13. Třídící algoritmy a násobení matic

Minulou přednášku jsme probírali QuickSort, jeden z historicky prvních třídících algoritmů, které překonaly kvadratickou složitost aspoň v průměrném případě.

Proč se dodnes používá, když známe algoritmy, které mají složitost $\Theta(n \log n)$ i v nejhorším případě? Protože QuickSort se k paměti chová téměř sekvenčně, takže na dnešních počítačích běží rychle.

Podívejme se na pár nápadů pro skutečné naprogramování tohoto algoritmu.

Určitě nám vadí překopírovávat z a do pomocných polí. Naštěstí můžeme přerovnávat přímo v původním poli. Zleva budeme dávat prvky menší než pivot, zprava větší než pivot. Na toto stačí udržovat si dva indexy a a b, které značí, jak daleko vlevo (vpravo) už jsou správné prvky.

Rekurzivní programy mají zbytečně velkou režii. Proto implementujme vlastní zásobník na hranice úseků, které zbývá setřídit. Vždy větší interval vložíme na zásobník a menší rovnou zpracujeme. Na zásobníku proto bude maximálně $\log n$ položek.

Malé podproblémy dotřídíme nějakým triviálním algoritmem například InsertSortem. Odhadem pro $n = 10$ je to pro dnešní počítače výhodné (zjišťuje se experimentálně).

Zoo třídících algoritmů

Porovnejme nyní známé třídící algoritmy.

Definice: *Stabilní třídění* říkáme takovému, které u prvků se stejnou hodnotou klíče zachová jejich vzájemné pořadí, v jakém byly na vstupu. (To se hodí například při lexikografickém třídění, kde se napřed třídí podle nejméně významné složky a pak podle významějších.)

Definice: Pokud třídíme prvky *na místě* (tedy vstup dostaneme zadaný v poli a v tomtéž poli pak vracíme výstup), za *pomocnou paměť* třídícího algoritmu prohlásíme veškerou využitou paměť mimo vstupní pole.

	Čas	*Pomocná paměť*	*Stabilní*
InsertSort	$\Theta(n^2)$	$\Theta(1)$	+
MergeSort	$\Theta(n \log n)$	$\Theta(n)$	+
HeapSort	$\Theta(n \log n)$	$\Theta(1)$	−
QuickSort	$\Theta(n \log n)$	$\Theta(\log n)$	−

Poznámky k tabulce:

- QuickSort má jen průměrnou časovou složitost $\Theta(n \log n)$. Můžeme ale říct, že porovnáváme průměrné časové složitosti, protože u ostatních algoritmů vyjdou stejně jako jejich časové složitosti v nejhorším případě.
- HeapSort – třídění pomocí haldy. Do haldy vložíme všechny prvky a pak je vybereme. Celkem $\Theta(n)$ operací s haldou, každá za $\Theta(\log n)$. Navíc tuto haldu mohu stavět i rozebírat v poli, ve kterém dostanu vstup.

- MergeSort jde implementovat s konstantní pomocnou pamětí za cenu konstantního zpomalení, ovšem konstanta je neprakticky velká.
- MergeSort je stabilní, když dělím pole na poloviny. Není při třídění spojových seznamů s rozdělováním prvků na sudé a liché.
- QuickSort se dá naprogramovat stabilně, ale potřebuje lineárně pomocné paměti.

Žádný algoritmus v tabulce netřídil rychleji než $\Theta(n \log n)$. To není náhoda – následující věta nám říká, že to nejde:

Věta: Každý deterministický třídící algoritmus, který tříděné prvky pouze porovnává a kopíruje, má časovou složitost $\Omega(n \log n)$.

(O průměrné časové složitosti pravděpodobnostních třídících algoritmů se dá dokázat podobná věta.)

Důkaz: Dokážeme, že každý porovnávací třídící algoritmus potřebuje v nejhorším případě provést $\Omega(n \log n)$ porovnání, což dává přirozený dolní odhad časové složitosti.

Přesněji řečeno, dokážeme, že pro každý algoritmus existuje vstup libovolné délky n, na němž algoritmus provede $\Omega(n \log n)$ porovnání. Bez újmy na obecnosti se budeme zabývat pouze vstupy, které jsou permutacemi množiny $\{1, \ldots, n\}$. (Stačí nám najít jeden „těžký“ vstup, pokud ho najdeme mezi permutacemi, úkol jsme splnili.)

