Kapitola 1

Časová složitost

... aneb jak dlouho program poběží, než dostanu výsledek.

Velikost vstupu: Jak měřit velikost vstupních dat? Velikost vstupních dat je počet bitů, které jsou potřeba k zápisu vstupu do počítače. Často ale velikost vstupních dat měříme hrubě a uvádíme jen počet čísel na vstupu, počet vrcholů grafu, počet hran grafu apod. Zápis každého čísla, vrcholu či hrany, odpovídá jedné proměnné v počítači a obsahuje jenom konstantní počet bitů. Proto je skutečná velikosti vstupu (počet bitů) jen konstantě krát větší. Později si ukážeme, že nám při výpočtu časové složitosti na multiplikativních konstantách nezáleží.

Na binární zápis čísla n je potřeba $\log n$ bitů. Proto musíme rozlišovat, jaká čísla dostaneme na vstupu. Čísla typu integer mají omezenou velikost 32 bitů (konstantní počet bitů). Pokud na vstupu dostaneme dlouhé číslo n, například s miliónem cifer, tak ho nemůžeme uložit do jedné proměnné typu integer, ale budeme ho muset reprezentovat v poli. Proto bude jeho velikost na vstupu odpovídat počtu políček pole, nebo přesněji počtu bitů, což je $\log n$.

Časová složitost: Časová složitost algoritmu spuštěného na vstup $\mathcal{D}$ je počet kroků, které algoritmus provede. Časová složitost algoritmu je funkce $T : \mathbb{N} \to \mathbb{N}$, kde $T(n)$ je (maximální) počet kroků, které provede algoritmus běžící na datech o velikosti n (pro všechny vstupy $\mathcal{D}$ velikosti n).

Co je to krok algoritmu? Krok algoritmu je jedna operace/instrukce daného stroje/počítače. Je to například přiřazení, aritmetická operace $+$, $-$, $*$, $/$, vyhodnocení podmínky apod. Zjednodušeně budeme za krok algoritmu považovat libovolnou operaci proveditelnou v konstantním čase.

Označme velikost vstupu jako n a nechť c je nějaká konstanta. Časové složitosti $c \cdot n$ budeme říkat lineární, časové složitosti $c \cdot n^2$ kvadratická, $c \cdot n^3$ kubická a $c \cdot a^n$ pro $a > 1$ exponenciální.

Jak můžeme jedno přiřazení považovat za krok algoritmu stejně jako padesát přiřazení? Proč můžeme zjednodušeně říci, že jedno číslo na vstupu má velikost jedna a ne 32, i když zabere 32 bitů? Copak na těchto konstantách v teoretických odhadech nezáleží? Ano je to tak. Na těchto konstantách moc nezáleží a ukážeme si proč (prakticky na nich záleží, ale to patří do praktického porovnávání algoritmů).[1]

Podívejme se do následující tabulky, která ukazuje, jak dlouho budou trvat výpočty algoritmů s různou časovou složitostí. Čísla v tabulce jsou přibližné hodnoty funkcí na vstupech o velikostech 10, 100, 1000 a 1000000.

[1] Ve skutečnosti trvá každá instrukce procesoru jiný čas (jiný počet taktů), a navíc je to na každém typu procesoru jinak. Některé instrukce trvají 1 takt, 2 takty, ale jsou i instrukce trvající 150 taktů. Je to takový guláš, že nám nic jiného než hrubý odhad času nezbude.

	$n = 10$	$n = 100$	$n = 1000$	n=1000000
$\log n$	3.3	6.7	10	20
$\sqrt{n}$	3.2	10	31.6	1000
n	10	100	1000	10^6
$n \log n$	33	664	9966	$20 \cdot 10^6$
n^2	100	10^4	10^6	10^{12}
n^3	1000	10^6	10^9	10^{18}
2^n	1024	$13 \cdot 10^{30}$	$11 \cdot 10^{302}$	$\approx \infty$
$n!$	$36 \cdot 10^5$	$93 \cdot 10^{157}$	$40 \cdot 10^{2567}$	$\approx \infty$

Běžný počítač zvládne spočítat 10^9 operací za vteřinu. Následující tabulka udává, jak dlouho budou trvat výpočty algoritmů na běžném počítači.

	$n = 10$	$n = 100$	$n = 1000$	n=1000000
$\log n$	$3.3ns$	$6.7ns$	$10ns$	$20ns$
$\sqrt{n}$	$3.2ns$	$10ns$	$31.6ns$	$1\mu s$
n	$10ns$	$100ns$	$1\mu s$	$1ms$
$n \log n$	$33ns$	$664ns$	$9.9\mu s$	$20ms$
n^2	$100ns$	$10\mu s$	$1ms$	$16,5min$
n^3	$1\mu s$	$1ms$	$1s$	$31let$
2^n	$1\mu s$	$3 \cdot 10^{14} let$	$3 \cdot 10^{286} let$	$\approx \infty$
$n!$	$3ms$	$3 \cdot 10^{142} let$	$\approx \infty$	$\approx \infty$

Co můžeme z tabulky vypozorovat? Z tabulky je vidět, že skoro všechny výpočty až na ty s časovou složitostí 2^n a $n!$, budou trvat rozumný čas. Proto budeme považovat algoritmy s polynomiální časovou složitostí za rozumné a těm s exponenciální časovou složitostí se budeme snažit vyhnout. Dále je vidět, že pro zpracování velkých dat jsou algoritmy s časovou složitostí menší než $c \cdot n \log n$ výrazně lepší než ostatní polynomiální algoritmy s vyšším stupněm.

To, jestli má program rozumnou časovou složitost, nerozhoduje jen o tom, jestli odpověď dostaneme hned a nebo jestli budeme muset chvilku čekat. Je otázkou, jestli se výsledku vůbec dožijeme. Vždyť i naše planeta Země existuje teprve $4,5$ miliardy let. To je $4,5 \cdot 10^9$ let což je zhruba jen $14 \cdot 10^{16}$ vteřin.[2]

Z tabulky je také vidět, že například funkce n^2, $5n^2$ a $30n^2$ porostou skoro stejně rychle. Na dostatečně velkých vstupech je snadno odlišíme od lineárních funkcí, ale i od kubických až exponenciálních funkcí. Proto není důležitá ta konstanta před funkcí, ale řád n^2, se kterým funkce roste. Z toho důvodu nebudeme při posuzování algoritmů brát ohled na konstanty. To nás přivádí k pojmu asymptotická časová složitost. (Prakticky záleží i na těchto konstantách. Přeci jen je rozdíl, jestli bude algoritmus počítat rok a nebo 10 let, případně 2 hodiny a nebo 1 den.)

Poznámka: Kdy se vyplatí koupit si nový počítač? O kolik větší data/vstupy budeme moci zpracovávat? Předpokládejme, že nový počítač bude dvakrát rychlejší než ten, co máme, a že používáme aplikaci, která musí nejpozději do minuty vydat odpověď. Pokud odpověď počítáme podle algoritmu s lineární časovou složitostí, tak stihneme spočítat dvakrát větší vstup. Naproti tomu pokud má algoritmus exponenciální časovou složitost, tak budeme rádi, když spočítáme výsledek pro o jedna větší vstup. A možná ani to ne.

