# INFORMATIKA

# Zjištění počtu možností

*PAVEL TÖPFER*

Matematicko-fyzikální fakulta UK, Praha

V článku si ukážeme, jak lze navrhnout efektivní algoritmus pro řešení úloh typu *Kolika různými způsoby lze něco udělat?* Postup řešení si předvedeme na třech zdánlivě zcela rozdílných úlohách: kolika různými způsoby můžeme dojet z jednoho bodu silniční sítě do druhého, kolika různými způsoby lze vydláždit pěšinu pomocí dlaždic zadaných rozměrů, kolik existuje různých binárních stromů dané velikosti. Jak uvidíme, základní princip řešení všech těchto úloh bude shodný – použijeme metodu dynamického programování.

Dynamické programování je programovací technika používaná tehdy, když zadanou úlohu pro vstupní data velikosti $N$ sice neumíme bezprostředně vyřešit, ale její řešení dokážeme poměrně snadno sestavit z výsledků úloh stejného charakteru, ale pracujících s menšími vstupními daty. Pro velmi malá vstupní data (např. pro $N = 1$) bývá řešení úlohy triviální. Vyřešíme tedy nejprve tuto triviální úlohu pro $N = 1$, pomocí jejího řešení pak o něco větší úlohu pro $N = 2$, s využitím výsledků těchto dvou úloh dále vyřešíme úlohu pro $N = 3$, atd. Takto stále postupujeme, dokud nedojdeme až k vyřešení úlohy původně požadované velikosti. Na výsledky všech těch menších řešených úloh se nás nikdo neptal a my je ani nedáváme na výstup, ale ukládáme si je a využíváme je při výpočtu řešení větších úloh.

Detailní práce s uloženými hodnotami se liší u různých konkrétních úloh, ale princip zůstává vždy stejný. V některé úloze použijeme každou zapamatovanou hodnotu pouze jednou, často ale budeme využívat tutéž uloženou hodnotu postupně vícekrát. Někdy se dokonce stane, že některé zaznamenané hodnoty nevyužijeme vůbec, ale to při jejich postupném vý-

počtu dopředu nevíme a pro jistotu si je ukládáme všechny. Někdy je zapotřebí pamatovat si v tabulce po celou dobu výpočtu výsledky všech menších úloh, někdy naopak odvodíme řešení pro data velikosti $N$ třeba jen na základě výsledku získaného pro $N - 1$ a stačí tedy pamatovat si v každém kroku výpočtu jenom jednu předchozí hodnotu. Všechny tyto varianty si postupně předvedeme.

Dynamické programování se používá zejména pro řešení dvou základních typů algoritmických úloh:

1. pro kombinatorické úlohy odpovídající na otázku, kolika různými způsoby lze něco udělat,
2. pro optimalizační úlohy odpovídající na otázku, jak nejlépe lze něco udělat.

V tomto článku se budeme věnovat pouze prvnímu případu a ukážeme si několik příkladů takových úloh. K úlohám druhého typu se vrátíme někdy jindy.

Začneme jednoduchým příkladem z teorie grafů. Máme zadán acyklický orientovaný graf s $N$ vrcholy. Slovo acyklický znamená, že graf neobsahuje žádný cyklus, neboli neobsahuje žádnou orientovanou cestu, po níž bychom přišli zpět do výchozího vrcholu. Každý orientovaný graf bez cyklů je možné topologicky uspořádat, tzn. označit jeho vrcholy různými celými čísly od 1 do $N$ takovým způsobem, že každá hrana grafu povede z vrcholu s menším číslem do vrcholu s větším číslem. Jinými slovy, pokud v takto očíslovaném grafu existuje orientovaná hrana z vrcholu $i$ do vrcholu $j$, potom $i < j$. Předpokládejme tedy, že zkoumaný graf je již topologicky uspořádán. Máme za úkol určit, kolik existuje různých cest z vrcholu 1 do vrcholu $N$.

Zadaný graf si můžeme představit jako silniční síť s jednosměrnými silnicemi mezi $N$ městy, která jsou očíslována od 1 do $N$ podle výše uvedeného topologického uspořádání. Chceme zjistit, kolika různými způsoby lze po silnicích dojet z města 1 do města $N$.

Úlohu je možné řešit prohledáváním zadaného grafu do hloubky. Prohledávat začneme ve startovním vrcholu číslo 1. Díky acykličnosti grafu se vůbec nemusíme zabývat evidencí již navštívených vrcholů, výpočet se nemůže zacyklit. Stačí tedy jenom počítat, kolikrát během prohledávání celého grafu přijdeme do cílového vrcholu $N$. Tím získáme požadovaný výsledek. Implementace takového řešení pomocí rekurzivní funkce je snadná, ale podstatnou nevýhodou tohoto postupu je jeho neefektivita. Počet různých cest z vrcholu 1 do vrcholu $N$ může být značný, v některých grafech

až exponenciální vzhledem k počtu vrcholů $N$. Algoritmus bude každou z těchto cest procházet, takže jeho asymptotická časová složitost bude v nejhorším případě exponenciální vzhledem k $N$.