Mějme tedy deterministický algoritmus a nějaké pevné n. Sledujme, jak algoritmus porovnává – u každého porovnání zaznamenáme polohy porovnávaných prvků tak, jak byly na vstupu. Jelikož algoritmus je deterministický, porovná na začátku vždy tutéž dvojici prvků. Toto porovnání mohlo dopadnout třemi různými způsoby (větší, menší, rovno). Pro každý z nich je opět jednoznačně určeno, které prvky algoritmus porovná, a tak dále. Po provedení posledního porovnání algoritmus vydá jako výstup nějakou jednoznačně určenou permutaci vstupu.

Chování algoritmu proto můžeme popsat rozhodovacím stromem. Vnitřní vrcholy stromu odpovídají porovnáním prvků, listy odpovídají vydaným permutacím. Ze stromu vynecháme větve, které nemohou nastat (například pokud už víme, že $x_1 < x_3$ a $x_3 < x_6$, a přijde na řadu porovnání x_1 s x_6, už je jasné, jak dopadne).

Počet porovnání v nejhorším případě je roven hloubce stromu. Jak ji spočítat?

Všimneme si, že pro každou z možných permutací na vstupu musí chod algoritmu skončit v jiném listu (jinak by existovaly dvě různé permutace, které lze setřídit týmiž prohozeními, což není možné). Strom tedy musí mít alespoň $n!$ různých listů.

Hloubka rozhodovacího stromu odpovídá počtu porovnání. My chceme dokázat, že porovnání musí být aspoň $\Omega(n \log n)$.

Lemmátko: Ternární strom hloubky k má nejvýše 3^k listů.

Důkazík: Uvažme ternární strom hloubky k s maximálním počtem listů. V takovém stromu budou všechny listy určitě ležet na poslední hladině (kdyby neležely, můžeme pod některý list na vyšší hladině přidat další dva vrcholy a získat tak „listnatější“ strom stejné hloubky). Jelikož na i-té hladině je nejvýše 3^i vrcholů, všech listů je nejvýše 3^k. ♡

Z tohoto lemmátka plyne, že rozhodovací strom musí být hluboký alespoň $\log_3 n!$. Zbytek už je snadné cvičení z diskrétní matematiky:

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

Hloubka stromu tedy činí minimálně $\log_3 n! \geq \log_3(n^{n/2}) = n/2 \cdot \log_3 n = \Omega(n \log n)$, což jsme chtěli dokázat. ♡

Ukázali jsme si třídění v čase $\mathcal{O}(N \log N)$ a také dokázali, že líp to v obecném případě nejde. Naše třídící algoritmy jsou tedy optimální (až na multiplikativní konstantu). Opravdu?

Překvapivě můžeme třídit i rychleji – věta omezuje pouze třídění pomocí porovnávaní. Co když o vstupu víme víc, třeba že je tvořen čísly z omezeného rozsahu.

Counting sort

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

Algoritmus postupně prochází vstup a počítá si ke každému prvku z rozsahu, kolikrát jej viděl. Poté až projde celý vstup, projde počítadla a postupně vypíše všechna čísla z rozsahu ve správném počtu kopií.

Algoritmus: (třídění posloupnosti $x_1, \ldots, x_N \in \{1, \ldots, R\}$ pomocí *Counting sort*u)

1. Pro $i = 1 \ldots R$ opakujeme:
2. $\quad p_i \leftarrow 0$
3. Pro $i = 1 \ldots N$ opakujeme:
4. $\quad p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$
6. Pro $i = 1 \ldots R$ opakujeme:
7. $\quad$ Pokud $p_i \neq 0$:
8. $\quad\quad v_j \leftarrow i$
9. $\quad\quad j \leftarrow j + 1$
10. Vrátíme výsledek $v_1, \ldots, v_N$.

Přihrádkové třídění

Counting sort nám moc nepomůže, pokud chceme třídit ne přímo celá čísla, nýbrž záznamy s celočíselnými klíči. Na ty se bude hodit přihrádkové třídění neboli *Bucket-sort* („kbelíkové třídění“).

Uvažujme opět N prvků s klíči v rozsahu $1, \ldots, R$. Pořídíme si R přihrádek $P_1, \ldots, P_R$, prvky do nich roztřídíme a pak postupně vypíšeme obsah přihrádek v pořadí podle klíčů.

