Rozděl a panuj

1 Základní princip

Technika rozděl a panuj je založená na následující myšlence. Z dané vstupní instance I vygenerujeme několik menších instancí I_k stejného problému takovým způsobem, že jsme schopni, pokud známe řešení instancí I_k, sestavit řešení původní instance I. Přitom pro nalezení řešení k instancím I_k použijeme rekurzivně stejný postup. Rekurzi zastavíme v momentě, kdy jsou instance natolik malé, že pro ně známe řešení, nebo je jejich řešení možno rychle spočítat jiným algoritmem.

Příklad 1. Přístup můžeme demonstrovat na problému vypočtení n-tého Fibonnaciho čísla. Toto číslo budeme značit jako F(n). Připomeňme si, že toto číslo je definováno následujícím rekurentním vztahem.

$$F(n) = \begin{cases} F(n-1) + F(n-2) & n \geq 2 \\ n & n \in \{0, 1\} \end{cases} \tag{1}$$

Předchozí vztah ihned nabízí sestavení algoritmu přístupem rozděl a panuj. Je-li n rovno 0 nebo 1, výsledek známe. Jinak spočítáme $F(n)$ tak, že nejprve spočteme $F(n-1)$ a $F(n-2)$ a poté tato čísla sečteme. Tedy, vstupní instancí I našeho algoritmu je číslo n. Z toho čísla vygenerujeme dvě menší instance $n-1$ a $n-2$. Spočítáme pro ně výsledky, a řešení pro původní instanci, dostaneme jednoduchým sečtením. Malé instance, pro které již známe výsledek, jsou v tomto případě čísla 0 a 1. □

V algoritmech realizujících techniku Rozděl a panuj můžeme rozpoznat dvě oddělené fáze, které ji dali název. Během první fáze, které říkáme *Rozděl*, algoritmus vygeneruje množinu menších instancí. Pojmenování fáze naznačuje, že původní instanci rozdělíme na několik menších. I když je označení *Rozděl* nepřesné, protože instanci nemusíme přímo rozdělit, toto označení se zažilo (+ existuje očividná analogie s pořekadlem o panovnících). Během druhé fáze, pojmenované jako *Panuj*, sestavíme ze řešení vypočtených pro menší instance řešení instance původní.

Příklad 2. `Mergesort` je algoritmus určený pro třídění prvků porovnáváním, který znáte z ALM2. V tomto algoritmu ve fázi rozděl vstupní instanci skutečně rozdělujeme (přibližně na poloviny). Pro jednoduchost předpokládejme, že chceme setřídit pole A, který obsahuje celá čísla. Algoritmus uvádíme jako Algoritmus 1, kde je rozepsán do dvou procedur. V proceduře MergeSort, jejíž cílem je setřídit část pole A omezenou indexy l a p, nejdříve na řádku 2 ověříme, zda-li už není instance triviální, to jest zda-li nemá $A[l] \dots A[p]$ méně než dva prvky. Pokud ano, můžeme tuto část pole považovat za setříděnou. Na řádku 3 probíhá fáze *Rozděl*, kdy část pole k setřízení rozdělíme na dvě části (jejichž velikost se liší maximálně o jeden prvek). Na následujících dvou řádcích poté setřídíme obě tyto části stejným postupem (tedy pomocí MergeSort). Na řádku 7 pak setříděné části slijeme pomocí procedury Merge. To odpovídá fázi *Panuj*. □

Nyní si již napišme obecný předpis pro algoritmus používající techniku rozděl a panuj.

```
1:  procedure Divide-and-Conquer(I)
2:      if |I| ≤ c then                          ▷ Jeli I dostatečně malé (menší než konstanta c),
3:          return BasicAlgoritm(I)              ▷ použijeme jiný algoritmus
4:      end if
5:      Vytvoř instance I_1 ... I_k menší velikosti než I
6:      for i ← 1 to k do
7:          x_i ←Divide-and-Conquer(x_i)         ▷ x_i je řešení instance I_i
8:      end for
9:      Zkombinuj x_1 ... x_k do x               ▷ x je řešení instance I
10:     return x
11: end procedure
```

Algoritmus 1 Mergesort

```
1: procedure MergeSort(A, l, p)
2:     if p < l then
3:         q ← ⌊(l + p)/2⌋                ▷ rozděl
4:         MergeSort(A, l, q)
5:         MergeSort(A, q + 1, p)
6:     end if
7:     Merge(A, l, q, p)                  ▷ panuj
8: end procedure
```

```
1: procedure Merge(A, l, q, p)
2:     n1 ← q − l + 1
3:     n2 ← r − q
4:     for i ← 0 to n1 − 1 do
5:         L[i] ← A[l + 1]
6:     end for
7:     for i ← 0 to n1 − 1 do
8:         R[i] ← A[q + i + 1]
9:     end for
10:    L[n1] ← R[n2] ← ∞
11:    i ← j ← 0
12:    for k ← l to p do
13:        if L[i] ≤ R[j] then
14:            A[k] ← L[i]
15:            i ← i + 1
16:        else
17:            A[k] ← R[j]
18:            j ← j + 1
19:        end if
20:    end for
21: end procedure
```

