UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2024 David Polívka

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Vizualizace a analýza třídících algoritmů

Bakalářská práce

2024 David Polívka

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**David Polívka**
Osobní číslo:	**I21363**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Vizualizace a analýza třídících algoritmů.**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Bakalářská práce se bude zabývat zkoumáním různých třídících algoritmů. V teoretická části se student zaměří na principy, výkonnost a efektivitu jak základních,tak i pokročilých třídících metod. V praktické části dojde k implementaci aplikace pro vizualizaci těchto algoritmů, poskytující uživatelům intuitivní způsob, jak pochopit a zkoumat jejich fungování.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

VIRIUS, Miroslav. Programování v C#: od základů k profesionálnímu použití. Knihovna programátora (Grada). Praha: Grada Publishing, 2021. ISBN 978-80-271-1216-6
MEHLHORN, Kurt a SANDERS, Peter. Algorithms and data structures: the basic toolbox. Berlin: Springer, c2010. ISBN 978-364-2096-822.
CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L. a STEIN, Clifford. Introduction to algorithms. Fourth edition. Cambridge, Massachusetts: The MIT Press, [2022]. ISBN 978-026-2046-305
RYANT, Ivan. Algoritmy a datové struktury objektově. V Praze: Ivan Ryant, [2017]. ISBN 978-80-270--1660-0

Vedoucí bakalářské práce: **Ing. Jan Panuš, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. prosince 2023**
Termín odevzdání bakalářské práce: **10. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2024

Prohlašuji:

Práci s názvem Vizualizace a analýza třídících algoritmů jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 1. 5. 2024

David Polívka v.r.

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat vedoucímu mé bakalářské práce Ing. Janu Panušovi, Ph.D. za jeho čas, pomoc a odborné vedení. Jeho znalosti, připomínky a trpělivost byly podstatné pro můj úspěch při zpracovávání této práce.

ANOTACE

Bakalářská práce se bude zabývat zkoumáním různých třídících algoritmů. V teoretické části se student zaměří na principy, výkonnost a efektivitu jak základních, tak i pokročilých třídících metod. V praktické části dojde k implementaci aplikace pro vizualizaci těchto algoritmů, poskytující uživatelům intuitivní způsob, jak pochopit a zkoumat jejich fungování.

KLÍČOVÁ SLOVA

programování, algoritmus, třídící algoritmy, výkon, vizualizace, porovnávací metody algoritmů, C#, WPF

TITLE

Visualization and analysis of sorting algorithms

ANNOTATION

The bachelor's thesis will focus on the examination of various sorting algorithms. In the theoretical part, the student will concentrate on the principles, performance, and efficiency of both basic and advanced sorting methods. In the practical part, an application for these algorithms will be implemented, providing users with an intuitive way to understand and examine their functioning.

KEYWORDS

programming, algorithm, sorting algorithms, performance, visualization, algorithm comparison methods, C#, WPF

OBSAH

SEZNAM ILUSTRACÍ A TABULEK....10
TERMINOLOGIE....12
ÚVOD....13
1 ANALÝZA TŘÍDÍCÍCH ALGORITMŮ....14
1.1 Úvod....14
1.2 Základní pojmy....14
1.2.1 Algoritmus....14
1.2.2 Třídění....14
1.2.3 Časová složitost....14
1.2.4 Prostorová (paměťová) složitost....15
1.2.5 In-place třídění....15
1.2.6 Out-of-place třídění....15
1.2.7 Kategorizace....16
1.3 Základní třídící algoritmy....17
1.3.1 Bubble sort....17
1.3.2 Selection sort....19
1.3.3 Insertion sort....21
1.3.4 Merge sort....23
1.3.5 Quick sort....26
1.4 Pokročilé třídící algoritmy....30
1.4.1 Heap sort....30
1.4.2 Counting sort....36
1.4.3 Radix sort....39
1.4.4 Bucket sort....41
1.4.5 Shell sort....44
1.4.6 Comb sort....46
1.4.7 Cocktail sort....48
1.4.8 Timsort....51
1.4.9 Bitonic sort....54
1.4.10 Pigeonhole sort....57
1.5 Porovnání algoritmů....59

2 PRAKTICKÁ ČÁST ... 60
2.1 Použité technologie ... 60
2.1.1 Visual Studio 2022 ... 60
2.1.2 Programovací jazyk C Sharp (C#) ... 60
2.1.3 Windows Presentation Framework (WPF) ... 60
2.2 Požadavky aplikace ... 60
2.3 Návrh aplikace ... 61
2.3.1 Ovládací panel ... 61
2.3.2 Panel s informacemi ... 61
2.3.3 Interaktivita a ovládání ... 61
2.3.4 Export výsledků a import dat ... 62
2.4 Implementace ... 62
2.4.1 Třídy ... 62
2.4.2 Ukázka třídy BubbleSort ... 64
2.4.3 Třída MainWindow ... 65
2.4.4 Třída ResultsWindow ... 68
2.4.5 Třída CompareAlgorithms ... 68
2.5 Ukázka aplikace ... 69
2.5.1 Vizualizace ... 69
2.5.2 Bez vizualizace ... 70
2.5.3 Hromadné porovnání algoritmů ... 71
2.5.4 Import vlastních dat ... 72
2.5.5 Export výsledků ... 73
2.5.6 Naměřené hodnoty ... 74
ZÁVĚR ... 77
POUŽITÁ LITERATURA ... 78

SEZNAM ILUSTRACÍ A TABULEK

Obrázek 1 Bubble sort [10]....18
Obrázek 2 - Selection sort [11]....20
Obrázek 3 - Insertion sort [12]....22
Obrázek 4 - Merge sort [13]....24
Obrázek 5- Quick sort [14]....26
Obrázek 6 - Quick sort[15]....26
Obrázek 7 - Quick sort [16]....27
Obrázek 8 - Quick sort [17]....27
Obrázek 9 - Quick sort [18]....28
Obrázek 10 - Quick sort [19]....28
Obrázek 11 - Min, Max heap [20]....31
Obrázek 12 - Heap sort [21]....34
Obrázek 13 - Counting sort [22]....36
Obrázek 14 - Counting sort [23]....36
Obrázek 15 - Counting sort [24]....37
Obrázek 16 - Counting sort [25]....37
Obrázek 17 - Counting sort [26]....38
Obrázek 18 - Radix sort [27]....40
Obrázek 19 - Bucket sort [28]....42
Obrázek 20 - Shell sort [29]....45
Obrázek 21 - Comb sort [30]....47
Obrázek 22 - Comb sort [31]....47
Obrázek 23 - Cocktail sort [32][33][34][35][36][37][38]....49
Obrázek 24 - Tim sort [39]....51
Obrázek 25 - Tim sort [40]....52
Obrázek 26 - Tim sort [41]....52
Obrázek 27 - Tim sort [42]....52
Obrázek 28 - Bitonic sort [43]....54
Obrázek 29 - Bitonic sort [44]....55
Obrázek 30 - Pigeonhole sort [45]....57
Obrázek 31 - Porovnání více algoritmů menu Zdroj: vlastní....67
Obrázek 32 - Main Window Zdroj: vlastní....67
Obrázek 33 - Okno results Zdroj: vlastní....68
Obrázek 34 - Okno CompareAlgorithms Zdroj: vlastní....69
Obrázek 35 - Ukázka vizualizace Zdroj: vlastní....70
Obrázek 36 - Ukázka bez vizualizace Zdroj: vlastní....70
Obrázek 37 - Hromadné třídění Zdroj: vlastní....71
Obrázek 38 - Testovací csv data Zdroj: vlastní....72
Obrázek 39 - Import dat Zdroj: vlastní....72
Obrázek 40 - Export dat Zdroj: vlastní....73
Obrázek 41 - Exportovaný soubor Zdroj: vlastní....73

Tabulka 1 Porovnání algoritmů....59
Tabulka 2 - Porovnání algoritmů 1. Test....74
Tabulka 3 - Porovnání algoritmů 2. Test....75

Tabulka 4 - Porovnání algoritmů 3. Test 76

TERMINOLOGIE

C#	programovací jazyk
WPF	framework pro tvorbu formulářových aplikací
.NET	framework, který poskytuje různé služby pro své běžící aplikace
Framework	rozsáhlá sada vývojářských nástrojů
XAML	jazyk značek pro definování uživatelského rozhraní
Python	programovací jazyk
VS	Visual studio – vývojové prostředí
IDE	softwarové prostředí, které poskytuje komplexní služby pro vývoj
TextBox	grafický prvek, který umožňuje vstup textu do aplikace
Canvas	grafický kontejner pro kreslení obrazců
CheckBox	grafický prvek ve formě čtverečku pro zaškrtnutí
Slider	grafický prvek ve formě posuvníku

ÚVOD

Tato bakalářská práce se zaměřuje na zkoumání a porovnávání různých třídících algoritmů. Cílem práce je poskytnout čtenářům hlubší pochopení jejich principů, použití, výkonnosti a efektivity, a to jak teoreticky, tak i prakticky.

V teoretické části se práce soustředí na zkoumání základních, ale také pokročilých třídících metod, kde bude zkoumat jejich principy, složitost, vhodnost nasazení v různých situacích a efektivitu. Toto vše je nezbytné pro pochopení, správný výběr a využití třídících algoritmů v praxi.

V praktické části bude popsán vývoj aplikace pro vizualizaci třídících algoritmů. Tato aplikace bude sloužit pro intuitivní porozumění fungování a vlastností různých algoritmů. V rámci praktické části budou rovněž popsány použité technologie a programovací jazyky k vývoji aplikace. Konkrétně využití programovacího jazyka C# a frameworku WPF (Windows Presentation Foundation).

1 ANALÝZA TŘÍDÍCÍCH ALGORITMŮ

1.1 Úvod

Třídící algoritmy jsou v oblasti informačních technologií a programování základními nástroji, které se používají pro uspořádání prvků např. čísel v určitém seznamu nebo struktuře do určitého pořadí. Tyto algoritmy jsou často navrženy tak, aby řadily data, prvky podle číselných nebo abecedních kritérií. Třídící algoritmy jsou schopny zefektivnit vyhledávání a přístup k datům. To je důležité z hlediska výkonu aplikací. Takové aplikace jsou například vyhledávače, rozsáhlé databázové systémy nebo aplikace pro analýzu dat. Existuje mnoho různých třídících algoritmů, které mají své vlastnosti, využití, výhody a nevýhody.[1]

1.2 Základní pojmy

1.2.1 Algoritmus

Algoritmus lze obecně definovat jako soubor přesně definovaných instrukcí nebo pravidel pro řešení problému nebo dosažení určitého výsledku. Takový algoritmus by měl splňovat následující vlastnosti:

a) **Správnost** – Výsledek algoritmu musí být správný.
b) **Konečnost** – Algoritmus musí někdy po konečném počtu kroků skončit.
c) **Determinovanost** – Postup algoritmu je jasně daný.
d) **Opakovatelnost** – Při vložení stejných vstupních dat musí algoritmus vrátit znovu stejný výstup.
e) **Rezultativnost** – Algoritmus musí mít vždy výstup nebo hlášení o chybě.
f) **Obecnost** – Algoritmus by měl být aplikovatelný nejen na jeden konkrétní problém.[2]

1.2.2 Třídění

Třídění nebo řazení je proces uspořádávání prvků (čísel, slov atd.) do požadovaného pořadí nebo struktury. Třídění je závislé na zvolených kritériích. Kritéria jako vzestupné nebo sestupné řazení určují jestli prvky budou řazeny od nejnižší hodnoty k nejvyšší nebo naopak. Další běžné kritérium je abecední řazení, kde jsou prvky řazeny podle abecedního pořadí. Tento proces je důležitý zejména pro efektivní vyhledávání či zpracování dat.[3]

1.2.3 Časová složitost

Časová složitost u algoritmů definujeme jako měřítko toho, jak se mění doba provádění algoritmu v závislosti na velikosti vstupních dat. Čas se ale neměří v sekundách, ale počtem

provedených operací, kde se každá operace bere jako bezrozměrná jednotka. Časová složitost může být vyjádřena pro nejhorší, průměrný a nejlepší případ.

Časovou složitost vyjadřujeme pomocí asymptotické notace, kterou zapisujeme pomocí Landauovy notace jako $O(f(N))$. Tento zápis znamená, že náročnost algoritmu je menší než $A + B \times f(N)$, kde A a B jsou vhodně zvolené konstanty a N je velikost vstupních dat. Obecně nás nezajímá počet operací pro různé rozsahy vstupních dat n, ale tendence jejich růstu při zvětšujícím se n. Proto lze zanedbat aditivní a multiplikativní konstanty. Zajímá nás jen chování funkce pro N.[4]

1.2.4 Prostorová (paměťová) složitost

Paměťová složitost algoritmu nám určuje velikost paměťového prostoru potřebného pro své provádění. Zde zahrnujeme jak paměť potřebnou pro uložení vstupních dat, tak i dodatečnou paměť pro dočasné proměnné. Prostorová složitost se nemusí rovnat časové složitosti. Prostorová notace se vyjadřuje také pomocí asymptotické notace. Celkovou prostorovou složitost můžeme tedy vyjádřit jako například $O(n)$, kde n je velikost vstupu.[5]

1.2.5 In-place třídění

In-palce nebo také In-situ třídění je algoritmus pro řazení dat, který provádí řazení přímo na původní datové struktuře bez potřeby významného množství dodatečné paměti. Používá pro změnu pořadí prvků pouze malý, fixní počet proměnných, které nejsou závislé na velikosti vstupních dat. To znamená, že tyto algoritmy minimalizují paměťové nároky při třídění a jsou tedy vhodné pro situace, kde máme omezený paměťový prostor, na druhou stranu některé In-place algoritmy mohou být méně efektivní z hlediska časové složitosti. In-place algoritmus obvykle používá dodatečný prostor $O(\log n)$, což je méně než lineární růst.[6]

Příklady In-place algoritmů: Bubble sort, Selection sort, Insertion sort, Heapsort, Quciksort

1.2.6 Out-of-place třídění

Out-of-place třídění jsou algoritmy, které vyžadují dodatečný prostor pro třídění dat. Algoritmy tedy vytváří struktury, do kterých ukládají již seřazená data. Tyto algoritmy často vyžadují více paměťového prostoru než In-place, protože je potřeba alokovat místo pro kopii vstupních dat nebo pro dočasné proměnné či pomocné struktury.

