2. Rozděl a panuj

(zapsali J. Záloha a P. Bašista)

Na minulé přednášce jsme si zavedli výpočetní model a časovou složitost. Nyní se pustíme do analýzy složitosti různých algoritmů, zejména rekurzivních založených na metodě Rozděl a panuj *(Divide et Impera)*.

Asymptotická notace

Jak už víme, při porovnávání algoritmů je obvykle rozhodující asymptotické chování a multiplikativní konstanty můžeme zanedbávat (beztak závisí na konkrétním počítači). Proto si zavedeme takzvanou asymptotickou notaci, která slouží pro právě takové porovnávání růstu funkcí:

Definice: Pro funkce $f, g : \mathbb{N} \to \mathbb{R}^+$ řekneme, že f je $\mathcal{O}(g)$ právě tehdy, když $\exists c > 0 : \forall^* n \in \mathbb{N} : f(n) \leq c \cdot g(n)$. Zde $\forall^* n \in \mathbb{N}$ je zkratka za „$\exists n_0 \in \mathbb{N} : \forall n \geq n_0$“, tedy „pro všechna n až na konečně mnoho výjimek.“

Poznámka: $\mathcal{O}$-notace tedy vyjadřuje, že funkce f je skoro všude menší nebo nejvýše rovná nějakému reálnému násobku funkce g. Ačkoliv zápis vypadá jako rovnost, rozhodně není symetrický: například platí $\log n = \mathcal{O}(n)$, ale neplatí $n = \mathcal{O}(\log n)$. Formálně by bylo lepší považovat $\mathcal{O}(g)$ za třídu funkcí, pro které platí, že se dají shora odhadnout kladným násobkem funkce g, a psát tedy $f \in \mathcal{O}(g)$, ale zvyk je bohužel železná košile.

Příklady: $2.5n^2 = \mathcal{O}(n^2)$, $2.5n^2 + 30n = \mathcal{O}(n^2)$.

Také platí:

$$\mathcal{O}(f) + \mathcal{O}(g) \in \mathcal{O}(f + g),$$

čímž myslíme, že pokud vezmeme libovolnou $f' = O(f)$ a $g' = O(g)$, bude $f' + g' = O(f + g)$. To platí, jelikož skoro všude je $f' \leq cf$, $g' \leq dg$, a tedy $f' + g' \leq cf + dg \leq (c + d)(f + g)$.

Cvičení: Ukažte, že:

- $\mathcal{O}(f) \cdot \mathcal{O}(g) = \mathcal{O}(f \cdot g)$,
- $\mathcal{O}(f + g) = \mathcal{O}(\max(f, g))$,
- $\mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2 + n) = \mathcal{O}(n^2)$.

$\mathcal{O}$-notace popisuje horní odhad asymptotického chování funkce. Mnohdy však potřebujeme také odhandout funkci zespodu (chceme-li říci, že algoritmus potřebuje *alespoň* nějaké množství času nebo paměti), případně z obou stran:

Definice:

- $f = \Omega(g) \equiv \exists c > 0 : \forall^* n \in \mathbb{N} : f(n) \geq c \cdot g(n)$.
  Ω-notace tedy říká, že hodnota funkce f je vždy stejná nebo vyšší než nějaký c-násobek funkce g, a tedy $g = \mathcal{O}(f)$.
- $f = \Theta(g) \equiv f = O(g) \wedge f = \Omega(g)$
  nebo výřečněji:
  $f = \Theta(g) \equiv \exists\ c_1, c_2 > 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ To znamená, že existují nezáporné reálne konstanty c_1, c_2 takové, že se funkce $f(n)$ dá ohraničit c_1- a c_2-násobkem funkce $g(n)$.

