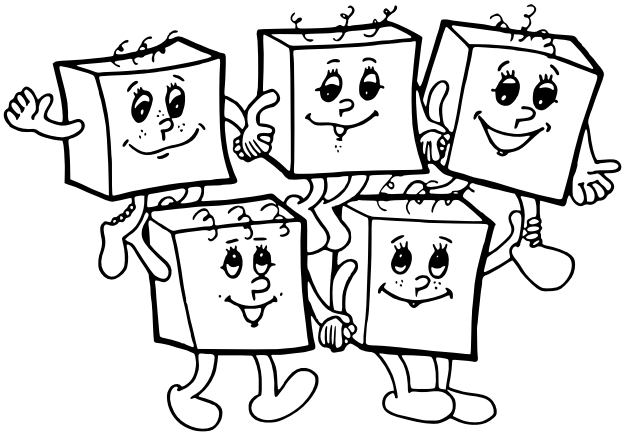
OLYMPIÁDA V INFORMATIKE

NA STREDNÝCH ŠKOLÁCH

http://oi.sk/



dvadsiaty šiesty ročník
školský rok 2010/11

riešenia krajského kola
kategória B

B-II-1 Hľadanie mín II

Na reprezentáciu hracieho plánu použijeme dvojrozmerné pole. Pri načítaní vstupu si do neho zaznačíme polohu mín a následne vypočítame hodnoty ostatných políčok, t.j. pre každé políčko neobsahujúce mínu spočítame počet jeho susediacich mínových políčok. (Toto bola presne vaša úloha v domácom kole.)

Algoritmus pre túto fázu má časovú zložitosť $O(RS)$, teda čas výpočtu rastie približne priamo úmerne s plochou hracieho plánu – hodnotou RS. To preto, že pre každé políčko spravíme len konštantné množstvo operácií: najskôr si najviac raz zaznačíme, že je na ňom mína, a ak nie je, tak prezrieme jeho najviac 8 susedov, aby sme zistili, s koľkými mínami susedí.

Odkrývanie políčok – pomalšie riešenie

V druhej časti riešenia potrebujeme zistiť, ktoré políčka budú odkryté. Priamočiary prístup môže vyzerať napríklad nasledovne: Odkryjeme políčko v ľavom hornom rohu. Odteraz bude výpočet prebiehať v kolách. Každé kolo vyzerá nasledovne: Prejdeme zaradom všetky políčka hracieho plánu a pre každé z nich si položíme otázku, či ho môžeme odkryť. (Podľa definície v zadaní: políčko môžeme odkryť, ak ešte nie je odkryté a zároveň susedí s odkrytým prázdnym políčkom.) Skončíme, akonáhle sa nám stane, že v niektorom kole vôbec nič nové neodkryjeme – inými slovami, keď už na celom pláne nenájdeme žiadne políčko, ktoré môžeme odkryť.

Akú má tento postup časovú zložitosť? V každom kole je počet krokov výpočtu priamo úmerný hodnote RS, teda ploche hracieho plánu. A koľko najviac môže byť kôl? Jeden možný horný odhad hovorí: najviac RS, teda toľko ako políčok na pláne. Totiž v každom kole okrem posledného aspoň jedno nové políčko odkryjeme, a na začiatku už je jedno políčko odkryté.

O celkovej časovej zložitosti teda vieme povedať, že v najhoršom prípade je priamo úmerná hodnote $(RS)^2 = R^2S^2$. Inými slovami, časová zložitosť je $O(R^2S^2)$.

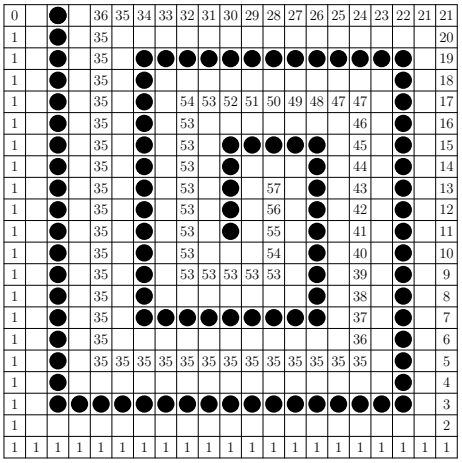
Odbočka pre zaujímavosť: aké pomalé je to naozaj?

