ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA PEDAGOGICKÁ
Katedra matematiky

BAKALÁŘSKÁ PRÁCE

Některé řadící algoritmy

Dudek Martin

obor Matematická studia

Vedoucí práce: PhDr. Lukáš HONZÍK, Ph.D.

Plzeň 2018

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně za použití uvedeného zdroje informací, který je součástí této práce. Dále prohlašuji, že veškerý použitý software je legální.

V Plzni, 1.června 2018 .....

Podpis

Poděkování

Panu PhDr. Lukáši Honzíkovi, Ph.D., za jeho odborné vedení mé bakalářské práce.

Panu doc. RNDr. Přemyslu Holubovi, Ph.D., za jeho vedení mé první vysokoškolské práce a za jeho připomínky k celkové struktuře odborných prací.

Panu doc. Ing. Romanu Čadovi, Ph.D., za jeho informace k tématu algoritmů a za jeho vedení výuky KMA/DMB a KMA/TSI.

Paní RNDr. Blance Šedivé, Ph.D., za její vedení výuky softwaru LaTeX, který byl užit při vzniku této práce.

Mé přítelkyni, rodině a přátelům za jejich věčnou podporu a pomoc.

ZDE BUDE ZADÁNÍ BCP

Obsah kapitol

Obsah

1 Úvod

1.1 Cíl práce

Cílem této práce je uvést základní principy většiny řadících algoritmů a algoritmů obecně. Text by měl sloužit jako odrazový můstek pro čtenáře, který není zcela obeznámen s nutnými fakty z oboru informatiky. Zároveň je tento text psán s ohledem na čtenáře, kteří mají základní, ne-li žádné znalosti z diskrétní matematiky, nebo z teorie sítí.

Dále je cílem práce vyložit veškerou látku řadících algoritmů tak, aby bylo snadné ji pochopit. Mimo jiné je cílem aby čtenář byl schopen po přečtení těchto několika stránek pochopit kterýkoli algoritmus a vytvořit jej. Tvorba takového algoritmu se předpokládá alternativně v krocích, bez znalosti jakéhokoli programovacího jazyka, nebo softwaru.

V závěru bych se rád pokusil zodpovědět otázku, zda-li jsou řadící algoritmy důležité a k čemu vlastně slouží. Druhá otázka, kterou bych chtěl zodpovědět zní: "Existuje řadící algoritmus, který je jednoznačně nejlepší?"

Na závěr této kapitoly bych rád zdůraznil, že tento text neslouží k naučení programovacího jazyka. Celá práce na téma řadících algoritmů a algoritmů vůbec s programováním souvisí, nicméně se jedná spíše o práci popisnou.

1.2 Základní definice

Algoritmus *je schematický postup pro řešení určitého druhu problémů, který je prováděn pomocí konečného množství přesně definovaných kroků. tato definice pochází z webových stránek [3] Jana Neckáře.*

Graf *je definován jako dvojice množin U(G) a H(G), tedy množina uzlů a množina hran.*

Strom *je souvislý graf bez kružnic. Jeho nesouvislá varianta se nazývá* ***les****.*

Vrchol grafu *nebo také uzel, prvek či bod, je prvek množny U(G), graficky znázorněný jako bod.*

Hrana grafu *je spojení dvou vrcholů, vyjadřující jejich vzájemný vztah.*

Sled *je definován jako posloupnost hran, které na sebe navazují.*
Tah *je sled, ve kterém se neopakuje žádná hrana.*
Cesta *je tah, ve kterém se neopakuje žádný vrchol.*

Kružnicí *se nazývá uzavřená cesta. Kružnice má i orientovanou variantu, která respektuje orientaci hran a nazývá se* ***cyklus****.*

Podgraf H *grafu G je graf, jenž vznikl odebráním některých vrcholů a hran původního grafu G.*

Souvislý graf *je graf, v němž mezi každými dvěma vrcholy existuje cesta. Jinak je graf* ***nesouvislý****.*

Komponenta *grafu G je maximální souvislý podgraf. V tomto podgrafu najdeme cestu mezi libovolnými vrcholy.*

Datová struktura *je specifické zařazení dat.*

Pole *je datová struktura která sdružuje konečný počet prvků stejného datového typu.*

1.3 Algoritmy

Pro správnou funkčnost algoritmů je potřeba, aby algoritmus splnil některé základní vlastnosti. Začněme tedy tím, že algoritmus má dle definice řešit problémy pomocí přesně definovaných kroků. První vlastnost je tedy slyšet již z definice:

Konečnost kroků algoritmu
Otázka tedy zní, proč by měl mít algoritmus omezený počet kroků a taky zda-li je vůbec možné, aby každý algoritmus měl konečný počet kroků.

Algoritmy by měly mít konečný počet kroků, neboť nejsme schopni vytvořit funkční algoritmus o počtu kroků n kdy n jde do nekonečna. Předpokládejme existenci algoritmu, který má nekonečně mnoho kroků. Mohlo by existovat řešení, kdy algoritmus k němu právě v nekonečnu dojde, tudíž by nebyl schopen se ukončit. Jako poznámku bych si dovolil uvést fakt, že jsme schopni vytvořit algoritmus, který pracuje do nekonečna. Otázkou by pak pouze zůstávalo, zda-li je to v našem zájmu. Algoritmy jsou používány zejména v informatice a tudíž je důležité zachovat jejich jednoduchost. Čím méně znaků je použito pro napsání algoritmu, tím méně je celková velikost kódu (tím menší váha, chcete-li). Software poté pracuje s méně prvky a celý proces se urychlí.

Vraťme se ovšem druhé otázce, která je zajímavější. Je možné, aby měl každý algoritmus omezený počet kroků? To samozřejmě záleží na tom, jaký druh problémů má být řešen. Jestliže má předpis například prohledat prvky grafu (prohledávání grafů do hloubky, do šířky), potom je neefektivní zavádět stále více kroků pro každý vrchol grafu. Je třeba tedy nějak zařídit, aby se algoritmus stále vracel k předchozímu kroku (tedy jej zacyklit, takto je možné i vytvořit algoritmus pracující do nekonečna).

Potom je ovšem nutné, aby byla zavedena podmínka ukončení tohoto algoritmu, která cyklus přeruší po nalezení řešení. Dále je však nutné i myslet na alternativu, kdy řešení neexistuje a i přesto je třeba algoritmus nějakou další podmínkou ukončit. Obvykle se

tedy jako další zastavovací podmínka uvádí libovolný počet iterací (počet kroků), nebo se algoritmus sám ukončí po prohledání celého seznamu a nenalezne hledané řešení. V odstavci výše jsem uvedl, že je třeba docílit jednoduchosti předpisu. Ta je důležitá kvůli další vlastnosti algoritmu:

Určitost algoritmu

Proč je důležité docílit určitosti algoritmu?

Zmínil jsem, že se algoritmus užívá především v oboru informatiky a veškeré počítačové systémy pracují na souboru jedniček a nul (tedy pravda, nepravda). Vracíme se tedy k samotným kořenům výrokové logiky a je třeba připomenout i logické operace.

A	B	not A	A and B	A or B	A nand B	A nor B	A xor B	A xnor B
0	0	1	0	0	1	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	1	0	0	0	1

Obrázek 1.1 - tabulka logických operací

Zleva vysvětleno jako: not=negace, and=konjunkce, or=disjunkce, nand=negace konjunkce, nor=negace disjunkce, xor=ostrá disjunkce, xnor=negace ostré disjunkce.

Stejně jako jsou všechny tyto logické operace neměnné a každý program pracuje na jejich bázi, je tedy i nutné, aby náš algoritmus pracoval stejně. Za pomocí těchto spojek musí být náš algoritmus schopen rozlišit výroky na pravdivé, či nepravdivé. Na základě rozhodnutí v jednom kroku algoritmu může potom algoritmus přesměrovat další postup na jiný krok. Například by mohl krok algoritmu znít:

"Jdi na krok 4, jestliže je hodnota prvku (vrcholu grafu) kladná."

Takovou informaci jsme schopni zpracovat a algoritmus je přesně určen, je přesně definován. Jiný, složitější příklad kroku algoritmu může být následující:

"Jdi na krok 4, jestliže je hodnota prvku kladná. Jestliže je záporná jdi na krok 5."

Krok algoritmu má za úkol nás spojit s dalším krokem, zavolat znovu celý algoritmus nebo předpis zcela ukončit (zastavovací podmínka). Přejděme k další vlastnosti algoritmů:

Obecnost algoritmu

Proč má být algoritmus obecný?

Dejme tomu, že máme předpis, který řeší kvadratickou rovnici a nalezne nám její kořeny. Tento předpis je ale užitečný pouze tehdy, když jej můžeme uplatnit i na jinou kvadratickou funkci. Algoritmus by uměl vyřešit $f_x = x^2$, ale už by si neporadil s $f_x = x^2 + 1$, takový předpis by byl nepotřebný a nedal by se užít znovu. Algoritmus musí být nějakým způsobem prospěšný pro větší množství problémů. Navíc je občas potřeba jej užít i pro jiné obory. Příkladem by mohl být algoritmus v oboru financí (třeba porovnávání příjmů a nákladů). Ten se dále může uplatnit u jiných provozů.

Shrňme si tedy základní vlastnosti algortimů:

Algoritmus má konečný počet kroků, řeší všechny úlohy daného typu a každé jeho kroky jsou přesně definovány.

Autor Jan Neckář v práci [3] ještě dále hovoří o **korektnosti** algoritmu. Tato vlastnost říká, že algoritmus skončí správným výsledkem (tedy nezpracovává v jeho průběhu vstupní data chybně). Tento problém se dá eliminovat za užívání správných metod výpočtů v algoritmu, popřípadě přesným měřením (za předpokladu že algoritmus zpracovává nějaká vstupní data). Cílem je, aby předpokládaná chyba ve výpočtech již dále nerostla (popřípadě, aby se zmenšovala). Algoritmy dále rozdělme:

Rekurzivní algoritmy	Iterativní algoritmy
Algoritmus v určitém svém kroku znovu zavolá sám sebe, ještě než je ukončen. V dalším průběhu algoritmu se sám může zavolat několikrát.	Algoritmus ve svém kroku odkáže na jiný krok a vytvoří se jakýsi cyklus. Následně se tyto kroky, nebo několik kroků opakují, dokud algoritmus není ukončen.

Poznámka: Rekurzivní forma se dá převést vždy na formu iterativní. To vyvolává otázku, zda jsou rekurzivní zápisy algoritmů potřeba. Jedná se hlavně o kosmetický problém, někdy je rekurzivní forma lehčí na zápis.

Existuje několik dalších způsobů jak algoritmy rozdělovat. Příkladem je rozdělení na deterministický, popřípadě nedeterministický algoritmus.

Deterministický algoritmus	Nedeterministický algoritmus
Algoritmus má v každém svém kroku přesně definovaný další krok.	Algoritmus má více možností jak pokračovat v postupu.

Algoritmy dále můžeme dle práce [3] dělit na sériové, paralelní a distribuované.

Sériové algoritmy řeší jeden krok po druhém. Příkladem může být algoritmus stroje, který vykonává jednu činnost.
Paralelní algoritmy řeší kroky ve více vláknech, příkladem opět u stroje je algoritmus vykonávající dvě a více činností v jeden časový moment.
Distribuované algoritmy jsou takové, které kroky vykonávají v rámci několika strojů. Dá se říci, že propojují různé algoritmy pro různé stroje.