Porovnání růstu funkcí: (aneb jak moc máme algoritmy rádi podle jejich chování od nejlepších k nejhorším)

- $\Theta(1)\ldots$ funkce zespoda i shora ohraničené konstantami
- $\Theta(\log(\log n))$
- $\Theta(\log n)$ ... logaritmická
- $\Theta(n^\varepsilon), \varepsilon \in (0,1)$ ... sublineární
- $\Theta(n)$ ... lineární
- $\Theta(n^2)$ ... kvadratická
- $\Theta(n^k)$ ... polynomiální
- $\Theta(2^n)$ ... exponenciální při základu 2
- $\Theta(3^n)$ ... exponenciální při základu 3
- $\Theta(k^n)$ ... exponenciální při základu $k > 1$
- $\Theta(n!)$ ... faktoriálová
- $\Theta(n^n)$
- ... nekonečně mnoho dalších tříd (i mezi těmi výše uvedenými)

Poznámka: Pokud se v odhadu složitosti vyskytne logaritmus (jinde než v exponentu), nezáleží na tom, jaký má základ, protože platí:

$$\log_k n = \frac{\log_c n}{\log_c k} = \frac{1}{\log_c k} \cdot \log_c n,$$

kde $1/\log_c k$ je jen konstanta, takže ji můžeme „schovat do $\mathcal{O}$."

Příklad: Eukleidův algoritmus: Pokud jej pustíme na 2 čísla o n bitech, počet iterací bude $\mathcal{O}(n)$, každá iterace trvá $\mathcal{O}(n^2)$ kroků. Takže celkově má tento algoritmus časovou složitost $\mathcal{O}(n^3)$.

Metoda Rozděl a panuj

Nyní přestaňme chodit kolem horké kaše a řekněme si, co to ono výše zmiňované „rozděl a panuj" znamená. Mějme nějaký problém, který má tu vlastnost, že když jej rozdělíme na nějaké podproblémy, které mají stejný charakter, a ty vyřešíme, složením jejich řešení můžeme získat řešení původního problému. Algoritmus tedy bude rekurzivně volat sám sebe, než se dostane k podproblému nějaké konstantní velikosti, který už umí vyřešit triviálně, a pak se začne z rekurze vracet a skládat jednotlivá dílčí řešení.

Příklad 1 – MergeSort: Algoritmus *MergeSort* pro třídění posloupnosti vypadá takto: vstup rozdělíme na dvě (skoro) stejné části, ty rekurzivním voláním setřídíme, a nakonec výsledné dvě posloupnosti slijeme do jedné. Rozdělování a slévání nám trvá lineárně dlouho, takže pro časovou složitost MergeSortu platí tato rekurentní rovnice:

$$T(n) = 2 \cdot T(n/2) + \mathcal{O}(n).$$

Jak ji vyřešíme? Třeba tak, že si představíme strom rekurzivních volání. Každý vrchol má dva syny (dělíme vstup na dvě části), v nichž jsou vstupy poloviční velikosti.

V každém vrcholu trávíme čas lineární s velikostí jeho vstupu, součet velikostí vstupů přes každou hladinu je n a hloubka stromu musí být $\mathcal{O}(\log n)$. Vyjde nám tedy, že $T(n) = \mathcal{O}(n \log n)$.

Příklad 2 – násobení čísel: Pokud násobíme dvě čísla X a Y způsobem, který nás učili na základní škole, dostaneme se na časovou složitost $\Theta(n^2)$. Protože se jedná o dost častou operaci, zamysleme se, jestli by nešla zrychlit. Nasměrujme naše úvahy na postup „rozděl a panuj". Rozdělíme každého činitele na dvě stejně dlouhé části a pro jednoduchost předpokládejme, že toto rozštěpení činitele proběhne vždy bez zbytku:

$$X = A \cdot 10^{n/2} + B, \qquad Y = C \cdot 10^{n/2} + D.$$

Zde A, B, C, D jsou už jen $n/2$-ciferná čísla. Původní součin získáme jako:

$$XY = (A \cdot 10^{n/2} + B)(C \cdot 10^{n/2} + D) = AC \cdot 10^n + (AD + BC) \cdot 10^{n/2} + BD.$$

Nyní, jak vidíme, stačí spočítat součin čtyř $n/2$-ciferných čísel. Uvažme, jakou bude mít tento algoritmus časovou složitost:

$$T(n) = 4T(n/2) + \Theta(n).$$

Jak takovou rekurenci vyřešíme?