Tu si ale zvedavé povahy položia otázku: „Tak moment, toto bol predsa len horný odhad! Je vôbec tesný? Je naozaj tento algoritmus niekedy až taký pomalý?“

Ukážeme si, že naozaj áno: nájdeme konkrétny vstup, ktorý tento algoritmus donúti spraviť veľa kôl. Budeme pre jednoduchosť predpokladať, že každé kolo vyzerá rovnako:

```
for riadok:=1 to R do for stlpec:=1 to S do skus_odkryt_policko(riadok,stlpec);
```

Sľubovaným vstupom bude jednoduchá špirála, ako na obrázku vpravo. Míny sú znázornené plnými krúžkami. Prázdne políčka, t.j. políčka, ktoré nesusedia s mínou, sú práve tie políčka, ktoré na obrázku obsahujú čísla. Číslo na políčku je poradové číslo kola, v ktorom je toto prázdne políčko odkryté. (Okrem prázdnych políčok budú odkryté aj všetky políčka, ktoré s nimi susedia, na tých nám ale momentálne nezáleží.)



Vidíme, že pre $R = S = 21$ náš algoritmus posledné prázdne políčko odkryje až v kole číslo 57. A takúto špirálu vieme zostrojiť pre každé $R = S = 4k + 1$. Koľko kôl na spracovanie takejto špirály potrebujeme? Hrubý odhad hovorí, že približne $RS/8$. Prečo? Prázdne políčka tvoria približne štvrtinu všetkých políčok, a z nich polovicu (tie, ktoré ležia na úsekoch vedúcich dohora a doľava) budeme odkrývať rýchlosťou políčko za kolo.

Pre zaujímavosť, toto pozorovanie sa dá aj exaktne dokázať. Vieme spočítať (alebo uhádnuť a dokázať matematickou indukciou), že pre špirálu nakreslenú na pláne rozmerov $(4k + 1) \times (4k + 1)$ odkryjeme posledné prázdne políčko presne v kole číslo $2k^2 + k + 2$. Pre takýto vstup je teda potrebný počet kôl približne osminou plochy – takže skutočne potrebný počet kôl rastie priamo úmerne s veľkosťou plochy hracieho plánu.

Odkrývanie políčok – vzorové riešenie

Prečo bolo predchádzajúce riešenie neefektívne? Preto, že sme pri hľadaní políčok, ktoré budeme odkrývať, vždy zbytočne prezerali celý hrací plán. Prečo je to zbytočné? Preto, že my predsa vieme, kde tie ďalšie políčka hľadať: hneď vedľa tých, ktoré sme odkryli doteraz!

Aby sme každé políčko odkryli najviac raz, použijeme pomocné dvojrozmerné pole, v ktorom si budeme značiť, či dané políčko už je alebo ešte nie je odkryté. Na začiatku priradíme každému políčku v tomto poli hodnotu `false`, reprezentujúcu ešte neodkryté políčko.

Následne implementujeme jednoduchú rekurzívnu funkciu `odkry`, ktorá odkryje jedno dané políčko. Táto funkcia bude fungovať nasledovne: Ak odkrývané políčko nie je prázdne, nič sa neudeje. Ak však je prázdne, postupne prezrieme jeho susedov. Zakaždým, keď stretneme nejakého ešte neodkrytého, môžeme ho odkryť. To urobíme rekurzívnym volaním našej funkcie, aby sme zabezpečili, že sa odkryje aj všetko za ním. Tým splníme definíciu zadania, podľa ktorého ak sa odkryje prázdne políčko, majú sa odkryť aj všetky jeho susedné políčka. A keďže máme naše pomocné pole, tak platí, že pre každé políčko hracej plochy sa naša funkcia zavolá nanajvýš raz. Vďaka tomu je časová zložitosť nášho riešenia lineárna od veľkosti hracej plochy, teda $O(RS)$.