Poznámka (Moorův zákon): Moorův zákon popisuje zajímavý trend v historii počítačového hardwaru. Říká, že se rychlost nových počítačů přibližně za každé

[2] Vzpomeňte si na Stopařova průvodce po galaxii. V této knize si myši nechaly postavit super počítač – planetu Zemi, aby jim odpověděl na základní otázku života, vesmíru a tak vůbec. Jejich výpočet byl o proti algoritmům s exponenciální časovou složitostí spuštěných na vstupu velikosti tisíc docela efektivní.

dva roky zdvojnásobí.[3] Mohli byste proto namítat, že můžeme výpočty zrychlovat tím, že si počkáme na výkonnější počítače. Jak jsme si ale ukázali v předchozí poznámce, tak nám některých případech dvojnásobné zrychlení výpočtu moc nepomůže. Proto je lepší se zaměřit na vývoj efektivnějších algoritmů.

Také se proslýchá, že se v brzké době Moorův zákon zastaví. Tranzistory na procesoru už jsou tak malé, že už se blíží k velikosti atomů.

Poznámka: Některé časové složitosti vůbec nemusí být rostoucí funkce. Klidně mohou oscilovat. Podívejme se například na algoritmus, který zjišťuje, zda je číslo n prvočíslo. Postupně bude procházet čísla 2 až $\sqrt{n}$ a testovat, zda dané číslo dělí n. Pokud uspěje, tak máme dělitele, můžeme skončit a odpovědět, že číslo n není prvočíslo. Jinak projdeme všechny možné dělitele a nakonec odpovíme, že n je prvočíslo.[4] Pro každé sudé n algoritmus skončí hned v prvním kroku. Na druhou stranu pro každé n prvočíslo algoritmus projde všech $\sqrt{n}$ čísel. Proto časová složitost osciluje mezi funkcemi 1 a $\sqrt{n}$.

1.1 Asymptotická časová složitost

Při určování asymptotické časové složitosti nás zajímá chování algoritmů na hodně velkých datech. Vezměte si papír, hodně velký papír, a nakreslete na něj grafy všech funkcí představujících časovou složitost, které vás napadnou. Při pohledu zblízka uvidíme velký rozdíl mezi funkcemi $\log n$, n, 2^n, ale i mezi n, $5n$ a $50n$. Asymptotická časová složitost se na tento papír dívá z velké dálky, třeba až z Marsu (musíme mít hodně velký papír). Při jejím pohledu všechny funkce, které se liší jen multiplikativní konstantou, „splynou“ v jednu „rozmazanou funkci“. Z takové dálky zůstane vidět jen propastný rozdíl mezi funkcemi $\log n$, n, 2^n (viz tabulka na straně 1 zachycující, jak rychle rostou některé funkce).

Chceme zavést značení, které bude říkat, že funkce lišící se pouze multiplikativní konstantou, patří do stejné třídy.

Definice: Nechť f a g jsou dvě funkce z přirozených čísel do přirozených čísel. Řekneme, že

$f(n) = \mathcal{O}(g(n))$ (čte se „f je velké O od funkce g“) právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

$f(n) = \Omega(g(n))$ (čte se „f je omega od funkce g“) právě tehdy, když existuje konstanta $c > 0$ a n_0 takové, že pro každé $n \geq n_0$ platí $f(n) \geq c \cdot g(n)$.

$f(n) = \Theta(g(n))$ (čte se „f je theta od funkce g“) právě tehdy, když zároveň $f(n) = \mathcal{O}(g(n))$ i $f(n) = \Omega(g(n))$.

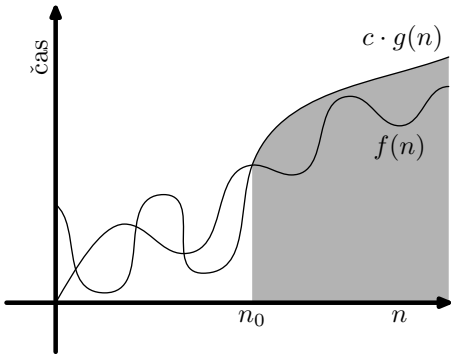
Slovy se dá asymptotická notace $f(n) = \mathcal{O}(g(n))$ popsat jako f neroste řádově rychleji než funkce g. Zápis $f(n) = \Omega(g(n))$ znamená, že funkce f roste řá-

[3] Podobný trend lze vypozorovat i pro velikost pevných disků, počtu pixelů ve foťácích apod.

[4] Samozřejmě si můžete vymyslet daleko lepší algoritmus. Tento je jen pro ukázku oscilující časové složitosti.

dově aspoň tak rychle jako funkce g a zápis $f(n) = \Theta(g(n))$, že obě funkce rostou řádově stejně rychle.

Můžeme si to vysvětlit i na obrázku vpravo. Notace $f(n) = \mathcal{O}(g(n))$ říká, že existuje taková konstanta c, že od určitého n_0 už leží graf funkce $f(n)$ pod grafem funkce $c \cdot g(n)$ (v šedě vyznačené oblasti).



Uveďme ještě pár příkladů. Platí $2^{56} = \mathcal{O}(1)$, $30n = \mathcal{O}(n)$, $n = \mathcal{O}(n^2)$, $n^{30} = \mathcal{O}(2^n)$, ale také $5n^2 + 30n = \mathcal{O}(n^2)$, protože $5n^2 + 30n \leq 35n^2 = \mathcal{O}(n^2)$.

Pozor na značení! Přesněji bychom měli psát $f \in \mathcal{O}(g)$ a říkat „f je třídy $\mathcal{O}(g)$“ a nebo „f patří do třídy $\mathcal{O}(g)$“. Značení s „=“ může být zavádějící, protože $\mathcal{O}(x) = \mathcal{O}(x^2)$, ale $\mathcal{O}(x^2) \neq \mathcal{O}(x)$. Musíme ho chápat spíše jako „$= \mathcal{O}$“ ve významu „$\preceq$“. Proto se také $f = \mathcal{O}(g)$ někdy čte jako „f je asymtoticky menší nebo rovno g“.

Lemma 1 *Nechť f_1, f_2, g_1, g_2 jsou funkce takové, že $f_1 \in \mathcal{O}(g_1)$, $f_2 \in \mathcal{O}(g_2)$, k je konstanta a h je rostoucí funkce. Potom*

- $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$
- $f_1 + f_2 \in \mathcal{O}(g_1 + g_2) = \mathcal{O}(\max(g_1, g_2))$
- $k \cdot f_1 \in \mathcal{O}(g_1)$
- $f_1(h(n)) \in \mathcal{O}(g_1(h(n)))$*, ale také* $h(f_1(n)) \in \mathcal{O}(h(g_1(n)))$.

Mezi často používané odhady patří následující. Pro každé $a \leq b$ platí $n^a = \mathcal{O}(n^b)$. Pro každé k platí $n^k = \mathcal{O}(2^n)$. To je ekvivalentní s $\log^k n = \mathcal{O}(n)$. Zkuste si tyto odhady i předchozí lemma dokázat (je to jen cvičení s funkcemi, limitami a použití definice velkého $\mathcal{O}$).