2 Řešení rekurencí

Analýza složitosti algoritmů rozděl a panuj vede k rekurencím. Z pohledu celého algoritmu Divide-and Conquer je složitost BasicAlgorithm brána jako konstantní, voláme ho totiž vždy pro instance, které mají shora omezenou velikost. Označíme-li si složitost Divide-and-Conquer jako $T(n)$, pak ji lze vyjádřit rekurentně jako

$$T(n) = \sum_{i=1}^{k} T(n_i) + f(n).$$

s okrajovou podmínkou

$$T(a) = O(1),$$

pro všechna $a \leq c$. Okrajové podmínky v rekurencích se často neuvádí. Pokud je tomu tak, většinou předpokládáme, že má tvar

$$T(1) = 1.$$

Při konkrétní práci s rekurencí pak můžeme, za určitých podmínek, jedničku na pravé i na levé straně změnit na jinou konstantu. Číslo n_i ve výrazu je velikost instance I_i (v konkrétních rekurencích bývá n_i vyjádřena funkcí závislou na n) a $f(n)$ zachycuje složitost vytvoření menších instancí (řádek 5) a kombinace řešení těchto instancí (řádek 9).

Cílem této kapitoly je ukázat několik základních metod řešení takových rekurencí. Řešením rekurence je její uzavřená forma, tj. vyjádření, které neobsahuje rekurentní "volání sama sebe" na pravé straně rovnosti. To, že budeme řešení odhadovat znamená, že výsledkem nebude přesné znění uzavřené formy, ale její asymptotický odhad.

2.1 Substituční metoda

První metoda, kterou si ukážeme je *metoda* substituce. Její princip spočívá ve dvou krocích

1. Odhadneme asymptotický odhad uzavřené formy
2. Vybereme jednu funkci z toho odhadu a indukcí se pokusíme dokázat správnost.

Postup si ukážeme na příkladu. Uvažujme rekurenci

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1. \tag{2}$$

Odhadneme, že řešením je lineární funkce, tedy že odhadem je $O(n)$. Zvolíme jednoho reprezentanta z $O(n)$, např. $cn - b$ pro nějaké konstanty c, b. Nyní musíme indukcí dokázat, že pro vhodné konstanty c a b platí, že $T(n) \leq cn - b$. Předpokládejme tedy, že platí $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor - b$ a $T(\lceil n/2 \rceil) \leq c\lceil n/2 \rceil - b$. Po dosazení zpět do rekurence dostaneme

$$T(n) \leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = cn - 2b + 1 \tag{3}$$

Výraz $cn - 2b + 1$ tedy rozdělíme na sumu výrazu, který chceme obdržet a zbytku, tzv. residentu

$$cn - 2b + 1 = cn - b + (-b + 1).$$

Vidíme, že aby rovnost platila, musí se b rovnat 1 (zvolíme tak totiž jako řešení $cn - 1$ a na konci nerovnosti (3) dostaneme také $cn - 1$, čímž dokončíme důkaz indukčního kroku). Pro potřeby důkazu indukcí zbývá ověřit, že nerovnost platí pro okrajovou podmínku $T(1) = 1$. Pokud do ní dosadíme, dostaneme nerovnost

$$T(1) = 1 \leq c - 1,$$

která platí pro $c \geq 2$. Indukcí jsme tedy dokázali, že řešení rekurence leží v $O(n)$.

Protože se v substituční metodě nachází několik voleb, které se zdají být těžké (např. jak uhodnout odhad řešení rekurence, kterou funkci z asymptotického odhadu vybrat pro důkaz indukcí) záleží schopnost použít substituční metodu na zkušenostech, citu a ochotě experimentovat. Přesto si uvedeme několik triků, které mohou být užitečné.

Často se stane, že problém, na který narazíme při důkazu indukcí nám napoví, jak lépe zvolit reprezentanta z asymptotického odhadu. Uvažme opět rekurenci (2), tentokrát však z $O(n)$ zvolíme funkci cn. Po dosazení indukčního předpokladu do rekurence pak získáme

$$T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1$$

Vidíme, že v residentu se nevyskytuje žádná konstanta a důkaz tedy nelze dokončit. Potřebujeme tedy dostat do residentu konstantu, jejímž vhodným nastavením bychom jedničku odečetli. To můžeme provést tak, že místo cn zvolíme $cn - b$ jako v předchozím případě.

Pokud se stane, že jsme dokázali nerovnost pro obecný indukční krok, ale pro okrajovou podmínku naše řešení nefunguje, můžeme to vyřešit tím, že okrajovou podmínku vhodně změníme. Ukážeme si na příkladu. Uvažme rekurenci

$$T(n) = 2T(\lfloor n/2 \rfloor) + n. \tag{4}$$

Správný odhad je $O(n \lg n)$. Pokud vybereme funkci $cn \lg n$ můžeme dokázat správnost obecného indukčního kroku pro $c \geq 1$

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &\leq cn \lg n \end{aligned}$$