Příklady Out-of-place algoritmů: Merge sort, Radix sort, Timsort

1.2.7 Kategorizace

Třídící algoritmy můžeme kategorizovat podle různých kritérií, které popisují jejich vlastnosti a použití. To je užitečné pro správný výběr vhodného algoritmu pro konkrétní problém. Níže jsou uvedeny různé aspekty, podle kterých můžeme třídící algoritmy kategorizovat.

Stabilita

Stabilita je vlastnost, která zachovává původní pořadí stejných prvků ve vstupním poli i po seřazení. Třídící algoritmus je tedy považován za stabilní, pokud prvky s rovnocennými klíči zůstanou po třídění v takovém relativním pořadí, v jakém byly ve vstupním poli. Za stabilní algoritmy považujeme například Bubble sort, Insertion sort a Merge sort. Nestabilními algoritmy jsou například Heap sort, Quick sort a Selection sort. [72]

Adaptibilita

Adaptibilní algoritmus je takový, který je schopen efektivněji třídit data, která jsou již částečně seřazena, nebo s malým počtem inverzí. Za adaptibilní algoritmus můžeme považovat takový, u kterého se jeho výkon zlepšuje s rostoucím počtem již seřazených prvků ve vstupním poli. Za adaptivní algoritmus můžeme označit například Insertion sort, protože provede méně porovnání a výměn na již částečně seřazeném poli.

Rekurzivní algoritmus

Rekurzivní algoritmus je takový, který řeší třídění tím, že rozdělí data na menší podmnožiny, které jsou poté rekurzivně tříděny stejnou metodou. Po dokončení třídění menších částí jsou výsledky sloučeny do konečného seřazeného celku.

Iterativní algoritmus

Iterativní algoritmus je takový, který používá opakované průchody skrz data pro seřazení. Opakuje určité kroky, dokud není pole zcela seřazené. Takový algoritmus je například Bubble sort, který postupně prochází prvky a provádí porovnávání a výměny na základě iterativních pravidel.

Komparativní algoritmus

Komparativní algoritmus závisí na porovnávání prvků mezi sebou pro určení jejich relativního pořadí. Tato metoda je flexibilní a může být použita pro téměř jakýkoliv datový typ, kde lze definovat porovnávací operaci.[72]

Nekomparativní algoritmus

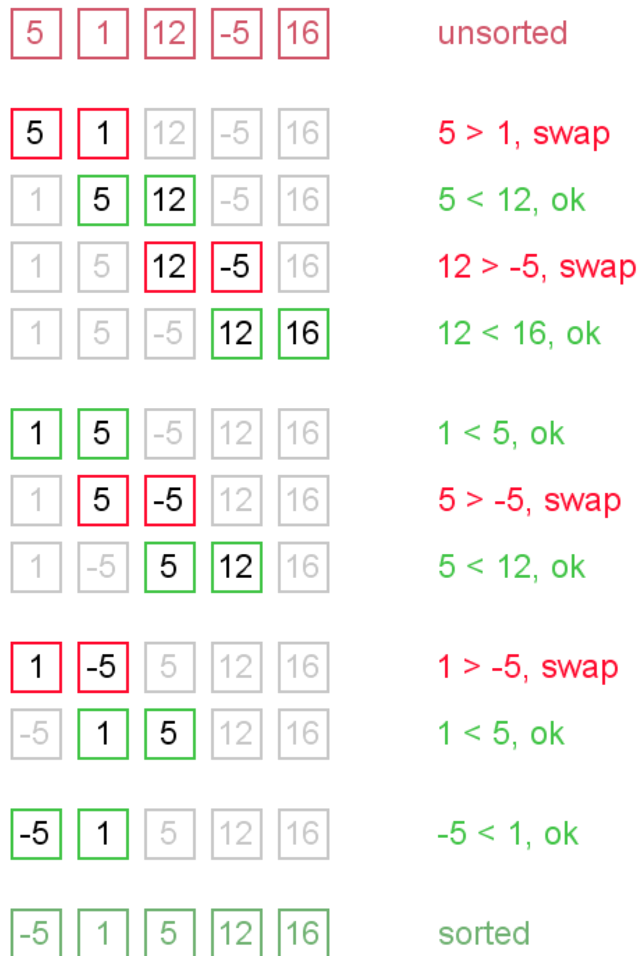
Nekomparativní algoritmus netřídí data na základě přímého porovnávání prvky mezi sebou. Místo toho používá strukturální vlastnosti dat k jejich organizaci. Například Counting sort, Bucket sort nebo Radix sort jsou příklady nekomparativních algoritmů, které závisí na distribuci datových prvků podle jejich hodnot, a ne na přímém porovnávání.[72]

1.3 Základní třídící algoritmy

1.3.1 Bubble sort

Bubble sort je jedním ze základních populárních algoritmů, který je stabilní a jednoduchý. Algoritmus opakovaně prochází seznam a porovnává sousední prvky a vyměňuje je, dokud není seznam ve správném pořadí. Funguje na principu „bublání" nejmenšího nebo největšího prvku v seznamu na jeho konec nebo začátek.

Algoritmus začíná na začátku seznamu. Prochází seznam od začátku do konce a porovnává každý pár sousedních prvků. Pokud je první prvek v páru větší než druhý, vymění jejich pozice – pro vzestupné řazení. Pro sestupné řazení vyměňuje prvky, pokud je první prvek menší než druhý. Po dokončení prvního kompletního průchodu seznamem se algoritmus opět vrátí na začátek seznamu a opakuje porovnávání a výměny. Toto se opakuje, dokud nebude seznam správně seřazený. Obecně lze říci, že k seřazení pole potřebuje *n-1* vnějších cyklů, kde *n* je počet prvků seznamu.[7]



Obrázek 1 Bubble sort [10]

Možná optimalizace

Algoritmus můžeme do jisté míry optimalizovat tak, že si zapamatujeme pozici poslední výměny v předchozím průchodu a v dalším průchodu budeme procházet jen do této pozice, protože seznam za touto pozicí bude již seřazený.[9]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $O(n)$[8]

Časová složitost v nejhorším případě: $O(n^2)$[8]

Časová složitost v průměrném případě: $O(n^2)$[7]

Prostorová složitost: $O(1)$

Stabilní: **Ano**

Výhody a nevýhody

Výhody Bubble sortu spočívají v tom, že je jednoduchý na pochopení a implementaci, nepotřebuje dodatečnou alokaci paměti (In-place třídění) a je stabilní.

Na druhou stranu má nízkou efektivitu, protože má průměrnou i nejhorší časovou složitost $O(n^2)$, a to z něj dělá neefektivní algoritmus pro třídění velkých polí. Oproti algoritmům jako je Quick sort nebo Merge sort je výrazně pomalejší, a to zejména u větších datových polí. Dále je nepraktický pro reálné využití, kvůli své nízké efektivitě.

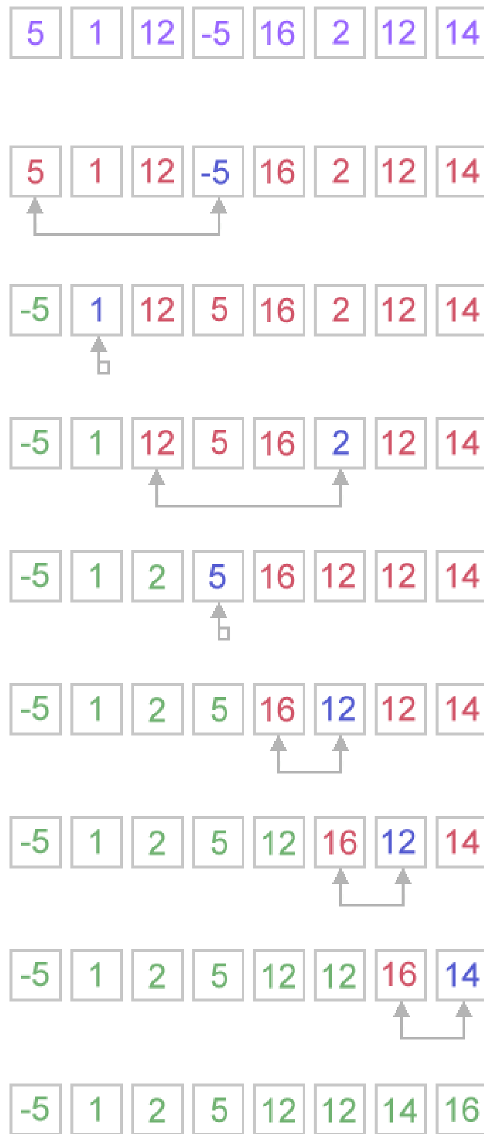
Implementace v jazyce C#

```
static void BubbleSort(int[] arr)
        {
            int temp;
            for (int i = 0; i < arr.Length - 1; i++)
            {
                for (int j = 0; j < arr.Length - i - 1; j++)
                {
                    // Porovnání páru prvků
                    if (arr[j] > arr[j + 1])
                    {
                        // Prohození prvků
                        temp = arr[j];
                        arr[j] = arr[j + 1];
                        arr[j + 1] = temp;
                    }
                }
            }
        }
```

1.3.2 Selection sort

Selection sort je další třídící základní populární třídící algoritmus. Funguje na principu opakovaného vyhledávání minimálního – pro vzestupné řazení nebo maximálního – pro sestupné řazení prvku v neuspořádané části pole a jeho výměny s prvkem na aktuální pozici v uspořádání. Toto se opakuje pro všechny pozice v poli, dokud není seřazené celé.

Algoritmus začne na první pozici pole, která se označí za aktuální pozici. Poté prochází zbytek pole za aktuální pozicí a hledá nejmenší prvek (pro vzestupné řazení). Když najde nejmenší prvek, prohodí ho s prvkem na aktuální pozici. Poté je aktuální pozice posunuta a celý proces se opakuje, dokud není celé pole seřazené. Algoritmus tedy postupně vybírá nejmenší nebo největší prvek a umisťuje ho na správnou pozici v poli. [50][51]



Obrázek 2 - Selection sort [11]

Možná optimalizace

Algoritmus můžeme optimalizovat tím, že zkrátíme délku procházené části pole s každou iterací, protože s každým nalezením maximálního nebo minimálního prvku a jejich výměnou s prvkem na aktuální pozici je konec či začátek (záleží z jaké strany postupujeme) pole již seřazený a nemusí se znovu procházet.

Časová a prostorová složitost

Časová složitost v nejlepším, nejhorším a průměrném případě: $O(n^2)$

Prostorová složitost: $O(1)$

Stabilní: **Ne**

Výhody a nevýhody

Výhodami Selection sort jsou jednoduchost na pochopení a implementaci. Zároveň je to In-place algoritmus, tudíž nevyžaduje dodatečnou paměť. Je vhodný pro malé datové sady.

Nevýhodou je vysoká časová složitost ve všech případech $O(n^2)$, což z něj dělá méně efektivní algoritmus, než jsou například Quick sort, Heap sort nebo Merge sort a je tedy nevhodný pro velké datové sady. [50]

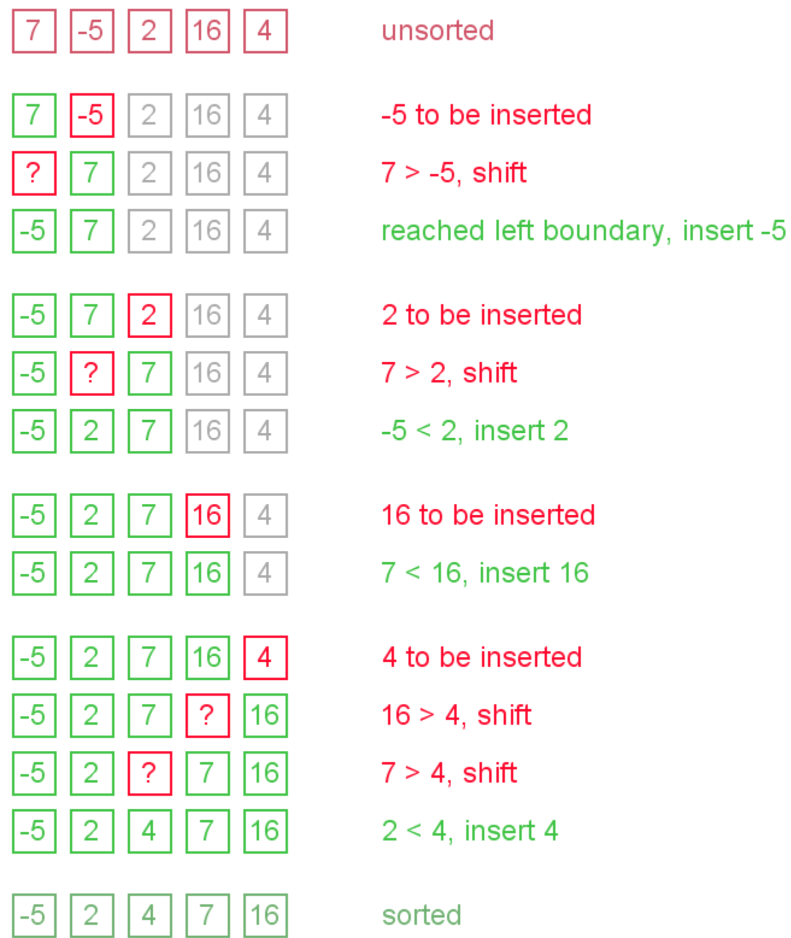
Implementace v jazyce C#

```
static void SelectionSort(int[] arr)
        {
            int n = arr.Length;

            for (int i = 0; i < n - 1; i++)
            {
                // Nalezení indexu minimálního prvku v neuspořádané části
                int minIndex = i;
                for (int j = i + 1; j < n; j++)
                    if (arr[j] < arr[minIndex])
                        minIndex = j;

                // Výměna minimálního prvku s prvním prvkem neuspořádané části
                int temp = arr[minIndex];
                arr[minIndex] = arr[i];
                arr[i] = temp;
            }
        }
```

