INFORMATIKA

Rozkopané křižovatky (Úlohy z MO kategorie P, 42. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

V dalším dílu cyklu zajímavých úloh z Matematické olympiády kategorie P (programování) se seznámíme s jednou praktickou úlohou z celostátního kola 45. ročníku MO (školní rok 1995/96). Úkolem v ní bylo určit počet různých cest mezi dvěma body v pravidelné pravoúhlé silniční síti, ve které jsou ovšem některé křižovatky neprůjezdné. K efektivnímu vyřešení této úlohy použijeme poměrně známou programátorskou techniku dynamického programování. Ukážeme si různé způsoby, jak lze takové řešení implementovat, a také skutečnost, že se tyto způsoby mohou lišit z hlediska výsledné časové složitosti. Stejné principy ovšem platí i pro řešení mnoha jiných úloh využívajících dynamické programování.

Pro potřeby našeho článku jsme původní zadání trochu upravili a zjednodušili, na principu úlohy a jejího řešení se tím ovšem nic nezměnilo.

* * * * * * * * * * * *

Ve městě vedou všechny ulice buď ze severu na jih, nebo ze západu na východ. Předpokládejme, že se všechny ulice táhnou na obě strany dostatečně daleko. V průsečíku každých dvou ulic různých směrů je křižovatka.

Křižovatku označíme dvojicí čísel (i, j), jestliže se jedná o křižovatku v pořadí i-té západovýchodní ulice počítáno od severu a j-té severojižní ulice počítáno od západu. Křižovatky jsou tedy očíslovány od $(1, 1)$ do (m, n), kde m je počet západovýchodních ulic a n je počet severojižních ulic. Na některých křižovatkách se pracuje na opravě vozovky, a proto přes ně není možné přejet. Na křižovatkách $(1, 1)$ a (m, n) se nepracuje.

Arpád vyjíždí každé ráno z křižovatky $(1, 1)$, kde bydlí, a potřebuje se dostat do své firmy, která sídlí na opačném konci města na křižovatce (m, n). Napište program, který zjistí, kolika různými cestami se může Arpád dostat z domova do své firmy, jestliže pojede vždy jen ve směru na jih nebo na východ (neboli jede vždy nejkratší možnou cestou) a neprojede žádnou z rozkopaných křižovatek.

Vstup

Na prvním řádku vstupu jsou uvedena celá čísla m a n, kde m je počet západovýchodních ulic a n je počet severojižních ulic. Na druhém řádku se nachází celé číslo k udávající počet křižovatek ve městě, na nichž se opravuje vozovka. Na každém z následujících k řádků jsou uvedena vždy dvě celá čísla, která určují polohu neprůjezdné křižovatky.

Výstup

Výstup programu je tvořen jedním celým číslem, které představuje hledaný počet různých cest.

Příklad 1

Vstup:	Výstup:
4 3 2 2 2 3 2	2

Příklad 2

Vstup:	Výstup:
2 4 2 1 2 2 3	0

Nejprve si musíme uvědomit, že Arpád při své cestě z křižovatky $(1, 1)$ na křižovatku (m, n) může jet pouze směrem na východ nebo na jih. To znamená, že na nerozkopanou křižovatku (i, j) může přijet jedině ze severu z křižovatky $(i - 1, j)$, pokud tato křižovatka existuje a není rozkopaná, nebo ze západu z křižovatky $(i, j-1)$, opět pokud tato křižovatka existuje a není rozkopaná. Počet různých cest vedoucích z výchozí křižovatky $(1, 1)$ na křižovatku (i, j) proto získáme tak, že sečteme počet cest vedoucích

z křižovatky $(1,1)$ na křižovatku $(i-1,j)$ a počet cest vedoucích z křižovatky $(1,1)$ na křižovatku $(i,j-1)$. Počet přípustných cest vedoucích z křižovatky $(1,1)$ na jakoukoliv neexistující nebo rozkopanou křižovatku je pochopitelně roven nule a počet cest vedoucích na počáteční křižovatku $(1,1)$ je 1.