V následující ukázkové implementaci předpokládáme, že zadaný acyklický orientovaný graf je již topologicky uspořádán a je reprezentován maticí sousednosti $g$. Hodnota $g[x][y]$ je typu boolean a určuje, zda existuje orientovaná hrana z vrcholu $x$ do vrcholu $y$.

```
def pocet_cest(g, N):
    pocitadlo = 0

    def pruchod(v):
        if v == N:
            nonlocal pocitadlo
            pocitadlo += 1
        else:
            for k in range(v+1, N+1):  # hrana do vrcholu k
                if g[v][k]: pruchod(k)

    pruchod(1)
    return pocitadlo
```

Nyní si ukážeme podstatně lepší řešení téže úlohy pomocí dynamického programování. Budeme si postupně počítat, kolik existuje různých cest ze startovního vrcholu 1 do vrcholu $i$ pro všechna $i = 1, 2, 3, \ldots, N$. Tyto hodnoty $t[i]$ si budeme ukládat do jednoduché tabulky tvořené jednorozměrným polem velikosti $N$. Zjevně platí $t[1] = 1$, hledaným výsledkem pak bude poslední hodnota $t[N]$. Zbývá ukázat, jak spočítáme $t[i]$ pro $i > 1$, když už známe předchozí hodnoty $t[1]$, $t[2]$, $\ldots$, $t[i-1]$. Každá cesta z vrcholu 1 do vrcholu $i$ musí končit hranou vedoucí z nějakého vrcholu $k$ do vrcholu $i$. Vzhledem k topologickému očíslování vrcholů grafu víme, že $k < i$ a hodnotu $t[k]$ proto již známe. Hodnotu $t[i]$ tedy získáme jako součet všech $t[k]$ takových, že v grafu existuje hrana z vrcholu k do vrcholu $i$.

Do každého vrcholu vede méně než $N$ hran, takže každé $t[i]$ spočítáme v čase $\mathrm{O}(N)$. Postupně počítáme $N$ hodnot $t[1]$, $t[2]$, $\ldots$, $t[N]$, a proto celková asymptotická časová složitost uvedeného algoritmu je $\mathrm{O}(N^2)$. Prostorová složitost je $\mathrm{O}(N)$, neboť si musíme v poli $t$ pamatovat všech $N$ postupně počítaných hodnot.

Také v této programové ukázce předpokládáme, že zadaný acyklický orientovaný graf je již topologicky uspořádán a je reprezentován stejně

jako minule maticí sousednosti $g$. Hodnota $g[x][y]$ určuje, zda existuje orientovaná hrana z vrcholu $x$ do vrcholu $y$.

```
def pocet_cest(g, N):
    t = [0] * (N+1)
    t[1] = 1
    for i in range(2, N+1):     # cesty do vrcholu i
        for k in range(1, i):   # hrana z vrcholu k
            if g[k][i]: t[i] += t[k]
    return t[N]
```

Ve druhé úloze budeme dláždit pěšinu šířky 2 a délky $N$. Použijeme k tomu $N$ obdélníkových dlaždic o velikosti $1 \times 2$ (délková jednotka zde není podstatná). Celá pěšina musí být pokryta dlaždicemi, žádné dvě dlaždice se nesmí překrývat. Naším úkolem je určit, kolik různých způsobů vydláždění existuje.

Řešení úlohy opět založíme na metodě dynamického programování. Budeme postupně počítat, kolika různými způsoby lze vydláždit pěšinu délky $i$ pro různé délky $i = 1, 2, 3, \ldots, N$. Takto získané hodnoty $t[i]$ si budeme ukládat do pole. Snadno určíme první dvě hodnoty $t[1] = 1$, $t[2] = 2$. Ukážeme, jak spočítat hodnotu $t[i]$ pro $i > 2$ na základě již známých předchozích hodnot. Na úplném konci vydlážděné pěšiny délky $i$ mohou nastat dvě případy. Může tam být

- buď jedna dlaždice napříč a před ní vydlážděná pěšina délky $i - 1$, což znamená $t[i - 1]$ možností,
- nebo dvě dlaždice položené podélně vedle sebe ve směru pěšiny a před nimi je vydlážděná pěšina délky $i - 2$, což představuje dalších $t[i - 2]$ možností.

Žádná jiná možnost vydláždění neexistuje, takže $t[i] = t[i - 1] + t[i - 2]$ pro $i > 2$. Požadovaný výsledek pro pěšinu délky $N$ získáme jako poslední hodnotu v poli, tedy číslo $t[N]$.

