
MO-P 2002

Soutěž dětí a mládeže v programování

kategorie starší žáci

obvodní kolo — obvod Praha 1

Zadání a řešení úloh

Tento dokument se pokouší vyložit řešení úloh řešených v obvodním kole soutěže v programování kategorie starší žáci, pořádané v rámci Prahy 1. Nejde o vyčerpávající vysvětlení a od čtenářů předpokládá určitou znalost programátorských postupů (alespoň takovou, jaká je očekávatelná od účastníků soutěže v programování).

Oficiální stránka oblastních kol soutěží v programování na Praze 1 je na

<http://voda.webz.cz/mo-p/index.html>

1 Strašidla

1.1 zadání

Město Hauntry bylo pravoúhlou sítí ulic rozděleno na M krát N obytných domů. Nebydlel-li v domě nikdo, záhy se tam usadila strašidla. Starosta města si nechal vypracovat plán města s počty obyvatel jednotlivých domů a položil si zajímavou otázku: kde je největší obdélník složený ze samých strašidelných domů?

Na vás je, abyste sestrojili program, který dostane na vstupu rozměry města (dvě čísla – první udává vodorovný, druhé svislý rozměr), poté čísla, která po řádcích představují počet obyvatel v jednotlivých domech a poté nalezne největší (co se týká plochy) obdélník domů, v nichž nikdo nebydlí (tj. počet obyvatel v každém z domů v tomto obdélníku je nulový).

Př.:

Vstup:

```
5 3
12 12 0 0 5
0 1 0 0 0
10 10 1 0 0
```

Výstup:

Největší neobydlený obdélník obsahuje 4 bloky a je mezi souřadnicemi (3,1) a (4,2).
odpovídá tomuto nalezenému obdélníku:

```
12 12 

|   |   |
|---|---|
| 0 | 0 |
| 0 | 0 |

 5
0 1 

|   |   |
|---|---|
| 0 | 0 |
| 0 | 0 |

 0
10 10 1 0 0
```

(Druhá možnost:)

Největší neobydlený obdélník obsahuje 4 bloky a je mezi souřadnicemi (4,2) a (5,3).
odpovídá tomuto nalezenému obdélníku:

```
12 12 0 0 5
0 1 0 

|   |   |
|---|---|
| 0 | 0 |
| 0 | 0 |


10 10 1 

|   |   |
|---|---|
| 0 | 0 |
|---|---|


```

1.2 řešení

Řešení této úlohy mohou být různá. Věnovat se tady budeme dvěma způsobům. Jednomu velmi jednoduchému a druhému trochu složitějšímu. V obou řešeních budeme hledat jediný obdélník, i když jich může existovat více se stejným obsahem, které vyhovují zadání.

Začneme přímočarým řešením: vyjdeme z jednoduché myšlenky, že chceme najít nějaký obdélník v zadaném vstupu. Proto můžeme vyzkoušet každý obdélník, který ve městě existuje. Každé dva domy ve městě určují obdélník. Pro každý takový obdélník zkontrolujeme, zda domy uvnitř něho jsou bez obyvatel (jsou v nich strašidla). Z nich vybereme ten s největším obsahem. K tomu nám stačí jedna proměnná pro obsah (označme MAX_OBSAH) a čtyři proměnné určující roh jednoho z obdélníků maximálního obsahu.

Nyní přejdeme k řešení trochu složitějšímu. Není příliš složité, pouze je potřeba využít více jednoduchých kroků. Předpokládejme na chvíli, že jsme již hledaný obdélník našli. Co musí splňovat? Na každé z jeho 4 stran existuje dům takový, že sousedí s obydleným domem nebo s hranicí města. Kdyby tomu tak nebylo, tak můžeme náš obdélník rozšířit (a tedy náš obdélník je menší než největší

možný, což je v rozporu s naším předpokladem). Abychom sjednotili obě možnosti (máme na mysli obydlený dům a hranice města), rozšíříme město o jednu řadu domů na každé straně. Do každého nového domu nastěhujeme jednoho obyvatele. Nyní tedy platí, že v maximálním obdélníku na každé jeho straně existuje dům, jehož sousední dům je obydlený.

