1. Vyhledávání a třídění

1.1. Úvod do problematiky

V následující kapitole se budeme zabývat problémem, který je pro programátory každodenním chlebem – vyhledávání a třídění údajů. Třídění je ovšem poněkud zavádějící pojem. Nechceme údaje rozdělovat do různých tříd, nebo je jinak klasifikovat, ale našim cílem bude data seřadit. Správný termín by měl být tedy spíše *řazení*, avšak slovo třídění se již vžilo do českého názvosloví natolik, že jej budeme používat i zde.

Než se začneme věnovat reálným třídícím algoritmům, prozkoumáme následující problém. Jsou dány rovnoramenné váhy a 3 kuličky a, b, c různých hmotností. Na každou misku vah můžeme položit jednu kuličku a váhy dají odpověd $<, =, >$ podle toho, zda první kulička je lehčí, stejně těžká nebo těžší než druhá kulička. Úkolem je uspořádat kuličky a, b, c podle velikosti. Kolik nejméně vážení k tomu budeme potřebovat? A co se změní, pokud je úkolem pouze najít nejtěžší kuličku?

Porovnejme nejprve a s b. Pokud vyšlo $a < b$ nebo $a = b$, jsou kuličky v pořadí (a, b). V opačném případě jsou kuličky v pořadí (b, a). Předpokládejme nyní, že například vyšla druhá možnost. Do pořadí (b, a) nyní musíme zařadit kuličku c. Porovnáme c s a, pokud vyjde $>$ nebo $=$, je správné pořadí (b, a, c). V opačném případě je třeba ještě jedním vážením rozhodnout, zda správně pořadí je (b, c, a) nebo (c, b, a).

Čtenář nechť si povšimne, že na méně než tři porovnání nelze úlohu řešit: pro každý výsledek porovnání dvou prvků, které algoritmus porovná jako první, totiž ještě musí nutně proběhnout aspoň jedno vážení obsahující zbývající třetí prvek, které však může dopadnout tak, že neposkytne plnou informaci o vztahu třetího prvku s dvěma prvky váženými nejdříve. Algoritmus používající jen 2 vážení tedy nemůže existovat a náš algoritmus je tedy nejrychlejší možný.

Pokud je úkolem pouze najít nejtěžší kuličku, zvážíme a a b, zapamatujeme si tu těžší, se kterou ještě porovnáme kuličku c; těžší kulička tohoto vážení je výsledek. Máme tedy algoritmus na 2 vážení, který je zjevně nejrychlejší možný, protože při jediném vážení by se mohla nejtěžší kulička skrývat v té, kterou jsme nezvážili.

Úloha s kuličkami je velmi jednoduchá a jistě ji dokáže vymyslet i čtenář bez znalosti pokročilých třídících algoritmů. Přes svou jednoduchost však slouží jako dobrá ilustrace problémů, které budeme v následujícím textu potkávat: je dán *komparátor* (v našem případě rovnoramenné váhy) a prvky, které umíme pouze porovnávat komparátorem a žádné další operace s nimi neumíme. Tomuto modelu se říká *porovnávací model třídění*. ⟨1⟩

⟨1⟩ Pro znalce knihovních funkcí některých vyšších programovacích jazyků určených k třídění dat dodáváme, že tyto funkce typicky vyžadují jako argument porovnávací funkci dvou prvků – to je přesně komparátor našeho modelu.

Protože neporovnatelné prvky by se obtížně znázorňovaly při popisech algoritmů této kapitoly, budeme v ukázkách typicky třídit celá čísla a na čtenáři ponecháme, aby si místo nich představil obecné prvky. Při odhadování časových složitostí také předpokládáme, že jednotlivé prvky lze porovnávat i prohazovat v konstantním čase ($\Theta(1)$).

Dále bychom měli zdůraznit, že v této kapitole se věnujeme pouze problémům *vnitřního* vyhledávání a třídění. U nich předpokládáme, že všechny tříděné prvky, se nacházejí ve vnitřní paměti počítače (RAM) a máme k nim přímý přístup v poli v čase $\Theta(1)$ pro čtení i zápis.