Listing programu:

```
const MINA = 47;
      MAX_ROZMER = 100;
var   a : array [0..MAX_ROZMER+1, 0..MAX_ROZMER+1] of longint;             {policka hracieho planu}
      odkryte : array [0..MAX_ROZMER+1,0..MAX_ROZMER+1] of boolean; {ci je dane policko uz odkryte}
      i, j, k, l, S, R, N, x, y : longint;

procedure odkry(i,j:longint);
var k,l:longint;
begin
    odkryte[i,j]:=true;
    if a[i,j]=0 then { prazdne policko: postupne odkryjeme vsetkych jeho susedov }
        for k:=-1 to +1 do for l:=-1 to +1 do if not odkryte[i+k,j+l] then odkry(i+k,j+l);
end;
```

```
begin
    { do pola "a" vytvorime hraci plan }
    read(S,R,N);
    for i:=0 to R+1 do for j:=0 to S+1 do a[i,j]:=0;
    for i:=1 to N do begin readln(x,y); a[x,y]:=MINA; end;

    { vyratame hodnoty ciselnych policok }
    for i:=1 to R do for j:=1 to S do
        if a[i,j]<>MINA then { prezrieme susedov policka i,j }
            for k:=-1 to +1 do for l:=-1 to +1 do if a[i+k,j+l]=MINA then inc(a[i,j]);

    { odkryjeme cely okraj, ktory sluzi ako zarazka }
    for i:=0 to R+1 do for j:=0 to S+1 do odkryte[i,j] := (i=0) or (j=0) or (i=R+1) or (j=S+1);

    { odkryjeme policko [1,1] a vsetko, co z toho vyplyva }
    odkry(1,1);

    for i:=1 to R do begin
        for j:=1 to S do
            if not odkryte[i,j] then write('?')
            else if a[i,j]=0 then write('.')
            else write(a[i,j]);
        writeln;
    end;
end.
```

Alternatívne vzorové riešenie

Vyššie popísané riešenie je vlastne aplikáciou všeobecnejšieho grafového algoritmu, tzv. prehľadávania do hĺbky. Úlohu je možné riešiť aj nerekurzívne, napríklad pomocou príbuzného algoritmu: prehľadávania do šírky.

Rovnako ako v predchádzajúcom riešení by sme mali pomocné pole hovoriace, ktoré políčka sú už odkryté. Tentokrát by sme ale implementovali v ďalšom poli dátovú štruktúru fronta. V tej by sme si pamätali zoznam políčok, ktoré je ešte potrebné spracovať. (Teda také, ktoré sme už odkryli, ale ešte sme neprezreli ich susedov.)

Myšlienka algoritmu je nasledujúca. V prvom kroku odkryjeme políčko $(1, 1)$ a zároveň ho aj vložíme do fronty. Potom systematicky v každom kroku vyberieme jedno políčko z fronty. Ak je práve vybrané políčko prázdne, tak každé políčko, ktoré s ním susedí a ešte nie je odkryté, odkryjeme a pridáme do fronty. Keďže každé políčko hracej plochy sa takto dostane do fronty najviac raz, časová zložitosť takéhoto riešenia je, rovnako ako v predchádzajúcom prípade, $O(RS)$.

B-II-2 Zakopaný pes

Označme si dĺžku zadaného reťazca n. Najjednoduchším riešením je pre každú trojicu písmen overiť, či je z nich prvé v poradí P, druhé E a tretie S. Tento prístup má však časovú zložitosť $O(n^3)$.

Takéto riešenie vieme mierne zlepšiť: pre každé z písmen P, E a S si spravíme zoznam jeho výskytov. Potom budeme prechádzať len cez tieto výskyty. Takéto riešenie má časovú zložitosť $O(pes)$, kde p, e a s sú počty výskytov P, E, a S. Toto zlepšenie nám však príliš nepomôže – v najhoršom prípade takéto riešenie ešte stále spraví počet krokov priamo úmerný tretej mocnine dĺžky vstupného reťazca. Totiž existujú vstupné reťazce dĺžky n, ktoré naozaj obsahujú rádovo n^3 výskytov slova PES. Napríklad taký reťazec dĺžky $n = 3k$, tvorený k znakmi P, k znakmi E a nakoniec k znakmi S. Z toho vyplýva, že ak chceme dosiahnuť lepšiu časovú zložitosť ako rádovo n^3, nesmieme výskyty slova PES rátať po jednom.