Definice vhodná pro výpočty: Asymptotickou notaci můžeme definovat i přes limitu.

$$f(n) = \mathcal{O}(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \in \langle 0, +\infty)$$

$$f(n) = \Omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} \in (0, +\infty\rangle$$

Tato metoda je lepší pro počítání, ale musíme být trochu zběhlí v počítání limit. Zkuste si dokázat, že definice přes limitu implikuje námi uvedenou definici. Je to jednoduché cvičení na definici limity.

Příklad: Potřebujeme zjistit, jestli platí $2n^2+4n-5 = \mathcal{O}(n^2)$. Podíváme se tedy na $\lim_{n\to\infty} \frac{2n^2+4n-5}{n^2} = 2$. Když si zvolíme $\varepsilon > 0$, například $\varepsilon = 3$, tak z definice limity dostaneme, že existuje n_0 takové, že pro každé $n \geq n_0$ platí $\frac{2n^2+4n-5}{n^2} < 2 + \varepsilon = 5$. Po přenásobení nerovnosti faktorem n^2 je to přesně definice vztahu $2n^2 + 4n - 5 = \mathcal{O}(n^2)$.

1.2 Časová složitost v nejhorším případě

Spočítat časovou složitost přesně je docela těžké. Protože jsme si ukázali, že na multiplikativních konstantách tolik nezáleží, budeme časovou složitost jenom odhadovat ze shora.

Některé úlohy přijímají více různých vstupů stejné velikosti (například když chceme setřídit n čísel). Doba výpočtu může být závislá na konkrétním vstupu. Potřebujeme odhadnout čas, za jak dlouho program určitě doběhne ve všech případech. K tomu nám stačí odhadnout dobu výpočtu v nejhorším případě. Té budeme říkat časová složitost v nejhorším případě.

Například pokud program počítá navádění letadel a snaží se zabránit srážkám, tak musíme vždycky dostat odpověď do pár vteřin. Nemůžeme říkat, že to normálně funguje rychle a bez problémů. Že došlo k nehodě jen proto, že tam ta letadla letěla blbě, nastala vyjímečná situace a my jsme to nestihli spočítat.

Uvedeme si několik příkladů, na kterých si ukážeme, jak počítat časovou složitost v nejhorším případě.

1.2.1 Hledání minima v poli

V poli $\texttt{A}[\,\cdot\,]$ dostaneme n čísel. Máme vrátit hodnotu nejmenšího z nich.

```
1: min := A[1]
2: for i = 2 to n do
3:         if A[i] < min then
4:             min := A[i]
5: return min
```

Řádky 3 a 4, tj. podmínka a případné zapamatování si nového minima, zaberou nejvýše konstantní čas. Označíme je za krok algoritmu. Spolu s testem uvnitř for-cyklu proběhnou $(n-1)$-krát. Ostatní řádky proběhnou jen jednou. Proto je celková časová složitost algoritmu $\mathcal{O}(n)$.

Umíte ukázat dolní odhad na časovou složitost hledání minima? Tím myslíme dokázali byste ukázat, kolik nejméně porovnání dvou čísel je potřeba k nalezení minima? Podívejme se na hledání minima jako na turnaj. Při utkání dvou prvků vyhrává ten menší. Chceme najít vítěze. Prvek, který je absolutním vítězem, musel porazit každý další prvek přímo a nebo nepřímo (porazit někoho, kdo ten prvek porazil – ať už přímo nebo nepřímo). Každý z $n-1$ prvků, které nevyhrály, musel být aspoň jednou poražen. Jinak o sobě může prohlašovat, že je také vítězem. Proto je potřeba alespoň $n-1$ porovnání (zápasů).

1.2.2 Sečtení prvků v matici

Dostaneme matici přirozených čísel o rozměrech $n \times n$ a chceme všechna čísla sečíst. Pomocí dvou for-cyklů projdeme celou matici a všechna čísla sečteme. Proto je časová složitost algoritmu $\mathcal{O}(n^2)$. Rychleji to nejde, protože musíme projít všech n^2 čísel.

Přesto můžeme o algoritmu říci, že je lineární ve velikosti vstupu. Musíme si uvědomit, že vstupem algoritmu je matice, která obsahuje n^2 čísel.

1.2.3 Vypisování n čísel

Dostaneme číslo n a úkolem je vypsat všechna čísla 1 až n. Jakou to bude mít časovou složitost? Pokud ji chceme vyjádřit v závislosti na čísle n, tak jednoduše $\mathcal{O}(n)$. Ovšem co když ji chceme vyjádřit vzhledem k velikosti vstupu? Velikost vstupu je $m := \log n$ bitů. Na výstup vypíšeme n čísel, každé o velikosti maximálně $\log n$. Časová složitost odpovídá počtu vypsaných bitů a to je $\mathcal{O}(n \log n) = \mathcal{O}(m2^m)$.

Vždy je potřeba si ujasnit, vzhledem k čemu budeme časovou složitost vyjadřovat.

1.2.4 Binární vyhledávání v setříděném poli

Máme pole `A`$[\,\cdot\,]$ obsahující n čísel setříděných od nejmenšího po největší. Chceme zjistit, jestli pole obsahuje číslo x.

```
1:  dolni := 1
2:  horni := n
3:  while horni ≥ dolni do
4:      stred := ⌊(dolni + horni)/2⌋
5:      if x = A[stred] then
6:          return TRUE
7:      else if x < A[stred] then
8:          horni := stred − 1
9:      else
10:         dolni := stred + 1
11: return FALSE
```

Invariant je vlastnost, která se v průběhu algoritmu nemění. Vhodných invariantů často využíváme v důkazech správnosti algoritmu, i při výpočtu časové složitosti.

Invariantem binárního vyhledávání je, že číslo x může být v poli `A`$[\,\cdot\,]$ pouze v úseku mezi indexy *dolni* a *horni*. Na začátku tento úsek obsahuje celé pole a v každé iteraci ho zmenšíme na polovinu. Pokud už je úsek tak malý, že neobsahuje žádný prvek, tak skončíme.

Díky půlení úseku proběhne while-cyklus nejvýše $\log n$ krát. To můžeme nahlédnout analýzou pozpátku. Průběh algoritmu sledujeme jako film, který si pustíme pozpátku. Nejprve má úsek jeden prvek, pak dvakrát tolik, to je dva prvky. A tak dále, až po $h = \lceil \log n \rceil$ krocích bude mít $2^h \geq n$ prvků.

Proto je časová složitost vyhledávání $\mathcal{O}(\log n)$.

1.2.5 Bublinkové třídění

V poli `A`$[\,\cdot\,]$ dostaneme n čísel. Čísla chceme setřídit pomocí porovnávání dvojic a prohazování prvků pole. Použijeme bublinkový algoritmus.

```
1: for j = 1 to n do
2: for i = 1 to n − j do
3:         if A[i] > A[i + 1] then
4:             prohoď(i, i + 1)
```

Prohoď(x,y) prohodí prvky v poli `A` na pozicích x a y. Algoritmus obsahuje dva for-cykly a každý z nich proběhne nejvýše n-krát. Řádky 3, 4 budou trvat konstantní čas a proto odhadneme časovou složitost jako $\mathcal{O}(n^2)$.

