1. Pár příkladů na úvod

Tématem této knihy má být návrh a analýza algoritmů. Měli bychom tedy nejdříve říci, co to algoritmus je. Formální definice je překvapivě obtížná. Nejspíš se shodneme na tom, že je to nějaký formální postup, jak něco provést, a že by měl být tak podrobný, aby byl srozumitelný i počítači. Jenže detaily už vůbec nejsou tak zřejmé. Proto s pořádným zavedením pojmů ještě kapitolu počkáme a zatím se podíváme na několik konkrétních příkladů algoritmů.

1.1. Úsek s největším součtem

Náš první příklad se bude týkat posloupností. Máme zadanou nějakou posloupnost $x_1, \ldots, x_n$ celých čísel a chceme v ní nalézt *úsek* (tím myslíme souvislou podposloupnost), jehož součet je největší možný. Takovému úseku budeme říkat *nejbohatší.* Jako výstup nám postačí hodnota součtu, nebude nutné ohlásit přesnou polohu úseku.

Nejprve si rozmyslíme triviální případy: Kdyby se na vstupu nevyskytovalo žádné záporné číslo, má evidentně maximální součet celá vstupní posloupnost. Pokud by naopak byla všechna x_i záporná, nejlepší je odpovědět prázdným úsekem, který má nulový součet; všechny ostatní úseky mají součet záporný.

Obecný případ bude komplikovanější: například v posloupnosti

$$1, -2, 4, 5, -1, -5, 2, 7$$

najdeme dva úseky kladných čísel se součtem 9 (totiž $4, 5$ a $2, 7$), ale dokonce se hodí spojit je přes záporná čísla $-1, -5$ do jediného úseku se součtem 12. Naopak hodnotu -2 se použít nevyplácí, jelikož přes ní je dosažitelná pouze počáteční jednička, takže bychom si o 1 pohoršili.

Nejpřímočařejší možný algoritmus by téměř doslovně kopíroval zadání: Vyzkoušel by všechny možnosti, kde může úsek začínat a končit, pro každou z nich by spočítal součet prvků v úseku a pak našel z těchto součtů maximum.

Algoritmus MaxSoučet1
Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *zatím jsme potkali jen prázdný úsek*
2. Pro $i = 1, \ldots, n$ opakujeme: ◁ *i je začátek úseku*
3. Pro $j = i, \ldots, n$ opakujeme: ◁ *j je konec úseku*
4. $s \leftarrow 0$ ◁ *součet úseku*
5. Pro $k = i, \ldots, j$ opakujeme:
6. $s \leftarrow s + x_k$
7. $m \leftarrow \max(m, s)$

Výstup: Součet m nejbohatšího úseku v X

Pojďme alespoň zhruba odhadnout, jak rychlý tento postup je. Prozkoumáme řádově n^2 dvojic (*začátek*, *konec*) a pro každou z nich strávíme řádově n kroků počítáním součtu. To dohromady dává řádově n^3 kroků, což už pro $n = 1\,000$ budou miliardy. Zkusme přijít na rychlejší způsob.

Podívejme se, čím náš první algoritmus tráví nejvíce času. Jistě počítáním součtů. Například sčítá jak úsek $x_i, \ldots, x_j$, tak $x_i, \ldots, x_{j+1}$, aniž by využil toho, že druhý součet je prostě o x_{j+1} vyšší než ten první. Nabízí se zvolit pevný začátek úseku i a pak zkoušet všechny možné konce j od nejlevějšího k nejpravějšímu. Každý další součet pak dovedeme triviálně spočítat z předchozího. Pro jedno i tedy provedeme řádově n kroků, celkově pak řádově n^2.

Algoritmus MaxSoučet2

Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *zatím jsme potkali jen prázdný úsek*
2. Pro $i = 1, \ldots, n$ opakujeme: ◁ *i je začátek úseku*
3. $\quad s \leftarrow 0$ ◁ *součet úseku*
4. $\quad$ Pro $j = i, \ldots, n$ opakujeme: ◁ *j je konec úseku*
5. $\quad\quad s \leftarrow s + x_j$
6. $\quad\quad m \leftarrow \max(m, s)$

Výstup: Součet m nejbohatšího úseku v X

Myšlenka průběžného přepočítávání se ale dá využít i lépe, totiž na celou úlohu. Uvažujme, jak se změní výsledek, když ke vstupu $x_1, \ldots, x_n$ přidáme ještě x_{n+1}. Všechny úseky původního vstupu zůstanou zachovány a navíc k nim přibudou nové úseky tvaru $x_i, \ldots, x_{n+1}$. Stačí tedy ověřit, zda součet některého z nových úseků nepřekročil dosavadní maximum, čili porovnat maximum se součtem nejbohatšího *koncového* úseku nové posloupnosti.