Nyní je třeba uvést jak algoritmy vlastně pracují. Algoritmus má tedy konečný počet kroků, řeší všechny úlohy daného typu, každé jeho kroky jsou přesně definovány a víme, že se mohou různě členit. Nicméně je důležité si položit a zodpovědět otázku. Je nějaký z al-

goritmů lepší než druhý? Na tento problém se můžeme dívat z několika pohledů. Nejčastěji ale od algoritmu očekáváme, že svůj úkol provádí bezchybně a nejrychleji.

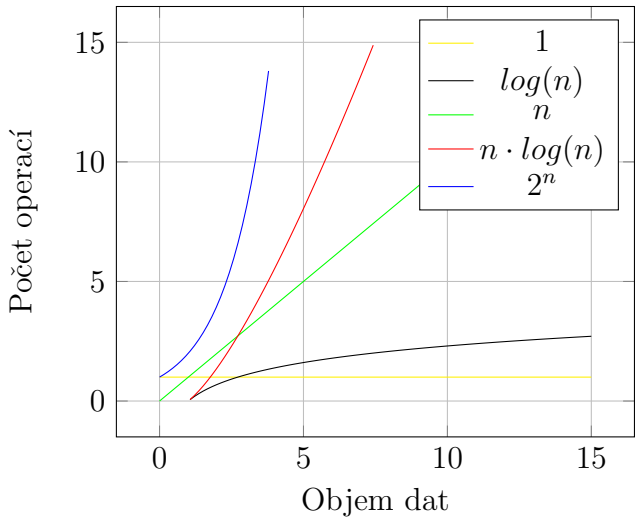
Dále můžeme u algoritmu požadovat jiné vlastnosti. Těmito vlastnostmi míníme jeho úspornost na náklady, jeho složitost nebo cenu. Je důležité se zamyslet nad tím, zda má cenu investovat do algoritmu, který s obrovskou přesností počítá čísla na několik stovek desetinných míst, když je nakonec číslo zaokrouhleno na setiny.

Zavádíme tedy pojem **složitost algoritmu**, který určuje právě rychlost algoritmu vzhledem k počtu vykonaných operací. Složitost algoritmu se klasifikuje pomocí **asymptotické** složitosti do tříd složitosti. U těchto tříd složitosti dále platí, že po určitém objemu dat je algoritmus jedné třídy vždy pomalejší než jiný, bez ohledu na výkonnost počítače. Asymptotická se nazývá proto, že se porovnávají počty vykonaných operací jednotlivých algoritmů v nekonečnu.

$$1 \ll log(n) \ll n \ll n \cdot log(n) \ll n^k \ll k^n \ll n! \ll n^n$$

Nerovnice 1.1 - porovnání chování funkcí (časových složitostí) v nekonečnu.

Řekněme, že **n** zobrazuje objem nebo množství dat. Když provedeme výpočet limity pro $n \to \infty$, zjistíme, že jsou všechny (až na první případ) nekonečno. Nicméně pokud si osu y představíme jako osu počtu operací, tak můžeme jednoznačně určit, že některé algoritmy jsou rychlejší než jiné. Samozřejmě můžeme funkci x^2 násobit konstantou (upravovat výkonnost počítače), nicméně ve srovnání s funkcí $log(n)$ bude vždy pomalejší pro nějaký objem dat. Tedy existuje vždy nějaké x_0 pro které má funkce $log(n)$ dále nižší funkční hodnotu.



Obrázek 1.2 - Porovnání chování funkcí (časových složitostí).

Tento objem dat je ovšem teoretický, a tak můžeme s jistotou říct, že nejefektivnější je konstanta 1. Nicméně už z pouhé úvahy lze vyvodit, že konstantní průběh v podstatě neexistuje. Každý výpočet v algoritmu trvá nějakou časovou jednotku (byť nepatrnou). Algoritmus označený konstantou by musel splňovat, že pro množství dat $n = 1$ je stejně rychlý jako pro $n \to \infty$. Zbytek algoritmů se porovnává podle toho, zda je výhodnější upravit složitost algoritmu, nebo upravit výkon počítače(tedy násobit složitost libovolnou konstantou). Objem dat pro toto zjištění může být znám, nebo alespoň přiblížen. Naopak nejméně efektivní je n^n, protože pro objem dat $n = 2$ bude časová jednotka práce algoritmu rovna $c = 4$, pro $n = 10$ už je časová jednotka rovna 10 biliónům.

Jednotlivé složitosti se dají vyčíst z nerovnice 1.1 a z obrázku 1.2. Následující rozdělení pochází ze stránek Mendelovy univerzity v Brně [1]. Nutno podotknout, že se jedná o zkrácený seznam příkladů složitosti.

Konstantní $O(1)$	Indexování prvku v poli
Logaritmická $O(logN)$	Vyhledávání prvku v seřazeném poli metodou půlení intervalu
Lineární $O(N)$	Vyhledávání prvku v neseřazeném poli sekvenčním hledáním
Lineárně logaritmická $O(N \cdot logN)$	Řazení binárním uspořádaným stromem (průměrná složitost)
Kvadratická $O(N^2)$	Některé řadící metody (Select sort, Insert sort...)
Kubická $O(N^3)$	Násobení matic, závislost řádu matice
Exponenciální $O(2^N)$	Přesné řešení problému obchodního cestujícího hrubou silou

Poznámka: Takové rozdělení časové složitosti algoritmů je poněkud nepřesné. Je nutné zmínit, že tyto vyjmenované třídy spadají do další kategorizace (NL, P, NP, PSPACE, EXPTIME, EXPSPACE). Toto roztřídění je už však silně zaměřené na obor informatiky. Sám autor [3] zdůrazňuje, že tříd složitosti jsou stovky a dokonce i tento zdroj se zabývá pouze několika, které jsem v této poznámce vyjmenoval. Algoritmy se rozdělují do těchto tříd podle času, který potřebují k vykonání sebe sama na různých typech Turingových strojů.

Poznámka: Alan Turing byl britský matematik, známý je díky své práci na dešifrování Enigmy. Položil základy dnešní informatiky díky zavedení pojmu Turingova stroje. To je teoretický model počítače [3], jenž se skládá z nekonečné pásky rozdělené do buněk, řídící pásky, která se může nacházet v nekonečně mnoha stavech a hlavy, která čte a přepisuje jednotlivé záznamy.

Řadící algoritmus [3] je algoritmus, který slouží k setřídění prvků souboru dat nebo seznamu dle velikosti. Existuje mnoho řadících algoritmů. Jak jsem již popisoval, algoritmy pracují s určitou složitostí, a tedy trvají různě dlouhou dobu. Stejně tak je to i u řadících algoritmů. Na různé problémy volíme různé řadící algoritmy na základě našich kritérií. Kritéria mohou být například přesnost algoritmu, kolikrát je třeba ho spustit, jeho náročnost, ale nejčastěji nám jde o jeho rychlost.

Jako příklady těchto algoritmů mohu uvézt zástupce třídy složitosti $O(n^2)$ Bubble sort 2.2, Insertion sort 2.3. U složitosti $O(n \cdot log(n))$ jsou to například Merge sort a Quicksort. U složitosti $O(n)$ mohu uvézt Radix sort a Counting sort(viz kapitola o ostatních algoritmech 3.1.4).

2 Řadící algoritmy

2.1 vlastnosti řadících algoritmů

Jako první a pravděpodobně nejdůležitější vlastnost řadících algoritmů je především již zmíněná **časová složitost**. V rychlosti připomeneme, že nejrychlejší algoritmy pracují teoreticky v konstantní složitosti. Ty nejpomalejší pracují ve složitosti exponenciální. Je nutné k tomu ještě zmínit, že existuje spousty řadících algoritmů. Rozlišujeme je právě podle toho, jakou mají časovou složitost a na jaké případy je můžeme uplatnit. Tento pojem bude ještě objasněn v konkrétních příkladech řadících algoritmů, jako je Bubble sort a Insertion sort.

Stabilita řazení je další vlastností řadících algoritmů. Přítomnost této vlastnosti v algoritmu se ověří snadno, neboť kdyby algoritmus nebyl stabilní, mohl by pracovat do nekonečna. Je-li algoritmus stabilní, potom v něm nedochází k přehazování prvků se stejnou hodnotou. Oproti tomu nestabilní algoritmus ve svém k-tém kroku provede operaci prohození prvků a a b, nicméně protože mají stejnou hodnotu, ve svém kroku $k + 1$ je prohodí nazpět.

Poslední vlastností řadících algoritmů je **přirozenost** algoritmů. Pokud je algoritmus přirozený, potom seřazení již částečně seřazené posloupnosti provede rychleji. Pokud algoritmus přirozený není, potom tento fakt nehraje žádnou roli a algoritmus pracuje stejně rychle.

Z těchto vlastností je možné vyčíst, že algoritmy pracující v časové složitosti blížící se konstantě jsou nejčastěji stabilní a přirozené. Nicméně je třeba zohlednit, že se pohybujeme v teoretické hodnotě n prvků, kdy n se může pohybovat ve velmi vysokých hodnotách přirozených čísel. Tato práce je zaměřena především na již zmíněné algoritmy Bubble sort a Insertion sort. Tyto algoritmy se pohybují v časové složitosti $O(n^2)$ (důkaz viz 2.2.4, 2.3.4), tudíž jsou při vyšším počtu prvků neefektivní. Příklady jsou tedy provedeny na pouze krátkých seznamech prvků.

2.2 Bubble sort

2.2.1 Princip algoritmu

Jedná se o nejběžnějšího zástupce třídy složitosti $O(n^2)$ a možná i nejběžnějšího zástupce řadících algoritmů vůbec. Bubble sort je volně přeložené jako bublinové řazení. Není to náhoda, že se řadící algoritmus takto jmenuje. Jeho princip je přímo odvozen z toho, jak se chovají bubliny ve vodě. Menší bublinky stoupají k povrchu rychleji než velké. Kdybychom měli nekonečně hlubokou vodu a v ní různě se vyskytující bubliny, tak se tyto bubliny po čase srovnají. Nejmenší bubliny se budou vyskytovat nahoře, ty největší budou hluboko pod nimi.

Převeďme Bubble sort zpět do našeho světa. Mějme seznam prvků posloupnosti, který chceme seřadit $k_1, k_2, k_3, \cdots, k_n$. Základní forma bubble sortu porovnává první dva prvky. Krok algoritmu si tedy můžeme snadno domyslet:

"je-li hodnota prvku $k_1 < k_2$, potom tyto prvky prohoď. Pokud je hodnota prvku $k_1 \geq k_2$, potom pořadí prvků zachovej."

Poznámka: Znaménka lze prohodit, znamenalo by to pouze rozdíl v řazení prvků. Tato znaménka způsobují řazení prvků sestupně.

