Kapitola 1

Jak zrychlovat programy?

Zde si předvedeme efektivní programovací techniky.
Asymptoticky rychlejším algoritmem zrychlíme program nejvíce.
Uvedeme i praktické informace, kterými program zrychlíme 2krát, 3krát, ale někdy i 10krát.

Možností je několik. Nejprve je ale potřeba zjistit, ve které části programu strávíme nejvíce času. Tím zjistíme, která část je úzkým hrdlem (bottle neck), a tu budeme zrychlovat. Pokud budeme zrychlovat část, ve které výpočet stráví jen 2% celkového času, tak si celkově moc nepomůžeme. Akorát ztratíme spoustu času programováním úprav. Lepší je se zaměřit na proceduru, ve které strávíme 80% celkového času.[1]

Zjistit, ve kterých funkcích strávíme nejvíce času, je dnes velmi jednoduché. Nemusíte nic programovat. Stačí použít nástroj, který se jmenuje profiler. Bývá součástí vývojového prostředí. Spustíte profiler společně se svým programem a rovnou sledujete přehledné statistiky, kolikrát byla spuštěna která funkce a kolik času jste v ní strávili.

Zlaté pravidlo pro optimalizace kódu zní: „Hrajte si s tím, vyzkoušejte všechno možné, porovnejte to a nakonec vyberte to nejlepší řešení.“ Raději ještě jednou zopakujeme nejdůležitější myšlenku celé kapitoly: „Je potřeba mít jasno v tom, které části programu má smysl optimalizovat a které ne.“

Největších zrychlení dosáhneme lepším algoritmem. Teprve nakonec, když už lepší algoritmus nevymyslíme, nebo když si dokonce dokážeme, že lepší algoritmus neexistuje, má smysl se vrhnout na optimalizaci zdrojového kódu a optimalizaci programu pro hardware a operační systém.

Upozornění: kusy kódu algoritmů v této knize nejsou vždy ty nejefektivnější a nejelegantnější. Je to z toho důvodu, že cílem této knihy je hlavně jednoduše a srozumitelně prezentovat grafové algoritmy. Jejich elegantní implementace už hodně závisí na konkrétním programovacím jazyce.

1.1 Předpočítání si výsledků do paměti

Nejrychlejší program řešící zadaný problém je ten, který nic nepočítá a rovnou vypíše správný výsledek. Zdá se vám to nemožné? Výsledky si můžeme spočítat dopředu a uložit si je do tabulky. Tabulku můžeme vložit buď přímo do zdrojového

[1] Pravidlo 80 na 20. Toto pravidlo často používají ekonomové a manažeři při svých rozhodnutích. Empiricky je ověřeno, že zhruba 80% zákazníků přinese 20% zisku a 20% zákazníků přinese 80% zisku. Na které zákazníky se máme zaměřit? Zkuste se zamyslet nad tím, jak toto pravidlo může využít programátor.

kódu programu (jako statické inicializované pole)[2] nebo ji můžeme na začátku programu načíst ze souboru. Další možností je, že tabulku vygenerujeme po spuštění programu a oželíme krátké zdržení.

Nalezení výsledku v tabulce a jeho vypsání nám často zabere jen konstantní čas, případně čas úměrný velikosti odpovědi. Ovšem má to jednu mouchu. Zatím jsme tiše předpokládali, že výsledky půjdou poskládat do tabulky a že tabulka bude rozumně velká. To nemusí být u každé úlohy splněno. Ne každá úloha je pro předpočítání výsledků vhodná. Například řešení úlohy, která na vstupu dostane velký graf, se bude do tabulky ukládat špatně. Na druhou stranu úloha, která dostane číslo $n \in \{1, \ldots, 1000\}$ a má spočítat $f(n)$ pro nějakou složitou funkci f, vhodná je. Předpočítané hodnoty $f(n)$ si hravě uložíme do pole velikosti 1000.

Příklad: (Hašování do nafukovacího pole) – viz. nafukovací pole v kapitole **??** o amortizované časová složitosti. Je spousta možností, jak zvolit hašovací funkci. Běžně se používá funkce, kterou počítáme x modulo prvočíslo p. Při nafukování pole na velikost $2n$ bychom potřebovali znát největší prvočíslo menší nebo rovno $2n$. Jak takové prvočíslo rychle najít? Otázka je, proč ho vůbec během nafukování pole hledat. Můžeme si přeci dopředu pro každou velikost pole najít vhodné prvočíslo a uložit si ho do tabulky.

1.2 Výpočet hodnoty na základě předchozí

Úloha: Dostaneme polynom $P(x) = ax^2+bx+c$, kde a, b, $c \in \mathbb{Z}$ jsou předem známé konstanty. Pro všechna $x \in \{1, 2, \ldots, n\}$ bychom chtěli spočítat hodnotu polynomu. Například proto, abychom si mohli nakreslit graf funkce $P(x)$.

Řešení: (Hörnerovo schéma) Pro každé x spočítáme hodnotu $P(x)$ Hörnerovým schématem. Hörnerovo schéma vychází z možnosti částečně povytýkat x. Dostaneme $P(x) = (((a)x + b)x + c)$. Uzávorkování nám dává návod, jak vyhodnotit polynom v bodě x. Budeme postupovat v cyklu, začneme od nejvnitřnější závorky. V každém kroku přenásobíme vnitřní závorku hodnotou x a přičteme k ní odpovídající konstantu.

Pro vyhodnocení polynomu ve všech $x \in \{1, \ldots, n\}$ budeme $2n$ krát násobit a $2n$ krát sčítat.

Řešení: (výpočet hodnoty na základě předchozí) Tentokrát zkusíme hodnotu $P(x + 1)$ spočítat na základě předchozí hodnoty $P(x)$. Pro vypočtení $P(1)$ potřebujeme jen dvakrát sčítat. $P(x + 1) = a(x + 1)^2 + b(x + 1) + c = P(x) + (2a)x + (a + b)$. Pro výpočet $P(x + 1)$ tedy úplně stačí, když k předchozí hodnotě $P(x)$ přičteme $(2a)x + (a + b)$. Označíme $Q(x) = (2a)x + (a + b)$. Zbývá ukázat, jak rychle spočítat hodnotu $Q(x)$. Spočteme ji stejným trikem. Pro výpočet $Q(1)$ nám stačí tři sčítání. Pro výpočet hodnoty $Q(x+1)$ na základě předchozí nám stačí jednou sčítat, protože $Q(x + 1) = Q(x) + 2a$.

