1. Třídění

Vyhledávání ve velkém množství dat je každodenním chlebem programátora. Hledání bývá snazší, pokud si data vhodným způsobem uspořádáme. Seřadíme-li například pole n čísel podle velikosti, algoritmus binárního vyhledávání z oddílu **??** v nich dokáže hledat v čase $\Theta(\log n)$. V této kapitole prozkoumáme různé způsoby, jak data efektivně *seřadit* neboli *setřídit*.⟨1⟩

Obvykle budeme pracovat v takzvaném *porovnávacím modelu.* V paměti stroje RAM dostaneme posloupnost n prvků $a_1, \ldots, a_n$. Mimo to dostaneme *komparátor* – funkci, která pro libovolné dva prvky a_i a a_j odpoví, je-li $a_i < a_j$, $a_i > a_j$ nebo $a_i = a_j$. Úkolem algoritmu je přerovnat prvky v paměti do nějakého pořadí $b_1 \leq b_2 \leq \ldots \leq b_n$. S prvky nebudeme provádět jiného, než je porovnávat a přesouvat. Budeme předpokládat, že jedno porovnání i přesunutí stihneme provést v konstantním čase.

Čtenář znalý knihovních funkcí populárních programovacích jazyků si jistě vzpomene, že jedním z argumentů funkce pro třídění bývá typicky takovýto komparátor. My budeme pro názornost zápisu předpokládat, že třídíme čísla a komparátor se jmenuje prostě „<“. Stále se ale budeme hlídat, abychom nepoužili jiné operace než ty povolené.

U třídicích algoritmů nás kromě časové složitosti budou zajímat i následující vlastnosti:

Definice: Třídicí algoritmus je *stabilní,* pokud kdykoliv jsou si prvky p_i a p_j rovny, tak jejich vzájemné pořadí na výstupu se shoduje s jejich pořadím na vstupu. Tedy pokud je $p_i = p_j$ pro $i < j$, pak se p_i ve výstupu objeví před p_j.

Definice: Algoritmus třídí prvky *na místě,* pokud prvky neopouštějí paměťové buňky, v nichž byly zadány, s možnou výjimkou konstantně mnoha tzv. pracovních buněk. Kromě toho může algoritmus ukládat libovolné množství číselných proměnných (indexy prvků, parametry funkcí apod.), které prohlásíme za *pomocnou paměť* algoritmu.

Postupně ukážeme, že třídit lze v čase $\Theta(n \log n)$, a to stabilně a na místě. Také dokážeme, že v porovnávacím modelu nemůže rychlejší třídicí algoritmus existovat. Ovšem pokud si dovolíme provádět s prvky i další operace, nalezneme efektivnější algoritmy.

1.1. Základní třídicí algoritmy

Nejjednodušší třídicí algoritmy patří do skupiny tzv. *přímých metod*. Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí na místě. Za tyto příjemné vlastnosti zaplatíme kvadratickou časovou složitostí $\Theta(n^2)$. Přímé metody jsou proto použitelné jen tehdy, není-li tříděných dat příliš mnoho.

⟨1⟩ Striktně vzato, termín *řazení* je správnější. Data totiž nerozdělujeme do nějakých tříd, nýbrž je uspořádáváme, tedy řadíme, dle určitého kritéria. Ovšem pojem *třídění* je mezi českými informatiky natolik zažitý, že je bláhové chtít na tom cokoliv měnit.

Selectsort – třídění výběrem

Třídění přímým výběrem (Selectsort) je založeno na opakovaném vybírání nejmenšího prvku. Pole rozdělíme na dvě části: V první budeme postupně stavět setříděnou posloupnost a v druhé nám budou zbývat dosud nesetříděné prvky. V každém kroku nalezneme nejmenší ze zbývajících prvků a přesuneme jej na začátek druhé (a tedy i na konec první) části. Následně zvětšíme setříděnou část o 1, čímž oficiálně potvrdíme členství právě nalezeného minima v konstruované posloupnosti a zajistíme, aby se při dalším hledání již s tímto prvkem nepočítalo.

Algoritmus SelectSort (třídění přímým výběrem)
Vstup: Pole $P[1 \ldots n]$

1. Pro $i = 1, \ldots, n-1$:
2. $\quad m \leftarrow i \quad \triangleleft$ *m bude index nejmenšího dosud nalezeného prvku*
3. $\quad$ Pro $j = i+1, \ldots, n$:
4. $\quad\quad$ Pokud je $P[j] < P[m]$: $m \leftarrow j$
5. $\quad$ Prohodíme prvky $P[i]$ a $P[m]$. $\quad \triangleleft$ *pokud $i = m$, nic se nestane*

Výstup: Setříděné pole P

V i-tém průchodu vnějším cyklem hledáme minimum z $n - i + 1$ čísel, na což potřebujeme čas $\Theta(n - i + 1)$. Ve všech průchodech dohromady tedy spotřebujeme čas $\Theta(n + (n-1) + \ldots + 3 + 2) = \Theta(n \cdot (n-1)/2) = \Theta(n^2)$.

Bubblesort – bublinkové třídění

Další z rodiny přímých algoritmů je *bublinkové třídění (Bubblesort).* Jeho základem je myšlenka nechat stoupat větší prvky v poli podobně, jako stoupají bublinky v limonádě.

V algoritmu budeme opakovaně procházet celé pole. Jeden průchod postupně porovná všechny dvojice sousedních prvků $P[i]$ a $P[i+1]$. Pokud dvojice není správně uspořádaná (tedy $P[i] > P[i+1]$), prvky prohodíme. V opačném případě necháme dvojici na pokoji. Menší prvky se nám tak posunou blíže k začátku pole, zatímco větší prvky „bublají“ na jeho konec. Pokaždé, když pole projdeme celé, začneme znovu od začátku. Tyto průchody opakujeme, dokud dochází k prohazování prvků. V okamžiku, kdy výměny ustanou, je pole setříděné.