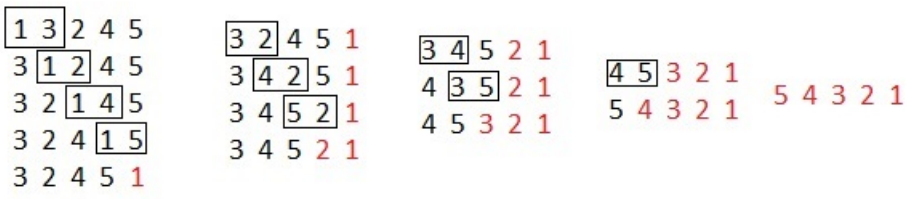
Poté se algoritmus posune a bude porovnávat prvek k_2 s k_3 a tak dále. Algoritmus zde pracuje s něčím, co můžeme nazvat "dočasná pozice"(nebo také "dočasné zařazení", obdobně "trvalá pozice"). Všechny prvky mají v počátku spuštění algoritmu nastavenou dočasnou pozici. Bubble sort poprvé projde všechny prvky posloupnosti a teprve potom se změní pozice nejmenšího prvku vpravo z dočasné na trvalou. Algoritmus již dále ví, že nemusí procházet znovu celou posloupnost, ale pouze její podposloupnost, která je menší o jeden prvek. Algoritmus pracuje do té doby, dokud nemá každý prvek přiřazenou trvalou pozici(Popř. pokud nemáme nastavenou jinou zastavovací podmínku).

Průběh řadícího algoritmu si můžeme představit na sloupcích s různou výškou, na množinách s určitým počtem prvků, nebo jednoduše na číslech s různými hodnotami. Je nutné podotknout, že řadící algoritmus je schopen seřadit čísla všechna kromě imaginárních. Důvodem je prostý fakt, že nejsme schopni porovnat například i a $-i$. Na zakreslení imaginárních čísel je potřeba nový rozměr a v rovině již těžko určíme, které číslo je větší.

Poznámka: Tyto prvky se ovšem stále dají nějakým způsobem setřídit, jen mají například odlišná pravidla řazení a standartní řadící algoritmy musí projít potřebnou úpravou, aby byly schopné seřadit imaginární čísla.

2.2.2 Příklad

Mějme posloupnost náhodně uspořádaných čísel, na které provedeme příklad postupu algoritmu. Prvky s dočasnou pozicí označme černě, prvky s trvalou pozicí označme červeně. Začneme porovnáváním prvků 1 a 3. Prvek 3 je větší, proto si s prvkem 1 vymění pozice v posloupnosti. Obdobně je prvek 2 větší než 1 a také si vymění pozice. To pokračuje do momentu, kdy se prvek 1 "probublá"až na konec posloupnosti. Prvek 1 je tedy nejmenším prvkem v posloupnosti a jeho pozice se změní na trvalou.



Obrázek 2.1

- Příklad Bubble sortu.

Příklad pokračuje obdobně dále, opět se porovnávají prvky, dokud se jeden z nich "neprobublá"na konec posloupnosti. Je třeba zmínit, že se průběh podobá algoritmu prohledávání grafu do hloubky. Tam každý prvek (vrchol) měl přiřazený čas vstupu do vrcholu a čas výstupu z něj. Když se s jednotlivými prvky manipulovalo v zásobníku, řadily se velmi podobně.

2.2.3 Algoritmus

V této práci uvedu dva možné způsoby zápisu algoritmu. První způsob je zapsán alternativně po jednotlivých krocích (viz Úvod - Cíl práce 1.1), druhý způsob se užívá již v informatice. Před uvedením těchto algoritmů je třeba zadefinovat některé pojmy. Především je třeba si uvědomit, že zde budeme operovat s členy posloupnosti, které lze označit jako A_i, pro $i \in \langle 0, n-2 \rangle$. Dále řekněme, že **n** je počet prvků posloupnosti **A**. Proto je zajímavé si všimnout horní hranice pro indexy. Proč je horní hranice právě $n-2$ bude patrné ze zápisu algoritmu.

Dále si zaveďme pro přehlednost pojmy **neseřazená (N)** a **seřazená (S)** množina prvků posloupnosti. Tím pádem řekneme že prvky s vlastností **SORT(A)=TRUE** jsou prvky množiny S, obdobně prvky **SORT(A)=FALSE** budou prvky množiny N. Na závěr zadefinujme jakousi funkci **SWAP**$\{A_i; A_{i+1}\}$, tato funkce prohodí hodnoty členů posloupnosti a jejich indexy zůstanou zachované.

1. Všem prvkům A nastav příznak SORT(A)=FALSE

Jinak řečeno, všechny prvky posloupnosti označ jako neseřazené. Nebo také A:=N. Jdi na 2.

2. SET i:=0, IF N=$\{A_0\}$ {SORTA_0=TRUE, ukonči algoritmus}

Nastav hodnotu aktuálního indexu na i=0. Pokud je podposloupnost neseřazených prvků N jednoprvková množina, tak tento člen označ jako seřazený a ukonči algoritmus. Pokud N není jednoprvková množina, pokračuj na krok 3.

3. IF $A_i < A_{i+1}$**{SWAP**$\{A_i; A_{i+1}\}$**}**

Pokud je prvek A_i menší než A_{i+1}, pak tyto prvky prohoď. Poté pokračuj na další krok. Zde bych se rád odkázal na předchozí odstavec, kdy jsem upozorňoval na horní hranici indexu i. V momentě, kdy začínáme označovat všechny členy posloupnosti od 0, je poslední prvek označen indexem n-1. Když se budeme nacházet v tomto prvku, nemůžeme jej porovnat s žádným dalším prvkem, protože je A_i posledním prvkem posloupnosti.

4. IF $i = n_N - 2$**{SORT**(A_{i+1})**=TRUE, jdi na 2.; SET i:=i+1, jdi na 3.** }

Označení n značí počet prvků. Jeho index N potom značí počet prvků neseřazené části posloupnosti. Pokud se $i = n_N - 2$*, tak poslednímu neseřazenému prvku nastav vlastnost - seřazený. Hned potom jdi na krok č.2. Pokud se* $i \neq n_N - 2$*, tak nastav index i na hodnotu i+1, následně jdi na krok č.3.*

Poznámka: Tento zápis algoritmu jsem napsal podle sebe. Chtěl jsem poukázat na vše důležité srozumitelně, i pro čtenáře bez znalostí vybraných kapitol informatiky, či programování.

Konkrétně výše uvedený zápis algoritmu je velmi jednoduchý a zohledňuje i fakt, že již posloupnost byla částečně srovnána. Následující zápis algoritmu je psán v programovacím jazyce a jeho průběh je částečně odlišný. Zatímco obecný zápis výše kontroloval prvky na základě jejich indexu, tak u tohoto zápisu se vyskytují dva cykly. První cyklus je jeden průběh algoritmu celou posloupností. Druhý cyklus je porovnávání jednotlivých prvků a přehazování hodnot. Zápis algoritmu pochází od Vojtěcha Hordějčuka [2]. Implementace tohoto algoritmu je v softwaru Java. Můžeme si povšimnout výskyt svou indexů **i** a **j**. Zde si postup algoritmus lze představit jako převodovku. Počáteční prvek označíme indexem **i** a porovnáváme jej se všemi prvky posloupnosti, které označujeme jako **j**. Jakmile porovnáme prvek **i** se všemi prvky **j**, zařadíme jej na správnou pozici. Potom se otáčí velký převod a posouváme se na prvek s indexem **i+1**. Tento prvek opětovně porovnáváme se všemi prvky **j**.

```java
/**
* Implementace algoritmu bubble sort.
* @param input vstupní pole
* @author Vojtěch Hordějčuk
*/
public static <T extends Comparable<? super T>> void bubbleSort(final T[] input)
{
 // průchod skončí na předposledním prvku (index 'i - 2')
 // (prvek se vždy porovnává se svým následníkem)

 for (int i = 0; i < input.length - 1; i++)
 {
   // už jsme seřadili 'i' prvků
   // (na konci pole je tedy nemusíme kontrolovat)

   for (int j = 0; j < input.length - 1 - i; j++)
   {
     // prohoď sousední prvky, pokud jsou ve špatném pořadí

     if (input[j].compareTo(input[j + 1]) == 1)
     {
       final T temp = input[j];
       input[j] = input[j + 1];
       input[j + 1] = temp;
     }
   }
 }
}
```

Obrázek 2.2 - Bubble sort, Java

Můžeme si povšimnout, že algoritmus má taktéž nastavenou horní hranici jako n-2, ze stejného důvodu. Dále je zde taktéž zohledněn fakt, že již srovnané prvky posloupnosti neprocházíme znovu a algoritmus je tak rychlejší. Teprve až vnitřní funkce algoritmu má napsáno ono prohazování prvků, jak jej již známe.

Na závěr můžeme zmínit, že algoritmus tohoto typu je stabilní, je přirozený, nicméně je velmi pomalý. Jeho časovou náročnost si dokážeme v následující podkapitole.

2.2.4 Časová složitost

Časová složitost algoritmu Bubble sort je $O(n^2)$. Dá se zjistit z mnoha zdrojů, nicméně všude je jeho interpretace stejná. Tato konkrétní interpretace je inspirována zejména ze

stránek pana Neckáře [3]. Důkaz se provádí přes aritmetickou posloupnost.

$$n \cdot \frac{A_1 + A_n}{2}$$

Nicméně toto je pouze obecný vzorec, je třeba ho trochu poupravit. Vnitřní cyklus (tj. cyklus prohazování členů posloupnosti) se provádí nejprve n-1 krát. Po seřazení a následném odstranění posledního členu se provede n-2 krát. Celkem se tedy cyklus provede $(n-1)+(n-2)+\cdots 2+1$ krát. Nyní třeba jen zbývá dosadit do vzorce z aritmetické posloupnosti patřičné hodnoty:

$$(n-1) \cdot \frac{(n-1)+1}{2}$$

$$\frac{1}{2} \cdot (n^2 + n)$$

Po několika aritmetických i kosmetických úpravách vidíme, že nejvyšší mocnina (vedoucí člen) je n^2. Náš závěr tedy zní, že Bubble sort patří do řadících algoritmů s časovou složitostí $O(n^2)$.

Je nutno ještě dodat, že kupříkladu časová složitost Insertion sortu je také $O(n^2)$ a většina dalších algoritmů v této práci má také stejnou časovou složitost. Neznamená to však, že pracují stejně rychle. Nejrychlejší v této časové složitosti je Shell sort, který je modifikací Insertion sortu. Oproti tomu Bubble sort se pokládá za nejpomalejší. Podívejme se však na vylepšení Bubble sortu, které je znatelně rychlejší.

2.2.5 Případná vylepšení algoritmu

Vzpomeňme si na průběh Bubble sortu a na to, jak algoritmus pracoval. Bubble sort začal porovnávání v počátečním členu posloupnosti a poté postupoval až na jeho konec, kde nastavil pozici posledního prvku jako trvalou. Následně se "přesunul"zpět do počátečního prvku posloupnosti a celý proces opakoval. Je patrné, že v algoritmu je jakési "hluché"místo a doslova se nabízí jeho vylepšení v této mezeře. Bubble sort postupoval pouze jedním směrem, a to z počátku nakonec, jeho vylepšením je tzv. Shaker sort. Shaker sort stejně jako Bubble sort pokračuje do posledního prvku posloupnosti, ten nastaví jeho pozici na trvalou. Poté se však nevrací do počátku posloupnosti, ale jaksi se převrátí a

postupuje pozpátku, prvek po prvku. Výsledkem toho je, že Shaker sort cestou na konec uzamkne nejmenší prvek a cestou na začátek uzamkne prvek největší (popř. opačně, pokud máme opačná znaménka v algoritmu). Shaker sort je stále stabilním a přirozeným řadícím algoritmem.