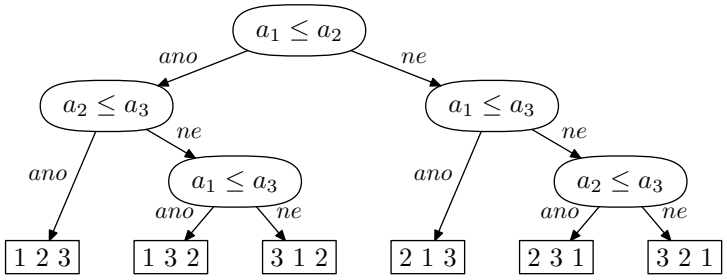
Mohli byste namítat, že jsme časovou složitost počítali příliš hrubě. Druhý for-cyklus přeci neproběhne vždycky n-krát, ale poprvé jen $(n-1)$-krát, pak $(n-2)$-krát,... až dvakrát a naposledy jen jednou. Přesný počet provedení řádek 3 a 4 je součtem aritmetické řady $(n-1) + (n-2) + \cdots + 2 + 1 = (n-1)\frac{(n-1)+1}{2} = {}^1\!/_2(n^2-n) = \mathcal{O}(n^2)$. I po přesnějším výpočtu nám vyšla stejná asymptotická časová složitost. Z toho vidíme, že si v určitých případech můžeme dovolit počítat hrubě.[5]

[5] Jak poznám, kdy mohu počítat hrubě? To se naučíte časem. Chce to jen trochu cviku.

1.2.6 Dolní odhad pro třídění

A co dolní odhad? Kolik nejméně porovnání (a případných prohození) je potřeba k setřídění n čísel a_1, a_2,..., a_n? Průběh každého deterministického algoritmu[6] lze zachytit rozhodovacím stromem.

Na následujícím obrázku je příklad rozhodovacího stromu pro setřídění a_1, a_2, a_3. Každý vnitřní vrchol odpovídá porovnávání dvou čísel a_i a a_j (a jejich případnému prohození). Toto porovnání můžeme vyjádřit otázkou „Je $a_i \leq a_j$?“ Listy stromu jsou označeny permutací, podle které musíme přerovnat vstup, abychom dostali setříděnou posloupnost. Tedy například listu označenému permutací „1 3 2“ odpovídá pořadí prvků $a_1 \leq a_3 \leq a_2$.



Jak funguje algoritmus odpovídající rozhodovacímu stromu? Algoritmus začne v kořenu stromu, kde se zeptá na první otázku. Odpověď mu určí větev stromu, kterou bude pokračovat. Po hraně dorazí k další otázce. Zeptá se na tuto otázku a podle odpovědi bude pokračovat dále. Postupně se ptá na otázky, které potká, a odpovědi mu určují cestu, kudy bude pokračovat. Nakonec dorazí do listu, kde si už musí být jist, jaké je uspořádání prvků ze vstupu podle velikosti.

Kontrolní otázka: Jakou permutaci setříděné posloupnosti musíme dát na vstup, aby algoritmus došel do listu s permutací π? Permutaci inverzní k π, to je π^{-1}.

Listy stromu musí obsahovat všechny možné permutace. Jinak by existovaly dvě permutace setříděné posloupnosti, které když dáme na vstup, tak povedou do stejného listu.[7] Algoritmus by nebyl schopen je rozlišit. Proto má rozhodovací strom alespoň $n!$ listů.

Rozhodovací strom je binární strom s alespoň $n!$ listy. Časová složitost v nejhorším případě je délka nejdelší cesty od kořene do listu. Ta je nejkratší, pokud je strom vyvážený a jeho listy obsahují každou permutaci jen jednou. Protože strom má $n!$ listů, tak je jeho výška alespoň

$$\log(n!) = \log n + \log(n-1) + \cdots + \log 1 \geq \frac{n}{2} \cdot \log(n/2) \geq \frac{1}{2} n \log n - \frac{n}{2} \log 2$$

(v první nerovnosti jsme si nechali jen první polovinu členů součtu a odhadli je zdola velikostí nejmenšího ze zbylých členů). Tím jsme ukázali, že každý algoritmus založený na porovnávání dvojic musí udělat $\Omega(n \log n)$ porovnání. Poznamenejme, že odhad je asymptoticky nejlepší možný, protože třídící algoritmy s časovou složitostí $\mathcal{O}(n \log n)$ v nejhorším případě existují (například heapsort nebo mergesort).

Pozor, jsou i algoritmy, které nejsou založené na porovnávání dvou čísel, ale využívají znalostí o tom, jak čísla vypadají. Například pokud na vstupu dostaneme n různých čísel z rozsahu 1 až N, tak si můžeme vytvořit pole $\mathtt{x}[\,\cdot\,]$ o velikosti N znázorňující charakteristický vektor množiny ze vstupu. Hodnota $\mathtt{x}[i]$ je 1, pokud

[6] To je takový algoritmus, který postupuje podle předem známého plánu. Nerozhoduje se ani náhodně, ani si nenechá věštit postup od vědmy či z orákula.

[7] Plyne to z Dirichletova holubníkového principu. Pro každý vstup se algoritmus dostane do nějakého listu. Pokud je vstupů více než listů, tak musí existovat list, do kterého vedou alespoň dva vstupy.

je číslo i součástí vstupu a 0 jinak. Pole $\mathtt{x}[\,\cdot\,]$ vyplníme jedním průchodem vstupu. Jedním průchodem pole $\mathtt{x}[\,\cdot\,]$ jsme schopni vypsat setříděnou posloupnost čísel. Časová složitost tohoto algoritmu je tedy $\mathcal{O}(n+N)$. Drobným vylepšením dostaneme například BucketSort, RadixSort, kterým se česky říká přihrádkové třídění. Nevýhodou těchto algoritmů jsou vyšší paměťové nároky a časová složitost závislá na velikosti čísel N. Záleží, co víme o datech, které chceme třídit.

1.3 Časová složitost v průměrném případě

Může se stát, že program bude na většině dat fungovat celkem rychle, akorát na pár „blbých“ výjimkách poběží hrozně dlouho. Abychom lépe popsali časovou složitost takového algoritmu, tak kromě časové složitosti v nejhorším případě uvedeme i průměrnou časovou složitost. Tu spočítáme jako průměr časových složitostí přes všechny možné vstupy.

Často není rychlost odpovědi otázkou života a smrti a proto nám nebude vadit, když si ve výjimečných případech počkáme déle. Důležité je, že v průměru budeme dostávat odpovědi relativně rychle.

Příkladem programu, který má průměrnou časovou složitost lepší než časovou složitost v nejhorším případě, je quicksort. Při nevhodné volbě pivota může mít až kvadratickou časovou složitost, ale v průměrném případě je jeho složitost $\mathcal{O}(n \log n)$. Přesným výpočtem se dá ukázat, že konstanta před $n \log n$ je o proti jiným třídícím algoritmům malá. To je důvod, proč se v praxi quicksort tolik používá.

1.3.1 QuickSort

Quicksort využívá techniku Rozděl a Panuj (Divide et Empera) o které si povíme později v sekci **??**.

Algoritmus dostane n čísel v poli $\mathtt{A}[\,\cdot\,]$ a má je setřídit od nejmenšího po největší.