U třídících algoritmů nás budou zajímat i jiné vlastnosti, než jen jejich časové nároky.

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 významnějších.)

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.

Stručně projdeme, co je obsahem této kapitoly. Na začátku se podíváme na problém vyhledávání údajů. Jak zjistíme, v setříděných datech se hledá daleko lépe než v nesetříděných, a tak, abychom mohli data uspořádat, probereme nejjednodušší algoritmy třídění, konkrétně algoritmy pracující v kvadratickém čase. Ty jsou bohužel pro reálně použití na větších datech velmi pomalé.

Existují však rychlejší třídící algoritmy pracující v čase $\mathcal{O}(N \log N)$. Mezi zástupce těchto algoritmů patří například HeapSort, neboli třídění haldou (protože vyžaduje znalost datové struktury halda, popisujeme ho v kapitole o haldách), QuickSort a MergeSort (ty jsou typickými představiteli algoritmů založených na myšlence Rozděl & Panuj a popisujeme je opět v příslušné kapitole).

Nabízí se přirozená otázka, zda lze třídit rychleji než v čase $\mathcal{O}(N \log N)$. Ukážeme, že odpověď je negativní a dokážeme, že rychlejší algoritmus pro třídění prvků, které mezi sebou umíme pouze porovnávat, nemůže existovat. Zdůrazněme však, že tuto skutečnost umíme dokázat pouze v případě, že s tříděnými daty nelze provádět nic jiného než porovnání. Pokud mají tříděné prvky určité speciální vlastnosti (například jsou to malá čísla), můžeme na ně použít algoritmy, které již nejsou založené výhradně na porovnávání, a dopracovat se až ke složitosti $\Theta(N)$.

Cvičení:

1. 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 porovnání najít těžší kuličku.
2. 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 porovnání najít tuto jinou kuličku.

3. Stejná úloha jako předchozí, avšak s 13 kuličkami.
4. Řešte úlohu 1.1.1 obecně pro N kuliček a navrhněte algoritmus používající co nejmenší počet vážení.
5. Řešte úlohu 1.1.2 obecně pro N kuliček a navrhněte algoritmus používající co nejmenší počet vážení.
6. Dokažte, že každé řešení úloh 1.1.4 a 1.1.5 musí nutně provést alespoň $\lceil \log_3 N \rceil$ vážení.

1.2. Vyhledávání údajů v poli

Vyhledávání v nesetříděném poli

Hledání čísla v nesetříděném poli probíhá úplně stejně jako hledání předmětu v neuklizeném pokoji. Zkrátka se musíme podívat na všechna místa, zda tam předmět není. Pokud při procházení pole narazíme na hledaný prvek, vrátíme jeho index a algoritmus končí. V případě, že daný prvek v poli není, je třeba ho projít celé, jinak nemáme jistotu, že se nám prvek někde neschovává.

Algoritmus SEARCH1
Vstup: Pole $P[1 \ldots N]$, hledaný prvek x.
Výstup: Index i hledaného prvku, případně 0, pokud prvek v poli není.

1. Pro i jdoucí od 1 do N opakujeme:
2. Pokud je $P[i] = x$: vrátíme i a skončíme.
3. Vrátíme 0. *(prvek v poli není)*

Na první pohled by se mohlo zdát, že reprezentace dat nesetříděným polem není příliš výhodná. Časová složitost hledání jakéhokoli prvku je $\mathcal{O}(N)$ (tedy nejhorší možná. Na druhou stranu nesetříděné pole nevyžaduje žádnou údržbu. Nové prvky přidáme jednoduše na konec a nemusíme se starat o jejich uspořádání. Při mazání prvku z prostředka pole můžeme jednoduše zalepit vzniklou díru přesunutím posledního prvku na místo smazaného. Tím docílíme časové složitosti $\Theta(1)$ na přidávání a odebírání prvků, takže údržba nás nebude stát v podstatě nic.