Jeden možný trik: Pre každé písmeno E v zadanom reťazci nájdeme počet takých výskytov PES, ktoré obsahujú toto E. Zjavne takto dokopy zarátame každý výskyt slova PES práve raz.

Pre konkrétne E každý výskyt slova PES dostaneme tak, že vezmeme nejaké písmeno P naľavo od nášho E a nejaké písmeno S napravo od nášho E. A tieto P a S vieme voliť nezávisle na sebe. Stačí nám teda len zistiť, koľko máme na výber P a koľko S, a tieto dve hodnoty vynásobiť. Takto dostávame algoritmus s časovou zložitosťou $O(n^2)$.

Listing programu:

```
var A: AnsiString;
    i, j, pocet_p, pocet_s, n, res: longint;

begin
    ReadLn(A);
    n := Length(A);
    res := 0;
    for i := 1 to n do if A[i] = 'E' then begin
        pocet_p := 0;  for j := 1 to i-1 do if A[j] = 'P' then inc(pocet_p);
        pocet_s := 0;  for j := i+1 to n do if A[j] = 'S' then inc(pocet_s);
        res := res + pocet_p * pocet_s;
    end;
    WriteLn(res);
end.
```

Časovú zložitosť vieme zlepšiť až na $O(n)$. Stačí si pre každú pozíciu predpočítať dve čísla: počet P od nej doľava a počet S od nej doprava. Presnejšie, v poli P si budeme na i-tej pozícii pamätať počet písmen P od začiatku reťazca po i-te písmeno. Hodnotu $P[i]$ ľahko zistíme v konštantnom čase z $P[i-1]$. Podobne si v poli S budeme pamätať počet S od i-teho písmena po koniec reťazca. Hodnoty v poli S spočítame v opačnom smere, od konca reťazca ku začiatku. Teda hodnotu $S[i]$ spočítame z i-teho znaku reťazca a z hodnoty $S[i+1]$.

Listing programu:

```
var A: AnsiString;
    n, i, res: longint;
    P, S: array of longint;

begin
    ReadLn(A);
    n := Length(A);
    SetLength(P,n+1); SetLength(S,n+1);
    P[0] := 0;
    for i := 1 to n do     if A[i] = 'P' then P[i] := P[i - 1] + 1 else P[i] := P[i - 1];
    S[n + 1] := 0;
    for i := n downto 1 do if A[i] = 'S' then S[i] := S[i + 1] + 1 else S[i] := S[i + 1];
    res := 0;
    for i := 1 to n do     if A[i] = 'E' then res := res + P[i] * S[i];
    WriteLn(res);
end.
```

Alternatívne vzorové riešenie

Budeme postupne prechádzať vstupný reťazec zľava doprava a v každom kroku si pamätať, koľko výskytov slov P, PE a PES bolo v zatiaľ spracovanom prefixe.

So slovami P je to jednoduché: ich počet sa zmení (zväčší o 1) iba v prípade, ak pridaným písmenom je práve P. Nové slovo PE nám môže pribudnúť len vtedy, keď načítame písmeno E. Počet slov PE sa potom zväčší o počet slov P v zatiaľ spracovanom prefixe reťazca. S novými výskytmi slova PES je to rovnaké: v prípade, že pridaným písmenom je S, ich počet sa zväčší o počet doteraz nájdených slov PE.

Nakoniec po spracovaní celého reťazca vypíšeme vypočítaný počet výskytov slova PES. Časová zložitosť tohto riešenia je $O(n)$. A navyše si pri ňom vystačíme s konštantnou pamäťou – vstupný reťazec stačí čítať po znakoch a rovno ich spracúvať.

Listing programu:

```
var c : char;
    P, PE, PES: longint;

begin
    P := 0; PE := 0; PES := 0;
    while not SeekEOF do begin
        read(c);
        if c='P' then P := P+1;
        if c='E' then PE := PE+P;
        if c='S' then PES := PES+PE;
    end;
    WriteLn(PES);
end.
```

B-II-3 Poriadok na polici II

Najprv popíšeme spôsob, ktorým budeme pre konkrétne poradie kníh argumentovať, že optimálne riešenie vyžaduje práve k výmen.