$\mathtt{P}[1] := a + b + c$; $\mathtt{Q}[1] := 3a + b$
`for` $i = 2$ `to` n `do`
  $\mathtt{P}[i] := \mathtt{P}[i - 1] + \mathtt{Q}[i - 1]$
  $\mathtt{Q}[i] := \mathtt{Q}[i - 1] + 2a$

Takto vypočítáme hodnotu $P(x)$ ve všech bodech $x \in \{1, \ldots, n\}$ pomocí pouhých $2n + 5$ sčítání.[3] Pokud by a, b, c byly opravdu předem známé konstanty a

[2] Pomocný program, který předpočítá výsledky do tabulky, může rovnou vypisovat zdrojový kód pro vložení tabulky.

[3] Na většině počítačů je instrukce pro násobení časově náročnější než instrukce pro sčítání. Takže je ve skutečnosti druhý výpočet ještě rychlejší, než se zdá.

ne proměnné, tak můžeme ušetřit i těch prvních 5 sčítání tím, že si $P(1)$, $Q(1)$ předpočítáme.

Dokážete úlohu zobecnit pro vyhodnocování polynomu stupně 3, stupně 4 nebo obecně pro polynomy stupně k?

1.3 Využití předchozích hodnot

Palindrom je slovo které se čte stejně zepředu i pozpátku. Příkladem palindromů jsou řetězce: „madam“, „mam“, „rotor“, „kuna nese nanuk“, „kobyla ma maly bok“ (po vynechání mezer).

Úloha: (Nejdelší palindrom) Na vstupu dostanete řětězec n znaků. Navrhněte algoritmus, který v řetězci co nejrychleji nalezne nejdelší palindrom.

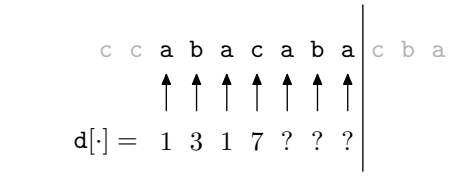
Řešení:(jednoduché kvadratické) Každý palindrom má svůj střed kolem kterého je symetrický (levá půlka palindromu je zrcadlovým obrazem pravé půlky). Středy palindromu jsou dvou druhů. Palindromy liché délky mají uprostřed písmeno, palindromy sudé délky mají uprostřed mezeru mezi písmeny.

Řešení, které nás hned napadne, je vyzkoušet všechny možné středy a z každého středu expandovat dosud nalezený palindrom do stran. Takové řešení má v nejhorším případě časovou složitost $\mathcal{O}(n^2)$. Příkladem řetězce, na kterém algoritmus dosáhneme kvadratické časové složitosti je řetězec se samých `a`, například `aaaaaaaaa`.

Řešení: (v lineárlním čase) Jak se dá předchozí algoritmus vylepšit, abychom dosáhli lineální časové složitosti? Základní myšlenka zůstane stejná, chceme pro všechny středy s vyplnit

$$\mathtt{d}[s] = \text{ délka nejdelšího palindromu se středem } s.$$

Ale jak to udělat? Podívejme se nejprve na malý příklad. Předpokládejme, že už jsme nalezli nejdelší palindrom se středem na pozici s. Říkejme mu aktuální master-palindrom. Na obrázku je vyznačen tučně, má délku 7 znaků a jeho střed je na pozici 6.



Jaké hodnoty vyplnit do $\mathtt{d}[7]$, $\mathtt{d}[8]$,...? Kdybychom tyto hodnoty hledali expanzí, tak se vrátíme zpátky ke kvadratické časové složitosti. Využijeme malý trik, kterým je zrcadlení. Protože je pravá půlka palindromu zrcadlovým obrazem levé půlky palindromu, platí $\mathtt{d}[s+i] = \mathtt{d}[s-i]$ pokud pravý okraj ozrcadleného palindromu nepřesáhne pravý okraj master-palindromu.

Na obrázku je pravý okraj master-palindromu vyznačen čárou. Ta zároveň odděluje oblast řetězce ze vstupu, která ještě nebyla prozkoumána. V příkladu na obrázku můžeme nastavit $\mathtt{d}[7] := 1$. Podobně $\mathtt{d}[8] := 3$, ale to ještě nemusí být délka nejdelšího palindromu se středem na pozici 8. Tento palindrom délky 3 se dotýká čáry označující neprozkoumanou oblast a proto možná půjde rozšířit.

Pro všechny středy, jejichž palindromy jsou celé uvnitř aktuálního master-palindromu, můžeme nastavit $\mathtt{d}[s+i] := \mathtt{d}[s-i]$. První střed z leva, jehož palindrom se dotýká nebo překračuje hranici master-palindromu, se stane středem S nového master-palindromu.

Nový master-palindrom nalezneme expanzí do stran. Nemusíme začínat úplně od začátku. Už víme, že $\mathtt{d}[S]$ znaků kolem středu S tvoří palindrom a proto stačí tento palindrom rozšiřovat do stran. První znak řetězce ze vstupu, na který „sáhneme“, bude ten napravo od posledního znaku původního master-palindromu (první znak za čárou).

Algoritmus začne s master-palindromem, jehož střed je na prvním znaku řetězce ze vstupu. Postupně hledá následující master-palindromy a za každý master-palindrom vyplní příslušný úsek pole $\mathtt{d}[\,\cdot\,]$. Na závěr už jen stačí najít maximální hodnotu v tomto poli.

Algoritmus má lineální časovou složitost, protože na každý znak řetězce ze vstupu sáhneme nejvýše dvakrát a protože každé políčko pole $\mathtt{d}[\,\cdot\,]$ vyplňujeme jen jednou.

Poznámka: Nezapomeňte, že palindromy mají dva druhy středů. Jedny leží na pozici písmenka a druhé mezi sousedními písmenky.

1.4 Přímé generování výsledků

Úloha: Množina M obsahuje pouze taková přirozená čísla, která nejsou dělitelná jiným prvočíslem než 2, 3 a 5. Vypište co nejrychleji prvních n nejmenších čísel množiny M.

