1. Eukleidův algoritmus *(zapsaly T. Klimošová a K. Böhmová)*

Problém hledání největšího společného dělitele

Cílem je najít největšího společného dělitele zadaných čísel $a_0, b_0 \in N$, dále jen $\gcd(a_0, b_0)$, tedy Greatest Common Divisor.

Algoritmus: (Eukleidův algoritmus verze 1.0; snad 200 let před Eukleidem)

1. *Vstup:* čísla a_0, b_0.
2. $a \leftarrow a_0, b \leftarrow b_0$.
3. while $a \neq b$:
4. if $a > b \Rightarrow a \leftarrow a - b$
5. else $\Rightarrow b \leftarrow b - a$
6. *Výstup:* $a = \gcd(a_0, b_0)$.

Důkaz správnosti:

1) EA se zastaví

V každém průchodu cyklem se buď a zmenší o hodnotu b anebo se b zmenší o hodnotu a. Proto součet $a+b$ klesne pokaždé alespoň o 1. Tedy celkem se uskuteční nanejvýš $a + b$ průchodů cyklem a pak algoritmus skončí.

2) EA vydá $\gcd(a_0, b_0)$

Při důkazech správnosti algoritmu se často používá *invariant.* To je něco, co se v průběhu algoritmu nemění (obvykle hodnota nějakého výrazu nebo jeho vlastnost, např. dělitelnost dvěma).

Dokážeme, že platí invariant: $\gcd(a, b) = \gcd(a_0, b_0)$.

Algoritmus při svém běhu postupně mění hodnoty a a b. Tedy $(a_0, b_0) \rightarrow (a_1, b_1) \rightarrow \ldots \rightarrow (a_k, b_k)$. A jakmile je splněna rovnost $a_k = b_k$, tak skončí. My chceme ukázat, že platí $\gcd(a_0, b_0) = \gcd(a_1, b_1) = \ldots = \gcd(a_k, b_k)$.

K tomu bude třeba několik pozorování:

Pozorování 1: $\gcd(a, a) = a$ (pro $a > 0$).

Nahlédneme snadno. Největší společný dělitel jediného čísla je ono samo.

Pozorování 2: $\gcd(a, b) = \gcd(b, a)$.

Taktéž jednoduché. Při určování největšího společného dělitele nám na pořadí čísel nezáleží.

Pozorování 3: $\gcd(a, b) = \gcd(a \pm b, b)$.

Dokážeme silnější věc – platnost ekvivalence $k|a, k|b \Leftrightarrow k|a, k|(a \pm b)$.

$\Rightarrow$: Jelikož $k|a, k|b$, dá se a vyjádřit jako k-násobek nějakého a'. Tedy $a = k \cdot a'$. Obdobně $b = k \cdot b'$, pro nějaká čísla $a', b' \in N$. Takže součet $a \pm b$ si můžeme rozepsat jako $(a' \pm b') \cdot k$ z čehož je již vidět, že $k|(a \pm b)$.

$\Leftarrow$: Podobně $k|(a \pm b), k|b \Leftrightarrow k|a$.

Uvědomíme si, že tato tři pozorování nám stačí k tomu, aby invariant platil. Algoritmus tedy opravdu vydá $\gcd(a_0, b_0)$. ♡

3) **Rychlost algoritmu**

Algoritmus provede v nehorším případě řádově $a+b$ iterací. To je velmi pomalé, a proto vyvstává otázka, dá-li se nějak zrychlit.

Všimněme si, že opakované odčítání jednoho čísla od druhého lze nahradit výpočtem zbytku po dělení. Algoritmus tedy mužeme upravit následovně:

Algoritmus: (Eukleidův alg. v2.0; neznámý Eukleidův pokračovatel, neznámo kdy)

1. *Vstup:* čísla a_0, b_0.
2. $a \leftarrow a_0, b \leftarrow b_0$.
3. while $b \neq 0$:
4. $\quad a \leftarrow a \bmod b$
5. $\quad a \leftrightarrow b$
6. *Výstup:* $a = \gcd(a_0, b_0)$.

Důkaz správnosti:

Algoritmus určitě vydá správný výsledek, protože počítá totéž, co předchozí verze. Ukážeme, že počet kroků, který provede verze 2.0, je řádově $\log(\max(a, b))$. K tomu se nám bude hodit následující:

Lemma: Za dva průchody cyklem se buď hodnota a nebo hodnota b zmenší alespoň dvakrát.

Důkaz: Rozebereme, co se stane po dvojím projití cyklem. Na začátku máme čísla a_0 a b_0. Bez újmy na obecnosti předpokládejme, že $a_0 > b_0$. Po prvním průchodu získáme $a_1 = b_0; b_1 = a_0 \bmod b_0$, což je méně než b_0. Po druhém průchodu: $a_2 = b_1; b_2 = a_1 \bmod b_1 = b_0 \bmod b_1$. Chceme ukázat, že když je $b_1 < b_0$, tak $b_0 \bmod b_1 \leq b_0/2$.

- Buď platí $b_1 \leq b_0/2$. Pak jistě $b_0 \bmod b_1 < b_1 \leq b_0/2$.
- Nebo nastane druhá možnost $b_1 > b_0/2$. Pak tedy $b_0 \bmod b_1 = b_0 - b_1$, což je taktéž maximálně $b_0/2$. ♡

Nyní stačí lemma šikovně využít: Dvojic průchodů, které zmenšují hodnotu a, je nejvýše $\lceil \log_2 a \rceil$. Obdobně těch snižujících hodnotu b je nejvýše $\lceil \log_2 b \rceil$. Celkový počet průchodů tedy je nejvýše $2(\lceil \log_2 a \rceil + \lceil \log_2 b \rceil)$.

Ještě bychom chtěli dokázat, že náš horní odhad na počet průchodů není příliš velkorysý. Zkusme se podívat, co se stane, když algoritmus necháme spočítat $\gcd(F_n, F_{n-1})$, kde F_i je i-té Fibonacciho číslo. Tehdy všechna modula budou ve skutečnosti odčítání a algoritmus bude počítat $\gcd(F_n, F_{n-1}) \rightarrow \gcd(F_{n-1}, F_{n-2}) \rightarrow \ldots \rightarrow \gcd(1, 1)$.

Díky exponenciální velikosti Fibonacciho čísel tento výpočet bude opravdu trvat logaritmicky dlouho vzhledem k velikosti většího ze zadaných čísel. (Zároveň je to také pěkný důkaz toho, že Fibonacciho čísla jsou navzájem nesoudělná.)

Výpočetní model

Když chceme studovat algoritmy, měli bychom se začít zaobírat otázkou, co takový algoritmus vlastně je. Žádná obecně uznávaná definice algoritmu neexistuje, nadefinujeme si proto alespoň výpočetní model a za algoritmy budeme považovat programy v tomto modelu.

Základní vlastnosti výpočetního modelu:

V první řadě potřebujeme modelu sdělit, s čím má pracovat, a pak se dovědět, jak to dopadlo. Bez újmy na obecnosti můžeme považovat n čísel za *vstup* a za *výstup* m čísel (můžeme se omezit na čísla, neboť cokoli jiného umíme do čísel zakódovat).

Dále by měl výpočetní model umět provádět *elementární operace*, což jsou nejen ty aritmetické $(+, -, *, /, \mathrm{mod}, \ldots)$ a logické (negace, and, or, …), ale také řídící operace (skoky, podmíněné skoky, halt, …) a práci s pamětí.

Definice:

- *Čas běhu algoritmu* $t(x)$ pro vstup x měříme jako počet elementárních operací, které program provedl při zpracování vstupu x.
- *Prostor běhu algoritmu* $s(x)$ je analogicky počet paměťových buněk spotřebovaných při výpočtu se vstupem x.
- *Časová složitost* (v nejhorším případě) je:

$$T(n) := \max\{t(x); x \text{ je vstup délky } n\}.$$

- *Prostorová složitost* (v nejhorším případě) je:

$$S(n) := \max\{s(x); x \text{ je vstup délky } n\}.$$

Díra v definici: Co jsou to čísla? Když mohou být libovolně velká, pak se dá podvádět při počítání časové i prostorové složitosti. Napravit to můžeme buďto tak, že vstup budeme měřit v bitech a operace s k-bitovými čísly bude trvat řádově $\log k$ kroků, nebo že omezíme čísla na polynomiálně velká vzhledem k velikosti vstupu. Pro normální algoritmy tyto dvě definice dávají časové složitosti lišící se vždy logaritmem velikosti vstupu, takže pro srovnávání algoritmů nemusíme tento rozdíl uvažovat. Pro tuto přednášku si vybereme omezení čísel na polynomiálně velká.

Panorama složitostí

Abychom měli představu o tom, jak použitelné jsou algoritmy se kterou časovou složitostí, zkusme si pár složitostí srovnat za předpokladu, že náš počítač za sekundu provede 10^9 operací:

funkce / $n =$	10	20	30	50	100	1 000	1 000 000
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	5.6 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	50 ns	100 ns	1 μs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	147 ns	282 ns	664 ns	10 μs	20 ms
n^2	100 ns	400 ns	900 ns	2.5 μs	100 μs	1 ms	1 000 s
n^3	1 μs	8 μs	27 μs	125 μs	1 ms	1 s	10^9 s
2^n	1 μs	1 ms	1 s	10^6 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	10^6 s	10^{18} s	10^{32} s	10^{64} s	10^{158} s	10^{2567} s	$\approx \infty$

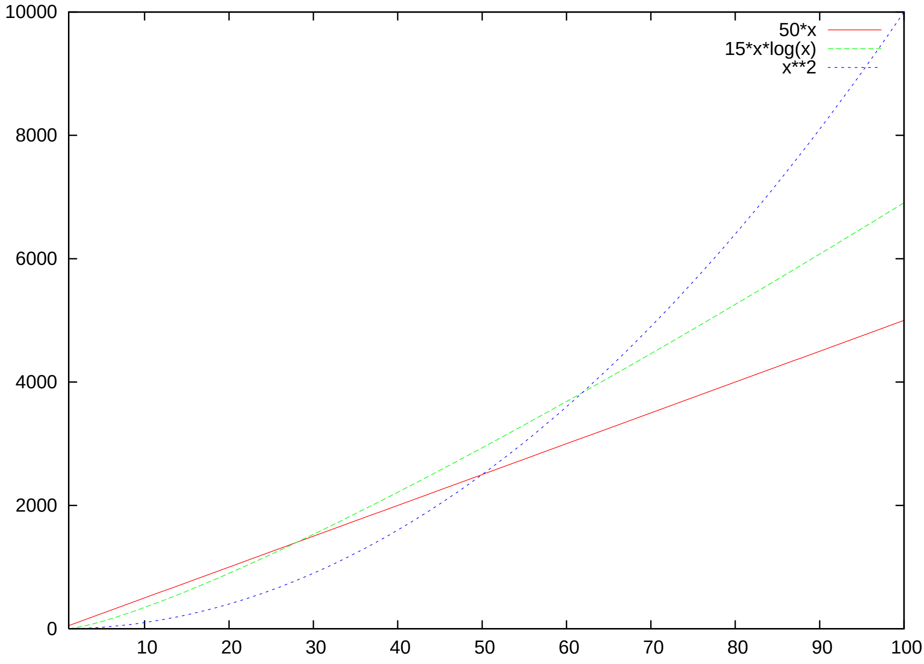
Pro představu:	1 000 s je asi tak čtvrt hodiny
	1 000 000 s je necelých 12 dní
	10^9 s je 31 let
	10^{18} s je asi tak stáří Vesmíru.

Počítače se nám ale zrychlují⟨1⟩ – není tedy lepší chvíli počkat než vymýšlet lepší algoritmus? :-) Podívejme se, jak velký vstup dokážeme zpracovat za 10^6 a $2 \cdot 10^6$ operací:

funkce	10^6 op.	$2 \cdot 10^6$ op.	změna
n	1 000 000	2 000 000	2-krát více
n^2	1 000	1 414	$\sqrt{2} \approx 1.41$-krát více
n^3	100	126	$\sqrt[3]{2} \approx 1.26$-krát více
2^n	20	21	o 1 více

Na některých časových složitostech se tedy exponenciální zrychlování počítačů projevuje jen minimálně.

Jak je vidět na následujícím grafu, konstanty v časové složitosti vůbec nehrají příliš velkou roli, proto se vyplácí najít nejdříve algoritmus, jehož složitost je asymptoticky dobrá, a teprve pak optimalizovat konstanty.



⟨1⟩ „Každé dva roky se výkon počítačů zdvojnásobí“ – Gordon Moore (prý)

V praxi také často pomůže zpracovávat velké vstupy asymptoticky rychlým algoritmem a pro malé vstupy přepnout na „hloupý“ algoritmus, který má malé multiplikativní konstanty.

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?

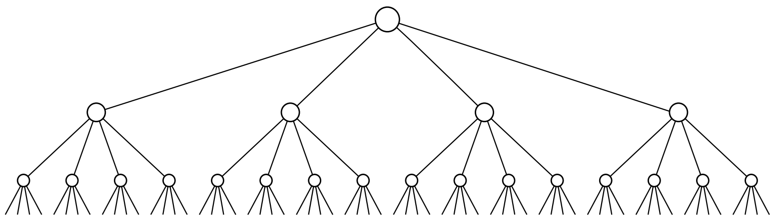
1. způsob: Řešení pomocí rozepisování:

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

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):

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

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

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

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

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:

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

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

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	$\mathcal{O}(\log n)$

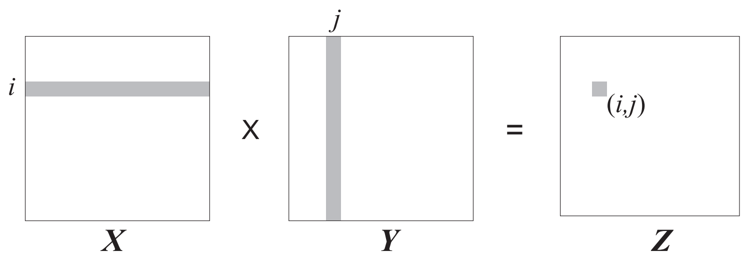
Domácí úkol nakonec: Vymyslete algoritmus, který by z n zadaných bodů v rovině (prostoru) našel takové dva, které jsou od sebe nejméně vzdálené (zde existuje takový algoritmus s časovou složitostí $\mathcal{O}(n \cdot \log n)$).

3. Rozděl a panuj II *(zapsali Lukáš Hermann, Vincent Kríž, Oto Petřík)*

Násobení matic n×n a Strassenův algoritmus

Nejdříve si připomeneme definici násobení dvou čtvercových matic typu $n \times n$. Platí, že prvek v i-tém řádku a j-tém sloupci výsledné matice Z se rovná standardnímu skalárnímu součinu i-tého řádku první matice X a j-tého sloupce druhé matice Y. Formálně zapsáno:

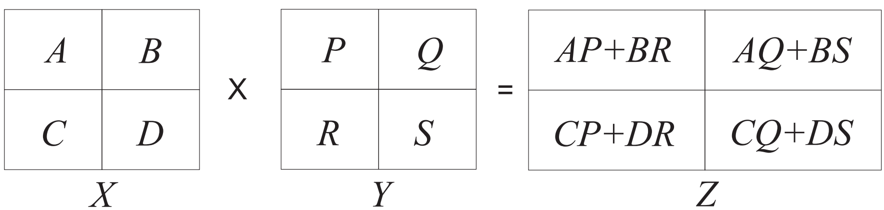
$$Z_{ij} = \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}.$$



Násobení matic

Algoritmus, který by násobil matice podle této definice, by měl časovou složitost $\Theta(n^3)$, protože počet prvků ve výsledné matici je n^2 a jeden skalární součin vektorů dimenze n vyžaduje lineární počet operací.

My se s touto časovou složitostí ovšem nespokojíme a budeme postupovat podobně jako při vylepšování algoritmu na násobení velkých čísel. Bez újmy na obecnosti předpokládejme, že budeme násobit dvě matice typu $n \times n$, kde $n = 2^k, k \in \mathbb{N}$. Obě tyto matice rozdělíme na čtvrtiny a tyto části postupně označíme u matice X písmeny A, B, C a D, u matice Y písmeny P, Q, R a S. Z definice násobení matic zjistíme, že čtvrtiny výsledné matice Z můžeme zapsat pomocí součinů částí násobených matic. Levá horní čtvrtina bude odpovídat výsledku operací $AP + BR$, pravá horní čtvrtina bude $AQ + BS$, levá dolní $CP + DR$ a zbylá $CQ + DS$ (viz obrázek).



Násobení rozčtvrcených matic

Převedli jsme tedy problém násobení čtvercových matic řádu n na násobení čtvercových matic řádu $n/2$. Tímto rozdělováním bychom mohli pokračovat, dokud bychom se nedostali na matice řádu 1, jejichž vynásobení je triviální. Dostali jsme tedy klasický algoritmus typu *rozděl a panuj*. Pomohli jsme si ale nějak ? V každém kroku provádíme 8 násobení matic polovičního řádu a navíc konstantní počet operací na n^2 prvcích. Dostáváme tedy rekurentní zápis časové složitosti:

$$T(n) = 8T\left(\frac{n}{2}\right) + \mathcal{O}(n^2).$$

Použitím Master Theoremu lehce dojdeme k závěru, že složitost je stále $\Theta(n^3)$, tedy stejná jako při násobení matic z definice. Zdánlivě jsme si tedy nepomohli, ale stejně jako tomu bylo u násobení velkých čísel, i teď můžeme zredukovat počet násobení matic polovičního řádu, které nejvíce ovlivňuje časovou složitost algoritmu. Není to bohužel nic triviálního, a proto si raději rovnou řekneme správné řešení. Jedná se o Strassenův algoritmus, který redukuje potřebný počet násobení na 7, a ještě před tím, než si ukážeme, jak funguje, dokážeme si, jak nám to s časovou složitostí vlastně pomůže:

$$T(n) = 7T\left(\frac{n}{2}\right) + \mathcal{O}(n^2) \Longrightarrow \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.808}).$$

Výsledná složitost Strassenova algoritmu je tedy $\mathcal{O}(n^{2.808})$, což je sice malé, ale pro velké matice znatelné zlepšení oproti algoritmu vycházejícímu přímo z definice.