Podobne ako v riešení domáceho kola použijeme pojem *inverzia* pre označenie stavu kedy kniha s väčším číslom je (ľubovoľne ďaleko) naľavo od knihy s menším číslom. Napríklad v postupnosti $2, 4, 1, 6, 3, 5$ nájdeme 5 inverzií, konkrétne sú to dvojice 2–1, 4–1, 4–3, 6–3 a 6–5. Každá inverzia má tú vlastnosť, že v ľubovoľnom riešení musíme niekedy jej prvky vzájomne vymeniť, kedže skôr alebo neskôr sa menšie číslo musí dostať naľavo od väčšieho čísla. Preto každé preusporiadanie postupnosti do správneho poradia musí použiť minimálne toľko výmen, koľko inverzií sa v danej postupnosti nachádza.

Takéto riešenie vždy aj existuje. Presnejšie, každé riešenie, ktoré sa drží postupu opísaného v zadaní, je naozaj optimálne. Totiž ak postupnosť nie je usporiadaná, tak obsahuje aspoň jednu dvojicu **po sebe idúcich** kníh, ktoré tvoria inverziu. (Rozmyslite si, prečo.) Tým, že hocijakú takúto dvojicu vymeníme, znížime celkový počet inverzií o jednu. Každý algoritmus, ktorý takéto dvojice hľadá a vymieňa, teda vyrobí optimálne riešenie.

Tým sme dokázali, že ľubovoľné optimálne riešenie potrebuje na usporiadanie presne toľko výmen, koľko má postupnosť inverzií. Našou úlohou je teda k danému n a k nájsť postupnosť n čísel, ktorá má práve k inverzií.

Prvé dve podúlohy

Pre $n = 6$ a $k = 17$ žiadna vyhovujúca postupnosť neexistuje. To vyplýva z výsledku, ktorý sme si dokázali v domácom kole: každú postupnosť n čísel sa dá usporiadať na nanajvýš $n(n-1)/2$ výmen, teda pre $n = 6$ vždy stačí nanajvýš 15 výmen. (Iný pohľad: každá 6-prvková postupnosť má nanajvýš $5+4+3+2+1 = 15$ inverzií.)

Príkladom 10-prvkovej postupnosti, ktorá obsahuje 22 inverzií, je postupnosť $10, 9, 6, 1, 2, 3, 4, 5, 7, 8$.

Obsahuje nasledovné inverzie: 10–1, 10–2, ..., až 10–9 (9 inverzií), 9–1, 9–2, ..., až 9–8 (ďalších 8 inverzií), 6–1, 6–2, 6–3, 6–4 a 6–5. Samozrejme, existuje aj mnoho ďalších postupností. Postup, ako sme našli túto, popíšeme vo zvyšku vzorového riešenia.

Jednoduchšie ale pomalšie riešenie tretej podúlohy

Na vytvorenie n-prvkovej postupnosti, ktorá by obsahovala práve k inverzií, môžeme napríklad využiť postup presne opačný od toho, ktorý používame pri jej usporadúvaní. Začneme s usporiadanou postupnosťou, ktorá má 0 inverzií. Následne v každom kroku nájdeme a vymeníme dva susedné prvky tak, aby vytvorili inverziu. (Teda nájdeme dvojicu „menší väčší“ a zmeníme ju na „väčší menší“). Tým zvýšime počet inverzií v tejto postupnosti o 1. Toto zjavne môžeme robiť až kým nedostaneme obrátene usporiadanú postupnosť. A tá už obsahuje maximálny počet inverzií: $1 + 2 + \ldots + (n-1)$.

Kostra programu pre takéto riešenie by mohla vyzerať nasledovne:

```
if K > (N*(N-1) div 2) then begin writeln('Taka postupnost neexistuje'); halt; end;
for i:=1 to N do a[i]:=i;
for j:=1 to K do begin
    for i:=1 to N-1 do
        if a[i] < a[i+1] then begin
            swap(a[i],a[i+1]);   { vymenime hodnoty a[i] a a[i+1] }
            break;
        end;
end;
```

Časová zložitosť tohto riešenia je $O(kn)$, pretože k-krát vyhľadávame dvojicu susedných prvkov, ktoré vymeníme. No a pri každom hľadaní v najhoršom prípade prejdeme celú postupnosť – teda každé hľadanie trvá nanajvýš rádovo n krokov. Tento postup je najpomalší vtedy, keď je k najväčšie, čiže keď $k = n(n-1)/2$. Pre takto zvolené k je počet krokov tohto algoritmu kubickou funkciou čísla n, teda $\Theta(n^3)$.