1.3.3 Insertion sort

Insertion sort je základní řadící algoritmus, který je založen na porovnávání prvků. Algoritmus funguje tak, že postupně prochází pole a v každém kroku vloží prvek na správné místo v již seřazené části. Je In-place, takže nevyžaduje další alokaci paměti. [52]

Algoritmus zprvu označí první prvek v seznamu již za seřazený a začíná od druhého prvku. Pro každý prvek od druhého prvku dále opakuje: uložení hodnoty aktuálního vybraného prvku do dočasné proměnné. Dále prochází seřazenou část pole zprava doleva a posouvá prvky, které jsou větší (pro vzestupné řazení) o jednu pozici dopředu (doprava), dokud nenajde správnou pozici pro vložení dočasného prvku. Poté je prvek vložen na správné místo. Toto se opakuje pro každý prvek pole tzn., že počet iterací je $n-1$, kde n je počet prvků v poli. [53]



Obrázek 3 - Insertion sort [12]

Časová a prostorová složitost

Časová složitost v nejlepším případě: ***O(n)***

Časová složitost v nejhorším případě: $O(n^2)$

Časová složitost v průměrném případě: $O(n^2)$

Prostorová složitost: ***O(1)***

Stabilní: **Ano**

Výhody a nevýhody

Výhodami Insertion sortu jsou jednoduchost, efektivita pro malé nebo téměř seřazené datové sady, stabilita, a to, že je In-place, tudíž nevyžaduje další alokaci dodatečné paměti. V nejlepším případě je jeho časová složitost ***O(n)***.

Hlavní nevýhodou je neefektivnost pro velké datové sady s časovou složitostí $O(n^2)$ v nejhorším případě. [52]

Implementace v jazyce C#

```
static void InsertionSort(int[] array)
        {
            // Začíná se od druhého prvku
            for (int i = 1; i < array.Length; i++)
            {
                // Aktuální prvek
                int current = array[i];
                // Předcházející prvek
                int j = i - 1;

                // Dokud není nalezen prvek větší než current
                while (j >= 0 && array[j] > current)
                {
                    // Posunutí prvku
                    array[j + 1] = array[j];
                    // Posunutí se k dalšímu prvku
                    j = j - 1;
                }
                // Vložení prvku na správnou pozici
                array[j + 1] = current;
            }
        }
```

1.3.4 Merge sort

Merge sort je základní, často používaný, stabilní třídící algoritmus. K řazení využívá přístup „rozděl a panuj“. Nejdříve rozdělí pole na menší podpole, dokud každé pole neobsahuje pouze jeden prvek. Poté tato pole začne zpět slučovat, přičemž je každé podpole seřazeno, dokud není získáno původní seřazené pole.[54]

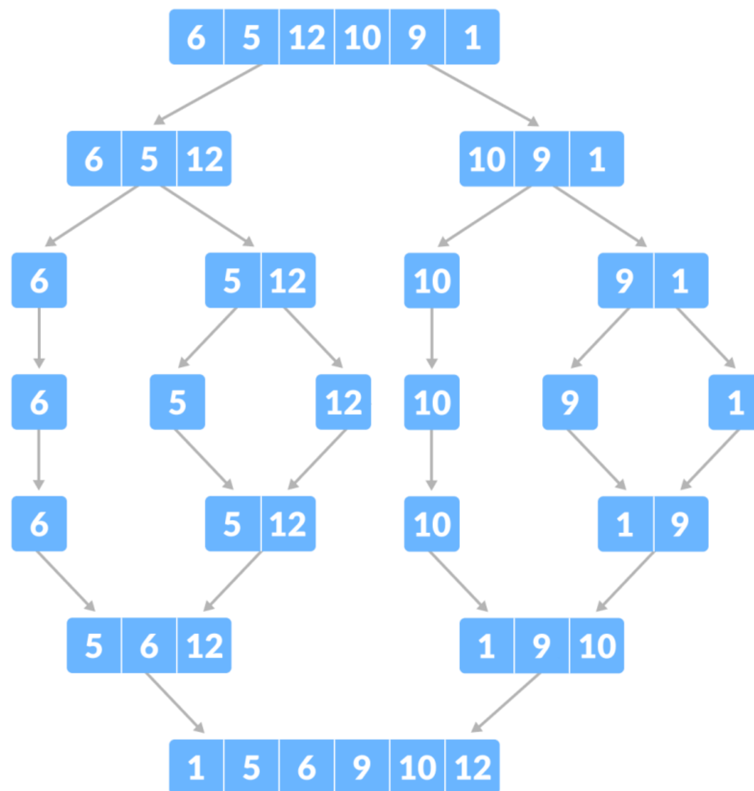
Rozdělování

Algoritmus začne původní pole rozdělovat na menší podpole, obvykle rozdělením na polovinu. Pokud původní pole obsahuje sudý počet prvků, je rozděleno na půlku. Pokud pole obsahuje lichý počet prvků, tak bude jedna část obsahovat o jeden prvek více než druhá část. Dále se nové podpole rekurzivně dělí, dokud nejsou získána podpole, která mají pouze jeden prvek nebo jsou prázdná.

Slučování

Poté co je původní pole rozděleno na nejmenší možné podpole, začne je algoritmus opět slučovat. Při slučování dvou podpolí do jednoho vybírá menší prvek z každého podpole a vkládá jej do výsledného pole v seřazeném pořadí, dokud se jedno z polí nevyprázdní. Poté zbytek druhého pole nakopíruje do nově vzniklého pole. Poté je celé původní pole nahrazeno

seřazeným seznamem. Toto opakuje, dokud nejsou všechna podpole sloučena do jednoho seřazeného pole. [55]



Obrázek 4 - Merge sort [13]

Časová a prostorová složitost

Časová složitost v nejlepším, nejhorším, průměrným případě: $O(n \times \log n)$

Prostorová složitost: $O(n)$

Stabilní: **Ano**

Výhody a nevýhody

Merge sort je velmi efektivní pro třídění velkých datových sad, protože jeho časová náročnost je konzistentně $O(n \times \log n)$. A jelikož je stabilní, tak se hodí pro třídění dat, kde je důležité zachovat původní pořadí stejných prvků.

Z důvodu, že to není In-place algoritmus, tak vyžaduje dodatečný prostor pro své operace, a to může být limitující při práci v prostředí s omezenou pamětí. Je také méně efektivní pro práci s menšími datovými sady.

Implementace v jazyce C#

```
// Slučování dvou podpolí
        static void Merge(int[] array, int left, int middle, int right)
        {
            // Dočasné pole pro levou a pravou část
            int[] leftArray = new int[middle - left + 1];
            int[] rightArray = new int[right - middle];

            // Kopírování do polí
            Array.Copy(array, left, leftArray, 0, middle - left + 1);
            Array.Copy(array, middle + 1, rightArray, 0, right - middle);

            // Indexy pro aktuální pozici v polích
            int i = 0; // Levé pole
            int j = 0; // Pravé pole
            int k = left; // Hlavní pole

            // Sloučení polí do původního pole
            while (i < leftArray.Length && j < rightArray.Length)
            {
                if (leftArray[i] <= rightArray[j])
                {
                    array[k] = leftArray[i];
                    i++;
                }
                else
                {
                    array[k] = rightArray[j];
                    j++;
                }
                k++;
            }

            // Kopírování zbylých prvků z levého pole, pokud existují
            while (i < leftArray.Length)
            {
                array[k] = leftArray[i];
                i++;
                k++;
            }

            // Kopírování zbylých prvků z pravého pole, pokud existují
            while (j < rightArray.Length)
            {
                array[k] = rightArray[j];
                j++;
                k++;
            }
        }



       // Hlavní metoda
        public static void MergeSort(int[] array, int left, int right)
        {
            if (left < right)
            {
                // Střed pole
                int middle = left + (right - left) / 2;
                // Rekurze pro levou polovinu
                MergeSort(array, left, middle);
```

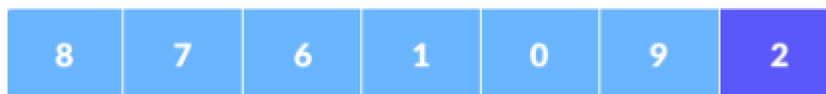
```
            // Rekurze pro pravou polovinu
            MergeSort(array, middle + 1, right);
            // Sloučení polovin
            Merge(array, left, middle, right);
        }
    }
```

1.3.5 Quick sort

Quick sort patří mezi populární a efektivní třídící algoritmy, který využívá princip, stejně jako Merge sort „rozděl a panuj“. Ke svému třídění používá pivota, který dělí pole na dvě části. Je často preferovaný pro velké datové sady, protože jeho průměrná časová složitost je ***O(n*** × ***log n)***.[57]

Výběr pivota

Vybere se pivot, který slouží k rozdělení pole na dvě části. Výběr pivotu může být náhodný nebo podle určité strategie, například: střední prvek, první prvek, medián několika prvků atd.



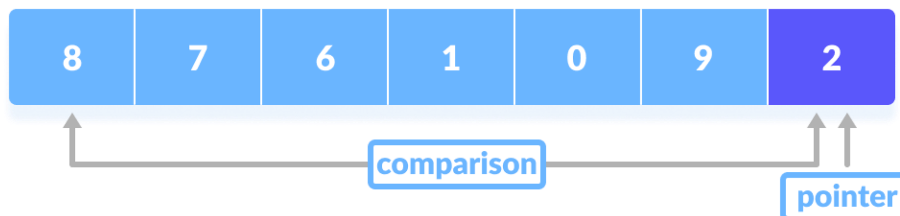
Obrázek 5- Quick sort [14]

Rozdělování

Pole se rozdělí na dvě části, kde do jedné části se umístí všechny prvky, které jsou menší než pivot a do druhé všechny prvky, které jsou větší nebo rovny pivotu.

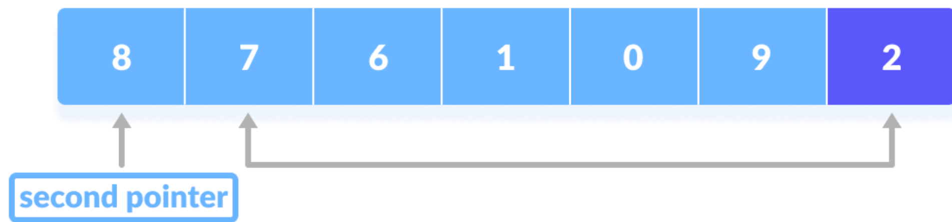
Postup při výběru prvku 2 jako pivota:

1) Ukazatel je fixně nastaven na pivot. A pivot je porovnán s prvním prvkem pole.



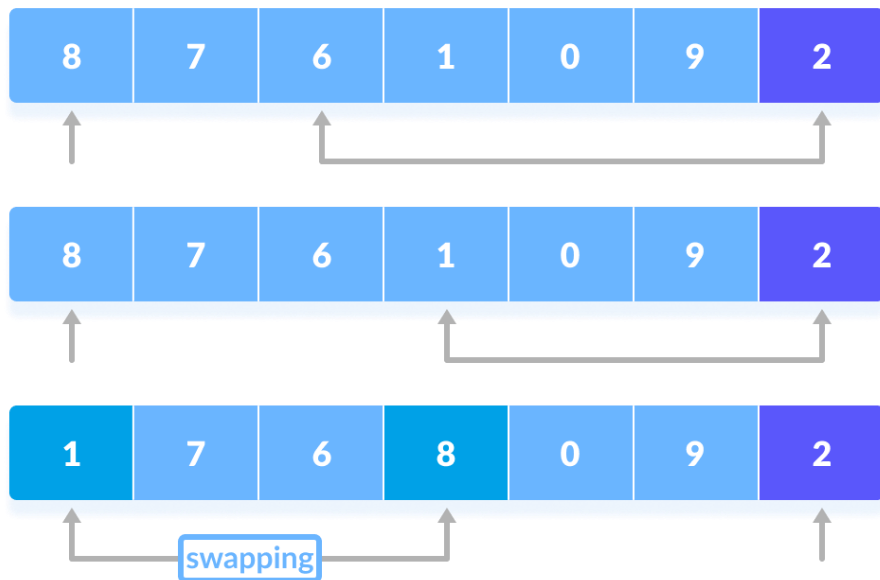
Obrázek 6 - Quick sort[15]

2) Pokud je prvek větší než pivot, tak na tento prvek nastavíme druhý ukazatel.