Lemma: (vzorce pro násobení blokových matic ve Strassenově algoritmu)

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} P & Q \\ R & S \end{pmatrix} = \begin{pmatrix} T_1 + T_4 - T_5 + T_7 & T_3 + T_5 \\ T_2 + T_4 & T_1 - T_2 + T_3 + T_6 \end{pmatrix},$$

kde:

$$\begin{aligned} T_1 &= (A + D) \cdot (P + S) & T_5 &= (A + B) \cdot S \\ T_2 &= (C + D) \cdot P & T_6 &= (C - A) \cdot (P + Q) \\ T_3 &= A \cdot (Q - S) & T_7 &= (B - D) \cdot (R + S) \\ T_4 &= D \cdot (R - P) \end{aligned}$$

Důkaz: Do čtverců 4×4 si napíšeme znaky $+$ nebo $-$ podle toho, jestli se při výpočtu dané matice přičítá nebo odečítá příslušný součin dvou matic. Řádky znamenají matice A, B, C a D a sloupce značí matice P, Q, R a S. Pokud se tedy v prvním řádku a prvním sloupci vyskytuje znak $+$, znamená to že přičteme součin matic A a P. Nejdříve si spočítáme pomocné matice T_1 až T_7 a z nich pak dopočítáme, co bude na příslušných místech ve výsledné matici.

$$T_1 = \boxed{\begin{array}{cccc} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{array}} \qquad T_2 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{array}} \qquad T_3 = \boxed{\begin{array}{cccc} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}} \qquad T_4 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{array}}$$

$$T_5 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}} \qquad T_6 = \boxed{\begin{array}{cccc} - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}} \qquad T_7 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{array}}$$

$$T_1 + T_4 - T_5 + T_7 = \boxed{\begin{array}{cccc} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}} = AP + BR$$

$$T_3 + T_5 = \boxed{\begin{array}{cccc} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{array}} = AQ + BS$$

$$T_2 + T_4 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{array}} = CP + DR$$

$$T_1 - T_2 + T_3 + T_6 = \boxed{\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{array}} = CQ + DS$$

Jak je vidět, výsledná matice je tvořena stejnými částmi jako při obyčejném násobení. Touto kapitolou jsme tedy dokázali následující větu:

Věta: Strassenův algoritmus pro násobení matic $n \times n$ má časovou složitost v nejhorším případě $\mathcal{O}(n^{2.808})$. ♡

Poznámka: Zatím nejlepší dokázaný algoritmus má časovou složitost $\mathcal{O}(n^{2.376})$, ovšem s velkou multiplikativní konstantou.

Hledání k-tého nejmenšího prvku (mediánu)

V tomto oddílu se budeme zabývat tím, jak co nejrychleji najít v jakékoli posloupnosti n čísel k-tý nejmenší prvek popřípadě medián. Pro ty, kdo medián neznají, tu máme definici:

Definice: Medián posloupnosti $a_1, a_2, \ldots, a_n$ je takové $m = a_i$, kde nejvýše $n/2$ prvků je menších než m a nejvýše $n/2$ prvků je větších než m.

Nejjednodušším řešením by určitě bylo celou posloupnost nejdříve setřídit a pak už jednoduše vyhmátnout požadovaný prvek. To bychom dokázali v celkem slušném čase $\mathcal{O}(n \log n)$, ale už teď můžeme prozradit, že to jde v čase $\mathcal{O}(n)$. Jak?

Používat budeme metodu *rozděl a panuj*. Nějakým způsobem si zvolíme jeden prvek posloupnosti, který nazveme *pivot*. Poté rozdělíme zadanou posloupnost na tři disjunktní množiny. Do první dáme všechny prvky menší než pivot, do druhé stejné jako pivot a do třetí větší než pivot. Tímto máme zajištěno, že prvky z první množiny jsou určitě menší než prvky z druhé a ty než prvky z třetí. O tom, jak jsou prvky uspořádány uvnitř těchto množin, ale nic nevíme.

V posledním kroku našeho algoritmu se pak rozhodneme, na kterou množinu svůj algoritmus rekurzivně zavoláme. Pokud je k menší než velikost první množiny, pokračujeme v první množině, pokud je k menší než součet velikostí první a druhé množiny, pak hledaným prvkem je právě vybraný pivot a algoritmus skončí, a nakonec pokud ani jedna podmínka splněna nebyla, pustíme se do hledání ve třetí množině, ovšem už nehledáme k-tý nejmenší prvek, ale l-tý, kde l se rovná k minus velikost prvních dvou množin. Pro větší názornost zapíšeme tento algoritmus formálněji:

Select(k, X): (Hledání k-tého nejmenšího prvku v množine X)

1. Jestliže $|X| \leq 1$, vyřešíme triviálně.
2. Zvolíme pivota $p \in X$.
3. Rozdělíme množinu X na tři podmnožiny: $M = \{x \in X; x < p\}$, $P = \{x \in X; x = p\}, V = \{x \in X; x > p\}$.
4. Jestliže $k \leq |M|$, vrátím výsledek funkce *Select*(k, M).
5. Jestliže $k \leq |M| + |P|$, vrátíme pivota p.
6. Jinak vrátíme výsledek funkce *Select*$(k - |M| - |P|, V)$.

Na první pohled je vidět, že se algoritmus zastaví (vstup se vždy zmenší alespoň o 1) a že vydá vždy správný výsledek. Jak je to ovšem s časovou složitostí? Rozdělení do množin a podmínky v druhém a třetím kroku mají lineární složitost, čemuž se nevyhneme. Při nešťastné volbě pivota se nám může stát, že počet rekurencí může být až n, tedy celková složitost v nejhorším případě je $\Theta(n^2)$, čímž jsme si oproti prostému setřídění ještě pohoršili. Co s tím? Jak je vidět, velmi důležitá je volba pivota. Tu můžeme provést několika způsoby:

a) Pivot by se v setříděné posloupnosti vyskytoval uprostřed, vstup by se tedy stále půlil. Časovou složitost vypočteme z rekurentního zápisu:

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(n) = \mathcal{O}\left(n + \frac{n}{2} + \frac{n}{4} + \dots\right) = \mathcal{O}(n).$$

To by bylo sice skvělé, ale nalezení takového pivota je vlastně vyřešení úlohy hledání mediánu, o což se snažíme. Tedy jsme si vůbec nepomohli.

b) Pivot by se v setříděné posloupnosti náchazel v prostředních dvou čtvrtinách (nazvěme tento prvek "lžimedián"). Tím bychom v každém kroku určitě odstranili množinu velikosti čtvrtiny vstupu. Časová složitost tohoto řešení by byla:

$$T(n) = T\left(\frac{3}{4}n\right) + \mathcal{O}(n) = \mathcal{O}\left(n + \frac{3}{4}n + \frac{9}{16}n + \dots\right) = \mathcal{O}(n).$$

Tímto bychom tedy také dosáhli lineární časové složitosti. Ale jak vybrat pivota tak, aby se nacházel v prostředních dvou čtvrtinách, aby nám nám to nepokazilo lineární složitost?

c) Pivot bude vždy prostřední prvek zadané posloupnosti. V tomto případě je zřejmé, že existují určité vstupy, pro které bude časová složitost $\Omega(n^2)$, ale v průměrném případě, bude časová složitost $\mathcal{O}(n)$, což dokážeme v příští kapitole.

d) Pivot bude náhodně zvolený prvek zadané posloupnosti. Tímto dosáhneme něčeho podobného jako v předchozím případě, ale už nebude existovat vstup, pro který by časová složitost byla $\Omega(n^2)$, jen některé průběhy tohoto algoritmu na různých vstupech budou trvat takto dlouho. Průměrná časová složitost je tedy zase $O(n)$.

My se ale nespokojíme pouze s lineární průměrnou časovou složitostí. Chceme dosáhnout lineární složitosti i v nejhorším případě. Než si ale prozradíme takové řešení, rozmyslíme si, že už teď známe 3 časové složitosti, i když ne všechny ještě umíme analyzovat. Je to časová složitost v nejhorším přídadě, v průměrném případě přes všechny vstupy a v průměrném případě přes všechny běhy programu. Všimněte si hlavně rozdílu posledních dvou.

A nyní už slibované řešení s lineární nejhorší časovou složitostí. Vyjdeme z možnosti b), tedy pokusíme se najít takového pivota, který by na příští krok zaručeně omezil velikost analyzované množiny. Dosáhneme toho tímto algoritmem:

Volba pivota:

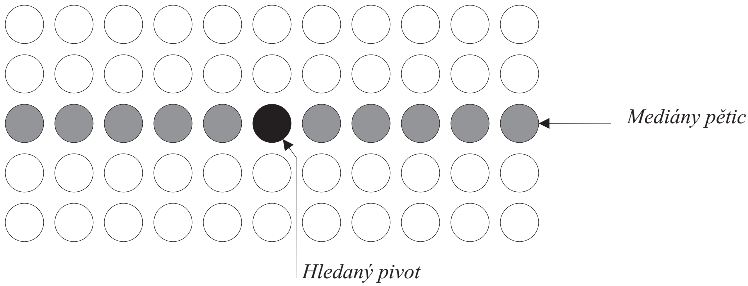
1. Rozdělíme vstup na pětice . . . $\mathcal{O}(n)$
2. Spočteme medián každé pětice . . . $\mathcal{O}(n)$
3. Spočteme medián mediánů pětic tak, že rekurzivně zavoláme tentýž algoritmus *Select*$(n/10$, mediány pětic) a to je pivot.

Abychom dokázali, že tento algoritmus bude mít opravdu lineární časovou složitost, musíme si nejdříve dokázat následující lemma:

Lemma: V každém kroku vypadne alespoň $3/10 \cdot n$ prvků.

Důkaz: Ten provedeme obrázkem. Představme si vybrané pětice seřazené podle velikosti od největšího prvku a zakresleme je do sloupců. Jejich mediány tedy vyplňují prostřední řadu. Tyto pětice pak seřaďme podle velikosti jejich mediánů (nejmenší vlevo). Hledaný pivot se tedy nachází (pokud předpokládáme pro jednoduchost lichý počet pětic) přesně uprostřed. O prvcích nad pivotem a napravo od něj můžeme určitě říct, že jsou větší nebo rovny pivotu, prvky pod ním a nalevo od něj jsou zase určitě menší nebo rovny pivotu.

Podle konstrukce algoritmu tedy zaručeně vypadne jedna nebo druhá skupina prvků. Obě tyto skupiny přitom obsahují, jak je vidět z obrázku, alespoň $3/10 \cdot n$ prvků. ♡



Pětice

Nyní už se tedy můžeme pustit do výpočtu časové složitosti. V každém kroku funkce zavolá sama sebe nejdříve na vstup velikosti $\frac{n}{5}$ a poté na vstup velikosti nejvýše $\frac{7}{10}n$. Ostatní operace zůstávají lineární. Časovou složitost v nejhorším případě tedy můžeme zapsat rekurentním vzorcem:

$$T(n) = \mathcal{O}(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right).$$

Tento rekurentní vzorec ale zatím neumíme obecně řešit. Mohli bychom ho postupně rozepsat, ale trvalo by dlouho, než bychom z toho něco vykoukali. Lepší bude rovnou dokázat, že tento rekurentní vzorec odpovídá lineární složitosti, zkusíme tedy dosadit $T(n) = cn$:

$$cn = n + \frac{1}{5}cn + \frac{7}{10}cn \quad \Longleftrightarrow \quad c = 10.$$

Tímto jsme tedy dokázali, že časová složitost tohoto algoritmu v nejhorším případě je $\mathcal{O}(n)$. Jelikož to rychleji určitě není možné, můžeme na závěr této kapitoly zformulovat tuto větu:

Věta: Hledání k-tého nejmenšího prvku v posloupnosti délky n má časovou složitost v nejhorším případě $\Theta(n)$. ♡

4. Průměrná časová složitost *(R. Cinkais, R. Barczi a R. Vansa)*

První pomoc z teorie pravděpodobnosti

Definice: *Diskrétní pravděpodobnostní prostor* je uspořádaná dvojice (Ω, P), kde Ω je nejvýše spočetná množina *elementárních jevů* a $P : \Omega \to [0,1]$ je funkce, pro kterou platí:

$$\sum_{x \in \Omega} P(x) = 1.$$

Hodnotám $P(x)$ budeme říkat *pravděpodobnosti* jednotlivých elementárních jevů.

Příklad: Množina elementárních jevů při vrhu kostkou je $\Omega = \{1, 2, 3, 4, 5, 6\}$ a pokud je kostka spravedlivá, platí $\forall x \in \Omega : P(x) = 1/6$.

Definice: *Jevem* A rozumíme libovolnou množinu $A \subseteq \Omega$. *Pravděpodobnost* jevu A počítáme jako:

$$\Pr[A] = \sum_{a \in A} P(a).$$

Příklad: Jaká je pravděpodobnost složeného jevu „na kostce padne sudé číslo“? Označíme $A = \{2, 4, 6\} \subseteq \Omega$. Hledanou pravděpodobností je pak $P(A) = 1/6 + 1/6 + 1/6 = 1/2$.

Definice: *Náhodná proměnná* X je funkce $X : \Omega \to \mathrm{R}$. *Střední hodnotou* náhodné proměnné X rozumíme výraz

$$\mathrm{E}\, X = \sum_{x \in \Omega} P(x) \cdot X(x).$$

Příklad: Nechť H je náhodná proměnná přiřazující vrhu kostkou dosaženou hodnotu, tedy $\forall x \in \Omega : H(x) = x$. Střední (očekávanou) hodnotou této proměnné je $\mathrm{E}\, H = 1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2$.

Věta: (*Linearita střední hodnoty*) Nechť $A, B : \Omega \to \mathrm{R}$ jsou náhodné proměnné a $\alpha \in \mathrm{R}$ je libovolná konstanta. Potom:

1. $\mathrm{E}\,(A + B) = \mathrm{E}\,(A) + \mathrm{E}\,(B)$,
2. $\mathrm{E}\,(\alpha \cdot A) = \alpha \cdot \mathrm{E}\,(A)$.

Důkaz:

1. Podle definice střední hodnoty je:

$$\begin{aligned}\mathrm{E}\,(A + B) &= \sum_{x \in \Omega} (A(x) + B(x)) \cdot P(x) = \sum_{x \in \Omega} (A(x) \cdot P(x) + B(x) \cdot P(x)) = \\ &= \sum_{x \in \Omega} A(x) \cdot P(x) + \sum_{x \in \Omega} B(x) \cdot P(x) = \mathrm{E}\, A + \mathrm{E}\, B.\end{aligned}$$

2. Opět přímo z definice střední hodnoty:

$$\mathrm{E}\,(\alpha \cdot A) = \sum_{x \in \Omega} \alpha \cdot A(x) \cdot P(x) = \alpha \cdot \sum_{x \in \Omega} A(x) \cdot P(x) = \alpha \cdot \mathrm{E}\, A.$$

♡

Příklad: Házíme dvěmi odlišnými hracími kostkami. Uvažovaný prostor elementárních jevů je proto $\Omega = \{(1, 1), (1, 2), \ldots, (6, 5), (6, 6)\}$ a $\forall (a, b) \in \Omega : P((a, b)) = 1/36$. Mějme náhodnou proměnnou S přiřazující vrhu kostkami součet dosažených hodnot, tedy $\forall (a, b) \in \Omega : S((a, b)) = a + b$. Chceme určit $\mathrm{E}\, S$.

Označme A náhodnou proměnnou, která při vrhu kostkami vrací hodnotu na první kostce, tedy $\forall(a,b) \in \Omega : A((a,b)) = a$. Podobně označme B náhodnou proměnnou, která vrací hodnotu na druhé kostce, tedy $\forall(a,b) \in \Omega : B((a,b)) = b$. V předcházejícím příkladě jsme si ukázali, že $\mathrm{E}\,A = \mathrm{E}\,B = 7/2$. Protože $S((a,b)) = a + b = A((a,b)) + B((a,b))$, použitím věty o linearitě střední hodnoty dostáváme $\mathrm{E}\,S = \mathrm{E}\,(A+B) = \mathrm{E}\,A + \mathrm{E}\,B = 7/2 + 7/2 = 7$.

Poznámka: Pro každou náhodnou proměnnou Y platí:

$$\mathrm{E}\,Y = \sum_{x \in \mathrm{R}} x \cdot \Pr[Y = x].$$

Důkaz: Uvedený vztah plyne okamžitě z definice střední hodnoty, sloučíme-li dohromady všechny členy $x \in \Omega$ se stejnou hodnotou $Y(x)$. ♡

Průměrná složitost algoritmu

Průměrnou složitost algoritmu můžeme počítat přes všechny vstupy nebo přes náhodné volby vstupů. V prvním případě je algoritmus pravděpodobnostní („hází mincí“) a složitost počítáme jako průměr přes všechny vstupy. Ve druhém je zase algoritmus deterministický a zajímá nás pouze složitost v nejhorším případě. Pokud jde o konečný výsledek, jsou oba uvedené způsoby ekvivalentní.

Našim cílem bude odvodit průměrnou složitost algoritmu *Select*(k, X) z minulé přednášky. K tomu si ukážeme algoritmus sloužící k výběru pivota pro *Select*(k, X) (*Select*(k, X) jej při svém běhu bude spouštět jako podprogram). Bez újmy na obecnosti předpokládejme, že všechny prvky množiny X jsou navzájem různé.

Algoritmus: (*Hledání lžimediánu v množině* X)

1. Zvol náhodně $x \in X$ (všechna x se stejnou pravděpodobností).
2. Spočítej $A = |\{y \in X : y < x\}|$ a $B = |\{y \in X : y > x\}|$.
3. Pokud $A \geq |X|/4$ a zároveň $B \geq |X|/4$, vrať x, jinak skoč na 1.

Pozorování: Uvažujme, že házíme spravedlivou mincí. Aby nedošlo k žádnému mezinárodnímu sporu, označme strany mince jako tučňák (T) a ryba (R). Tedy pravděpodobnost pádu na nějakou stranu je $1/2$. Budeme zkoumat, jaká je střední hodnota čekání na pád tučňáka $\mathrm{E}\,T$. Množinu všech jevů (posloupností vrhů), které mohou nastat, označme jako $\Omega = \{T, RT, RRT, \ldots, RRR\ldots\}$. Platí:

$$\Pr[T] = \frac{1}{2}, \Pr[RT] = \frac{1}{4}, \Pr[R^kT] = \frac{1}{2^{k+1}}, \Pr[RRR\ldots] = \lim_{k\to\infty} \frac{1}{2^k} = 0.$$

Podle pozorování máme:

$$\mathrm{E}\,T = \sum_{k=1}^{\infty} k \cdot \underbrace{\Pr[T = k]}_{\frac{1}{2^k}} = \sum_{k=1}^{\infty} \frac{k}{2^k}.$$

Tuto sumu je však trochu obtížné spočítat (kdo chce si to může zkusit), proto provedeme následující trik:

$$\mathrm{E}\,T = 1 + \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot \mathrm{E}\,T,$$

to znamená, že po jednom tahu jsme s poloviční pravděpodobností vyhráli a s poloviční pravděpodobností se situace nezměnila. Jednoduchou úpravou tedy zjistíme, že:

$$\mathrm{E}\,T = 2.$$

Lemma: Čekání na jev, který nastane s pravděpodobností p, trvá průměrně $1/p$.