Lepšie riešenie tretej podúlohy

Na predchádzajúcom riešení bolo neefektívne to, že sme dvojicu na výmenu hľadali vždy od začiatku. To vôbec nie je potrebné – stačí nám ľubovoľná taká dvojica. Program teda zrýchlime tak, že budeme po postupnosti dokola prechádzať od začiatku až úplne po koniec a **zakaždým**, keď stretneme vhodnú dvojicu, tak ju vymeníme.

V programe by to vyzeralo takto:

```
if K > (N*(N-1) div 2) then begin writeln('Taka postupnost neexistuje'); halt; end;
for i:=1 to N do a[i]:=i;
while K > 0 do begin
    for i:=1 to N-1 do
        if a[i] < a[i+1] then begin
            swap(a[i],a[i+1]);
            dec(K);
            if K=0 then break; { len ak uz koncime }
        end;
end;
```

Ako veľmi sme tým zrýchlili naše riešenie? Pozrime sa, čo sa napríklad stane pre $n = 6$ a najväčšie možné k.

- Začneme s postupnosťou $1, 2, 3, 4, 5, 6$.
- Počas prvej iterácie while-cyklu náš program postupne vymení dvojice 1–2 (teraz máme postupnosť 2,1,3,4,5,6), potom 1–3, ..., až 1–6 a dostane postupnosť $2, 3, 4, 5, 6, 1$.
- Počas druhej iterácie while-cyklu náš program postupne vymení dvojice 2–3, 2–4, ..., až 2–6 a dostane postupnosť $3, 4, 5, 6, 2, 1$.
- A už je zjavné, ako to bude pokračovať ďalej: po ďalších troch (teda dokopy piatich) iteráciách dostaneme postupnosť $6, 5, 4, 3, 2, 1$ a skončíme.

Zovšeobecnením tejto úvahy ľahko dokážeme, že pre konkrétne n sa nikdy nevykoná viac ako $n - 1$ iterácií while-cyklu, bez ohľadu na to, aké veľké bude k.

Časová zložitosť tohto algoritmu je teda v najhoršom prípade kvadratická od n, čiže $O(n^2)$.

Poznámka: Za povšimnutie stojí, že to, čo sme práve naprogramovali, je vlastne obyčajný triediaci algoritmus BubbleSort, postupne preusporadúvajúci prvky do poradia od najväčšieho po najmenší.

Vzorové riešenie tretej podúlohy

Existuje viacero riešení, ktoré vedia hľadané poradie čísel vyrobiť v optimálnom čase – lineárnom od n. Ukážeme si dve podobné. Prvé z nich dostaneme ďalším zrýchlením predchádzajúceho riešenia.

Môžeme si totiž všimnúť, čo presnejšie sa pri predchádzajúcom algoritme udeje. V prvej iterácii while-cyklu presunieme jednotku zo začiatku na koniec. Na to potrebujeme $n - 1$ výmen, a teda vyrobíme $n - 1$ inverzií. V druhej iterácii presunieme dvojku zo začiatku až skoro na koniec – tesne pred jednotku. Na to potrebujeme $n - 2$ výmen, a teda vyrobíme $n - 2$ inverzií. A tak ďalej. Prvých niekoľko prvkov posunieme najďalej, ako sa dá, a potom (v poslednej ešte vykonanej iterácii while-cyklu) jeden prvok posunieme toľkokrát, koľko inverzií nám ešte chýba.

Uvažujme napríklad $n = 6$ a $k = 13$. Pre tieto hodnoty si priebeh vyššie popísaného algoritmu môžeme zhrnúť nasledovne:

1. Presuň jednotku najďalej ako to ide, teda až na koniec (5 inverzií).
2. Presuň dvojku najďalej ako ide (ďalšie 4 inverzie).
3. Presuň trojku najďalej ako ide (ďalšie 3 inverzie, to už je dokopy 12).
4. Následne ešte vymeň štvorku s päťkou.