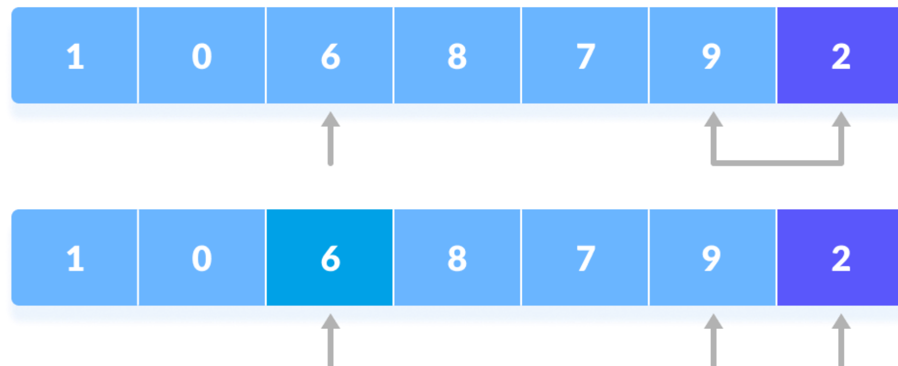
Obrázek 7 - Quick sort [16]

3) Dále je pivot porovnáván s dalšími prvky v poli, dokud není nalezen prvek menší než pivot. Poté se nově nalezený prvek menší, než pivot vymění s prvkem, na který ukazuje druhý ukazatel, který byl zvolen v kroku 2.



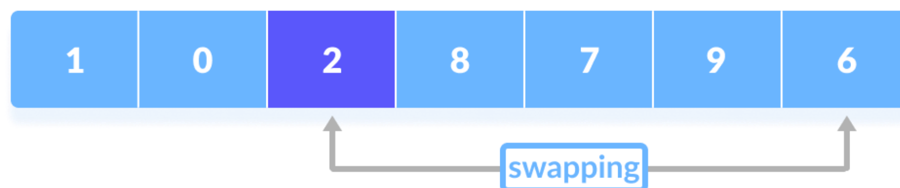
Obrázek 8 - Quick sort [17]

4) Tento proces se opakuje, dokud není dosáhnuto předposledního prvku.



Obrázek 9 - Quick sort [18]

5) Nakonec se pivot vymění s prvkem, na který ukazuje druhý ukazatel.



Obrázek 10 - Quick sort [19]

Rekurze

Rekurzivně se pak toto opakuje na obě části pole, tj. na část menšími prvky než pivot a na část s většími prvky než pivot.

Možná optimalizace

Možností, jak optimalizovat Quick sort je celá řada, jako například správný výběr pivotu, limit rekurze nebo paralelizace a třeba zastavení rekurze na určité úrovni.

Výběr pivotu

Pivot můžeme vybrat jako medián ze tří hodnot v poli (například první, prostřední a poslední prvek pole). Touto strategií můžeme zabránit nejhorším scénářům ve více případech.

Pivot můžeme vybrat náhodně, což pomáhá zajistit, že algoritmus má dobrou průměrnou časovou složitost bez ohledu na původní uspořádání prvků.[56]

Limit rekurze

Pro malé podpole můžeme použít jiný řadící algoritmus, jako je Insertion sort, který je efektivní pro malé datové sady.

Můžeme omezit hloubku rekurzivních volání, tím že nejdříve budeme třídit menší podpole a použijeme iteraci místo rekurze pro řazení delších podpolí.

Časová a prostorová složitost

Časová složitost v nejlepším a průměrným případě: $O(n \times \log n)$

Časová složitost v nejhorším případě: $O(n^2)$

Prostorová složitost: $O(\log n)$

Stabilní: **Ne**

Výhody a nevýhody

Výhodami Quick sortu můžeme označit jeho efektivitu $O(n \times \log n)$, což je efektivní pro velké datové sady. Další výhodou je, že nepotřebuje dodatečnou alokaci paměti, pouze zanedbatelné množství. Díky „rozděl a panuj" lze tento algoritmus snadno paralelizovat. Funguje dobře pro různé datové sady a lze ho optimalizovat pro různá vstupní data.

Naopak v nejhorším případě je jeho časová složitost $O(n^2)$, to nastává, když jsou špatně vybrány pivoty (např. když je pole již seřazeno, nebo jsou všechny prvky stejné atd.) Není to stabilní algoritmus a jeho intenzivní rekurze může vést k vysoké spotřebě zásobníku, což může způsobit přetečení zásobníku na systémech s omezeným množstvím paměti.[56]

Implementace v jazyce C#

```
public static void QuickSort(int[] array, int low, int high)
        {
            if (low < high)
            {
                // pi je indexace partition
                int pi = Partition(array, low, high);

                // Rekurzivní řazení prvků
                QuickSort(array, low, pi - 1);
                QuickSort(array, pi + 1, high);
            }
        }
        // Rozdělení pole na dvě části
        static int Partition(int[] array, int low, int high)
        {
            int pivot = array[high];
```

```
        // index menšího prvku
        int i = (low - 1);

        for (int j = low; j < high; j++)
        {
            if (array[j] <= pivot)
            {
                i++;
                // výměna
                Swap(array, i, j);
            }
        }

        // výměna array[i+1] a array[high] (neboli pivotu)
        Swap(array, i + 1, high);

        return i + 1;
    }
    // Výměna prvků v poli
    static void Swap(int[] array, int i, int j)
    {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

1.4 Pokročilé třídící algoritmy

1.4.1 Heap sort

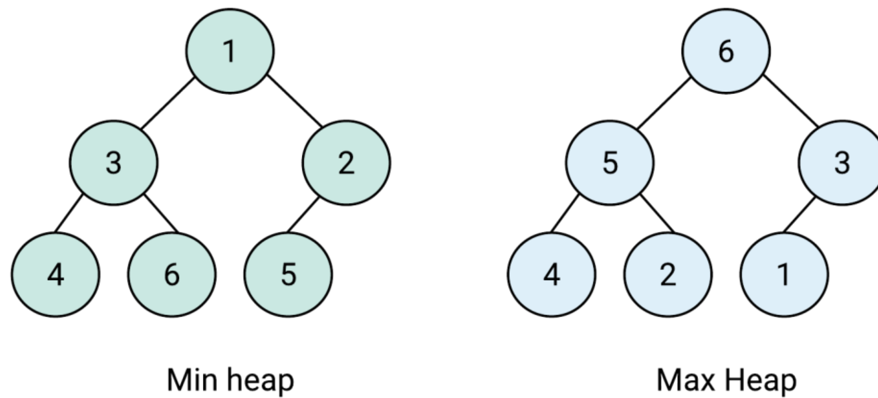
Algoritmus Heap sort, se řadí mezi výběrové algoritmy a využívá datovou strukturu tzv. haldu (heap) k seřazení prvků. Algoritmus Heap sort funguje, že nejdříve vytvoří haldu, po vytvoření haldy začne třídit. Toto opakuje, dokud v haldě nezůstane pouze jediný prvek.[58]

Max-halda (Max-heap)

Halda je speciální druh binárního stromu, který splňuje podmínku, každý uzel je větší nebo roven (max-heap) nebo menší nebo roven (min-heap) všem svým potomkům. Toto zajišťuje, že největší nebo nejmenší prvek se vždy nachází v kořeni stromu.

Binární strom

Binární strom je stromová struktura vizuálně připomínající strom, kde každý uzel má nejvýše dva potomky, kteří se označují jako levý a pravý potomek.



Obrázek 11 - Min, Max heap [20]

Vytvoření haldy

Vytváření haldy (heapify) je proces, kdy z neuspořádaného pole vytváříme strukturu haldy. Pro Heap sort se obvykle používá max-heap. Tento proces zahrnuje několik kroků:

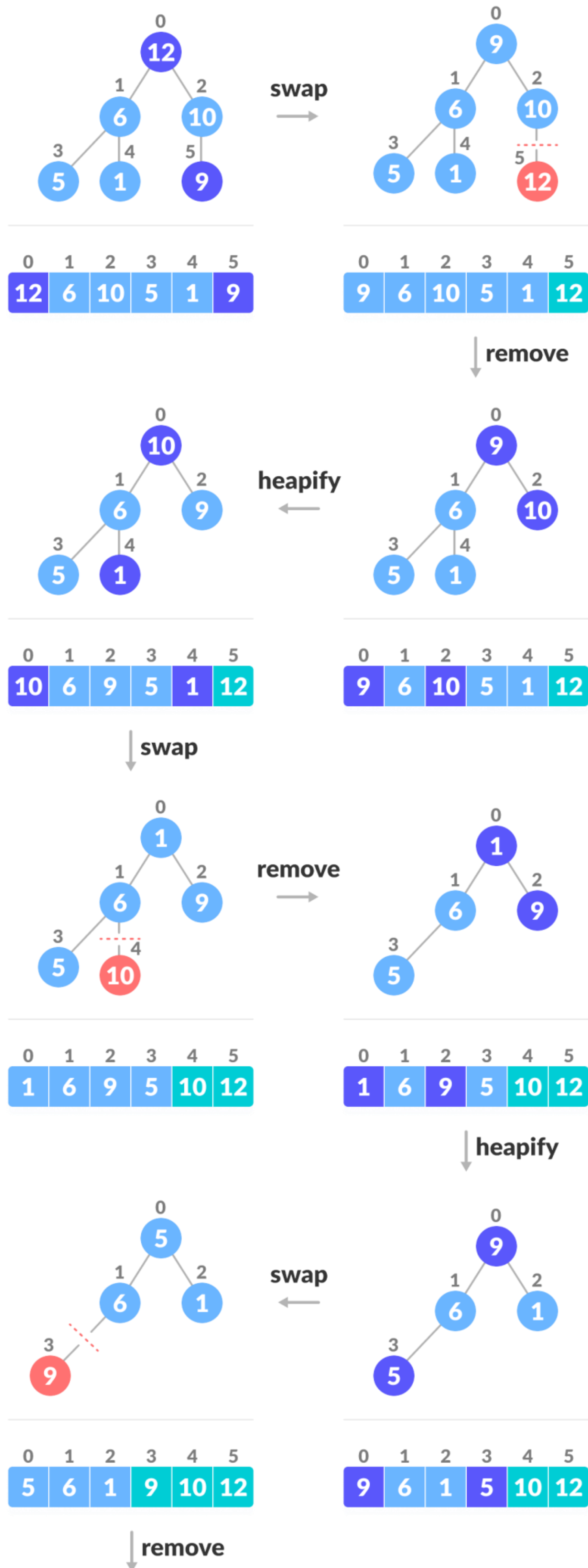
Inicializace: Neuspořádané pole budeme považovat za úplný binární strom. Indexování uzlů binárního stromu v poli začíná od 0 a pro každý uzel i, levý potomek je na indexu $2 \times i + 1$ a pravý potomek na indexu $2 \times i + 2$**.**

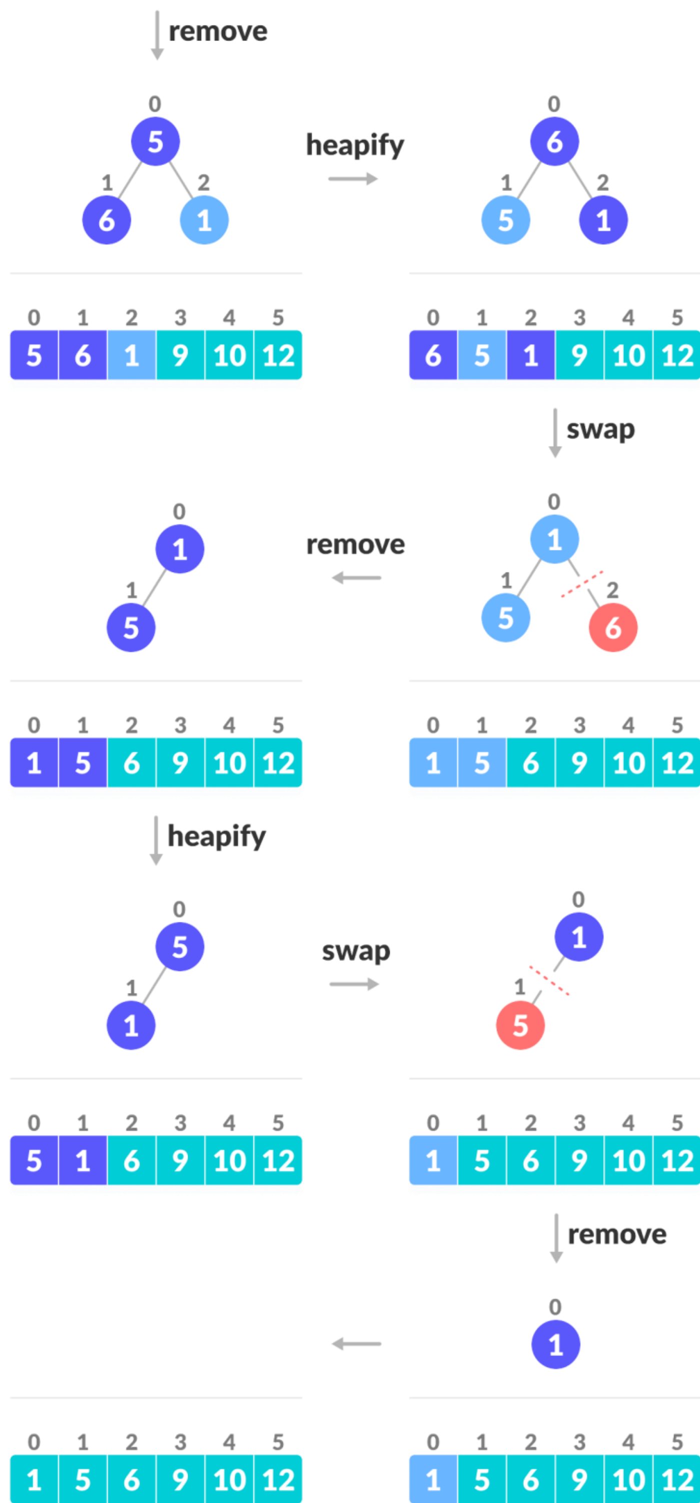
Heapify: Vytváření haldy začíná od posledního uzlu, který má potomky a pokračuje směrem nahoru ke kořeni stromu. Pro každý takový uzel provedeme operaci heapify, která zajišťuje, že daný uzel splňuje podmínku haldy. Pokud uzel nesplňuje podmínku tzn., že není větší než jeho potomci, vymění se s jeho největším potomkem. Toto se opakuje rekurzivně, dokud nejsou všechny uzly správně umístěny podle podmínky.[58]

Řazení pomocí Heap sort

Po vytvoření max-haldy je největší prvek v kořeni haldy. Tento prvek se vymění s posledním prvkem v poli a velikost haldy se sníží o jeden poslední prvek. Poté se opět provede na nově vzniklé haldě heapify, aby se zajistilo, že první prvek bude opět největší. Toto se opakuje, dokud nejsou všechny prvky seřazeny. Pole je postupně řazeno od konce směrem na začátek,

přičemž se po každém heapify zařídí, že největší prvek je přesunut na konečnou pozici. v seřazeném poli.[58]


swap
12 6 10 5 1 9
9 6 10 5 1 12
remove
heapify
10 6 9 5 1 12
9 6 10 5 1 12
swap
remove
1 6 9 5 10 12
1 6 9 5 10 12
heapify
swap
5 6 1 9 10 12
9 6 1 5 10 12
remove



Obrázek 12 - Heap sort [21]

Časová a prostorová složitost

Časová složitost v nejlepším, nejhorším, průměrným případě: $O(n \times \log n)$

Prostorová složitost: $O(1)$

Stabilní: **Ne**

Výhody a nevýhody

Výhodami Heapsortu je tedy časová složitost, která je pro všechny případy $O(n \times \log n)$. Zároveň je to In-place algoritmus, díky čemu nevyžaduje žádnou další alokaci paměti, a také není citlivý na vstupní data.

Na druhou stranu to není stabilní algoritmus, takže může změnit relativní pořadí stejných klíčových hodnot v seřazeném seznamu. Heapsort také není úplně nejjednodušší algoritmus, takže pro nezkušené programátory může být složitější na implementaci.[58]

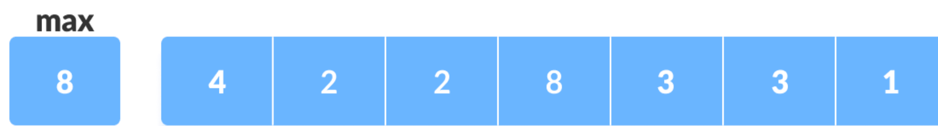
Implementace v jazyce C#