Důkaz: $\mathrm{E}\,T = 1 + (1-p)\,\mathrm{E}\,T$, tedy $\mathrm{E}\,T = 1/p$. ♡

Při hledání lžimediánu máme poloviční šanci, že se trefíme při náhodném výběru do prostředních dvou čtvrtin – průměrně tedy vykonáme dva průchody cyklem, přičemž jeden průchod trvá $\Theta(n)$. Celkem bude tedy hledání lžimediánu trvat průměrně $\Theta(n)$.

Věta: *Select* s náhodnou volbou pivota má v průměru časovou složitost $\Theta(n)$.

Důkaz: Rozdělíme algoritmus na *fáze* $\equiv$ posloupnosti iterací končící dobrou volbou pivota (jako lžimedián). Všimneme si, že jedna fáze zmenší vstup na nejvýše $\frac{3}{4} \cdot n$ a trvá v průměru $\Theta(n)$ (průměrně 2 iterace po $\Theta(n)$). Tedy $\mathrm{E}\,T = \mathrm{E}\,T_1 + \mathrm{E}\,T_2 + \ldots = \Theta(n)$, kde $\mathrm{E}\,T_i$ označuje i-tou fázi. ♡

Věta: *Select* s pivotem rovným prostřednímu prvku má v průměru přes permutace na vstupu časovou složitost $\Theta(n)$.

Důkaz: Ukážeme, že algoritmus s pevnou volbou pivota se na náhodném vstupu chová stejně, jako algoritmus s náhodnou volbou pivota na pevném vstupu, jehož průměrnou složitost odhaduje předchozí věta.

Pokud je na vstupu náhodná permutace π, jsou všechny hodnoty pivota p stejně pravděpodobné. Zbývá ukázat, že další iterace algoritmu dostane na vstupu prvky větší nebo menší než p opět v náhodném pořadí (s rovnoměrným rozdělením pravděpodobností). K tomu sestrojíme bijekci mezi množinou všech permutací na $\{1, \ldots, n\}$ a množinou čtveřic (p, L, π_M, π_V), kde:

- $p \in \{1, \ldots, n\}$ je prostřední prvek permutace π (pivot), tedy $\pi[l]$ ležící na pozici $l = \lfloor (1+n)/2 \rfloor$,
- $L \in \binom{\{1,\ldots,l-1,l+1,\ldots,n\}}{p-1}$ je množina pozic, na nichž jsou umístěny prvky menší než p (ty, které náš algoritmus umístí do množiny M),
- π_M permutace na $\{1, \ldots, p-1\}$ popisující pořadí menších prvků a
- π_V permutace na $\{p+1, \ldots, n\}$, tedy pořadí větších prvků.

Jakmile zvolíme p, jsou volby L, π_M a π_V navzájem nezávislé. Každé volbě π_M tedy odpovídá stejný počet permutací π, a proto jsou všechny π_M stejně pravděpodobné. Analogicky pro π_V.

♡

5. QuickSort a složitost třídění *(Michal Staša, Jan Návrat)*

Dostaneme posloupnost, jejíž prvky dovedeme porovnávat, a snažíme se co nejefektivněji posloupnost setřídit. Ukážeme si třídící algoritmus QuickSort (pro přátele QSort nebo QS) založený na metodě Rozděl a panuj:

Algoritmus: (QuickSort)

1. Pokud $|X| \leq 1$, vrátíme X.
2. Vybereme pivota $p \in X$.⟨2⟩
3. $M \leftarrow \{x \in X; x \leq p\}$.
4. $P \leftarrow \{x \in X; x = p\}$.
5. $V \leftarrow \{x \in X; x \geq p\}$.
6. Vrátíme $\mathit{Quicksort}(M) \,.\, P \,.\, \mathit{Quicksort}(V)$. ⟨3⟩

Rozbor: Všimněme si, že QS se určitě zastaví a také že vydá správný výsledek. To můžeme ověřit například indukcí podle $|X|$.

Podobně jako u vybírání k-tého nejmenšího prvku v minulých přednáškách i zde časová složitost závisí hlavně na volbě pivota. Kdybychom za pivota zvolili medián, vyšla by časová složitost stejně jako u MergeSortu:

$$T(n) = 2T(n/2) + O(n) = O(n \log n).$$

Pokud naopak budeme volit pivota nešikovně, dostaneme:

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2).$$

V ideálním případě bychom tedy chtěli za pivota zvolit medián, avšak jeho přímým výpočtem bychom algoritmus příliš zpomalili. Používá se proto mnoho způsobů, jak vybrat rychle pivota blízkého mediánu. Často používanou metodou je náhodný výběr, v praxi realizovaný nějakým pseudonáhodným generátorem. Ukážeme, že v tomto případě je QS v průměrném případě rychlý:

Věta: QS s náhodnou volbou pivota má časovou složitost v průměru $\mathcal{O}(n \log n)$.

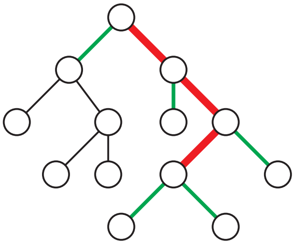
Poznámka: Stejně jako u výběru k-tého nejmenšího prvku bychom také mohli ukázat, že QS s pevnou volbou pivota spuštěný na náhodnou permutaci má tutéž časovou složitost. Detaily nicméně vynecháme.

Důkaz věty:

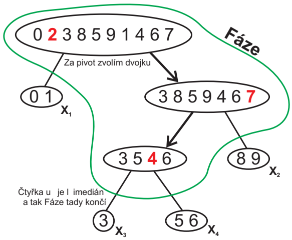
Důkaz provádíme rozdělením algoritmu na fáze. Rekurzivní volání QSortu zle zobrazit jakožto strom, přičemž fází rozumíme cestu ve stromu volání, která sleduje větší díl a končí, když se povede vybrat za pivota lžimedián.

⟨2⟩ později upřesníme, jak

⟨3⟩ Operátor „.“ značí zřetězení seznamů



Důkaz rozdělením na fáze



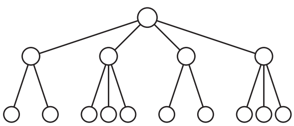
Fáze

Každá fáze přitom rozdělí vstup na disjunktní části $X_1, \ldots, X_k$ $(k \geq 2)$ a pivoty, kteří je oddělují. Označme si n za velikost vstupu (počet prvků vstupní posloupnosti) a n_i za velikost i-té části. Nahlédneme, že platí $\sum_i n_i \leq n$. Velikost každé části je navíc nejvýše $3/4 \cdot n$ (na konci fáze to platí proto, že jsme zvolili lžimedián, předtím jsme vždy oddělili menší z částí, čili nejvýše $n/2$).

Jedna iterace algoritmu trvá $\mathcal{O}(n)$ a jelikož lžimedián vybereme s pravděpodobností alespoň 1/2, je v jedné fázi v průměru $\mathcal{O}(1)$ iterací a celá fáze proto v průměru trvá čas $\mathcal{O}(n)$. Z toho dostaneme následující rekurenci pro průměrnou časovou složitost celého algoritmu:

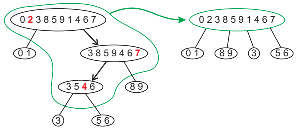
$$\mathrm{E}\, T(n) = \sum_i \mathrm{E}\, T(n_i) + \mathcal{O}(n).$$

Tento typ rekurence jsme ještě nepotkali a Kuchařková věta⟨4⟩ na ni nezabere, ovšem můžeme si pomoci jednoduchou úvahou. Představme si, že v našem stromu rekurzivních volání zkomprimujeme každou fázi do jednoho vrcholu. Tím vznikne strom, který odpovídá algoritmu, jenž v jedné iteraci v průměrně lineárním čase rozdělí vstup na několik částí a rekurzivně se na ně zavolá.



Komprimovaný Strom

⟨4⟩ MasterTheorem



Způsob vytvoření komprimovaného stromu

Nový strom má logaritmickou hloubku, protože na každé další hladině jsou délky posloupností nejvýše 3/4 délek z předchozí hladiny. Navíc součet délek přes každou hladinu je maximálně n. Proto na každé hladině trávíme čas v průměru $\mathcal{O}(n)$ a v celém stromu tedy $\mathcal{O}(n \log n)$. ♡

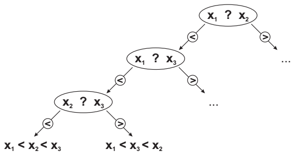
Pozorování: Naše první verze QS spotřebuje lineární množství paměti na pomocné pole a na zásobník. Na přednášce jsme ukazovali různá jeho praktická vylepšení, které stačí pomocná paměť o velikosti $\mathcal{O}(\log n)$. Detaily viz webové stránky přednášky.

Známe už několik třídících algoritmů s časovou složitostí $\mathcal{O}(n \log n)$. Následující věta ukazuje, že efektivnější algoritmus v obecném případě neseženeme.

Věta: Každý třídící algoritmus založený na porovnávání a prohazování prvků potřebuje na vstup délky n v nejhorším případě $\Omega(n \log n)$ porovnání.

Důkaz: Bez újmy na obecnosti budeme nejdříve předpokládat o algoritmu dvě věci: Jednak to, že algoritmus nejprve porovnává, a teprve potom prohazuje.[5] Také předpokládáme, že vstup algoritmu je permutace na množině $\{1, \ldots, n\}$.

Chování tohoto algoritmu popíšeme rozhodovacím stromem. V rozhodovacím stromu vnitřní vrcholy určují jednotlivá porovnání prvků a listy odpovídají okamžikům, kdy algoritmus přestal porovnávat a začal prohazovat.



Rozhodovací Strom

Vstup je tedy permutace n prvků, a víme že počet různých permutací je $n!$. Existuje tedy právě $n!$ různých vstupů. Dále si všimneme, že nemohou existovat dvě vstupní permutace, pro které by algoritmus skončil v témže listu rozhodovacího stromu. Listů stromu je tedy alespoň tolik, kolik je různých vstupů, tedy $n!$.

Lemmátko: Binární strom hloubky k má nejvýše 2^k listů.

Důkazík: Uvažme binární strom hloubky k s maximálním počtem listů. V takovém stromu budou všechny listy určitě ležet na poslední hladině (kdyby neležely, můžeme pod některý list na vyšší hladině přidat další

[5] Algoritmus můžeme upravit tak, aby si pamatoval aktuální permutaci prvků a podle ní prohazoval až na konci.

dva vrcholy a získat tak „listnatější“ strom stejné hloubky). Jelikož na i-té hladině je nejvýše 2^i vrcholů, všech listů je nejvýše 2^k. ♡

Z tohoto lemmátka plyne, že rozhodovací strom musí být hluboký alespoň $\log n!$.

Zbytek je už snadné cvičení z diskrétní matematiky:

Lemmátko: $n! \geq n^{n/2}$.

Důkazík: $n! = \sqrt{(n!)^2} = \sqrt{1(n-1)\cdot 2(n-2)\cdot \ldots \cdot n\cdot 1}$, což můžeme také zapsat jako $\sqrt{1(n-1)}\cdot\sqrt{2(n-2)}\cdot\ldots\cdot\sqrt{n\cdot 1}$. Přitom pro každé $1 \leq k \leq n$ je $k(n+1-k) = kn+k-k^2 = n+(k-1)n+k(1-k) = n+(k-1)(n-k) \geq n$. Proto je každá z odmocnin větší nebo rovna $n^{1/2}$ a $n! \geq (n^{1/2})^n = n^{n/2}$. ♡

Hloubka stromu tedy činí minimálně $\log n! \geq \log(n^{n/2}) = n/2 \cdot \log n = \Omega(n \log n)$, což také zdola odhaduje počet porovnání, který algoritmus provede v nejhorším případě. ♡

6. Třídění v lineárním čase *(zapsal M. Kupec, J. Volec, J. Ivánek)*

Na minulých přednaškách jsme si ukázali třídění v čase $\mathcal{O}(N \log N)$ a také dokázali, že líp to v obecném případě nejde. Jak je tedy možné třídit v lineárním čase?

Všechny dosud předvedené alrogitmy třídily pouze pomocí porovnávní hodnot na vstupu. Ale co když o vstupu víme víc, třeba rozsah hodnot na vstup, nedá se toho využít?

Následující algoritmy využívají znalosti rozsahu hodnot na vstupu a toho, že tento rozsah je malý.

Counting sort

Counting sort je algoritmus pro třídění N celých čísel s maximálním rozsahem hodnot R. Třídí v čase $\mathcal{O}(N+R)$ s paměťovou náročností $\mathcal{O}(R)$.

Algoritmus postupně prochází vstup a počítá si ke každému prvku z rozsahu kolikrát jej viděl. Poté až projde celý vstup, projde počítadla a indexy u tech, kde něco napočítal vydá jako výstup.

Algoritmus: (třídění posloupnosti $(x_i)_0^N$ pomocí Counting sortu)

1. Pro $i \leftarrow 1$ do R opakuj:
2. $\quad p_i \leftarrow 0$
3. Pro $i \leftarrow 1$ do N opakuj:
4. $\quad p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$
6. Pro $i \leftarrow 1$ do R opakuj:
7. $\quad$ Pokud $p_i \neq 0$
8. $\qquad v_j \leftarrow i$
9. $\qquad j \leftarrow j + 1$
10. Vrať v

Přihrádkové třídění

Counting sort nám moc nepomůže, pokud chceme třídit neco jiného než celá čísla, ale i to se dá řešit. Přihrádkové, popřípadě kbelíkové třídění, neboli bucket-sort umí třídit množinu prvků očíslovaných čísly $1 \ldots R$.

Potřebuje k tomu čas $\mathcal{O}(N+R)$ a paměť $\mathcal{O}(N+R)$.

Bucket sort ma ještě jednu příjemnou vlastnost: jedná se o stabilní třídění.

Definice: *Stabilní třídění* je takové, že každé 2 prvky vstupu se stejnou hodnotou klíče jsou na výstupu ve stejném pořadí jako na vstupu.

Algoritmus: (třídění množiny $x_1, x_2, \ldots, x_n$ očíslované $c_1, c_2, \ldots, c_n$ pomocí BUCKET-SORTu)

1. Pro $k \leftarrow 1$ do R opakuj:
2. $\quad p_k \leftarrow 0$
3. Pro $i \leftarrow 1$ do N opakuj:
4. $\quad p_{c_i} \leftarrow p_{c_i} + 1$
5. $b_1 \leftarrow 1$
6. Pro $k \leftarrow 2$ do R opakuj:
7. $\quad b_k \leftarrow b_{k-1} + p_{b-1}$
8. Pro $i \leftarrow 1$ do N opakuj:
9. $\quad t \leftarrow c_i$
10. $\quad v_{b_t} \leftarrow x_i$
11. $\quad b_t \leftarrow b_t + 1$
12. Vrať v

Lexikografické třídění k-tic

Mějme n uspořádaných k-tic prvků z množiny $\{1 \ldots R\}^k$. Úkol zní seřadit k-tice slovníkově (lexikograficky). Můžeme použít metodu rozděl a panuj, takže prvky setřídíme nejprve podle první souřadnice k-tic a pak se rekurzivně zavoláme na každou přihrádku a třídíme podle následující souřadnice. Nebo můžeme využít toho, že bucket-sort je stabilní a třídit takto:

Algoritmus: (třídění k-tic $x_1, x_2, \ldots, x_n$ lexikograficky pomocí BUCKET-SORTu)

1. $S \leftarrow \{x_1, x_2, \ldots, x_n\}$
2. Pro $i \leftarrow k$ do 1 opakuj:
3. $\quad S \leftarrow$ bucket-sort S podle i-té souřadnice
4. vysledek S

Pro přehlednost v následujícím pozorování označme $l = k - i + 1$, což přesně odpovídá tomu, v kolikátém průchodu cyklu jsme.

Pozorování: Po l-tém průchodu cyklem jsou prvky uspořádány lexikograficky podle i-té až k-té souřadnice.

Důkaz: Indukcí podle l

Pro $l = 1$ jsou prvky uspořádány podle poslední souřadnice. Po l průchodech již máme prvky setříděny lexikograficky podle i-té až k-té souřadnice a spouštíme $(l+1)$-ní průchod, tj. budeme třídit podle $(i-1)$-ní souřadnice. Protože bucket-sort třídí stabilně, zůstanou prvky se stejnou $(i-1)$-ní souřadnicí vůči sobě seřazeny tak, jak byly seřazeny na vstupu. Z IP tam však byly seřazeny lexikograficky podle i-té až k-té souřadnice. Tudíž po $(l+1)$-ním průchodu jsou prvky seřazeny podle $(i-1)$-ní až k-té souřadnice.

♡

Časová složitost je $\mathcal{O}(k \cdot (n + R))$, což je lineární s délkou vstupu $(k \cdot n)$ pro pevné k a paměťová složitost je $\mathcal{O}(n + R)$.

Třídění čísel $1 \ldots R$ podruhé (Radix sort)

Zvolíme základ Z a čísla zapíšeme v soustavě o základu Z, čímž získáme $(\lfloor \log_z R \rfloor + 1)$-tice, na které spustíme předcházející algoritmus. Díky tomu budeme třídit v čase $\mathcal{O}(\frac{\log R}{\log Z} \cdot (n + Z))$. A jak zvolit vhodně Z?

Pokud bychom zvolili $Z = konstanta$, časová složitost bude $\mathcal{O}(\log R \cdot n)$, což může být $n * \log n$ nebo i víc. Zvolíme-li $Z = n$, dostáváme $\mathcal{O}(\frac{\log R}{\log n} \cdot n)$, což pro $R \leq n^\alpha$ znamená $\mathcal{O}(\alpha \cdot n)$.