V úvodu této kapitoly jsem zmínil princip Bubble sortu 2.2.1, ten v podstatě nechává probublávat nejmenší bublinky výš, zatímco ty větší se drží při zemi. Shaker sort pracuje na začátku zcela stejně do té doby, než prvek na konci posloupnosti označí za seřazený. Představme si, že algoritmus je jakási ruka. Tahle ruka vyzdvihne nejmenší bublinku nejvýše a poté se vrací přes všechny bubliny zpět a cestou najde tu největší a přesune ji nejníže. Hovoříme tedy o řadícím algoritmu, který se dá chápat jako oboustranný Bubble sort. Příklad zápisu tohoto algoritmu pochází opět ze stránek o algoritmech [3]. Shaker sort je zapsán v Javě.

Následující zápis algoritmu pracuje na principu dvou různých cyklů, které již do sebe nejsou vnořené jako v předchozím případě Bubble sortu (viz obrázek 2.2). V prvním cyklu Shaker sort postupuje jako Bubble sort staticky do poloviny posloupnosti. V této části algoritmus porovnává prvky první poloviny posloupnosti a menší posouvá na její konec. Dále se zde nachází kontrola tohoto cyklu a po jeho dokončení pokračuje algoritmus na cyklus následující. Druhý cyklus postupuje od konce posloupnosti opět pevně do poloviny posloupnosti jako "zpáteční"Bubble sort. Tento cyklus má naopak za úkol postupně posouvat největší prvky směrem na počátek posloupnosti. Zde je nastavena podmínka na případné přerušení cyklu a navrácení k původnímu.

```java
/**
 * Shaker sort (obousmerny Bubblesort)
 * radi od nejvyssiho
 * @param array pole k serazeni
 */
public static void shakerSort(int[] array) {
    for (int i = 0; i < array.length/2; i++) {
        boolean swapped = false;
        for (int j = i; j < array.length - i - 1; j++) {
            if (array[j] < array[j+1]) {
                int tmp = array[j];
                array[j] = array[j+1];
                array[j+1] = tmp;
                swapped = true;
            }
        }
        for (int j = array.length - 2 - i; j > i; j--) {
            if (array[j] > array[j-1]) {
                int tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
                swapped = true;
            }
        }
        if(!swapped) break;
    }
}
```

Obrázek 2.3 - Shaker sort, Java

V případě, že bychom chtěli algoritmus zapsat alternativně (tedy krok po kroku), jako tomu bylo u Bubble sortu 2.2.3, museli bychom do algoritmu přidat další dva kroky. Zde by nastal pouze problém v indexech, v Bubble sortu byly přesouvány prvky z množiny N na základě vzdálenosti od A_0. Nyní odstraňujeme prvky na základě vzdálenosti od počátečního prvku, který se neustále mění. Zároveň se neustále posouvá i poslední prvek posloupnosti, neboť oba tyto prvky neustále vyřazujeme(přesouváme do množiny S).

V prvním přidaném kroku by byla obdoba kroku č. 3, Pokud by $A_i > A_{i-1}$, potom bych tyto dva prvky prohodil. Další přidaný krok by byl obdobou kroku č. 4. Za předpokladu, že aktuální index je druhým nejmenším indexem z prvků množiny N, poté bychom označili A_{i-1} jako prvek srovnaný. Následně bychom pokračovali tak, jako by to dělal samotný Bubble sort. Nejvíce problematický by byl zápis podmínek 4. a 6. kroku. Dalším problémem

by pak byl ještě krok, který by obsahoval podmínku na ukončení celého algoritmu. Zápis Shaker sortu v krocích je proto o dost delší a již postrádá pozitivní vlastnost přehlednosti a jednoduchosti. Je možné, že by byl algoritmus napsaný sice srozumitelně, nicméně v 7 nebo 8 krocích.

2.3 Insert sort

2.3.1 Princip algoritmu

Insert sort se dá do češtiny přeložit jako řazení vložením. Insert je pravděpodobně nejvíce známý lidem jako klávesa, jejíž funkcí je podobně jako u caps locku, num locku nebo scroll locku přepínání nějakého režimu. Konkrétně insert prohazuje vkládání a přepisování textu. Stává se, že někdo insert zapne omylem a na první pohled se zdá vše v pořádku. Pokud se však dotyčný člověk vrátí a chce nějakou část textu opravit nebo přepsat, setká se s problémem. Najednou veškerý nový text přepisuje ten starý, místo aby se vložil.

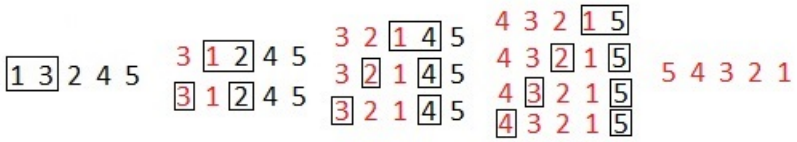
Insert sort je postaven velmi podobně, pracuje totiž na oné hranici "kurzoru". Bubble sort přesouval prvek až na konec dané posloupnosti prvků a potom ten poslední nastavil jako seřazený. Tento algoritmus porovná první prvky, rozhodne o jejich pozici a hned jejich pozici uzamkne. U dalších prvků algoritmus rozhoduje, zda ostatní prvky patří před, mezi nebo za prvky již seřazené.

Dá se tedy říci, že první prvky posloupnosti jsou seřazeny poměrně rychle, nicméně čím více prvků je seřazených, tím déle algoritmus rozhoduje o dalším prvku, kam jej zařadit, protože jej musí porovnat se všemi ostatními. Zde je nutné připomenout, že posloupnost prvků rozdělujeme na seřazenou a neseřazenou. U Bubble sortu hrály tyto pojmy menší roli, nežli zde. Je to patrné v další podkapitole na příkladu 2.3.2.

2.3.2 Příklad

Obdobně jako u Bubble sortu, proveďme příklad průběhu Insert sortu na téže posloupnosti. Stejně tak prvky s dočasnou pozicí označme černě, prvky s trvalou pozicí označme červeně. Začneme porovnáváním prvků 1 a 3. Prvek 3 je větší, proto si s prvkem 1 vymění pozice v posloupnosti a nadále jsou označeny jako seřazené. Dále je vzat první prvek posloupnosti, který není seřazen a porovnáváme jej s prvky ze seřazené množiny. Jakmile je prvek porovnáván se seřazeným prvkem, který je větší, tak se člen označí jako seřazený. Jeho pozice je nastavena trvale právě za prvek, který byl větší (prvkům za ním se posouvá index o +1).

V našem případě se prvek 2 porovnává s 1, tam nedošlo k žádné změně. Dál se tedy prvek 2 porovná s prvkem 3, protože má větší hodnotu, tak je dvojka zařazena před něj. Zbytek posloupnosti se algoritmem srovnává, dokud není posloupnost seřazena.



Obrázek 2.4 - Příklad Insertion sortu

Připomeňme si příklad u Bubble sortu 2.2.2, čím více prvků bylo srovnaných, tím rychleji algoritmus seřadil zbytek. U Insert sortu je tomu naopak, čím více prvků je seřazeno, tím déle jsou řazeny ostatní prvky posloupnosti.

2.3.3 Algoritmus

V této podkapitole budou opět uvedeny dva různé zápisy algoritmu. Druhý zápis pochází ze stránek voho, jeho autorem je Vojtěch Hordějčuk [2]. Připomeňme opět, že prvky posloupnosti lze označit jako A_i a stejně tak v tomto případě bude první člen posloupnosti označen jako A_0. Nicméně množina indexů $i \in \{-1, 0, 1, \ldots, n-1\}$ je již odlišná. Dále definujme **n** jako počet prvků posloupnosti **A**. Dále si připomeňme pojmy **neseřazená**

(N) a **seřazená (S)** množina prvků posloupnosti, zde hrají větší roli. Na závěr řekněme, že prvky s vlastností **SORT(A)=TRUE** jsou srovnané a prvky **SORT(A)=FALSE** jsou neseřazené. Obdobně ještě připomeňme námi zadefinovanou funkci **SWAP**$\{A_i; A_{i+1}\}$, která prohodí hodnoty dvou členů posloupnosti bez prohození indexů.

1. Všem prvkům A **nastav příznak SORT**(A)**=FALSE**

Jinak řečeno, všechny prvky posloupnosti označ jako neseřazené. Nebo také A:=N. Jdi na 2.

2. SET i:=0, IF $N = \{A_0\}\{$**SORT**$\{A_0\}$ **=TRUE, ukonči algoritmus**$\}$

Nastav hodnotu aktuálního indexu na i=0. Pokud je podposloupnost neseřazených prvků N jednoprvková množina, tak tento člen označ jako seřazený a ukonči algoritmus. Pokud N není jednoprvková množina, pokračuj na krok 3.

3. IF $A_i < A_{i+1}\{$**SWAP**$\{A_i; A_{i+1}\}\}$**, SORT**$\{A_i; A_{i+1}\}$ **=TRUE**

Pokud je první člen posloupnosti menší než druhý, prohoď jejich pozice. Následně je oba označ jako srovnané. Jdi na další krok.

4. IF N= $\emptyset\{$**ukonči algoritmus**$; i := i + 1\}$

Pokud byla posloupnost složena pouze ze dvou prvků A_0 *a* A_1*, potom ukonči algoritmus. Jinak pokračuj na krok 5. Do teď byly všechny kroky pouze jakési obranné opatření proti posloupnostem s jedním, nebo dvěma členy.*

5. IF $A_{i+1} < A_i\{$**SORT**$\{A_{i+1}\}$**=TRUE, jdi na 7.; SWAP**$\{A_{i+1}; A_i\}$**, jdi na 6.**$\}$

Pokud je první neseřazený prvek posloupnosti menší než poslední seřazený, potom jej označ jako seřazený a jdi na 7. krok. Pokud tomu tak není, tak tyto dva členy prohoď a jdi na 6. krok.

6. $i := i - 1$**, IF** $i < 0\{$**SORT**A_{i+1} **=TRUE, jdi na 7.; jdi na 5.**$\}$

Nastav aktuální index na $i := i - 1$. *Dále pokud je index i menší než nula, označ člen posloupnosti* A_{i+1} *jako označený a jdi na 7. krok. Pokud tomu tak není, jdi na 5. krok.*

7. IF N= $\emptyset\{$**ukonči algoritmus**$; i := min\{i_N\} - 1$**, jdi na 5.** $\}$

Pokud je množina neseřazených členů posloupnosti prázdná, potom ukonči algoritmus. Pokud prázdná není, nastav aktuální index i jako nejmenší index z množiny neseřazených prvků a odečti od něj jedničku. Následně jdi na 5. krok.

