```
    color: $color;
  }
}
```

5. Závěr

Stručně jsme představili preprocesor SASS, který významně rozšiřuje syntaxi CSS. Samotný preprocesor je velmi jednoduchý a není složité jej pochopit. Jeho začlenění do výuku umožní studentům efektivnější psaní CSS kódu. Pro úplnost ještě zmíníme zajímavou alternativu CSS preprocesorů a to stylované komponenty.[20)] Zjednodušeně řečeno stylované komponenty umožňují generovat CSS kód pomocí JavaScriptu. JavaScript nahrazuje omezený kód preprocesoru, čímž získáme výrazně větší možnosti generování CSS. Dodejme, že se ale nejedná o technologii, která je stejně rozšířená jako CSS preprocesory.

Silniční síť (Úlohy z MO kategorie P, 49. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

V 35. ročníku Matematické olympiády byla do soutěže zařazena nová kategorie P (programování) se soutěžními úlohami zaměřenými na algoritmizaci a programování. Stalo se tak ve školním roce 1985/86. V té době se u nás začaly objevovat první mikropočítače, ale nebyly ještě rozšířeny natolik, aby je měl k dispozici každý student a na každé škole. V počátečních ročnících soutěže nebyly proto v kategorii P zadávány žádné praktické úlohy. Soutěžící odevzdávali řešení všech soutěžních úloh pouze v písemné formě, svoje programy psali na papír bez odladění na počítači.

V porovnání s úlohami zadávanými v MO-P v současné době, byly tehdejší soutěžní úlohy o něco snazší. To odpovídalo situaci, kdy se výuka informatiky a programování na střední školy teprve postupně zaváděla a její úroveň byla oproti dnešku podstatně nižší. Přesto ale i mezi úlo-

[20)] styled-components.com

hami z prvních ročníků MO kategorie P najdeme zajímavé problémy, na jejichž řešení si můžeme ukázat mnohé užitečné principy, dodnes používané při návrhu efektivních algoritmů. S jednou takovou úlohou se nyní seznámíme. Pochází z krajského kola 37. ročníku MO (školní rok 1987/88), tedy v pořadí z třetího ročníku kategorie P. Na jejím řešení si ukážeme různé možnosti použití rekurze a metody odstraňování neefektivity při nevhodně zvoleném rekurzivním návrhu.

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

Mezi n městy očíslovanými postupně od 1 do n je vybudována jednosměrná silniční síť podle těchto pravidel:

a) z každého města vedou nejvýše dvě silnice,

b) vedou-li z města dvě silnice, pak směřují do různých měst,

c) silnice vedoucí z města i směřuje vždy do města s číslem větším než i.

Uvedená silniční síť je zadána pomocí dvou celočíselných polí a, b indexovaných od 1 do n takových, že $n > 1$ a pro všechna i, $1 \leq i \leq n$ platí následující tři podmínky:

1. $i < a[i] \leq n$ nebo $a[i] = 0$
2. $i < b[i] \leq n$ nebo $b[i] = 0$
3. $a[i] \neq b[i]$ nebo $a[i] = b[i] = 0$

Pro všechna i, $1 \leq i \leq n$, vede přímá silnice z města i do města $a[i]$, jestliže $a[i] > 0$, a z města i do města $b[i]$, jestliže $b[i] > 0$.

Sestrojte algoritmus, který určí, kolika různými způsoby se lze dostat z města 1 do města n.

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

Uvedené zadání přesně odpovídá původnímu znění soutěžní úlohy. Když si popsanou silniční síť vyjádříme v současné terminologii teorie grafů, jedná se o acyklický orientovaný graf s n vrcholy, v němž z každého vrcholu vedou nejvýše dvě hrany. Navíc máme přímo zadáno topologické uspořádání vrcholů tohoto grafu: vrcholy jsou očíslovány tak, že každá hrana grafu vede z vrcholu s nižším číslem do vrcholu s vyšším číslem. To znamená, že pokud bychom si nakreslili vrcholy s čísly od 1 do n v tomto pořadí zleva doprava, všechny hrany grafu by směřovaly zleva doprava. Naším úkolem je určit počet různých cest, které vedou z vrcholu 1 do vrcholu n.

Než začneme úlohu řešit, uvědomme si, že počet různých cest, vedoucích z města 1 do města n v silniční síti s n městy, může být velmi vysoký, dokonce až exponenciální vzhledem k hodnotě n. Představte si třeba situaci

pro $n = 100$ takovou, že z města 1 vedou silnice do měst 2 a 3, z každého z nich vedou silnice do měst 4 a 5, z každého z nich vedou silnice do měst 6 a 7, a tak dále, až se dostaneme ke dvojic měst 98 a 99, z nichž vede už jen jedna silnice do cílového města 100. Nakreslete si sami obrázek, ze kterého jasně vyplývá, že cestou po silnicích z města 1 do města 100 se budeme celkem 49krát rozhodovat vždy mezi dvěma možnostmi, kam jít dále. Celkový počet různých cest v tomto případě proto dosahuje obrovské hodnoty 2^{49}.