Potřebujeme k tomu čas $\Theta(N + R)$ a paměť $\Theta(N + R)$. Navíc se jedná o stabilní algoritmus.

Algoritmus: (třídění prvků $x_1, \ldots, x_n$ s klíči $c_1, \ldots, c_n \in \{1, \ldots, R\}$ pomocí *bucket-sort*u)

1. $P_1 \ldots P_R \leftarrow \emptyset$
2. Pro $i = 1 \ldots n$:
3. Vložíme x_i do P_{c_i}.
4. Pro $j = 1 \ldots R$
5. Vypišeme obsah P_j.

Lexikografické třídění k-tic

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

Algoritmus: (třídění k-tic $x_1, \ldots, x_n$ lexikograficky pomocí *Bucket-sort*u)

1. $S \leftarrow x_1, \ldots, x_n$.
2. Pro $i = k$ až 1 opakujeme:
3. $S \leftarrow$ bucket-sort S podle i-té souřadnice.
4. Vydáme výsledek S.

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

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

Důkaz: Indukcí podle ℓ:

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

♡

Časová složitost je $\Theta(k \cdot (n + R))$, což je lineární s délkou vstupu $(k \cdot n)$ pro pevné k a R; paměťová složitost činí $\Theta(n + R)$.

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

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

Pokud bychom zvolili Z konstantní, časová složitost bude $\Theta(\log R \cdot n)$, což může být $n \log n$ nebo i víc. Zvolíme-li $Z = n$, dostáváme $\Theta(\frac{\log R}{\log n} \cdot n)$, což pro $R \leq n^\alpha$ znamená $\mathcal{O}(\alpha n)$. Polynomiálně velká celá čísla jde tedy třídit v lineárním čase.

Třídění řetězců

(Na přednášce letos nebylo, ale pro úplnost uvádíme.)

Mějme n řetězců $r_1, r_2 \dots r_n$ dlouhých $l_1, l_2 \dots l_n$ Označme si $L = \max_{1 \le i \le n} l_i$ délku nejdelšího řetězce a R počet znaků abecedy.

Problém je, že řetězce nemusí být stejně dlouhé (pokud by byly, lze se na řetězce dívat jako na k-tice, které už třídit umíme). S tím se můžeme pokusit vypořádat doplněním řetězců mezerami tak, aby měly všechny stejnou délku, a spustit na něj algoritmus pro k-tice. Tím dostaneme algoritmus, který bude mít časovou složitost $\mathcal{O}(Ln)$, což bohužel může být až kvadratické vzhledem k velikosti vstupu.

Příklad: na vstupu mějme k řetězců, kde prvních $k-1$ z nich bude mít délku 1 a poslední řetězec bude dlouhý přesně k. Vstup má tedy celkovou délku $2k-1$ a my teď doplníme prvních $k-1$ řetězců mezerami. Vidíme, že algoritmus teď bude pracovat v čase $\mathcal{O}(k^2)$. To, co nám nejvíce způsobovalo problémy u předchozího příkladu, bylo velké množství času zabraného porovnáváním doplněných mezer. Zkusíme proto řešit náš problem trochu chytřeji a koncové mezery do řetězů vůbec přidávat nebudeme.

Nejprve roztřídíme bucket-sortem řetězce do přihrádek (množin) P_i podle jejich délek, kde i značí délku řetězců v dané přihrádce, neboli definujme $P_i = \{r_j | l_j = i\}$. Dále si zavedeme seznam setříděných řetězců S takový, že v něm po k-tém průchodu třídícím cyklem budou řetězce s délkou alespoň $L-k+1$ (označme l) a zároveň v něm tyto řetězce budou setříděny lexikograficky od l-tého znaku po L-tý. Z definice tohoto seznamu je patrné, že po L krocích třídícího cyklu bude tento seznam obsahovat všechny řetězce a tyto řetězce v něm budou lexikograficky seřazeny.