Nejbohatší koncový úsek také neumíme najít v konstantním čase, ale pojďme tutéž myšlenku použít ještě jednou. Jak se změní koncové úseky po přidání x_n? Všem stávajícím koncovým úsekům stoupne součet o x_n a navíc vznikne nový koncový úsek obsahující samotné x_n. Maximální součet je proto roven buď předchozímu maximálnímu součtu plus x_n, nebo samotnému x_n – podle toho, co je větší.

Označíme-li si tedy k maximální součet koncového úseku, přidáním nového prvku se tato hodnota změní na $\max(k + x_n, x_n) = x_n + \max(k, 0)$. Jinými slovy: počítáme průběžné součty, ale pokud součet klesne pod nulu, tak ho vynulujeme. Hledaný maximální součet m je pak maximem ze všech průběžných součtů. Tímto principem se řídí náš třetí algoritmus:

Algoritmus MaxSoučet3

Vstup: Posloupnost $X = x_1, \ldots, x_n$ uložená v poli

1. $m \leftarrow 0$ ◁ *prázdný úsek je tu vždy*
2. $k \leftarrow 0$ ◁ *maximální součet koncového úseku*
3. Pro i od 1 do n opakujeme:
4. $\quad k \leftarrow \max(k, 0) + x_i$

5. $m \leftarrow \max(m, k)$

Výstup: Součet m nejbohatšího úseku v X

V každém průchodu cyklem potřebujeme na přepočítání proměnných k a m pouze konstantně mnoho operací. Celkem jich tedy algoritmus provede řádově n, tedy lineárně s velikostí vstupu. Hodnoty ze vstupu navíc potřebuje jen jednou, takže je může číst postupně a vystačí si tudíž s konstantním množstvím paměti.

Dodejme ještě, že úvaha typu „jak se změní výstup, když na konec vstupu přidáme další prvek?“ je poměrně častá. Vysloužila si proto zvláštní jméno, algoritmům tohoto druhu se říká *inkrementální*. Ještě se s nimi několikrát potkáme.

Cvičení

1. Upravte algoritmus MaxSoučet3, aby oznámil nejen maximální součet, ale také polohu příslušného úseku.
2. Na vstupu je text složený z písmen české abecedy a mezer. Vymyslete algoritmus, který najde nejdelší úsek textu, v němž se žádné písmeno neopakuje.
3. Najděte v českém textu nejkratší úsek, který obsahuje všechna písmena abecedy. Malá a velká písmena nerozlišujte.

4.* Úsek posloupnosti je *k-hladký* (pro $k \geq 0$), pokud se každé dva jeho prvky liší nejvýše o k. Popište co nejefektivnější algoritmus pro hledání nejdelšího k-hladkého úseku.

5. Jak spočítat kombinační číslo $\binom{n}{k}$? Výpočtu přímo podle definice brání potenciálně obrovské mezivýsledky (až $n!$), které se nevejdou do celočíselné proměnné. Navrhněte algoritmus, který si vystačí s čísly omezenými n-násobkem výsledku.

1.2. Binární vyhledávání

Jak se hledá slovo ve slovníku? Jistě můžeme slovníkem listovat stránku po stránce a pečlivě zkoumat slovo po slovu. Jsme-li dostatečně trpěliví, hledané slovo nakonec najdeme, nebo slovník skončí a můžeme si být jistí, že slovo neobsahoval.

Listování slovníkem může být dobrá zábava na dlouhé zimní večery (nebo spíš na celou polární noc), ale obvykle hledáme jinak: otevřeme slovník někde uprostřed, podíváme se, jak blízko jsme k hledanému slovu, a na základě toho nadále aplikujeme stejný postup buďto v levé, anebo pravé části rozevřeného slovníku.

Nyní se tento postup pokusíme popsat precizně. Získáme tak algoritmus, kterému se říká *binární vyhledávání* nebo také *hledání půlením intervalu*. Na vstupu dostaneme nějakou uspořádanou posloupnost $x_1 \leq x_2 \leq \ldots \leq x_n$ a hledaný prvek y.