Ukážeme si nyní několik různých způsobů řešení, které se budou lišit asymptotickou časovou složitostí algoritmu, a tedy také rychlostí výpočtu výsledného programu. Přirozeným způsobem se nabízí použít k řešení rekurzivní funkci s parametrem x, která bude určovat počet různých cest vedoucích z města 1 do města x. Pro získání požadovaného výsledku pak bude postačovat, když tuto funkci zavoláme s parametrem n. Funkce může vypadat například takto:

```
def cesty1(x):
    """ počet cest z města 1 do města x """
    if x == 1:
        return 1
    pocet = 0
    for i in range(1, x):
        if a[i] == x or b[i] == x:
            pocet += cesty1(i)
    return počet
```

Uvedená funkce *cesty1* přesně kopíruje zadání úlohy. Pro $x = 1$ funkce vrací výsledek 1, neboť z města 1 do města 1 se jistě dostaneme jedním způsobem (již tam jsme). Pro $x > 1$ najdeme všechny silnice vedoucí do města x. Podle zadání úlohy tyto silnice mohou vést jedině z měst s nižším číslem. Příslušná města vyhledáme v polích a, b a sečteme počty cest vedoucích do těchto měst z výchozího města 1. Tyto počty získáme rekurzivním voláním funkce *cesty1*.

Popsané řešení je sice naprosto správné, ale velmi neefektivní. Jeho neefektivita spočívá v tom, že rekurzivní funkce *cesty1* může být během jednoho výpočtu opakovaně mnohokrát volána se stejnou hodnotou parametru. Každé takové zavolání funkce představuje značné množství práce a velké časové nároky způsobené zejména dalšími rekurzivními voláními, přitom ale vede vždy k témuž výsledku. Mnoho stejné práce se stejným výsledkem se tedy vykonává zbytečně opakovaně. Výslednou hodnotu zís-

káme tak, že všechny možné cesty rekurzivně postupně projdeme vždy ve směru od města n k městu 1. Již jsme si ukázali, že těchto cest může být až exponenciálně mnoho vzhledem k n, takže výše uvedená funkce *cesty1* má v nejhorším případě exponenciální asymptotickou časovou složitost. Pro vyšší hodnoty n je tudíž prakticky nepoužitelná.

K řešení úlohy můžeme přistoupit také z druhé strany a navrhnout rekurzivní funkci *cesty2* s parametrem x, která bude určovat počet různých cest vedoucích z města x do cílového města n. Výsledek úlohy pak získáme zavoláním této funkce s parametrem 1. Funkce bude podobná jako v předchozím případě. Pro parametr $x = n$ funkce vrátí výsledek 1, jelikož z města n do města n se dostaneme jediným způsobem. Pokud $x < n$, snadno určíme města, do nichž vede přímá silnice z města x – jsou to města s čísly $a[x]$, $b[x]$ (pokud jsou tyto hodnoty nenulové). Výsledkem pak je součet počtů cest vedoucích z měst $a[x]$, $b[x]$ do města n. Tyto počty získáme rekurzivním voláním funkce *cesty2*.

```
def cesty2(x):
    """ počet cest z města x do města n """
    if x == 0:
        return 0
    if x == n:
        return 1
    return cesty2(a[x]) + cesty2(b[x])
```

Funkce je o něco jednodušší než v předchozím případě, neobsahuje ve svém těle žádný cyklus. Přesto má ale stále exponenciální časovou složitost vzhledem k n. Zdůvodnění je stejné jako v první verzi řešení.

Pokud se při řešení úlohy setkáme s neefektivní rekurzivní funkcí, která je opakovaně volána se stejnou hodnotou parametru a počítá tak zbytečně vícekrát tutéž hodnotu, máme zpravidla dvě možnosti, jak tuto neefektivitu odstranit. Oba postupy si nyní ukážeme. První možností je doplnit rekurzivní funkci globálním polem, do kterého si budeme ukládat všechny hodnoty naší funkce, které již známe. Pole bude indexováno čísly jednotlivých měst (neboli hodnotou parametru při zavolání funkce). Před každým zamýšleným rekurzivním voláním v tomto poli zkontrolujeme, zda jsme potřebnou funkční hodnotu již počítali někdy dříve. Jestliže ano, použijeme hodnotu uloženou v poli a nebudeme ji opakovaně počítat rekurzí. Jestliže ne, provedeme normálně rekurzivní volání a jeho výsledek nejen použijeme k dalšímu výpočtu, ale také ho uložíme do našeho pole pro případné další využití.