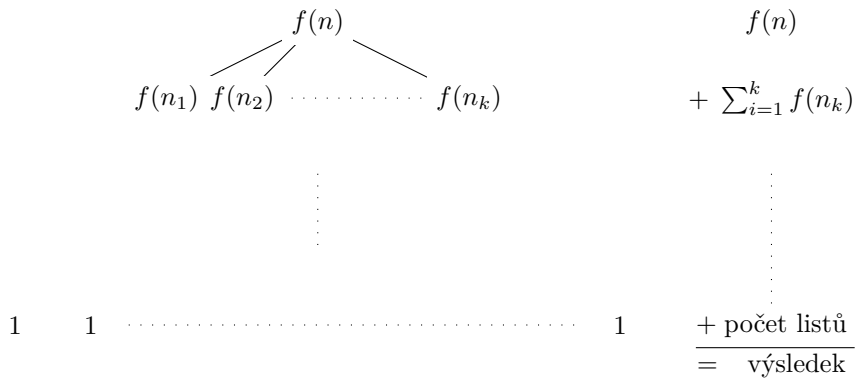
U okrajové podmínky ovšem narazíme na problém.

$$T(1) = 1 \neq 1 \lg 1 = 0$$

Můžeme využít toho, že dokazujeme asymptotický odhad a nerovnost proto může platit až od určitého n_0. Pro $n > 3$ už (4) nezávisí na $T(1)$. Volbou $n_0 = 2$ můžeme „posunout“ okrajovou podmínku směrem nahoru (čímž případ $n = 1$, který způsobuje problémy, z důkazu vypustíme) a nahradit okrajovou podmínku pomocí

$$\begin{aligned} T(2) &= 2, \\ T(3) &= 5, \end{aligned}$$

s tím, že nyní musí být $c \geq 2$.



Obrázek 1: Idea metody odhadu pomocí stromu rekurze

2.2 Odhad pomocí stromu rekurze

Metoda popsaná v této kapitole je užitečný nástroj jak odhadnout asymptotické omezení dané rekurence. Neměli bychom zapomenout, že je nutné ještě dokázat správnost odhadu pomocí substituční metody.

Metoda odhadu pomocí stromu rekurze je založená na jednoduché myšlence. Rekurentní vztah

$$T(n) = \sum_{i=1}^{k} T(n_i) + f(n).$$

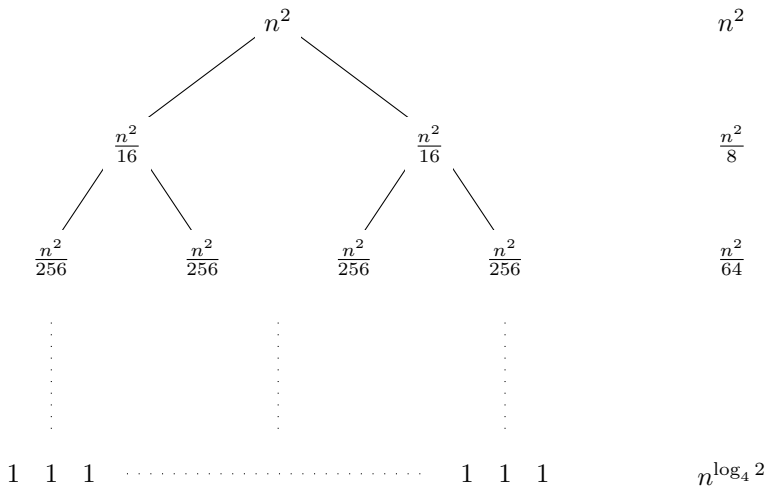
si představíme jako strom. Uzly stromu odpovídají jednotlivým rekurzivním „voláním." Každému uzlu přiřadíme množství práce, kterou pomyslný algoritmus při daném volání provede. Listové uzly tak budou mít přiřazenu konstantu, která odpovídá okrajovým podmínkám rekurentního vztahu. Množství práce vnitřních uzlů pak odpovídá aplikaci funkce f na argument daného rekurzivního volání. V kořenu tedy provedeme $f(n)$ práce, v jeho i-tém potomku pak $f(n_i)$. Přirozeně, pokud sečteme práci přiřazenou všem uzlům, dostaneme řešení rekurence. Součet provedeme ve dvou krocích. Nejdříve pro každou úroveň stromu sečteme práci provedenou v uzlech na oné úrovni, poté sečteme sumy z jednotlivých vrstev.

Metodu si opět předvedeme na příkladu. Uvažujme rekurenci

$$T(n) = 2T(n/4) + n^2. \tag{5}$$

Začněme konstruovat strom rekurze. Kořenu přiřadíme n^2. Potomkům kořene, kteří odpovídají $T(n/4)$ přiřadíme $(n/4)^2$. Uzlům v další vrstvě, odpovídajícím $T(n/16)$ (rekurzivní volání pro (5) s argumentem $n/4$) přiřadíme $(n/16)^2$. V první vrstvě se nachází 2 uzly, suma práce je tedy $2(n/4)^2$, v druhé vrstvě se nachází 4 uzly, suma je tedy $4(n/16)^2$. Nyní si můžeme všimnout určité pravidelnosti. Práce, kterou provedeme v jednom uzlu v i-té vrstvě (kořen je v nulté vrstvě) odpovídá $(n/4^i)^2$, počet uzlů v i-té vrstvě je 2^i, suma přes všechny uzly v této vrstvě je tedy $2^i(n/4^i)^2 = n^2/8^i$. Listy se nacházejí ve vrstvě $\log_4 n$, jejich počet je tedy $2^{\log_4 n} = n^{\log_4 2} = n^{1/2}$. Pokud nyní sečteme všechny vrstvy, dostaneme

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_4 n - 1} (n^2/8^i) + n^{1/2} \\ &= n^2 \sum_{i=0}^{\log_4 n - 1} (1/8^i) + n^{1/2} \end{aligned}$$



Obrázek 2: Strom rekurze pro rekurenci (5).