Řešení: První řešení, které člověka napadne, je procházet postupně všechna přirozená čísla a testovat, zda nejsou dělitelná jiným prvočíslem než 2, 3 a 5. Nebudeme si vysvětlovat detaily tohoto řešení a raději si ukážeme daleko rychlejší řešení.

Řešení: Nejprve učiníme několik pozorování. Teprve v posledním odstavci si ukážeme, jak provést jednu iteraci algoritmu.

Množina M obsahuje pouze čísla tvaru $2^i 3^j 5^k$, pro všechny možné indexy $i, j, k \geq 0$. Dejme tomu, že už jsme vypsali prvních s čísel m_1, m_2, m_3, …, m_s množiny M. Vydělením čísla m_{s+1} jedním z čísel $2, 3, 5$ dostaneme menší číslo patřící do M, které už bylo vypsáno. Proto následující číslo m_{s+1} dostaneme tak, že jedno z předchozích čísel m_i pro $i \leq s$ vynásobíme buď 2 nebo 3 a nebo 5.

Protože $m_1 < m_2 < m_3 < \ldots$, tak i $2m_1 < 2m_2 < 2m_3 < \ldots$. Najdeme si takový index I, že $2m_i \leq m_s$ pro všechna $i < I$ a $2m_i > m_s$ pro všechna $i \geq I$. Podobně pro čísla tvaru $3m_j$ a $5m_k$ najdeme indexy J a K.

Následující číslo, které máme vypsat, je $m_{s+1} = \min\{\, 2m_I, 3m_J, 5m_K \,\}$. Až ho vypíšeme, tak stačí posunout příslušný index o 1. Tím nastavíme indexy I, J, K na správnou hodnotu. Pro výpis dalšího čísla můžeme postup zopakovat.

1.5 Předzpracování dat

Výpočet můžeme často zrychlit tím, že si vstupní data nejprve upravíme do vhodné podoby, případně si něco dalšího předpočítáme a teprve z těchto dat počítáme výstup. Celý výpočet tedy rozdělíme do dvou fází – předzpracování a vlastního výpočtu.

Úloha: (nejčastěji se vyskytující číslo) Dostanete pole obsahující n celých čísel a máte vypsat číslo, které se v poli vyskytuje nejčastěji.

Řešení: Hloupé řešení si pro každý prvek pole spočítá, kolikrát se v poli vyskytuje (pro každý prvek projde celé pole), a vybere ten s největším počtem výskytů. Jeho časová složitost je $\mathcal{O}(n^2)$.

Řešení: Určitě vás napadne nejjednodušší možné předzpracování a to je setřídění dat. Setřídit pole umíme v čase $\mathcal{O}(n \log n)$. Potom bude k nalezení nejčastěji se

vyskytujícího čísla stačit jen jeden průchod pole. Ten proběhne v čase $\mathcal{O}(n)$. Celkem toto řešení zabere čas $\mathcal{O}(n \log n)$.

Úloha: (největší jedničková podmatice) Matice A velikosti $n \times m$ obsahuje nuly a jedničky. Podmatice je souvislý obdélníkový výřez z matice A (určený levým horním a pravým dolním rohem). Jedničková podmatice je podmatice obsahující samé jedničky. Najděte největší jedničkovou podmatici matice A.

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Na obrázku je matice A velikosti 6×9. Největší jedničková podmatice obsahuje 9 jedniček a její levý horní roh leží na pozici $(3, 4)$.[4]

Řešení: Nejjednodušší řešení vyzkouší všechny možné polohy levého horního i pravého dolního rohu a pro jimi určenou podmatici zkontroluje, jestli se skládá ze samých jedniček. Této kontrole budeme jednoduše říkat testování podmatice.

Všech možných poloh levého horního rohu je mn. Stejně tak i poloh pravého dolního rohu. Otestování, jestli se jimi určená podmatice skládá ze samých jedniček vyžaduje průchod celé podmatice a trvá v nejhorším případě čas $\mathcal{O}(mn)$. Celkem je časová složitost tohoto řešení $\mathcal{O}(m^3 n^3)$.

Řešení: Zamysleme se nad tím, co je na výše uvedeném řešení neefektivní. Často testujeme podmatice, které se překrývají. Testování průniku obou podmatic by stačilo dělat jen jednou. Chceme-li se vyhnout opakovanému testování některých podmatic, tak si vhodně předzpracujeme vstupní data. Ke každé jedničce si spočítáme, kolik jedniček leží v souvislé řadě pod ní. Tento počet k jedničce přičteme. Získané údaje si můžeme uložit přímo do matice A (nepotřebujeme pomocnou datovou strukturu), protože nuly zůstanou tam, kde byly a místo jedniček budeme mít v matici uložena nenulová čísla.

$$\text{Z matice } A = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \text{ dostaneme matici } B = \begin{pmatrix} 2 & 0 & 4 & 3 \\ 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

Toto předzpracování nám zabere čas $\mathcal{O}(mn)$. Stačí každým sloupcem projít jednou od zdola nahoru, nuly ponechat beze změny a ke každé jedničce přičíst hodnotu prvku ležícího pod ní.

Jak hledat největší jedničkovou podmatici? Opět vyzkoušíme všechny polohy levého horního rohu (mn možností). Pro každý levý horní roh budeme zkoumat všechny možné polohy pravého horního rohu (až m možností). Pravý horní roh leží ve stejném řádku jako levý horní roh a mezi oběma rohy nesmí ležet žádná nula, jinak podmatice nebude jedničková.

Když už budeme znát levý i pravý horní roh podmatice, jak zjistíme velikost největší podmatice s těmito rohy? K tomu využijeme předvýpočtu. Každé číslo v řádku mezi levým a pravým horním rohem určuje velikost souvislého úseku jedniček ležícího ve sloupci pod ním (včetně jeho pozice). Takže stačí vzít minimum z těchto čísel. Velikost největší jedničkové podmatice určené horními rohy bude součin tohoto minima a vzdálenosti mezi horními rohy.

[4] Pozor, pozice se udává jako (řádek, sloupec) – jak už je u matic zvykem. Je to naopak než souřadnice (x, y).