Vyhledávání v setříděném poli

Nyní si představme, že nám v poli někdo uklidil a všechny prvky jsou úhledně seřazené (bez újmy na obecnosti řekněme vzestupně). To by nám mělo pomoci při hledání prvků. Daní za pohodlnější vyhledávání však bude složitější údržba. Vložení nového prvku už nemůže být tak ledabylé jako v předchozím případě. Nejprve musíme nalézt správnou pozici, kam prvek patří (abychom neporušili uspořádání), a následně mu ještě musíme vytvořit místo tak, že všechny větší prvky posuneme o jednu pozici dále. S mazáním je obdobný problém, protože nestačí pouze zalepit vzniklou díru libovolným prvkem, ale všechny prvky, které jsou větší než odstraňovaný, se musí opět posunout o jednu pozici. Díky tomu se nám zhoršila časová

složitost vkládání i odebírání prvků na lineární. Pokud ale budeme v poli mnohem častěji vyhledávat než přidávat a odebírat prvky, může se nám to celkově vyplatit.

Podívejme se, jak by nám mohlo setřídění pomoci při hledání prvku. Nejprve zkusíme přímočarý postup, který jsme používali v poli nesetříděném. Pole budeme procházet od začátku do konce a pokud narazíme na hledaný prvek, můžeme skončit. Jediná výhoda se projeví v situaci, kdy se hledaný prvek v poli nevyskytuje. V takovém případě již nemusíme prohledávat pole celé, ale můžeme skončit v okamžiku, kdy narazíme na prvek, který je větší než hledaný. Od takového prvku dál se totiž budou v poli vykytovat jen stále větší a větší prvky, mezi kterými se ten hledaný už opravdu vyskytovat nemůže. Podívejme se na lehce upravený algoritmus:

Algoritmus SEARCH2
Vstup: Pole $P[1 \ldots N]$ setříděné vzestupně, hledaný prvek x.
Výstup: Index i hledaného prvku, případně 0, pokud prvek v poli není.

1. Pro i jdoucí od 1 do N opakujeme:
2. Pokud je $P[i] = x$, vrátíme i a skončíme.
3. Pokud je $P[i] > x$, vrátíme 0 a skončíme. *(prvek v poli není)*
4. Vrátíme 0.

I přes výhodu, kterou setříděnost přinesla do původního algoritmu vyhledávání, se nám nepodařilo vylepšit asymptotickou časovou složitost. Naštěstí existuje i jiný přístup, který lépe využije vlastností setříděného pole a díky tomu dosáhne lepší než lineární složitosti.

Binární vyhledávání

Opusťme myšlenku klasického vyhledávání postupným procházením a zkusme se na problém podívat trochu z jiného úhlu. Efektivně vyhledávat nemusí znamenat jen rychle určit, kde se prvek nachází, ale třeba také rychle vyloučit velké části pole, kde se prvek nacházet nemůže.

Například vyhledáváme-li určité slovo ve slovníku, zcela jistě neprocházíme slovníkem od začátku. Namísto toho otevřeme slovník někde uprostřed, podíváme se, jak moc blízko jsme se trefili k hledanému slovu, a na základě toho nadále aplikujeme stejný postup buďto v levé nebo pravé části rozevřeného slovníku.

Zkusme z této strategie vytvořit plnohodnotný algoritmus. Pole si rozdělíme na dvě téměř stejné[⟨2⟩] poloviny. Prostřední prvek, který funguje jako mezník oddělující tyto poloviny, označme s. Pokud bychom měli extrémní štěstí, byl by s zároveň hledaným prvkem a my bychom mohli vyhledávání ukončit. Takové náhody se ale často nestávají takže raději zkusíme určit, ve které polovině, by mohl hledaný prvek (označme ho x) být. Porovnáme-li x a s, můžeme dojít ke dvěma závěrům: Buď je $x < s$ a pak se hledaný prvek může nacházet pouze v první polovině pole, nebo je $x > s$ a pak bychom měli x hledat v polovině druhé. Tím jsme efektivně eliminovali alespoň polovinu dat k prohledávání.

[⟨2⟩] Jejich velikost se bude lišit maximálně o 1.