Tato úvaha nás již rovnou vede k jednoduchému řešení úlohy. Celou úlohu nám vyřeší rekurzivní funkce *Cesty* se dvěma celočíselnými parametry i, j, která přímo implementuje výše uvedenou úvahu a vrací počet různých cest vedoucích z výchozí křižovatky $(1,1)$ na zvolenou křižovatku (i,j). Pro získání hledaného výsledku funkci v programu zavoláme s dvojicí vstupních parametrů m, n.

```
def Cesty(i, j):
    if i == 0 or j == 0: return 0 # neexistující křižovatka
    if r[i][j]: return 0          # rozkopaná křižovatka
    if i == 1 and j == 1: return 1   # počáteční křižovatka
    return Cesty(i-1, j) + Cesty(i, j-1)

m, n = [int(_) for _ in input().split()]
k = int(input())
r = [[False]*(n+1) for _ in range(m+1)]  # rozkopané křižovatky
for p in range(k):
    i, j = [int(_) for _ in input().split()]
    r[i][j] = True

print(Cesty(m, n))
```

Uvedené řešení je teoreticky neprosto správné, ale pro větší město je značně neefektivní. Jeho neefektivita spočívá v tom, že pro mnohé dvojice vstupních parametrů je funkce *Cesty* během jednoho výpočtu programu zbytečně zavolána opakovaně vícekrát. Pokud například $m = n = 100$ a v programu tedy zavoláme *Cesty*(100, 100), funkce rekurzivně zavolá *Cesty*$(99,100)$ a *Cesty*$(100,99)$. Funkce volaná jako *Cesty*$(99,100)$ rekurzivně zavolá sama sebe dvakrát, a to se vstupními parametry $(98,100)$ a $(99,99)$. Funkce volaná *Cesty*$(100,99)$ provede také dvě rekurzivní volání – první se vstupními parametry (99, 99) a druhé s parametry $(100,98)$. Jak tedy vidíme, postupně po sobě se dvakrát vykonalo stejné volání *Cesty*$(99,99)$. Zbytečně se tak dvakrát provádí tentýž časově velmi náročný výpočet, pokaždé pochopitelně se shodným výsledkem. Obdobný problém s neefektivitou se dále mnohokrát opakuje i při volání funkce *Cesty* s nižšími hodnotami vstupních parametrů, počet zbytečně vykonaných shodných volání dokonce ještě narůstá.

Výše zapsaná funkce *Cesty* má v nejhorším případě exponenciální asymptotickou časovou složitost. To si můžeme názorně ukázat pro případ vstupních dat $m = n$, $k = 0$, tedy ve čtvercovém městě bez rozkopaných křižovatek. Představíme-li si strom všech provedených rekurzivních volání funkce *Cesty*, kde kořen odpovídá zavolání se vstupními parametry (n, n), až do hloubky rekurze $n-1$ provede funkce vždy dvě rekurzivní volání. V hloubce $n-1$ má tedy tento binární strom 2^{n-1} uzlů. Celkový počet všech provedených volání funkce *Cesty* do hloubky rekurze $n-1$ včetně je proto roven $1+2+4+\ldots+2^{n-1}$, což je 2^n-1. Strom všech volání funkce pak pokračuje dalšími uzly až do hloubky $2n-2$, přičemž vzhledem k efektům na okraji sítě ulic v některých případech funkce provede dvě a v některých už jenom jedno rekurzivní volání. Přesný počet všech vykonaných volání funkce *Cesty* ale nemusíme odvozovat, už z prvních $n-1$ hladin stromu je zjevné, že je tento počet skutečně exponenciální vzhledem k n.

Pro řešení podobných situací, kdy rekurzivní funkce zbytečně volá opakovaně sama sebe se stejnými parametry, s jakými již byla dříve zavolána, používáme programovací techniku označovanou jako memoizace (odvozeno z anglického memory = paměť) nebo také kešování mezivýsledků (odvozeno z anglického cache = vyrovnávací paměť). Je založena na velmi jednoduché myšlence. Náš rekurzivní algoritmus doplníme pomocným polem, ve kterém si budeme ukládat všechny hodnoty, které rekurzivní funkce již někdy spočítala. Před provedením rekurzivního volání pokaždé zkontrolujeme, zda jsme již někdy dříve nespočítali tu hodnotu, kterou právě potřebujeme. Jestliže ano, vezmeme hodnotu z pole a neprovádíme rekurzivní volání. Jestliže potřebnou hodnotu ještě neznáme, funkci normálně rekurzivně zavoláme a po skončení jejího výpočtu výsledek nejen vrátíme z funkce jako návratovou hodnotu, ale také ho uložíme do našeho pomocného pole pro případné budoucí využití.