Nyní shrneme, jak bude probíhat celý výpočet. Vyzkoušíme všechny možné pozice levého horního rohu. Pro každý levý horní roh stačí postupovat vpravo, dokud nenarazíme na nulu nebo na pravý okraj matice A. V průběhu postupu zkoušíme polohy pravého horního rohu a počítáme celkové minimum na základě předchozí hodnoty.

Vlastní hledání jedničkové podmatice bude trvat čas $mn \cdot \mathcal{O}(m) = \mathcal{O}(m^2 n)$. Dohromady s předzpracováním toto řešení vyžaduje čas $\mathcal{O}(m^2 n)$.

Řešení: Předchozí řešení můžeme ještě zrychlit. Provedeme ještě jedno předzpracování a spočítáme si i počet jedniček ležících nad každou pozicí. Kromě matice $B = (b_{ij})_n^m$, kde b_{ij} je délka souvislého úseku jedniček začínajícího na ij a ležícího pod pozicí ij, si vytvoříme matici $C = (c_{ij})_n^m$, kde c_{ij} je délka souvislého úseku jedniček začínajícího na ij a ležícího nad pozicí ij. Obě předzpracování stihneme provést v čase $\mathcal{O}(mn)$.

Klíčovým pojmem pro celé řešení je význačná pozice. Pozice ij je *význačná*, pokud za prvé $a_{ij} = 1$ a za druhé buď $a_{i,j-1} = 0$ nebo $a_{i,j-1}$ leží mimo matici A. Jinými slovy na význačné pozici leží 1 a vlevo vedle ní je buď 0 a nebo je pozice ij na levém okraji matice A.

Každá maximální jedničková podmatice nejde rozšířit o sloupec doleva, protože se vedle její levé hranice nachází 0 a nebo už tam končí matice A. To ale neznamená nic jiného, než že levý sloupec každé maximální jedničkové podmatice obsahuje význačnou pozici.

Jak bude probíhat hledání největší jedničkové podmatice? Pro každou význačnou pozici prozkoumáme všechny jedničkové podmatice, které ji obsahují ve svém levém sloupci. Provedeme to následovně. Začneme ve význačné pozici ij. Postupně budeme procházet doprava, dokud nenarazíme na nulu nebo pravý okraj matice A. V každém kroku, tedy na pozici ik, $k \geq j$, zjistíme velikost největší jedničkové podmatice obsahující obě pozice ij, ik. To provedeme stejně jako v předchozím řešení. Najdeme $b = \min\{b_{i,j}, b_{i,j+1}, \ldots, b_{i,k-1}, b_{i,k}\}$ a $c = \min\{c_{i,j}, c_{i,j+1}, \ldots, c_{i,k-1}, c_{i,k}\}$. Velikost největší takové jedničkové podmatice bude $(b+c-1)\cdot(k-j+1)$. Všechny pozice, na které jsme při postupu vpravo z význačné pozice vstoupili, nemohou být význačné, protože vlevo vedle nich leží 1.

Když si vše dáme dohromady, tak stačí matici A procházet po řádcích zleva doprava. V každém kroku jsme v jednom ze 3 stavů. Buď stojíme na nule, nebo na význačné pozici a nebo rozšiřujeme předchozí význačnou pozici doprava. V každém kroku děláme jen konstantně mnoho práce. Proto bude nalezení největší jedničkové podmatice trvat jen čas $\mathcal{O}(mn) + \mathcal{O}(mn)$. To je obdivuhodné zrychlení, ne?

1.6 Odstranění rekurze

Nejprve připomeňme základní znalost o fungování rekurze a předávání parametrů. Rekurze je realizována zásobníkem.[5] Všechny rekurzivně volané funkce se ukládají na zásobník. Na zásobníku se pro každou volanou funkci vytvoří záznam, který obsahuje informace o volané funkci a předané parametry. Po jejím skončení se tento záznam odebere ze zásobníku a program se podle předchozího záznamu (ten co je na vrcholu zásobníku) vrátí k předchozí funkci, a to do místa za rekurzivní volání právě proběhlé funkce.

Všechny parametry volané funkce se kopírují na zásobník a odtamtud je volaná funkce teprve využívá. Říká se tomu *předávání parametrů hodnotou.* Pokud bychom chtěli volané funkci předat pole, tak se nejprve celé pole zkopíruje na zásobník a

[5] Tak zvaný `STACK`. Většinou leží na začátku paměťového prostoru programu. Z druhé strany proti němu roste `HEAP` – paměť která je přidělována dynamicky alokovaným proměnným.

teprve pak s ním začne funkce pracovat. Abychom se vyhnuli zbytečnému kopírování, tak můžeme použít *předávání parametrů odkazem.* To funguje tak, že funkci předáme pouze ukazatel na předávané pole (adresu paměti, kde pole bydlí). Místo kopírovaní celého pole tedy stačí uložit na zásobník jeden ukazatel.

V některých případech můžeme rekurzi odstranit tím, že ji nahradíme jednoduchým cyklem.

Například funkci faktoriál naprogramovanou pomocí rekurze nahradíme for-cyklem. Ušetříme čas za volání funkce včetně předávání parametrů. Druhou výhodou je paměťová úspora. Zásobník, přes který se rekurzivní volání realizuje, zabere paměť $\mathcal{O}(n)$. Má to ještě jednu výhodu. Pro cyklus může překladač uplatňovat optimalizace (predikce podmínky), kdežto pro rekurzivní volání ne.

```
Faktorial(n):
   if n > 1 then
      return(n·Faktorial(n − 1))
   else
      return 1
```

```
Faktorial(n):
   fakt := 1
   for i := 2 to n do
      fakt := fakt · i
   return fakt
```

Ne vždy můžeme rekurzi nahradit cyklem. Co ale funguje vždy je nahrazení rekurze vlastním zásobníkem. Jak jsme si už vysvětlili, rekurze je už sama o sobě realizována zásobníkem. Na ten se ale ukládají úplně všechny funkce, které program rekurzivně volá. Každá rekurzivně volaná funkce má jiný počet různě velkých parametrů. Proto se na tento zásobník ukládá více informací, než je v některých případech potřeba. Z těchto a ještě i dalších důvodů můžeme realizací vlastního zásobníku něco ušetřit.[6]

Podobného efektu docílíme i správnou volbou programovacího jazyka, nepoužíváním věcí, které nejsou potřeba (například objektů).[7]

