6. Třídění v lineárním čase *(zapsal M. Kupec, J. Volec, J. Ivánek)*

Na minulých přednaškách jsme si ukázali třídění v čase $\mathcal{O}(N \log N)$ a také dokázali, že líp to v obecném případě nejde. Jak je tedy možné třídit v lineárním čase?

Všechny dosud předvedené alrogitmy třídily pouze pomocí porovnávní hodnot na vstupu. Ale co když o vstupu víme víc, třeba rozsah hodnot na vstup, nedá se toho využít?

Následující algoritmy využívají znalosti rozsahu hodnot na vstupu a toho, že tento rozsah je malý.

Counting sort

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

Algoritmus postupně prochází vstup a počítá si ke každému prvku z rozsahu kolikrát jej viděl. Poté až projde celý vstup, projde počítadla a indexy u tech, kde něco napočítal vydá jako výstup.

Algoritmus: (třídění posloupnosti $(x_i)_0^N$ pomocí Counting sortu)

1. Pro $i \leftarrow 1$ do R opakuj:
2. $\quad p_i \leftarrow 0$
3. Pro $i \leftarrow 1$ do N opakuj:
4. $\quad p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$
6. Pro $i \leftarrow 1$ do R opakuj:
7. $\quad$ Pokud $p_i \neq 0$
8. $\quad\quad v_j \leftarrow i$
9. $\quad\quad j \leftarrow j + 1$
10. Vrať v

Přihrádkové třídění

Counting sort nám moc nepomůže, pokud chceme třídit neco jiného než celá čísla, ale i to se dá řešit. Přihrádkové, popřípadě kbelíkové třídění, neboli bucket-sort umí třídit množinu prvků očíslovaných čísly $1 \ldots R$.

Potřebuje k tomu čas $\mathcal{O}(N + R)$ a paměť $\mathcal{O}(N + R)$.

Bucket sort ma ještě jednu příjemnou vlastnost: jedná se o stabilní třídění.

Definice: *Stabilní třídění* je takové, že každé 2 prvky vstupu se stejnou hodnotou klíče jsou na výstupu ve stejněm pořadí jako na vstupu.

Algoritmus: (třídění množiny $x_1, x_2, \ldots, x_n$ očíslované $c_1, c_2, \ldots, c_n$ pomocí bucket-sortu)

1. Pro $k \leftarrow 1$ do R opakuj:
2. $\quad p_k \leftarrow 0$

3. Pro $i \leftarrow 1$ do N opakuj:
4. $\qquad p_{c_i} \leftarrow p_{c_i} + 1$
5. $b_1 \leftarrow 1$
6. Pro $k \leftarrow 2$ do R opakuj:
7. $\qquad b_k \leftarrow b_{k-1} + p_{b-1}$
8. Pro $i \leftarrow 1$ do N opakuj:
9. $\qquad t \leftarrow c_i$
10. $\qquad v_{b_t} \leftarrow x_i$
11. $\qquad b_t \leftarrow b_t + 1$
12. Vrať v

Lexikografické třídění k-tic

Mějme n uspořádaných k-tic prvků z množiny $\{1 \dots R\}^k$. 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 $x_1, x_2, \dots, x_n$ lexikograficky pomocí Bucket-sortu)

1. $S \leftarrow \{x_1, x_2, \dots, x_n\}$
2. Pro $i \leftarrow k$ do 1 opakuj:
3. $\qquad S \leftarrow$ bucket-sort S podle i-té souřadnice
4. vysledek S

Pro přehlednost v následujícím pozorování označme $l = k - i + 1$, 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 $l = 1$ jsou prvky uspořádány podle poslední souřadnice.

Po l průchodech již máme prvky setříděny lexikograficky podle i-té až k-té souřadnice a spouštíme $(l + 1)$-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 $(l + 1)$-ním průchodu jsou prvky seřazeny podle $(i - 1)$-ní až k-té souřadnice.

$\heartsuit$

Časová složitost je $\mathcal{O}(k \cdot (n + R))$, což je lineární s délkou vstupu $(k \cdot n)$ pro pevné k a paměťová složitost je $\mathcal{O}(n + R)$.

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 třídít v čase $\mathcal{O}(\frac{\log R}{\log Z} \cdot (n+Z))$. A jak zvolit vhodně Z?

Pokud bychom zvolili $Z = konstanta$, časová složitost bude $\mathcal{O}(\log R \cdot n)$, což může být $n * \log n$ nebo i víc. Zvolíme-li $Z = n$, dostáváme $\mathcal{O}(\frac{\log R}{\log n} \cdot n)$, což pro $R \leq n^\alpha$ znamená $\mathcal{O}(\alpha \cdot n)$.