Zápis algoritmu v Javě je od obecného zápisu lehce odlišný. Tyto odlišnosti jsou patrné zejména v momentě samotného porovnávání prvků. V obecném zápisu prohazujeme prvky hned při jejich porovnávání jako u Bubble sortu. Mimo tuto odlišnost jsou ještě prvky porovnávány na základě indexů. V zápisu od pana Hordějčuka je vnořen opět cyklus indexů **i** a **j**. Aktuálně řazený prvek se označí indexem **i** (jako velký převod) a na místo prohazování pozice s jinými prvky se zapamatuje fixně jeho počáteční pozice. Prvky označené jako **j** (malý převod) jsou s prvkem **i** porovnávány a jakmile je pro tento prvek nalezena jeho pozice, algoritmus jej na tuto pozici umístí. Prvky napravo od prvku **i** se posunou o jednu pozici doprava a zároveň algoritmus hlídá aby všechny prvky zůstali ve vytyčené velikosti pole.

```java
/**
* Implementace algoritmu insertion sort.
* @param input vstupní pole
* @author Vojtěch Hordějčuk
*/
public static <T extends Comparable<? super T>> void insertionSort(final T[] input)
{
 for (int i = 1; i < input.length; i++)
 {
   // začni na aktuálním indexu

   int j = i;

   // zapamatuj si číslo, které se bude posouvat na správnou pozici
   // (na toto místo v poli se totiž mohou dostat větší čísla)

   final T moved = input[i];

   // posouvej ostatní čísla vpravo, dokud:
   // 1) nejsi na konci pole
   // 2) a zároveň jsou tato čísla větší než posouvané číslo

   while ((j > 0) && (input[j - 1].compareTo(moved) == 1))
   {
     input[j] = input[j - 1];
     j--;
   }
```

Obrázek 2.5 - Insertion sort, Java

Nyní je nutno se zamyslet nad samotným průběhem algoritmu. V kapitole 2.3.1 je uvedeno, že právě srovnávaný člen se po nalezení jeho pozice pouze "vrazí" mezi dva jiné prvky jako klín. Nicméně po důkladnějším prozkoumání zápisu algoritmu v krocích je patrné, že srovnávaný člen posloupnosti se nepřenese, nýbrž se prohazuje s prvky dokud nenalezne svojí polohu. Jak tento problém vyřešit a algoritmus přepsat? Jedná se sice o maličkost, která by v rychlosti průběhu algoritmu hrála jen malou roli (při tak vysoké časové složitosti řadícího algoritmu), ale někdo by to mohl žádat. Dalo by se to vyřešit připsáním dalšího kroku do algoritmu, který by zavedl index k, který by zůstával pevný na rozdíl od indexu i. Následně by se tento prvek s konstantním indexem posunul na pozici v posloupnosti, kam by podle seřazení patřil. Všechny prvky před ním by měly zachovaný index a všechny prvky po něm by měly index větší o jedničku.

Takové vyřešení zápisu algoritmu by samozřejmě pomohlo a algoritmus by již pracoval správně. Podívejme se však na délku samotného algoritmu. Bez této úpravy má algoritmus už 7 kroků, přičemž úvodní kroky by se daly po pečlivé práci přepsat do jediného kroku (viz kapitola o diagramech 3.3). Jedná se pouze o kroky, které zvládají i posloupnosti s jedním, nebo pouze dvěma členy. Následně má algoritmus 3 klíčové cykly. Hlavní otázkou tedy zůstává, zda se dají tyto kroky omezit a vytvořit stejně funkční algoritmus.

Tato malá a jednoduchá myšlenka však otvírá dveře do veliké místnosti. Na celé diagramy řadících i jiných algoritmů se dá pohlížet jako na orientované grafy, přičemž kroky jsou vrcholy tohoto grafu. Naopak překreslením algoritmu do orientovaného stromu získáme jasný nástin jeho funkčnosti. Je třeba zařídit, aby z každého vrcholu orientovaného grafu existoval tah do výstupního vrcholu grafu.

Z tohoto důvodu bych mohl výše uvedený zápis Insert sortu osekat o druhý krok a třetí krok. Výsledkem bych měl skutečně kratší zápis algoritmu. Otázkou zůstává, zda by výsledný efekt byl žádoucí, neboť by si tento zápis již neporadil s posloupnostmi s jedním, nebo dvěma členy. Nebo ještě hůře, algoritmus by se ani neukončil a zobrazil by chybové hlášení.

2.3.4 Časová složitost

Časová složitost Insert sortu je stejně jako u Bubble sortu $O(n^2)$. Důkaz se provádí stejně tak přes aritmetickou posloupnost.

$$n \cdot \frac{A_1 + A_n}{2}$$

Je nutné si ovšem uvědomit, které cykly zde pracují a co vlastně algoritmus dělá. Insert sort vybere člen posloupnosti a ten se následně porovnává se všemi členy před ním, přičemž před nějaký z nich se zařadí a posouváme se k řazení dalšího členu posloupnosti.

Zkusme si představit nejhorší možný případ řazení. Takový případ by nastal, kdyby se

každý člen posloupnosti musel posunout až na začátek a tedy se porovnat se všemi srovnanými členy posloupnosti. Poslední prvek posloupnosti by tedy musel projít zbylých $n-1$ členů a tedy jeho čas je $n-1$. Jeho celková časová složitost je tedy součet dílčích operačních časů všech prvků. Dostáváme se zpět ke vzorci na součet členů aritmetické posloupnosti $(n-1) \cdot \frac{1+n-1}{2}$. Po vypočtení dostáváme výsledek $\frac{1}{2} \cdot (n^2 - n)$ to je tedy výsledná časová složitost $\mathrm{O}(n^2)$.

Podívejme se však na zajímavější případ pohledu na časovou složitost. Toto je i důvodem, proč je Insert sort ve většině případů rychlejší, než Bubble sort. Jaký je nejlepší možný průběh Insert sortu? Aktuálně srovnávaný člen se porovnává se všemi členy seřazené podposloupnosti a pokud je některý z nich větší, zařadí se napravo od něj. Co když aktuálně řazený prvek bude vždy menší než člen nalevo od něj(jinak řečeno, co když je posloupnost částečně srovnána)? Potom se všechny prvky porovnávají vždy pouze s jedním členem srovnané podposloupnosti a čas tohoto srovnávání je tedy 1 pro každý prvek. Dostáváme tedy součet $1+1+1+\cdots+1$, přičemž se zde nachází $n-1$ krát jedniček. Výsledná časová složitost Insert sortu v jeho nejlepším případě je tedy lineární $\mathrm{O}(n)$.

Část této časové složitosti pochází z kanálu Agilowen [4], který se věnuje především řadícím algoritmům.

2.3.5 Případná vylepšení algoritmu

Vylepšením Insertion sortu se sice stále zdržíme v časové složitosti $\mathrm{O}(n^2)$, ale Shell sort je z nich zcela nejrychlejší. Také je známý pod jménem Shellovo řazení nebo řazení se snižujícím se přírůstkem [3].

Insert sort pracoval s množinou seřazených prvků a s množinou neseřazených prvků. Rozdělil tedy posloupnost odpovídajícím "klínem"a na jedné straně (na levé) se nacházely všechny seřazené prvky. Následně docházelo k seřazení prvku neseřazeného, avšak přímo sousedícího s množinou seřazených.

Shell sort pracuje velmi obdobně. Jeho jediný rozdíl spočívá ve vytvoření většího množství "rozdělovacích klínů". Představme si tedy posloupnost čísel, u nichž je první prvek označený jako seřazený. Dále je třeba určit počet mezer a také jejich délku. Od prvního členu posloupnosti tedy volíme například konstantní délku mezery jako 5 neseřazených členů. Za mezerou se bude nacházet opět další seřazený člen a tak dále. Pokud se bude nacházet v posloupnosti **k** různých seřazených prvků, tak je to jako bychom rozdělili posloupnost do **k** různých neseřazených podposloupností a na nich aplikovali Insert sort.

Tímto jedním krokem se tedy neustále zmenšují mezery. V momentě, kdy Shell sort zmenší všechny mezery na délku jednoho prvku, postupuje dále jako běžný Insertion sort. Výhoda spočívá v tom, že neseřazený prvek sousedící s množinou seřazených prvků posloupnosti již nemusí procházet v případě potřeby všechny možné prvky. Otázkou tedy zůstává, s kolika prvky se tedy sám porovná, než se označí jako seřazený. To záleží na volbě mezery.

Volba mezery byla problémem, se kterým si lámalo hlavu více lidí. Prvním byl samozřejmě objevitel Shell sortu - Donald Shell. Neckář na svých stránkách dále uvádí například přístup Hibbarda nebo Sedgewicka. Dále zde uvádí například Fibonacciho posloupnost umocněnou na dvojnásobek zlatého řezu.

Poznámka: Vojtěch Hordějčuk [2] uvádí, že Zlatý řez je ideální poměr mezi dvěma úsečkami. Jinak se nazývá i jako zlatý poměr, popřípadě zlaté číslo. Jedná se o číslo iracionální, jeho zaokrouhlená hodnota činí 1.618. *Fibonacciho posloupnost je posloupnost, kdy každý další člen je součtem dvou předchozích (tj.* $1, 1, 2, 3, 5, 8, \ldots$*).*

Údajně nejlepší výsledky [3] na základě pokusů podává posloupnost čísel 1, 4, 10, 23, 57, 132, 301, 701, 1750, dále mezera $\cdot 2,2$. Autorem je Marcin Ciura. Jinými slovy se mezera volí jako polovina prvků posloupnosti (klín se "vrazí"doprostřed posloupnosti) a následně po každém průběhu algoritmu skrz posloupnost vydělím velikost mezery číslem

2,2. Výslednou hodnotu mezery je následně třeba zaokrouhlit a posloupnost rozdělit. Po dalším průchodu algoritmu posloupností opět dělíme mezeru číslem 2,2.

Přenesení takového algoritmu do řeči kroků by bylo velmi složité. Jeho implemetace v Javě pochází ze serveru geeksforgeeks a jejím autorem je Rajat Mishra [5]. Složitost tohoto algoritmu je patrná z délky celého zdrojového kódu. Můžeme si povšimnout způsobu omezování mezery v obrázku 2.8, zde autor po každém průchodu algoritmu posloupností omezuje šířku mezery na polovinu. Následující řádky popisují cyklus indexů **i** a **j**. Index **i** se nastaví jako aktuálně řazený prvek v dané mezeře a porovnává se s ostatními prvky značenými indexem **j**. Průběh tohoto cyklu zpracovává všechny prvky ve všech mezerách posloupnosti stejně, jako předtím Insertion sort v prvky v celé posloupnosti.

```
// Java implementation of ShellSort
class ShellSort
{
    /* An utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
```

Obrázek 2.6 - 1.část vnějšího cyklu Shell sortu, Java

Zde se nachází vnitřní cyklus *(obrázek 2.8)*.

```
    // Driver method
      public static void main(String args[])
      {
          int arr[] = {12, 34, 54, 2, 3};
          System.out.println("Array before sorting");
          printArray(arr);

          ShellSort ob = new ShellSort();
          ob.sort(arr);

          System.out.println("Array after sorting");
          printArray(arr);
      }
  }
  /*This code is contributed by Rajat Mishra */
```