1.7 Odstranění opakujících se výpočtů

Fibonacciho číslo F_n je určeno rekurencí následovně: $F_0 = 0$, $F_1 = 1$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$. Fibonacciho čísla jsou 0, 1, 1, 2, 3, 5, 8, ... Pokud bychom naprogramovali funkci vracející n-té Fibonacciho číslo pomocí rekurze[8], tak přepsáním do for-cyklu dosáhneme podstatného zrychlení, které je způsobeno odstraněním opakujících se výpočtů.

```
Fib(n):
   if n > 1 then
      return(Fib(n − 1)+Fib(n − 2))
   else
      return n
```

```
Fib(n):
   F[0] := 0
   F[1] := 1
   for i := 2 to n do
      F[i] := F[i − 1] + F[i − 2]
   return F[n]
```

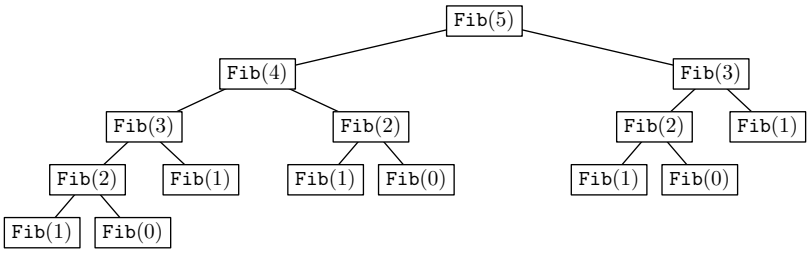
Na následujícím obrázku je strom větvení rekurzivního řešení. Zkuste si podle něj počítat.[9]

[6] Zkuste si například změřit, jak dlouho poběží quicksort naprogramovaný pomocí rekurze a jak dlouho poběží quicksort realizovaný pomocí zásobníku.

[7] To byste se divili, jak často studenti programují i triviální věci v objektech. Jakoby nic jiného neuměli. Napíší dvakrát tolik řádek zdrojového kódu, který je navíc pomalejší.

[8] To je typický odstrašující příklad.

[9] Kolega Kryl o efektivnosti algoritmů říká: „Zkuste si podle toho algoritmu počítat. A to doslova. Nevynechejte jediný příkaz. Pokud se při tom poblijete, tak to není efektivní.“.



Z obrázku vidíme, že jsme F_5 počítali jednou, F_4 také jednou, F_3 dvakrát, F_2 třikrát a F_1 pětkrát. Připomínají vám tyto čísla něco? Od toho pozorování už není daleko k tomu, abychom ukázali, že rekurzivní řešení má časovou složitost alespoň $\Omega(F_n)$. O Fibonacciho číslech je známo, že $F_n \approx \varphi^n$, kde $\varphi = \frac{1+\sqrt{5}}{2}$. Není těžké ukázat, že rekurzivní řešení má časovou složitost $\Theta(\varphi^n)$. O proti němu má nerekurzivní řešení jen lineální časovou složitost.[10] Dosáli jsme tedy exponenciálního zrychlení. A světe div se, nejlepší řešení s časovou složitostí $\mathcal{O}(\log n)$ je ještě exponenciálně krát rychlejší! (viz. cvičení).

Odstranění opakujících se výpočtů tím, že si je uložíme do tabulky, je jedna ze základních myšlenek *dynamického programování* (viz kapitola **??**).

1.8 Optimalizace pro hardware a operační systém

Fungování hardwaru je jedna velká magie.

Tuto optimalizaci nejčastěji provádíme tím, že optimalizujeme zdrojový kód. Příkazy a možnosti překladače uvedené v této sekci fungují například překladači `gcc`.

Často bereme počítač jen jako černou skříňku, která dobře počítá, a nezajímáme se o to, jak funguje vevnitř. Pokud ale chceme psát optimální kód, tak se neobejdeme bez hlubších znalosti chování hardwaru a fungování operačního systému. Jinak se nám může stát, že budeme naší černé skříňce házet klacky pod nohy a ona bude muset uvnitř pracovat tím nejsložitějším a taky nejpomalejším způsobem.

Nejprve bychom si měli rozmyslet, jestli je optimalizace kódu dané části programu opravdu nutná. Optimalizace totiž často svádí programátory k takzvaným „prasárnám“, které dokonale znečitelňují zdrojový kód. Nečitelný kód může pěkně potrápit další čtenáře, nebo po pár týdnech i nás samotné. Navíc se zvyšuje riziko, že někde uděláme chybu.

Optimalizace zdrojového kódu má smysl, pokud je daný kus kódu „úzkým hrdlem (bottle neckem)“ celého programu a pokud zvládneme kód optimalizovat lépe než překladač.[11] Přiznejme si, že dnešní překladače jsou na optimalizace profíci. Je těžké být lepší. Optimalizace kódu se většinou píší přímo v Assembleru. Často dosahují zrychlení tím, že využijí i specializované instrukce procesoru/grafického procesoru, které se běžně nepoužívají (případně počítají paralelně – buď na nezávislých obvodech téhož procesoru a nebo klidně i na více procesorech, CPU, GPU – graphical processing unit, ...).

[10] Poznamenejme, že u nerekurzivního řešení můžeme ještě snížit paměťovou složitost, protože si stačí místo celého pole pamatovat poslední dvě hodnoty.

[11] Například překladač `gcc` už dělá řadu základních optimalizací sám od sebe. Úroveň optimalizací můžete nastavit pomocí parametru na vstupu.

1.8.1 Jak to funguje uvnitř počítače?

Jak to zhruba funguje uvnitř černé skříňky? Procesor funguje na určité frekvenci. Frekvence procesoru je počet taktů za vteřinu. Každý procesor má svoji instrukční sadu. To jsou příkazy (instrukce), které jsou „zadrátované“ do elektrických obvodů. Každá instrukce trvá jiný počet taktů. Některé instrukce trvají jeden takt, jiné třeba až 150 taktů. Aby to nebylo jednoduché, tak má každý typ procesoru jinou instrukční sadu. Některé instrukce mohou být společné pro více typů procesorů, ale pro změnu mohou trvat jiný počet taktů (a to i výrazně).