Postupujeme takto: pamatujeme si interval $x_\ell, \ldots, x_r$, ve kterém se prvek může nacházet. Na počátku je $\ell = 1$ a $r = n$. V každém kroku vybereme prvek ležící uprostřed (nebo přibližně uprostřed, pokud je prvků sudý počet). Ten bude sloužit jako mezník oddělující levou polovinu od pravé. Pokud se mezník rovná hledanému y, můžeme hned úspěšně skončit. Pokud je menší než y, znamená to, že y se může

nacházet jen napravo od něj – všechny prvky nalevo jsou menší než mezník, tím pádem i menší než y. A pokud je naopak mezník větší než y, víme, že y se může nacházet pouze v levé polovině.

Postupně tedy interval $[\ell, r]$ zmenšujeme na polovinu, čtvrtinu, atd., až se dostaneme do stavu, kdy prohledávaný úsek pole má velikost jednoho prvku, nebo je dokonce prázdný. Pak už se snadno přesvědčíme, zda jsme hledaný prvek našli.

V pseudokódu náš algoritmus vypadá následovně.

Algoritmus BinSearch (hledání půlením intervalu)
Vstup: Uspořádaná posloupnost $x_1 \leq \ldots \leq x_n$, hledaný prvek y

1. $\ell \leftarrow 1$, $r \leftarrow n$ ◁ $x_\ell, \ldots, x_r$ *tvoří prohledávaný úsek pole*
2. Dokud je $\ell \leq r$:
3. $\quad s \leftarrow \lfloor (\ell + r)/2 \rfloor$ ◁ *střed prohledávaného úseku*
4. $\quad$ Pokud je $y = x_s$: vrátíme s a skončíme.
5. $\quad$ Pokud je $y > x_s$:
6. $\quad\quad \ell \leftarrow s + 1$ ◁ *přesouváme se napravo*
7. $\quad$ Jinak:
8. $\quad\quad r \leftarrow s - 1$ ◁ *přesouváme se nalevo*
9. Vrátíme 0. ◁ *nenašli jsme*

Výstup: Index hledaného prvku, případně 0, pokud prvek v poli není

Pokusme se poctivě dokázat, že algoritmus funguje. Především nahlédneme, že výpočet se vždy zastaví: v každém průchodu cyklem zmenšíme prohledávaný úsek alespoň o 1. Korektnost pak plyne z toho, že kdykoliv oblast zmenšíme, odstraníme z ní jen prvky, které jsou zaručeně různé od y. Jakmile tedy algoritmus skončí, buďto jsme y našli, nebo jsme naopak vyloučili všechny možnosti, kde by mohlo být.

Nyní ukážeme, že binární vyhledávání je mnohem rychlejší než probrání všech prvků.

Věta: Při hledání v posloupnosti délky n provede algoritmus BinSearch nejvýše $\log_2 n$ průchodů cyklem.

Důkaz: Stačí nahlédnout, že v každém průchodu cyklem se velikost prohledávaného úseku zmenší alespoň dvakrát. Proto po k průchodech úsek obsahuje nejvýše $n/2^k$ prvků, takže pro $k > \log_2 n$ je úsek nutně prázdný. □

Na závěr dodejme, že prvky naší posloupnosti vůbec nemusí být čísla: stačí, aby to byly libovolné objekty, které jsme schopni mezi sebou porovnávat. Třeba slova ve slovníku. V oddílu **??** navíc dokážeme, že logaritmický počet porovnání je nejlepší možný.

Dvojice se zadaným součtem

Podívejme se ještě na jeden příbuzný problém. Opět dostaneme na vstupu nějakou uspořádanou posloupnost $x_1 \leq x_2 \leq \ldots \leq x_n$ a číslo s. Tentokrát ovšem nehledáme jeden prvek, nýbrž dva (ne nutně různé), jejichž součet je s.

Řešení „hrubou silou“ by zkoušelo sečíst všechny dvojice $x_i + x_j$, ale těch je řádově n^2. Pokud ovšem zvolíme nějaké x_i, víme, že x_j musí být rovno $s - x_i$. Můžeme tedy vyzkoušet všechna x_i a pokaždé půlením intervalu hledat $s - x_i$. Každé vyhledávání spotřebuje řádově $\log_2 n$ kroků, celkově jich tedy bude řádově $n \cdot \log_2 n$.