Abychom výpočet urychlili, spočítáme si některé informace předem. Pro každý dům ve městě spočítáme, kolik domů nad tímto domem je neobydlených (aniž by se mezi nimi vyskytoval obydlený dům). Pro obydlený dům je toto číslo rovné 0. Určení bude jednoduché. Začneme horní řadou domů. Pokud je dům obydlený, tak mu přiřadíme 0. Pokud je dům prázdný, přiřadíme 1. Pro domy v dalších řadách určíme čísla takto: pokud je dům obydlený, tak přiřaď 0; v opačném případě přiřaď o 1 vyšší hodnotu, než jaká je přiřazena domu o řadu nad ním. Označme takto získaná čísla NAHORU.

Stejně tak určíme počet domů, které jsou neobydlené pod každým domem. Označme tyto počty DOLU. Toto pole však musíme vytvářet zdola nahoru. V následující tabulce jsou pro demonstraci tato čísla spočítána. Velké cifry představují počty obyvatel v jednotlivých domech, horní indexy určují u každého domu hodnotu NAHORU, dolní hodnotu DOLU.

1_0^0	0_4^1	1_0^0	2_0^0
0_1^1	0_3^2	0_3^1	3_0^0
2_0^0	0_2^3	0_2^2	4_0^0
4_0^0	0_1^4	0_1^3	1_0^0

Nyní další vlastnost. Uvažujme libovolný obdélník z neobydlených domů. Zvolme libovolný dům na jeho levé straně. Označme A počet domů od zvoleného domu k severní straně obdélníku včetně zvoleného domu. Stejně tak označme B počet domů od zvoleného domu k jižní straně obdélníku včetně zvoleného domu. Pokud budeme postupovat vodorovně zleva doprava až k pravé straně obdélníka, tak pro každý dům při tomto průchodu musí platit, že $NAHORU \geq A$ a $DOLU \geq B$ (každý z procházených domů má své vlastní A , B , $NAHORU$ i $DOLU$). Označme takový průchod jako cesta.

V následující tabulce je naznačen obdélník neobydlených domů, šedou barvou jsou v tomto obdélníku znázorněny dva domy. Obdélník má výšku 3, $A = 2$ a $B = 2$. Pokud se podíváte na hodnoty $NAHORU$ a $DOLU$, zjistíte, že splňují nerovnosti zapsané v předchozím odstavci.

1_0^0	0_4^1	1_0^0	2_0^0
0_1^1	0_3^2	0_3^1	3_0^0
2_0^0	0_2^3	0_2^2	4_0^0
4_0^0	0_1^4	0_1^3	1_0^0

Nyní již máme všechny potřebné vlastnosti hledaného obdélníka a můžeme začít psát program. Budeme v něm postupovat opačně, než jak je tady napsáno. Budeme procházet mapu po řádcích, testovat všechny možné cesty a hledat k nim obdélníky. Postup bude jednoduchý. Najdeme obydlený dům, od kterého bude těsně vpravo neobydlený dům. Tento neobydlený dům bude začátek naší cesty. Konec cesty může být libovolný dům při postupu doprava, dokud při procházení nedojdeme k obydlenému domu. Takto projdeme všechny možné obdélníky (některé vícekrát). Teď už nám zbývá vybrat z možných ten největší. K tomu využijeme hodnoty NAHORU a DOLU pro každý dům na cestě.

Nyní již k samotnému výpočtu. Nejprve vstupní dvourozměrné pole rozšíříme o 1 sloupec vlevo i vpravo a 1 sloupec nahoře a dole. Nastavíme počet obyvatel v přidávaných domech na 1. Vytvoříme a vyplníme další dvě dvourozměrná pole – do jednoho uložíme hodnoty NAHORU a do druhého hodnoty DOLU. Postupně projdeme všechny obydlené domy na cestě. Pokud je vpravo vedle obydleného domu neobydlený dům, začíná tam nějaká cesta. Vyzkoušíme postupně všechny cesty. Na prohledávané cestě žádný obydlený dům není, proto další neobydlený dům budeme hledat na konci procházené cesty. Tímto dosáhneme toho, že se nemusíme vracet – stačí procházet pole po řádcích, vždy zprava doleva.

Až projdeme všechny obydlené domy a jim příslušné cesty, tak máme práci hotovou. Stačí už jen vypsát maximální obdélník.