Procesor potřebuje komunikovat s pamětí. Rychlá paměť je drahá a proto je paměť rozdělena do hierarchie podle klesající rychlosti, ale rostoucí velikosti: registry procesoru, cache procesoru, rozšířená paměť,[12] paměť na pevném disku. První dvě paměti jsou pro běžného programátora skryté, ale jsou výrazně rychlejší než rozšířená paměť.[13] Rozšířená paměť je zase výrazně rychlejší než pevný disk.

Procesor pracuje pouze s registry a s cache. Pokud potřebuje proměnnou z rozšířené paměti, tak si ji nejprve nechá nahrát do cache (v cache se vytvoří kopie) a tam s ní pracuje (s rychlejším přístupem). Často se stane, že s proměnnou pracuje více než jednou. Procesor pracuje pouze s lokální kopií v cache. Hodnota odpovídající proměnné v rozšířené paměti se zatím neaktualizuje. Aktualizace rozšířené paměti proběhne až v momentě, kdy chceme její lokální kopii v cache zrušit a uvolnit.

Cache často funguje na principu LRU (Least Recently Used). To znamená, že pokud už je cache plná a další proměnná se do ní nevejde, tak poslední dobou nejméně používaná proměnná uvolní místo nově příchozí proměnné. Při vymazání proměnné z cache se přepíše její aktuální hodnota zpátky do externí paměti.[14] [15]

Překladač sám provádí určité optimalizace kódu. Ovšem nic není dokonalé. Překladač se snaží pochopit strukturu kódu a když najde něco, co umí vylepšit, tak to vylepší. Stále se ale vyplatí optimalizovat si časově náročné části sám. Přeci jen toho o fungování algoritmu víme více než překladač, který to musí rozpoznat či odhadnout ze zdrojového kódu. Na druhou stranu použitím vlastních optimalizací můžeme zablokovat optimalizace, které mohl provést překladač (zneprůhlednili jsme mu tím kód) a ve výsledku můžeme dostat pomalejší kód.

To je vše, co si k fungování černé skříňky řekneme. Bližší zájemce odkazuji na literaturu o operačních systémech, hardwaru a překladačích.

Jestli ale nevíte, jak to na vašem počítači s vaším překladačem vašeho oblíbeného programovacího jazyka opravdu funguje, tak si s tím hrajte. Experimentujte. Napište si vlastní prográmek, ve kterém si změříte, jak dlouho které příkazy trvají. (Dobrý programátor by měl vědět, jak dlouho trvá sčítání oproti násobení. Jak pomalé je čtení z disku, apod.)

1.8.2 Zásady pro psaní efektivního kódu

Ačkoliv je situace složitá, je pár zásad, kterých se můžeme držet, abychom psali efektivní kód. V následujícím textu používáme slovo „drahý“ ve významu časově náročný.

- **Přístup do paměti je drahý.** Měli bychom se snažit vejít do cache. Případně bychom měli nejprve dokončit práci s proměnnými, které už jsou v cache, a pak se teprve vrhnout na nová data. Například pokud pracujeme s hodně

[12] Rozšířené paměti běžně říkáme „RAM“.

[13] Třeba až desetkrát.

[14] Při optimalizaci kódu můžeme speciálními příkazy určit, které proměnné se mají/nemají ukládat do cache.

[15] Pokud píšeme kód v assembleru, tak můžeme určovat přiřazení registrů proměnným. Běžně to za nás udělá překladač. A musíme uznat, že to dělá dobře.

dlouhým polem, tak je rychlejší dělat více práce na malých kusech, než vícekrát procházet celé pole.

Varování: V předchozích doporučeních jsme uváděli slova jako velký, malý apod. Ta správná velikost není univerzální, ale je na každém počítači jiná. Proto je nejlepší experimentovat s různými hodnotami parametrů a vybrat tu optimální. Další připomínkou je, že některá zlepšení se vyplatí až na hodně velkých datech.

Pokud potřebujeme vynulovat či překopírovat velký kus paměti, tak je lepší použít funkce k tomu určené (`memset`, `memcopy`).

Pokud se vše nevejde do cache, tak může být rychlejší, když si jednoduché věci dopočítáme znova, než když je hledáme v rozšířené paměti.

- **Zapisování a čtení ze souboru je drahé.** Připomeňme, že vypisování na obrazovku se chová stejně jako zápis do souboru a tudíž, je také celkem drahé. Hodně pomůže zavedení bufferů. Potom stačí zapisovat do souboru méně krát a po větších kusech. To vede k výraznému zrychlení. Také můžeme využít možnosti, namapovat si kus souboru do paměti, a s ním teprve pracovat.

- **Podmínky jsou drahé.** Aby procesor fungoval rychleji, tak si některé věci předpočítává. Například ví, že o pár instrukcí dál bude potřebovat určitou proměnnou, tak si ji už dopředu začne nahrávat do cache.[16] Pokud je ale následující nezpracovaná instrukce podmínka (větvení programu), tak se dopředu neví, jak dopadne a tudíž nevíme, co si předpočítat. Procesor se často připravuje na obě varianty a po vyhodnocení podmínky nepotřebnou variantu zahodí. Tím promarní část svého času.

  Příklad: Máme proměnnou a obsahující buď 0 nebo 1. Pokud potřebujeme změnit její hodnotu na opačnou, tak se dá podmínka s kódem nalevo přepsat na kód napravo.[17]

  `if` $a = 1$ `then` $a := 0$ `else` $a := 1$ $\qquad\qquad$ $a := 1 - a$

- Registry procesoru jsou dnes 64-bitové, proto je lepší zpracovávat data po 64 bitech a ne po menších dílech. Toho se dá využít například při zpracovávání řetězců, obrázků a dalších médií.

  Z podobných důvodů se paměť zarovnává na 64 bitů. To znamená, že první bit proměnné začíná na bitu v paměti, který je dělitelný 64. Pro jednoduchost si můžeme představit, že paměť funguje jako pole 64-bitových položek. Pokud bychom chtěli přistoupit k nezarovnané 64-bitové proměnné, tak budeme muset přečíst dvě políčka pole a z nich si vytáhnout potřebné bity.