To je mnohem lepší, ale ještě ne optimální. Představme si, že k x_1 hledáme $s - x_1$. Tentokrát ale nebudeme hledat binárně, nýbrž pěkně prvek po prvku od konce pole. Dokud jsou prvky větší, přeskakujeme je. Jakmile narazíme na prvek menší, víme, že už se můžeme zastavit, protože dál už budou jen samé menší.

Pozici, kde jsme skončili, si zapamatujeme. Máme tedy nějaké j takové, že $x_j < s - x_1 < x_{j+1}$. (Pokud protestujete, že x_{j+1} může ležet mimo posloupnost, představte si za koncem posloupnosti ještě $+\infty$.)

Nyní přejdeme na x_2 a hledáme $s - x_2$. Jelikož $x_2 \geq x_1$, musí být $s - x_2 \leq s - x_1$. Všechna čísla, která byla větší než $s - x_1$ jsou tedy také větší než $s - x_2$, takže v nich nemá smysl hledat znovu. Proto můžeme pokračovat od zapamatované pozice t dále doleva. Pak si zase zapamatujeme, kde jsme skončili, což se bude hodit pro x_3, a tak dále.

Existuje hezčí způsob, jak formulovat totéž. Říká se mu *metoda dvou jezdců*. Máme dva indexy: levý a pravý. Levý index i popisuje, které x_i zrovna zkoušíme jako první člen dvojice: začíná na pozici 1 a pohybuje se doprava. Pravý index j ukazuje na místo, kde jsme se zastavili při hledaní $s - x_i$: začíná na pozici n a postupuje doleva.

Kdykoliv je $x_j > s - x_i$, posuneme j doleva (pokračujeme v hledání $s - x_i$). Je-li naopak $x_j < s - x_i$, posuneme i doprava ($s - x_i$ se v posloupnosti určitě nenachází, zkoušíme další x_i). Takto pokračujeme, dokud buďto neobjevíme hledanou dvojici, nebo se jezdci nesetkají – tehdy dvojice zaručeně neexistuje.

Algoritmus DvojiceSeSoučtem
Vstup: Uspořádaná posloupnost $x_1 \leq \ldots \leq x_n$, hledaný součet s
Výstup: Indexy i a j, pro něž je $x_i + x_j = s$, nebo neúspěch

1. $i \leftarrow 1$, $j \leftarrow n$
2. Dokud $i \leq j$:
3. Je-li $x_j + x_j = s$:
4. Vrátíme jako výsledek dvojici (i, j).
5. Jinak je-li $x_i + x_j < s$: ◁ *totéž jako* $x_i < s - x_j$
6. $i \leftarrow i + 1$
7. Jinak:
8. $j \leftarrow j - 1$
9. Ohlásíme neúspěch.

Snadno nahlédneme, že cyklus proběhne nejvýše $2n$-krát. Pokaždé se totiž pohne jeden z jezdců, ale každý z nich může urazit nejvýše n kroků, než vyjede ven.

Překonali jsme tedy rychlost opakovaného binárního vyhledávání. Povedlo se nám to díky tomu, že mezi hledanými prvky existoval nějaký vztah: konkrétně každý prvek byl menší nebo roven předchozímu.