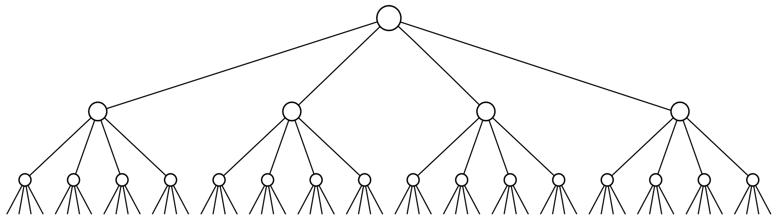
1. způsob: Řešení pomocí rozepisování:

$$\begin{aligned}
T(n) &= 4T(n/2) + \Theta(n) = \\
&= 4T(n/2) + cn = \\
&= 4 \cdot (4T(n/4) + cn/2) + cn = 4^2T(n/4) + 2cn + cn = 4^2T(n/4) + 3cn = \\
&= 4^2 \cdot (2T(n/8) + cn/4) + 3cn = 4^3T(n/8) + 4cn + 3cn = 4^3T(n/8) + 7cn = \\
&\ldots
\end{aligned}$$

Odtud snadno vypozorujeme, že jednotlivé vztahy se vyvíjí podle vzorce $T(n) = 4^kT(n/2^k) + (2^k - 1)cn$. Pro $k = \lceil \log_2 n \rceil$ je ovšem $2^k \leq 1$, takže $T(n/2^k) = \Theta(1)$ a dostaneme (horní celou část zanedbáme, ta ovlivní jen konstanty):

$$T(n) = 4^{\log_2 n}\Theta(1) + (2^{\log_2 n} - 1)cn = n^2\Theta(1) + (n-1)cn = \Theta(n^2).$$

2. způsob: Úvaha o stromu: Nakreslíme si strom rekurzivních volání našeho algoritmu:



Na i-té hladině stromu máme 4^i vrcholů, v nich jsou vstupy velikosti $n/2^i$, takže na celé hladině trávíme čas celkem $\Theta(4^i \cdot n/2^i) = \Theta(2^i n)$. Velikosti vstupů klesají exponenciálně, takže celý strom je hluboký $k = \log_2 n$ (opět si dovolíme zapomenout na horní celou část). Celkem tedy trávíme čas $\sum_{i=0}^{k} \Theta(2^i n) = \Theta(n \cdot \sum_{i=0}^{k} 2^i) = \Theta(n^2)$.

Oba způsoby analýzy se tedy shodují, že náš algoritmus má kvadratickou časovou složitost a že jsme si oproti klasickému algoritmu nikterak nepomohli. Podívejme se ještě jednou na to, jak se náš algoritmus větví:

hloubka	počet úloh	velikost podúlohy
0	4^0	$n/2^0$
1	4^1	$n/2^1$
2	4^2	$n/2^2$
3	4^3	$n/2^3$
$\vdots$	$\vdots$	$\vdots$
k	4^k	$n/2^k$

Naskýtá se otázka, jestli bychom nemohli časovou složitost zlepšit. Toho bychom mohli dosáhnout buďto zlepšením členu cn v naší rekurenci, čili zefektivněním spojování podúloh. To ovšem není příliš nadějné (pokud čtenář nevěří, může si to dokázat), takže místo toho využijeme druhou šanci, a to omezení větvení ze čtyř větví na tři. Připomeňme si, že potřebujeme spočítat:

$$XY = AC \cdot 10^n + (AD + BC) \cdot 10^{n/2} + BD.$$

Přitom ale nepotřebujeme znát součiny AD ani BC samostatně, neboť nám stačí zjistit celý člen $AD + BC$. Kdybychom počítali AC, BD a potom $(A+B)(C+D) = AC + AD + BC + BD$, tak odčítáním $(AC + BD)$ od $AC + AD + BC + BD$ dostaneme hledaný prostřední člen $AD + BC$. Nyní nám již stačí jen tři násobení, ale potřebujeme tři sčítání a jedno odčítání navíc. Ukážeme, že tato komplikace je zanedbatelná oproti práci ušetřené menším větvením. Podívejme se opět na tabulku:

hloubka	počet úloh	velikost podúlohy
0	3^0	$n/2^0$
1	3^1	$n/2^1$
2	3^2	$n/2^2$
3	3^3	$n/2^3$
$\vdots$	$\vdots$	$\vdots$
k	3^k	$n/2^k$