Poznámka na závěr: Druhé řešení je poměrně náročné, zkuste ho alespoň pochopit. Existuje mnoho dalších řešení, která řeší tento problém. Uvedené řešení je ale výjimečné v tom, že nelze vytvořit žádný postup, který by byl příliš rychlejší. Všimněte si, že v celém výpočtu pouze třikrát čteme vstupní hodnoty (porovnejte si s prvním uvedeným řešením).

-pm-

2 Náhrdelník

2.1 zadání

Rozmarná princezna nosí na krku náhrdelník z perel. Jelikož se jí často stane, že se jí při hře na zahradě náhrdelník rozsype, očíslovala si perly čísly od jedné do N . Vždy, když se jí náhrdelník rozsype, musí sluhové perly posbírat, zjistit, zda nějaká nechybí a pak jej znovu svázat.

Často se stane, že chybí právě jedna perla. Pak ji musí sluhové hledat tak dlouho, dokud ji nenajdou, přičemž hledají perlu s konkrétním číslem a jiné nalezené perly si mohou ponechat.

Vášim úkolem je napsat program, který pro dané N a neuspořádanou posloupnost čísel jedna až N zjistí, zda některé číslo v posloupnosti chybí a v případě,

že ano, tak které. Chybějící číslo je maximálně jedno a žádné číslo se v posloupnosti neopakuje.

Př.:

Vstup:

10

5 2 10 3 7 6 1 8 9 4

Výstup:

Žádné číslo nechybí

Vstup:

12

12 11 10 9 8 7 5 4 3 2 6

Výstup:

Chybí číslo 1

2.2 řešení

Na této úloze můžeme dobře vidět, jak je důležité pozorné přečtení zadání a využití všech poskytnutých informací. Při prvním pohledu se může zdát, že nejlepší bude načíst čísla nalezených perel, setřídít, setříděnou posloupnost projít a vypsat chybějící čísla. Tento postup však není ani úsporný, ani rychlý. O něco lepší algoritmus si připraví pole s položkou pro každou perlu, inicializuje všechny položky na nulovou hodnotu, při načítání potom nastavuje položky odpovídající načítaným hodnotám na jedničku. Na konci pole projde a vypíše položky, které obsahují nulu (ty se ve vstupu nevyskytly). Napišme si algoritmus:

Přečti počet

Inicializuj počet položek pole na 0

Opakuj (počet - 1) krát: {

 Přečti číslo

 Nastav pole[číslo] na 1

}

Pro i od 1 do počet {

 Pokud pole[i] rovno 0, pak {

 Vypiš i

 }

}

Toto řešení bude o něco rychlejší než řešení s tříděním (navíc pro každou perlu stačí hodnota 0/1, tedy jeden bit, takže když se budeme snažit, můžeme ušetřit i dost paměti, neboť do jednoho celočíselného typu se vejde 16 až 32 bitů, podle počítače, operačního systému a překladače).

Doposud jsme ale nezohlednili jednu důležitou informaci. Totiž tu, že chybějící číslo je nejvýš jedno. Její využití nám přitom podstatně zjednoduší práci: když na začátku uložíme do proměnné součet čísel od 1 do N (buď v cyklu, nebo použitím vzorce $N * (N + 1) / 2$) a potom od této proměnné odečítáme načítané hodnoty, na konci nám v proměnné zbyde buď nula (to znamená, že na vstupu nechybělo žádné číslo), v opačném případě tam zbyde chybějící hodnota. Zdůvodnění neuvádím, zkuste si promyslet. Pro úplnost uvedme algoritmus:

```
Přečti počet
součet = počet * (počet - 1) / 2
Opakuj (počet - 1) krát: {
    Přečti číslo
    součet = součet - číslo
}
Pokud součet roven 0, pak {
    Napiš, že nic nechybí
} jinak {
    Napiš, že chybí perla součet
}
```

Čeho jsme dosáhli? Jednak jsme výrazně snížili objem spotřebované paměti (pro celý výpočet nyní potřebujeme tři proměnné, jednu pro načítání, jednu pro načtení počtu perel, jednu pro součet a jednu pro načítání čísel jednotlivých perel), dále jsme ušetřili i čas: výpočet probíhá souběžně s načítáním čísel a ihned po načtení posledního čísla umíme dát správnou odpověď (na rozdíl od předchozích řešení, kdy bylo nutno setřídít, nebo alespoň projít pomocná data). Jak je vidět, vyplatí se před samotným řešením dobře pročíst zadání problému a promyslet, jak nám informace poskytnuté v zadání mohou pomoci při řešení problému.