Obrázek 2.7 - 2.část vnějšího cyklu Shell sortu, Java

Vnitřní cyklus celého zdrojového kódu v sobě ukrývá podstatnou část informací. Nachází se zde předpis pro tvorbu mezery, pro její následné zmenšování a nakonec se zde nachází i samotný algoritmus Insertion sortu.

```
/* function to sort arr using shellSort */
   int sort(int arr[])
   {
       int n = arr.length;

       // Start with a big gap, then reduce the gap
       for (int gap = n/2; gap > 0; gap /= 2)
       {
           // Do a gapped insertion sort for this gap size.
           // The first gap elements a[0..gap-1] are already
           // in gapped order keep adding one more element
           // until the entire array is gap sorted
           for (int i = gap; i < n; i += 1)
           {
               // add a[i] to the elements that have been gap
               // sorted save a[i] in temp and make a hole at
               // position i
               int temp = arr[i];

               // shift earlier gap-sorted elements up until
               // the correct location for a[i] is found
               int j;
               for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                   arr[j] = arr[j - gap];

               // put temp (the original a[i]) in its correct
               // location
               arr[j] = temp;
           }
       }
       return 0;
   }
```

obrázek 2.8 - vnitřní cyklus Shell sortu, Java

3 Algoritmy, algoritmy a zase ...

3.1 Další algoritmy

3.1.1 Heap sort

Heap sort je také známý v českém názvu jako řazení haldou. Tento algoritmus používá především vlastnost speciálního případu grafů, tedy binárních stromů. Heap sort nejprve

roztřídí prvky do takového binárního stromu. Před dalším popisem principu je nutné zadefinovat nové pojmy, které se vyskytují pouze v této podkapitole:

Binární stromy *jsou dynamické datové struktury, ve kterých jsou prvky hierarchicky uspořádány pomocí ukazatelů tak, že každý prvek ukazuje nejvýše na dva následující prvky a je určen jeden počáteční prvek, ze kterého všechny ukazatele vychází (tzv. kořen)[2]. Těmto stromům se také říká kořenové.*

Binární strom označuje vždy hranu k prvnímu potomkovi jako 1, druhou jako 0. V podstatě to odpovídá rozhodnutí o něčem, kdy existuje pouze varianta ano nebo ne. Není to náhoda, takový binární strom se používá i na Huffmanovo kódování, z kořene vede ke každému vrcholu právě jedna, originální cesta. Ke koncovému vrcholu se tedy mohu dostat například pouze po hranách 0, 1, 1, atd., postupujeme tedy hlouběji do binárního stromu. Tím se dostáváme k další definici tzv. hloubky binárního stromu.

Patrem binárního stromu *označujeme jednotlivé hladiny binárního stromu, rozlišujeme jednotlivé "generace".*

Jinak řečeno, kořen patří jako jediný do nultého patra (hladiny) bin. stromu. Jeho potomci patří do prvního patra, obdobně potomci potomků do druhého.

Úplný binární strom *má všechna patra (kromě posledního) zaplněna vrcholy. Dále rozlišujeme binární stromy úplné* ***zleva*** *a* ***zprava****. Tím označujeme, že poslední patro úplného binárního stromu má prvky seřazené prioritně v levé, nebo v pravé straně bin. stromu.*

Nyní můžeme konečně přikročit k principu Heap sortu. Algoritmus nejprve rozdělí všechny prvky posloupnosti do vrcholů binárního stromu. Kořenem je první prvek posloupnosti, jeho potomci jsou druhý a třetí prvek atd., binární strom musí být zleva úplný. Dále začne Heap sort porovnávat rodiče a potomky z nejspodnějších hladin bin. stromu až ke kořenu.

Vždy, když je rodič větší než alespoň jeden z potomků, tak si rodič prohodí pozici s menším z potomků. Potom algoritmus pokračuje blíže ke kořenu do nižší hladiny. Jakmile skončí s řazením, nejmenší prvek posloupnosti se ukáže být jako kořen tohoto binárního stromu.

Kořen prohodíme s posledním vrcholem binárního stromu tak, aby byla zachována proporce zleva úplného binárního stromu. Tento vrchol potom odstraníme ze stromu a zapíšeme jej jako srovnaný prvek posloupnosti. Heap sort poté znovu srovná rodiče a potomky celého stromu a celý proces opakuje.

Protože se zde využívá vlastnosti binárních stromů, má tento algoritmus velmi příznivé výsledky ohledně vyššího počtu dat. Navíc je Heap sort stabilní a přirozený. Jeho časová složitost je $O(n \cdot log(n))$, tedy mluvíme o algoritmu mnohem výkonnějším a rychlejším než dosud zmíněné algoritmy.

3.1.2 Quick sort

Neboli v překladu - rychlé seřazení. Jedná se o poměrně rychlý řadící algoritmus se stejnou časovou složitostí jako Bubble sort, nebo Insertion sort. Jeho očekávaná časová složitost je však $O(n \cdot log(n))$. Aby se dostalo vysvětlení této přívětivější časové složitosti, je nejprve nutné přikročit k tomu, jak Quick sort funguje.

Quick sort je lehce podobný průběhu Insertion sortu. Z celé posloupnosti Quick sort vybere prvek, který se označí jako pivot. Dále rozdělí posloupnost tak, aby na jedné straně byly prvky větší než pivot a na druhé menší. U obou těchto podposloupností se dále vybere nový pivot (ten starý se označil jako seřazený) a prvky zbývající se podle něj znovu seřadí. To nahrává možnému ideálnímu průběhu algoritmu. Pokud se při volbách pivota vždy trefíme tak, že roztrhneme posloupnost přesně v půli, tak dosáhneme oné lepší časové složitosti. Mluvíme zde ovšem o ideálním výsledku řazení algoritmu, v naprosto drtivé většině případů je řazení provedeno v časové složitosti $O(n^2)$.

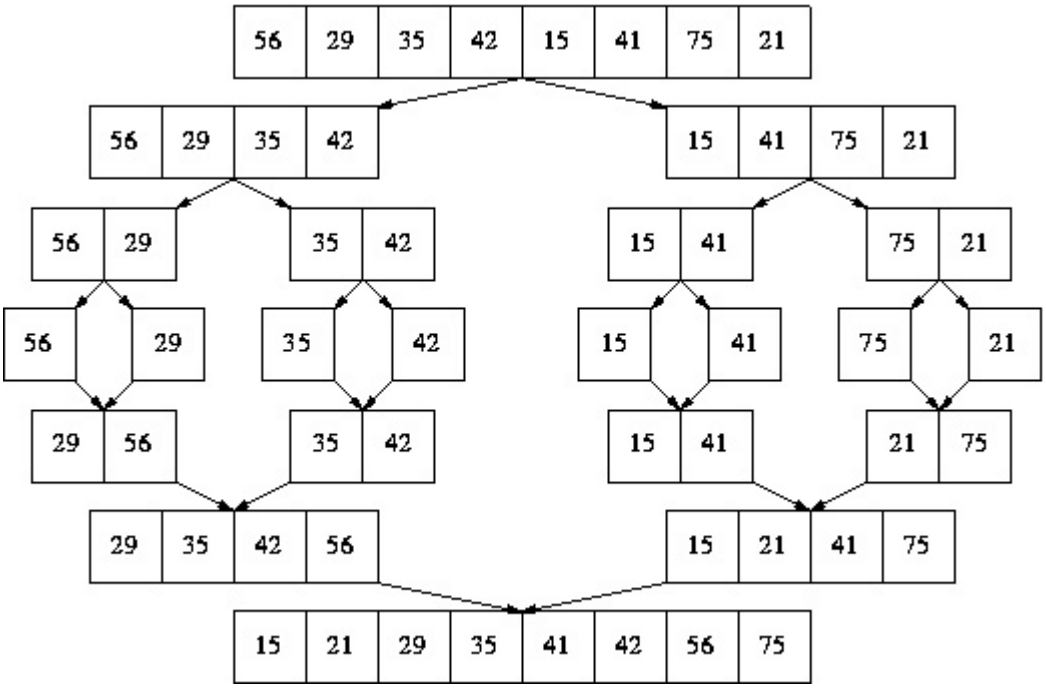
Algoritmus byl přirovnán k Insert sortu. To zejména proto, že oba tyto algoritmy pra-

cují na bázi výběru jednoho prvku posloupnosti a ostatní se mu musí přizpůsobit. Jinými slovy je tedy i podobný Shell sortu. Bubble sort naopak vždy vybral prvek a ten zarovnal podle ostatních. Otázkou u Quick sortu ovšem zůstává, jak tedy volit pivota. Ze zamyšlení by nás mohlo napadnout prosté, avšak poměrně časté řešení volby. Pivota vždy volíme jako medián posloupnosti, pro kterou ho volíme. Dále bychom stejně tak mohli volit prvek nejblíže hodnotě průměru maximální a minimální hodnoty poslopnosti. Nebo bychom mohli vybrat prvek zcela náhodně.

3.1.3 Merge sort

Merge sort je poměrně jednoduchý. Jak to tak ovšem bývá, jednoduché věci jsou velmi chytré. Jeho časová složitost je $O(n \cdot log(n))$, takže je srovnatelný s rychlostí Heap sortu. Princip tohoto řadícího algoritmu je prostý, neustále posloupnost rozděluje na podposloupnosti, dokud nezíská posloupnosti s jedním prvkem. Dále je opětovně "slévá"nazpět s tím, že vyšší prvek posune na počátek nově složené posloupnosti. Merge sort nejprve získá sestupně seřazené podposloupnosti a poté je složí do jedné, která je již seřazená. Závěrečný krok tedy vypadá tak, že z každé podposloupnosti se vyberou první prvky a porovnají se. Vyšší z nich se vypíše jako první prvek srovnané posloupnosti a z původního seznamu se vymaže. Algoritmus poté porovná opět první prvky obou podposloupností a takto pokračuje, dokud není jeden ze seznamů prázdný. Poté už jenom převede zbytek prvků do nové, již seřazené posloupnosti.

Můžeme si všimnout, že pokud má posloupnost sudý počet členů, potom je rozdělování snadné. V prvním kroku se sudý počet rozdělí na dva stejně dlouhé seznamy. V případě, že má posloupnost lichý počet prvků, potom algoritmus jednomu seznamu přidělí o prvek navíc. Pro názornost průběhu Merge sortu je zde zobrazen obrázek příkladu. Tento obrázek pochází z učebního textu profesora Roberta C. Holta [6].



Obrázek 3.1 - Příklad Merge sortu

3.1.4 Ostatní algoritmy

V této kapitole budou připomenuty některé další řadící algoritmy. Jak už bylo jednou zmíněno, těchto algoritmů je spousta a i když zbývající "sleji"do jedné kapitoly (pozn. jako Merge sort), tak nezmíním všechny. Hodně ze zmíněných algoritmů je upravených do různých vylepšení a ještě více jich má vlastní základ, nebo přebírá z jiných algoritmů tu, či onu myšlenku. Zde je jenom pár řadících algoritmů, které stojí za to připomenout.

Block Merge Sort je algoritmus, který je v základu vylepšený Merge sort. Jeho časová složitost je $O(n \cdot log(n))$ a využívá kromě znalostí z Merge sortu i princip Insertion sortu. Tento algoritmus funguje tak, že posloupnost rozdělí na poloviny a v každé z těchto polovin prvky prochází jako Insert sort. Následně v obou polovinách posloupnosti vytváří něco jako srovnané podposloupnosti (bloky), které potom celé posouvá a slévá dohromady.