Třídění řetězců

Mějme n řetězců $r_1, r_2 \ldots r_n$ dlouhých $l_1, l_2 \ldots l_n$ Označme si $L = \max_{1 \leq i \leq n} l_i$ délku nejdelšího řetězce a R počet znaků abecedy. Problém je, že řetězce nemusí být stejně dlouhé (pokud by byly, lze se na řetězce dívat jako na k-tice, které už třídit umíme). S tím se můžeme pokusit vypořádat doplněním řetězců mezerami tak, aby měly všechny stejnou délku, a spustit na něj algoritmus pro k-tice. Tím dostaneme algoritmus, který bude mít časovou složitost $\mathcal{O}(L * n)$, což bohužel může být až kvadratické vzhledem k velikosti vstupu. Příklad: na vstupu mějme k řetězců, kde prvních $k - 1$ z nich bude mít délku 1 a poslední řetězec bude dlouhý přesně k. Vstup má tedy celkovou délku $2k - 1$ a my teď doplníme prvních $k - 1$ řetězců mezerami. Vidíme, že algoritmus teď bude pracovat v čase $\mathcal{O}(k^2)$. To, co nám nejvíce způsobovalo problémy u předchozího příkladu, bylo velké množství času zabraného porovnáváním doplněných mezer. Zkusíme proto řešit náš problem trochu chytřeji a koncové mezery do řetězů vůbec přidávat nebudeme. Nejprve roztřídíme bucketsortem řetězce do přihrádek (množin) P_i podle jejich délek, kde i značí délku řetězců v dané přihrádce, neboli definujme $P_i = \{r_j | l_j = i\}$. Dále si zavedeme seznam setříděných řetězců S takový, že v něm po k-tém průchodu třídícím cyklem budou řetězce s délkou alespoň $L - k + 1$ (označme l) a zároveň v něm tyto řetězce budou setříděny lexikograficky od l-tého znaku po L-tý. Z definice tohoto seznamu je patrné, že po L krocích třídícího cyklu bude tento seznam obsahovat všechny řetězce a tyto řetězce v něm budou lexikograficky seřazeny. Zbývá už jen popsat, jak tento cyklus pracuje. Nejprve vezme l-tou množinu P_l a její řetězce roztřídí do přihrádek Q_j (kde index j značí j-tý znak abecedy) podle jejich l-tého (neboli posledního) znaku. Dále vezme seznam S a jeho řetězce přidá opět podle jejich l-tého znaku do stejných přihrádek Q_j za již dříve přidané řetězce z P_l. Na závěr postupně projde všechny přihrádky Q_j a řetězce v nich přesune do seznamu S. Protože řetězce z přihrádek Q_j bude brát ve stejném pořadí, v jakém do nich byly umístěny, a protože ze seznamu S, který je setřízený podle $(l+1)$-ního znaku po L-tý, bude také brát řetězce postupně, bude seznam S po k-tém průchodu přesně takový, jaký jsme chtěli (indukcí bychom dokázali, že cyklus pracuje skutečně správně). Zároveň z popisu algoritmu je jasné, že během třídění každý znak každého řetězce použijeme právě jednou, tudíž algoritmus bude lineární s délkou vstupu (pro úplnost dodejme, že popsaný algoritmus funguje v případech, kdy abeceda má pevnou velikost).

Algoritmus: (třídění řetězců)

1. $L \leftarrow \max(l_1, l_2, \ldots, l_n)$
2. Pro $i \leftarrow 1$ do L opakuj:
3. $\quad P_i \leftarrow \emptyset$
4. Pro $i \leftarrow 1$ do n opakuj:
5. $\quad$ *pridej* (P_{l_i}, r_i)
6. $S \leftarrow \emptyset$
7. Pro $i \leftarrow L$ do 1 opakuj:
8. $\quad$ Pro $j \leftarrow 1$ do R opakuj:
9. $\qquad Q_j \leftarrow \emptyset$
10. $\quad$ Pro $j \leftarrow 1$ do velikost(P_i) opakuj:
11. $\qquad$ *vezmi*(P_i, r)
12. $\qquad$ *pridej* $(Q_{r[i]}, r)$
13. $\quad$ Pro $j \leftarrow 1$ do velikost(S) opakuj:
14. $\qquad$ *vezmi*(S, r)
15. $\qquad$ *pridej* $(Q_{r[i]}, r)$
16. $\quad S \leftarrow \emptyset$
17. $\quad$ Pro $j \leftarrow 1$ do R opakuj:
18. $\qquad$ Pro $k \leftarrow 1$ do velikost(Q_j) opakuj:
19. $\qquad\quad$ *vezmi*(Q_j, r)
20. $\qquad\quad$ *pridej* (S, r)
21. výsledek S

Časová složitost tohoto algoritmu tedy bude $\mathcal{O}(RN)$, kde N je délka vstupu a R počet znaků abecedy.

Sorting Zoo

	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
$\mathcal{O}(N^2)$	*Buble*		
$\mathcal{O}(N \log N)$	*Heap*	*Quick*	*Merge*
$\mathcal{O}(N)$			*Bucket*

K čemu je vlastně třídění dobré?

Díky němu můžeme rychle vyhledávat prvky v množině, konkrétně v čase $\mathcal{O}(\log(n))$ např. pomocí půlení intervalů. Dalším problémem, na který se hodí použít třídění, je zjištění, zda se v posloupnosti nějaké její hodnoty opakují. Dá se dokázat, že tuto úlohu nelze vyřešit lépe (rychleji), než tak, že hodnoty nejprve setřídíme a poté setřiděnou posloupnost projdeme.

Ukázali jsme si tedy, že v se setřízené množině jsme schopni vyhledávat prvky v čase $\mathcal{O}(\log(n))$. Bohužel do ní v čase $\mathcal{O}(\log(n))$ neumíme prvky přidávat. Jedním ze způsobu, jak zrychlit přidávání nových prvků a zároveň nepřijít o logaritmické vyhledávání, je použít další datouvou strukturu, vyhledávací stromy.

Vyhledávací stromy

Problém: chceme udržovat nějaká data setřízená. To znamená, že chceme udržovat množinu X prvků z lineárně uspořádaného universa U s operacemi Find (najdi prvek), Insert (vlož prvek), Delete (smaž prvek). K tomu se nám bude hodit reprezentace dat pomocí stromu.

Definice: Binární strom je zakořeněný strom; každý vrchol má max. 2 syny, pravého a levého.

Dále budeme používat toto značení: v ... vrchol $l(v)$ ... levý syn, $p(v)$ ... pravý syn $L(v)$ ... levý podstrom, $p(v)$ ... pravý podstrom $T(v)$ ... podstrom s kořenem v.

Definice: Binární vyhledávací strom je takový binární strom, že každý vrchol obsahuje prvek z U a pro $\forall v : \forall x \in L(v) : x < v, \forall x \in P(v) : x > v$.

Find($T(v)$,x) Procházíme strom od kořene, na každém vrcholu se podle hodnoty x rozhodneme jak budeme pokračovat. Buď se hodnota vrcholu rovná x, pak končíme (nalezli jsme hledaný prvek). V opačném případě pokračujeme v hledání v levém podstromu (x je menší než hodnota vrcholu), nebo v pravém podstromu (x je větší než hodnota vrcholu). Pokud dojdeme do listu a prvek jsme nenašli, znamená to že tento prvek ve stromě není. Toto nám zabere $\mathcal{O}$(Hloubka stromu).

Insert($T(v)$,x) Nejprve najdeme místo kam je potřeba prvek vložit, k tomu použijeme operaci Find. Pokud Find najde x neprovedeme nic (máme strom s jedinečnými hodnotami). Jinak vložíme prvek na místo, kde by měl být (jako syna vrcholu ve kterém skončil Find). Nejsložitější je operace Find, která trvá $\mathcal{O}$(Hloubka stromu).

Min($T(v)$) Minimum se nachází v nejlevějším listu stromu. Jdeme tedy z kořene pořád doleva, dokud nenarazíme na list. Časová složitost je $\mathcal{O}$(hloubka stromu).

Delete($T(v)$,x) Opět použijeme Find, zjistíme zda prvek vůbec existuje a kde se nachází. Pokud existuje můžou nastat tyto případy:

> x je list: Prostě ho smažeme. x má jednoho syna: Zrušíme x a syna připojíme na uzel, který ukazoval na x. x má dva syny: Najdeme minimum $P(v)$, toto minimum přehodíme s x a x smažeme.

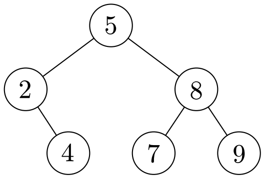
Čas je tentokrát maličko složitější, Find nám trvá $\mathcal{O}$(Hloubka stromu) a v posledním případě nalezení Min($P(v)$) trvá taky až $\mathcal{O}$(Hloubka stromu), což nám dohromady dává pěkných $\mathcal{O}$(Hloubka stromu).

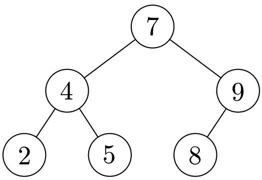
Strom se může používáním operací "kazit", např. opakováním Insert na max. hodnotu což vede až k hloubce $\sim n$. Můžeme

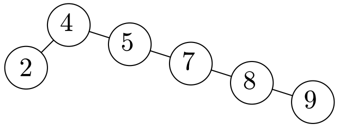
vykašlat se na to → potřeba dokázat průměrný případ vylepšit Insert a Delete → vyvažování stromů

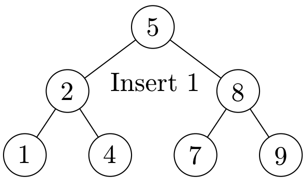
7. Vyhledávací stromy *zapsali M. Řezáč, Š. Masojídek, B. Urbancová*

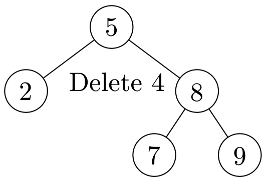
Pár obrázků, které by stály za přesun do předchozí kapitoly:

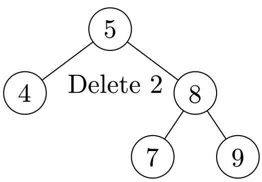


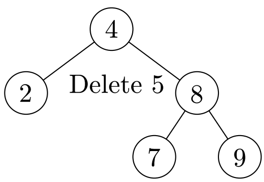












Vyvážené binární vyhledávací stromy

V minulé kapitole jsme se zabývali problematikou přidávání a ubírání prvků binárního vyhledávacího stromu a jeho složitostí a zjistili, že vše záleží na hloubce stromu. Víme, že chceme hloubku logaritmickou, ale jak ji můžeme udržet při operacích? Řešením je následující definice:

Definice: *Dokonale vyvážení* je takové vyvážení, ve kterém pro všechny vrcholy v platí $||L(v)| - |P(v)|| \leq 1$.

Toto nám jistě zajišťuje logaritmickou hloubku, ale je velmi pracné na udržování. Zkusíme proto slabší podmínku:

Definice: *Hloubkové vyvážení* je takové vyvážení, ve kterém pro všechny vrcholy v platí $|h(L(v)) - h(P(v))| \leq 1$. Stromům s hloubkovým vyvážením se říká *AVL stromy* a platí o nich následující lemma.

Lemma: AVL strom o n vrcholech má hloubku $\mathcal{O}(\log n)$.

Důkaz: Uvažme a_k = minimální počet vrcholů AVL stromu o hloubce k. Lehce spočteme:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_2 &= 2 \\ &\vdots \\ a_k &= 1 + a_{k-1} + a_{k-2}. \end{aligned}$$

Rekurentní vzorec jsme dostali rekurzivním stavěním stromu hloubky k: nový kořen a 2 podstromy o hloubce $k-1$ a $k-2$.

Indukcí dokážeme, že $a_k \geq 2^{\frac{k}{2}}$. První indukční krok jsme si už ukázali, teď pro $k \geq 2$ platí: $a_k = 1 + a_{k-1} + a_{k-2} > 2^{\frac{k-1}{2}} + 2^{\frac{k-2}{2}} = 2^{\frac{k}{2}} \cdot (2^{-\frac{1}{2}} + 2^{-1}) \cong 2^{\frac{k}{2}} \cdot 1.21 > 2^{\frac{k}{2}}$

Tímto jsme dokázali, že na každé hladině je minimálně exponenciálně vrcholů, což nám zaručuje hloubku $\mathcal{O}(\log n)$ ♡

Operace s AVL stromy

Find se neliší od operace find v binárních stromech.

Důraz klademe na operace *Insert* a *Delete*, protože při nich musíme ošetřit udržení struktury AVL stromů.

První nutnou podmínkou je, že si musíme *pamatovat stav* v každém vrcholu tohoto stromu. A to *vyvážení* hloubky jeho podstromů.

Umluvíme se např. na tomto označení:

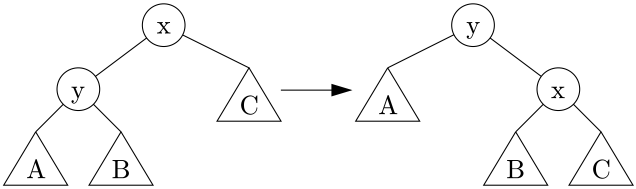
Dostaneme tři typy vrcholů, které se mohou v AVL stromu vyskytnout:

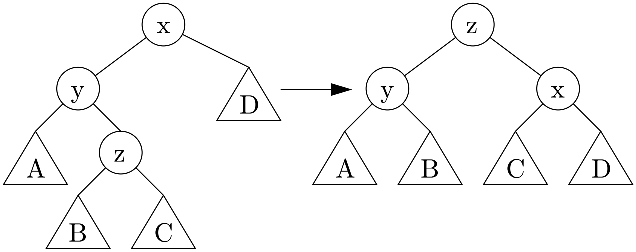
- *Vrchol typu* $\oplus$, pokud je pravý podstrom hlubší
- *Vrchol typu* $\ominus$, pokud je levý podstrom hlubší a
- *Vrchol typu* $\odot$ *(nulou)*, který má oba syny schodné hloubky.

Sestavení AVL stromu:

Postupujeme po struktuře binárního stromu od listů ke kořeni a kontrolujeme, zda jsou vrcholy v jednom ze tří uvedených stavů. Pokud ne, opravíme ho operací jménem rotace.

Rotace





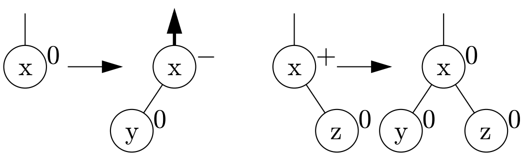
Jde o převrácení hrany mezi původním otcem (kořenem podstromu) a nevyváženým vrcholem tak, aby byli i po přeskupení synové vzhledem k otcům správně uspořádáni.

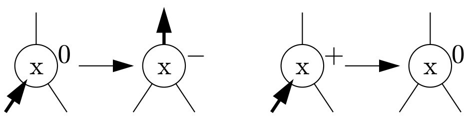
Insert - vložení vrcholu do AVL stromu.

Vložíme jej jako list. Nový list má vždy „znaménko“ nula $\odot$. Předpokládáme, že patří nalevo od posledního otce. Podíváme se na znaménko jeho otce:

- *měl $\odot$ (neměl syna) $\rightarrow$ teď má $\ominus$*, po struktuře stromu nahoru posíláme informaci, že se podstrom prohloubil o 1, což může mít samozřejmě vliv na znaménka vrcholů na cestě ke kořeni.
- *měl $\oplus$ (měl pravého syna, který je listem) $\rightarrow$ teď má $\odot$*, hloubka podstromu se nemění
- *měl $\ominus$* – nenastane, protože v binární struktuře nemohou být dva leví synové

Připadne-li přidaný list napravo, řešíme zrcadlově.



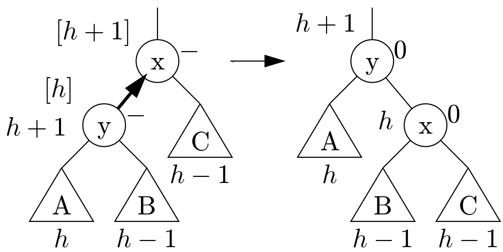


Prohloubil-li se strom vložením nového listu, musíme pracovat s vyvážením:

- Informace o prohloubení přišla zleva *do vrcholu typu* $\odot$ $\rightarrow$ mění jej na vrchol se znaménkem $\ominus$ a informace o prohloubení je třeba poslat o úroveň výš.
- Informace o prohloubení přišla zleva *do vrcholu typu* $\oplus$ $\rightarrow$ mění jej na vrchol se znaménkem $\odot$, hloubka je vyrovnána, dál nic neposíláme.
- Informace o prohloubení přišla zleva *do vrcholu s* $\ominus$ $\rightarrow$

  rozebereme na tři případy podle znaménka vrcholu, ze kterého přišla informace o prohloubení:

  - Informace přišla *z vrcholu typu* $\ominus$ $\rightarrow$ provedeme rotaci doprava tak, že novým kořenem se stane vrchol y, ze kterého přišla informace o prohloubení.

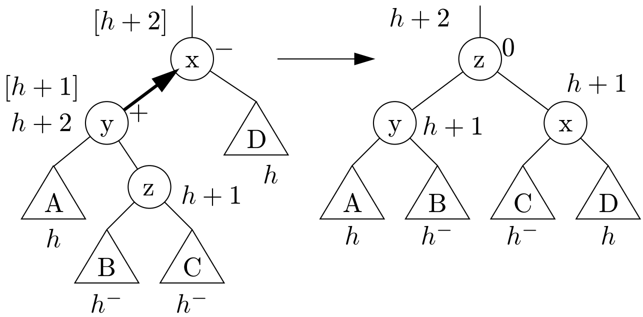


    Pozorování 1: znaménko vrcholů y a x je $\odot$

    Pozorování 2: hloubka před vkládáním byla $h+1$ a nyní je také $h+1$, tedy nemusíme dále posílat informaci o prohloubení a můžeme skončit

  - Informace přišla *z vrcholu typu* $\oplus$

    - uvažme ještě vrchol z jako pravého syna vrcholu y, ze kterého prišla informace o prohloubení, a jeho podstromy B a C
    - vrcholy B a C mají hloubku h nebo $h-1$ $\rightarrow$ označme ji tedy $h-$ (to zřejmě protože vrchol y má znaménko $\oplus$, tedy jeho pravý podstrom s kořenem z má hloubku $h+1$)
    - provedeme dvojrotaci tak, že novým kořenem se stane vrchol z



Pozorování 1: znaménko vrcholu z bude $\odot$
Pozorování 2: znaménka vrcholu x a y se dopočítají v závislosti na hloubce B a C
Pozorování 3: rozdíl hloubky pravého a levého podstromu bude u těchto vrcholů 0 nebo 1
Pozorování 4: hloubka před vkládáním byla $h + 2$ a nyní je také $h+2$, tedy nemusíme dále posílat informaci o prohloubení a můžeme skončit

- informace přišla *z vrcholu typu* $\odot$ – to nemůže nastat, protože v tom případě by nešlo o prohloubení

Delete – odebrání vrcholu z AVL stromu Buď mažeme list nebo mažeme vrchol, který měl nějaké syny.