Ten samý postup můžeme použít znovu na zvolenou polovinu, čtvrtinu atd., až se dostaneme do stavu, že prohledávaný úsek pole má velikost jednoho prvku. Na něm už se snadno přesvědčíme, zda je tento jediný potenciální kandidát hledaným prvkem.

Algoritmus BinSearch

Vstup: Pole $P[1 \ldots N]$ setříděné vzestupně, hledaný prvek x.

Výstup: Index i hledaného prvku, případně 0, pokud prvek v poli není.

1. $l \leftarrow 1$, $r = N$ *($[l \ldots r]$ tvoří prohledávaný úsek pole)*
2. Dokud je $l < r$:
3. $\quad s \leftarrow \lfloor (l + r)/2 \rfloor$ *(střed prohledávaného intervalu)*
4. $\quad$ Pokud je $x = P[s]$: vrať s a skonči.
5. $\quad$ Pokud je $x > P[s]$:
6. $\quad\quad l \leftarrow s + 1$
7. $\quad$ jinak:
8. $\quad\quad r \leftarrow s$
9. Pokud je $l < N$ a zároveň $P[l] = x$ vrať l, jinak vrať 0.

Algoritmus je zcela jistě konečný, protože v každém kroku zmenšíme prohledávanou část pole alespoň o 1. Po celou dobu běhu algoritmu platí invariant, že hledaný prvek nemůže ležet vně úseku $[l \ldots r]$. Snadno tedy nahlédneme, že algoritmus nalezne náš prvek, nebo s jistotou oznámí, že tam žádný takový prvek není.

Na závěr ještě spočítejme, jakou bude mít binární vyhledávání časovou složitost. V každém kroku algoritmu úspěšně zamítneme alespoň polovinu z prohledávaného intervalu. Pokud má tento interval navíc lichou délku, zamítneme vždy více než polovinu prvků.

Předpokládejme na chvíli, že N je rovno mocnině dvojky. To znamená, že velikost intervalu bude klesat exponenciálně[3] $(N, N/2, N/4, N/8, \ldots)$. Po k krocích bude velikost $N/2^k$. Nás zajímá, kolik kroků potřebujeme, abychom dostali interval délky jedna. Položme $N/2^k = 1$, takže $N = 2^k$ a po zlogaritmování dostaneme $k = log_2 N$.

Pokud by N nebylo přímo rovné 2^k, můžeme např. pole doplnit zprava nekonečně velkými prvky na nejbližší vyšší mocninu dvojky. Tím určitě nezhoršíme časovou složitost víc než o jedno půlení (což je konstanta, kterou v asymptotickém odhadu složitosti zanedbáme).

V každém kroku vykonáme pouze konstantní práci, takže celková složitost binárního vyhledávání je $\Theta(\log N)$.

Dolní odhad složitosti vyhledávání

Skutečnost, že uvedený vyhledávací algoritmus měl časovou složitost $\mathcal{O}(\log N)$ není náhoda. Ukážeme, že lepšího času není možné dosáhnout.

[3] Ve skutečnosti se může interval v jednom kroku zmenšit ještě o jeden prvek navíc, pokud budeme mít štěstí. My ale počítáme nejhorší možnou variantu.

Věta: *(o složitosti vyhledávání)* Každý algoritmus založený na porovnávání prvků, který vyhledává prvek x v setříděném poli P, potřebuje provést alespoň $\Omega(\log N)$ operací.

Důkaz: Zvolme libovolný vyhledávací algoritmus A. Uvažme rozhodovací strom T, který popisuje chování algoritmu A. Strom T v každém vnitřním uzlu provede porovnání prvku x s nějakým prvkem pole P a v listech obsahuje výstupní indexy nalezeného prvku Strom T je ternární (výstup komparátoru je $<$, $>$ nebo $=$) a počet jeho listů je zjevně n.