Drop Sort je poněkud zvláštní algoritmus. Hodí se pouze na konkrétní nasazení. Drop sort totiž připouští, že na výstupu je posloupnost sice srovnaná, ale rozhodně nemá stejný počet prvků. Slovo drop se dá přeložit jako "upustit", takže Drop sort v průběhu algoritmu

prvky z posloupnosti maže. Jinými slovy Drop sort probíhá stejně jako Bubble sort, ale ten si dal práci se zarovnáváním všech prvků. Drop sort tyto prvky porovná a nevyhovující člen posloupnosti jednoduše odstraní. Drop sort se dá naprogramovat aby z posloupnosti vytvořil posloupnost sestupnou (popřípadě vzestupnou). Nevyhovující prvek je v posloupnosti tedy ten, který je větší (popřípadě menší)než prvek před ním.

Tento algoritmus se dá tedy ještě vylepšit. Na stránkách o algoritmech [3] je například odstavec o vylepšení Drop sortu. Stačilo by prvky nevymazávat a vytvořit z nich novou posloupnost a na ní znovu zavolat Drop sort. Jedná se tedy o jakousi aplikaci "sít"na sypký materiál. Posloupnost by byla sice srovnána, ale i tak přicházíme o nějaké prvky. Tyto prvky zůstávají na dně pod "síty".

Selection sort je další algoritmus, který má složitost O(n^2). Tento algoritmus je teda svoji rychlostí srovnatelný s Bubble sortem, nebo Insert sortem. Co do rychlosti se nachází přesně mezi nimi. Neckář uvádí, že jeho hlavní výhodou (dokonce i oproti algoritmům se složitostí O($n \cdot log(n)$) je konstantní paměťová složitost. Selection sort zde uvádím zejména proto, že tento algoritmus neporovnává prvky. Jeho hlavním údělem je vybrat (z anglického slova select) prvky s maximální hodnotou a přesunout je na odpovídající pozici v posloupnosti. Jako první krok tedy vybere největší prvek, posune ho na počátek posloupnosti a poté ze zbylých prvků vybírá maximum na druhou pozici. Jednoduchá myšlenka, která se na rozdíl od jiných algoritmů velmi snadno napíše do jednotlivých kroků pro zápis algoritmu.

Další poněkud vtipný algoritmus se nazývá **Bogo sort**. V několika zdrojích je tento algoritmus uveden jako naprosto nepoužitelný, což je v pořádku, neboť byl vytvořen pouze pro studijní účely. Tento algoritmus se také nazývá Stupid sort (hloupý řadící algoritmus) nebo také Drunk man sort (řazení ochmelky). Funguje na principu, kdy zkontroluje seřazenost všech členů posloupnosti. Pokud jsou správně seřazené, tak se algoritmus ukončí. Pokud jsou prvky seřazené špatně, zcela náhodně posloupnost setřídí.

3.2 Užití algoritmů

Automatizace úkolů zjednodušuje práci člověku. Člověku by taková práce trvala nesrovnatelně déle a výsledek takového snažení by byl pravděpodobně i s výskytem většího množství chyb. Veškerá práce odvedená algoritmy na posloupnostech čísel se dále využívá v nesčetném množství provozů.

Nyní přejděme k praktické aplikaci těchto řadících algoritmů. Hledání minimální kostry v ohodnocených grafech je kupříkladu algoritmus s praktickým využitím. Graf si totiž můžeme představit jako body na mapě a hrany jsou jejich spoje. Vrcholy mohou být města, letiště, nádraží, cokoli tohoto typu. Při hledání minimální kostry v podstatě hledáme nejkratší cestu z bodu A do bodu B a k nalezení takové cesty se používá mnoha algoritmů. Pro příklad mohu uvést Jarníkův algoritmus (nebo také Primův), dále i Borůvkův a nebo i poslední z nich Kruskalův.

A je to právě Kruskalův algoritmus na nalezení minimální kostry v grafu, který využívá řadících algoritmů. Libovolný řadící algoritmus má za úkol seřadit všechny hrany grafu (ohodnocené) a Kruskalův algoritmus je poté využívá na vytvoření minimální cesty bez kružnice. Z toho všeho lze předpokládat, že tyto algoritmy mohou být využívány aplikacemi na hledání trasy, popřípadě i GPS navigací.

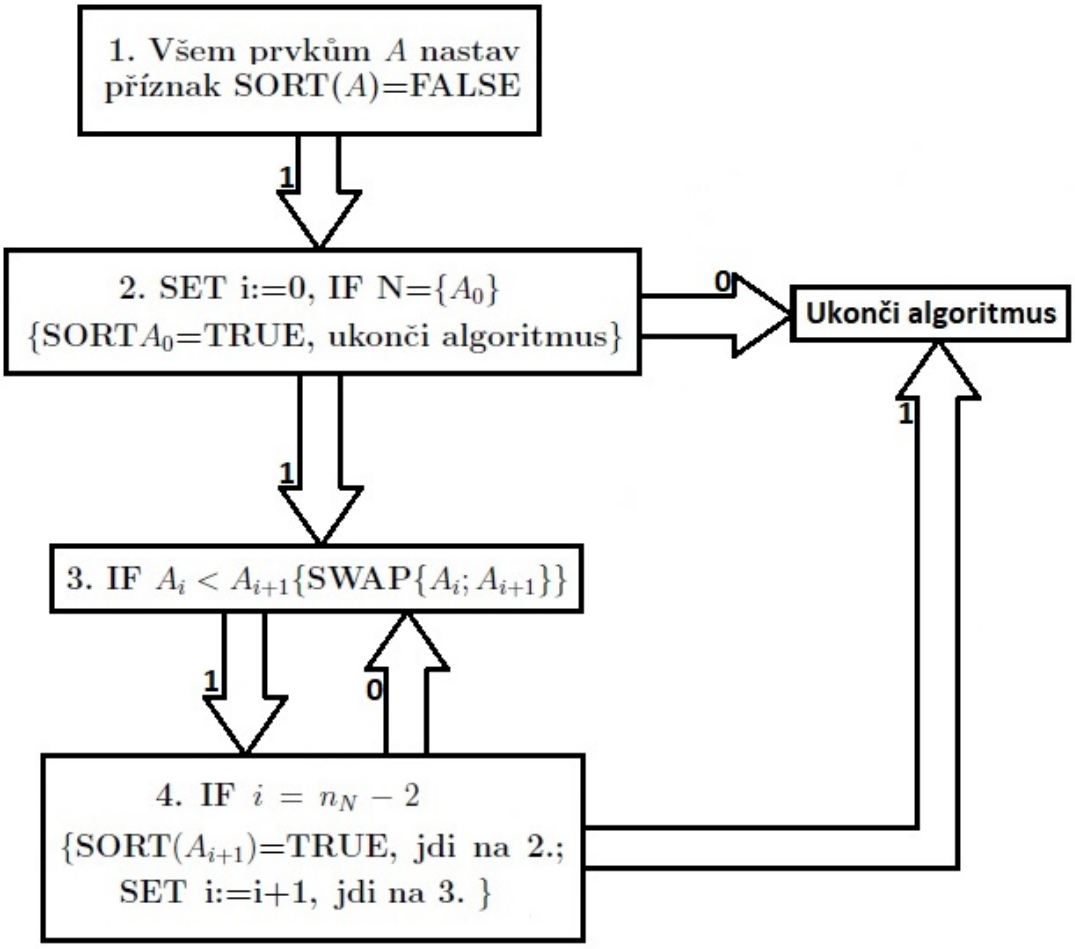
Učební text o aplikacích řazení [7] uvádí několik praktických použití řadících algoritmů. Ve článku zmiňují, že lidé na práci s daty upřednostňují seřazená data. Taková data se totiž dobře čtou a v seřazené množině se mnohem snadněji vyhledává. Dále tento článek uvádí, že v seřazeném seznamu je snadnější objevovat statistické vlastnosti jako jsou medián, momenty a podobně. V posledním bodu tohoto článku je i uvedeno, že řazení napomáhá porovnávání dvou a více různých seznamů. Krom toho i napomáhá provádění různých operací mezi nimi. Snadno si tedy z tohoto odstavce vyvodit, že řadící algoritmy usnadňují jakoukoli budoucí práci s daty. Takovou informaci si ostatně i lze představit na prvním příkladu použití v Kruskalově algoritmu.

Zdaleka největší shrnutí aplikace těchto algoritmů poskytuje článek Roberta Sedgewicka a Kevina Wayna [8]. Zde se uvádí praktické využití ve výpočetní technice, vládních organizacích, finančních institucích a v komerčních podnicích. Podle autorů je drtivá většina informací nejprve zpracována za pomocí třídících (řadících) algoritmů. Dále se vyjadřují k praktickým využitím v těchto oborech a institucích. Řadící algoritmy se používají například na seřazení dle jména, čísel transakce, seřazení dle času a místa, dle pošty a poštovního směrovacího čísla nebo podle adresy. Poté s takovými daty mohou všechny instituce řádně pracovat dále.

Velice konkrétní použití mají tyto algoritmy v simulacích, které jsou řízeny událostmi. Pro provedení takových vědeckých simulací efektivně je třeba užít vhodné datové struktury a vhodné algoritmy. Dále tento článek uvádí rozsáhlé psaní o aplikacích algoritmů v již zmíněném Kruskalově a Primově algoritmu. Krom těchto užití je zde zmíněno i užití v Huffmanově kódování, numerických operacích (výpočtech) a dalších typech prohledávání.

3.3 Diagramy

Na konci kapitoly 2.3.3 o řadícím algoritmu Insert sortu byla zmíněna zajímavost ohledně diagramů. Vzpomeňme si na zápis algoritmu po krocích, každý jeho krok si můžeme představit jako vrchol grafu a jednotlivé odkazy na jiné kroky můžeme vnímat jako hrany. Kupříkladu obrázek na další straně zobrazuje Bubble sort překreslený do diagramu.



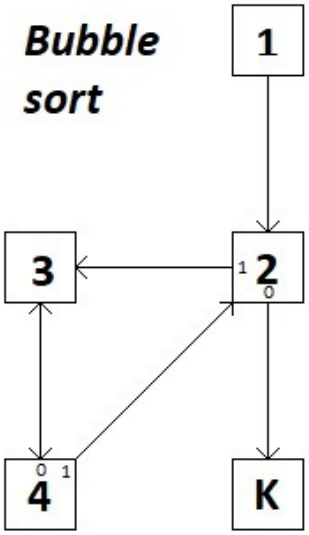
Obrázek 3.2 - Diagram Bubble sortu

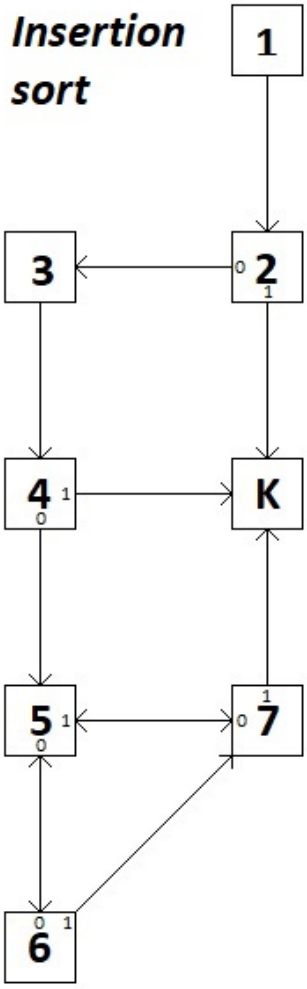
Obdobně by se přepsal do diagramu i Insertion sort, pro názornost však dejme pryč příkazy a detaily těchto kroků. O jakémkoli algoritmu tedy můžeme vyvodit několik závěrů už z pouhého pohledu na ně. Diagram algoritmu (dále pouze graf) je vždy neohodnocený a je orientovaný.