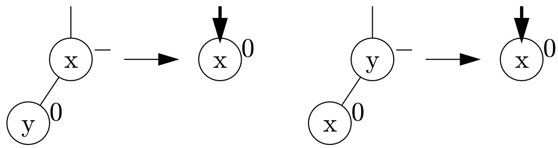
- pokud mažeme list, podíváme se na typ otce. Předpokládáme mazání levého syna.
  - byl typu $\ominus$ (neměl pravého syna) $\rightarrow$ změní se na $\odot$ (vrchol teď nemá žádné syny)
  - byl typu $\odot$ (měl oba syny) $\rightarrow$ změní se na $\oplus$

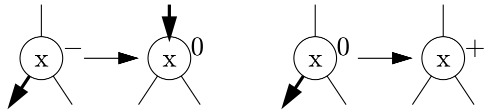
  (mažeme-li pravý list, řešíme zrcadlově)
- mažeme vrchol s jedním (levým nebo pravým) synem $\rightarrow$ syn nastupuje na místo otce a získává typ $\odot$

  V obou případech posílame informaci o změně hloubky stromu…
- mazaný vrchol měl oba syny (listy) $\rightarrow$ vybereme jednoho ze synů na místo smazaného otce. Hloubka se nemění.
- mazaný vrchol měl syny podstromy $\rightarrow$ na jeho místo vezmeme největší prvek levého podstromu (nebo nejmenší prvek pravého podstromu) a od odebraného (nahrazujícího) listu kontrolujeme vyvážení podstromu.

Úprava vyvážení stromu po odebrání listu z podstromu

- informace o změně hloubky přišla z levého podstromu do vrcholu typu $\odot$ $\rightarrow$ vrchol se změní na $\oplus$ a dál se hloubka nemění
- informace přišla zleva do vrcholu s $\ominus$ $\rightarrow$ mění se na $\odot$ a posíláme informaci o změně hloubky.

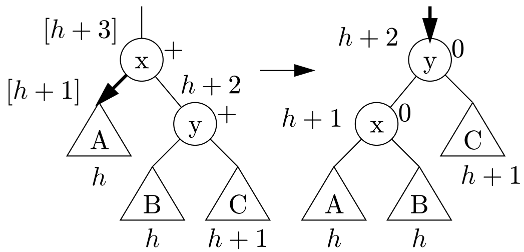




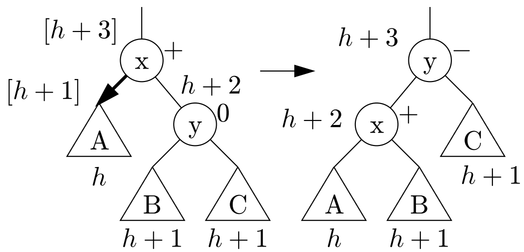
- problémová situace nastává, když informace o změně přišla zleva do vrcholu se znaménkem $\oplus$

Rozebereme na tři případy podle znaménka pravého syna nevyváženého vrcholu

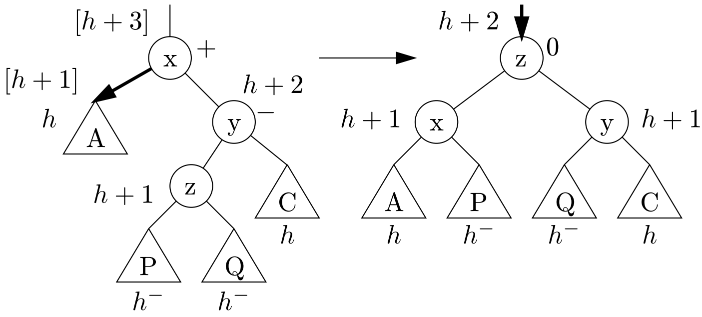
- *pravý syn je typu* $\oplus$ $\rightarrow$ provedeme rotaci vlevo, novým kořenem se stává y (pravý syn), oba vrcholy změní typ na $\odot$ a posíláme informaci o změně hloubky.



- *pravý syn je typu* $\odot$ $\rightarrow$ provedeme opět rotaci vlevo, kořenem se stává y, následně se u y změní typ na $\ominus$, u vrcholu x se typ nemění. Hloubka stromu se nemění, tudíž není třeba posílat informaci.



- *pravý syn je typu* $\ominus$ $\rightarrow$ v tomto případě uvažujeme ještě vrchol z jako levého syna vrcholu y, s podstromy B a C, podstromy B a C mají hloubku h nebo $h-1$. Provedeme dvojrotaci, napřed vpravo rotujeme vrcholy z a y, potom vlevo vrcholy x a z tak, že se z stane novým kořenem, typ vecholu x bude potom $\ominus$ nebo $\odot$, typ y $\oplus$ nebo $\odot$ (podle toho, jaké znaménko měl původně vrchol z), typ z bude $\odot$ a opět posíláme informaci o změně hloubky stromu.



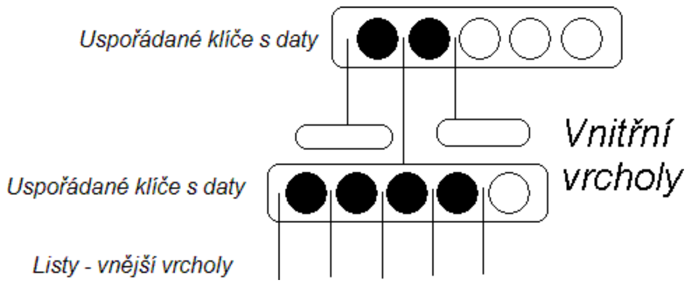
Obecné vyhledávací stromy

Při uložení dat na disku se snažíme, aby se čtení z disku provádělo pokud možno co nejméněkrát a nezáleží nám tolik na tom, kolik operací se vykoná v jednom uzlu. (Časově je operace porovnávání zanedbatelná oproti čtení z disku.)

Definice: (a, b)-*strom* pro parametry a, b, $a \geq 2$, $b \geq 2a - 1$ je zakořeněný strom s uspořádanými syny a vnějšími vrcholy, pro který platí následující axiomy:

1) Data jsou uložena ve vnitřních vrcholech a každý vrchol obsahuje o 1 méně klíčů než má synů.
2) Platí stromové uspořádání, tedy že $A < x_1 < B < x_2 < C < x_3 < D$.
3) Kořen má 2 až b synů, ostatní vnitřní vrcholy a až b synů.
4) Všechny vnější vrcholy jsou ve stejné hloubce (vnější vrchol=list).

Poznámka: kdekoli by mohl být syn a není, připojíme vrchol, kterému říkáme vnější vrchol)



Lemma: (a, b)-strom na n vrcholech má hloubku $O(\log_a n)$.

Důkaz: Zjistíme jeho minimální počet listů (označme jej m): každý vrchol až na kořen má alespoň a synů, hloubku si označíme $d \rightarrow$

$$m \geq \ a^{(d-1)}$$

$$\log_a m \geq d - 1$$

$$d \leq 1 + \log_a m$$

což je řádově $O(\log_a n)$, kde n je počet vrcholů.

Operace s (a,b)-stromy:

Find

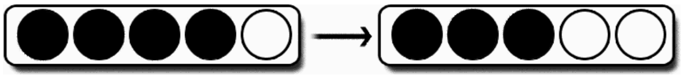
- Vždy zjistíme, mezi které 2 klíče hledaný vrchol patří a potom se zanoříme hlouběji.

Časová složitost nalezení prvku v (a, b)-stromu je $O(\log b \cdot \log_a n)$, kde $\log b$ je čas strávený na jednom vrcholu pro zjištení, mezi které 2 vrcholy hledaný patří, $\log_a n$ je hloubka stromu.

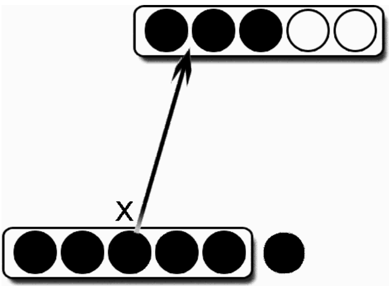
Insert

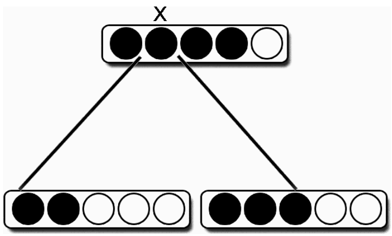
Jako Find, přičemž jestliže nenašel, skončí na posledním patře a přidáme klíč

- pokud přidáním nepřesáhneme maximální počet klíčů můžeme skončit



- pokud přidáním přesáhneme maximální počet klíčů
  1. rozdělíme vrchol na 3 části: L,x,P
  2. L a P jsou nové vrcholy
  3. x je hodnota mezi L a P, kterou vložíme o patro výš jako klíč oddělující nově vzniklé vrcholy L a P
  4. tím jsme převedli problém o patro výš a opakujeme algoritmus





Poznámka: Jestliže se dostaneme až do kořene, rozdělí se kořen na dvě části, vznikne nám nový kořen se dvěma syny (což je povoleno) a celému stromu vzroste hloubka o jedna.

Korektnost: Potřebujeme, aby

$$|L| \geq a - 1$$

$$|P| \geq a - 1$$

po sečtení obou nerovností a pričtení 1 na obě strany rovnice:

$$|L| + |P| + 1 \geq 2a - 2 + 1 = 2a - 1$$

pravá strana je rovna b a to podle definice $\geq 2a-1$.

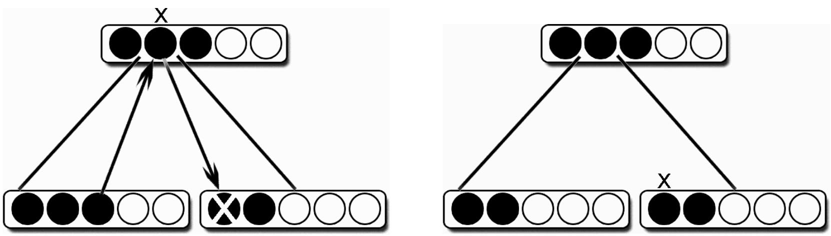
Časová složitost: vkládání prvku do (a,b)-stromu je $O(b \cdot \log_a n)$.

Delete

- převedeme na delete z listu (stejný postup jako u stromu: jestliže to není list, prohodíme tuto hodnotu s nejmenší hodnotou podstromu jeho pravého syna) – v tomto případě na klíč posledního vnitřního vrcholu, protože listy jsou vnější vrcholy bez dat.

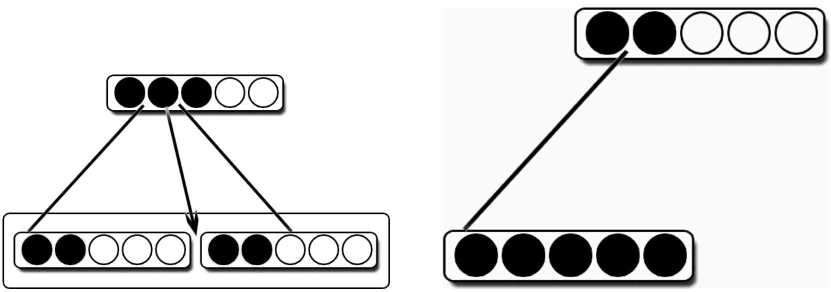
- pokud má vrchol, ze kterého odebíráme stále $a-1$ klíčů, můžeme skončit
- pokud má vrchol(V), ze kterého odebíráme $a-2$ klíčů a jeho levý sousední vrchol(L) alespoň a klíčů (klíč otce oddělující tyto vrcholy označme x):

  1. smažeme největší klíč levého sousedního vrcholu (L) a nahradíme tím klíč otce obou vrcholů (nahradíme x za tuto hodnotu)
  2. původní klíč otce(x) přidáme jako nejmenší klíč odebíranému vrcholu (V)
  3. tím mají oba tyto vrcholy $a-1$ klíčů a můžeme skončit



- pokud má vrchol, z kterého odebíráme(V) $a-2$ klíčů a jeho levý sousední vrchol(L) $a-1$ klíčů (klíč otce oddělující tyto vrcholy označme x):

  1. sloučíme V,x,L do jednoho vrcholu
  2. tím jsme problém převedli o patro výš a opakujeme algoritmus



Poznámka: Dojdeme-li takto až do kořene, na místo klíče odebraného z kořene lze použít nejmenší nebo největší klíč nově sloučeného podstromu. Ten odebrat lze, protože po sloučení (které bylo příčinou této situace), je v nejnižším vrcholu $2a-2$ klíčů.

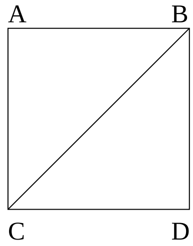
Časová složitost:

$$O(b \cdot \log_a n)$$

8. Základní grafové algoritmy *(Zapsali M. Petr, O.Tichý, Ľ. Magic)*

Reprezentace grafu

Označme vrcholy grafu na následujícím obrázku písmeny A, B, C, D. Pokud bychom chtěli tento graf uchovat v paměti počítače, máme na výběr hned několik způsobů, jak to udělat.



1. seznam hran

Jednoduše vypíšeme seznam všech dvojic vrcholů grafu spojených hranou.

AB, BC, DC, CA, BC

2. matice sousednosti

Matice sousednosti je pole A o velikosti $n \times n$, jehož prvky na souřadnicích i, j jsou dány následujícím předpisem:

$$A_{i,j} = \begin{cases} 0 \Leftrightarrow \{i,j\} \notin E \\ 1 \Leftrightarrow \{i,j\} \in E \end{cases}$$

Náš graf z obrázku výše by tedy v maticové reprezentaci vypadal takto:

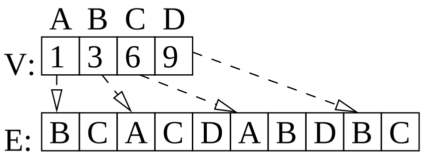
$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

S touto maticí se pracuje velmi snadno (např. všechny sousedy i-tého vrcholu zjistíme jednoduše tak, že projdeme i-tý řádek matice, což představuje časovou složitost $O(n)$), ale má i jednu zřejmou nevýhodu: její velikost je vždy kvadratická bez ohledu na to, jak "řídký" je graf. U grafu s mnoha vrcholy, ale s malým počtem hran, tedy budeme zbytečně plýtvat místem v paměti.

Nabízí se proto třetí, úspornější, způsob, a to:

3. seznam sousedů

V paměti počítače by se dal seznam sousedů uchovávat dvěma poli: polem vrcholů V grafu, jehož prvky postupně pro každý vrchol udávají index na začátek odpovídajícího úseku v poli E, ve kterém by byli uloženi jeho sousedé.



Znázornění polí seznamu sousedů.

Na tuto reprezentaci už stačí prostor $O(n+m)$, což už je, na rozdíl od předchozího kvadratického prostoru, docela příjemné.

Ještě dodejme, že ačkoliv jsme jednotlivé reprezentace ukazovali na příkladě neorientovaného grafu, orientované grafy se dají reprezentovat prakticky stejně. U orientovaných grafů je totiž jediný rozdíl: existuje-li hrana z vrcholu u do vrcholu v, neznamená to, že existuje i hrana z v do u (tedy (u, v) a (v, u) tvoří uspořádané dvojice).

Průchod grafem

Definice: Vrchol je dosažitelný, pokud do něj vede cesta z počátečního vrcholu.

Efektivní průchod by měl splňovat tyto podmínky:

- každou hranu projdu právě jednou (resp. maximálně dvakrát - 1x tam, 1x zpět)
- hranou se vracím zpět, až když už z vrcholu, na kterém momentálně jsem, nevede dosud neprozkoumaná hrana
- pokud jsem došel po hraně do již dříve navštíveného vrcholu, hned se vrátím

Konečnost

Důkaz: Plyne z prvního bodu. ♡

Korektnost (algoritmus projde celý graf)

Důkaz: Sporem: Nechť existuje vrchol, který je dosažitelný, ale algoritmus ho neprošel. Pak jsme ale při průchodu narazili na $\{u, v\} \in E$ takovou, že vrchol u je dosažitelný a w dosažitelný není. Tzn., že $\{u, w\}$ tvoří neprozkoumanou hranu $\Rightarrow$ spor s druhým bodem. ♡

Časová složitost

Z toho, že žádná hrana ani vrchol se neprojde vícekrát (první podmínka efektivního průchodu), plyne časová složitost $O(n+m)$.

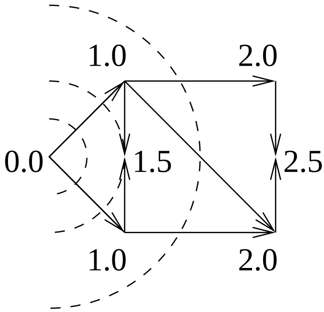
Nyní si něco povíme o dvou způsobech, kterak můžeme procházet grafy.

Průchod do hloubky (Depth First Search, DFS)

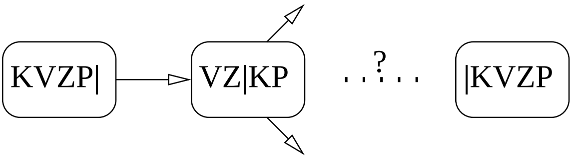
Průchod do hloubky je prvním základním postupem při procházení grafem. Pokud bychom chtěli DFS trochu přiblížit, mohli bychom ho přirovnat k Théseově postupu při průchodu Minotaurovho bludište za pomoci Ariadniny nitě.

Průchod do šířky (Breadth First Search, BFS)

Na rozdíl od prohledávání do hloubky využívá prohledávání do šířky namísto zásobníku frontu. Motivační úlohou by mohlo být např. hledání nejkratší cesty mezi dvěma vrcholy v grafu, nebo známý převozníkův problém.



Př. Koza, vlk a zelí



Řešení ulohy by se dalo popsat jako hledání cesty grafem stavů od počátečního stavu (koza, vlk i zelí jsou na jednom břehu) do koncového (všichni byli bez újmy převezeni na druhý břeh řeky).

DFS na neorientovaném grafu

Při průchodu grafu do hloubky používáme zásobník. Na začátku se v něm nachází pouze vstupní vrchol v. Dokud není zásobník prázdný, tak z něj odebíráme vrchol; ten označíme jako zpracovaný (abychom ho v budoucnu, kdybychom na něj ještě někdy narazili, nezpracovávali znovu) a do zásobníku přidáme všechny jeho sousedy takové, kteří ještě nebyli označeni jako zpracovaní.

Vrcholy značíme těmito barvami

- bílá - dosud nenavštívěné vrcholy (na začátku jsou tedy bílé všechny)
- šedá - označuje právě zpracovávaný vrchol
- černá - už zpracované vrcholy

U každého vrcholu si dále také zapamatujeme čas příchodu a čas odchodu ($in[v]$, $out[v]$).