- **Některé aritmetické operace jsou drahé.** Například dělení. Dělení mocninou dvojky, se dá obejít bitovým posunem doprava (bitová operace shift doprava). Dělení známou konstantou už ale stejným způsobem optimalizuje překladač. Problém nastává, když dělíme dopředu neznámou proměnou. To se při kompilaci programu optimalizovat nedá.

  Celkově je dobré se dělení vyhnout. Například místo výpočtu hashovací funkce modulo prvočíslo, kde se prvočíslo spočítá až za běhu programu a uloží do proměnné, si můžeme dopředu zjistit všechna prvočísla, které budeme používat. Při výpočtu hashovací funkce použijeme `switch`, kterým se podle prvočísla

[16] Leckdy natažení proměnné do cache trvá déle než samotný výpočet. Ovšem záleží na tom, co s proměnnou počítáme. Některé zběsilé výpočty (instrukce) trvají mnohem déle než natahování proměnné do cache.

[17] V takto jednoduchých případech to za nás udělá optimalizátor překladače.

rozskočíme na optimalizovanou rutinu modulení (modulo konstanta – optimalizace při dělení konstantou už provede překladač).

- K zajímavým trikům patří **použití zarážky.** Například když procházíme pole velikosti N a hledáme hodnotu x, tak běžně v podmínce cyklu kontrolujeme dvě věci: jestli index aktuálního políčka nepřekročil N a jestli není hodnota aktuálního políčka rovná x. Hledání můžeme zrychlit tím, že si do pole na pozici $N+1$ uložíme x, které bude fungovat jako zarážka. Potom máme jistotu, že x vždy najdeme, a můžeme podmínku v cyklu zjednodušit na test, jestli není hodnota aktuálního políčka rovná x. Až x najdeme, tak se podíváme, na kterém indexu leží. Podle toho zjistíme, jestli jsme x našli a nebo jen neúspěšně došli až na konec pole.

- Mezi **další triky** patří použití `inline` funkcí, nahrazení jednoduchých funkcí makrem (např. pro výpočet minima, maxima, absolutní hodnoty apod.)

1.9 Spousta dalších možností

Moto: „Vše co se učíme, se učíme tím, že to děláme.“

Řešte příklady z KSP (korespondenční seminář z programování), programátorských olympiád či soutěží ACM programming contest a učte se tím nové finty a triky. Jejich zadání najdete na webu. Můžete ho řešit i mimo soutěž, prostě jen pro radost. U prvních dvou najdete na webu i vzorové řešení.

1.10 Příklady

1. (Vyhodnocování polynomu) V sekci o výpočtu hodnoty na základě předchozí jsme si ukázali, jak rychle spočítat hodnoty polynomu $P(x) = ax^2 + bx + c$ ve všech bodech $x \in \{1, \ldots, n\}$. Potřebovali jsme je znát například proto, abychom si nakreslili graf funkce. Ale co kdybychom si chtěli nakreslit přesnější graf funkce $P(x)$? Co kdybychom chtěli znát hodnotu $P(x)$ ve všech bodech $0, 0.1, 0.2, 0.3, \ldots, n$? Jak to co nejrychleji spočítat?

2. (Výpočet hodnoty na základě předchozí) Je dána posloupnost celých čísel. Délka posloupnosti není známa. Posloupnost může být i tak dlouhá, že se nevejde do paměti, ale jen na pevný disk.

   (a) (Úsek posloupnosti s největším součtem) Nalezněte v posloupnosti souvislý úsek s největším součtem. Výstupem algoritmu bude jen hodnota největšího součtu. Například pro posloupnost 1 3 −1 1 −5 8 2 −3 4 −8 bude výsledkem 11 (to je délka úseku 8 2 −3 4).

   (b) (Nejdelší hladký úsek posloupnosti) Určete délku nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší pouze o 1. Například pro posloupnost 5 7 6 7 7 8 8 7 9 9 9 bude výsledkem 5 (to je délka úseku 7 7 8 8 7).

   (c) (Nejdelší D-hladký úsek posloupnosti) Určete délku nejdelšího souvislého úseku, v němž se libovolná dvě čísla liší pouze o D.

   Nápověda: existuje řešení s časovou složitostí $\mathcal{O}(n)$.

3. (Přímé generování výsledku) Množina Q obsahuje pouze taková přirozená čísla, která nejsou dělitelná ani jedním z prvočísel 2, 3 a 5.[18] Vypište co nejrychleji n-té nejmenší číslo množiny Q.

   (a) Zkusme si to nejprve zjednodušit. Co kdybychom chtěli vypsat n-té nejmenší přirozené číslo, které není dělitelné 2? Je to moc jednoduché? Tak jak co nejrychleji vypsat n-té nejmenší přirozené číslo, které není dělitelné 2 ani 3?

   (b) Vyřešte úlohu pro zadaná prvočísla 2, 3 a 5. Umíte odpovědět v čase $\mathcal{O}(1)$? Pokud ano, tak ještě dokažte, že je odpověď správně. To je, že jste žádné číslo množiny Q nevynechali.

   (c) Zobecněte řešení úlohy pro k různých prvočísel.

4. (Rozklad čísla na součet třetích mocnin) Rozložte číslo $n \in \mathbb{N}$ na součet dvou třetích mocnin přirozených čísel. Jinými slovy najděte $a, b \in \mathbb{N}$ splňující $n = a^3 + b^3$. Vypište všechna řešení.

   (a) Někdo by mohl najít následující řešení. Procházíme všechna možná $a \in \{0, 1, \ldots, \lfloor \sqrt[3]{n} \rfloor\}$ a pro každé a spočteme $\sqrt[3]{n - a^3}$. Pokud bude výsledek celočíselný, tak jsme nalezli řešení. Toto řešení je samozřejmě správně a má časovou složitost $\mathcal{O}(\sqrt[3]{n})$.

   Zkuste ale vymyslet řešení, ve kterém budeme počítat pouze s celočíselnými proměnnými a vystačíme si se sčítáním a násobením. Dá se vymyslet řešení s časovou složitostí $\mathcal{O}(\sqrt[3]{n})$.

   (b) Obě řešení naprogramuje a porovnejte, jak rychle počítají pro konkrétní n. Vyvoďte z toho závěr, jestli je opravdu výhodnější počítat v celočíselných proměnných, nebo jestli to vyjde stejně, jako když počítáme v plovoucí čárce a navíc počítáme funkce jako je odmocnina.