Poznámka: Očíslování 1,0 znázorňuje pouze rozhodnutí o funkci ***když****. Pokud je podmínka této funkce splněna (popř. nesplněna), značíme toto rozhodnutí jedničkou (popř. nulou).*

Podívejme se tedy na to, jak vypadají algoritmy Bubble sort a Insert sort v grafech. Poté se pokusme vyvodit další možné závěry o těchto algoritmech a zda je možné je pokud možno

"zhutnit". Čím méně vrcholů by algoritmus měl, tím rychleji by celý algoritmus pracoval. Mimo jiné by to i znamenalo, že algoritmus lze zapsat v krocích mnohem jednodušeji.





Nahoře obrázek 3.3 - Graf Bubble sortu.
Vpravo obrázek 3.4 - Graf Insertion sortu

Oba tyto algoritmy jsou psané bez jakéhokoli podkladu. Oba jsou ozkoušené na příkladech a fungují. Nicméně si můžeme povšimnout, že Insert sort je výrazně složitější, ačkoli má stejnou časovou složitost jako Bubble sort. V těchto řadících algoritmech jsou důležité především cykly a ty jsou patrné na obrázku. S těmito cykly operuje zejména funkce **když**, která opakovaně rozhoduje o nějaké proměnné, kterou jiným krokem měníme.

Nyní si povšimněme, že oba grafy mají vstupní a výstupní vrchol (vrcholy, do nichž hrany pouze vstupují, popř. vystupují). Vstupní vrchol (tedy krok č. 1) jsou vždy počáteční definice nutné pro celý průběh algoritmu. Výstupní vrchol je vždy konec algoritmu. Konec algoritmu je možné si tedy představit jako krok algoritmu. Dále si všimněme vrcholu č. 3 jak u Bubble sortu, tak u Insertion sortu. Tento vrchol nemá žádnou jinou funkci, než zadefinovat nové podmínky, nebo upravit nějakou z proměnných. Hlavní část cyklu tvoří u Bubble sortu vrcholy 2 a 4. Jinými slovy pro zjednodušení obou algoritmů můžeme vrchol č.3 vymazat. Nicméně je nezbytné jeho funkci (jeho příkazy) připsat jinému kroku v algoritmu, konkrétně tedy je možné tyto příkazy připsat na konec příkazů v kroku č.2, nebo na začátek v kroku č.4. Takovou úpravou by se celý algoritmus upravil a zjednodušil.

U Bubble sortu je toto jediná možná úprava, která by nezměnila průběh algoritmu. Ještě by se teoreticky dalo přesunout funkce kroku č.1 do druhého, obdobně tak se to dá udělat i u Insert sortu.

Druhá podstatná informace se dá vyčíst z obou těchto grafů a vlastně i odpovídá tomu, k čemu algoritmus má sloužit. Algoritmus musí být funkční, tedy musí se umět sám ukončit po dokončení své funkce. Oba řadící algoritmy jsou funkční, to jsme si již dokázali. Dokáží se algoritmy ukončit? Předpokládejme, že všechny podmínky funkce **když** fungují správně a algoritmus je tedy skutečně napsaný bez chyb. Další nutnost je dohlédnout na to, aby se algoritmus nedostal do "slepé uličky". Stejně tak je to i vidět v grafu, **NESMÍ** se tam nacházet vrchol, ze kterého neexistuje cesta do vrcholu K. Jinými slovy, každý vrchol musí mít alespoň jednu cestu do vrcholu K. Pokud se tohoto docílí, pak se algoritmus dokáže sám ukončit.

4 Závěr

4.1 Shrnutí

V první kapitole tohoto textu jsme shrnuli základní definice, princip a vlastnosti algoritmů obecně. Krom toho bylo i vysvětleno jak vlastně algoritmus "přemýšlí"a jak se na rozdíl od člověka rozhoduje. Bylo poukázáno na fakt, že ve světě psaní algoritmů a ve světe počítačů vůbec, je nutné definovat vše přesně. Dále jsme si připomněli jaké jsou různé typy algoritmů a jak je rozlišujeme. Další články této kapitoly poukázali více na řadící algoritmy a byly vysvětleny jejich různé časové složitosti.

V další kapitole jsme zaměřili již pouze na řadící algoritmy a byly popsány jejich unikátní vlastnosti. Tato kapitola se dále věnovala hlouběji dvěma základním řadícím algoritmům Bubble sortu a Insertion sortu. U obou z nich byl vysvětlen jejich princip důkladně a následně byl jejich princip osvětlen na příkladech. Poté jsme si u obou těchto algoritmů napsali algoritmus krok po kroku a dílčí kroky jsme si vysvětlili. Dále byly jako příklady uvedeny i zápisy v programovacích jazycích. Dokázali jsme si u nich jejich časovou složitost ze vzorce na součet prvků n prvků posloupnosti a uvedli jsme i případná vylepšení těchto algoritmů. U dílčích vylepšení byl naznačen jejich princip a také byl nastíněn algoritmus v krocích.

Třetí kapitola pojednávala o ostatních algoritmech, o aplikaci algoritmů a zvláštní kapitolu jsme věnovali i diagramům a grafům. Ostatní algoritmy byly zmíněny pouze okrajově. Popis jejich principu, funkce a složitosti byl srovnatelný s nástinem vylepšení algoritmů Bubble sortu a Insertion sortu (tj. Shaker sort a Shell sort). Krom ostatních algoritmů jsme si mohli přečíst i aplikaci všech těchto algoritmů a jejich použití ve světě. Na závěr jsme si připomněli, že algoritmy lze chápat jako grafy jak je známe z diskrétní matematiky. Tam jsme si ukázali, že nehledě na složitost našeho zápisu je můžeme upravit a tím je i vylepšit a zrychlit.

Nyní bych se rád vrátil úplně na začátek k otázkám, které byly položeny. Nejprve si je tedy připomeňme:

"Jsou řadící algoritmy důležité? Jsou potřebné? K čemu tedy slouží?"
Začněme nejprve od konce. Řadící algoritmy slouží tedy ke srovnání jakékoli číselné posloupnosti. U aplikací těchto algoritmů jsme si poukázali na fakt, že dokonce můžeme za vhodné úpravy seřadit libovolnou množinu podle čehokoli (ať už to je PSČ, adresa, jméno a podobně). Algoritmy jsou tedy skutečně potřebné, nicméně jejich práci je ve světě velmi snadné přehlédnout. Tyto algoritmy se přehlédnou snadno, neboť jsou jednoduše všude. Zvlášť nyní v dobách, kdy je elektronika takřka všudepřítomná. Ukázali jsme si, že tyto algoritmy usnadňují práci člověka a tuto práci usnadňuje tolik, že již existenci řadících algoritmů přestáváme vnímat. I kdyby řadící algoritmy neměly na první pohled patrnou důležitou funkci, tak jejich práce je nezbytná pro jakékoli jiné aplikace, softwary a další algoritmy.

"Existuje řadící algoritmus, který je jednoznačně nejlepší?"
Ano i ne. Jak jsme si odpověděli v předchozím dotazu, řadící algoritmy se používají všude možně. Algoritmy vyžaduje člověk ke konkrétní práci. Když potřebujeme k práci nějaký řadící algoritmus, vyhledáváme zejména jeho rychlost a přesnost. Nicméně v dnešním světě hraje roli i cena a lidská práce uplatněná při tvorbě těchto algoritmů. Pokud v práci chceme nejrychlejší algoritmus, vyhledáváme ty algoritmy s nejvhodnější časovou složitostí. Dále je třeba ale zohlednit i na jaká data algoritmus pouštíme. Pokud již máme částečně srovnaná data, hodí se nám ten algoritmus více než onen. Pokud máme data malého objemu, potom se nám nemusí vyplatit platit za složitý algoritmus s časovou složitosti $O(n \cdot log(n))$. Jeho práci může stejně dobře zvládnout i algoritmus s exponenciální složitostí. Proto je odpověď ano, nejlepší algoritmy jsou ty s nejlepší časovou složitostí, ale odpověď je i ne, algoritmus volíme dle své potřeby.

4.2 Resumé

Tato bakalářská práce pojednává o základních algoritmech, o jejich principech a využitích. Práce je rozdělena do tří hlavních částí. V první části jsou shrnuty základní definice a úvod do algoritmů obecně. Druhá část práce je o samotných řadících algoritmech se zvláštním zaměřením na Bubble sort a Insert sort. V této části se nachází jejich princip, obecný zápis algoritmů, jejich vylepšení a odvození jejich časové složitosti. Třetí část práce je zaměřena na ostatní řadící algoritmy, kterým byla věnována menší pozornost. Je zde osvětlen jejich princip a jejich časová složitost k porovnání. V této kapitole se nachází i zmínka o využití všech algoritmů obecně. Cílem této práce je vysvětlit základní principy řadících algoritmů a předvést jejich využití.

4.3 Summary

This bachelor thesis deals with basic algorithms, their principles and uses. Work is divided into three main parts. The first part summarizes basic definitions and introduction to algorithms in general. The second part of the thesis is about the sorting algorithms with a special focus on Bubble sort and Insert sort. In this section are their principle, the general entry of the algorithms, their improvement and inference of their time complexity. The third part of the thesis is focused on the other sorting algorithms, which have been given less attention. Their principle and their time complexity are highlighted in this chapter for their comparison. This chapter also includes a reference about usage of algorithms in general. The aim of this work is to explain the basic principles of sorting algorithms and to demonstrate their use.

4.4 Použité zdroje

Reference

[1] vývojový tým UIS Mendelu Přehled typických složitostí, 6.března 2018, [online] Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=27620.

[2] Hordějčuk V., *Bubble Sort*, aktualizace 2018, [online] Dostupné z: http://voho.eu.

[3] Neckář J., *Algoritmy*, aktualizace 2016, [online] Dostupné z: https://www.algoritmy.net.

[4] Agilowen Insertion Sort Algorithm and Time Complexity, 4.února 2011, [online] Dostupné z: https://www.youtube.com/watch?v=ghdvwdzrWWc.

[5] Rajat Mishra, vývojový tým GeeksforGeeks Shell sort algorithm, [online] Dostupné z: https://www.geeksforgeeks.org/shellsort/.

[6] Prof. Robert C. Holte, Dr. Denys Duchier, University of Ottawa Merge sort (Section 5.4.4), [online] Dostupné z: https://webdocs.cs.ualberta.ca/~holte/T26/merge-sort.html.

[7] CS 104 Teaching Team Data structures and Object-Oriented Design, 31.listopadu 2013, [online] Dostupné z: http://www-bcf.usc.edu/~dkempe/CS104/10-31.pdf.

[8] Robert Sedgewick, Kevin Wayne Sorting Applications, 18.března 2018, [online] Dostupné z: https://algs4.cs.princeton.edu/25applications/.