Cvičení

1. Rozmyslete, jak se bude chovat algoritmus binárního vyhledávání, pokud se bude hledaný prvek v posloupnosti nacházet vícekrát. Algoritmus upravte, aby vždy vracel první výskyt hledaného prvku (ne jen libovolný).
2. Upravte binární vyhledávání, aby v případě, kdy hledaný prvek v posloupnosti není, nahlásilo nejbližší větší prvek.
3.* *Nekonečná verze:* Popište algoritmus, který v nekonečné posloupnosti $x_1 < x_2 < \ldots$ najde pozici i takovou, že $x_i \leq y < x_{i+1}$. Počet kroků hledání by neměl přesáhnout řádově $\log_2 i$.
4. *Lokální minimum:* Je dána posloupnost $-\infty = x_0, x_1, \ldots, x_n, x_{n+1} = +\infty$. O prvku x_i řekneme, že je lokálním minimem, pokud $x_{i-1} \geq x_i \leq x_{i+1}$. Navrhněte co nejrychlejší algoritmus, který nějaké lokální minimum najde.
5. *Součet úseku:* Je dána posloupnost $x_1, \ldots, x_n$ kladných čísel a číslo s. Hledáme i a j taková, že $x_i + \ldots + x_j = s$. Navrhněte co nejefektivnější algoritmus.
6.* Jak se změní úloha z předchozího cvičení, pokud povolíme i záporná čísla?
7. *Implicitní vstup:* Posloupnost, v níž binárně hledáme, nemusí být nutně celá uložená v paměti. Stačí, když se tak dokážeme dostatečně přesvědčivě tvářit: kdykoliv se algoritmus zeptá na hodnotu nějakého prvku, rychle ho vyrobíme. Zkuste tímto způsobem spočítat *celočíselnou odmocninu* z čísla x. To je největší y takové, že $y^2 \leq x$.
8. *První díra:* Dostali jsme rostoucí posloupnost přirozených čísel. Chceme najít nejmenší přirozené číslo, které v ní chybí. Vymyslete, jak k tomu přesvědčit binární vyhledávání.
9.* Opět hledáme nejmenší chybějící číslo, ale tentokrát na vstupu dostaneme neuspořádanou posloupnost navzájem různých přirozených čísel. Kromě této posloupnosti máme k dispozici jenom konstantně mnoho paměti.
10. *Monotónní predikáty:* Na předchozích několik cvičení se můžeme dívat trochu obecněji. Mějme nějakou vlastnost φ, kterou všechna přirozená čísla od 0 do nějaké hranice k mají a žádná větší nemají. Popište, jak binárním vyhledáváním zjistit, kde se nachází tato hranice.
11. *Rovnoměrná data:* Mějme pole délky n. Na každé pozici se může vyskytovat libovolné celé číslo z rozsahu 1 až k. Čísla vybíráme rovnoměrně náhodně (všechny hodnoty mají stejnou pravděpodobnost). Následně pole setřídíme a budeme v něm chtít vyhledávat. Zkuste upravit binární vyhledávání, aby pro tyto vstupy fungovalo v průměru rychleji.
12.* Kolik porovnání provede takový algoritmus v průměru?
13.* Může se stát, že výše uvedený algoritmus nedostane pěkná data. Můžeme mu nějak pomoci, aby nebyl ani v takovém případě o mnoho horší než binární vyhledávání?

1.3. Euklidův algoritmus

Pro další příklad se vypravíme do starověké Alexandrie. Tam žil ve 3. století před naším letopočtem filosof Euklides (Ευκλείδης) a stvořil jeden z nejstarších algoritmů.[1] Ten slouží k výpočtu největšího společného dělitele a používá se dodnes.

Značení: *Největšího společného dělitele* celých kladných čísel x a y budeme značit $\gcd(x, y)$ podle anglického Greatest Common Divisor.

Nejprve si všimneme několika zajímavých vlastností funkce gcd.

Lemma G: Pro všechna celá kladná čísla x a y platí:

1. $\gcd(x, x) = x$,
2. $\gcd(x, y) = \gcd(y, x)$,
3. $\gcd(x, y) = \gcd(x - y, y)$ pro $x > y$.

Důkaz: První dvě vlastnosti jsou zřejmé z definice. Třetí dokážeme v silnější podobě: ukážeme, že dvojice (x, y) a $(x-y, y)$ sdílejí množinu všech společných dělitelů, tedy i největšího z nich.

Pokud nějaké d je společným dělitelem čísel x a y, musí platit $x = dx'$ a $y = dy'$ pro vhodné x' a y'. Nyní stačí zapsat $x - y$ jako $dx' - dy' = d(x - y)$ a hned je jasné, že d dělí i $x - y$.

Naopak pokud d dělí jak $x - y$, tak y, musí existovat čísla t' a y' taková, že $x-y = dt'$ a $y = dy'$. Zapíšeme tedy x jako $(x-y)+y$, což je rovno $dt'+dy' = d(t'+y')$, a to je dělitelné d. □

Díky lemmatu můžeme gcd počítat tak, že opakovaně odečítáme menší číslo od většího. Jakmile se obě čísla vyrovnají, jsou rovna největšímu společnému děliteli. Algoritmus nyní zapíšeme v pseudokódu.