Zbývá už jen popsat, jak tento cyklus pracuje. Nejprve vezme l-tou množinu P_l a její řetězce roztřídí do přihrádek Q_j (kde index j značí j-tý znak abecedy) podle jejich l-tého (neboli posledního) znaku. Dále vezme seznam S a jeho řetězce přidá opět podle jejich l-tého znaku do stejných přihrádek Q_j za již dříve přidané řetězce z P_l. Na závěr postupně projde všechny přihrádky Q_j a řetězce v nich přesune do seznamu S. Protože řetězce z přihrádek Q_j bude brát ve stejném pořadí, v jakém do nich byly umístěny, a protože ze seznamu S, který je setřízený podle $(l+1)$-ního znaku po L-tý, bude také brát řetězce postupně, bude seznam S po k-tém průchodu přesně takový, jaký jsme chtěli (indukcí bychom dokázali, že cyklus pracuje skutečně správně). Zároveň z popisu algoritmu je jasné, že během třídění každý znak každého řetězce použijeme právě jednou, tudíž algoritmus bude lineární s délkou vstupu (pro úplnost dodejme, že popsaný algoritmus funguje v případech, kdy abeceda má pevnou velikost).

Algoritmus: (třídění řetězců)

1. $L \leftarrow \max(l_1, l_2, \dots, l_n)$
2. Pro $i \leftarrow 1$ do L opakuj:
3. $\quad P_i \leftarrow \emptyset$
4. Pro $i \leftarrow 1$ do n opakuj:
5. $\quad pridej(P_{l_i}, r_i)$
6. $S \leftarrow \emptyset$
7. Pro $i \leftarrow L$ do 1 opakuj:

```
Pro j ← 1 do R opakuj:
    Q_j ← ∅
Pro j ← 1 do velikost(P_i) opakuj:
    vezmi(P_i, r)
    pridej(Q_{r[i]}, r)
Pro j ← 1 do velikost(S) opakuj:
    vezmi(S, r)
    pridej(Q_{r[i]}, r)
S ← ∅
Pro j ← 1 do R opakuj:
    Pro k ← 1 do velikost(Q_j) opakuj:
        vezmi(Q_j, r)
        pridej(S, r)
výsledek S
```

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

Zoo třídících algoritmů podruhé

Doplňme tedy naši tabulku:

	Čas	*Pomocná paměť*	*Stabilní*
InsertSort	$\Theta(n^2)$	$\Theta(1)$	+
MergeSort	$\Theta(n \log n)$	$\Theta(n)$	+
HeapSort	$\Theta(n \log n)$	$\Theta(1)$	−
QuickSort	$\Theta(n \log n)$	$\Theta(\log n)$	−
BucketSort	$\Theta(n + R)$	$\Theta(n + R)$	+
k-tice	$\Theta(k(n + R))$	$\Theta(n + R)$	+
RadixSort	$\Theta(n \log_n R)$	$\Theta(n)$	+

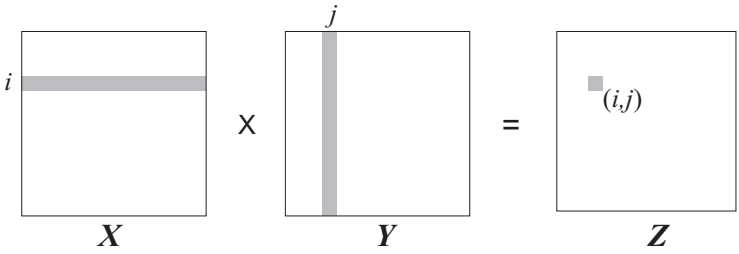
K čemu je vlastně třídění dobré?

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

Násobení matic n×n a Strassenův algoritmus

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

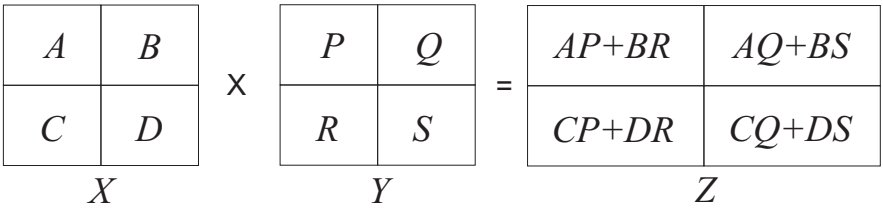
$$Z_{ij} = \sum_{k=1}^{n} X_{ik} \cdot Y_{kj}.$$



Násobení matic

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

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



Násobení rozčtvrcených matic

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

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

Použitím Master Theoremu lehce dojdeme k závěru, že složitost je stále $\Theta(n^3)$, tedy stejná jako při násobení matic z definice. Zdánlivě jsme si tedy nepomohli, ale stejně jako tomu bylo u násobení velkých čísel, i teď můžeme zredukovat počet násobení matic poloviního řádu, které nejvíce ovlivňuje časovou složitost algoritmu. Není to bohužel nic triviálního, a proto si raději rovnou řekneme správné řešení.