Popsaný postup si nyní předvedeme naprogramovaný. Je to obecný princip, který můžeme uplatnit jak na první, tak i na druhou variantu řešení, které již máme. V prvním případě budeme do pole c na indexy měst 1 až n ukládat počty cest vedoucích z 1 do příslušného města, tzn. prvek $c[x]$ bude mít stejnou hodnotu, jakou vrací volání funkce *cesty1* s parametrem x. Seznamy v Pythonu jsou indexovány od 0, takže prvek $c[0]$ nebudeme využívat. Na ostatní indexy seznamu c vložíme na začátku výpočtu -1 jako příznak, že tuto hodnotu dosud neznáme. Počáteční hodnota $c[1] = 1$ nám umožňuje zkrátit výsledný kód funkce.

```
c = [0, 1] + [-1] * (n-1)

def cesty3(x):
    """ počet cest z města 1 do města x """
    pocet = 0
    for i in range(1, x):
        if a[i] == x or b[i] == x:
            if c[i] == -1:
                c[i] = cesty3(i)
            pocet += c[i]
    return pocet
```

V případě druhého řešení postupujeme analogicky, hodnota $c[x]$ uložená v poli c bude nyní udávat počet různých cest vedoucích z města x do města n. Pole c si opět inicializujeme samými -1 jako označení, že žádné hodnoty zatím neznáme, pouze hodnoty $c[0] = 0$ a $c[n] = 1$ nám podobně jako v předchozím řešení zkrátí výsledný kód.

```
c = [0] + [-1] * (n-1) + [1]

def cesty4(x):
    """ počet cest z města x do města n """
    if c[a[x]] == -1: c[a[x]] = cesty4(a[x])
    if c[b[x]] == -1: c[b[x]] = cesty4(b[x])
    c[x] = c[a[x]] + c[b[x]]
    return c[x]
```

Použití pomocného pole na uložení již spočítaných hodnot nic nezmění na správnosti algoritmu, výpočet probíhá stále naprosto stejným způsobem jako dříve. Velmi zásadně se ale sníží počet provedených rekurzivních volání funkce *cesty3* nebo *cesty4*. Funkce bude nyní zavolána s každou hodnotou parametru nejvýše jednou, všechna další rekurzivní volání funkce

budou nahrazena vyzvednutím potřebné hodnoty z pole c. Celkový počet všech provedených rekurzivních volání funkce bude proto nejvýše n. Každé provedení těla funkce *cesty3* má lineární časovou složitost vzhledem k n, pokud již nepočítáme složitost rekurzivních volání, neboť tělo funkce obsahuje jeden cyklus délky nejvýše n. Celková časová složitost funkce *cesty3* je proto v nejhorším případě kvadratická. Naproti tomu každé provedení těla funkce *cesty4* proběhne v konstantním čase (opět bez započítání složitosti rekurzivních volání), takže toto řešení má celkově dokonce lineární časovou složitost vzhledem k n.

Na závěr ukážeme ještě jinou možnost, jak lze vyřešit problém s neefektivitou rekurzivního řešení. Místo doplnění rekurzivní funkce globálním polem na uložení již známých funkčních hodnot tentokrát rekurzi zcela opustíme a nahradíme ji cyklem. V něm budeme ve vhodném pořadí počítat příslušné hodnoty a ukládat je do pole c. Význam prvků tohoto pole přitom zůstane naprosto shodný jako v předchozím řešení. Rozdíl spočívá pouze v tom, že místo rekurzivního výpočtu shora budeme hodnoty počítat iteračně zdola. Důležité je zvolit takové správné pořadí výpočtu, abychom se v každém okamžiku odkazovali pouze na ty hodnoty, které již máme spočítané a uložené v poli. Zde nám pomůže skutečnost, že zkoumaná silniční síť představuje orientovaný graf bez cyklů, který je topologicky uspořádaný (tj. silnice v něm vedou pouze do měst s vyšším číslem).

Uvedenou náhradu rekurzivní funkce cyklem lze opět aplikovat jak na předchozí řešení s rekurzivní funkcí *cesty3*, tak i na druhou variantu řešení s funkcí *cesty4*. Viděli jsme, že funkce *cesty4* je o něco šikovnější na zápis i efektivnější časově, proto tentokrát zůstaneme pouze u tohoto postupu. Hodnoty $c[x]$ budou tedy udávat počet cest počet různých cest vedoucích z města x do města n. Prvky pole c budeme počítat postupně zprava doleva, tzn. od indexu n dolů k indexu 1. Tím budeme mít zajištěno, že při výpočtu $c[x]$ budeme již znát potřebné hodnoty $c[a[x]]$, $c[b[x]]$.

```
def cesty5():
    """ počet cest z města 1 do města n """
    c = [0]*n + [1]
    for x in range(n-1, 0, -1):
        c[x] = c[a[x]] + c[b[x]]
    return c[1]
```

Výsledné řešení je velmi krátké a elegantní. Má zjevně lineární časovou složitost vzhledem k počtu měst n, neboť funkce *cesty5* osahuje jediný cyklus délky n.