Algoritmus BubbleSort (bublinkové třídění)
Vstup: Pole $P[1 \ldots n]$

1. *pokračuj* $\leftarrow 1$ $\quad \triangleleft$ *má proběhnout další průchod?*
2. Dokud je *pokračuj* $= 1$:
3. $\quad$ *pokračuj* $\leftarrow 0$
4. $\quad$ Pro $i = 1, \ldots, n-1$:
5. $\quad\quad$ Pokud je $P[i] > P[i+1]$:
6. $\quad\quad\quad$ Prohodíme prvky $P[i]$ a $P[i+1]$.
7. $\quad\quad\quad$ *pokračuj* $\leftarrow 1$

Výstup: Setříděné pole P

Jeden průchod vnitřním cyklem (kroky 4 až 7) jde přes všechny prvky pole, takže má určitě složitost $\Theta(n)$. Není ovšem na první pohled zřejmé, kolik průchodů bude potřeba vykonat. To nahlédneme následovně:

Lemma: Po k-tém průchodu vnějším cyklem je na správných místech k největších prvků.

Důkaz: Indukcí podle k. V prvním průchodu se největší prvek dostane na samý konec pole. Na začátku k-tého průchodu je podle indukčního předpokladu na správných místech $k-1$ největších prvků. Během k-tého průchodu tyto prvky na svých místech zůstanou, takže průchod pracuje pouze s prvky na prvních $n-k$ pozicích. Mezi nimi správně najde maximum a přesune ho na konec. Toto maximum je právě k-tý největší prvek. Tím je indukční krok hotov. □

Cvičení

1. Nahlédněte, že v k-tém průchodu Bubblesortu stačí zkoumat prvky na pozicích $1, \ldots, n-k+1$. Změní se touto úpravou časová složitost?
2. Na jakých datech provede Bubblesort pouze jeden průchod? Na jakých právě dva průchody? Kolik přesně průchodů vykoná nad sestupně uspořádaným vstupem?
3. Všimněme si, že Bubblesort může provádět spoustu zbytečných porovnání. Například když bude první polovina pole setříděná a až druhá rozházená, Bubblesort bude stejně vždy procházet první polovinu, i když v ní nebude nic prohazovat. Navrhněte možná vylepšení, abyste eliminovali co nejvíce zbytečných porovnání.
4. Určete průměrnou časovou složitost Bubblesortu (v průměru přes všechny možné permutace prvků na vstupu).
5. *Insertsort* neboli třídění přímým vkládáním funguje takto: Udržujeme dvě části pole – na začátku leží setříděné prvky a v druhé části pak zbývající nesetříděné. V každém kroku vezmeme jeden prvek z nesetříděné části a vložíme jej na správné místo v části setříděné. Dopracujte detaily algoritmu a analyzujte jeho složitost.
6. Předpokládejme na chvíli, že by počítač, na kterém běží naše programy, uměl provést operaci posunutí celého úseku pole o 1 prvek na libovolnou stranu v konstantním čase. Řekli bychom například, že chceme prvky na pozicích 42 až 54 posunout o 1 doprava (tj. na pozice 43 až 55), a počítač by to uměl provést v jednom kroku. Zkuste za těchto podmínek upravit Insertsort, aby pracoval s časovou složitostí $\Theta(n \log n)$.
7. Určete, jakou složitost bude mít Insertsort, pokud víme, že se setříděním každý prvek posune nejvýše o k pozic. Záleží na způsobu, jakým hledáme místo k zatřídění prvku?
8. Dokažte, že za stejných podmínek jako v předchozím cvičení provede Bubblesort nejvýše k průchodů. Pokud vám to dělá potíže, ukažte alespoň, že stačí $\mathcal{O}(k)$ průchodů.

9.* Navrhněte pro úlohu z cvičení 7 efektivnější algoritmus. Můžete se inspirovat třeba Heapsortem z oddílu **??**.

10. Upravte třídicí algoritmy z tohoto oddílu, aby byly stabilní.

1.2. Třídění sléváním

Nyní představíme třídící algoritmus s časovou složitostí $\Theta(n \log n)$. Na vstupu je dána n-prvková posloupnost $a_0, \ldots, a_{n-1}$ v poli. Pro jednoduchost nejprve předpokládejme, že n je mocnina dvojky.

Základní myšlenka algoritmu je tato: Pole rozdělíme do tzv. *běhů* o délce mocniny dvou – souvislých úseků, které už jsou vzestupně setříděny. Na začátku budou všechny běhy jednoprvkové. Poté budeme dohromady slévat vždy dva sousední běhy do jediného setříděného běhu o dvojnásobné délce, který bude ležet na místě obou vstupních běhů. To znamená, že v i-té iteraci budou mít běhy délku 2^i prvků a jejich počet bude $n/2^i$. V poslední iteraci bude posloupnost sestávat z jediného běhu, a bude tudíž setříděná.

Algoritmus MergeSort1
Vstup: Posloupnost $a_0, \ldots, a_{n-1}$ k setřídění

1. $b \leftarrow 1$ ◁ *aktuální délka běhu*
2. Dokud $b < n$: ◁ *zbývají aspoň dva běhy*
3. Pro $i = 0, 2b, 4b, 6b, \ldots, n - 2b$: ◁ *začátky sudých běhů*
4. $X \leftarrow (a_i, \ldots, a_{i+b-1})$ ◁ *sudý běh*
5. $Y \leftarrow (a_{i+b}, \ldots, a_{i+2b-1})$ ◁ *následující lichý běh*
6. $(a_i, \ldots, a_{i+2b-1}) \leftarrow \text{Merge}(X, Y)$
7. $b \leftarrow 2b$

Výstup: Setříděná posloupnost $a_0, \ldots, a_{n-1}$

Procedura Merge se stará o samotné slévání. To zařídíme snadno: Pokud chceme slít posloupnosti $x_1 \le x_2 \le \ldots \le x_m$ a $y_1 \le y_2 \le \ldots \le y_n$, bude výsledná posloupnost začínat menším z prvků x_1 a y_1. Tento prvek z příslušné vstupní posloupnosti přesuneme na výstup a pokračujeme stejným způsobem. Pokud to byl (řekněme) prvek x_1, zbývá nám slít $x_2, \ldots, x_m$ s $y_1, \ldots, y_n$. Dalším prvkem výstupu tedy bude minimum z x_2 a y_1. To opět přesuneme a tak dále, než se buď x nebo y vyprázdní.