Každou z $N$ hodnot $t[1]$, $t[2]$, …, $t[N]$ spočítáme tentokrát v konstantním čase, takže celková asymptotická časová složitost popsaného řešení je pouze $O(N)$. Podle výše uvedeného algoritmu počítáme a ukládáme postupně $N$ hodnot $t[i]$, což vede k řešení s prostorovou složitostí $O(N)$. Můžeme ho snadno zapsat ve tvaru funkce v Pythonu:

```
def pocet_dlazdeni(N):
    t = [0] * (N+1)
    t[1], t[2] = 1, 2
```

```
    for i in range(3, N+1):  # počet dláždění délky i
        t[i] = t[i-1] + t[i-2]
    return t[N]
```

Všimněte si, že na rozdíl od předchozí úlohy spočítáme každé $t[i]$ pouze pomocí dvou bezprostředně předcházejících hodnot $t[i-1]$, $t[i-2]$. Ve skutečnosti tedy vůbec nepotřebujeme pole $t$ velikosti $N$. Místo něj vystačíme se dvěma proměnnými na uložení dvou dosud posledních hodnot, starší hodnoty si nemusíme pamatovat. Při dobré implementaci má tak celé řešení dokonce konstantní prostorovou složitost, jak ukazuje i následující funkce:

```
def pocet_dlazdeni(N):
    a, b = 1, 2
    for i in range(3, N+1):  # počet dláždění délky i
        a, b = b, a + b
    return b
```

Poznamenejme ještě, že takto získaná posloupnost hodnot

$$t[i] = 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

je v matematice velmi známou posloupností Fibonacciho čísel.

Ve třetím příkladu budeme zkoumat, kolik existuje různých binárních stromů tvořených přesně $N$ vrcholy. Binárním stromem rozumíme orientovaný strom s jedním význačným vrcholem (kořenem), ze kterého vede cesta do všech ostatních vrcholů. Každý vrchol může mít nejvýše dva potomky. Každý vrchol kromě kořene má právě jednoho předka, kořen předka nemá.

Také tuto úlohu budeme řešit dynamickým programováním. Postupně určíme počet binárních stromů s 0 vrcholy, s 1 vrcholem, se 2 vrcholy, … atd. Získané hodnoty si budeme ukládat do pole $t$, takže $t[i]$ bude určovat počet různých binárních stromů tvořených $i$ vrcholy. Hledaným výsledkem bude údaj $t[N]$. Nejprve opět stanovíme počáteční hodnoty, zjevně $t[0] = 1$, $t[1] = 1$. Nyní potřebujeme odvodit vzorec pro výpočet $t[i]$ pro $i > 1$, když už známe hodnoty $t[0]$, $t[1]$, …, $t[i-1]$.

Binární strom tvořený $i$ vrcholy má jeden vrchol v kořeni a o zbývajících $i-1$ vrcholů se musí podělit levý a pravý podstrom. Je-li v levém podstromu $k$ vrcholů, pak v pravém jich bude $i-k-1$. Každá volba $k = 0, 1, 2, \ldots, i-1$ vede k jinému celkovému tvaru binárního stromu. Pro

zvolené k má levý podstrom $t[k]$ různých možných tvarů, zatímco pravý podstrom má $t[i-k-1]$ různých možných tvarů. Přitom každá kombinace levého a pravého podstromu představuje jiný celkový tvar stromu, což pro pevně zvolené k dává $t[k] \cdot t[i-k-1]$ možností. Počet všech různých binárních stromů s $i$ vrcholy pro $i > 1$ tedy můžeme vyjádřit takto:

$$t[i] = t[0] \cdot t[i-1] + t[1] \cdot t[i-2] + t[2] \cdot t[i-3] + \cdots + t[i-1] \cdot t[0]$$

Z výše uvedeného vzorce pro výpočet $t[i]$ je zřejmé, že každou hodnotu $t[i]$ spočítáme v čase $\mathrm{O}(N)$. Postupně určujeme hodnoty $t[0]$, $t[1]$, $t[2]$, …, $t[N]$, takže asymptotická časová složitost celého výpočtu je $\mathrm{O}(N^2)$. Při výpočtu $t[i]$ využíváme všechny předcházející hodnoty, které proto musíme mít uložené v datové struktuře velikosti $\mathrm{O}(N)$. Prostorová složitost popsaného algoritmu je tudíž $\mathrm{O}(N$.

Opět si ukážeme, jak může vypadat jednoduchá implementace popsaného řešení pomocí funkce v Pythonu:

```
def pocet_binstromu(N):
    t = [0] * (N+1)
    t[0], t[1] = 1, 1
    for i in range(2, N+1):  # binární stromy s i vrcholy
        for k in range(i):  # počet vrcholů v levém podstromu
            t[i] += t[k] * t[i-k-1]
    return t[N]
```

Na závěr pro zajímavost doplníme poznámku, že takto získaná posloupnost čísel

$$1, 2, 5, 14, 42, 132, 429, 1\,430, 4\,862, 16\,796, 16\,796, \ldots$$

je v matematice známa jako Catalanova čísla. Nachází uplatnění při řešení řady různých kombinatorických problémů. $N$-té Catalanovo číslo lze spočítat podle explicitního vzorce bez využití dynamického programování, to ale není námětem našeho článku a případní zájemci si mohou informace sami dohledat.