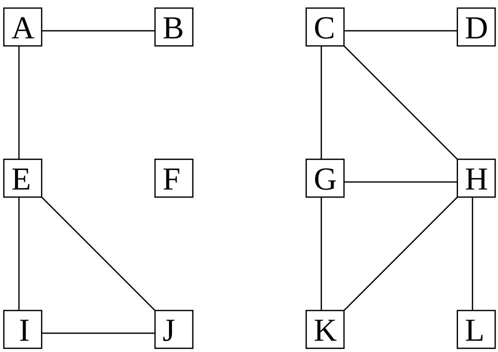
Algoritmus

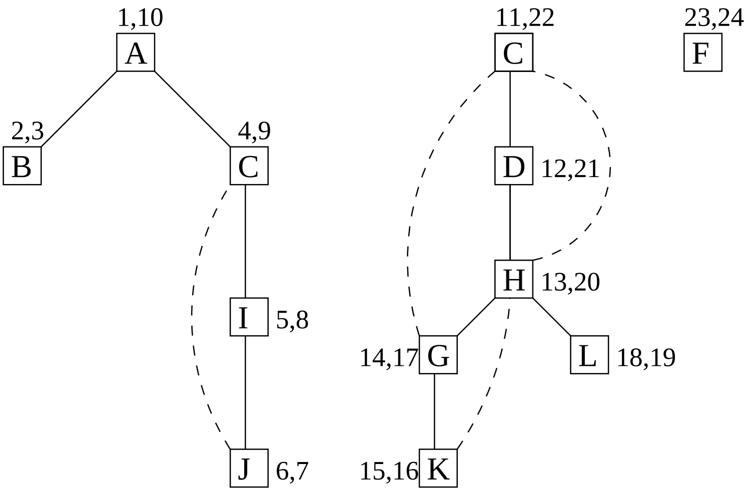
```
procedure Projdi(v)
    color[v] := šedá
    time := time + 1
    in[v] := time
    ∀ w ∈ Sousedi(v):
        if color[w] = bílá ⇒
            Projdi(w)
    color[v] := černá
    time := time + 1
    out[v] := time

procedure DFS()
    ∀ v: color[v] := bílá
    time := 0
    ∀ v ∈ V :
        if color[v] = bílá ⇒
            Projdi(v)
```

Implementaci zásobníku jsme v tomto meta-algoritmu nahradili použitím rekurze (která ale funguje na podobném principu, kdy se na zásobník vkládají jednotlivá volání rekurzivní funkce).

Na následujících obrázcích je znázorněno, jak probíhá průchod grafu pomocí DFS:





Znázornění průběhu algoritmu.

Hrany v tomto znázornění dělíme na:

- stromové - v momentě průchodu vedou do dosud nenavštíveného, bílého, vrcholu
- zpětné - ostatní hrany

Pozorování: plné hrany tvoří strom (DFS strom, strom průchodu do hloubky).

Aplikace:

(1) hledání kostry

(2) hledání komponent souvislosti

Časová složitost těchto aplikací DFS je stejná jako složitost původního meta-algoritmu, tedy $O(n+m)$.

(3) hledání artikulací

Tuto aplikaci si teď rozebereme trošku podrobněji. Nejprve však několik definic.

Definice: *Artikulace* je vrchol v grafu, jehož odebráním se graf rozpadne na alespoň dvě komponenty souvislosti.

Definice: *Most* je hrana v grafu, jejímž odebráním se graf rozpadne na dvě komponenty souvislosti.

Definice: Graf je *2-souvislý,* jestliže po odstranění libovolného vrcholu zůstane souvislý.

Definice: *2-souvislá komponenta* je maximální 2-souvislý podgraf.

Jak bychom tedy při hledání artikulací v grafu mohli postupovat? Jako první se nabízí tento algoritmus:

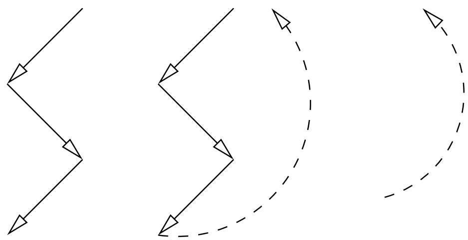
Hloupý algoritmus

Berme si postupně všechny vrcholy grafu; pro každý zjistíme, jestli se jeho odebráním zvýšil počet komponent. K tomu bychom mohli použít průchod do hloubky, který jsme si už ukázali. Procedura Projdi(v) projde a označí všechny vrcholy dosažitelné z v, tedy všechny vrcholy komponenty obsahující vrchol v. Můžeme tedy udělat takovou změnu, že si budeme navštívené vrcholy označovat číslem komponenty.

Takový algoritmus by měl složitost $n \times O(n + m))$ (pro každý vrchol grafu bychom n-krát spouštěli DFS), což se nám vůbec nelíbí. Pokusme se tedy najít něco lepšího. Nejprve učiňme toto pozorování:

Pozorování: v je artikulace $\Leftrightarrow$ v je kořen DFS stromu a má alespoň dva potomky nebo v není kořen a má syna w takového, že ze žádného jeho potomka nevede hrana do předchůdce v.

Definice: Přípustná cesta je taková cesta, pokud vede přes stromové hrany DFS stromu (může končit i skokem zpět).



Příklady přípustných cest.

Dále si definujme funkci low pro každý vrchol v, takto:

low(v) = min(in[w], z v do w vede přípustná cesta)

Pozorování: v je artikulace $\Leftrightarrow$ v je kořen DFS stromu a má alespoň dva potomky nebo má syna w takového, že platí $low(w) \geq in[w]$.

Jak ale hodnotu funkce low(w) spočítat?

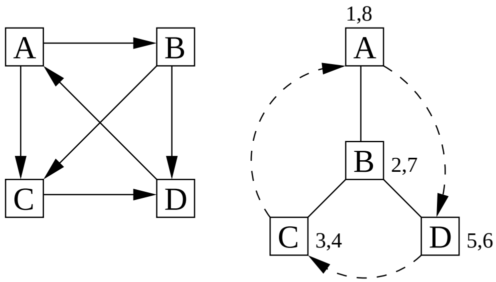
low(v) = min(x, y, z)

x = in[v]

y = min(in[w], $\{w, v\}$ je zpětná hrana)

z = min(low(w), $\{v, w\}$ je stromová hrana)

DFS na orientovaném grafu



Graf a znázornení průbehu:

Hrany v dělíme na:

- stromové $\{A, B\}$, $\{B, C\}$, $\{B, D\}$
- zpětné: vedou do předchůdce v DFS stromě (do šedého vrcholu)
- příčné: vedou do vrcholu v jiném podstromě (do černého vrcholu)
- dopředné: vedou do svého potomka (do černého vrcholu v témže podstromě)

Na závěr několik **pozorování:**

Intervaly (in[v], out[v]) $\forall v \in V$ tvoří dobré uzávorkování.

v je následník u $\Leftrightarrow (in[v], out[v]) \subseteq (in[u], out[u])$

$\{u, v\}$ je zpětná hrana $\Leftrightarrow (in[u], out[u]) \leq (in[v], out[v])$

$\{u, v\}$ je příčná hrana $\Leftrightarrow in[v] < out[v] < in[u] < out[u]$

9. Průchod grafem *(zapsali T. Hubík, V. Kolomičenko, L. Slinták)*

1. do hloubky (DFS)

in(v), *out(v)*

Hrany rozdělíme do nekolika tříd

- *stromové* postupují do neoznačených vrcholů
- *zpětné* vedou do už označených vrcholů
- *dopředné* vedou do vrcholů, odkud již jsme se vrátili
- *příčné* pouze zprava doleva

Pozn.: u neorientovaných grafů mohou být pouze stromové a zpětné hrany

Složitost $\mathcal{O}(m+n)$, n … počet vrcholů, m … počet hran

2. do šířky (BFS)

Složitost $\mathcal{O}(m+n)$

Definice: *Lineární uspořádání* $<$ vrcholů grafu G je topologické $\iff \forall(u,v) \in E(G) : u < v$

Pozorování: Pro cyklické grafy topologické uspořádání neexistuje.

Věta: Libovolný acyklický orientovaný graf (DAG) má topologické uspořádání a lze ho sestrojit v $\mathcal{O}(m+n)$.

Důkaz: Spustíme DFS, $u < v$ právě tehdy, když *out(u)* $>$ *out(v)*

hrany:

- stromová - *out(u)* $>$ *out(v)*
- zpětná - existuje cyklus a to je spor
- dopředná - *out(u)* $>$ *out(v)*
- příčná - *out(u)* $>$ *in(u)* $>$ *out(v)*

Důsledek: Orientovaný graf G je cyklický $\iff$ DFS najde zpětnou hranu, existuje zpětná hrana $\Rightarrow$ existuje cyklus, neexistuje zpětná hrana $\Rightarrow$ existuje topologické uspořádání $\Rightarrow$ neexistuje cyklus. ♡

Nejdelší cesta v DAGU:

(z S do každého vrcholu)

1. Zvolíme topologické uspořádání na G, BÚNO S je minimum.
2. $\forall v : D(v) = -inf; D(S) = 0$
3. Postupně procházíme vrcholy $v \in V(G)$ v topologickém pořadí a pro $\forall v$ spočítáme $D(v)$
4. $D(v) = max(D(n)) + 1$

$u : (u,v) \in E(G)$

Pozorování: Každý DAG obsahuje *zdroj* (vycházejí z něj cesty) a *stok* (cesty do něj vstupují)

Rozkládání orientovaných grafů na komponenty souvislosti

Definice: R bude relace na $V(G)$ tak, že $uRv \iff$ existuje orientovaná cesta v G z u do v a opačne

Definice: G je silně souvislý $\iff \forall(u,v) \in V(G) : uRv$

Pozorování: R je ekvivalence, $u...v$ $v...w$ je u-w sled $\Rightarrow$ existuje u-v cesta

Definice: Komponenty silné souvislosti grafu $G \iff$ ekvivalence třídy relace R

Lemma: $\forall(G)$ a $\forall(u,v) \in V(G)$: existuje u-v sled $\iff$ existuje u-v cesta

Důkaz: Buď S u-v sled, na kterém se opakuje vrchol z. Uvažme sled $u...z$ $z...v$, což je také u-v sled, ale je kratší. Iterujeme $\Rightarrow$ zbude cesta. ♡

Definice: Graf komponent $C(G)$

$V(C(G))$: SSK grafu G

$$(c_1, c_2) \in E(C(G)) \iff \exists v_1 \in c_1, v_2 \in c_2 : (v_1, v_2) \in E(G)$$

Lemma: $\forall(G) : C(G)$ je DAG.

Důkaz: obrázkem a sporem, existuje cyklus $c_1, c_2, ..., c_k$ v $C(G)$. ♡

Algoritmus hladání silně souvislých komponent grafu v O(n)

Pozorování 1: DFS z vrcholu v projde $\{w, \exists v\text{-}w \text{ cesta}\}$.

Pozorování 2: Je-li C stoková komponenta a $v \in V(C)$, pak DFS z v projde právě C.

Pozorování 3: Spustíme-li DFS na celý G, pak vchol v s maximální *out(v)* leží ve zdrojové SSK.

Pozorování 3:* Nechť C_1, C_2 jsou SSK a $C_1, C_2 \in E(C(G)$, pak $max\{out(u), u \in V(C_1)\} > max\{out(v), v \in V(C_2)\}$.

Pozorování 4: G a G^R mají stejné SSK (G^R má proti G otočené hrany).

Algoritmus:

1. Sestrojíme G^R $\quad \mathcal{O}(m+n)$
2. DFS na $G^R \rightarrow out(v)$ $\quad \mathcal{O}(m+n)$
3. T bude uspořádání vrcholů podle klesajících *out(v)* $\quad \mathcal{O}(n)$
4. $\forall v \in V(G)$ v pořadí podle T $\quad \mathcal{O}(m+n)$
5. pokud v ještě nepatří do žádné komponenty, tak
6. spustím DFS z v a dosažené vrcholy prohlásím za další SSK $\mathcal{O}(m+n)$

Věta: Každý orientovaný graf lze rozložit na SSK v čase $\mathcal{O}(m+n)$

Důkaz: Pozorovani 3*:

- 1. DFS vstoupí nedříve do C_1 : Z C_2 vyleze určitě dřív než z $C_1 \Rightarrow out(C_1) > out(C_2)$.
- 2. nejdříve do C_2: Nejdřív se vrátím z celé C_2, až pak někdy zase vlezu do $C_1 \Rightarrow out(C_1) > out(C_2)$.

10. Nejkratší cesty *(zapsali L. Banáková, O. Hoferek, J. Břečka)*

Definice: *Vzdálenost vrcholů* $d(u, v)$ je délka nejkratší cesty (u, v), pokud existuje, nebo $+\infty$, pokud taková cesta neexistuje.

Problém: Je dán graf G a funkce $l : E(G) \rightarrow \mathrm{R}$ přiřazující hranám jejich délky. Dále je dán startovací a cílový vrchol $s, c \in V(G)$. Chceme najít cestu $s = v_1, v_2, \ldots, v_k = c$ takovou, aby délka cesty

$$l(v_1, v_2) + l(v_2, v_3) + \ldots + l(v_{k-1}, v_k)$$

byla minimální. Takovéto cestě budeme říkat *nejkratší cesta* z s do c.

Bez újmy na obecnosti budeme předpokládat, že graf G je orientovaný, bez smyček a násobných hran.

Nejprve uvážíme případy, kdy jsou všechny hrany stejně dlouhé. Tehdy můžeme použít algoritmus prohledávání do šířky (BFS). K jeho implementaci je zapotřebí fronta Q s metodami $Put(Q, v)$ (přidání prvku v na konec fronty) a $Get(Q)$ (odebrání prvku ze začátku fronty). Algoritmus vrátí pole „značek“ z: $z(v) = 1$ právě tehdy pokus je vrchol v dostupný ze startovacího vrcholu s, jinak $z(v) = 0$.

Algoritmus: (Prohledávání do šířky (BFS))

$z(*) \leftarrow 0, z(s) \leftarrow 1$
$Q \leftarrow \{s\}$
while $Q \neq \emptyset$:
$\quad v \leftarrow Get(Q)$
$\quad$ for $\forall w : (v, w) \in E$
$\quad\quad$ if $z(w) = 0$ then
$\quad\quad\quad z(w) \leftarrow 1$
$\quad\quad\quad Put(Q, w)$

Lemma Č (časové): BFS doběhne v čase $\mathcal{O}(m + n)$.

Důkaz: Každý vrchol se do fronty dostane maximálně jednou a každou hranu zpracujeme maximálně jednou. ♡

Lemma D (dosažitelnost): BFS označí právě vrcholy dostupné ze s.

Důkaz:

„$\Rightarrow$“: Indukcí podle běhu algoritmu.

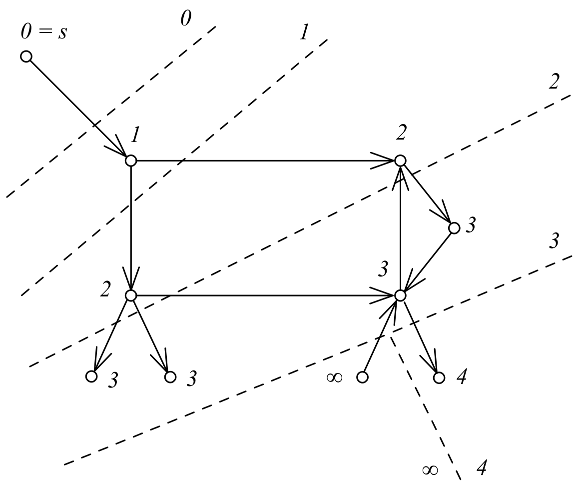
„$\Leftarrow$“: Dokážeme sporem. Uvažme, že existuje vrchol dostupný, ale algoritmem neoznačený. Vezměme takovýto „špatný“ vrchol v, který je s nejblíže. Uvažme nejkratší cestu (s, v): $s, \ldots, u, v$. Předchozí vrchol na této cestě u musí být „dobrý“. Vrchol u se dostane do fronty, pak je z ní vybrán a tím se zpracuje i vrchol $v \Rightarrow$ spor.

Předchozím algoritmem jsme pouze zjistili, které vrcholy jsou z s dosažitelné. Nyní si tento algoritmus modifikujeme tak, abychom našli $d(s, v)$ pro všechna $v \in V(G)$.

Rozšířený algoritmus: (Prohledávání do šířky (BFS))

$z(*) \leftarrow 0, z(s) \leftarrow 1$
$D(*) \leftarrow +\infty, D(s) \leftarrow 0$
$Q \leftarrow \{s\}$
while $Q \neq \emptyset$:
$\quad v \leftarrow Get(Q)$
$\quad$ for $\forall w : (v, w) \in E$
$\quad\quad$ if $z(w) = 0$ then
$\quad\quad\quad z(w) \leftarrow 1$
$\quad\quad\quad D(w) \leftarrow D(v) + 1$
$\quad\quad\quad Put(Q, w)$

Průběh tohoto algoritmu si můžeme ukázat na následujícím obrázku („prasečí graf“):



Prasečí graf

Definice: *Vrstva* $L_i \subseteq V$: $L_0 = \{s\}$, $L_{i+1} = \{w : w$ označen při procházení vrcholu $v \in L_i\}$

Lemma V (vzdálenosti): Na konci BFS je $\forall v : D(v) = d(s, v)$.

Důkaz:

a) pro nedosažitelné vrcholy: OK (algoritmus je neoznačí $\Rightarrow D(v) = \infty$)

b) pro dosažitelné vrcholy: Víme, že pro vrchol $v \in L_i$ platí $D(v) = i$, stačí tedy zjistit, zda $\forall i : \forall v \in L_i$ je $d(s, v) = i$. To ověříme matematickou indukcí podle i:

pro $i = 0$: evidentní,

pro $i > 0$: Při procházení vrstvy L_{i-1} musí být vrchol v takový, že $d(s, v) = i$, označen a zároveň nemohl být označen dříve a tudíž patří do L_i. ♡

Všimněme si, že v grafu existují pouze tři typy hran vzhledem k našemu algoritmu:

- dopředu o jednu vrstvu
- dozadu o jednu nebo víc vrstev
- ve vrstvě

Abychom našli i samotné cesty, nejen jejich délky, modifikujeme svůj algoritmus tak, že si pro každý vrchol v, který označíme, budeme pamatovat jeho předchůdce $p(v)$:

1. $z(*) \leftarrow 0, z(s) \leftarrow 1$
2. $D(*) \leftarrow +\infty, D(s) \leftarrow 0$
3. $p(*) \leftarrow ?$
4. $Q \leftarrow \{s\}$
5. while $Q \neq \emptyset$:
6. $\quad v \leftarrow Get(Q)$

    for $\forall w : (v, w) \in E$
        if $z(w) = 0$ then
            $z(w) \leftarrow 1$
            $D(w) \leftarrow D(v) + 1$
            $p(w) \leftarrow v$
            $Put(Q, w)$

Pozorování: Pokud $p(v) \neq\ ? \Rightarrow D(v) = D(p(v)) + 1$.