Procedura Merge (slévání)
Vstup: Běhy $x_1, \ldots, x_m$ a $y_1, \ldots, y_n$

1. $i \leftarrow 1$, $j \leftarrow 1$ ◁ *zbývá slít* $x_i, \ldots, x_m$ *a* $y_j, \ldots, y_n$
2. $k \leftarrow 1$ ◁ *výsledek se objeví v* $z_k, \ldots, z_{m+n}$
3. Dokud $i \le m$ a $j \le n$, opakujeme:
4. Je-li $x_i \le y_j$, přesuneme prvek z x: $z_k \leftarrow x_i$, $i \leftarrow i + 1$.
5. Jinak přesouváme z y: $z_k \leftarrow y_j$, $j \leftarrow j + 1$.
6. $k \leftarrow k + 1$
7. Je-li $i \le m$, zkopírujeme zbylá x: $z_k, \ldots, z_{m+n} \leftarrow x_i, \ldots, x_m$.

8. Je-li $j \leq n$, zkopírujeme zbylá y: $z_k, \ldots, z_{m+n} \leftarrow y_j, \ldots, y_n$.

Výstup: Běh $z_1, \ldots, z_{m+n}$

Nyní odvodíme asymptotickou složitost algoritmu MergeSort. Začneme funkcí Merge: ta pouze přesouvá prvky a každý přesune právě jednou. Její časová složitost je tedy $\Theta(n+m)$, v i-té iteraci proto na slití dvou běhů spotřebuje čas $\Theta(2^i)$. V rámci jedné iterace se volá Merge řádově $n/2^i$-krát, což dává celkem $\Theta(n)$ operací na iteraci. Protože se algoritmus zastaví, když $2^i = n$, počet iterací bude $\Theta(\log n)$, což dává celkovou časovou složitost $\Theta(n \log n)$.

Mergesort bohužel neumí třídit na místě: při slévání musí být zdrojové běhy uloženy jinde než cílový běh. Proto potřebujeme pomocnou paměť velikosti $\mathcal{O}(n)$, například v podobě pomocného pole stejné velikosti jako vstupní pole.

Nyní doplníme, co si počít, když počet prvků není mocninou dvojky. Pořád budeme velikost běhů zvyšovat po mocninách dvojky, ale připustíme, že poslední z běhů může být menší a že celkový počet běhů může být lichý. Proto bude platit, že v i-té iteraci činí počet běhů $\lceil n/2^i \rceil$, takže po $\Theta(\log n)$ iteracích se algoritmus zastaví. Implementace je jednoduchá, jak je ostatně vidět z následujícího pseudokódu.

Algoritmus MergeSort (třídění sléváním)

Vstup: Posloupnost $a_0, \ldots, a_{n-1}$ k setřídění

1. $b \leftarrow 1$ ◁ *aktuální délka běhu*
2. Dokud $b < n$: ◁ *zbývají aspoň dva běhy*
3. $\quad i \leftarrow 1$, $j \leftarrow b+1$ ◁ *začátek aktuálního sudého a lichého běhu*
4. $\quad$ Dokud $j \leq n$: ◁ *ještě zbývá nějaká dvojice běhů*
5. $\quad\quad k \leftarrow \min(j+b-1, n)$ ◁ *konec lichého běhu*
6. $\quad\quad X \leftarrow (a_i, \ldots, a_{j-1})$ ◁ *sudý běh*
7. $\quad\quad Y \leftarrow (a_j, \ldots, a_k)$ ◁ *lichý běh*
8. $\quad\quad (a_i, \ldots, a_k) \leftarrow \text{Merge}(X, Y)$
9. $\quad\quad i \leftarrow i+2b$, $j \leftarrow j+2b$ ◁ *posuneme se na další dvojici*
10. $\quad b \leftarrow 2b$

Výstup: Setříděná posloupnost $a_0, \ldots, a_{n-1}$

Na závěr dodejme, že Mergesort také můžeme formulovat jako elegantní rekurzivní algoritmus. S tímto přístupem se setkáme v kapitole **??**.

Cvičení

1. Navrhněte algoritmus pro efektivní třídění dat uložených v jednosměrném spojovém seznamu. Algoritmus smí používat pouze $\mathcal{O}(1)$ buněk pomocné paměti a jednotlivé položky seznamu smí pouze přepojovat, nikoliv kopírovat na jiné místo v paměti.
2. Je Mergesort stabilní? Pokud ne, uměli byste ho upravit, aby stabilní byl?
3.* *Knížky v knihovně:* Mějme posloupnost, která vznikla ze setříděné tím, že jsme přesunuli k prvků. Navrhněte algoritmus, který ji co nejrychleji dotřídí. Pozor

na to, že k předem neznáme. Můžete nicméně předpokládat, že k je mnohem menší než délka posloupnosti.

4. *Pišvejcova čísla* říkejme číslům tvaru $2^i 3^j 5^k$. Vymyslete, jak co nejrychleji vygenerovat prvních n Pišvejcových čísel.

5.** *Mergesort na místě:* Na naší verzi Mergesortu je nešikovné, že potřebuje lineární množství pomocné paměti, neboť neumíme slévat na místě. Jde to i lépe: paměťové nároky procedury MERGE lze srazit až na konstantu při zachování lineární časové složitosti. Je to ale docela složité (a v praxi se to kvůli vysokým konstantám nevyplatí), tak zkuste přijít na slévání v lineárním čase a pomocné paměti velikosti $\mathcal{O}(\sqrt{n})$.

1.3. Dolní odhad složitosti třídění

Jak už jsme zmínili v úvodu kapitoly, nabízí se otázka, zda je možné třídit v čase rychlejším než $\Theta(n \log n)$. Nyní dokážeme, že odpověď je negativní, ale budeme k tomu potřebovat dva předpoklady:

- Algoritmus pracuje v porovnávacím modelu, smí tedy tříděné prvky pouze vzájemně porovnávat a přesouvat (přiřazovat).
- Algoritmus je deterministický – každý krok je jednoznačně určen výsledky kroků předchozích (algoritmus tedy nepoužívá žádný zdroj náhody).