Quicksort funguje rekurzivně. Dostene úsek pole $\mathtt{A}[l..p]$, který nejprve rozdělí na dva podúseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$ podle hodnoty, které se říká pivot. První podúsek obsahuje všechny prvky menší nebo rovné pivotu a druhý podúsek všechny prvky větší než pivot. Nakonec quicksort nechá oba podúseky setřídit rekurzivně.

```
1: Quicksort(l, p)
2:     if l < p then
3:         pivot := A[⌊(l + p)/2⌋]
4:         q :=Partition(l, r, pivot)
5:         Quicksort(l, q)
6:         Quicksort(q + 1, p)
```

Funkce q :=Partition(l, p, $pivot$) rozdělí úsek pole $\mathtt{A}[l..p]$ na dva úseky $\mathtt{A}[l..q]$ a $\mathtt{A}[q+1..p]$, kde první úsek obsahuje pouze prvky menší nebo rovny pivotu a druhý úsek pouze prvky větší než pivot. Funkce vrátí index q, který odpovídá hranici mezi podúseky.

```
1: Partition(l, p, pivot)
2:     while l < p do
3:         while A[l] ≤ pivot do l := l + 1
4:         while A[p] > pivot do p := p − 1
5:         Prohoď(l, p)
6:     return p
```

V ideálním případě, kdy se nám podaří vybrat všechny pivoty tak, aby se každý úsek pole rozdělil přesně napůl, dosteneme v nulté hladině rekurze 1 úsek o n prvcích, v první hladině rekurze 2 úseky o $n/2$ prvcích, ve druhé hladině rekurze

4 úseky o $n/4$ prvcích,... Časová složitost quicksortu při tomto dělení úseků je $\mathcal{O}(n + 2(n/2) + 4(n/4) + \cdots + 2^{\lceil \log n \rceil} \cdot 1) = \mathcal{O}(\lceil \log n \rceil \cdot n) = \mathcal{O}(n \log n)$.

Ovšem pokud budeme mít smůlu a pivot bude vždy nejmenším nebo největším prvkem z aktuálního úseku, tak bude bude časová složitost quicksortu $\mathcal{O}(n + (n - 1) + (n - 2) + \cdots = \mathcal{O}(n^2)$.

Quicksort má v nejhorším případě kvadratickou časovou složitost, ale dá se o něm ukázat, že časová složitost v průměrném případě je pouze $\mathcal{O}(n \log n)$. Upočítání vyžaduje dobrou znalost teorie pravděpodobnosti a proto ho přenecháme jiným knížkám (ale jinak je to jednoduché).

Poznámky k implementaci: Při implementaci quicksortu můžeme místo rekurze využít zásobníku, na který si budeme ukládat úseky pole, které je potřeba setřídit pomocí quicksortu. Při zpracování úseku uloženého na vrcholu zásobníku vyrobíme 2 nové podúseky. Myslíte, že záleží na pořadí, v jakém nové podúseky uložíme na zásobník? Samozřejmě, že ano. Vždy uložíme větší úsek pole dospod. Rozmyslete si, jak to ovlivní velikost paměti potřebné pro zásobník.

Druhým trikem při implementaci je pozorování, že pro malé vstupy je Insersort rychlejší než Quicksort. Proto od určité velikosti podúseků skončíme s rekurzivním voláním Quicksortu a podúsek dotřídíme Insertsortem.

1.4 Amortizovaná časová složitost

Amortizovaná časová složitost je v něčem podobná časové složitosti v průměrném případě. Podává lepší informaci o algoritmu než časová složitost v nejhorším případě, ale tentokrát nepotřebujeme nic počítat přes všechny možné vstupy, nepotřebujeme používat žádnou pravděpodobnost.

Když pracujeme s datovou strukturou (například s polem), tak můžeme veškerou práci s datovou strukturou realizovat pomocí několika operací. *Operace* je něco jako funkce pro práci s datovou strukturou. Operace provedou, co potřebujeme, a odstíní nás od znalosti fungování datové strukury. Příkladem operace pro práci s polem je vložení prvku do pole, smazání prvku, nalezení minima, apod.

Definice (Amortizovaná časová složitost): Je dána datová struktura D, na které postupně provádíme posloupnost stejných operací. Začneme s $D_0 := D$. První operace zavolaná na D_0 upraví datovou strukturu na D_1. Druhá operace zavolaná na D_1 upraví datovou strukturu na D_2. A tak dále. Postupně zavoláme i-tou operaci na D_{i-1} a ta upraví datovou strukturu na D_i. Některá operace může trvat krátce, jiná déle. Průměrný čas doby trvání operace nazveme amortizovanou časovou složitostí. *Amortizovanou časovou složitost* jedné operace spočítáme tak, že spočteme celkovou časovou složitost posloupnosti operací v nejhorším případě a vydělíme ji počtem operací.

K čemu je amortizovaná časová složitost? Pomůže nám lépe odhadnout časovou složitost některých algoritmů v nejhorším případě.

Příklad: Máme algoritmus, který používá jen jednu datovou strukturu a n krát volá operaci, která tuto datovou strukturou upravuje. Nic jiného nedělá. Časová složitost operace je v nejhorším případě $\mathcal{O}(n)$, ale její amortizovaná časová složitost je jen $\mathcal{O}(\log n)$. Celkovou časovou složitost celého algoritmu můžeme spočítat jako $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$. To ale může být mnohem více než skutečná časová složitost algoritmu. Druhá možnost odhadu časové složitosti je s pomocí amortizované časové složitosti. Ikdyž bude jeden konkrétní průběh operace trvat $\mathcal{O}(n)$, můžeme odhadnout časovou složitost algoritmu v nejhorším případě jako n krát amortizovaná časová složitost jedné operace. To je jen $\mathcal{O}(n \log n)$.

Výpočet amortizované časové složitosti. K výpočtu amortizované časové složitosti jedné operace se nejčastěji používají následující metody. Pro lepší pochopení metod se podívejte na konkrétní příklady výpočtu v následujících podsekcích.

1. **Přímo z definice.** Spočteme časovou složitost posloupnosti n operací v nejhorším případě a podělíme ji n.

2. **Účetní metoda.** Předpokládejme, že uhodneme, kolik je amortizovaná časová složitost operace. Jak ověřit, že je to pravda? Představme si výpočet na stroji, kde musíme za každou časovou jednotku strávenou výpočtem zaplatit jednu korunu. Na začátku přidělíme každé operaci[8] tolik korun, kolik je její amortizovaná složitost. V algoritmu pracujeme s objekty jako je vrchol, hrana, políčko pole,... Každému objektu otevřeme účet.

   Pokud bude operace trvat kratší čas, než kolik je její amortizovaná složitost, tak zaplatí za svůj výpočet a ještě jí zbydou peníze. Ty uloží na účty objektů, na kterých pracuje. Pokud bude naopak trvat delší čas, než je její amortizovaná složitost, tak si peníze sebere z účtů objektů, na kterých pracuje. Díky tomu bude moci zaplatit za svůj výpočet.

   Metoda spočívá v nalezení takových pravidel ukládání a vybírání peněz z účtů, aby se žádný účet nedostal do mínusu a abychom byli schopni zaplatit za všechnu vykonanou práci. Pokud se nám to povede, tak pomocí pravidel ověříme, že je náš odhad hodnoty amortizované časové složitosti správný.

3. **Metoda potenciálu**. Využijeme účtů z účetní metody. Potenciál je celkový vklad v bance, kde máme účty. Jinými slovy je to součet aktuálních hodnot vkladů na všech účtech.