Lemma S (stromové): Graf (W, F), kde W = dosažitelná část V, $F = \{(v, p(v))\}$ je strom orientovaný do kořene s. Navíc ukazuje nejkratší cestu pro každý dosažitelný vrchol (pro každý vrchol v tomto grafu, existuje jediná orientovaná cesta do kořene s, která je zároveň nejkratší cesta z s do tohoto vrcholu).

Nyní uvažme, že délky hran l nejsou všechny stejně dlouhé:

Ukážeme si trik pro ohodnocené hrany: hranu délky l podrozdělíme na l hran délky 1 a pak spustíme BFS. Algoritmus poběží s časovou složitostí $O((m+n)l_{max})$. To není zrovna moc výhodné, a proto si ukážeme tzv. *Budíkovou taktiku*:

Místo toho, abychom každou hranu podrozdělovali, si budeme pro každý vrchol pamatovat „čas budíku“ b. Časem $b(v)$ budíku vrcholu v rozumíme čas, kdy se do tohoto vrcholu vlna dostane, když se bude šířit po dosud známých hranách. Na začátku nastavíme $\forall v \in V : b(v) = l(s, v)$ a pokud při běhu algoritmu objevíme hranu, která nám umožní dostat se do v rychleji, tak hodnotu $b(v)$ změníme. Algoritmus pak bude pracovat tak, že bude spát, dokud nezazvoní nějaký budík (což znamená, že jsme dorazili do nějakého vrcholu).

Tato taktika nás přivede k Dijkstrovu algoritmu, kde se ze dvoustavového $z(v)$ stane třístavové $z(v)) \in \{\mathbf{N}, \mathbf{V}, \mathbf{H}\}$, kde **N** znamená neviděn, **V** viděn a **H** hotov. Podle hodnoty $z(v)$ se pak $D(v)$ rovná $+\infty$ pro $z(v) = \mathbf{N}$, času budíku pro $z(v) = \mathbf{V}$ nebo vzdálenosti $d(s, v)$ pro $z(v) = \mathbf{H}$.

Algoritmus: (Dijkstrův algoritmus)

$z(*) \leftarrow \mathbf{N}, z(s) \leftarrow \mathbf{V}$
$D(*) \leftarrow +\infty, D(s) \leftarrow 0$
$p(*) \leftarrow ?$
while $\exists v : z(v) = \mathbf{V}$:
    $v \leftarrow$ vrchol, pro nějž $z(v) = \mathbf{V}$ a zároveň $D(v)$ je minimální
    $z(v) \leftarrow \mathbf{H}$
    for $\forall w : (v, w) \in E$:
        if $D(w) > D(v) + l(v, w)$ then
            $D(w) \leftarrow D(v) + l(v, w)$
            $z(w) \leftarrow \mathbf{V}$
            $p(w) \leftarrow v$

Věta: Pokud $\forall v, w : l(v, w) \in \mathrm{Z}^+ \cup \{\infty\}$, pak Dijkstrův algoritmus najde nejkratší cesty v čase $\mathcal{O}(n^2)$.

Důkaz: V průběhu algoritmu se maximálně n-krát hledá vrchol v s minimálním $D(v)$, maximálně m-krát se přenastavuje $D(v)$. Pokud bude struktura D uložena jako pole (hledání minima v $\mathcal{O}(n)$, změna hodnoty $\mathcal{O}(1)$), bude celková časová složitost Dijkstrova algoritmu $\mathcal{O}(n^2+m) = \mathcal{O}(n^2)$. Fakt, že se algoritmus zastaví je evidentní, protože se stav žádného vrcholu nevrací z **H** do **V**.

Úplný důkaz správnosti a časové složitosti i pro hrany ohodnocené reálnými čísly bude na příští přednášce. ♡

Nahlédněme nyní, zda by vedlo k vylepšení časové složitosti, kdybychom při implementaci Dijkstrova algoritmu použili jako datovou strukturu pro ukládání viděných vrcholů různé typy hald. Prováděli bychom s nimi operace *DeleteMin* (maximálně n-krát) a *Decrease* (maximálně m-krát). Zmíněné operace mají pro různé typy hald následující časové složitosti:

	binární halda	*k-regulární halda*	*Fibonacciho halda*
DeleteMin	$\mathcal{O}(\log n)$	$\mathcal{O}(k \log_k n)$	$\mathcal{O}(\log n)$
Decrease	$\mathcal{O}(\log n)$	$\mathcal{O}(\log_k n)$	$\mathcal{O}(1)$
celková složitost	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(m \log_{m/n} n)$	$\mathcal{O}(n \log n + m)$

Použijeme-li klasickou haldu, dostaneme celkovou časovou složitost $\mathcal{O}((m + n)\log n)$, což znamená, že jsme si pomohli v případech, kdy je graf „řídký“. U k-regulární haldy se celková časová složitost rovná $\mathcal{O}(nk \log_k n + m \log_k n)$, ideálně pro $k = m/n$: $\mathcal{O}(m \log_{m/n} n)$. Prozkoumejme, jak se nám v tomto případě bude měnit časová složitost pro různě „husté“ grafy:

- $m \approx n \rightarrow \mathcal{O}((m+n)\log n)$
- $m \approx n^2 \rightarrow \mathcal{O}(m)$
- $m \approx n^{1+\epsilon} \rightarrow \mathcal{O}(m) \ldots (k = m/n = n^{\epsilon})$

Při použití Fibonacciho haldy je pak celková časová složitost rovna $\mathcal{O}(n \log n + m)$.

Nyní si předvedeme trochu jiný algoritmus na prohledávání grafu. Mějme matici sousednosti A ke grafu $G = (V, E)$: $A_{ij} = 1 \Leftrightarrow (v_i, v_j) \in E$, jinak $A_{ij} = 0$. Potom $A^2_{ij} = \sum_{k=1}^{n} A_{ik}A_{kj}$. Tento součet je nenulový právě tehdy, když vrcholy i,k,j tvoří sled délky 2. Jinými slovy A^2_{ij} se rovná počtu sledů délky 2 v G z i do j. Obdobně A^k_{ij} = počet sledů délky k. Pokud k matici A přičteme jednotkovou matici E_n (čímž si přimyslíme smyčky), dostaneme po umocnění:

$(A + \mathrm{E}_n)^k_{ij} \neq 0 \Leftrightarrow \exists\ i, j$-sled délky maximálně k.

Výstupem algoritmu tedy bude matice $(A + \mathrm{E}_n)^n$. Můžeme si také všimnout, že pokud matici $A + \mathrm{E}_n$ umocníme na číslo větší než n, tak výsledek bude pořád správný, jen bude výpočet trvat déle. Takže algoritmus může pracovat v iteracích umocňováním matice na druhou: $M_0 = A + \mathrm{E}_n$, $M_1 = M_0^2$, $M_{i+1} = M_i^2$. Výstupní matice bude tudíž:

$M_{\lceil \log n \rceil (ij)} \neq 0 \Leftrightarrow \exists$ cesta z i do j

Na výpočet dosažitelnosti vrcholů v grafu tedy stačí $\mathcal{O}(\log n)$ násobení matic, a pokud na toto použijeme Strassenův algoritmus, dostaneme ve výsledku algoritmus se složitostí $\mathcal{O}(n^{\log_2 7} \log n)$.

11. Nekratší cesty podruhé *(zapsali Dušan Renát a Radek Tupec)*

Na této přednášce budeme studovat problém hledání nejkratších cest v grafech ohodnocených reálnými čísly.

Situace: Máme orientovany graf G a funkce $l : E(G) \to \mathrm{R}$ přiřazující hranám jejich ohodnocení (délky). Pro vrcholy $u, v \in V(G)$ budeme chtít spočítat jejich vzdálenost $d(u, v)$, což bude délka nejkratší cesty z u do v nebo ∞, pokud žádná cesta neexistuje.

Aby se vzdálenosti chovaly „rozumně“ (tj. jako metrika), budeme chtít, aby platily následující vlastnosti:

- $d(u, u) = 0$,
- $d(u, v) \leq d(u, w) + d(w, v)$ (trojúhelníková nerovnost).

To nemusí obecně platit (kazí nám to záporné cykly), proto budeme studovat pouze grafy, v nichž záporné cykly neexistují. Pak už budou obě vlastnosti splněny, jak plyne například z následujícího lemmatu:

Lemma: V grafu bez záporných cyklů existuje ke každému nejkratšímu sledu z u do v stejně dlouhá uv-cesta.

Důkaz: Máme-li nejkratší uv-sled, který není cestou, opakuje se v něm nějaký vrchol $w \in V(G)$. Délka cyklu $l(c) \geq 0 \Rightarrow l(u \ldots v \ldots w) \leq l(původnísled)$. Tento postup můžeme opakovat a po konečném počtu kroků dostaneme cestu, tedy platí trojúhelníková nerovnost. ♡

Opakování: Dijkstrův algoritmus

- $D(v)$ ... dočasná vzdálenost z s do v
- Značky $Z(v)$
  - :Neviděn
  - :Viděn
  - :Hotov

1. $D(*) \leftarrow +\infty, D(s) \leftarrow 0$
2. $Z(*) \leftarrow Neviden, Z(s) \leftarrow V$
3. while $\exists v : Z(v) = V$
4.   vybereme $v : Z(v) = V, D(v) = min$
5. $Z(v) = H$
6.   for $\forall w : (v, w) \in E(G)$
7. $D(w) \leftarrow min(D(w), D(w) + l(v, w))$
8.   if $Z(w) = N \Rightarrow Z(w) \leftarrow V$

Věta: Pokud G je nezáporně ohodnocený graf, pak se Dijkstrův algoritmus zastaví a vydá $\forall v : D(v) = d(s, v)$ (tedy správné hodnoty).

Důkaz: Následující posloupností lemmat. ♡

Lemma 1: Pokud $v_0, \ldots, v_k$ je nejkratší $v_0 v_k$-cesta, pak $v_0, \ldots, v_{k-1}$ je nejkratší $v_0 v_{k-1}$-cesta.

Důkaz: Pokud by tomu tak nebylo, můžeme $v_0, \ldots, v_{k-1}$ vyměnit za kratší cestu, a tím získat kratší $v_0 v_k$-cestu. ♡

Lemma 2: Algoritmus se zastaví po $\leq n$ průchodech cyklem.

Důkaz: Zřejmé z toho, že každý vrchol uzavřeme nejvýše jednou. ♡

Lemma 3: Po zastavení jsou hotovy právě vrcholy dosažitelné z s.

Důkaz: Viz minulá přednáška. ♡

Lemma 4: $D(v)$ uzavíraných vrcholů tvoří neklesající posloupnost.

Důkaz: V okamžiku, kdy uzavíráme v platí $\forall w \notin H : D(w) \geq D(v)$, případně přepočítáme $D(w)$ na $D(v) - l(v, w) \geq D(v)$. ♡

Lemma 5: Pokud $v \in H$, pak $D(v)$ se už nezmění.

Důkaz: Indukcí podle běhu algoritmu. ♡

Lemma 6: Pro $\forall v D(v)$ je délka nejkratší sv-cesty, jejíž vnitřní vrcholy leží všechny v H.

Důkaz:

- po 1. průchodu OK
- uzavíráme-li další vrchol v:

  a) $D(w)$ pro $w \in H$ Podle Lemma 5 se $D(w)$ nemění. Musíme nahlédnout, že se opravdu změnit nemá.
  b) $D(w)$ pro $w \notin H$ $D(w) = min D(v) + l(v, w)$

♡

Existuje pomalejší algoritmus pro grafy se zápornými hranami bez záporných cyklů. **Bellman-Fordův algoritmus**

1. $D(*) \leftarrow \infty, D(s) \leftarrow 0$
2. Opakuji
3. Pro $\forall v \in V$
4. *prozkoumej*(v)
5. (koukne se, jestli nejde vylepšit cestu
6. do sousedů v)
7. dokud se nějaké $D(\ldots)$ mění

Lemma 1: Pokud $D(v) < \infty$, pak existuje sled z s do v délky $D(v)$. (speciálně z toho plyne $\forall v D(v) \geq d(s, v)$)

Lemma 2: $\forall v D(v)$ nikdy neroste. **Lemma 3:** Po k fázích je $\forall v D(v) \leq$ délka nejkratšího sv-sledu o $\leq k$ hranách. *Důkaz:* Indukcí…pro $k \geq 0$ OK Indukční krok: Doběhla $k - 1$ fáze, pouštíme k-tou

♡

Věta: Pro graf bez záporných cyklů se Bellman-Fordův algoritmus zastaví po nejvýše m fázích a vydá $D(v) = d(s, v)$ pro všechna v. (zřejmé z lemmat)

Lemma 4: Pokud v grafu existuje záporný cyklus dosažitelný z s, algoritmus se nezastaví. (Zajímavý test na to, zda graf obsahuje záporný cyklus.)

Časová složitost Bellman-Fordova algoritmu : $O(n * m)$

Floyd-Warshallův algoritmus G je graf se záporným ohodnocením hran, bez záporných cyklů $D^k_{i,j}$ = délka nejkratší cesty z v_i do v_j přes $v_1 \dots v_k$ $D^0_{ij} = l(v_i, v_j)$, $D^0_{ii} = 0$ $D^n_{ij} = d(v_i, v_j)$ - skutečná vzdálenost v grafu

1. $D_{i,j} \to l(v_i, v_j), D_{i,i} \to 0 \forall i, j$
2. for $k = 1$ to n
3.   for $i = 1$ to n
4.   for $j = 1$ to n
5. $D_{i,j} = min(D_{i,j}, D_{i,k} + D_{k,j})$

Věta: Floyd-Warshallův algoritmus spočítá $D_{i,j} = d(v_i, v_j)$ v čase $O(n^3)$.

Shrnutí:

$s \to *$ Dijkstrův algoritmus $O(n^2)$ ... s haldou $O((n+m) \log m)$ ... s regulární haldou $O(n + m * logn)$ ... s Fibbonaciho haldou $O(n * logn + m)$ Bellman-Ford $O(n * m)$

$* \to *$ Floyd-Warshall $O(n^3)$

12. Problém minimální kostry *(zapsali K. Kaščák a M. Vachna)*

Zadání úlohy: Pro neorientovaný graf G s ohodnocením hran *váhami* $w : E(G) \to \mathrm{R}$, chceme najít kostru T s minimálním ohodnocením $w(T) = \sum_{e \in E(T)} w(e)$.

Navíc předpokládáme: (bez újmy na obecnosti)

- Graf G je souvislý (jinak ho nejprve rozložíme na komponenty).
- $\forall e, f \in E(G) : e \neq f \Rightarrow w(e) \neq w(f)$.

Nyní si ukážeme tři algoritmy pro hledání minimální kostry, konkrétně se jedná o Jarníkův, Borůvkův a Kruskalův algoritmus.

Jarníkův algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. Zvolíme libovolný vrchol $v_0 \in V(G)$.
3. $T \leftarrow (\{v_0\}, \emptyset)$ (zatím jednovrcholový strom)
4. Dokud $|V(T)| \neq n$:
5.   Vybereme hranu $uv \in E(G) : u \in V(T), v \notin V(T)$ tak, aby $w(uv)$ byla minimálni.
6. $T \leftarrow T + uv$.
7. *Výstup:* Minimální kostra T.

Věta: Jarníkův algoritmus se zastaví po maximálně n iteracích a vydá minimální kostru grafu G.

Důkaz: Při každé iteraci algoritmus přidá jeden vrchol do T, a proto se po maximálně n iteracích zastaví. Vydaný graf je strom, protože se stále přidává list k již existujícímu stromu, a jelikož má n vrcholů, je to kostra. Zbývá nám už jen dokázat, že nalezená kostra je minimální. K tomu pomůže následující lemma:

> **Definice:** *Řez* v grafu $G = (V, E)$ je množina hran $F \subseteq E$ taková, že $\exists U \subset V : F = \{uv \in E : u \in U, v \notin U\}$.
>
> **Lemma:** Pokud G je graf, w jeho prosté ohodnocení, F je řez v grafu G a f je nejlehčí hrana v řezu F, pak pro každou minimální kostru T grafu G je $f \in E(T)$.
>
> *Důkaz:* Buď T kostra a $f = uv \notin E(T)$. Pak existuje cesta $P \subseteq T$ spojující u a v. Cesta musí řez alespoň jednou překročit. Proto existuje $e \in P \cap F$ a navíc víme, že $w(e) > w(f)$. Uvažme $T' = T - e + f$. Tento graf je rovněž kostra grafu G, protože odebraním hrany e se graf rozpadne na dvě komponenty a přidáním hrany f se tyto komponenty opět spojí. Navíc $w(T') = w(T) - w(e) + w(f) < w(T)$. ♡

V důkazu korektnosti Jarníkova algoritmu toto lemma využijeme tak, že si všimneme, že hrany mezi vrcholy stromu T a zbytkem grafu tvoří řez a algoritmus nejlehčí hranu tohoto řezu přidá do T. Podle lemmatu tedy všechny hrany T musí být součástí každé minimální kostry a jelikož T je strom, musí být minimální kostrou.

♡

Důsledky: Graf G s prostým ohodnocením má pravě jednu minimální kostru. Minimální kostra je jednoznačně určená lineárním uspořádáním hran.

Implementace:

- Přímočará: pamatujeme si, které vrcholy a hrany jsou v kostře T a které ne. Časová složitost je $\mathcal{O}(nm)$.
- Chytřejší: Pro $v \notin V(T)$ si pamatujeme $D(v) = \min\{w(uv) : u \in T\}$. Při každém průchodu hlavním cyklem pak procházíme všechna $D(v)$ (to vždy trvá $\mathcal{O}(n)$) a při přidání vrcholu do T kontrolujeme okolní $D(w)$ pro $vw \in E$ a případně je snižujeme (za každou hranu $\mathcal{O}(1)$). Časovou složitost tím celkově zlepšíme na $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$.
- Také se dá použít halda pro uchovávání hran nebo hodnot $D(v)$.

Borůvkův algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. $F \leftarrow (V(G), \emptyset)$
3. Dokud F má alespoň dvě komponenty:
4. Pro každou komponentu T_i grafu F vybereme nejlehčí incidentní hranu t_i.

5. Všechny hrany t_i přidáme do F.
6. *Výstup:* Minimální kostra F.

Věta: Borůvkův algoritmus se zastaví po $\lceil \log_2 n \rceil$ iteracích a vydá minimální kostru grafu G.