3 Letiště

3.1 zadání

V království XY potřebují proinvestovat 100 miliard frnků. Rozhodli se proto, že je utratí za výstavbu nového letiště. Letiště je potřeba stavět v obydlých oblastech, na druhou stranu ale provoz letiště ruší všechny, kdo bydlí poblíž. Rada starších proto rozhodla, že v zájmu duševního zdraví obyvatel postaví letiště v nějaké obci tak, aby bylo hlukem ovlivněno co nejméně obyvatel. Ovlivnění jsou přitom všichni do vzdálenosti pěti kilometrů (včetně). Obyvatelé obce, jejíž náves byla pro stavbu letiště použita, budou přitom nadšení, a proto se nezapočítávají.

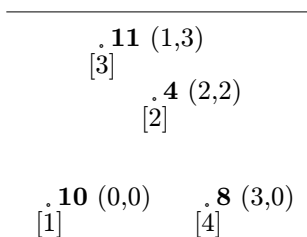
Napište program, který na vstupu dostane počet obcí v království, následovaný příslušným počtem trojic, z nichž každá je ve tvaru x, y, n, kde x, y jsou souřadnice v kilometrické síti, n počet obyvatel. Program určí, v které obci bude letiště postaveno.

Př.:

Vstup:

```
4
0 0 10
2 2 4
1 3 11
3 0 8
```

popisuje následující situaci (body představují jednotlivé obce, silná čísla počty jejich obyvatel a čísla v závorkách souřadnice):



Jelikož všechny obce jsou od sebe vzdáleny méně než pět kilometrů, je nejlepším řešením zvolit obec s největším počtem obyvatel (proč?).

Výstup:

Letiště je nejlepší postavit v 3. obci, ovlivněno bude 22 obyvatel.

Vstup:

```
4
7 0 4
5 0 4
3 0 10
0 3 10
4 5 10
6 5 4
8 5 4
11 2 10
```

popisuje následující situaci (body představují jednotlivé obce, silná čísla počty jejich obyvatel a čísla v závorkách souřadnice):

<u>. 10 (4,5)</u>	<u>. 4 (6,5)</u>	<u>. 4 (8,5)</u>
[5]	[6]	[7]

. 10 (0,3)
[4]

. 10 (11,2)
[8]

<u>. 10 (3,0)</u>	<u>. 4 (5,0)</u>	<u>. 4 (7,0)</u>
[3]	[2]	[1]

Výstup:

Letiště je nejlepší postavit v 8. obci, ovlivněno bude 8 obyvatel.

3.2 řešení

Jedná se o typický příklad, ve kterém „není co řešit“. Alespoň v případě, kdy je plocha království malá a/nebo je malý počet obcí v království. Na tento případ se nyní omezíme, posléze si povíme něco málo o tom, co se dá dělat v případě, že je zadání větší.

Naším cílem je najít pro každou obec sousední obce do vzdálenosti pěti kilometrů a sečíst počet jejich obyvatel. Vypsát potom máme obec, jejíž okolí obývá nejmenší počet obyvatel. V praxi nám nezbyvá, než projít všechny dvojice obcí, otestovat, zda mají vzájemnou vzdálenost menší než 5 km (vzdálenost dvou bodů se známými souřadnicemi ve čtvercové síti se snadno spočítá pomocí Pythagorovy věty) a pokud ano, přičíst k okolí obou členů dvojice počet obyvatel druhého člena dvojice. Během tohoto sčítání si budeme pamatovat minimum, kterého jsme doposud dosáhli spolu s místem, kde jsme tohoto minima dosáhli. Na konci tyto dva údaje vypíšeme.