Algoritmus výpočtu tedy zůstane v principu stejný, jako byl dosud, nadále má podobu rekurzivní funkce. Asymptotická časová složitost řešení se ovšem výrazně snížila, neboť pro každou přípustnou hodnotu vstupních parametrů bude funkce zavolána nejvýše jednou. V naší úloze s Arpádem a křižovatkami bude takových volání nejvýše $m \cdot n$ a protože každé z nich se vykoná v konstantním čase, asymptotická časová složitost upraveného algoritmu se sníží na velmi příznivou hodnotu $\mathrm{O}(m \cdot n)$ v nejhorším případě. Opět si ukážeme programovou realizaci uvedeného postupu.

```
def Cesty(i, j):
    if i == 0 or j == 0: return 0 # neexistující křižovatka
```

```
    if r[i][j]: return 0                    # rozkopaná křižovatka
    if a[i][j] == -1:
        a[i][j] = Cesty(i-1, j) + Cesty(i, j-1)
    return a[i][j]

m, n = [int(_) for _ in input().split()]
k = int(input())
r = [[False]*(n+1) for _ in range(m+1)]  # rozkopané křižovatky
a = [[-1]*(n+1) for _ in range(m+1)]  # počet cest (1,1)->(i,j)
a[1][1] = 1                                 # počáteční křižovatka
for p in range(k):
    i, j = [int(_) for _ in input().split()]
    r[i][j] = True

print(Cesty(m, n))
```

Po skončení výpočtu výše uvedeného programu udávají prvky pole $a[i, j]$, kolika různými cestami může jet Arpád z křižovatky $(1, 1)$ na křižovatku (i, j). Při řešení úloh podobného typu často nepoužijeme rekurzivní funkci, ale hodnoty $a[i, j]$ namísto toho počítáme iteračně ve vhodném pořadí, tzn. od menších indexů k větším. Nejdříve snadno stanovíme hodnoty pole a v řádku 1 a ve sloupci 1. Po první západovýchodní ulici můžeme jet pouze jediným způsobem od křižovatky $(1, 1)$ směrem východním, dokud poprvé nenarazíme na rozkopanou křižovatku. Od této křižovatky dále na východ se už nedostaneme. Obdobná úvaha platí i pro první severojižní ulici.

Hodnoty $a[i, j]$ pro všechny zbývající křižovatky ve městě, tzn. pro i, $j > 1$, spočítáme na základě úvahy, kterou jsme provedli hned na začátku našeho prvního řešení. Pokud je křižovatka (i, j) rozkopaná, položíme $a[i, j] = 0$. Jestliže rozkopaná není, můžeme na ni přijít ze severu z křižovatky $(i - 1, j)$, nebo ze západu z křižovatky $(i, j - 1)$. Hodnotu $a[i, j]$ tedy spočítáme jako součet $a[i-1, j]+a[i, j-1]$. Pole a budeme zaplňovat ve vhodném pořadí, například po řádcích zleva doprava, abychom při výpočtu $a[i, j]$ již znali obě potřebné hodnoty $a[i - 1, j]$ a $a[i, j - 1]$. Po vyplnění celého pole a bude výsledek úlohy uložen v prvku $a[m, n]$.

```
m, n = [int(_) for _ in input().split()]
k = int(input())
r = [[False]*(n+1) for _ in range(m+1)]  # rozkopané křižovatky
a = [[-1]*(n+1) for _ in range(m+1)]  # počet cest (1,1)->(i,j)
for p in range(k):
    i, j = [int(_) for _ in input().split()]
    r[i][j] = True
```

```
a[1][1] = 1                                  # počáteční křižovatka
for i in range(2, m+1):
    a[i][1] = 0 if r[i][1] else a[i-1][1]
for j in range(2, n+1):
    a[1][j] = 0 if r[1][j] else a[1][j-1]

for i in range(2, m+1):
    for j in range(2, n+1):
        a[i][j] = 0 if r[i][j] else a[i-1][j] + a[i][j-1]

print(a[m][n])
```