Důkaz: Všimněme si nejprve, že po k iteracích mají všechny komponenty grafu F minimálně 2^k vrcholů (na počátku jsou všechny komponenty jednovrcholové, v každé další iteraci se komponenty slučují do větších, každá s alespoň jednou sousední, takže se velikosti komponent minimálně zdvojnásobí). Proto nejpozději po $\lceil \log_2 n \rceil$ iteracích už velikost komponenty dosáhne počtu všech vrcholů a algoritmus se zastaví.

Hrany mezi každou komponentou a zbytkem grafu tvoří řez, takže podle předchozího lemmatu všechny hrany přidané do F musí být součástí (jednoznačně určené) minimální kostry. Graf $F \subseteq G$ je tedy vždy les a až se algoritmus zastaví, bude roven minimální kostře. ♡

Implementace:

- Inicializace přímočará.
- Pomocí DFS rozložíme les na komponenty. Pro každý vrchol si pamatujeme číslo komponenty.
- Pro každou hranu zjistíme, do které komponenty patří, a pro každou komponentu si uchováme nejlehčí hranu.

Takto dokážeme každou iteraci provést v čase $\mathcal{O}(m)$ a celý algoritmus doběhne v $\mathcal{O}(m \log n)$.

Kruskalův neboli hladový algoritmus

Algoritmus:

1. *Vstup:* Graf G s ohodnocením w.
2. Setřídíme všechny hrany z $E(G)$ tak, aby: $w(e_1) < ... < w(e_m)$.
3. $F \leftarrow (V(G), \emptyset)$.
4. Pro $i = 1$ do m:
5. Pokud $F + e_i$ je acyklický, provedeme $F \leftarrow F + e_i$.
6. *Výstup:* Minimální kostra F.

Věta: Kruskalův algoritmus se zastaví po m iteracích a vydá minimální kostru.

Důkaz: Každá iterace algoritmu zpracovává jednu hranu, takže iterací je m. Indukcí dokážeme, že F je vždy podgrafem minimální kostry: prázdné počáteční F je podgrafem čehokoliv, každá hrana, kterou pak přidáme, je minimální v řezu oddělujícím nějakou komponentu F od zbytku grafu (ostatní hrany tohoto řezu ještě nebyly zpracovány, a tudíž jsou těžší). Naopak žádná hrana, kterou jsme se rozhodli do F nepřidat, nemůže být součástí minimální kostry, jelikož s hranami, o kterých již víme, že v minimální kostře leží, tvoří kružnici. ♡

Implementace:

- Setřídění v čase $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$.
- Pak potřebujeme udržovat komponenty souvislosti grafu F, abychom uměli rychle určit, jestli právě zpracovávaná hrana vytvoří kružnici. Potřebujeme tedy strukturu pro udržování komponent souvislosti, které se m-krát zeptáme, zda dva vrcholy leží v téže komponentě (tomu budeme říkat operace *Find*), a $(n-1)$-krát spojíme dvě komponenty do jedné (*Union*).

Kruskalův algoritmus tedy poběží v čase $\mathcal{O}(m \log n + mT_f + nT_u)$, kde T_u je čas na provedení jedné operace *Union* a T_f na operaci *Find*.

Jednoduchá struktura pro komponenty: Budeme pamatovat v poli čísla komponent, ve kterých leží jednotlivé vrcholy. *Find* zvládneme v čase $\mathcal{O}(1)$, ale *Union* bude stát $\mathcal{O}(n)$. Celý algoritmus pak poběží v čase $\mathcal{O}(m \log n + m + n^2) = \mathcal{O}(m \log n + n^2)$.

Chytřejší struktura: Každou komponentou si uložíme jako strom orientovaný směrem ke kořeni – každý vrchol si pamatuje svého otce, navíc každý kořen si pamatuje velikost komponenty. Operace *Find* vystoupá z obou vrcholů ke kořeni a kořeny porovná. *Union* rovněž najde kořeny a připojí kořen menší komponenty pod kořen té větší. Obojí zvládneme v čase lineárním v hloubce stromu a jak si ukážeme, tato hloubka je vždy nejvýše logaritmická, a proto celý Kruskalův algoritmus poběží v čase $\mathcal{O}(m \log n + m \log n + n \log n) = \mathcal{O}(m \log n)$.⟨6⟩

Lemma: Strom hloubky h má alespoň 2^h prvků.

Důkaz: Indukcí: Pokud *Union* spojí strom s hloubkou h s jiným s hloubkou menší než h, pak hloubka výsledného stromu zůstává h. Pokud spojuje dva stromy stejné hloubky h, pak má výsledný strom hloubku $h + 1$. Z indukčního předpokladu víme, že strom hloubky h má minimálně 2^h vrcholů, a tedy výsledný strom hloubky $h + 1$ má alespoň 2^{h+1} vrcholů. ♡

Červeno-černé stromy

Ve zbytku přednášky se vrátíme k vyhledávacím stromům a ukážeme si ještě jeden způsob, jak udržet logaritmickou hloubku stromu.

Definice: *Červeno-černý strom* je binární vyhledávací strom, jehož vrcholy jsou obarvené (některé červeně, zbylé černě), a ke kterému byly přidány externí vrcholy všude tam, kde původnímu vrcholu chybí některý ze synů. Listy č-č stromu jsou tedy vždy externí. Navíc platí následující axiomy:

Axiomy:

- Kořen a externí vrcholy jsou černé.
- Otec červeného vrcholu je černý.
- Na všech cestách z kořene do externího vrcholu je stejný počet černých vrcholů.

Věta: Červeno-černý strom o n vrcholech má hloubku $\mathcal{O}(\log n)$.

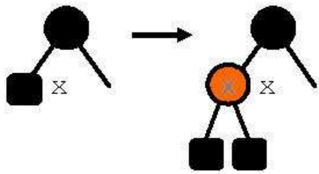
⟨6⟩ Drobnou úpravou bychom mohli dosáhnout daleko efektivnější struktury, ale tu bychom neupotřebili, jelikož by nás stejně brzdilo třídění, a analýza složitosti by byla … inu, složitější.

Důkaz: Zkomprimujeme všechny hrany spojující černý a červený vrchol, čímž všechny červené vrcholy „zasuneme“ do jejich černých otců. Vznikne $2,4$-strom a ten, jak už víme, má hloubku $\mathcal{O}(\log n)$. Původní č-č strom má tedy hloubku nejvýše $2\log n$, což je asymptoticky taky $\mathcal{O}(\log n)$. ♡

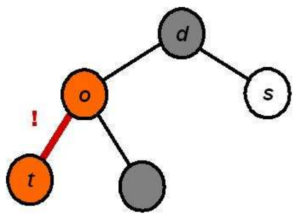
Alternativní důkaz: ⟨7⟩ Předpokládejme, že T je červeno-černý strom, který má na cestě z kořene do listu k černých vrcholů. Pak pro počet vrcholů stromu T platí $2^k - 1 \leq n \leq 2^{2k} - 1$. Nejmenší takový strom má všechny vrcholy obarvené černě a je to úplný pravidelný binární strom o hloubce $k-1$, což dává dolní odhad. Největší takový strom má všechny vrcholy v sudých hladinách obarveny červeně a v lichých hladinách černě, je to úplný pravidelný binární strom o hloubce $2k-1$ a tím je dán horní odhad. Tedy $k \leq 1 + \log n$. Z vlastností červeno-černých stromů plyne, že $k \leq \mathit{hloubka}(T) \leq 2k$. ♡

Nyní si rozebereme vkládaní do červeno-černých stromů.

Insert: Vložíme hodnotu do stromu jako do standardního binárního vyhledávacího stromu, tedy jako list, a obarvíme červeně. První ani třetí axiom jsme neporušili a pokud je předek černý, tak ani axiom druhý, čili jsme hotovi:



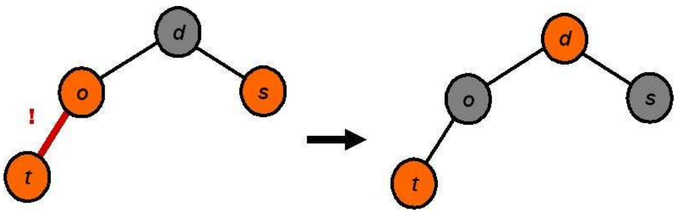
Jestliže je otec o našeho vrcholu t také červený, budeme říkat, že došlo k *poruše* a budeme se ji snažit opravit. Situace musí vypadat následovně:



Nyní rozlišíme následující tři případy:

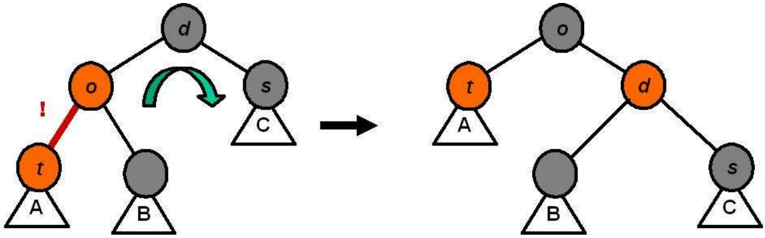
- Pokud strýc s vrcholu t je červený, pak pouze přebarvíme o, d a s podle následujícího obrázku. Nyní d může být porucha, ovšem posunutá o 2 hladiny výše, takže při nejhorším pokračujeme do kořene a poslední poruchu opravíme přebarvením kořene na černo.

⟨7⟩ „Vy máte protipříklad? Nevadí, já mám dva důkazy :)“

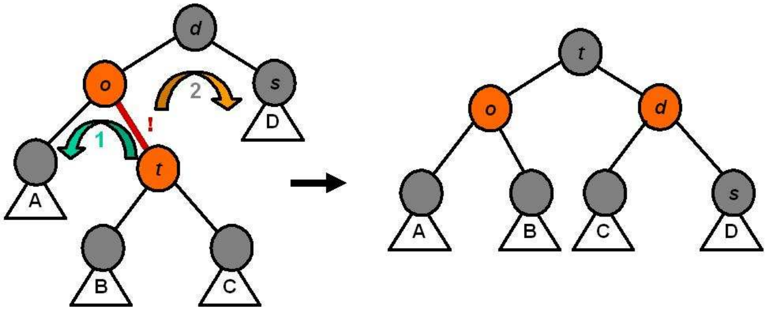


- Strýc s je černý. Pak záleží na tom, zda hodnota t leží mezi hodnotami o a d nebo ne. Jinými slovy, zda cesta $t - o - d$ obsahuje zatáčku.

  a) Bez zatáčky: Provedeme rotaci a přebarvíme. Splněny budou axiomy 1, 2 i 3, tedy máme červenočerný strom:



  b) Se zatáčkou: Provedeme dvojitou (LR) rotaci a přebarvíme. Opět získáme korektní červenočerný strom:



13. Hashování

(zapsali Martin Franců a Bohdan Maslowski)

V předchozích kapitolách jsme ukázali, že ačkoliv v obecném případě nelze třídit rychleji než v čase $\Theta(n \log n)$, pro některé speciální druhy hodnot (malá celá čísla, řetězce apod.) existuje efektivnější lineární algoritmus. Nyní si podobným způsobem posvítíme na reprezentaci množin – navrhneme datovou strukturu, která bude umět s vhodnými typy prvků provádět základní množinové operace (*Insert*, *Delete*, *Find*) v *průměrně* konstantním čase (pomocí vyhledávacích stromů jsme to zatím uměli v čase $\mathcal{O}(\log n)$ na operaci).

Jaké máme možnosti? Začněme od těch nejjednodušších a označme si n velikost množiny a U velikost univerza, z nějž prvky množiny vybíráme.

1. Pole indexované od 0 **do** $U-1$

Insert, *Delete* i *Find* pracují v čase $\mathcal{O}(1)$, ale platíme za to paměťovou složitostí $\Theta(U)$.

2. Číslicový strom (radix tree)

Zvolíme základ soustavy b, prvky universa zapisujeme jako k-tice číslic. Čili:

$$x \in U \mapsto [b]^k, k = \lceil \log_b U \rceil.$$

Tím nám vznikne strom s hloubkou k, na k-té úrovni se rozhodujeme podle k-té číslice.

Operace *Find* bude trvat $O(\log_b U)$, operace *Insert* a *Delete* pak $O(b \log_b U)$. Strom zabere prostor velikosti $O(nb \log_b U)$.

Čím bude základ b menší, tím lepší paměťová a horší časová složitost, a obráceně. Číslicový strom tedy vyžaduje pečlivé nastavení parametrů.

3. Písmenkový strom

Používá se pro řezězce, za základ soustavy b zvolíme abecedu.

$|x|$...délka řetězce

Find, Insert, Delete: $O(|x|)$

Paměť: $O(\sum_i |x|)$

4. Hashování

h ... hashovací funkce $[U] \to [m]$

m ... počet přihrádek, v každé přihrádce spojový seznam.

Perfektní hashovací funkce přiřadí do každé z m přihrádek jediný prvek z $[U]$.

Příklad: Jaká je pravděpodobnost, že 2 z k lidí mají narozeniny ve stejný den? Stačí 23 lidí, aby nastala kolize s pravděpodobností větší než jedna polovina. Aby nenastala, potřebovali bychom větší počet přihrádek: $p = cm^2$

Hashování se separovými řetezci

Insert, Delete i Find pracují v čase $O(|x|)$, paměťová složitost je $O(m+n)$

vložili jsme $x_1 \dots x_n \in [U]$

nyní pracujeme s $x \in [U]$

předpokládejme, že $h(x_i)$ a $h(x)$ jsou náhodné

$p := h(x)$

$N_p := \sharp : h(x_1) = h(x)$

$N_{i,p} := 0$ pokud $h(x_i) \neq p$

$N_{i,p} := 1$ pokud $h(x_i) = p$

$N_p = \sum_{i=1}^{n} N_{i,p}$
$EN_{i,p} = Pr[h(x_i) = p] = \frac{1}{m}$
$EN_p = n\frac{1}{m} = \frac{n}{m}$

Za předpokladu, že hash přiděluje náhodnou přihrádku, je v každé přihrádce $\frac{n}{m}$ prvků.

Druhá půlka

VĚTA: *Pokud jsme do hashovací tabulky vložili hodnoty $x_1, \ldots, x_m$ tak, že h_{x_i} jsou všechny stejně pravděpodobné, pak následující Insert, Delete, Find pracují v čase $\mathcal{O}\left(\frac{n}{m}+1\right)$. (Za předpokladu, že výpočet h trvá $\mathcal{O}(1)$.)*

Insert: Spočítá si $h(x)$ a vloží jej do spojového seznamu v patřičné přihrádce.

Delete: Spočítá si $h(x)$ a najde jej v spojovém seznamu v patřičné přihrádce a odstraní jej.

Find: Spočítá si $h(x)$ a najde jej v spojovém seznamu v patřičné přihrádce.

Tato metoda má dva háčky:

1. vysoká koliznost
2. musíme dopředu vědět, kolik prvků budeme vkládat

Nafukovací hashovací tabulka

na počátku: $n = 0; m = 4$

v každém okamžiku budeme udržovat poměr $\frac{n}{m} \in \langle\frac{1}{2}, 1)$

tedy: (pokud nastane)

$$n = 4 \Rightarrow m := 8$$

$$n = 8 \Rightarrow m := 16$$

Při každém zvětšování tabulky musíme zvětšit obor hodnot hashovací funkce a všechny prvky již v tabulce uložené znovu "zhashovat".

Nafukování trvá $\mathcal{O}(n)$, ale mezi dvěma nafouknutími je minimálně $\frac{n}{2}$ Insertů.

Pokud tedy každý Insert přispěje konstantou, tak to máme zadarmo!

PROBLÉM: Jak udělat funkci Delete? Jednoduché zmenšování tabulky při poměru $\frac{n}{m} = \frac{1}{2}$ není dostačující, neboť by se mohlo stát, že bychom nafukovali a zfukovali hned po sobě.

Verze s Insert a Delete

Udržujeme $\frac{n}{m} \in \langle\frac{1}{4}, 1)$

při zfukování tabulku půlíme, při nafukování zdvojnásobujeme

Po přehashování je poměr $\frac{n}{m} = \frac{1}{2}$ $\Rightarrow$mezi dvěma přehashováními nastane minimálně $\frac{m}{2}$ Insertů či $\frac{m}{4}$ Deletů, tedy alespoň $\frac{m}{4}$ operací.

Zároveň zabereme vždy lineárně prostoru vzhledem k n.

VĚTA: *Nafukovací hashování (se separovanými řetězci) za předpokladu rovnoměrné pravděpodobnosti pracuje v průměrném čase $\mathcal{O}(1)$ na operaci a v prostoru $\mathcal{O}(n)$ (kde n je počet přítomných prvků).*

Kde přijít k rovnoměrné hashovací funkci?

Dědečkovo vyprávění:

a) $h(x) = x \; mod(m)$ -rychlá, snadná kolize, sudé x vždy v sudých přihrádkách. Na náhodných číslech funguje v pohodě.

POZOROVÁNÍ: $\alpha \in (0, 1)$ iracionální;

$x \in N$ pak $\alpha x \; mod(1)$ se chová "náhodně".

b) $h(x) = \lfloor m(\alpha x \; mod(1)) \rfloor$

$$\frac{A}{W} \sim \alpha \Rightarrow h(x) = \lfloor m(Ax \; mod(W)) \rfloor$$

Jako velmi výhodné se jeví používat $\alpha = \frac{\sqrt{5}-1}{2}$ - převrácená hodnota zlatého řezu.

c) funkce k hashování řetězců

$$h(\emptyset) = 0$$
$$h(x_1, \ldots, x_n) = c(h(x_1, \ldots, x_{n-1}) + x_n)$$
$$\Rightarrow h(x_1, \ldots, x_n) = x_n + cx_{n-1} + c^2 x_{n-2} + \ldots$$

(polynom)

d)Pokud x je k-tice (celých) čísel. Pak můžeme jako hashovací funkci použít skalární součín s náhodně vybraným prvkem z vektorového prostoru Z_p^k (nad konečným tělesem):

$$h(x) = \overrightarrow{a} * \overrightarrow{x} mod(p)$$

Pravděpodobnost přes všechny volby $\overrightarrow{a}$, že x, y se zahashují stejně je pak:

$$p[h(x) = h(y)] = p[\overrightarrow{a} * \overrightarrow{x} = \overrightarrow{a} * \overrightarrow{y}] = p[\overrightarrow{a} * (\overrightarrow{x} - \overrightarrow{y}) = 0)]$$

Pokud platí $p[\overrightarrow{a} * (\overrightarrow{x} - \overrightarrow{y} = 0)]$, pak $\overrightarrow{a}$ je prvkem ortogonálního doplňku k $(x - y)$, ten má $dim(W^\perp) = (k - 1)$

$$p = \frac{W^\perp}{Z_p^k} = \frac{p^{k-1}}{p^k} = \frac{1}{p} = \frac{1}{m}$$

kde m je počet přihrádek. Což je dostačující.