Shrňme tento postup do náznaku algoritmu č.1:

```
[ Datová struktura pro každou obec obsahuje její souřadnice
  a počet jejích obyvatel, tedy x,y, počet_obyvatel ]
```

```
Minimum=nekonečno
```

```
KdeMinimum=??
```

```
Načti obce
```

```
Pro každou obec (proměnná i): {
```



```

Okolí = 0
Pro každou obec (proměnná j): pokud i <> j, pak {
    Rozdíl_x = i.x - j.x
    Rozdíl_y = i.y - j.y
    Pokud Odmocnina(Rozdíl_x*Rozdíl_x+Rozdíl_y*Rozdíl_y)<=5, pak {
        Okolí = Okolí + j.počet_obyvatel
    }
}
Pokud Okolí<Minimum, pak {
    Minimum = Okolí
    KdeMinimum = i
}
}
Vypiš KdeMinimum a případně Minimum

```

Na tomto postupu nemáme příliš možností, jak jej urychlit. Přesto si můžeme všimnout alespoň dvou věcí: pokud chceme v programu něco urychlovat, je vhodné upravovat místa, která se vykonávají často. O která místa kódu se jedná, nám dost často může prozradit nástroj zvaný profiler, u jednoduchých programů můžeme ale tato místa najít „od oka“. Když se podíváte na předchozí algoritmus, snadno odhadnete, že nejčastěji se opakuje test vzdálenosti (Pokud Odmocnina ... <=5). Tento test se přitom opakuje pro každou dvojici obcí (m, n) dvakrát: jednou pro obec m v roli i , n v roli j , podruhé pro m v roli j , n v roli i (pokud nechápete, zkuste si projít, jak se bude algoritmus chovat pro dvě obce).

V čem je problém? V tom, že pokud zjistíme, že obec j je od i vzdálena ne více než pět kilometrů, využíváme tuto informaci jen z poloviny: přičítáme počet obyvatel obce j k okolí obce i . Přitom bychom mohli zároveň přičíst počet obyvatel obce i k okolí obce j . Zatímco v prvním algoritmu nám stačila pro okolí jediná proměnná (do té jsme sčítali počet obyvatel v okolí obce i), budeme potřebovat pro každou obec zvláštní proměnnou pro její okolí: už při zpracovávání první obce budeme přidávat počet jejích obyvatel do okolí každé obce, která není dál, než pět kilometrů, stejně při zpracovávání druhé obce, atd.

Tím jsme počet testů vzdálenosti zmenšili na polovinu. Výpočet odmocniny použitý při určení vzdálenosti je ale na většině počítačů dost pomalý. Nešlo by se jej zbavit? Kupodivu ano, a to velice snadno: jestliže potřebujeme otestovat, zda odmocnina z nějaké hodnoty není větší než 5, je to totéž, jako když budeme testovat, zda samotná hodnota není větší než 25. (Tento „trik“ se dá s úspěchem použít i v případě, kdy neporovnáváme s konstantou, v tomto případě 5, ale s proměnnou – násobením je většinou výrazně rychlejší než výpočet odmocniny,

takže porovnání ' $a < b * b$ ' se provede rychleji, než ' $\text{Odmocnina}(a) < b$ ').
Zkusme vytvořit algoritmus 2:

```
[ Datová struktura pro každou obec obsahuje její souřadnice, počet  
jejích obyvatel a počet obyvatel v jejím okolí, tedy x,y,  
počet_obyvatel, okolí ]
```

```
Minimum=nekonečno
```

```
KdeMinimum=??
```

```
Načti obce
```

```
Pro každou obec (proměnná i) nastav i.okolí na 0
```

```
Pro každou obec (proměnná i): {
```

```
  Pro každou obec (proměnná j), kde  $i < j$ : {
```

```
    Rozdíl_x =  $i.x - j.x$ 
```

```
    Rozdíl_y =  $i.y - j.y$ 
```

```
    Pokud  $\text{Rozdíl}_x * \text{Rozdíl}_x + \text{Rozdíl}_y * \text{Rozdíl}_y \leq 25$ , pak {
```

```
       $i.\text{okolí} = i.\text{okolí} + j.\text{počet\_obyvatel}$ 
```

```
       $j.\text{okolí} = j.\text{okolí} + i.\text{počet\_obyvatel}$ 
```

```
    }
```

```
  }
```

```
  Pokud  $i.\text{okolí} < \text{Minimum}$ , pak {
```

```
    Minimum =  $i.\text{okolí}$ 
```

```
    KdeMinimum = i
```