```
private static void Heapify(int[] array, int length, int i)
        {
            int largest = i; // Inicializace největšího jako roota
            int left = 2 * i + 1;
            int right = 2 * i + 2;

            // Levý potomek je větší než root
            if (left < length && array[left] > array[largest])
            {
                largest = left;
            }

            // Pravý potomek je větší než největší dosud
            if (right < length && array[right] > array[largest])
            {
                largest = right;
            }

            // Pokud největší není root
            if (largest != i)
            {
                int swap = array[i];
                array[i] = array[largest];
                array[largest] = swap;

                Heapify(array, length, largest);
            }
        }

        public static void HeapSort(int[] array)
        {
            int length = array.Length;

            // Vytvoření haldy
            for (int i = length / 2 - 1; i >= 0; i--)
```

```
            {
                Heapify(array, length, i);
            }

            for (int i = length - 1; i >= 0; i--)
            {
                // Přesunutí roota na konec
                int temp = array[0];
                array[0] = array[i];
                array[i] = temp;

                Heapify(array, i, 0);
            }
        }
```

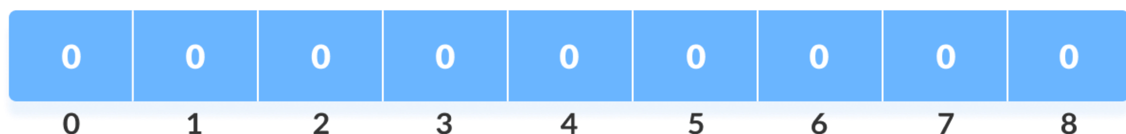
1.4.2 Counting sort

Counting sort je třídící algoritmus, který nepracuje na principu porovnávání jednotlivých hodnot, ale na výčtu jejich výskytů. Je používán pro třídění malých nebo omezených rozsahů celočíselných klíčů a často se používá jako pomocný algoritmus pro složitější třídící algoritmy, jako je například Radix sort.[59]



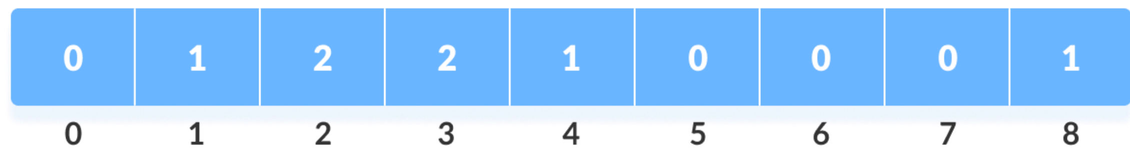
Obrázek 13 - Counting sort [22]

Algoritmus začne vytvořením pomocného pole, často se nazývá „counting array“, kde indexy v tomto poli odpovídají možným hodnotám ve vstupním poli a inicializujeme ho na velikost *max + 1,* kde *max* je největší hodnota ve vstupním poli. Každý prvek v počítacím poli se inicializuje na hodnotu 0.



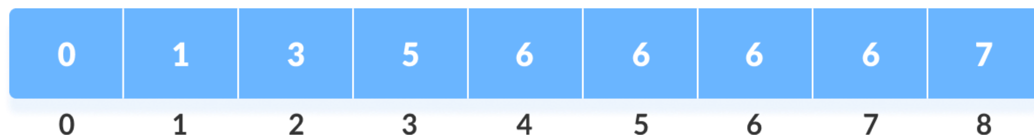
Obrázek 14 - Counting sort [23]

Poté pro každý prvek vstupního pole se zvýší hodnota v „counting array“ o 1, čímž se zjistí počet výskytů každé hodnoty ve vstupním poli.



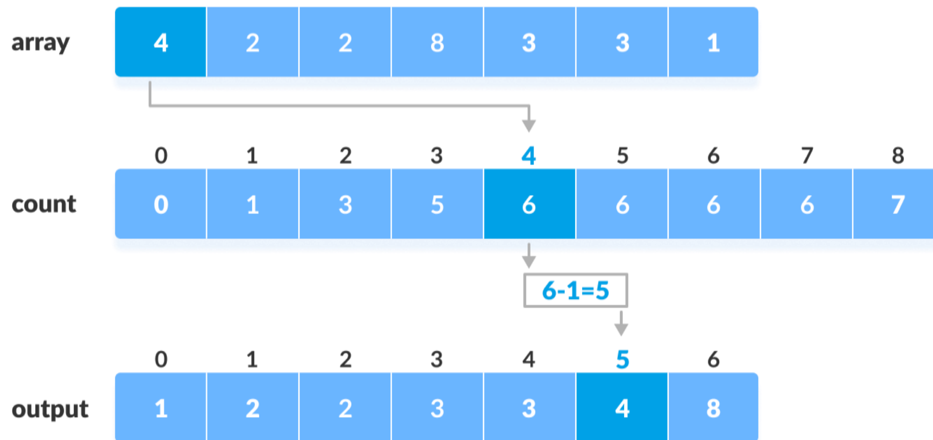
Obrázek 15 - Counting sort [24]

Následně se „counting array“ upraví tak, že každá jeho hodnota bude součtem hodnoty původní a hodnot předchozích prvků. Toto pole nám bude ukazovat, kolik prvků se vyskytovat před každým prvkem v seřazeném poli.



Obrázek 16 - Counting sort [25]

Toto pole je pak použito k určení přesné pozice každého prvku v seřazeném poli. Pro každý prvek v původním poli je zjištěna jeho hodnota, která je použita jako index do kumulativního pole. Prvek je poté umístěn na pozici určenou touto hodnotou v kumulativním poli minus jedna. Po umístění prvku na správné místo je hodnota v kumulativním poli dekrementována o jedna. Tento postup se opakuje pro každý prvek v původním poli. Vzhledem k tomu, že se hodnoty v kumulativním poli postupně snižují s každým správným umístěním prvku do seřazeného pole, tak bude zajištěno, že prvky se stejnou hodnotou budou umístěny ve správném pořadí, v jakém byly v původním poli.[59]



Obrázek 17 - Counting sort [26]

Časová a prostorová složitost

Časová složitost v nejlepším, nejhorším a průměrném případě: ***O(n+k)***

Prostorová složitost: ***O(max)***

Stabilní: **Ano**

Výhody a nevýhody

Výhodou Counting sortu je, že díky své lineární složitosti je extrémně rychlý pro malé rozsahy hodnot. Dále je stabilní, takže zachovává relativní pořadí stejných prvků a je předvídatelný a nezávisí na pořadí vstupních dat.

Efektivně umožňuje třídit pouze celočíselné hodnoty, nebo hodnoty, které lze převést na celá čísla. Pro velké rozsahy hodnot *k*, může být velmi náročný na paměť, což omezuje jeho použitelnost.

Implementace v jazyce C#

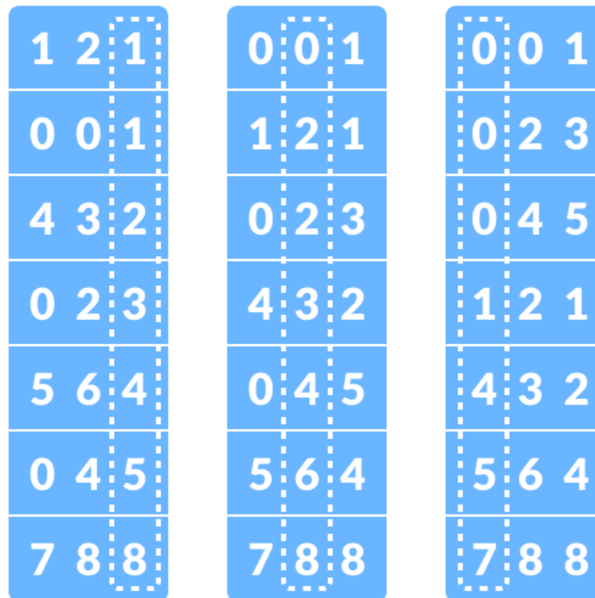
```
public static void CountingSort(int[] array)
        {
            int n = array.Length;

            // Nalezení maxima pro zjištění velikosti pole count
            int max = array[0];
            for (int i = 1; i < n; i++)
            {
                if (array[i] > max)
                    max = array[i];
            }

            int[] count = new int[max + 1];
            int[] output = new int[n];
```

```
        // Sčítání výskytů každého čísla
        for (int i = 0; i < n; i++)
        {
            count[array[i]]++;
        }

        // Akumulace počtů
        for (int i = 1; i <= max; i++)
        {
            count[i] += count[i - 1];
        }

        // Výstupní pole
        for (int i = n - 1; i >= 0; i--)
        {
            output[count[array[i]] - 1] = array[i];
            count[array[i]]--;
        }

        //  Zkopírování výstupního pole do původního
        for (int i = 0; i < n; i++)
        {
            array[i] = output[i];
        }
    }
```

1.4.3 Radix sort

Radix sort je stejně, jako Counting sort nekomparativní algoritmus, který třídí prvky s pevnou délkou. Rozděluje prvky podle každé cifry zleva doprava (MSD) nebo naopak (LSD). Neporovnává prvky, ale zaměřuje se na individuální číslice nebo části prvků. Často ke svému třídění používá další stabilní třídící algoritmus.

Algoritmus začne třídit prvky podle nejméně významné cifry (LSD) nebo nejvíce významné (MSD). Je potřeba získat z každého prvku relevantní cifru (jednotky, desítky, stovky atd.). Toho dosáhne obvykle pomocí matematických operací dělení a modulo. Pro získání jednotek se použije operace modulo 10 (číslo*%10),* pro získání desítek se číslo nejdříve vydělí 10 a poté se použije modulo 10 (*(číslo/10)%10*), pro získání stovek se postupuje obdobně, jako u desítek, nejdříve se vydělí číslo stem, a poté se použije modulo 10, a tak dále pro vyšší řády. Pro seřazení číslic každého řádu prvku se může použít například algoritmus Counting sort. Ten bude prvky třídit na základě získaných cifer. Po každém třídění na základě cifer umístí Radix sort prvky zpět do původního nebo dočasného pole. Proces se opakuje, a po každém třídění jsou prvky přesněji seřazeny, dokud po konečném třídění (na základě nejvýznamnějšího řádu) nejsou seřazeny úplně.[61][60]



Obrázek 18 - Radix sort [27]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $\boldsymbol{O(n+k)}$

Časová složitost v průměrném případě: $\boldsymbol{O(nk)}$

Časová složitost v nejhorším případě: $\boldsymbol{O(nk)}$

Prostorová složitost: $\boldsymbol{O(max)}$

Stabilní: **Ano**

Výhody a nevýhody

Mezi hlavní výhody Radix sortu patří díky jeho časové složitosti efektivita při třídění velkého množství dat. Na rozdíl od jiných třídicích algoritmů, jejíž výkon závisí na vstupních datech, tak je Radix sort konzistentní ve výkonu, který se dá snadno předvídat na základě velikosti vstupního pole a rozsahu hodnot. Nadále je to stabilní nekomparativní algoritmus.

K omezením tohoto třídícího algoritmu patří, že je omezen na určité typy dat. Především je vhodný na celá čísla nebo jiné data, která lze rozdělit na cifry nebo segmenty. Pro složitější nebo nečíselná data může být jeho použití nevhodné a neefektivní. Dále potřebuje pro svou práci dodatečný prostor pro dočasné uložení dat a pomocné struktury. V porovnání s jednoduššími třídícími algoritmy může být jeho implementace složitější, obzvláště když zahrnuje optimalizace pro vyšší výkon.[60]

Implementace v jazyce C#