   Pojďme si to ale vysvětlit pořádně. Nechť a je amortizovaná cena jedné operace. Začneme s datovou strukturou D_0 a postupně budeme provádět n operací. Nechť c_i je skutečná cena provedení i-té operace a D_i je datová struktura, která vznikla z D_{i-1} zavoláním i-té operace. Potenciální funkce Φ přiřazuje každé datové struktuře D_i reálné číslo $\Phi(D_i)$, které nazveme potenciál spojený s datovou strukturou D_i. Každá operace dostala tolik korun, kolik je její amortizovaná složitost. Pokud trvala kratší čas, než je její amortizovaná časová složitost, tak ušetřila peníze. Ušetřené peníze vloží do banky. Tím se zvýší potenciál (vklad v bance) o $\Phi(D_i) - \Phi(D_{i-1})$. Pokud operace trvala déle, než je její amortizovaná složitost, tak si peníze z banky naopak vybrala. V tomto případě je potenciálový rozdíl $\Phi(D_i) - \Phi(D_{i-1})$ záporný. Amortizovaná cena i-té operace vzhledem k potenciálu Φ je $a = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

   Celková amortizovaná cena všech n operací je $\sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$. Pokud skončíme s $\Phi(D_n) - \Phi(D_0) \geq 0$, tak nám na účtech zůstaly ještě nějaké penízky a amortizovaná časová složitost všech n operací je horním odhadem časové složitosti. Pokud skončíme s $\Phi(D_n) - \Phi(D_0) < 0$, tak se banka zadlužila musíme tuto hodnotu započítat do celkové časové složitosti všech operací.

   Mohlo se stát, že jsme na začátku nezačínali s prázdnou datovou strukturou, ale už jsme ji přebrali odjinud. V takovém případě potřebujeme na začátku vložit na účty počáteční vklad $\Phi(D_0)$, aby fungovala pravidla pro vkládání a vybírání peněz (jinak se banka dostane do mínusu). Podobně pokud skončíme s datovou strukturou, o kterou jsme pečovali a pilně střádali penízky na budoucí drahé operace, tak nám na účtech zbyde $\Phi(D_n)$ peněz.

[8] Abychom byli korektní, měli bychom místo „každé operaci“ říkat „každé instanci operace“.

Metoda potenciálu spočívá v nalezní potenciální funkce Φ a ověření, že potenciálový rozdíl $\Phi(D_{i-1}) - \Phi(D_i)$ spolu s amortizovanou složitostí operace pokryje náklady na provedení i-té operace.

Účetní metoda se používá i v následující variantě. Peníze rozdělíme na účty jednotlivých objektů. Operace nedostane žádné peníze, ale vždycky řekne, ze kterého účtu se bude její práce platit (většinou to jsou účty objektů, na kterých operace pracuje). Po provedení všech operací musíme ukázat, že se žádný účet nedostal do mínusu. Celková časová složitost posloupnosti operací bude odpovídat množství peněz, které jsme na začátku vložili na účty.

Amortizovanou složitost nemusíme počítat jen pro operace, ale i pro řádky zdrojového kódu, které se provádí několikrát. Například pro řádky v cyklu nebo v několika do sebe vnořených cyklech. Protože časová složitost vybraných řádek může být proměnlivá, tak nepočítáme celkovou časovou složitost podle počtu do sebe vnořených cyklů, ale přes účty. Díky tomu se nám často podaří ukázat lepší odhad.

V následujících podsekcích si na ukážeme výpočet amortizované časové složitosti pomocí všech tří metod na příkladech.

1.4.1 Kavárna „U Zavěšeného kafe“

V Praze je kavárna „U Zavěšeného kafe“. Na jedné zdi v kavárně visí několik hrnečků. Někteří hosté se mohou projevit jako dobrodinci. Když platí stovkou nebo jinou papírovou bankovkou, tak řeknou: „To je dobrý, nic mi nevracejte. Zbytek dejte do toho hrnku za zdi.“ A když do kavárny přijde nějaký chudý student, tak může barmana poprosit, jestli si může dát kafe na účet toho hrnku na zdi. Pokud je v hrnku dostatek peněz, tak dostane kafe.

Amortizovaná cena jednoho kafe je přesně tolik, kolik si účtuje kavárna. Ale jeho reálná cena je pro každého jiná. Bohatí dobrodinci vydají ze své peněženky za kafe víc. Normální lidé platí tolik, kolik kafe stojí a chudí studenti platí méně, protože část ceny hradí z hrnku na zdi.

1.4.2 Nafukovací pole

Potřebujeme navrhnout funkci *pridej*(x), která do pole přidá prvek x.

Pokud dopředu nevíme, jak velké pole budeme potřebovat, tak začneme s malým polem velikosti jedna[9] a v případě potřeby ho zvětšíme. Vždy, když se nám přidávaná položka nevejde do aktuálního pole, tak vytvoříme nové pole o dvojnásobné velikosti. Všechny položky do něj zkopírujeme, staré pole zrušíme a x přidáme až do nového pole. Vytvoření nového pole, kopírování prvků a rušení starého pole dohromady zabere čas $tn = \mathcal{O}(n)$, kde n je velikost starého pole a t je nějaká konstanta. Tolik je i časová složitost operace *pridej* v nejhorším případě. Časová složitost operace *pridej* je v nejhorším případě výrazně větší než v obyčejném poli, kde přidání prvku trvá jen $\mathcal{O}(1)$.

Ve skutečnosti to není tak zlé, protože časově náročné vytváření nového pole nastává málo často. Pokud jsme právě vytvořili nové pole o $2n$ položkách, tak musíme přidat dalších n čísel, než se pole zaplní a bude ho potřeba znova nafouknout.

Pro přidání n čísel do prázdného pole musíme provést nafukování celkem k-krát, kde $k = \lfloor \log n \rfloor$. Jedno nafouknutí pole n čísel trvá čas tn. Celkový čas všech nafukování je $t(1 + 2 + 4 + 8 + \cdots + 2^{k-1}) = t \cdot 2^k \leq 2tn$. Dohromady s časem za samotné přidání prvků do pole dostáváme amortizovanou složitost jedné operace $(2t + 1)n/n = 2t + 1 = \mathcal{O}(1)$.

[9] Při konkrétním použití bychom začali s větším polem. Přeci jen jsme schopni nějak odhadnout minimální velikost pole.

Druhá možnost výpočtu amortizované složitosti je pomocí účetní metody. Na začátku dáme každé operaci *pridej* $(2t+1)$ korun a každému políčku v poli otevřeme účet. Operace zaplatí jednu korunu za vlastní přidání prvku do pole a zbylých $2t$ korun dá na účet aktuálního políčka. Pokud se pole velikosti k zaplní, znamená to, že jsme od posledního nafouknutí přidali dalších $k/2$ prvků. Na účtě tedy máme tk korun, které použijeme na vytvoření nového pole a zkopírování všech k prvků.

Podobně můžeme počítat i pomocí metody potenciálu. Za potenciál můžeme zvolit $\Phi = 2t \cdot \#\text{prvků v poli}$.