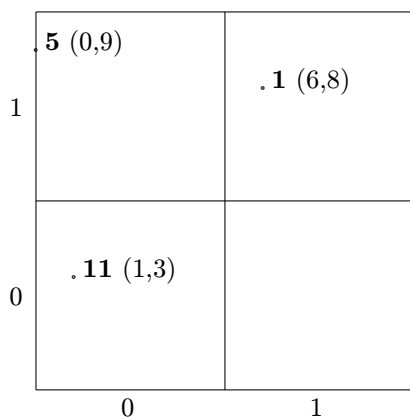
```
  }
```

```
}
```

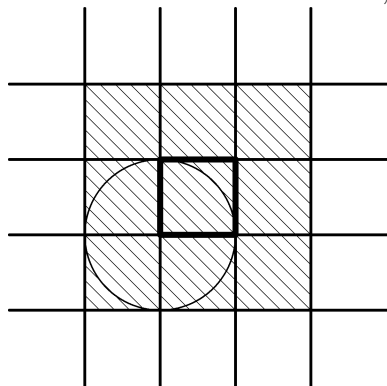
```
Vypiš KdeMinimum a případně Minimum
```

Pro úlohy malého rozsahu jsme dosáhli rozumné hranice efektivity (další zrychlování je možné, ale bylo by pracné a ne příliš výrazné). Zamysleme se ještě krátce nad tím, co bychom mohli udělat v případě, že by se na vstupu objevila data většího rozsahu. Představme si například ČR, která má plochu 78864 km². Když se podíváme na způsob, jakým pracuje předchozí algoritmus, zjistíme, že většinu práce bude na takových datech dělat naprosto zbytečně. Jistě není nutné zjišťovat vzdálenosti mezi obcemi západočeského a severomoravského kraje, neboť tato území jsou od sebe vzdálena rozhodně víc, než pět kilometrů. To znamená, že pokud si zkoumané území rozdělíme na vhodné oblasti, do nichž lze obce snadno umístit (tj. určit, do které z nich každá náleží) a o kterých můžeme rychle rozhodnout, zda jsou od sebe vzdáleny víc než pět kilometrů, ušetříme si při výpočtu dost práce. Pro počítač je ideální rozdělit území čtvercovou sítí, se stranou jednoho čtverce pět kilometrů (To, že je zvolená strana čtverce stejně velká, jako naše „kritická“ vzdálenost, není náhoda. Nebudeme zde ale popisovat, jak jsme k této volbě dospěli.).

Algoritmu přidáme paměť, do níž budeme pro každou oblast (v našem případě čtverce 5 krát 5 km) ukládat seznam obcí, které do ní patří. Tyto seznamy budou na začátku prázdné. Při načítání u každé obce určíme, do kterého čtverce patří (to zjistíme snadno celočíselným vydělením jejich souřadnic pěti, tím dostaneme souřadnice čtverce, do kterého obec náleží). Na následujícím obrázku vidíme grafický význam: obec se souřadnicemi $(1, 3)$ – po vydělení $(1/5, 3/5) = (0, 0)$, tedy náleží do čtverce se souřadnicemi $(0, 0)$. Obdobně pro další dvě $(6/5, 8/5) = (1, 1)$ a $(0/5, 9/5) = (0, 1)$.



Poté, co jsou všechny obce načteny a zařazeny do čtverců, můžeme přistoupit k výpočtu: jak jsme naznačili, naším cílem bude nepočítat vzdálenost mezi čtverci, které jsou od sebe příliš daleko. Které to jsou? Nebo naopak, které to nejsou? Na následujícím obrázku je znázorněn zpracovávaný čtverec (obtažen silně), vyšrafovány jsou okolní čtverce vzdálené ne více, než 5 km.



Pro větší názornost je zakreslen kruh o poloměru 5 km se středem v levé dolní

obci zpracovávaného čtverce. Pro tuto obec leží všichni blízcí sousedé uvnitř tohoto kruhu nebo na hraniční kružnici. Jak je vidět (kdo to nevidí, musí věřit), kruh nezasahuje do žádného nevyšrafovaného čtverce. Pokud si podobně představíte další kruhy se středy v ostatních obcích, které by spadaly do silně vytaženého čtverce, zjistíte, že ani žádný jiný nebude zasahovat do nevyšrafované oblasti. Co z toho plyne? K tomu, abychom pro obec v silně vytaženém čtverci našli všechny, které od ní nejsou vzdáleny více než 5 km, stačí prozkoumat osm okolních čtverců.