A teraz si už len stačí všimnúť, že prvé tri kroky predchádzajúceho riešenia vôbec netreba simulovať. Je predsa jasné, že ako bude vyzerať pole po tom, ako presunieme na koniec čísla 1, 2 a 3: na začiatku budú v rastúcom poradí čísla, ktoré sme nepresúvali, a za nimi v klesajúcom poradí tie, ktoré sme presúvali. V našom prípade teda $4, 5, 6, 3, 2, 1$. A takéto poradie vieme rovno vyrobiť.

A teda namiesto toho, aby sme simulovali každú jednu výmenu prvkov, si naše riešenie spočíta, koľko prvkov treba posunúť až na koniec, potom vyrobí taký obsah poľa, ktorý by vznikol ich posunutím, a následne ešte dorobí posledných pár inverzií.

Listing programu

```
procedure swap(var a,b : longint);
var c : longint;
begin c:=a; a:=b; b:=c; end;

var   a : array[1..1000] of longint;
      i,N,K,mam,presunul : longint;

begin
    if K > (N*(N-1) div 2) then
        writeln('Taka postupnost neexistuje')
    else begin
        { spocitame, kolko prvkov treba presunut az na koniec }
        mam := 0;                      { kolko inverzii uz mam }
        presunul := 0;         { kolko prvkov som uz presunul }
        while true do begin
            { dalsi prvok by vyrobil (N-presunul-1) inverzii }
            if (mam + N - presunul - 1) > K then break;
            mam := mam + N - presunul - 1;
            inc(presunul);
        end;

        { vyrobime pole po danom pocte presunov }
        for i:=1 to N-presunul do a[i] := i+presunul;
        for i:=1 to presunul do a[i+N-presunul] := presunul+1-i;

        { a uz len dorobime chybajuci pocet inverzii a vypiseme }
        for i:=1 to K-mam do swap(a[i],a[i+1]);
        for i:=1 to N do writeln(a[i]);
    end;
end.
```

Počas každej fázy (počítanie, výroba poľa, záverečné úpravy a výpis) je počet krokov tohto algoritmu priamo úmerný veľkosti poľa. Celková časová zložitosť je teda $O(n)$.

Stručnejšie vzorové riešenie tretej podúlohy

Na záver si ukážeme, ako jedno hľadané poradie ľahko zostrojiť po jednotlivých číslach zľava doprava. Keď už rozumieme tomu, ako vznikajú inverzie, bude to veľmi jednoduché.

Pozrime sa na hodnoty n a k. Ak $k \leq n-1$, vieme úlohu vyriešiť jednoducho: na prvé miesto dáme číslo $k+1$. To nám vyrobí presne k inverzií (ono samo s každým z čísel od 1 po k). Žiadne ďalšie inverzie už teda nechceme, preto ostatné čísla vypíšeme v rastúcom poradí a skončili sme.

A čo v opačnom prípade? Nech $k \geq n$, teda ešte treba viac inverzií ako vieme vyrobiť len prvým číslom. V takomto prípade určite nič nepokazíme, keď na začiatok umiestnime číslo n. To nám vyrobí najviac inverzií, presne $n-1$. A čo nám zostalo? Čísla 1 až $n-1$, z ktorých treba vyrobiť postupnosť s $k-(n-1)$ inverziami. A to je presne tá istá úloha ako na začiatku, len už máme menej čísel a menej inverzií. Zopakujeme teda odznova ten istý postup.

Listing programu

```
var i,N,K : longint;
begin
    readln(N,K);
    if K > (N*(N-1) div 2) then begin writeln('Taka postupnost neexistuje'); halt; end;
    while K > N-1 do begin writeln(N); dec(K,N-1); dec(N); end;
    writeln(K+1);
    for i:=1 to K do writeln(i);
    for i:=K+2 to N do writeln(i);
end.
```

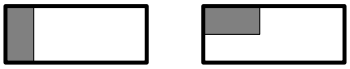
Časová zložitosť tohto programu je, rovnako ako v predchádzajúcom riešení, lineárna od n. Mimochodom, všimli ste si, že výsledné poradie čísel vyrobíme presne „opačne“ ako v predchádzajúcom riešení? V tomto riešení najskôr akoby presunieme n z konca na začiatok, potom $n-1$ z konca za n, a tak ďalej.