Časová složitost algoritmu A je dána délkou hloubkou nejhlubšího listu stromu T, protože průchod stromem od kořene směrem dolů popisuje jeden průběh algoritmu. Ternární strom s n listy má však alespoň $\log_3 n$ hladin, což dává dolní odhad $\Omega(\log N)$ na dobu běhu algoritmu A. □

Cvičení:

1. Upravte vyhledávání v nesetříděném poli tak, aby algoritmus vrátil indexy všech výskytů (nejen prvního).
2. Zkuste totéž pro setříděné pole. V čem je pro nás setříděné pole lepší?
3. Rozmyslete si, jak se bude chovat algoritmus binárního vyhledávání, pokud bude hledaný prvek v poli víckrát. Následně ho upravte tak, aby vždy vracel první výskyt hledaného prvku (ne jen libovolný).
4. Mějme pole délky N. Na každé pozici se může vyskytovat libovolné celé číslo z rozsahu 1 až K. Čísla vybíráme rovnoměrně (všechny hodnoty můžeme vybrat se stejnou pravděpodobností). Následně pole setřídíme a budeme v něm chtít vyhledávat. Zkuste upravit binární vyhledávání, aby na takovém poli fungovalo v průměrném případě rychleji.

5.* Jakou časovou složitost bude mít takový algoritmus v průměrném případě?

6.* Může se stát, že výše uvedený algoritmus nedostane pěkná data. Můžeme mu nějak pomoci, aby nebyl ani v takovém případě o mnoho horší, než binární vyhledávání?

1.3. Základní třídící algoritmy

V předchozí kapitole jsme ukázali, že vyhledávání je mnohem příjemnější, když máme data setříděná, ale zatím nám zůstalo skryto tajemství třídění samotného. Třídící algoritmy patří do základního arsenálu ostříleného programátora a jejich implementace bývají součástí standardních knihoven běžných jazyků. I když se v dnešní době stává velice zřídka, že si musí člověk napsat nějaký třídící algoritmus svépomocí, je víc než důležité jim dobře rozumět.

Nejjednodušší třídící algoritmy patří do skupiny tzv. *přímých metod*. Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají kvadratickou časovou složitost ($\Theta(N^2)$). Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho.

Selectsort

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 není těžké naprogramovat. Budou nám k tomu stačit dva vnořené cykly:

Algoritmus SelectSort
Vstup: Pole $P[1 \ldots N]$.
Výstup: Setříděné pole P.

1. Pro i jdoucí od 1 do $N-1$ opakujeme:
2. $\quad m \leftarrow i$ *(m bude index nejmenšího dosud nalezeného prvku)*
3. $\quad$ Pro j jdoucí od $i+1$ do N opakujeme:
4. $\quad\quad$ Pokud je $P[j] < P[m]$: $m \leftarrow j$
5. $\quad$ Pokud $i \neq m$: Prohodíme prvky $P[i]$ a $P[m]$.

Závěrem doplňme ještě několik slov k časové složitosti. V i-tém kroku algoritmu hledáme minimum z $N-i+1$ čísel, na což potřebujeme $\Theta(N-i+1)$ kroků. Ve všech krocích dohromady tedy spotřebujeme čas $\Theta(N + (N-1) + \ldots + 3 + 2 + 1) = \Theta(N \cdot (N-1)/2) = \Theta(N^2)$.

Insertsort

Třídění přímým vkládáním (Insertsort) přistupuje k problému se stejnou jednoduchostí jako Selectsort, ale z opačné strany. Postup velmi pěkně ilustruje příklad s karetním hráčem. Každému hráči se mnohem lépe hraje, když má karty v ruce uspořádané (podle barev a velikostí). K vytvoření setříděné posloupnosti pak intuitivně používá třídění vkládáním. Rozdané karty leží před ním nesetříděné na hromádce. Hráč bere postupně karty z hromádky a vkládá si je do ruky, přičemž je vždy zařadí na správné místo.

Představme si opět algoritmus v poli. Stejně jako u třídění přímým výběrem i zde budeme udržovat dvě části – na začátku pole budou 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, přičemž se větší prvky posunou o jednu pozici dále, aby vytvořily pro nového kolegu místo. Formálně zapsaný algoritmus bude vypadat takto:

Algoritmus InsertSort
Vstup: Pole $P[1 \ldots N]$.
Výstup: Setříděné pole P.