1.4.3 Přičítání jedničky

V čítači máme hodně dlouhé binární číslo x a postupně k němu přičítáme jedničku. Číslo x má n bitů a jeho bity jsou uloženy v poli $\mathtt{A}[\,\cdot\,]$. Nejnižší bit je uložen v $\mathtt{A}[0]$ a nejvyšší bit v $\mathtt{A}[n-1]$, takže $x = \sum_{i=0}^{n-1} \mathtt{A}[i] \cdot 2^i$.

Přičítání děláme tak, jak nás to učili na základní škole. Zkusíme přičíst jedničku k poslednímu bitu. Pokud je nultý bit nula, tak jej přepíšeme na jedničku a jsme hotoví. Pokud je nultý bit jednička, tak obě jedničky sečteme a dostaneme "10", zapíšeme nulu a jedničku přeneseme o bit výše. Tam ji přičítáme k prvnímu bitu úplně stejně, jako jsme to dělali u nultého bitu. Pokud dojde k dalšímu přenosu, tak pokračujeme analogicky. Práci na úrovni jednoho bitu označíme za jeden krok.

Jak dlouho bude trvat přičtení jedničky? To záleží na aktuálním čísle v čítači. Pokud bude na konci nula, tak jsme po jednom kroku hotoví. Ale pokud bude na konci čísla k jedniček, tak budeme muset provést $k+1$ kroků. V nejhorším případě bude přičtení jedničky trvat $\mathcal{O}(n)$.

Jak dlouho bude trvat přičtení m jedniček? Předpokládejme, že je čítač na začátku vynulován. V každém druhém přičtení je poslední bit čítače nula (v čítači je sudé číslo) a proto bude přičítání trvat jen jeden krok. Tuto úvahu můžeme zobecnit. Pokud dojde k přenosu na k-tém bitu, tak muselo dojít i k přenosu na všech nižších bitech a tím pádem jsou všechny nižší bity vynulovány. Aby se na pozici k-tého bitu dostala opět jednička, budeme muset aspoň 2^k krát přičíst jedničku.

Proto můžeme říci, že krok na úrovni nultého bitu proběhne pokaždé, krok na úrovni prvního bitu jen při každém druhém přičtení, krok na úrovni třetího bitu jen při každém čtvrtém přičtení a tak dále. Celkem tedy proběhne $m + m/2 + m/2^2 + m/2^3 + \cdots + m/2^l \leq 2m$ kroků, kde $l = \lfloor \log m \rfloor$ (použili jsme vzorec pro součet geometrické řady $1 + 1/2 + 1/4 + 1/8 + \cdots = 2$). Z toho dostáváme amortizovanou časovou složitost jednoho přičtení $2m/m = 2$.

Druhou možností, jak určit amortizovanou časovou složitost, je použití účetní metody. Každému bitu otevřeme účet. Po celou dobu bude platit, že hodnota bitu udává počet korun, které má na účtě. Začneme s vynulovaným čítačem. Každá operace přičtení jedničky dostane dvě koruny. Jednu použije na první krok a druhou vloží na účet nultého bitu. Pokud by při tomto přičítání došlo k přenosu, tak jsou na účtu nultého bitu dvě koruny. Ty vybereme, dáme je přenášené jedničce a nultý bit nastavíme na nulu. Přenášená jednička má zase dvě koruny a může je na úrovni vyššího bitu použít úplně stejně. Tímto způsobem je každá operace přičtení schopna zaplatit za svoji práci. Proto je amortizovaná složitost jedné operace přičtení jedničky dva.

Pokud na začátku nezačínáme s prázdným čítačem, tak musíme každé jedničce v čítači vložit na účet jednu korunu. Zbytek už proběhne stejně.

Třetí možnost výpočtu amortizované složitosti je přes potenciál. Za potenciál $\Phi(i)$ zvolíme počet jedničkových bitů binárního čísla (po přičtení i jedniček). To, že potenciál funguje, se ukáže stejně jako u účetní metody.

1.4.4 Počítání stupňů vrcholů

Dostaneme graf s vrcholy $\{1, 2, \ldots, n\}$ a m hranami reprezentovaný seznamem sousedů,[10] tj. pro každý vrchol dostaneme spojový seznam sousedních vrcholů. V tomto grafu bychom chtěli pro každý vrchol spočítat jeho stupeň.[11] Můžeme postupovat podle následujícího algoritmu.

```
1: for v = 1 to n do
2:     deg[v] := 0
3:     for all w ∈ sousedi[v] do
4:         deg[v] := deg[v] + 1
```

Jaká je časová složitost tohoto algoritmu? Mohli byste říci, že se algoritmus skládá ze dvou for-cyklů, každý for-cyklus proběhne nejvýše n-krát a proto bude časová složitost algoritmu $\mathcal{O}(n^2)$. Ale ono se dá ukázat, že časová složitost je jen $\mathcal{O}(n + m)$.

K výpočtu amortizované časové složitosti použijeme variantu účetní metody. Za operace budeme považovat jednotlivé řádky algoritmu. Každému vrcholu grafu a každé hraně otevřeme v bance účet. Řádku 2 a průběh prvním for-cyklem budeme účtovat aktuálnímu vrcholu v. Řádku 4 a průběh druhým for-cyklem budeme účtovat hraně vw.

Po skončení algoritmu jsme z účtů zaplatili všechnu práci, kterou algoritmus vykonal. Každému vrcholu jsme z účtu strhli jen 2 koruny za provedení řádek 1 a 2. Každé hraně jsme strhli $2 + 2$ koruny za provedení řádek 3 a 4 dvakrát (na hranu jsme se podívali z každého koncového vrcholu jednou). Celkem jsme zaplatili $2n + 4m = \mathcal{O}(n + m)$ korun.

1.5 Příklady

1.5.1 Výpočet časové složitosti a asymptotické notace

1. (Procvičení notací velké $\mathcal{O}$, Ω a Θ) Pro následující dvojice funkcí f a g rozhodněte, jestli platí $f = \mathcal{O}(g)$ nebo $f = \Omega(g)$ a nebo oboje, tj. $f = \Theta(g)$.

(a)	$n - 100$	$n - 200$
(b)	$(n + 42)^8$	n^8
(c)	$n^{1/2}$	$n^{2/3}$
(d)	$100n + \log n$	$n + (\log^2 n)$
(e)	$n \log n$	$100n \log(16n)$
(f)	$\log 2n$	$\log 3n$
(g)	$10 \log n$	$\log(n^2)$
(h)	$\log \log n$	$\sqrt{\log n}$
(i)	$n^{1.01}$	$n \log n$
(j)	$n^2 / \log n$	$n \log^2 n$
(k)	$n^{0.1}$	$\log^{10} n$
(l)	$(\log n)^{\log n}$	$n / \log n$
(m)	$\sqrt{n}$	$\log^5 n$
(n)	$\sqrt{n}$	$n^{\sin n}$

[10] Reprezentace grafu seznamem sousedů je vysvětlena v kapitole **??** o reprezentacích grafu.
[11] Stupeň vrcholu v je počet hran, které z vrcholu v vycházejí.

(o)	$n^{1/2}$	$2^{\log_2 n}$
(p)	$n2^n$	3^n
(q)	2^n	2^{n+1}
(r)	$n!$	2^n
(s)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(t)	$\sum_{i=1}^{n} i^k$	n^{k+1}
(u)	$2^{\mathcal{O}(n)}$	$5^{\mathcal{O}(n)}$

Pokud se chcete pocvičit ještě o něco více, tak seřaďte všechny výše uvedené funkce podle podle asymptotické časové složitosti (zápis "$= \mathcal{O}$" bereme jako uspořádání "$\preceq$").