```
public static void RadixSort(int[] arr)
        {
            // Nalezení maxima k určení počtu průchodu
            int max = GetMax(arr);

            // Pro každý exponent odvozený z 10
            for (int exp = 1; max / exp > 0; exp *= 10)
            {
                CountSort(arr, exp);
            }
        }
        private static int GetMax(int[] arr)
        {
            int max = arr[0];
            for (int i = 1; i < arr.Length; i++)
                if (arr[i] > max)
                    max = arr[i];
            return max;
        }
        private static void CountSort(int[] arr, int exp)
        {
            int n = arr.Length;
            int[] output = new int[n]; // Výstupní pole
            int[] count = new int[10]; // Početní pole

            // Naplnění početního pole
            for (int i = 0; i < n; i++)
                count[(arr[i] / exp) % 10]++;

            // Akumulace počtů
            for (int i = 1; i < 10; i++)
                count[i] += count[i - 1];

            // Výstupní pole
            for (int i = n - 1; i >= 0; i--)
            {
                output[count[(arr[i] / exp) % 10] - 1] = arr[i];
                count[(arr[i] / exp) % 10]--;
            }

            // Zkopírování výstupního pole do původního
            for (int i = 0; i < n; i++)
                arr[i] = output[i];
        }
```

1.4.4 Bucket sort

Bucket sort patří mezi stabilní a nekomparativní třídící algoritmy, který vstupní pole rozdělí do tzv. „bucketů“ – proto nese název Bucket sort. Algoritmus nejdříve vstupní data rozdělí do několika bucketů na základě jejich hodnot a poté tyto buckety samostatně seřadí. K seřazení prvků uvnitř těchto bucketů se obvykle používá jiný třídící algoritmus v závislosti na konkrétní implementaci.

Algoritmus nejdříve vytvoří pevný počet bucketů s daným rozsahem. Počet a rozsah bucketů závisí na rozsahu vstupních dat. Takovéto buckety jsou reprezentovány buď jako seznamy nebo pole. Algoritmus pak rozřadí prvky vstupního pole do jednotlivých bucketů na základě jejich hodnoty. Toto rozdělení se provádí pomocí specifické funkce, která mapuje hodnoty prvků na buckety. Po rozdělení všech prvků do bucketů se prvky v každém bucketu seřadí. Seřazení prvků v každém bucketu může být provedeno pomocí jiného třídícího algoritmu, jako je například Insertion sort. Po seřazení prvků v každém bucketu jsou všechny prvky zpět spojeny do jednoho pole. To algoritmus provádí iterací přes buckety a přidáváním prvků každého bucketu do výsledého pole v pořadí, ve kterém byly buckety vytvořeny.[62][63]

Obrázek 19 - Bucket sort [28]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $\boldsymbol{O(n+k)}$

Časová složitost v průměrném případě: $\boldsymbol{O(n)}$

Časová složitost v nejhorším případě: $\boldsymbol{O(n^2)}$

Prostorová složitost: $\boldsymbol{O(n+k)}$

Stabilní: **Ano**

Výhody a nevýhody

Bucket sort může být velmi rychlý pro vhodně rozdělená data díky. Zároveň je flexibilní, protože může být efektivní na široký rozsah dat díky možnosti přizpůsobit velikost bucketů a jejich počet.

Na druhou stranu nerovnoměrně rozložená vstupní data mohou mít za následek špatný třídící výkon. Dále vyžaduje dodatečnou paměť pro buckety. Pokud je rozsah vstupních velký, bude vyžadováno velké množství dodatečné paměti. Zároveň je algoritmus primárně navržen k třídění celočíselných dat nebo čísel s desetinnou čárkou a nemusí být použitelný pro třídění jiných typů dat.[62]

Implementace v jazyce C#

```
public static void Sort(int[] array)
        {
            if (array.Length <= 1)
                return;

            // Určení počtu bucketů
            int maxValue = array[0];
            for (int i = 1; i < array.Length; i++)
            {
                if (array[i] > maxValue)
                    maxValue = array[i];
            }

            // Vytvoření bucketů
            List<int>[] buckets = new List<int>[maxValue / 10 + 1];
            for (int i = 0; i < buckets.Length; i++)
            {
                buckets[i] = new List<int>();
            }

            // Rozdělení prvků do bucketů
            foreach (var item in array)
            {
                int bucketIndex = item / 10;
                buckets[bucketIndex].Add(item);
            }

            // Seřazení prvků v každém bucketu
            int position = 0;
            foreach (var bucket in buckets)
            {
                bucket.Sort();

                // Přidání seřazených prvků zpět
                foreach (var item in bucket)
                {
                    array[position] = item;
                    position++;
                }
            }
        }
```

1.4.5 Shell sort

Shell sort (řazení se snižujícím se přírůstkem) je zlepšení třídícího algoritmu Insertion sort. Narozdíl od Insertion sortu, který porovnává sousední prvky a postupně je přesouvá na správné místo, Shell sort umožňuje vyměnit prvky, které nejsou vedle sebe, ale jsou od sebe vzdáleny o určitý interval. Tento interval se nazývá „gap“ a v průběhu algoritmu se postupně zmenšuje, obvykle na polovinu. Tímto umožňuje Shell sort efektivně posouvat prvky blíže k jejich konečné pozici předtím, než dosáhne gap hodnoty 1 a algoritmus se „přepne“ na Insertion sort.[64]

Sekvence intervalů (gap)

Výběr tohoto počátečního intervalu a způsobu, jakým se tento interval bude zmenšovat, má zásadní vliv na výkon Shell sortu. Neexistuje ovšem nejlepší sekvence pro všechny případy, protože optimální sekvence může záviset na velikosti a distribuci dat. Sekvence by měla postupně zmenšovat intervaly tak, aby byl poslední interval velikosti 1 a měla by minimalizovat vytváření špatných vzorů v poli.

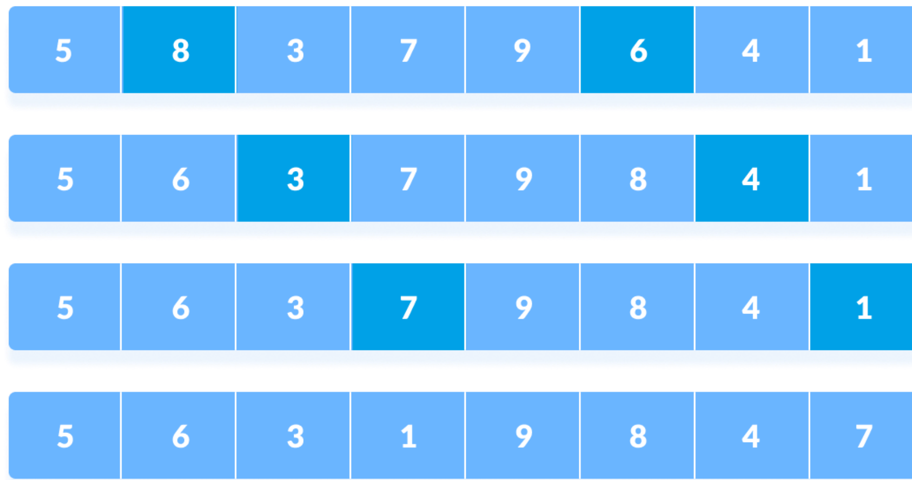
Optimální sekvence

Původní sekvence od Shella: Interval je zmenšován přibližně na polovinu předchozí hodnoty (n/2), kde *n* je velikost pole.[64]

Hibbardova sekvence: Používá se sekvence $2^k - 1$, kde *k* je celé číslo, které začíná od 1 a postupně se zvyšuje, určuje tedy krok v sekvenci. [65]

Knuthova sekvence: Používá se sekvence $(3^k – 1) / 2$, kde *k* je celé číslo, které začíná od 1 a postupně se zvyšuje.[65]

Algoritmus začíná výběrem vhodného intervalu (gapu), obvykle polovina délky pole. Pro každý interval se prochází pole a porovnávají se prvky, které jsou od sebe vzdáleny o tento interval. Pokud je prvek menší pozice větší než prvek větší pozice (pro vzestupné řazení), tak se prvky vymění. Po každém průchodu se interval zmenší a proces se opakuje do té doby, dokud není gap 1. Po dosažení gapu 1 začne Shell sort fungovat jako Insertion sort, ale jelikož je pole z přechozích průchodů přetříděno, je to mnohem rychlejší. Insertion sort zajistí, že se pole kompletně seřadí.



Obrázek 20 - Shell sort [29]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $O(n \times \log n)$

Časová složitost v průměrném případě: $O(n \times \log n)$

Časová složitost v nejhorším případě: $O(n^2)$

Prostorová složitost: $O(1)$

Stabilní: **Ne**

Výhody a nevýhody

Shell sort je obecně efektivní pro středně velká data a díky použití intervalů pro porovnávání vzdálenějších prvků dokáže rychleji přemístit prvky na správné pozice, což znamená, že celkově vyžaduje méně porovnání a výměn než Insertion sort. Zároveň nepotřebuje dodatečnou paměť kromě malého konstantního množství, protože je in-place. A přestože je složitější než Insertion sort, tak stále zůstává relativně jednoduchý pro implementaci.

Naopak jeho výkon závisí na výběru sekvence intervalů, protože špatně zvolená sekvence může vést k podstatnému snížení výkonu. Shell sort zároveň není stabilní algoritmus.

Implementace v jazyce C#

```
public static void ShellSort(int[] array)
        {
            int n = array.Length;
            // Originální Shellova sekvence n/2
            for (int gap = n / 2; gap > 0; gap /= 2)
            {
                // Pro každý krok se seřadí části pole
                for (int i = gap; i < n; i += 1)
                {
                    // Uložení aktuálního prvku a jeho indexu
                    int temp = array[i];
                    int j;
                    // Posouvání prvků, které jsou větší o jeden krok dozadu,
                    // dokud nejsou správně
                    for (j = i; j >= gap && array[j - gap] > temp; j -= gap)
                        array[j] = array[j - gap];

                    array[j] = temp;
                }
            }
        }
```

1.4.6 Comb sort

Comb sort je algoritmus, který byl navržen jako vylepšení Bubble sortu. Základní myšlenkou je eliminovat těžké tzv. „želví“ prvky na začátku pole. Takovéto prvky se pohybují velmi pomalu na konec pole. Tyto prvky se eliminují zavedením snižujícího přírůstku. Stejně jako u Shell sortu se interval každým krokem postupně snižuje dokud není 1. Po dosažení intervalu 1 se „stane“ Comb sort Bubble sortem, a v tomto okamžiku by mělo být pole již téměř seřazené. Stejně jako u Shell sortu je důležité vybrat správný interval mezi prvky.

Algoritmus na začátku inicializuje počáteční interval, který je typicky nastaven na délku tříděného pole. Často se používá empiricky zjištěný faktor zmenšení 4/3, pro postupné zmenšování intervalu. Algoritmus dále porovnává prvky, které jsou od sebe vzdáleny o určený gap (interval). Pokud je prvek menší pozice větší než prvek větší pozice, tak se prvky prohodí. Toto se opakuje pro celé pole, dokud prvek na větší pozici nedosáhne konce pole. Po takovémto průchodu polem je aktuální interval zmenšen faktorem zmenšení (4/3) a zaokrouhlen. Tento celý proces se opakuje, dokud není dosaženo hodnoty intervalu 1. Poté se stane Comb sort Bubble sortem, a jelikož bylo pole v předchozích krocích již téměř přetříděno, tak vyžaduje méně průchodů pro dokončení.[66]

	0	1	2	3	4	5	6	7	
i = 0	49	11	24	44	29	27	2	22	49>2, swap
i = 1	2	11	24	44	29	27	49	22	11<22, no swap

Obrázek 21 - Comb sort [30]

	0	1	2	3	4	5	6	7	
i = 0	2	11	24	44	29	27	49	22	2<29, No swap
i = 1	2	11	24	44	29	27	49	22	11<27, No swap
i = 2	2	11	24	44	29	27	49	22	24<49, No swap
i = 3	2	11	24	44	29	27	49	22	44>22, swap

Obrázek 22 - Comb sort [31]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $O(n \times \log n)$

Časová složitost v průměrném případě: $O(n^2 / 2^p)$, kde p je počet inkrementací

Časová složitost v nejhorším případě: $O(n^2)$

Prostorová složitost: $O(1)$

Stabilní: **Ano**

Výhody a nevýhody

Hlavní předností Comb sortu je, že eliminuje těžké prvky, které jsou daleko od svého cílového umístění. Tímto dosahuje obecně lepšího výkonu než Bubble sort. Dále je poměrné snadný na implementaci, nevyžaduje žádnou dodatečnou paměť a ukazuje se často jako efektivní při řazení velkých datasetů díky své schopnosti rychle zpracovat data s využitím větších gapů na začátku třídění.

Jeho výkon je také závislý na zmenšování gapu pro specifické datasety a nemusí být vhodný pro rozmanité datasety.

Implementace v jazyce C#

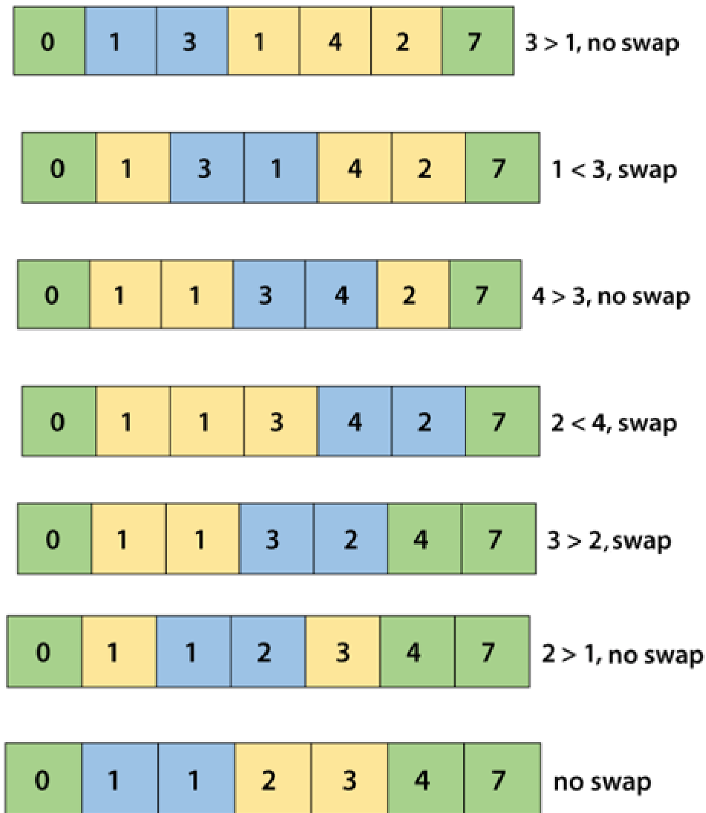
```
static int GetGap(int gap)
        {
            gap = (gap * 10) / 13;
            if (gap < 1)
            {
                return 1;
            }
            return gap;
        }
        public static void Sort(int[] arr)
        {
            int n = arr.Length;
            int gap = n;
            bool swapped = true;