1. Pro i jdoucí od 2 do N opakujeme:

2. $x \leftarrow P[i]$ *(Do x na chvíli uložíme zatřiďovaný prvek.)*
3. $j \leftarrow i - 1$ *(Index, kam nový prvek přijde.)*
4. Dokud $j < 0$ a zároveň $P[j] > x$:
5. $P[j+1] \leftarrow P[j]$ *(Zároveň posouváme prvky o 1 dál v poli.)*
6. $j \leftarrow j - 1$
7. $P[j+1] \leftarrow x$ *(Vložíme zatřiďovaný prvek na nalezené místo.)*

Časová složitost bude naprosto stejná, jako u předchozího algoritmu. V i-tém kroku bude zatřiďovaný prvek v nejhorším případě posunut až na první místo, na což je potřeba $\mathcal{O}(i)$ operací. Celková složitost tedy bude $\mathcal{O}(1+2+3+\ldots+(N-1)+N) = \mathcal{O}(N(N-1)/2) = \mathcal{O}(N^2)$.

Porovnejme nyní Insertsort s dříve uvedeným Selectsortem. Asymptotická složitost v nejhorším případě bude u obou algoritmů kvadratická, ale přesto nemusí být oba stejně rychlé. Zatím co výběr nejmenšího prvku trvá Selectsortu vždy stejně, zatřídění prvku na správné místo může zabrat potenciálně méně operací. V extrémním případě, kdy jsou data již setříděná, poběží Selectsort stále v čase $\Theta(N^2)$, zatímco Insertsort nebude potřebovat vůbec žádné zatřiďování, takže vystačí s časem $\Theta(N)$.

Abychom ale byli spravedliví, je zde i obrácená strana mince. Insertsort při zatřiďování prvky nejenom porovnává, ale také prohazuje. Tím může na jedno zatřídění spotřebovat až $\mathcal{O}(N)$ operací prohození, zatím co Selectsort prvky pouze porovnává a k prohození se uchýlí až v okamžiku, kdy nalezne minimum.

Nelze říci, že by byl jeden z těchto algoritmů vysloveně lepší než druhý. Skutečná efektivita bude záviset na charakteru tříděných dat.

Bubblesort

Poslední z trojice přímých algoritmů je *bublinkové třídění (Bubblesort)*. Tento algoritmus pracuje na jiném principu než jeho dva předchůdci. Základem je myšlenka nechat stoupat menší prvky v poli podobně jako stoupají bublinky v limonádě.

V algoritmu budeme opakovaně procházet celé pole. Jeden průchod (může být v libovolném směru) postupně porovná všechny dvojice sousedních prvků (i a $i+1$). Pokud dvojice není správně uspořádaná (tedy $P[i] > P[i+1]$), prohodíme oba prvky. V opačném případě necháme dvojici na pokoji. Menší prvky se nám tak posunou blíže k začátku pole zatímco větší prvky „klesají“ 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
Vstup: Pole $P[1 \ldots N]$.
Výstup: Setříděné pole P.

1. *změna* $\leftarrow$ *true* *(Bude hlídat, zda došlo k nějakému prohození.)*
2. Dokud je *změna* = *true*:
3. *změna* $\leftarrow$ *false*
4. Pro i jdoucí od 1 do $N-1$ opakujeme:

5. Pokud je $P[i] > P[i+1]$:
6. Prohodíme prvky $P[i]$ a $P[i+1]$.
7. *změna* $\leftarrow$ *true*

S odhadem časové složitosti to bude trochu komplikovanější než v předchozích případech. 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.