Vyhledávání

Nejprve dokážeme jednodušší výsledek pro vyhledávání. Budeme se snažit ukázat, že binární vyhledávání je optimální (asymptoticky, tedy až na multiplikativní konstantu). Připomeňme, že jeho časová složitost je $\mathcal{O}(\log n)$. Zjistíme, že každý algoritmus potřebuje $\Omega(\log n)$ porovnání, tedy musí celkově provést $\Omega(\log n)$ operací.

Věta (o složitosti vyhledávání): Každý deterministický algoritmus v porovnávacím modelu, který nalezne zadaný prvek v n-prvkové uspořádané posloupnosti, použije v nejhorším případě $\Omega(\log n)$ porovnání.

Myšlenka důkazu: Porovná-li algoritmus nějaká dvě čísla x a y, dozví se jeden ze tří tří možných výsledků: $x < y$, $x > y$, $x = y$. Navíc nastane-li rovnost, výpočet skončí, takže všechna porovnání kromě posledního dávají jenom dva možné výsledky. Jedním porovnáním tedy získáme jeden bit informace. Žádná jiná operace nám o vztahu hledaného čísla s prvky posloupnosti nic neřekne.

Výstupem vyhledávacího algoritmu je pozice hledaného prvku v posloupnosti; to je číslo od 1 do n, na jehož určení je potřeba $\log_2 n$ bitů. Abychom ho určili, musíme tedy porovnat alespoň $(\log_2 n)$-krát. □

V této úvaze nicméně spoléháme na intuitivní představu o množství informace – poctivou teorii informace jsme nevybudovali. Větu proto dokážeme formálněji zkoumáním možných průběhů algoritmu.

Důkaz: Zvolíme pevné n a vstupní posloupnost $1, \ldots, n$. Budeme zkoumat, jak se výpočet algoritmu vyvíjí pro jednotlivá hledaná čísla $x = 1, \ldots, n$. Pokud ukážeme, že je potřeba provést $\Omega(\log n)$ porovnání pro vstupy tohoto speciálního typu, tím spíš to bude platit v nejhorším případě.

Spustíme algoritmus. Zpočátku jeho výpočet nezávisí na x (zatím jsme neprovedli jediné porovnání), takže první porovnání, které provede, bude vždy stejné. Pokud je to porovnání typu $a_i < a_j$, dopadne také vždy stejně. Až první porovnání typu $a_i < x$ může pro různá x dopadnout různě.

Pro každý z možných výsledků porovnání ale algoritmus pokračuje deterministicky, takže je opět jasné, jaké další porovnání provede. A tak dále, až se algoritmus rozhodne zastavit a vydat výsledek.

Možné průběhy výpočtu tedy můžeme popsat takzvaným *rozhodovacím stromem.* V každém vnitřním vrcholu tohoto stromu je jedno porovnání typu $x < a_i$. Vrchol má tři syny, kteří odpovídají možným výsledkům tohoto porovnání (menší, větší, rovno). Může se stát, že některé z výsledků nemohou nastat, protože by byly ve sporu s dříve provedenými porovnáními. V takovém případě příslušného syna vynecháme.

V listech rozhodovacího stromu jsou jednotlivé výsledky algoritmu: některé listy odpovídají nahlášení výskytu na nějaké pozici, v jiných odpovídáme, že prvek nebyl nalezen.

Rozhodovací strom je tedy ternární strom (vrcholy mají nejvýše 3 syny) s alespoň n listy. Použijeme následující lemma:

Lemma T: Ternární strom hloubky k má nejvýše 3^k listů.

Důkaz: 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 tři vrcholy a získat tak „listnatější“ strom stejné hloubky). Jelikož na i-té hladině je nejvýše 3^i vrcholů, všech listů je nejvýše 3^k. □

Strom má proto hloubku alespoň $\log_3 n$, takže v něm existuje cesta z kořene do listu, která obsahuje alespoň $\log_3 n$ vrcholů. Tudíž existuje vstup, na němž algoritmus provede alespoň logaritmicky mnoho porovnání. □

Třídění

Nyní použijeme podobnou metodu pro odhad složitosti třídění. Opět budeme předpokládat speciální typ vstupů, totiž permutace množiny $\{1, \ldots, n\}$. Různé permutace je přitom potřeba třídit různými posloupnostmi prohození, takže algoritmus musí správně rozpoznat, o kterou permutaci se jedná.

I zde funguje intuitivní úvaha o množství informace: jelikož jsou všechny prvky na vstupu různé, jedním porovnáním získáme nejvýše 1 bit informace. Všech permutací je $n!$, takže potřebujeme získat $\log_2(n!)$ bitů. Zbytek zařídí následující lemma:

Lemma F: $n! \geq n^{n/2}$.

Důkaz: Je $n! = \sqrt{(n!)^2} = \sqrt{1 \cdot n \cdot 2 \cdot (n-1) \cdot \ldots \cdot n \cdot 1}$, což můžeme také zapsat jako $\sqrt{1 \cdot n} \cdot \sqrt{2 \cdot (n-1)} \cdot \ldots \cdot \sqrt{n \cdot 1}$. 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 $n! \geq (n^{1/2})^n = n^{n/2}$. □

Počet potřebných bitů, a tím pádem i potřebných porovnání, tedy musí být $\log_2(n!) \geq \log_2(n^{n/2}) = (n/2) \cdot \log_2 n = \Omega(n \log n)$. Nyní totéž precizněji...