Abychom se zbavili závislosti na n v sumě, nahradíme ji sumou celé geometrické posloupnosti. Dostaneme tedy

$$\begin{aligned} T(n) &= n^2 \sum_{i=0}^{\log_4 n-1} (1/8^i) + n^{1/2} \\ &\leq n^2 \sum_{i=0}^{\infty} (1/8^i) + n^{1/2} \\ &= n^2 \frac{1}{1-1/8} + n^{1/2} \end{aligned}$$

Odhad řešení rekurence je tedy $O(n^2)$.

2.3 Master theorem

Master theorem je jednoduchý návod pro nalezení odhadu řešení rekurencí, které se vyskytují při analýze algoritmů, které ve fázi rozděl dělí vstupní instanci na několik stejně velkých podinstancí.

Věta 1. *Nechť $a \geq 1$ a $b \geq 1$ jsou konstanty a $f(n)$ je funkce a $T(n)$ je definovaná na nezáporných celých číslech pomocí rekurence*

$$T(n) = aT(n/b) + f(n),$$

přičemž n/b interpretujeme jako $\lfloor n/b \rfloor$ nebo $\lceil n/b \rceil$. Pak můžeme $T(n)$ následovně asymptoticky omezit.

1. *Pokud $f(n) = O(n^{\log_b a - \epsilon})$ pro $\epsilon \geq 0$, pak $T(n) = \Theta(n^{\log_b a})$.*
2. *Pokud $f(n) = \Theta(n^{\log_b a})$ pak $T(n) = \Theta(n^{\log_b a} \lg n)$.*
3. *Pokud $f(n) = \Omega(n^{\log_b a + \epsilon})$ pro $\epsilon \geq 0$ a pokud $af(n/b) \leq cf(n)$ pro konstantu $c < 1$ a všechna dostatečně velká n, pak $T(n) = \Theta(f(n))$.*

3 Příklady algoritmů

3.1 Nalezení dvojice nejbližších bodů na ploše

Je dána množina bodů $P = \{p_1, p_2, \dots\}$, kde bod p_i je dán souřadnicemi $\langle x_i, y_i \rangle$. Pro libovolné dva body $p_i, p_j \in P$ je jejich vzdálenost $d(p_i, p_j)$ definována jako

$$d(p_i, p_j) = \sqrt{|x_i - x_j|^2 - |y_i - y_j|^2}.$$

Cílem je nalézt dvojici různých bodů $p_i, p_j \in P$, jejichž vzdálenost je nejmenší mezi všemi dvojicemi bodů z P, jinými slovy takové p_i, p_j, že $d(p_i, p_j) = \min\{d(p_l, p_k) \mid p_l \neq p_k;\ p_j, p_k \in P\}$.

Pro navržení algoritmu pro tento problém se dá výhodně použít techniky rozděl a panuj. Algoritmus je založen na následující myšlence. Pro množinu bodů P

1. Rozdělíme body vertikální čarou na poloviny (v případě lichého počtu bodů má levá polovina o jeden bod více).
2. Rekurentně nalezneme dvojici nejbližších bodů pro levou i pravou polovinu. Rekurze končí v případě, že množina obsahuje pouze 3 body, to nalezneme dvojici nejbližších bodů hrubou silou.
3. Dvojicí nejbližších bodů v P je pak buď lepší z dvojic nejbližších bodů v levé a pravé polovině, nebo dvojice s jedním bodem z levé a s jedním bodem z pravé poloviny. Existenci takové dvojice lze efektivně ověřit.

Nyní si projdeme jednotlivé body podrobně. Pro množinu bodů P označíme pomocí P_x seznam bodů z P uspořádaný vzestupně podle x-ové souřadnice. Podobně P_y označuje seznam bodů z P uspořádaných vzestupně podle y-ové souřadnice. Množinu prvních $\lceil |P|/2 \rceil$ bodů ze seznamu P_x označíme jako Q, zbývající body jako R. Rekurzivně spustíme algorimus pro Q a R. Dvojici nejbližších bodů ve Q označíme jako q_1^* a q_2^*, dvojice nebližších bodů v R je pak r_1^* a r_2^*.

Označme si nyní jako δ kratší ze vdáleností $d(q_1^*, q_2^*)$ a $d(r_1^*, r_2^*)$. Ukážeme si, že pokud v P existuje dvojice bodů $q \in Q$ a $r \in R$ taková, že $d(q, r) < \delta$, lze tuto dvojici efektivně najít.

Věta 2. *Nechť x^* je x-ová souřadnice nejpravějšího bodu v Q a nechť L je svislá čára daná rovnicí $x = x^*$. Pokud existují body $q \in Q$ a $r \in R$ takové, že $d(q, r) < \delta$, pak tyto body leží maximálně ve zdálenosti δ od L.*

Důkaz. Označme $q = \langle q_x, q_y \rangle$ a $r = \langle r_x, r_y \rangle$. Z definice x^* plyne, že $q_x \leq x^* \leq r_x$. Odtud máme, že platí

$$x^* - q_x \leq r_x - q_x \leq d(q, r) \leq \delta,$$

a

$$r_x - x^* \leq r_x - q_x \leq d(q, r) \leq \delta,$$

z čehož už trvzení plyne. □

Z předchozí věty plyne, že stačí, když se při hledání q a r můžeme omezit na body ležící ve vzdálenosti maximálně δ od L. Označme si množinu takových bodů jako S.