V algoritmu, který je popsaný výše, si lze všimnout jedné zajímavosti. Bez ohledu na uspořádání prvků bude platit, že po prvním průchodu se na posledním místě v poli ocitne maximum a žádný další průchod už s ním nepohne. Po druhém průchodu bude na předposledním místě druhý největší prvek atd. Nejpozději po $N-1$ krocích bude na druhém místě druhý největší prvek (tedy i na prvním místě minimum) a pole bude setříděné. Budeme také ještě potřebovat jeden průchod navíc, abychom ověřili, že už nedojde k žádným výměnnám. Algoritmus může skončit i dřív, ale v případě, že nám nepřítel zadá pole s prvky uspořádanými sestupně, budeme potřebovat všech N kroků, abychom ho setřídili. Celková složitost bude tedy $\Theta(N^2)$.

Na závěr dodejme, že Bubblesort patří k nejméně efektivním algoritmům. Zde jej uvádíme jen jako ukázku jiného přístupu a pěkné logické cvičení, neboť se v praxi téměř nepoužívá.

Cvičení:

1. 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 $\Theta(N \log N)$.
2. Určete, jakou složitost bude mít Insertsort, pokud víme, že se ve vstupním poli každý prvek nachází nejvýše ve vzdálenosti k od pozice, na které se tento prvek bude nacházet po setřídění. Přesněji pro každý prvek na pozici i ve vstupním poli zaveďme p_i jako pozici, kde se bude prvek nacházet po setřídění. Pak platí pro všechny prvky, že $|i - p_i| \leq k$.
3. Všimněte si, že Bubblesort může provádět spoustu zbytečných porovnání. Např. 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.

1.4. Dolní odhad složitosti problému třídění

Jak už jsme zmínili v úvodu, nabízí se otázka, zda existuje třídící algoritmus pro porovnávací model rychlejší než $\mathcal{O}(N \log N)$. Odpověď je negativní, každý takový algoritmus bude potřebovat $\Omega(N \log N)$ operací. Porovnání je také jedinou operací,

která nás bude zajímat v našem odhadu, protože snadno nahlédneme, že ostatních operací budeme potřebovat asymptoticky nejvýše stejné množství.

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

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.)

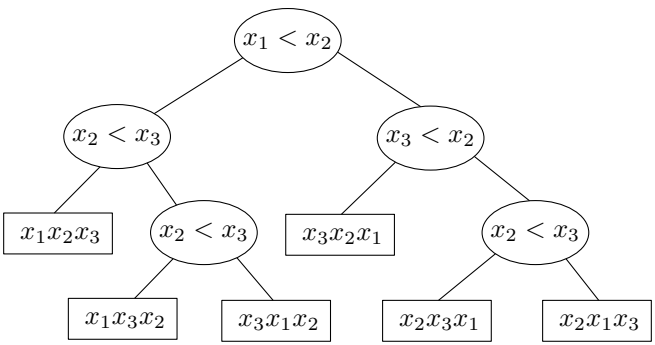
Mějme tedy deterministický algoritmus a nějaké pevné n. Sledujme, jak algoritmus porovnává – u každého porovnání zaznamenáme polohy porovnávaných prvků tak, jak byly na vstupu. Jelikož algoritmus je deterministický, porovná na začátku vždy tutéž dvojici prvků. Toto porovnání mohlo dopadnout třemi různými 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 $\Omega(n \log n)$.



Obr. 1.1: Příklad rozhodovacího stromu pro 3 prvky

Lemma 1: 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 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 Lemmatu 1 plyne, že rozhodovací strom musí být hluboký alespoň $\log_3 n!$.

Zbytek už je snadné cvičení z diskrétní matematiky:

Lemma 2: $n! \geq n^{n/2}$. *Důkaz:* Je $n! = \sqrt{(n!)^2} = \sqrt{1(n-1)\cdot 2(n-2)\cdot \ldots \cdot n\cdot 1}$, 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.

1.5. Lineární třídící algoritmy

Nyní již víme, že třídění založené na porovnávání nemůže dosáhnout lepší časové složitosti než $\Theta(N\log N)$, bez ohledu na to, jak moc se budeme snažit. Podívejme se, zda bychom nemohli složitost přeci jen vylepšit, když budeme na klíče klást větší nároky než jen možnost efektivního porovnávání.