            // Dokud gap není 1 nebo nedošlo k výměně
            while (gap != 1 || swapped == true)
            {
                // Nalezení další mezery
                gap = GetGap(gap);
                swapped = false;

                // Porovnání prvků, které jsou od sebe o gap
                for (int i = 0; i < n - gap; i++)
                {
                    if (arr[i] > arr[i + gap])
                    {
                        // Výměna prvků
                        int temp = arr[i];
                        arr[i] = arr[i + gap];
                        arr[i + gap] = temp;

                        swapped = true;
                    }
                }
            }
        }
```

1.4.7 Cocktail sort

Cocktail sort nebo také Bidirectional Bubble sort a Shaker sort, je variace na běžný Bubble sort. Zatímco Bubble sort prochází pole od začátku do konce a postupně řadí největší prvek na konec pole tím, že prohazuje sousední prvky, tak Cocktail sort prochází pole oběma směry. Tímto obousměrným procházením je Cocktail sort schopen redukovat počet iterací k seřazení pole.

Algoritmus začne porovnávat od začátku pole každý sousední pár prvků, a pokud jsou prvky v nesprávném pořadí (první je větší než druhý v případě vzestupného řazení), tak se prvky prohodí. Toto se opakuje, dokud se algoritmus nedostane na konec pole. Během tohoto je na konec vždy vypuštěn prvek, který je největší. Po dokončení prvního průchodu se algoritmus obrátí a začne pole procházet od konce zpět na začátek. Opět se budou prohazovat prvky, které

jsou v nesprávném pořadí (pokud prvek, který je blíže začátku pole, je větší - pro vzestupné řazení). Tento průchod zpět je schopen vypouštět menší prvky na začátek pole. Každým průchodem se sníží počet prvků, které je třeba v dalších průchodech pole porovnat. Celý proces se opakuje, dokud není celé pole seřazené.[67]



Obrázek 23 - Cocktail sort [32][33][34][35][36][37][38]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $O(n)$

Časová složitost v průměrném případě: $O(n^2)$

Časová složitost v nejhorším případě: $O(n^2)$

Prostorová složitost: $O(1)$

Stabilní: **Ano**

Výhody a nevýhody

Výhodou Cocktail sortu je, že je efektivnější než klasický Bubble sort v případech, kdy je potřeba minimalizovat těžké prvky blízko konci pole. Je jednoduchý a funguje dobře na malé

nebo částečně seřazené pole a zároveň je stabilním algoritmem a nevyžaduje dodatečnou paměť.

Podobně jako Bubble sort, Cocktail sort není příliš vhodný pro velké pole kvůli své složitosti ***O(n²)*** v nejhorším případě.

Implementace v jazyce C#

```
public static void CocktailSort(int[] array)
        {
            bool swapped = true;
            int start = 0;
            int end = array.Length;

            while (swapped)
            {
                swapped = false;

                // Procházení pole od začátku do konce
                for (int i = start; i < end - 1; ++i)
                {
                    if (array[i] > array[i + 1])
                    {
                        int temp = array[i];
                        array[i] = array[i + 1];
                        array[i + 1] = temp;
                        swapped = true;
                    }
                }

                // Seřazené pole
                if (!swapped)
                    break;

                // Pro další průchod pole
                swapped = false;

                // Poslední prvek je již na správném místě
                end--;

                // Procházení pole od konce k začátku
                for (int i = end - 1; i >= start; i--)
                {
                    if (array[i] > array[i + 1])
                    {
                        int temp = array[i];
                        array[i] = array[i + 1];
                        array[i + 1] = temp;
                        swapped = true;
                    }
                }

                // První prvek je již na správném místě
                start++;
            }
        }
```

1.4.8 Timsort

Timsort je třídící algoritmus, který efektivně kombinuje vlastnosti Insertion sortu a Merge sortu. Byl navržen tak, aby optimálně pracoval s různými typy reálných dat, které často již obsahují seřazené sekce. Od verze 2.3 je standardním třídícím algoritmem Pythonu. [68]

Algoritmus začíná procházením pole a hledáním přirozeně seřazených sekvencí tzv. „runs". Tyto sekvence mohou být sestupné nebo vzestupné, přičemž sestupné sekvence budou převráceny, aby tvořily vzestupné. Pokud je ale délka nalezené sekvence menší než minimální limit tzv. „minrun", použije algoritmus Insertion sort k rozšíření této sekvence na minimální požadovanou délku. Hodnota minrunu je vypočítána z délky pole tak, aby se optimalizoval počet a délka sekvencí a zároveň, že jich bude méně, než je stanovený limit.[68]

Timsort identifikuje runs jako sekci, kde jsou prvky seřazeny vzestupně, kde je každý následující prvek větší nebo roven předchozímu. Pro sestupně seřazené prvky algoritmus takovou sekvenci invertuje, aby se stala vzestupnou.

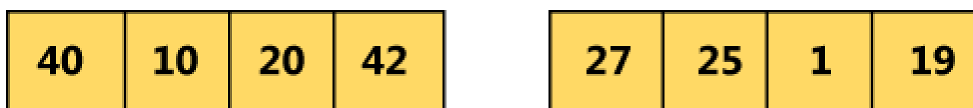
Velikost minrun je dynamicky vypočítána podle velikosti vstupního pole, tak aby byl vybalancován výkon Insertion sortu na malých datech a efektivitou Merge sortu při slučování větších bloků. Obvykle je minrun vybrán tak, aby délka pole *n* byla rozdělena na co nejmenší počet minrun tzn. *n/minrun* je co nejmenší, ale zároveň minrun je blízko své maximální hodnotě. Obvykle chceme, aby minrun bylo mocninou 2, protože to zjednodušuje logiku slučování díky rovnoměrnému rozdělení.[69]

Dále algoritmus k samotnému slučování runs používá Merge sort, který kombinuje dvě nebo více seřazených sekvencí do jednoho seřazeného celku. Timsort má zásobník (stack) pro runs, které jsou čekající na sloučení. Vždy když je přidán nový run na zásobník, tak se kontroluje, zda nejsou splněna pravidla pro slučování. Pokud ano jsou vybrány vhodné runs ze zásobníku ke sloučení. Pokud jsou tyto pravidla porušeny, tak algoritmus vybere dva nejmenší runs ze zásobníku a sloučí je.[69]

40	10	20	42	27	25	1	19

Obrázek 24 - Tim sort [39]

Za předpokladu, že run má velikost 4, rozdělíme vstupní pole na dva runs.

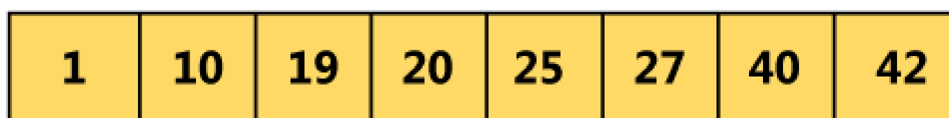


Obrázek 25 - Tim sort [40]

Každé podpole, poté setřídíme pomocí Insertion sortu.

Obrázek 26 - Tim sort [41]

Pomocí Merge sortu obě podpole zpět spojíme.



Obrázek 27 - Tim sort [42]

Časová a prostorová složitost

Časová složitost v nejlepším případě: ***O(n)***

Časová složitost v průměrném případě: ***O(n* × *log n)***

Časová složitost v nejhorším případě: ***O(n* × *log n)***

Prostorová složitost: ***O(n)***

Stabilní: **Ano**

Výhody a nevýhody

Timsort je velmi efektivní na datech, které již obsahují částečně seřazené sekvence. Díky časové složitosti ***O(n* × *log n)*** v průměrném případě a v nejlepším ***O(n)*** je velmi rychlý. Zároveň díky jeho efektivitě je použit v několika programovacích jazycích jako výchozí třídící algoritmus.

Na druhou stranu je jeho implementace složitější než mnoho základních třídících algoritmů, jako jsou Insertion sort nebo Bubble sort, což může být nevýhodné pro nové programátory. Také vyžaduje pomocnou paměť pro slučování runs, to může být nevýhodou v prostředí, kde je omezené množství paměti.

Implementace v jazyce C#

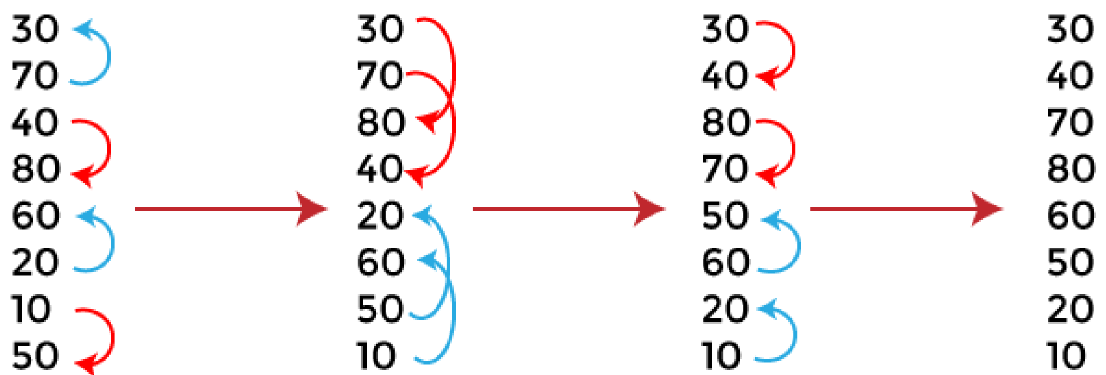
```
public class Timsort
    {
        private const int RUN = 32;

        // Slučování runs
        public static void Merge(int[] array, int l, int m, int r)
        {
            // Velikost dvou polí pro sloučení
            int len1 = m - l + 1, len2 = r - m;
            int[] left = new int[len1];
            int[] right = new int[len2];

            Array.Copy(array, l, left, 0, len1);
            Array.Copy(array, m + 1, right, 0, len2);

            int i = 0;
            int j = 0;
            int k = l;

            // Sloučení polí do array
            while (i < len1 && j < len2)
            {
                if (left[i] <= right[j])
                {
                    array[k] = left[i];
                    i++;
                }
                else
                {
                    array[k] = right[j];
                    j++;
                }
                k++;
            }

            // Zkopírování zbývajících prvků z left[]
            while (i < len1)
            {
                array[k] = left[i];
                k++;
                i++;
            }

            // Zkopírování zbývajících prvků z right[]
            while (j < len2)
            {
                array[k] = right[j];
                k++;
                j++;
            }
        }

        // Seřazení jednotlivého run
        public static void InsertionSort(int[] array, int left, int right)
        {
            for (int i = left + 1; i <= right; i++)
            {
                int temp = array[i];
                int j = i - 1;
                while (j >= left && array[j] > temp)
```

```
                {
                    array[j + 1] = array[j];
                    j--;
                }
                array[j + 1] = temp;
            }
        }

        public static void TimSort(int[] array, int n)
        {
            // Seřazení runs
            for (int i = 0; i < n; i += RUN)
                InsertionSort(array, i, Math.Min((i + RUN - 1), (n - 1)));