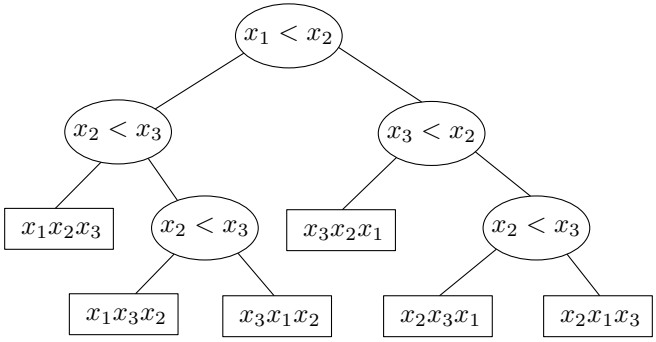
Věta (o složitosti třídění): Každý deterministický algoritmus v porovnávacím modelu, který třídí n-prvkovou posloupnost, použije v nejhorším případě $\Omega(n \log n)$ porovnání.

Důkaz: Jak už jsme naznačili, budeme uvažovat vstupy $a_1, \ldots, a_n$, které jsou permutacemi množiny $\{1, \ldots, n\}$. Stačí nám najít jeden „těžký“ vstup, pokud ho najdeme mezi permutacemi, úkol jsme splnili.

Mějme nějaký třídicí algoritmus. Upravíme ho tak, aby nejprve prováděl všechna porovnaní, a teprve pak prvky přesouval. Můžeme si například pro každou pozici v poli pamatovat, kolikátý z prvků a_i na ni zrovna je. Průběžné prohazování bude pouze měnit tyto pomocné údaje a skutečná prohození provedeme až na konci.

Nyní sestrojíme rozhodovací strom popisující všechny možné průběhy algoritmu. Ve vnitřních vrcholech budou porovnání typu $a_i < a_j$ se třemi možnými výsledky. Opět vynecháme výsledky, které jsou ve sporu s předchozími porovnáními. V listech stromu algoritmus provede nějaká prohození a zastaví se.

Pro různé vstupní permutace musí výpočet skončit v různých listech (dvě permutace nelze setřídit toutéž posloupností přesunů prvků). Listů je tedy alespoň $n!$, takže podle lemmatu **T** musí mít strom hloubku alespoň $\log_3(n!)$, což je podle lemmatu **F** $\Omega(n \log n)$. Proto musí existovat vstup, na němž se provede $\Omega(n \log n)$ porovnání. □



Obr. 1.1: Příklad rozhodovacího stromu pro 3 prvky

Dokázali jsme tedy, že algoritmus Mergesort je optimální (až na multiplikativní konstantu). Brzy ale uvidíme, že oprostíme-li se od porovnávacího modelu, lze v některých případech třídit rychleji.

Cvičení

1. Jsou dány rovnoramenné váhy a 3 různě těžké kuličky. Ukažte, že je není možné uspořádat dle hmotnosti na méně než 3 vážení. Co se změní, pokud je cílem pouze najít nejtěžší kuličku?
2. Jsou dány rovnoramenné váhy a 12 kuliček, z nichž právě jedna je těžší než ostatní. Na misku lze dát i více kuliček naráz. Navrhněte, jak na 3 vážení najít těžší kuličku. Dokažte, že na 2 vážení to není možné.
3. Jsou dány rovnoramenné váhy a 12 kuliček, z nichž právě jedna je jiná než ostatní, nevíme však zda je lehčí nebo těžší. Na misku lze dát i více kuliček naráz. Navrhněte, jak na 3 vážení najít tuto jinou kuličku a zjistit, jestli je lehčí nebo těžší. Dokažte, že na 2 vážení to nejde.
4. Stejná úloha jako předchozí, avšak s 13 kuličkami. Dokažte, že stále stačí 3 vážení, pokud slevíme z požadavku zjistit, zda je odlišná kulička lehčí nebo těžší.
5. Řešte cvičení 2 obecně pro n kuliček a navrhněte algoritmus používající co nejmenší počet vážení. Dokažte, že tento počet je optimální.

6.* Řešte cvičení 3 obecně pro n kuliček. Uměli byste dokázat, že váš algoritmus je optimální?

7. Uvažme verzi komparátoru, který rozlišuje pouze $a_i \leq a_j$ a $a_i > a_j$. Projděte algoritmy a důkazy vět v této kapitole a modifikujte je pro tento model.
8. *Průměrná složitost:* Dokažte, že $\Omega(\log n)$ resp. $\Omega(n \log n)$ porovnání je potřeba nejen v nejhorším případě, ale i v průměru přes všechny možné vstupy. V případě vyhledávání průměrujeme přes všechna možná hledaná x, u třídění přes všechny permutace.
9. *Matice:* Mějme matici A tvaru $n \times n$, v níž jsou uložena celá čísla a navíc každý řádek i sloupec tvoří rostoucí posloupnost. Jak najít i, j takové, že $A_{i,j} = i + j$? Pokud existuje více řešení, stačí vypsat jedno. Čas na načtení matice do paměti nepočítáme.

10.* Dokažte, že vaše řešení předchozí úlohy je asymptoticky nejrychlejší možné.

1.4. Přihrádkové třídění

Dosud jsme se snažili navrhovat třídicí algoritmy tak, aby si poradily s libovolným typem dat. Proto jsme prvky posloupnosti byli ochotni pouze porovnávat. Nyní se zaměříme na konkrétní druhy dat, například celá kladná čísla z předem daného intervalu.

Counting sort – třídění počítáním

Představme si, že třídíme n celých čísel vybraných z množiny $\{1, \ldots, r\}$ pro nepříliš velké r. Jak vypadá setříděná posloupnost? Nejdřív v ní jsou nějaké jedničky, pak dvojky, atd. Stačí tedy zjistit, kolik ma být kterých, čili spočítat, kolikrát se každé číslo od 1 do r na vstupu vyskytuje. Tomuto primitivnímu, ale překvapivě účinnému algoritmu se říká *třídění počítáním* neboli *Counting sort.*

Algoritmus CountingSort (třídění počítáním)
Vstup: Posloupnost $x_1, \ldots, x_n \in \{1, \ldots, r\}$

1. Pro $i = 1, \ldots, r$: ◁ *Inicializujeme počítadla*
2. $\quad p_i \leftarrow 0$
3. Pro $i = 1, \ldots, n$: ◁ *p_j bude počet výskytů čísla j*
4. $\quad p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$ ◁ *pozice ve výstupu*
6. Pro $i = 1, \ldots, r$: ◁ *zapisujeme výstup*
7. $\quad$ Opakujeme p_i-krát:
8. $\quad\quad v_j \leftarrow i$
9. $\quad\quad j \leftarrow j + 1$

Výstup: Setříděná posloupnost $v_1, \ldots, v_n$

Inicializace počítadel v krocích 1 a 2 trvá $\Theta(r)$, počítání výskytů v dalších dvou krocích $\Theta(n)$. V krocích 6 až 9 znovu zapíšeme všech n prvků a navíc musíme jednou sáhnout na každou (i prázdnou) přihrádku. Časová složitost celého algoritmu tedy činí $\Theta(n + r)$.