Třídění řetězců

Mějme n řetězců $r_1, r_2 \ldots r_n$ dlouhých $l_1, l_2 \ldots l_n$ Označme si $L = \max_{1 \leq i \leq n} l_i$ 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 třídít 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 $\mathcal{O}(L * n)$, 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 bucketsortem 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. $L \leftarrow \max(l_1, l_2, \ldots, l_n)$

```
Pro i ← 1 do L opakuj:
    P_i ← ∅
Pro i ← 1 do n opakuj:
    pridej(P_{l_i}, r_i)
S ← ∅
Pro i ← 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.

Sorting Zoo

	$\mathcal{O}(1)$	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
$\mathcal{O}(N^2)$	*Buble*		
$\mathcal{O}(N \log N)$	*Heap*	*Quick*	*Merge*
$\mathcal{O}(N)$			*Bucket*

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 setřiděnou posloupnost projdeme.

Ukázali jsme si tedy, že v se setřízené množině jsme schopni vyhledávat prvky v čase $\mathcal{O}(\log(n))$. Bohužel do ní v čase $\mathcal{O}(\log(n))$ neumíme prvky přidávat. Jedním ze způsobu, jak zrychlit přidávání nových prvků a zároveň nepřijít o logaritmické vyhledávání, je použít další datouvou strukturu, vyhledávací stromy.

Vyhledávací stromy

Problém: chceme udržovat nějaká data setřízená. To znamená, že chceme udržovat množinu X prvků z lineárně uspořádaného universa U s operacemi Find (najdi prvek), Insert (vlož prvek), Delete (smaž prvek). K tomu se nám bude hodit reprezentace dat pomocí stromu.

Definice: Binární strom je zakořeněný strom; každý vrchol má max. 2 syny, pravého a levého.

Dále budeme používat toto značení: $v \ldots$ vrchol $l(v) \ldots$ levý syn, $p(v) \ldots$ pravý syn $L(v) \ldots$ levý podstrom, $p(v) \ldots$ pravý podstrom $T(v) \ldots$ podstrom s kořenem v.

Definice: Binární vyhledávací strom je takový binární strom, že každý vrchol obsahuje prvek z U a pro $\forall v : \forall x \in L(v) : x < v, \forall x \in P(v) : x > v$.

Find($T(v)$,x) Procházíme strom od kořene, na každém vrcholu se podle hodnoty x rozhodneme jak budeme pokračovat. Buď se hodnota vrcholu rovná x, pak končíme (nalezli jsme hledaný prvek). V opačném případě pokračujeme v hledání v levém podstromu (x je menší než hodnota vrcholu), nebo v pravém podstromu (x je větší než hodnota vrcholu). Pokud dojdeme do listu a prvek jsme nenašli, znamená to že tento prvek ve stromě není. Toto nám zabere $\mathcal{O}$(Hloubka stromu).

Insert($T(v)$,x) Nejprve najdeme místo kam je potřeba prvek vložit, k tomu použijeme operaci Find. Pokud Find najde x neprovedeme nic (máme strom s jedinečnými hodnotami). Jinak vložíme prvek na místo, kde by měl být (jako syna vrcholu ve kterém skončil Find). Nejsložitější je operace Find, která trvá $\mathcal{O}$(Hloubka stromu).

Min($T(v)$) Minimum se nachází v nejlevějším listu stromu. Jdeme tedy z kořene pořád doleva, dokud nenarazíme na list. Časová složitost je $\mathcal{O}$(hloubka stromu).

Delete($T(v)$,x) Opět použijeme Find, zjistíme zda prvek vůbec existuje a kde se nachází. Pokud existuje můžou nastat tyto případy:

x je list: Prostě ho smažeme.

x má jednoho syna: Zrušíme x a syna připojíme na uzel, který ukazoval na x.

x má dva syny: Najdeme minimum $P(v)$, toto minimum přehodíme s x a x smažeme.

Čas je tentokrát maličko složitější, Find nám trvá $\mathcal{O}$(Hloubka stromu) a v posledním případě nalezení Min($P(v)$) trvá taky až $\mathcal{O}$(Hloubka stromu), což nám dohromady dává pěkných $\mathcal{O}$(Hloubka stromu).

Strom se může používáním operací "kazit", např. opakováním Insert na max. hodnotu což vede až k hloubce $\sim n$. Můžeme

vykašlat se na to → potřeba dokázat průměrný případ

vylepšit Insert a Delete → vyvažování stromů