2. (Najděte chybu) Indukcí dokážeme, že $f(n) = n^2 \in \mathcal{O}(n)$. Pro $n = 1$ to platí. Předpokládejme tedy, že tvrzení platí až do nějakého n. Ukážeme, že platí i pro následníka. $f(n) = n^2 = (2n+1)+(n-1)^2 = \mathcal{O}(n)+f(n-1) = \mathcal{O}(n)+\mathcal{O}(n) = \mathcal{O}(n)$. Čtvrtá rovnost platí díky indukčnímu předpokladu $f(n-1) = O(n)$.

3. (Procvičení výpočtu časové složitosti – násobení $a \cdot b$ pomocí sčítání)

   Dostaneme dvě čísla $a, b \in \mathbb{N}$ a chceme spočítat jejich součin. Předpokládejme, že pracujeme na počítači, kde má každá proměnná neomezený počet bitů. Práce s takovými proměnnými není jednoduchá a tak nemůžeme použít instrukci pro násobení ani instrukci pro bitový posun. Máme k dispozici pouze instrukci pro sčítání. Pro jednoduchost předpokládejme, že součet dvou čísel trvá konstantní čas.[12]

   (a) Triviální řešení je následující:

   ```
   mezivysledek := b
   for i = 2 to a do
       mezivysledek := mezivysledek + b
   return mezivysledek
   ```

   Ukažte, že tento algoritmu má exponenciální časovou složitost ve velikosti vstupu.

   (b) Vymyslete řešení s lineární časovou složitostí ve velikosti vstupu.

1.5.2 Dolní odhad časové složitosti

1. (Dolní odhad pro vyhledávání v setříděném poli na základě porovnávání)

   Pomocí binárního vyhledávání (půlení intervalu) umíme v čase $\mathcal{O}(\log n)$ zjistit, jestli setříděné pole $\mathtt{A}[1, \ldots, n]$ obsahuje číslo x. Ukažte, že každý algoritmus, který pouze porovnává prvky pole s číselnou hodnotou, tj. ptá se na "$\mathtt{A}[i] \leq z$?", má časovou složitost v nejhorším případě alespoň $\Omega(\log n)$. Jinými slovy každému takovému algoritmu může ďábel podstrčit ošklivý vstup, na kterém algoritmus vykoná alespoň $\log n$ kroků.

1.5.3 Hledání algoritmu s co nejlepší časovou složitostí

1. (Které číslo chybí?) Množina C obsahuje všechna čísla 1 až n kromě jednoho. Na vstupu postupně dostanete všechna čísla z množiny C a vaším úkolem je zjistit, které číslo v množině chybí.

[12] Ve skutečnosti takoví kouzelníci nejsme. Sečtení dlouhých čísel vyžaduje čas úměrný počtu bitů potřebných pro reprezentaci obou čísel.

(a) Umíte to v lineárním čase?

(b) A co když můžeme použít jen konstantně mnoho paměti?

(c) Co když v množině budou chybět 2 čísla? Jak rychle zjistíte, která to jsou?

2. (Max gap) Dostanete n reálných čísel z intervalu $\langle 0, 1\rangle$. První dvě z nich jsou vždy 0 a 1. Velikost mezery mezi čísly $a, b \in \mathbb{R}$ je $|b - a|$. Mezera je prázdná, pokud interval mezi a a b neobsahuje žádné jiné ze zadaných čísel. Zjistěte, která dvě zadaná čísla mají mezi sebou největší prázdnou mezeru, a vypište je spolu s velikostí mezery.

   Umíte je najít v čase $\mathcal{O}(n)$?

3. (Palindrom) Palindrom je slovo které se čte stejně zepředu i pozpátku. Na vstupu dostanete řětězec n znaků. Navrhněte algoritmus, který v řetězci co nejrychleji najde nejdelší palindrom a vypíše jeho délku. Své řešení otestujte například na řetězcích: “madam”, “mam”, “rotor”, “kuna nese nanuk”.

   Umíte to rozhodnout v čase $\mathcal{O}(n)$?

1.5.4 Amortizovaná časová složitost

1. (Nafukovací i smršťovací pole)

   V sekci o amortizované časové složitosti jsme si vysvětlili, jak funguje nafukovací pole. Co kdybychom chtěli přidat mazání prvků? Když už bude pole poloprázdné, tak bychom ho mohli smrštit na poloviční velikost. Rozmyslete si, při jaké obsazenosti pole by se mělo provádět nafukování nebo smršťování pole, aby amortizovaná časová složitost operací `delete(`x`)` a `insert(`x`)` byla stále konstantní.

2. (Binární vyhledávání + přidávání prvků)

   Chceme si reprezentovat n-prvkovou množinu čísel M. Často budeme volat operace `najdi(`x`)` a `pridej(`x`)`. Operace `najdi(`x`)` zjistí, jestli $x \in M$. Operace `pridej(`x`)` provede $M := M \cup \{x\}$.

   K reprezentaci množiny M použijeme následující datovou strukturu. Nechť $[n_{k-1}n_{k-2}\ldots n_1n_0]_2$ je binárním zápisem čísla n. Množinu M si místo jednoho pole reprezentujeme pomocí $k := \lceil \log(n+1) \rceil$ uspořádaných polí A_0, A_1, …, A_{k-1} o velikostech 1, 2, 4, …, 2^{k-1}. Velikost pole A_i je 2^i. Každé pole A_i je buď celé prázdné nebo celé plné, podle toho, jestli je $n_i = 0$ nebo $n_i = 1$. Celkový počet prvků ve všech polích je $\sum_{i=0}^{k-1} n_i 2^i = n$. Ačkoliv je každé pole setříděné, o velikostech prvků v různých polích nic nevíme.

   (a) Popište, jak v této datové struktuře provádět operaci `najdi(`x`)` a určete její časovou složitost v nejhorším případě.

   (b) Popište, jak v této datové struktuře provádět operaci `pridej(`x`)` a určete její časovou složitost v nejhorším případě. Kromě toho určete i její amortizovanou složitost.

   (c) Zkuste vymyslet, jakým způsobem by se v datové struktuře dalo provádět mazání prvků.

3. (Procvičení metody potenciálu: hledání následníka v binárním stromě)

Máme binární vyhledávací strom T a nějaký jeho vrchol v. Následník[13] vrcholu v je vrchol s nejmenším vyšším klíčem, než je hodnota klíče ve v. Nalezení následníka trvá v nejhorším případě čas $\mathcal{O}(\text{hloubka } T)$. Ukažte, že pokud budeme chtít najít k následníků, tak to zvládneme v čase $\mathcal{O}(k + \text{hloubka } T)$.

Nápověda: Nechť $w \in T$ je aktuální vrchol při hledání následníků. Zkuste potenciál $\Phi = (\text{hloubka } T - \text{hloubka } w \text{ v } T) + 2 \cdot \#\text{pravých hran na cestě z kořene do } w$.

[13] Pozor, následník vrcholu (syn) je něco jiného než zde zavedený následník. Pro definici hloubky stromu a podobných pojmů se podívejte do kapitoly **??**.

Literatura