Opět uvažme, kolik práce spotřebujeme v součtu přes všechny hladiny (hloubka stromu k je opět $\lceil \log_2 n \rceil$ a horní celou část zanedbáme):

$$\sum_{i=0}^{k} 3^i \cdot \frac{n}{2^i} = \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i \cdot n = n \cdot \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i = n \cdot \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} =$$

$$= n \cdot \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{1}{2}} = 2 \cdot n \cdot \left[\left(\frac{3}{2}\right)^{k+1} - 1\right] = \mathcal{O}\left(n \cdot \left(\frac{3}{2}\right)^{\log_2 n}\right) =$$

$$= \mathcal{O}\left(n \cdot \frac{3^{\log_2 n}}{2^{\log_2 n}}\right) = \mathcal{O}\left(n \cdot \frac{3^{\log_2 n}}{n}\right) = \mathcal{O}\left(3^{\log_2 n}\right) = \mathcal{O}\left((2^{\log_2 3})^{\log_2 n}\right) =$$

$$= \mathcal{O}\left(2^{(\log_2 n)\cdot \log_2 3}\right) = \mathcal{O}\left((2^{\log_2 n})^{\log_2 3}\right) = \mathcal{O}\left(n^{\log_2 3}\right) = \mathcal{O}\left(n^{1.585}\right).$$

Upravený algoritmus už tedy má lepší časovou složitost, konkrétně $\mathcal{O}(n^{1.585})$. V praxi bychom samozřejmě pro činitele neštěpili až na jednociferná čísla, ale zastavili se u nějaké dostatečně malé délky (řekněme 50 cifer) a tam přepnuli na kvadratický algoritmus, který má menší režii.

(Mimochodem, pro násobení čísel existuje ještě efektívnější algoritmus, který dosahuje časové složitosti $\mathcal{O}(n \log n)$, avšak na to už jsou potřeba trochu pokročilejší techniky, jako je diskrétní Fourierova transformace, takže si jej necháme na příští semestr.)

Master Theorem

Metody řešení rekurentních rovnic pro složitosti rozdělovacích a panovacích algoritmů by asi fungovaly i na jiné algoritmy než jen naše dva příklady, ale proč se pokaždé mořit upravováním výrazů? Raději si dokážeme obecnou větu, která půjde na většinu takových rekurencí nasadit. Říká se jí Master Theorem nebo také (vzhledem k tomu, jak se používá) Kuchařková věta.

Věta: *(Master Theorem)* Předpokládejme, že $T(1) = \mathcal{O}(1)$ a $T(n) = a \cdot T(\lceil \frac{n}{b} \rceil) + \mathcal{O}(n^d)$, kde $a \geq 1$, $b > 1$ a $d \geq 0$. Potom $T(n)$ je:

$\mathcal{O}(n^d)$ když $a < b^d$,
$\mathcal{O}(n^d \cdot \log n)$ když $a = b^d$,
$\mathcal{O}(n^{\log_b a})$ když $a > b^d$.

Důkaz: Předpokládejme nejdříve, že $n = b^m, m \in \mathbb{N}$, aby platilo $\lceil \frac{n}{b} \rceil = \frac{n}{b}$. Použijeme opět „důkaz stromem“. Strom rekurzivních volání se vždy větví na stejný počet větví, konkrétně a, a velikosti vstupů klesají b-krát. Podívejme se na tabulku:

hloubka	počet podúloh	velikost podúlohy	čas na hladině
0	1	n	$\mathcal{O}(n^d)$
1	a	n/b^1	$\mathcal{O}((n/b^1)^d) \cdot a^1$
2	a^2	n/b^2	$\mathcal{O}((n/b^2)^d) \cdot a^2$
3	a^3	n/b^3	$\mathcal{O}((n/b^3)^d) \cdot a^3$
$\vdots$	$\vdots$	$\vdots$	
k	a^k	n/b^k	$\mathcal{O}((n/b^k)^d) \cdot a^k$