Vstup a výstup mohou být uloženy v tomtéž poli, takže pracovní prostor algoritmu tvoří pouze r počítadel, tedy $\Theta(r)$ buněk paměti.

Bucketsort – přihrádkové třídění

Counting sort nám nepomůže, pokud místo celých čísel třídíme nějaké složitější záznamy, které kromě celočíselného *klíče,* podle nějž třídíme, obsahují navíc nějaká další data. Tehdy můžeme použít podobný algoritmus, kterému se říká *přihrádkové třídění (Bucketsort).*⟨2⟩ Místo počítadel si pořídíme pole r přihrádek $P_1, \ldots, P_r$. V i-té přihrádce se bude nacházet seznam záznamů s klíčem i.

Algoritmus nejprve projde všechny záznamy a rozmístí je do přihrádek podle klíčů. Poté postupně projde přihrádky od P_1 do P_r a vypíše jejich obsah.

Algoritmus BucketSort (přihrádkové třídění)
Vstup: Prvky $x_1, \ldots, x_n$ s klíči $k_1, \ldots, k_n \in \{1, \ldots, r\}$

1. Inicializujeme přihrádky: $P_1, \ldots, P_r \leftarrow \emptyset$
2. Pro $i = 1, \ldots, n$:
3. $\quad$ Vložíme x_i do P_{k_i}.
4. Vytvoříme prázdný seznam S.
5. Pro $j = 1, \ldots, r$:
6. $\quad$ Na konec seznamu S připojíme obsah P_j.

Výstup: Setříděný seznam S

Rozbor časové složitosti proběhne podobně jako u předchozího algoritmu: inicializace stojí $\Theta(r)$, rozmisťování do přihrádek $\Theta(n)$, procházení přihrádek $\Theta(r)$ a vypisování všech přihrádek dohromady $\Theta(n)$. Celkem tedy $\Theta(n + r)$.

⟨2⟩ To bychom mohli přeložit jako „kbelíkové třídění“.

V paměti máme uložené všechny přihrádky, jedna zabere konstantní prostor na hlavičku seznamu a pak konstantní na každý záznam. To celkem činí $\Theta(n + r)$ buněk paměti.

Bucketsort dokonce může třídit stabilně. K tomu stačí, abychom v každé přihrádce dodrželi vzájemné pořadí prvků. Nové záznamy tedy budeme přidávat na konec seznamu.

Lexikografický Bucketsort

Nyní uvažme případ, kdy klíče nejsou malá celá čísla, nýbrž uspořádané k-tice takových čísel. Úkolem je seřadit tyto k-tice *lexikograficky (slovníkově):* nejprve podle první souřadnice, v případě shody podle druhé, a tak dále.

Praktičtější je postupovat opačně: nejprve k-tice setřídit podle poslední souřadnice, pak je stabilně setřídit podle předposlední, ... až nakonec podle první. Díky stabilitě získáme lexikografické pořadí. Stačí tedy k-krát aplikovat předchozí algoritmus přihrádkového třídění.

Zadaná posloupnost:	173, 753, 273, 351, 171, 172, 059
Po 1. průchodu:	351, 171, 172, 173, 753, 273, 069
Po 2. průchodu:	351, 753, 069, 171, 172, 173, 273
Po 3. průchodu:	069, 171, 172, 173, 273, 351, 753

Obr. 1.2: Příklad lexikografického přihrádkového třídění trojic

Algoritmus LexBucketSort
Vstup: Posloupnost k-tic $x_1, \ldots, x_n \in \{1, \ldots, r\}^k$

1. $S \leftarrow x_1, \ldots, x_n$
2. Pro $i = k, k-1, \ldots, 1$:
3. Setřídíme S BucketSortem podle i-té souřadnice.

Výstup: Lexikograficky setříděná posloupnost S

Nyní dokážeme korektnost tohoto algoritmu. Pro přehlednost budeme písmenem ℓ značit, v kolikátém průchod cyklem jsme. Bude tedy $\ell = k - i + 1$.

Lemma: 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. Spouštíme $(\ell + 1)$-ní průchod, tj. budeme třídit podle $(i-1)$-ní souřadnice. Protože Bucketsort třídí stabilně, zůstanou prvky se stejnou $(i-1)$-ní souřadnicí vůči sobě seřazeny tak, jak byly seřazeny na vstupu. Z indukčního předpokladu 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 k-násobkem složitosti Bucketsortu, tedy $\Theta(k\cdot(n+r))$. Paměti spotřebujeme $\Theta(nk+r)$: první člen je paměť zabraná samotnými záznamy, druhý paměť potřebná na uložení pole přihrádek.

Radixsort

Přihrádkové třídění pro klíče z rozsahu $1,\ldots,r$ není efektivní, pokud r je řádově větší než počet záznamů. Tehdy totiž časové složitosti vévodí čas potřebný na inicializaci a procházení přihrádek. Můžeme si ale pomoci následovně.

Čísla zapíšeme v soustavě o vhodném základu z. Z každého čísla se tak stane k-tice cifer z rozsahu $0,\ldots,z-1$, kde $k=\lfloor\log_z r\rfloor+1$. Tyto k-tice pak stačí setřídit lexikograficky, což zvládneme k-průchodovým přihrádkovým tříděním v čase $\Theta((\log_z r)\cdot(n+z))=\Theta(\frac{\log r}{\log z}\cdot(n+z))$.