Pro vyřešení úlohy musíme vyplnit celé pole a velikosti $m \times n$ a při výpočtu hodnoty každého prvku vykonáme konstantní počet operací. Asymptotická časová složitost algoritmu je proto $O(m \cdot n)$. Je tedy stejná, jako v našem druhém řešení s rekurzí a pomocnou pamětí. To není nijak překvapující, neboť v obou případech se prováděl stejný výpočet stejných hodnot, jenom v jiném pořadí a s jinou organizací práce.

Popsaný způsob řešení úlohy označujeme jako *dynamické programování*. Spočívá v tom, že před vyřešením zadaného většího problému nejprve vyřešíme menší a tím i snadnější úlohy stejného typu a s využitím jejich řešení pak již dokážeme snadno určit hledaný výsledek. Stejný postup se aplikuje opakovaně na takto získané menší úlohy, až celý původně zadaný problém postupně rozložíme na tak malé části, které dokážeme vyřešit přímo. Z jejich řešení sestavíme řešení o něco větších podúloh, pomocí nich vyřešíme ty ještě větší, a takto pokračujeme až k vyřešení celé původní úlohy.

Naše druhé a třetí řešení úlohy s Arpádem představují dva základní přístupy, jak lze dynamické programování implementovat – buď rekurzívně shora, nebo iteračně zdola. Mezi oběma přístupy máme zpravidla na výběr a obě možnosti vedou ke stejné asymptotické časové složitosti v nejhorším případě. Použití iteračního přístupu bývá o něco častější, takto navržený program bývá pro většinu programátorů jednodušší a přehlednější. Jak jsme viděli ve třetím řešení naší dnešní úlohy, vynecháme zde rekurzivní rozklad na podúlohy, ale začneme rovnou od vhodně zvolených malých úloh, pomocí jejich řešení vyřešíme úlohy větší, a tak postupujeme dále až k vyřešení celého původně zadaného problému.

Ačkoliv obě varianty implementace dynamického programování mají stejnou asymptotickou časovou složitost v nejhorším případě, rekurzivní přístup může být u některých úloh a pro některá vstupní data časově výhodnější. Je tomu tak i u naší dnešní úlohy s rozkopanými křižovatkami. Při iteračním řešení úlohy budeme postupně počítat všech $m \cdot n$ hodnot

v poli a, takže výsledný program bude mít asymptotickou časovou složitost $\mathrm{O}(m \cdot n)$ nejen v nejhorším, ale v každém případě. Pro některá vstupní data ale mnohé z těchto hodnot počítáme zbytečně, protože je nakonec nevyužijeme, ve výsledku se nijak neprojeví. Jsou to hodnoty těch křižovatek (i, j), ze kterých se vůbec nedá dojet na cílovou křižovatku (m, n). Při postupném zaplňování pole a ale předem nevíme, o které křižovatky se jedná, takže pro jistotu počítáme všech $m \cdot n$ hodnot a ukládáme je všechny do pole a.

Naproti tomu při rekurzivním řešení úlohy můžeme výpočet některých nepotřebných hodnot ušetřit. Uvažujme například vstupní data $k = m-1$, rozkopané jsou křižovatky $(2, n-1)$, $(3, n-1)$, $(4, n-1)$, …, $(m, n-1)$, tedy celá předposlední severojižní ulice s výjimkou její první křižovatky $(1, n-1)$. V takovém případě bude výsledkem úlohy číslo 1, neboť pro Arpáda existuje jediná možná cesta z domova do práce (nakreslete si obrázek!). Po zavolání rekurzivní funkce *Cesty*(m, n) se pro získání tohoto výsledku provede nejprve volání funkce postupně na křižovatky $(m-1, n)$, $(m-2, n)$, …, $(1, n)$ a poté na křižovatky $(1, n-1)$, $(1, n-2)$, …, $(1, 1)$. Funkce *Cesty* bude tedy v tomto případě zavolána jenom přibližně $(m+n)$--krát a celý výpočet tak bude mít časovou složitost $\mathrm{O}(m + n)$. Uvedenou časovou úsporu ovšem nezískáme pokaždé, jak ukazuje následující příklad. Mějme opět $k = m - 1$, rozkopané jsou tentokrát křižovatky $(1, 2)$, $(2, 2)$, $(3, 2)$, …, $(m-1, 2)$, tedy celá druhá severojižní ulice s výjimkou její poslední křižovatky $(m, 2)$. Rovněž v tomto případě bude výsledkem úlohy číslo 1, Arpád má opět k dispozici jedinou možnou cestu (nakreslete si sami situaci). Počet vykonaných rekurzivních volání funkce *Cesty* bude ovšem v tomto případě zhruba $m \cdot n$.