Věta 3. *Pro všechny body $s', s \in S$ platí, že pokud $d(s', s) < \delta$, tak s' a s jsou od sebe v setřízeném seznamu S_y vzdáleny maximálně 15 míst.*

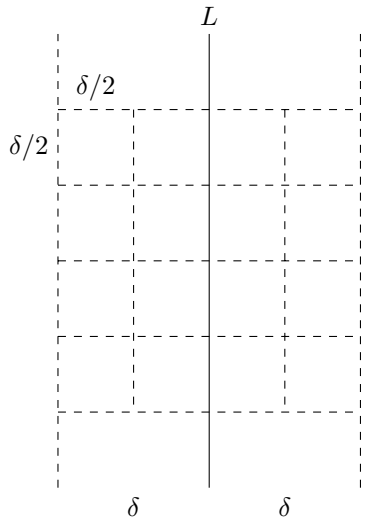
Důkaz. Větu dokážeme následující geometrickou konstrukcí. Představíme si plochu obsahující všechny prvky S, a tuto plochu rozdělíme na čtverce o velikosti stran $\delta/2$ (viz Obrázek 3). Všimněme si, že jedna řada se skládá ze 4 čtverců.

Nyní dokážeme, že každý takový čtverec může obsahovat pouze jeden bod z S. Důkaz provedeme sporem. Předpokládejme, že dva body z S leží ve stejném čtverci. Pak buď oba dva leží v Q nebo oba dva leží v R. Díky tomu, že strana čtverce je $\delta/2$, je maximální vzdálenost mezi těmito body rovna $\delta \cdot \sqrt{2}/2 \leq \delta$, což je spor s tím, že nejmenší vzdálenost mezi body uvnitř Q nebo uvnitř R je δ.

Vezměme nyní $s, s' \in S$ takové, že jsou od sebe v seznamu S_y vzdáleny 16 míst. Předpokládejme, že $s_y < s'_y$. Díky tomu, že v každém čtverci může být maximálně jeden bod, musí mezi s a s' ležet minimálně tři řady čtverců (v každé řadě jsou 4 čtverce). Tedy vzdálenost mezi s a s' je minimálně $3\delta/2$.

□

Díky předchozí větě se můžeme při pokusu o nalezení dvojice bodů se vzdáleností menší než δ omezit pouze na body, mezi nimiž je v seznamu S_y maximálně 15 prvků. Stačí pro každý prvek $s \in S$ nalézt nejbližší bod mezi 15 následujícími v S_y a z takto nalezených dvojici vybrat tu s nejmenší vzdálenstí. Pokud existuje v S dvojice se vzdálenosti menší než δ, je to právě tato dvojice.



Obrázek 3: Konstrukce z důkazu Věty 3.

Algoritmus 2 Nalezení dvojice nejbližších bodů

```
1: procedure ClosestPair(P)
2:     sestav P_x a P_y
3:     return ClosestPairHelp(P_x, P_y)
4: end procedure
5:
6: procedure ClosestPairHelp(P_x, P_y)
7:     if |P_x| ≤ 3 then
8:         najdi dvojici nejbližších bodů (p_0^*, p_1^*) hrubou silou
9:         return (p_0^*, p_1^*)
10:    end if
11:    sestav Q_x, Q_y, R_x, R_y
12:    (q_0^*, q_1^*) ←ClosestPairHelp(Q_x, Q_y)
13:    (r_0^*, r_1^*) ←ClosestPairHelp(R_x, R_y)
14:    if d(q_0^*, q_1^*) ≤ d(r_0^*, r_1^*) then
15:        δ ← d(q_0^*, q_1^*)
16:        (b_0, b_1) ← (q_0^*, q_1^*)
17:    else
18:        δ ← d(r_0^*, r_1^*)
19:        (b_0, b_1) ← (r_0^*, r_1^*)
20:    end if
21:    x^* ← maximální x-ová souřadnice bodu v Q
22:    S ← {p ∈ P | |p_x − x^*| ≤ δ}
23:    sestav S_y
24:    for s ∈ S do
25:        for s' ∈ S, s je maximálně 15 míst za s v S_y do
26:            if d(b_0, b_1) > d(s, s') then
27:                (b_0, b_1) ← (s, s')
28:            end if
29:        end for
30:    end for
31:    return (b_0, b_1)
32: end procedure
```

Protože proceduře ClosestPairHelp předáváme již setříděné seznamy, lze řádky 11 a 23 provést v lineárním čase, stejně tak cyklus na řádcích 24–30 a řádek 21. Řádky 7–10 a podmínku na řádcích 14-20 lze provést v konstantním čase. Díky tomu se dá složitost ClosestPairHelp vyjádřit rekurencí $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, jejímž řešením je $O(n \log n)$. Třídění v proceduře ClosestPair má složitost $O(n \log n)$, celkově je tedy složitost $O(n \log n)$, kde n je samozřejmě počet prvků v P.