Jak zvolit základ z? Pokud bychom si vybrali konstantní (třeba pokaždé čísla zapisovali v desítkové soustavě), časová složitost by vyšla $\Theta(\log r\cdot n)$. To není moc zajímavé, jelikož pro navzájem různé klíče máme $r\geq n$, takže jsme nepřekonali složitost porovnávacích algoritmů.

Užitečnější je zvolit $z=\Theta(n)$. Pak dosáhneme složitosti $\Theta(\frac{\log r}{\log n}\cdot n)$. Pokud by čísla na vstupu byla polynomiálně velká vzhledem z n, tedy $r\leq n^\alpha$ pro nějaké pevné α, byl by $\log r\leq\alpha\log n$, takže časová složitost by vyšla lineární. Polynomiálně velká celá čísla jde tedy třídit v lineárním čase (a také lineárním prostoru).

Tomuto algoritmu se říká *číslicové třídění* neboli *Radixsort.*⟨3⟩

Třídění řetězců

Myšlenku víceprůchodového přihrádkového třídění použijeme ještě k řazení řetězců znaků. Chceme je uspořádat lexikograficky – to definujeme stejně jako pro k-tice, jen navíc musíme říci, že pokud při porovnávání jeden řetězec skončí dřív než druhý, ten kratší bude menší.

Na vstupu jsme tedy dostali nějakých n řetězců $r_1,\ldots,r_n$ délek po řadě $\ell_1,\ldots,\ell_n$. Navíc označme $\ell=\max_i\ell_i$ délku nejdelšího řetězce a $s=\sum_i(\ell_i+1)$ celkovou velikost vstupu (+1 kvůli uložení délky řetězce – musíme umět uložit i prázdný řetězec). Budeme předpokládat, že znaky abecedy máme očíslované od 1 do nějakého r.

Kdyby všechny řetězce byly stejně dlouhé, stačilo by je třídit jako k-tice. To by mělo složitost $\Theta(\ell n)=\Theta(s)$, tedy lineární v celkové velikosti vstupu.

S různě dlouhými řetězci bychom se mohli vypořádat tak, že bychom všechny doplnili mezerami na stejnou délku (mezerou myslíme nějaký znak, který je menší než všechny ostatní znaky). Složitost algoritmu bude nadále $\Theta(\ell n)$, ale to v některých případech může být i kvadratické ve velikosti vstupu. Uvažme třeba případ s t řetězci

⟨3⟩ *Radix* je latinsky kořen, ale zde se tím myslí základ poziční číselné soustavy.

délky 1 a jedním délky t. To je celkem $n = t + 1$ řetězců o celkové délce $s = 2t - 1$. Třídění ovšem zabere čas $\Theta(\ell n) = \Theta(t^2) = \Theta(s^2)$.

Okamžitě vidíme, kde se většina času tráví: přidáváme ohromné množství mezer a pak ve většině průchodů většinu řetězců házíme do přihrádky odpovídající mezeře. Zkusme algoritmus vylepšit, aby mezery přidával jen pomyslně a řetězce, které jsou příliš krátké, v počátečních průchodech vůbec neuvažoval.

Začneme tím, že řetězce roztřídíme Bucketsortem do přihrádek (množin) P_j podle délky. V j-té přihrádce tedy skončí řetězce délky j. Při tom také spočítáme ℓ.

Pak budeme provádět ℓ průchodů přihrádkového třídění pro $i = \ell, \ell - 1, \ldots, 1$. Během nich budeme udržovat seznam Z tak, aby na konci průchodu pro konkrétní i obsahoval všechny řetězce s délkou alespoň i, a to seřazené podle i-tého až posledního znaku. Na konci posledního průchodu tedy budou v Z všechny řetězce seřazené lexikograficky podle všech znaků.

Zbývá popsat, jak jeden průchod pracuje. Použijeme přihrádky indexované znaky abecedy (Q_1 až Q_r) a budeme do nich rozhazovat řetězce podle jejich i-tého znaku. Nejprve rozházíme řetězce délky i z množiny P_i a pak přidáváme všechny, které zůstaly v seznamu Z z předchozího průchodu – kratší řetězce se tak dostaly před delší, jak má být. Nakonec všechny přihrádky vysbíráme a řetězce naskládáme do nového seznamu Z. Indukcí můžeme nahlédnout, že seznam Z splňuje požadované vlastnosti.

Ještě dodejme, že řetězce do přihrádek nebudeme kopírovat celé, znak po znaku. Postačí ukládat ukazatele a samotné řetězce nechávat netknuté ve vstupní paměti.

Algoritmus TŘÍDĚNÍŘETĚZCŮ
Vstup: Řetězce $r_1, \ldots, r_n$ délek $\ell_1, \ldots, \ell_n$ nad abecedou $\{1, \ldots, r\}$

1. $\ell \leftarrow \max(\ell_1, \ell_2, \ldots, \ell_n)$ ◁ *maximální délka*
2. Pro $i \leftarrow 1, \ldots, \ell$: ◁ *rozdělíme podle délek*
3. $\quad P_i \leftarrow \emptyset$
4. Pro $i \leftarrow 1, \ldots, n$ opakujeme:
5. $\quad$ Na konec P_{ℓ_i} přidáme řetězec r_i.
6. $Z \leftarrow \emptyset$ ◁ *výsledek předchozího průchodu*
7. Pro $i \leftarrow \ell, \ell - 1, \ldots, 1$: ◁ *průchody pro jednotlivé délky*
8. $\quad$ Pro $j \leftarrow 1, \ldots, r$: ◁ *inicializace přihrádek*
9. $\quad\quad Q_j \leftarrow \emptyset$
10. $\quad$ Pro řetězce u z přihrádky P_i: ◁ *nové, kratší řetězce*
11. $\quad\quad$ Na konec $Q_{u[i]}$ přidáme řetězec u.
12. $\quad$ Pro řetězce v ze seznamu Z: ◁ *řetězce z minulého průchodu*
13. $\quad\quad$ Na konec $Q_{v[i]}$ přidáme řetězec v.
14. $\quad Z \leftarrow \emptyset$ ◁ *vysbíráme přihrádky*
15. $\quad$ Pro $j \leftarrow 1, \ldots, r$:
16. $\quad\quad$ Pro řetězce w z přihrádky Q_j:

17. Na konec seznamu Z přidáme řetězec w.

Výstup: Setříděný seznam řetězců Z

Ještě stanovíme časovou složitost. Seřazení řetězců podle délky potrvá $\Theta(n+\ell)$. Průchod pro dané i spotřebuje r kroků na práci s přihrádkami a sáhne na ty řetězce, které mají délku alespoň i, tedy mají na pozici i nějaký znak.