Algoritmus OdčítacíEuklides
Vstup: Celá kladná čísla x a y

1. $a \leftarrow x$, $b \leftarrow y$
2. Dokud $a \neq b$, opakujeme:
3. Pokud $a > b$:
4. $a \leftarrow a - b$
5. Jinak:
6. $b \leftarrow b - a$

Výstup: Největší společný dělitel $a = \gcd(x, y)$

Nyní bychom měli dokázat, že algoritmus funguje. Důkaz rozdělíme na dvě části:

[1] Tehdy se tomu ovšem tak neříkalo. Pojem *algoritmu* je novodobý, byl zaveden až začátkem 20. století při studiu „mechanické“ řešitelnosti matematických úloh. Název je poctou perskému matematikovi al-Chorézmímu, jenž žil cca 1100 let po Euklidovi a v pozdějších překladech jeho díla mu jméno polatinštili na Algoritmi.

Lemma Z: Algoritmus se vždy zastaví.

Důkaz: Sledujme, jak se vyvíjí součet $a + b$. Na počátku výpočtu je roven $x + y$, každým průchodem cyklem se sníží alespoň o 1. Přitom zůstává stále nezáporný, takže průchodů nastane nejvýše $x + y$. □

Lemma S: Pokud se algoritmus zastaví, vydá správný výsledek.

Důkaz: Dokážeme následující *invariant,* neboli tvrzení, které platí po celou dobu výpočtu:

> **Invariant:** $\gcd(a, b) = \gcd(x, y)$.
>
> *Důkaz:* Obvyklý způsob důkazu invariantů je indukce podle počtu kroků výpočtu. Na počátku je $a = x$ a $b = y$, takže invariant jistě platí. V každém průchodu cyklem se pak díky vlastnostem 2 a 3 z lemmatu **G** platnost invariantu zachovává. □

Z invariantu plyne, že na konci výpočtu je $\gcd(a, a) = \gcd(x, y)$. Zároveň díky vlastnosti 1 z lemmatu **G** platí $\gcd(a, a) = a$. □

Víme tedy, že algoritmus je funkční. To bohužel neznamená, že je použitelný: například pro $x = 1\,000\,000$ a $y = 2$ vytrvale odčítá y od x, až po 499 999 krocích vítězoslavně ohlásí, že největší společný dělitel je roven 2.

Stačí si ale všimnout, že opakovaným odčítáním b od a dostaneme zbytek po dělení čísla a číslem b. Jen si musíme dát pozor, že pro a dělitelné b se zastavíme až na nule. To odpovídá tomu, že algoritmus provede ještě jedno odečtení navíc, takže skončí, až když se jedno z čísel vynuluje. Nahrazením odčítání za dělení se zbytkem získáme následující algoritmus. Když se v současnosti hovoří o Euklidově algoritmu, obvykle se myslí tento.

Algoritmus EUKLIDES
Vstup: Celá kladná čísla x a y

1. $a \leftarrow x$, $b \leftarrow y$
2. Opakujeme:
3. Pokud $a < b$, prohodíme a s b.
4. Pokud $b = 0$, vyskočíme z cyklu.
5. $a \leftarrow a \bmod b$ ◁ *zbytek po dělení*

Výstup: Největší společný dělitel $a = \gcd(x, y)$

Správnost je zřejmá: výpočet nového algoritmu odpovídá výpočtu algoritmu předchozího, jen občas provedeme několik původních kroků najednou. Zajímavé ovšem je, že na první pohled nenápadnou úpravou jsme algoritmus podstatně zrychlili:

Lemma R: Euklidův algoritmus provede nejvýše $\log_2 x + \log_2 y + 1$ průchodů cyklem.

Důkaz: Vývoj výpočtu budeme sledovat prostřednictvím součinu ab:

Tvrzení: Součin ab po každém průchodu cyklem klesne alespoň dvakrát.

Důkaz: Kroky 3 a 4 součin ab nemění. Ve zbývajícím kroku 5 platí $a \geq b$ a b se evidentně nezmění. Ukážeme, že a klesne alespoň dvakrát, takže ab také. Rozebereme dva případy:

- $b \leq a/2$. Tehdy platí $a \bmod b < b \leq a/2$.
- $b > a/2$. Pak je $a \bmod b = a - b \leq a - (a/2) = a/2$. □

Na počátku výpočtu je $ab = xy$ a díky právě dokázanému tvrzení po k průchodech cyklem musí platit $ab \leq xy/2^k$. Kromě posledního neúplného průchodu cyklem ovšem ab nikdy neklesne pod 1, takže k může být nejvýše $\log_2 xy = \log_2 x + \log_2 y$. □

Shrnutím všeho, co jsme o algoritmu zjistili, získáme následující větu:

Věta: Euklidův algoritmus vypočte největšího společného dělitele čísel x a y. Provede přitom nejvýše $c \cdot (\log_2 x + \log_2 y + 1)$ aritmetických operací, kde c je konstanta.