3.2 Násobení polynomů a FFT

Součin $AB(x)$ polynomů $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$ a $B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d$ je polynom definovaný jako

$$AB(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d},$$

kde

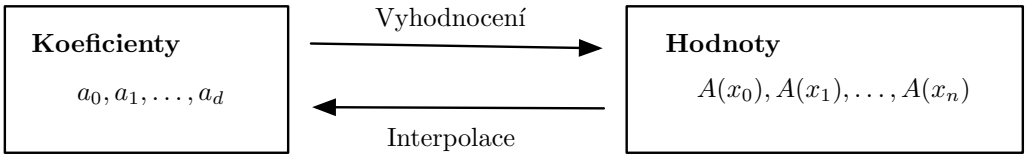
$$c_k = a_0 b_k + a_1 b_{k-1} + \cdots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}.$$

Algoritmus přímo založený na předchozím vztahu má složitost $O(d^2)$, kde d je stupeň polynomů. Pro výpočet c_k potřebujeme $k+1$ součinů a k součtů. Pro celý polynom, tj. $2d$ koeficientů je to $\sum_{k=0}^{2d}(2k+1) = O(d^2)$ operací.

Princip zefektivnění algoritmu pro násobení leží ve změně reprezentace polynomu a je založen na následující větě známé z algebry.

Věta 4. *Polynom stupně d je jednoznačně reprezentován svými hodnotami v alespoň $d+1$ různých bodech.*

Tedy místo hodnotami koeficientů lze polynom reprezentovat pomocí hodnot polynomu v odpovídajícím počtu bodů. Mezi oběma reprezentacemi lze libovolně přecházet.



Násobení polynomů A a B reprezentovaných hodnotami v stejných bodech x_i lze jednoduše provést vynásobením příslušných hodnot

$$AB(x_i) = A(x_i) \cdot B(x_i).$$

Tak získáme reprezentaci výsledného algoritmu pomocí hodnot, kterou můžeme převést zpět na koeficienty. Toto násobení má lineární složitost. Pokud tedy najdeme dostatečně efektivní algoritmus pro převod mezi reprezentacemi polynomu, dostaneme efektivní algoritmus pro násobení polynomů.

Pro rychlý výpočet hodnot polynomu i interpolaci lze použít Rychlou Fourierovu Transformaci (FFT). Vstupem FFT je polynom A reprezentovaný n koeficienty. Musí platit $n = 2^k$, což nás ale nijak neomezuje, protože koeficienty mocnin vyšších než je stupeň polynomu jsou rovny 0. Výstupem je reprezentace polynomu hodnotami v n bodech. FFT je založena na jednoduchém pozorování, že sudé mocniny v opačných bodech se rovnají, tedy, že pro sudé k platí $(-x)^k = x^k$. Předpokládejme tedy, že chceme spočítat hodnoty polynomu v bodech

$$\pm x_0, \pm x_1, \cdots \pm x_{n/2-1}. \tag{6}$$

Takovou volbu můžeme provést, protože počet koeficientů je vždy mocnina dvou. Polynom rozdělíme na členy se sudými a s lichými mocninami

$$A(x) = (a_0 + a_2 x^2 + \ldots) + x(a_1 + a_3 x^2 + \ldots)$$

Výhodné je, že oba výrazy v závorkách jsou opět polynomy. Výraz $(a_0 + a_2 x^2 + \ldots)$ je polynom $A_S(z) = a_0 + a_2 z + \ldots$, do kterého dosadíme x^2 za z, a tedy

$$(a_0 + a_2 x^2 + \ldots) = A_S(x^2). \tag{7}$$

Stejnou úvahou dostaneme pro polynom v pravé závorce polynom $A_L = a_1 + a_3 z + \dots$, pro který platí

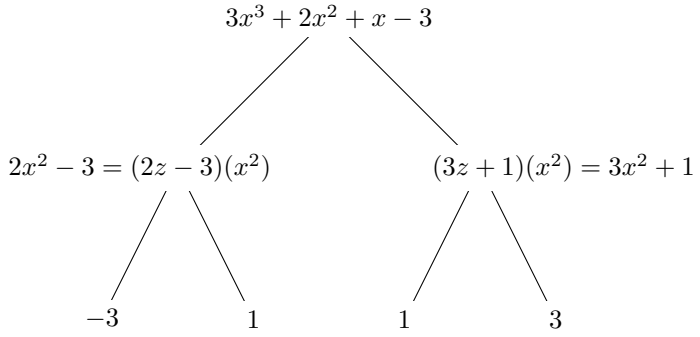
$$(a_1 + a_3 x^2 + \dots) = A_L(x^2). \tag{8}$$

Polynomy A_S a A_L mají poloviční stupeň jako n. Hodnoty pro $\pm x_i$ pak lze vyjádřit jako

$$A(x_i) = A_s(x_i^2) + x_i \cdot A_l(x_i^2), \tag{9}$$
$$A(-x_i) = A_s(x_i^2) - x_i \cdot A_l(x_i^2). \tag{10}$$

K výpočtu hodnot polynomu A v n bodech tedy potřebujeme vypočítat hodnoty polynomů A_S a A_L v $n/2$ bodech. Samozřejmě, v duchu rozděl a panuj použijeme stejný postup opakovaně pro výpočet hodnot A_S a A_L. Musíme ještě vyřešit jeden drobný zádrhel, ale nejdříve příklad.