V součtu přes všech ℓ průchodů tedy trvá práce s přihrádkami $\Theta(\ell r)$ a práce s řetězci $\Theta(\sum_i \ell_i) = \Theta(s)$ – každé sáhnutí na řetězec můžeme naúčtovat jednomu z jeho znaků.

Celý algoritmus proto běží v čase $\Theta(n + \ell r + s)$. Pokud je abeceda konstantně velká a vstup neobsahuje prázdné řetězce, můžeme tuto funkci zjednodušit na $\Theta(s)$. Algoritmus je tedy lineární ve velikosti vstupu.

Paměti spotřebujeme $\Theta(n + s)$ na řetězce a $\Theta(\ell + r)$ na přihrádky. To můžeme podobně zjednodušit na $\Theta(s)$.

Cvičení

1. *Rekurzivní Bucketsort:* Lexikografické třídění k-tic by se také dalo provést metodou Rozděl a panuj z kapitoly **??**: nejprve záznamy rozdělit do přihrádek podle první souřadnice a pak každou přihrádku rekurzivně setřídit podle zbývajících $k-1$ souřadnic. Jakou časovou a prostorovou složitost by měl takový algoritmus?
2. *Bucketsort v poli:* V implementaci Bucketsortu může být nešikovné ukládat přihrádky jako spojové seznamy, protože spotřebujeme hodně paměti na ukazatele. Upravte Bucketsort, aby si vystačil se vstupním polem, výstupním polem a jedním nebo několika r-prvkovými poli.
3. Může se někdy v Radixsortu vyplatit zvolit základ soustavy řádově větší než počet čísel?
4. *Třídění floatů:* Uvažujme čísla typu *floating point* zadaná ve tvaru $m \cdot 2^e$, kde m je celočíselná *mantisa*, e celočíselný *exponent* a platí $2^{24} \leq m < 2^{25}$, $-128 \leq e < 128$. Ověřte, že tato čísla lze ukládat do 32 bitů paměti. Rozmyslete si, jaký rozsah a přesnost mají. Navrhněte co nejrychlejší algoritmus na jejich třídění.
5.* *Velké abecedy:* Algoritmus TříděníŘetězců není efektivní pro velké abecedy, protože tráví příliš mnoho času přeskakováním prázdných přihrádek. Pokuste se předem předpočítat, ve kterém průchodu budou potřeba které přihrádky. Dosáhněte složitosti $\mathcal{O}(r + s)$.
6.** *Dírou* v množině $\{x_1, \ldots, x_n\} \subset \mathbb{Z}$ nazveme dvojici (x_i, x_j) takovou, že $x_i < x_j$ a žádné jiné x_k neleží v intervalu $[x_i, x_j]$. Chceme nalézt největší z děr (s maximálním $x_j - x_i$). Setříděním množiny to jde snadno, ale existuje i lineární algoritmus založený na šikovném dělení do přihrádek. Zkuste na něj přijít.

1.5. Přehled třídicích algoritmů

Na závěr uvedeme přehlednou tabulku se souhrnem informací o jednotlivých

algoritmus	*čas*	*pomocná paměť*	*stabilní*
Insertsort (1.1)	$\Theta(n^2)$	$\Theta(1)$	+
Bubblesort (1.1)	$\Theta(n^2)$	$\Theta(1)$	+
Mergesort (1.2, **??**)	$\Theta(n \log n)$	$\Theta(n)$	+
Heapsort (**??**)	$\Theta(n \log n)$	$\Theta(1)$	−
Quicksort (**??**, **??**)	$\Theta(n \log n)$	$\Theta(\log n)$	−
Bucketsort (1.4)	$\Theta(n + r)$	$\Theta(n + r)$	+
Bucketsort pro k-tice (1.4)	$\Theta(k \cdot (n + r))$	$\Theta(n + r)$	+
Radixsort (1.4)	$\Theta(n \log_n r)$	$\Theta(n)$	+
Bucketsort pro řetězce (1.4)	$\Theta(s)$	$\Theta(s)$	+

Obr. 1.3: Přehled třídicích algoritmů
(n je počet prvků, r rozsah klíčů, s délka vstupu)

třídicích algoritmech. Přidáme do ní i několik algoritmů, které představíme až v budoucích kapitolách. U všech algoritmů uvádíme číslo oddílu, kde jsou vyloženy.

Poznámky k tabulce:

- Quicksort má časovou složitost $\Theta(n \log n)$ pouze v průměru. Můžeme ale říci, že porovnáváme průměrné časové složitosti, protože u ostatních algoritmů vyjdou stejně jako jejich časové složitosti v nejhorším případě.
- Mergesort jde implementovat s konstantní pomocnou pamětí za cenu konstantního zpomalení, ovšem konstanta je neprakticky velká. Dále viz cvičení 1.2.5.
- Quicksort se dá naprogramovat stabilně, ale potřebuje lineárně pomocné paměti.
- Multiplikativní konstanta u Heapsortu není příliš příznivá a v běžných situacích tento algoritmus na celé čáře prohrává s efektivnějším Quicksortem.

Cvičení

1. Navrhněte algoritmus na zjištění, jestli se v zadané n-prvkové posloupnosti opakují některé prvky.

2.* Dokažte, že problém z předchozí úlohy vyžaduje v porovnávacím modelu čas alespoň $\Theta(n \log n)$.

3. Je dána posloupnost čísel. Najděte nejdelší úsek, v němž se žádné číslo neopakuje. (To je podobné cvičení **??**, ale zde není omezen rozsah prvků.)