Cvičení

1. Největšího společného dělitele bychom také mohli počítat pomocí prvočíselného rozkladu čísel x a y. Rozmyslete si, jak by se to dělalo a proč je to pro velká čísla velmi nepraktické.
2. V kroku 3 algoritmu EUKLIDES není potřeba porovnávat. Nahlédněte, že pokud bychom a s b prohodili pokaždé, vyjde také správný výsledek, jen nás to bude v nejhorším případě stát o jeden průchod cyklem navíc.
3. Dokažte, že počet průchodů cyklem je nejvýše $2 \log_2 \min(x, y) + 2$.
4. Pro každé x a y existují celá čísla α a β taková, že $\gcd(x, y) = \alpha x + \beta y$. Těmto číslům se říká *Bézoutovy koeficienty.* Upravte Euklidův algoritmus, aby je vypočetl.
5. Pomocí předchozího cvičení můžeme řešit *lineární kongruence.* Pro daná a a n chceme najít x, aby platilo $ax \bmod n = 1$. To znamená, že ax a 1 se liší o násobek n, tedy $ax + ny = 1$ pro nějaké y. Pokud je $\gcd(a, n) = 1$, pak x a y jsou Bézoutovy koeficienty, které to dosvědčí. Je-li $\gcd(a, b) \neq 1$, nemůže mít rovnice řešení, protože levá strana je vždy dělitelná tímto gcd, zatímco pravá nikoliv. Jak najít řešení obecnější rovnice $ax \bmod n = b$?
6. Nabízí se otázka, není-li logaritmický odhad počtu operací z naší věty příliš velkorysý. Abyste na ni odpověděli, najděte funkci f, která roste nejvýše exponenciálně a při výpočtu $\gcd(f(n), f(n+1))$ nastane právě n průchodů cyklem.
7. Binární algoritmus na výpočet gcd funguje takto: Pokud x i y jsou sudá, pak $\gcd(x, y) = 2\gcd(x/2, y/2)$. Je-li x sudé a y liché, pak $\gcd(x, y) = \gcd(x/2, y)$. Jsou-li obě lichá, odečteme menší od většího. Zastavíme se, až bude $x = y$. Dokažte, že tento algoritmus funguje a že provede nejvýše $c \cdot (\log_2 x + \log_2 y)$ kroků pro vhodnou konstantu c.

8.* Mějme permutaci π na množině $\{1, \ldots, n\}$. Definujme její mocninu následovně: $\pi^0(x) = x$, $\pi^{i+1}(x) = \pi(\pi^i(x))$. Najděte nejmenší $k > 0$ takové, že $\pi^k = \pi^0$.

1.4. Fibonacciho čísla a rychlé umocňování

Dovolíme si ještě jednu historickou exkurzi, tentokrát do Pisy, kde na začátku 13. století žil jistý Leonardo řečený Fibonacci.⟨2⟩ Příštím generacím zanechal zejména svou posloupnost.

Definice: *Fibonacciho posloupnost* $F_0, F_1, F_2, \ldots$ je definována následovně:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n.$$

Příklad: Prvních 11 Fibonacciho čísel zní 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

Pokud chceme spočítat F_n, můžeme samozřejmě vyjít z definice a postupně sestrojit prvních n členů posloupnosti. To nicméně vyžaduje řádově n operací, takže se nabízí otázka, zda to lze rychleji. V moudrých knihách nalezneme následující větu:

Věta (kouzelná formule): Pro každé $n \geq 0$ platí:

$$F_n = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

Důkaz: Laskavý, nicméně trpělivý čtenář jej provede indukcí podle n. Jak na kouzelnou formuli přijít, naznačíme v cvičení 3. □

Dobrá, ale jak nám formule pomůže, když pro výpočet n-té mocniny potřebujeme $n-1$ násobení? Inu nepotřebujeme, následující algoritmus to zvládne rychleji:

Algoritmus MOCNINA

Vstup: Reálné číslo x, celé kladné n

1. Pokud $n = 0$, vrátíme výsledek 1.
2. $t \leftarrow \text{MOCNINA}(x, \lfloor n/2 \rfloor)$
3. Pokud n je sudé, vrátíme $t \cdot t$.
4. Jinak vrátíme $t \cdot t \cdot x$.