Příklad 3. Uvažujme polynom $3x^3 + 2x^2 + x - 3$. Pak algoritmus projde postupně následující strom polynomů (A_S vlevo, A_L vpravo). Body, ve kterých se hodnoty počítají ještě neuvažujeme.



□

Řekněme, že v předchozím příkladě budeme chtít spočítat hodnoty polynomu pro body $\pm x_0, \pm x_1$. Při rekurzivním volání pak musíme spočítat hodnoty polynomů $(z - 3)$ pro a $(3z + 1)$ pro x_0^2 a x_1^2. Problém je, že x_0^2 a x_1^2 nemohou být opačná čísla, protože druhé mocniny reálných čísel jsou vždycky kladné. Abychom našli takové body, jejichž druhá mocnina je záporná, musíme přejít od reálných čísel ke komplexním. Nejjednodušší je případ, kdy je polynom stupně 1 a hodnoty počítáme pouze pro dvojici bodů (když je polynom jenom konstanta, je jeho hodnotou vždy tato konstanta a tedy nemá smysl volit body dvojici opačných bodů). Zvolme pro tuto úroveň rekurze (tedy úroveň stromu rekurze, na které se vyskytují polynomy stupně 1) body ± 1. O úroveň výše (úroveň, na které se vyskytují polynomy stupně 3) potřebujeme 2 páry bodů, druhá mocnina jednoho páru se musí rovnat 1, druhá mocnina druhého páru se musí rovnat -1. Přirozeně tyto body jsou $\pm 1, \pm i$, kde i je imaginární jednotka. Opakováním této konstrukce získáme body, pro které potřebujeme spočítat hodnoty polynomu v kořeni stromu rekurze (viz obrázek 4).

Každá z úrovní stromu na obrázku 4 obsahuje právě všechny odmocniny z 1. Přesněji, je-li v dané úrovni n uzlů, obsahuje právě všechny $\sqrt[n]{1}$. Algoritmus FFT využívá následujících vlastností $\sqrt[n]{1}$.

Pro $n = 2^k$ platí, že

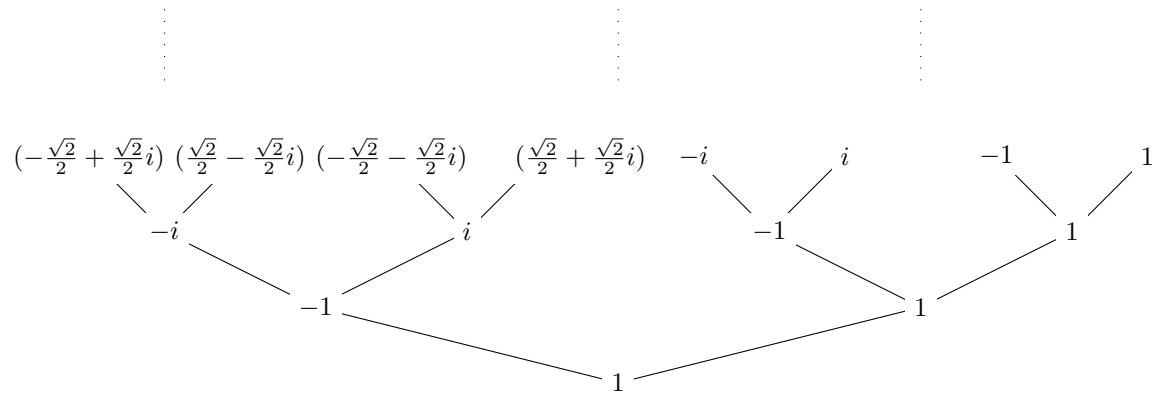
(a) n-té odmocniny jedné jsou $\omega^0 = 1, \omega, \omega^2, \dots, \omega^{n-1}$, kde $\omega = e^{\frac{2\pi i}{n}}$.

(b) $\omega^{\frac{n}{2}+j} = -\omega^j$

(c) Množina druhých mocnin $\sqrt[n]{1}$ jsou právě $\{1, \omega^2, (\omega^2)^2, \dots, (\omega^2)^{n/2-1}\}$, tedy $n/2$-té odmocniny 1.

Tyto vlastnosti nám umožňují iterovat přes všechny $\sqrt[n]{1}$. Stačí spočítat ω a pak iterovat přes její mocniny. Konec iterace poznáme podle toho, že se dostaneme na hodnotu 1. Například pro $n = 4$ je $\omega = i$ a jednotlivé mocniny jsou postupně $\omega^2 = -1$, $\omega^3 = -i$, $\omega^4 = 1$. Vlastnost (b) říká, že množina všech $\sqrt[n]{1}$ tvoří opravdu dvojice opačných bodů a k jejich nalezení stačí iterovat pouze přes polovinu mocnin. Druhá polovina jsou právě opačné body. Například pro $n = 4$ máme $\omega = i$ a opačný bod je $\omega^{1+2} = \omega^3 = -i$, pro $\omega^2 = -1$ je opačný bod $\omega^{2+2} = \omega^4 = 1$. Díky (c) v rekurzivních voláních skutečně počítáme s druhými mocninami bodů.

Nyní si již můžeme uvést pseudokód FFT.