Dosud jsme používali jako klíče celá čísla, abychom zpřehlednili popisované algoritmy a nemuseli se zabývat detaily porovnávání. V této kapitole budeme striktně vyžadovat, aby klíče byly celá kladná čísla z předem daného intervalu, případně aby byly na celá čísla snadno a jednoznačně převoditelné.

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 CountingSort
Vstup: Posloupnost $x_1, \ldots, x_N \in \{1, \ldots, R\}$

1. Pro $i = 1 \ldots R$ opakujeme:

2. $p_i \leftarrow 0$
3. Pro $i = 1 \dots N$ opakujeme:
4. $p_{x_i} \leftarrow p_{x_i} + 1$
5. $j \leftarrow 1$
6. Pro $i = 1 \dots R$ opakujeme:
7. Dokud $p_i > 0$, opakujeme:
8. $v_j \leftarrow i$
9. $j \leftarrow j + 1$
10. $p_i \leftarrow p_i - 1$
11. Vrátíme výsledek $v_1, \dots, v_N$.

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 $1, \dots, R$. Pořídíme si R přihrádek $P_1, \dots, P_R$, 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 BucketSort
Vstup: Prvky $x_1, \dots, x_n$ s klíči $c_1, \dots, c_n \in \{1, \dots, R\}$

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

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 Slovníkový BucketSort
Vstup: k-tice $x_1, \dots, x_n$
Výstup: k-rice lexikograficky setříděné

1. $S \leftarrow x_1, \dots, x_n$.
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 $1 \dots R$ 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

Mějme n řetězců $r_1, r_2 \dots r_n$ dlouhých $l_1, l_2 \dots l_n$ 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 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 problém 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 setříděný 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. $L \leftarrow \max(l_1, l_2, \ldots, l_n)$
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:
8. $\quad$ Pro $j \leftarrow 1$ do R opakuj:
9. $\quad\quad Q_j \leftarrow \emptyset$
10. $\quad$ Pro $j \leftarrow 1$ do velikost(P_i) opakuj:
11. $\quad\quad$ *vezmi*(P_i, r)
12. $\quad\quad$ *pridej*$(Q_{r[i]}, r)$
13. $\quad$ Pro $j \leftarrow 1$ do velikost(S) opakuj:
14. $\quad\quad$ *vezmi*(S, r)
15. $\quad\quad$ *pridej*$(Q_{r[i]}, r)$
16. $\quad S \leftarrow \emptyset$
17. $\quad$ Pro $j \leftarrow 1$ do R opakuj:
18. $\quad\quad$ Pro $k \leftarrow 1$ do velikost(Q_j) opakuj:
19. $\quad\quad\quad$ *vezmi*(Q_j, r)
20. $\quad\quad\quad$ *pridej*(S, r)
21. výsledek S

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

1.6. Přehled třídících algoritmů

Pro přehlednost uvádíme tabulku se souhrnem informací k jednotlivým třídícím algoritmům.

	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	$\Theta(n \log_n R)$	$\Theta(n)$	+

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 dostaneme 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.
- Multiplikativní konstanta u HeapSortu není příliš příznivá a v běžných situacích tento algoritmus na plné 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 čas alespoň $\Theta(N \log N)$.

1.7. Závěr

V této kapitole jsme pronikli do základů třídění a představili několik možných přístupů. Většina zmíněných algoritmů (vyjma kapitoly o přihrádkovém třídění) se ale v praxi příliš nepoužívá. Kvadratické algoritmy jsou vhodné pouze jako doplňková metoda pro třídění velmi malých polí. Heapsort, Mergesort a Quicksort již pracují

s časovou složitostí $\Theta(N \log N)$ a, jak jsme si ukázali v sekci 1.4, lepší složitosti u třídění založeném na porovnávání ani dosáhnout nelze.

U všech zmíněných algoritmů jsme také předpokládali, že tříděné prvky se vejdou do operační paměti. Při třídění velkých dat takový předpoklad již platit nemusí. V takovém případě je potřeba použít lehce upravené algoritmy, které jsou optimalizovány pro přístup do vnější paměti (např. na pevný disk).