Náš dnešní článek zakončíme jednou technickou poznámkou. Pokud si zkusíte sami naprogramovat kterýkoliv z uvedených algoritmů třeba v C++, Pascalu nebo Javě a program spustíte na počítači, nedivte se, že výpočet pro mnohá vstupní data skončí s běhovou chybou aritmetického přetečení. To není závadou použitého algoritmu, ale prostým důsledkem skutečnosti, že převážná většina programovacích jazyků pracuje s celočíselnými hodnotami pouze omezené velikosti. Jsou-li rozměry města m, n hodně velké a počet rozkopaných křižovatek k dostatečně nízký, potom počty přípustných cest pro Arpáda velmi rychle narůstají a ukládané hodnoty $a[i, j]$ snadno překročí obvyklý rozsah celočíselných proměnných. Z běžně užívaných programovacích jazyků jedině Python umožňuje pracovat s libovolně velkými celými čísly, proto jsme ho také použili v na-

šich programových ukázkách. Jiné programovací jazyky ukládají celočíselné hodnoty standardně do 4 bytů, kde 1 bit určuje znaménko a zbývajících 31 bitů obsahuje uloženou hodnotu vyjádřenou ve dvojkové soustavě. Maximální zobrazitelná celočíselná hodnota je v takovém případě rovna $2^{31} - 1 = 2147483647$. Již ve čtvercovém městě velikosti 18×18 ulic bez rozkopaných křižovatek vychází počet možných cest 2333606220 a je tedy vyšší než $2^{31} - 1$. Úloha zadaná v olympiádě vyžadovala napsat program, který zvládne vyřešit všechna města až do velikosti 100×100 ulic. Pro takto velké město bez rozkopaných křižovatek dostaneme výsledek 227508830794229349661819540395688853956041682601541047340000 (což je hodnota řádově 10^{58}). V době konání soutěže před 25 lety ovšem Python ještě neexistoval, takže soutěžící používající programovací jazyky C, C++ nebo Pascal si museli navíc naprogramovat vlastní celočíselnou aritmetiku pracující s mnohacifernými čísly.

Jak moc jsou okružáci okrouhlí?

LUDĚK SPÍCHAL

Česká lesnická akademie, Trutnov

Spirály různých tvarů nalezneme ve svém okolí velmi běžně. Přestože spirál je celá řada různých typů, v přírodních útvarech zaznamenáme převážně spirálu logaritmickou.

V minulosti byla u řady zvířat (např. ve tvarech rohů turovitých, špičáků divokých prasat, tvarech ulit a mušlí měkkýšů), v lidské anatomii (např. tvar žeber), ale také v botanice (např. v postavení listů na lodyze, v postavení šupin při tvorbě šišek) prokázána přítomnost logaritmické spirály [4, 5, 8]. Tvar a vlastnosti logaritmické spirály jsou využívány rovněž při konstrukci různých technických zařízení, která vyžadují stálý (tečný) úhel dotyku (např. řezací nástroje, lezecké vačky – tzv. friendy, ozubená kola v kuželových soukolích) [1]. Rozmanitost oblastí výskytu logaritmické spirály je skutečně udivující, např. [2, 3, 8].

Mezi měkkýši vyskytujícími se v České republice nalezneme řadu skupin tvořících spirální ulity s různým způsobem uspořádání závitů. Zástupci čeledi okružákovitých (Planorbidae) vynikají schránkami (obr. 3, 4, 5, 6, 7),