Obrázek 4: Idea nalezení bodů pro výpočet hodnoty polynomu pomocí FFT. Potomci každého uzlu jsou jeho druhé odmocniny.

Algoritmus 3 Rychlá Fourierova transformace

```
1:  procedure FFF(A[0,...,n−1], ω)                    ▷ n je mocnina dvou
2:      if ω = 1 then                                  ▷ V tomto případě už |A| = 1
3:          return A
4:      end if
5:      for i ← 0 to n/2 − 1 do
6:          A_S[i] ← A[i · 2]                          ▷ Koeficienty pro sudé mocniny
7:          A_L[i] ← A[i · 2 + 1]                      ▷ Koeficienty pro liché mocniny
8:      end for
9:      S ← FFT(A_S, ω^2)
10:     L ← FFT(A_L, ω^2)
11:     x ← 1                                          ▷ Začínáme od ω^0
12:     for j ← 0 to n/2 − 1 do
13:         R[j] ← S[j] + x · L[j]                     ▷ Rovnice (9)
14:         R[j + n/2] ← S[j] − x · L[j]               ▷ Rovnice (10)
15:         x ← x · ω                                  ▷ Další mocnina ω
16:     end for
17:     return R
18: end procedure
```

Složitost FFT vyjádříme pomocí rekurence. V algoritmu jsou dvě rekurzivní volání, každému z nich předáváme instanci o velikosti $n/2$. Zbývající část algoritmu (tedy cykly na řádcích 5 až 8 a 13 az 17) má složitost $O(n)$. Rekurence je tedy $T(n) = 2T(n/2) + O(n)$ s okrajovou podmínkou $T(1) = 1$. Podle master theoremu je pak složitost FFT vyjádřena $\Theta(n \log n)$. Připomeňme si, že algoritmus přímým dosazením do polynomu má složitost $O(n^2)$.

Nyní si můžeme ukázat rychlý algoritmus pro násobení polynomů. Nechť A a B jsou polynomy, které chceme násobit se stupni s a t. Výsledný polynom AB bude mít stupeň $s \cdot t$. Abychom tedy mohli interpolovat z hodnot výsledného polynomu jeho koeficienty, potřebujeme jich znát nejméně $s \cdot t + 1$. Pro potřeby FFT ovšem musí být počet hodnot mocninou dvou, proto je skutečně počítaný počet hodnot roven

$$n = 2^{\lceil \log_2(s \cdot t + 1) \rceil},$$

tedy nejbližší větší mocnině dvou. Algoritmus nejdříve volá FFT, aby spočítal hodnoty A a B v n bodech, poté tyto body vynásobí a získá hodnoty v těchto bodech pro AB. Posledním krokem je interpolace koeficientů. Díky vlastnostem $\sqrt[n]{1}$ ji lze vypočítat také pomocí FFT. Pro hodnoty $AB(\omega^0), AB(\omega), AB(\omega^2), \ldots, AB(\omega^{n-1})$ dostaneme koeficienty $ab_0, ab_1, \ldots, ab_{n-1}$ pomocí

$$[ab_0, ab_1, \ldots, ab_{n-1}] = \frac{1}{n}\mathrm{FFT}([AB(\omega^0), AB(\omega), AB(\omega^2), \ldots, AB(\omega^{n-1})], \omega^{-1}).$$

Pseudokód algoritmu je označen jako Algoritmus 4.

Algoritmus 4 Rychlé násobení polynomů

```
1:  procedure FastPolyMultiply(A[0,...,s], B[0,...,t])
2:      n ← 2^⌈log2(s·t+1)⌉
3:      ω ← e^(2πi/n)
4:      Doplň pomocí nul A i B na n prvků.          ▷ Nuly přidávám na konec
5:      V_A ←FFT(A, ω)                              ▷ Hodnoty pro A
6:      V_B ←FFT(B, ω)                              ▷ Hodnoty pro B
7:      for i ← 0 to n − 1 do
8:          V_AB[i] ← V_A[i] · V_B[i]               ▷ Násobení polynomů
9:      end for
10:     return (1/n)FFT(V_AB, ω^−1)                 ▷ Interpolace pomocí FFT
11: end procedure
```

Složitost FastPolyMultiply je $O(n \log n)$. Algoritmus třikrát volá FFT se složitostí $O(n \log n)$. Řádek 4, cyklus na řádcích 7 až 9, i násobení pomocí $1/n$ na řádku 10 mají složitost $O(n)$. Celkově je tedy složitost dominována FFT.

Reference

[1] Donald Knuth. The Art of Computer Programming, Volume I. Addison-Wesley, 1997

[2] Donald Knuth. Selected papers on analysis of algorithms. Center for the study of language an information, 2000

[3] Cormen et. al. Introduction to algorithms. The MIT press. 2008.

[4] Donald Knuth Concrete mathematics: A foundation for computer science. Addison-Wesley professional, 1994

[5] John Kleinberg, Éva Tardes. Algorithm design. Addison-Wesley, 2005.

[6] U. Vazirani et. al. Algorithms. McGraw-Hill, 2006.

[7] Steve Skiena. The algorithm design manual. Springer, 2008.

[8] Juraj Hromkovič. Algorithmics for hard problems. Springer, 2010.

[9] Oded Goldreich. Computational complexity: a conceptual perspective. Cambridge university press, 2008.