Výstup: x^n

Lemma: Algoritmus MOCNINA vypočte x^n pomocí nejvýše $2 \log_2 n + 2$ násobení.

Důkaz: Správnost je evidentní z toho, že $x^{2k} = (x^k)^2$ a $x^{2k+1} = x^{2k} \cdot x$. Co se počtu operací týče: Každé rekurzivní volání redukuje n alespoň dvakrát, takže po nejvýše $\log_2 n$ voláních musíme dostat jedničku a po jednom dalším nulu. Hloubka rekurze je tedy $\log_2 n + 1$ a na každé úrovni rekurze spotřebujeme nejvýše 2 násobení. □

To dává elegantní algoritmus pro výpočet F_n pomocí řádově $\log_2 n$ operací. Jen je bohužel pro praktické počítání nepoužitelný: Zlatý řez $(1+\sqrt{5})/2 \doteq 1.618\,034$ je

⟨2⟩ Což je zkratka z „filius Bonaccii“, tedy „syn Bonacciho“.

iracionální a pro vysoké hodnoty n bychom ho potřebovali znát velice přesně. To neumíme dostatečně rychle. Zkusíme to tedy menší oklikou.

Po Fibonacciho posloupnosti budeme posouvat okénkem, skrz které budou vidět právě dvě čísla. Pokud zrovna vidíme čísla F_n, F_{n+1}, v dalším kroku uvidíme $F_{n+1}, F_{n+2} = F_{n+1} + F_n$. To znamená, že posunutí provede s okénkem nějakou lineární transformaci a každá taková jde zapsat jako násobení maticí. Dostaneme:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}.$$

Levou matici označíme $\mathcal{F}$ a nahlédneme, že násobení okénka n-tou mocninou této matice musí okénko posouvat o n pozic. Tudíž platí:

$$\mathcal{F}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

Nyní stačí využít toho, že násobení matic je asociativní. Proto můžeme n-tou mocninu matice vypočítat obdobou algoritmu Mocnina a vystačíme si s řádově $\log n$ maticovými násobeními. Jelikož pracujeme s maticemi konstantní velikosti, obnáší každé násobení matic jen konstantní počet operací s čísly. Všechny matice jsou příjemně celočíselné. Proto platí:

Věta: n-té Fibonacciho číslo lze spočítat pomocí řádově $\log n$ celočíselných aritmetických operací.

Cvičení

1. Naprogramujte funkci Mocnina nerekurzivně. Může pomoci převést exponent do dvojkové soustavy.
2. Uvažujme obecnou *lineární rekurenci řádu k:* $A_0, \ldots, A_{k-1}$ jsou dána pevně, $A_{n+k} = \alpha_1 A_{n+k-1} + \alpha_2 A_{n+k-2} + \ldots + \alpha_k A_n$ pro konstanty $\alpha_1, \ldots, \alpha_k$. Vymyslete efektivní algoritmus na výpočet A_n, nejlépe pomocí řádově $\log_2 n$ operací.
3.* Jak odvodit kouzelnou formuli: Uvažujme množinu všech posloupností, které splňují rekurentní vztah $A_{n+2} = A_{n+1} + A_n$, ale mohou se lišit hodnotami A_0 a A_1. Tato množina tvoří vektorový prostor, přičemž posloupnosti sčítáme a násobíme skalárem po složkách a roli nulového vektoru hraje posloupnost samých nul. Ukažte, že tento prostor má dimenzi 2 a sestrojte jeho bázi v podobě exponenciálních posloupností tvaru $A_n = \alpha^n$. Fibonacciho posloupnost pak zapište jako lineární kombinaci prvků této báze.
4. Dokažte, že $F_n \geq \varphi^n$, kde $\varphi = (1 + \sqrt{5})/2 \doteq 1.618\,034$.
5.* Algoritmy založené na explicitní formuli pro F_n jsme odmítli, protože potřebovaly počítat s iracionálními čísly. To bylo poněkud ukvapené. Dokažte, že čísla tvaru $a + b\sqrt{5}$, kde $a, b \in \mathbb{Q}$ jsou uzavřená na sčítání, odčítání, násobení i dělení. K výpočtu formule si tedy vystačíme s racionálními čísly, dokonce pouze typu $p/2^q$, kde p a q jsou celá. Odvoďte z toho jiný logaritmický algoritmus.