5. (Kružnice) Na kružnici je rozmístěno n bodů. Vzdálenost dvou bodů budeme měřit po obvodu kružnice. Vzdálenosti libovolných dvou bodů jsou celočíselné. Začneme v nejvyšším bodě a body si očíslujeme čísly 1 až n po směru hodinových ručiček. Pro každé dva sousední body (v pořadí podle očíslování) dostanete jejich vzájemnou vzdálenost. Například pro kružnici s 5 body, můžete dostat vzdálenosti $d(1,2) = 1$, $d(2,3) = 4$, $d(3,4) = 2$, $d(4,5) = 5$, $d(5,1) = 2$.

   (a) Rozhodněte, jestli na kružnici existují dva body takové, že jejich spojnice prochází středem kružnice. Pokud ano, tak takovou dvojici bodů vypište. V příkladu kružnice s 5 body bychom mohli odpovědět $(1,4)$, ale také $(3,5)$, protože jejich spojnice prochází středem kružnice.

   (b) Rozhodněte, jestli na kružnici existují čtyři body takové, že tvoří čtverec. Pokud ano, tak takové 4 body vypište.

   Nápověda: Existuje řešení s časovou složitostí $\mathcal{O}(n)$.

6. (Fibonacciho čísla přes mocnění matic) Fibonacciho čísla jsou 0, 1, 1, 2, 3, 5, 8, ... Jsou definovány rekurencí $F_0 = 0$, $F_1 = 1$ a $F_n = F_{n-1} + F_{n-2}$ pro $n \geq 2$. Rovnost $F_1 = F_1$ a $F_2 := F_1 + F_0$ můžeme zapsat i pomocí matice:

[18] Pozor na odlišnost od ukázkové úlohy ze sekce o přímém generování výsledku. Tam jsme vyžadovali, aby číslo $x \in M$ bylo dělitelné pouze prvočísly 2, 3 a 5. Tady vyžadujeme, aby nebylo dělitelné ani jedním z prvočísel 2, 3 a 5. Není pravda, že $Q = \mathbb{N} \setminus M$, protože například číslo $21 = 7 \cdot 3$ nepatří ani do jedné množiny.

$$\begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Podobně

$$\begin{pmatrix} F_2 \\ F_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_1 \\ F_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

až zobecněním dostaneme

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}.$$

Označme tuto matici tvaru 2×2 jako A. Pro výpočet n-tého Fibonacciho čísla tedy stačí vymyslet, jak se dá rychle spočítat n-tá mocnina matice A.

(a) Ukažte, že n-tá mocnina libovolné matice B velikosti 2×2 se dá spočítat v čase $\mathcal{O}(\log n)$.

Nápověda: Zamyslete se nad tím, jak byste počítali B^2, B^4, B^8.

(b) Jak rychle dokážete spočítat číslo X_n, které je určeno následující rekurencí: $X_0 = a$, $X_1 = b$, $X_2 = c$ a $X_n = dX_{n-1} + eX_{n-2} + fX_{n-3}$ pro $n \geq 3$. Čísla a, b, c, d, e, f jsou pevně dané konstanty.

(c) Fibonacciho čísla se dají spočítat i podle následujícího vzorce:

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

Ukažte, jak tento vzorec souvisí s vlastními čísly matice A.

Nebylo by rychlejší počítat Fibonacciho čísla přímo z tohoto vzorce než přes mocnění matic? Nebylo. Čísla ve vzorci jsou iracionální a jejich výpočet s dostatečnou přesností by byl stejně pracný jako mocnění matic. Vyzkoušejte si to naprogramovat.[19]

Nápověda: Jaká jsou vlastní čísla matice A?

Poznámka: Když si vzorec vyčíslíte, tak zjistíte, že F_n je přibližně $0,44 \cdot (1,61^n + (-0,61)^n)$. Proto můžeme F_n počítat jen jako $0,44 \cdot 1,61^n$ a výsledek zaokrouhlit na nejbližší celé číslo.

7. (Počítání prvních n prvočísel) Napište program, který vypíše na obrazovku prvních $N := 1000$ prvočísel. U každého přístupu odhadněte časovou složitost.

(a) Procházejte postupně čísla 1, 2, 3,... (procházení kandidátů) a aktuálně testované číslo k zkoušejte dělit všemi čísly 2,3,..., $\sqrt{k}$ (testování prvočíselnosti).

(b) Jak můžeme předchozí výpočet zrychlit? Ukládejte si nalezená prvočísla do paměti a při testování prvočíselnosti zkoušejte dělit testované číslo k pouze dosud nalezenými prvočísly.

(c) Jak to urychlit ještě více? Můžeme předem zamítnout některé kandidáty. Například všechna sudá čísla kromě 2 určitě nebudou prvočísly. Jak se dají rychle generovat kandidáti, kteří nejsou dělitelní 2 ani 3?

[19] V algebře se můžete dozvědět, že počítání s celými čísly, ke kterým přidáme iracionální číslo tvaru $\sqrt{a}$ můžeme nahradit počítáním s maticemi velikosti 2×2. Hovoříme o okruhu celých čísel a okruhu celých čísel rozšířeném o $\sqrt{a}$.

Pro odhad časové složitosti se vám bude hodit vědět, že počet prvočísel menších než n je $\pi(n) \approx n/\ln n$.

8. (Největší společná podmatice) Dostaneme dvě matice A a B obsahující přirozená čísla.

   (a) (největší společná podmatice ležící na stejné pozici) Předpokládejte, že obě matice mají stejnou velikost $n \times m$. Najděte největší obdélník určený levým horním a pravým dolním rohem takový, aby podmatice matic A a B určené tímto obdélníkem obsahovaly stejná čísla.

   (b) (největší společná podmatice ležící na libovolných pozicích) Tentokrát mohou mít matice A a B různé velikosti. Najděte rozměry obdélníka s největším možným obsahem takové, aby matice A, B obsahovaly podmatice těchto rozměrů, které jsou stejné.

   Rozmyslete si, o kolik složitější by bylo zároveň vypsat i polohy společných podmatic.

9. (Podmatice s největším součtem) Dostanete matici A velikosti $n \times m$ obsahující přirozená čísla. Najděte podmatici s největším součtem.