Jedná se o Strassenův algoritmus, který redukuje potřebný počet násobení na 7, a ještě před tím, než si ukážeme, jak funguje, dokážeme si, jak nám to s časovou složitostí vlastně pomůže:

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

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

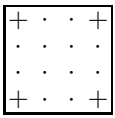
Lemma: (vzorce pro násobení blokových matic ve Strassenově algoritmu)

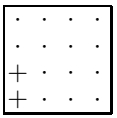
$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} P & Q \\ R & S \end{pmatrix} = \begin{pmatrix} T_1 + T_4 - T_5 + T_7 & T_3 + T_5 \\ T_2 + T_4 & T_1 - T_2 + T_3 + T_6 \end{pmatrix},$$

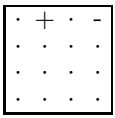
kde:

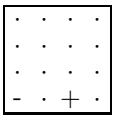
$$\begin{aligned} T_1 &= (A + D) \cdot (P + S) & T_5 &= (A + B) \cdot S \\ T_2 &= (C + D) \cdot P & T_6 &= (C - A) \cdot (P + Q) \\ T_3 &= A \cdot (Q - S) & T_7 &= (B - D) \cdot (R + S) \\ T_4 &= D \cdot (R - P) \end{aligned}$$

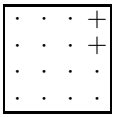
Důkaz: Do čtverců 4×4 si napíšeme znaky $+$ nebo $-$ podle toho, jestli se při výpočtu dané matice přičítá nebo odečítá příslušný součin dvou matic. Řádky znamenají matice A, B, C a D a sloupce značí matice P, Q, R a S. Pokud se tedy v prvním řádku a prvním sloupci vyskytuje znak $+$, znamená to že přičteme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

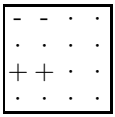
$T_1 =$ 

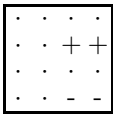
$T_2 =$ 

$T_3 =$ 

$T_4 =$ 

$T_5 =$ 

$T_6 =$ 

$T_7 =$ 

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

$$T_3 + T_5 = \begin{bmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix} = AQ + BS$$

$$T_2 + T_4 = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{bmatrix} = CP + DR$$

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

Jak je vidět, výsledná matice je tvořena stejnými částmi jako při obyčejném násobení. Touto kapitolou jsme tedy dokázali následující větu:
Věta: Strassenův algoritmus pro násobení matic $n \times n$ má časovou složitost v nejhorším případě $\mathcal{O}(n^{2.808})$. ♡
Poznámka: Zatím nejlepší dokázaný algoritmus má časovou složitost $\mathcal{O}(n^{2.376})$, leč s velkou multiplikativní konstantou.

Dosažitelnost v grafech pomocí Strassenova algoritmu

Matice mohou souviset s mnoha na první pohled nesouvisejícími problémy.
Lemma: Nechť A je matice sousednosti grafu, nechť $S^{(k)}_{z,j}$ označíme počet sledů délky k z vrcholu j do vrcholu i. Pak $S^{(k)} = A^k$.
Důkaz: Indukcí podle k:

- $S^{(1)} = A$
- $S^{(k+1)}_{i,j} = \sum_{z:(i,z)\in E(G)} S^{(k)}_{z,j} = \sum_{z=1}^{n} A_{i,z} S^{(k)}_{z,j} = (AS^{(k)})_{i,j}$

♡

Přidáním smyček do matice A zjistíme dostupnost vrcholů po cestách dlouhých k nebo kratších.

Stačí tedy spočítat matici $B = (A + E)^k$ pro libovolné $k \geq n$ (E je jednotková matice). Pak $B_{i,j} \neq 0$ právě když existuje cesta z vrcholu i do vrcholu j.

Pro vypočítání B nám stačí $\lceil \log n \rceil$ umocnění matice na druhou, což je speciální případ násobení matic. Časová složitost celého algoritmu tedy činí $\mathcal{O}(n^{\log_2 7} \cdot \log n)$.

Musíme však dávat pozor a normovat čísla (nulu necháme, nenulová nahradíme při každém násobení jedničkou), aby nám nepřerostla přes hlavu a hlavně přes maximální integer.