Zkoumejme čas potřebný na vyřešení všech podúloh na jedné hladině:

$$\mathcal{O}\left(\left(\frac{n}{b^k}\right)^d\right) \cdot a^k = \mathcal{O}\left(a^k \cdot n^d \left(\frac{1}{b^k}\right)^d\right) = \mathcal{O}\left(a^k \cdot n^d \left(\frac{1}{b^d}\right)^k\right) = \mathcal{O}\left(n^d \left(\frac{a}{b^d}\right)^k\right).$$

Celkem je tedy čas potřebný na vyřešení všech podúloh na všech hladinách (to znamená celé úlohy):

$$T(n) = \sum_{k=0}^{\log_b n} \mathcal{O}\left(n^d \left(\frac{a}{b^d}\right)^k\right) = \mathcal{O}\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)$$

Všimněme si členu $\frac{a}{b^d}$. Na jeho hodnotě závisí hodnota výsledné sumy, protože se vlastně jedná o kvocient geometrické řady. Proto rozlišme následující případy:

1. $\frac{a}{b^d} < 1$ – jinými slovy, práce na jednotlivých hladinách exponenciálně ubývá a součet sumy, i kdyby byla nekonečná, se dá omezit nějakou konstantou, která se schová do $\mathcal{O}$, a tak je $T(n) = \mathcal{O}(n^d)$.

2. $\frac{a}{b^d} = 1$ – práce na jednotlivých hladinách je stejně, to znamená, že součet sumy je právě $\log_b n$, a tedy $T(n) = \mathcal{O}(n^d \cdot \log_b(n))$.

3. $\frac{a}{b^d} > 1$ – to znamená, že práce na jednotlivých hladinách přibývá, takže musíme geometrickou řadu sečíst poctivě: $T(n) = \mathcal{O}(n^d \cdot (\frac{a}{b^d})^{\log_b n})$. Tento výraz vypadá poněkud ošklivě, ale ještě ho trochu (alespoň kosmeticky) upravíme:

$$\mathcal{O}\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right) = \mathcal{O}\left(\frac{a^{\log_b n} \cdot n^d}{(b^d)^{\log_b n}}\right) = \mathcal{O}\left(\frac{\left(b^{\log_b a}\right)^{\log_b n} \cdot n^d}{(b^d)^{\log_b n}}\right) =$$

$$= \mathcal{O}\left(\frac{\left(b^{\log_b n}\right)^{\log_b a} \cdot n^d}{\left(b^{\log_b n}\right)^d}\right) = \mathcal{O}\left(\frac{n^{\log_b a} \cdot n^d}{n^d}\right) = \mathcal{O}\left(n^{\log_b a}\right).$$

Tyto tři případy přesně odpovídají rozdělení případu v tvrzení věty.

Vraťme se nyní k možnosti, kdy n není mocnina b. Tehdy platí $b^l < n < b^{l+1}$ pro nějaké $l \in \mathbb{N}$. V tomto případě zaokrouhleme n a počítejme s $n' = b^{l+1} < bn$. Jelikož funkce $T(n)$ je neklesající, dostaneme $T(n) \leq T(n')$ a ve všech třech případech je výsledek pro n' maximálně konstanta-krát větší než pro n, což se „schová do $\mathcal{O}$." Věta tedy platí i v tomto případě. ♡

Příklad: Porovnejme si některé známé algoritmy a jejich časovou složitost pomocí *Master Theoremu*:

algoritmus	a	b	d	časová složitost
Mergesort	2	2	1	$\mathcal{O}(n \cdot \log n)$
Násobení I.	4	2	1	$\mathcal{O}(n^2)$
Násobení II.	3	2	1	$\mathcal{O}(n^{\log_2 3})$
Binární vyhledávání	1	2	0	$\mathcal{O}(\log n)$

Domácí úkol nakonec: Vymyslete algoritmus, který by z n zadaných bodů v rovině (prostoru) našel takové dva, které jsou od sebe nejméně vzdálené (zde existuje takový algoritmus s časovou složitostí $\mathcal{O}(n \cdot \log n)$).