Podrobnější rozbor a důkazy, že postup funguje, stejně jako způsob, jak čtverce uložit v paměti a procházet, jsou poměrně obsáhlé. Stačí, že jsme si ukázali, že při volbě postupu je někdy výhodné brát v úvahu i velikost vstupních dat. Jak je vidět v příloze, popsaná metoda je poměrně výkonná. Naznačené myšlenkové postupy se dají aplikovat na celou škálu problémů, kde je zapotřebí zpracovat větší množství dat.

A Porovnání rychlosti

A.1 Strašidla

V řešení úlohy 1 se objevily dva algoritmy. Změřili jsme čas jejich výpočtu nad daty různých velikostí

- Data č. 1 obsahují město o rozměrech 50 krát 50
- Data č. 2 obsahují město o rozměrech 100 krát 100
- Data č. 3 obsahují město o rozměrech 250 krát 250

Algoritmy **a** a **b** jsou v podobě, v jaké jsou popsány v řešení. Naměřené časy jsou v sekundách, u algoritmu **a** jsou navíc časy v hranatých závorkách. Jedná se o časy naměřené s tímž programem, kompilovaným se zapnutou optimalizací na rychlost.

Algoritmus	Čas (s)		
	Data 1	Data 2	Data 3
a	9.90 [5.08]	581.300 [290.74]	??
b	0.01	0.03	0.24

Jak je vidět, algoritmus **b** je výrazně rychlejší než algoritmus **a**, navíc čas vykonávání algoritmu při rostoucí velikosti města roste výrazně rychleji, než čas algoritmu **b**, takže pro velikost města 250 krát 250 se jej nepodařilo změřit (program v rozumném čase neskončil). Tuto časovou náročnost nezachraňuje ani optimalizace, která potřebný čas snižuje zhruba na polovinu.

A.2 Letiště

V řešení úlohy 3 jsme představili několik algoritmů. Pro ilustraci rozdílů mezi jejich rychlostí jsme změřili čas jejich výpočtu nad několika typy dat

- Data č. 1 obsahují mapu zahrnující 2000 obcí na ploše 10000 km²
- Data č. 2 obsahují mapu zahrnující 6258 obcí na ploše 78864 km² (to jsou data udávaná Českým statistickým úřadem pro ČR)
- Data č. 3 obsahují mapu zahrnující 60000 obcí na ploše 100000 km² (tedy skoro desetkrát více obcí, než ČR, ale jen o necelou polovinu větší území)

Aplikované algoritmy jsou čtyři

- Algoritmus 1 v podobě, v jaké je popsán v řešení úlohy 3
- Algoritmus 1.5 je algoritmus č.1 vylepšený o zmenšení počtu testů (tedy algoritmus č.2 s odmocninou v podmínce)
- Algoritmus 2 v podobě, v jaké je popsán v řešení úlohy 3
- Algoritmus 3 je algoritmus využívající rozdělení obcí do čtvercové sítě, jehož myšlenka je stručně nastíněna v závěru řešení úlohy 3

Naměřené časy jsou v sekundách, u algoritmů 1, 1.5 a 2 jsou ve třetím sloupci časy v hranatých závorkách. Jedná se o časy naměřené s týmiž programy, kompilovanými se zapnutou optimalizací na rychlost.

Algoritmus	Čas (s)		
	Data 1	Data 2	Data 3
1	0.88	8.49	972.04 [450.68]
1.5	0.44	4.19	501.83 [247.34]
2	0.10	0.85	237.32 [70.70]
3	0.00	0.01	0.27

Tabulka ukazuje několik důležitých věcí. V první řadě je zřetelně vidět, jak zdánlivě malá úprava kódu může vést k velkému rozdílu v dosažené rychlosti. Čas dosažený algoritmem 2 je oproti času běhu algoritmu 1 většinou méně než čtvrtinový. Dále je vidět, že optimalizace kódu dnešních kompilátorů je na docela slušné úrovni (čas výpočtu srazila vždy na méně než polovinu), ale nikdy nemůže vyrovnat kvalitu návrhu. Algoritmus 3 pracuje nad největšími daty více než 250 krát rychleji, než optimalizovaná verze algoritmu 2. Tento poměr by s rostoucí velikostí dat dále rostl. Na druhou stranu potřebuje algoritmus 3 ke svému výpočtu o něco více paměti, její spotřeba roste úměrně ploše území.