B-II-4 Dláždenie pre pokročilých

Dláždenia obdĺžnikov

Označme si počet vydláždení obdĺžnika $2 \times n$ ako D_n. Už vieme, že $D_1 = 1$, $D_2 = 2$ a $D_3 = 3$. Čo vieme povedať o D_n pre väčšie n?

Predstavme si, že ideme vyrobiť jedno konkrétne dláždenie obdĺžnika $2 \times n$. Ako prvé sa musíme rozhodnúť, či štvorček v ľavom hornom rohu pokryjeme zvislou alebo vodorovnou parketou:



Ak sme si vybrali zvislú parketu, zostal nám ešte nevydláždený (na obrázku biely) obdĺžnik rozmerov $2 \times (n-1)$. A pre ten máme na výber presne D_{n-1} rôznych dláždení.

Ak sme si vybrali vodorovnú parketu, pozrime sa teraz na štvorček v ľavom dolnom rohu. Tam už nemáme na výber: musíme ho tiež pokryť vodorovnou parketou. A akonáhle to spravíme, bude nevydláždená časť tvoriť obdĺžnik rozmerov $2 \times (n-2)$. V tomto prípade máme teda presne D_{n-2} možností, ako dokončiť naše dláždenie.

Tým sme práve dokázali, že pre každé $n \geq 2$ platí vzťah: $D_n = D_{n-1} + D_{n-2}$.

Pomocou neho môžeme ľahko spočítať požadované počty dláždení. Odpovede sú v nasledujúcej tabuľke:

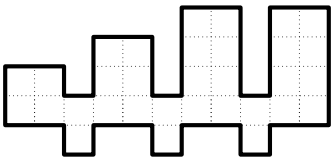
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
D_n	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987

Poznámka na záver: postupnosť, ktorú sme práve vypočítali, je známa pod menom Fibonacciho postupnosť.

Miestnosť so 150 dláždeniami

Číslo 150 si môžeme zapísať ako $2 \times 3 \times 5 \times 5$. Najjednoduchší spôsob, ako vyrobiť miestnosť so 150 dláždeniami, je vhodne spojiť štyri menšie miestnosti, ktoré budú mať 2, 3, 5 a 5 možných vydláždení. A takéto miestnosti už poznáme: sú to napríklad obdĺžniky 2×2, 2×3 a 2×4.

Jedno možné riešenie vyzerá nasledovne:

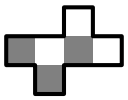


Štvorčeky v spodnom riadku zjavne musíme pokryť zvislými parketami. A tým, že tieto parkety položíme, sa nám miestnosť rozpadne na niekoľko nezávislých častí, ktoré môžeme dláždiť každú zvlášť. Preto celkový počet dláždení dostaneme vynásobením počtov dláždení pre jednotlivé časti.

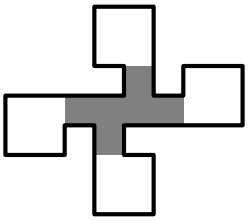
Nie každá miestnosť sa dá vydláždiť

V prvom rade zabudnime na druhú podmienku zo zadania a zamyslime sa nad vyváženými miestnosťami. Stačí samotná vyváženosť na to, aby sa miestnosť dala vydláždiť?

Ľahko nahliadneme, že nestačí. Existuje dokonca protipríklad tvorený iba šiestimi štvorčekmi:



Nás ale zaujímajú len také miestnosti, v ktorých nemáme štvorčeky s len jedným susedom. Skúsme teda náš protipríklad „vylepšiť“ – prirobíme každému z krajných štvorčekov susedov:



Je zjavné, že táto miestnosť je vyvážená a každý jej štvorček má aspoň dvoch susedov. Ale je tiež zjavné, že táto miestnosť nemá žiadne platné vydláždenie. (Nemôžeme položiť parketu tak, aby pokryla jedno políčko v bielom štvorci a jedno mimo neho, lebo by nám ostali tri biele políčka a tie nemáme ako vydláždiť. Musíme teda každý z pridaných štvorcov vydláždiť samostatne. Tým nám ale zostane miestnosť z predchádzajúceho obrázka a tá sa vydláždiť nedá.)

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX

© Slovenská komisia Olympiády v informatike, 2010