            // Sloučení runs do seřazeného pole
            for (int size = RUN; size < n; size = 2 * size)
            {
                for (int left = 0; left < n; left += 2 * size)
                {
                    int mid = left + size - 1;
                    int right = Math.Min((left + 2 * size - 1), (n - 1));

                    if (mid < right)
                        Merge(array, left, mid, right);
                }
            }
        }
    }
```

1.4.9 Bitonic sort

Bitonic sort je komparativní algoritmus, který je speciálně navržen pro paralelní zpracování a je založen na principu bitonických sekvencích. Algoritmus je vhodný pro hardwarové implementace a systémy, kde může plně využít svůj paralelní potenciál.

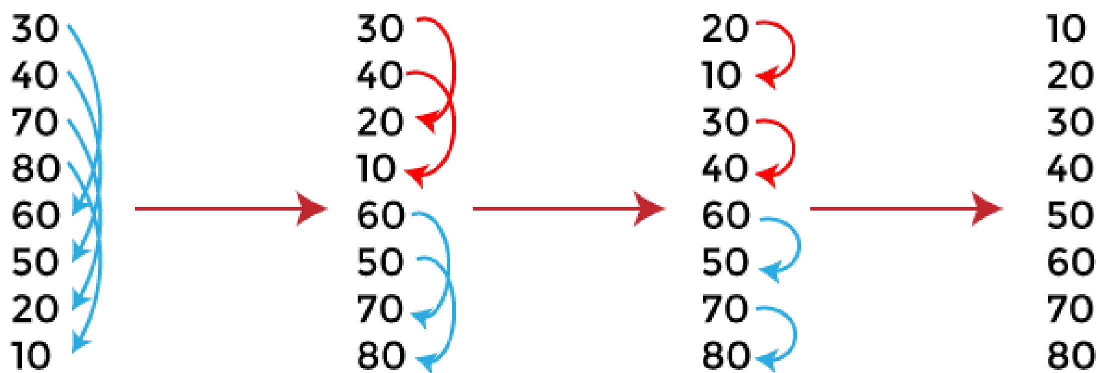
Bitonická sekvence je sekvence čísel, která nejprve stoupá a poté klesá, nebo naopak (M[0] < M[1] … M[i-1] < M[i] > M[i+1] > M[i+2] … M[n-1], kde 0 <= i <= n-1). [70]



Obrázek 28 - Bitonic sort [43]

Princip fungování

Algoritmus nejdříve rozdělí a uspořádá vstupní pole tak, aby tvořilo bitonické sekvence. Toho docílí tak, že pole rozdělí na menší segmenty o pevně stanovenou velikost (obvykle mocnina dvou). Každý segment poté uspořádá tak, že buď stoupá nebo klesá. To je zařízeno aplikací například Insertion sortu nebo jiného třídícího algoritmu. Poté algoritmus přistoupí k procesu bitonic merge. Tento proces porovnává a prohazuje symetrické prvky ve vzestupném a sestupném segmentu, které jsou vedle sebe. Každá dvojice segmentů se nejprve porovná prvek po prvku. Prvky na odpovídajících pozicích v obou segmentech se porovnávají, a případně prohozeny tak, aby byl první segment ve vzestupném pořadí a druhý v sestupném. Tímto jsou oba segmenty spojeny do větší bitonické sekvence. Tento proces se opakuje s dvojnásobnými velikostmi sekvencí, až dokud nejsou všechny bitonické sekvence sloučeny do jedné velké sekvence. Tato sekvence je poté připravená na poslední bitonic merge. Po posledním bitonic merge je pole zcela seřazeno.[70]



Obrázek 29 - Bitonic sort [44]

Časová a prostorová složitost

Časová složitost v nejlepším případě: $O(\log^2 n)$

Časová složitost v průměrném případě: $O(\log^2 n)$

Časová složitost v nejhorším případě: $O(\log^2 n)$

Prostorová složitost: $O(n \times \log^2 n)$

Stabilní: **Ne**

Výhody a nevýhody

Jednou z největších výhod Bitonic sortu je jeho schopnost využívat paralelní hardware, díky své struktuře, která dovoluje nezávislé porovnávání prvků. Díky časové složitosti ve všech případech $O(\log^2 n)$ je předvídatelný jeho výkon bez ohledu na uspořádání vstupního pole.

Na druhou stranu je neefektivní pro sekvenční zpracování dat a pro správné fungování vyžaduje dodatečné množství paměti k uložení mezivýsledků, zejména při paralelním zpracování.

Implementace v jazyce C#

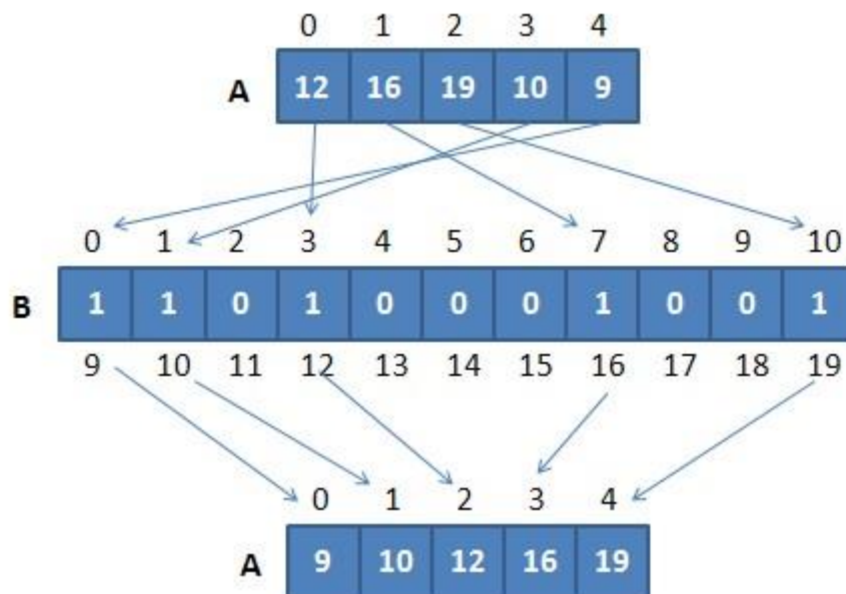
```
// up určuje zda má být část seřazena vzestupně
        public static void CompareAndSwap(int[] array, int i, int j, bool up)
        {
            if ((array[i] > array[j] && up) || (array[i] < array[j] && !up))
            {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }

        // start je index začátku podpole
        // n je počet prvků k seřazení
        public static void BitonicMerge(int[] array, int start, int n, bool up)
        {
            if (n < 2)
                return;

            int m = n / 2;
            for (int i = start; i < start + m; i++)
            {
                CompareAndSwap(array, i, i + m, up);
            }

            BitonicMerge(array, start, m, up);
            BitonicMerge(array, start + m, m, up);
        }


        // start je index začátku podpole
        // n je počet prvků k seřazení
        // up určuje, zda má být seřazení vzestupné nebo sestupné
        public static void BitonicSortRecursive(int[] array, int start, int n, bool
up)
        {
            if (n < 2)
                return;
            int m = n / 2;

            // První polovina seřazená vzestupně, druhá sestupně
            BitonicSortRecursive(array, start, m, true);
            BitonicSortRecursive(array, start + m, m, false);

            // Sloučení obou polovin
            BitonicMerge(array, start, n, up);
        }
```

1.4.10 Pigeonhole sort

Pigeonhole sort je nekomparativní třídící algoritmus, který je nejvhodnější pro třídění, kde počet prvků a rozsah možných klíčových hodnot jsou přibližně stejné. Je účinný, když rozsah prvků není příliš velký vzhledem k počtu prvků, které mají být seřazeny.

Princip fungování

Algoritmus nejprve zjistí maximální a minimální hodnoty v datové sadě a na základě těchto hodnot vytvoří počet „holubníků“ nebo „šuplíků“ pro data. Počet je obvykle roven rozdílu mezi maximální a minimální hodnotou plus jedna – N = max – min + 1. Každý prvek je následně umístěn do odpovídajícího holubníku na základě jeho hodnoty. Toho je dosaženo pomocí odečtení minimální hodnoty od hodnoty prvku, což dá index holubníku. Přestože každý holubník může obsahovat více než jeden prvek a je potřeba zachovat stabilitu, prvky se do holubníků vkládají a vybírají ve stejném pořadí v jakém byly zpracovány. Ke konci jsou prvky vybírány z holubníků od nejnižšího indexu k největšímu. To zajišťuje že jsou prvky vybrány ve vzestupném pořadí a jsou ukládány do seřazeného pole.[71]



Obrázek 30 - Pigeonhole sort [45]

Časová a prostorová složitost

Časová složitost v nejlepším případě: ***O(n + N),*** kde N je počet „holubníků“ a n velikost pole.

Časová složitost v průměrném případě: ***O(n + N)***

Časová složitost v nejhorším případě: ***O(n + N)***

Prostorová složitost: ***O(n + N),*** nejlépe: ***O(n)***

Stabilní: **Ano**

Výhody a nevýhody

Mezi výhody Pigeonhole sortu patří například to, že je extrémně rychlý pro malý rozsah hodnot a je jednoduchý na implementaci, a také je stabilní, pokud jsou prvky vybírány a umisťovány v pořadí, ve které byly zpracovány.

Není ale efektivní pro velký rozsah hodnot, protože jeho prostorová složitost může být vysoká a zároveň vyžaduje přesné znalosti o rozsahu dat, takže nelze být použit všude.

Implementace v jazyce C#

```
public static void PigeonholeSort(int[] arr)
        {
            int min = arr[0];
            int max = arr[0];
            int range, i;

            // Minimum a maximum v poli
            for (i = 1; i < arr.Length; i++)
            {
                if (arr[i] > max)
                    max = arr[i];
                if (arr[i] < min)
                    min = arr[i];
            }

            range = max - min + 1;
            List<int>[] holes = new List<int>[range];

            // Inicializace holubníků
            for (i = 0; i < range; i++)
                holes[i] = new List<int>();

            // Rozdělení prvků do holubníků
            for (i = 0; i < arr.Length; i++)
                holes[arr[i] - min].Add(arr[i]);

            // Vytahování prvků do setříděného pole
            int index = 0;
            for (i = 0; i < range; i++)
            {
                foreach (int val in holes[i])
                {
                    arr[index++] = val;
                }
            }
        }
```

1.5 Porovnání algoritmů

Tabulka 1 Porovnání algoritmů

Algoritmus	Nejlepší	Průměrná	Nejhorší	Prostorová složitost	Stabilita
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ano
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ne
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ano
Merge sort	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n)$	Ano
Quick sort	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n^2)$	$O(n)$	Ne
Heap sort	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(1)$	Ne
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	Ano
Radix sort	$O(n+k)$	$O(nk)$	$O(nk)$	$O(n+k)$	Ano
Bucket sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n \times k)$	Ano
Shell sort	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n^2)$	$O(1)$	Ne
Comb sort	$O(n \times log\ n)$	$O(n^2 / 2^p)$	$O(n^2)$	$O(1)$	Ano
Cocktail sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ano
Timsort	$O(n)$	$O(n \times log\ n)$	$O(n \times log\ n)$	$O(n)$	Ano
Bitonic sort	$O(log^2\ n)$	$O(log^2\ n)$	$O(log^2\ n)$	$O(n \times log^2\ n)$	Ne
Pigeonhole sort	$O(n + N)$	$O(n + N)$	$O(n + N)$	$O(n + N)$, $O(n)$	Ano

2 PRAKTICKÁ ČÁST

V této části práce se zaměřím na vývoj a demonstraci aplikace určené k vizualizaci třídících algoritmů, které jsou popsány v teoretické části. Cílem aplikace je poskytnout uživatelům vizuální nástroj, který jim umožní lepší chápání a analýzu jednotlivých třídících algoritmů. Aplikace byla naprogramována pomocí programovacího jazyka C# a WPF frameworku.

2.1 Použité technologie

2.1.1 Visual Studio 2022

Pro vývoj aplikace jsem zvolil vývojové prostředí Visual Studio 2022, což je velmi používané a robustní IDE podporující mnoho jazyků a frameworků. VS poskytuje řadu nástrojů pro design, testování a ladění, což mi značně usnadňovalo vývoj aplikace.

2.1.2 Programovací jazyk C Sharp (C#)

Jako programovací jazyk jsem zvolil C#, což je moderní, objektově orientovaný jazyk vyvinutý společností Microsoft. C# je ideální pro vývoj na platformě .NET, se kterou Visual Studio spolupracuje a je vhodný pro tvorbu desktopových aplikací s uživatelským rozhraním.

2.1.3 Windows Presentation Framework (WPF)

Pro vytvoření uživatelského rozhraní jsem použil framework Windows Presentation Foundation (WPF), který umožňuje vyvíjet dynamické a vizuálně atraktivní aplikace pro Windows. WPF podporuje XML-based markup jazyk XAML pro definování uživatelského rozhraní, což umožňuje oddělení designu od logiky aplikace.

2.2 Požadavky aplikace

Aplikace by měla splňovat následující požadavky:

<u>Výběr algoritmu</u>

Uživatelé by měli mít možnost z jednoduchého rozhraní vybrat třídící algoritmus, který chtějí použít. Každý algoritmus by měl být popsán svým názvem.

<u>Import a denerování dat</u>

Aplikace by měla umožňovat uživatelům importovat vlastní data celých čísel ve formátu CSV, což by umožnilo práci s reálnými daty. Kromě toho by měla také poskytovat funkci pro generování náhodných čísel, kde uživatelé mohou specifikovat rozsahy hodnot.

Vizualizace třídění

Pro lepší vizuální pochopení algoritmů by měla aplikace obsahovat komponentu pro vizualizaci procesu třídění. Tato funkce by měla být volitelná, s možností uživatele zapnout nebo vypnout vizualizaci.

Statistiky

Během a po dokončení třídění by aplikace měla poskytovat statistiky, jako jsou čas třídění, počet porovnání, prohození a další.

Export výsledků

Uživatelé by měli mít možnost exportovat výsledky třídění do CSV souboru. To by zahrnovalo setříděné a nesetříděné pole, statistiky třídění a časy běhu.

Nastavení parametrů třídění

Uživatelé by měli také mít možnost přizpůsobit klíčové parametry třídění, jako je rozsah hodnot, počet prvků a rychlost animace.

2.3 Návrh aplikace

2.3.1 Ovládací panel

Ovládací panel v hlavním okně aplikace bude obsahovat následující prvky: Výběr algoritmu, nastavení parametrů třídění, tlačítka pro ovládání, vizualizační panel. Uživatel si bude moci vybrat z 15 třídících algoritmů z combo boxu. Bude mít možnost nastavit parametry třídění, jako jsou rozsah hodnot, rychlost animace, možnost vizualizace nebo třídění bez vizualizace, interval výměn vizualizace. Dále bude obsahovat tlačítko pro import vlastních CSV souborů s celými čísly. Také bude obsahovat checkbox sekci pro hromadné třídění.

2.3.2 Panel s informacemi

Panel s informacemi bude zobrazovat statistické informace o probíhajícím třídění, včetně počtu porovnání a výměn, které budou dynamicky aktualizované, počet tříděných prvků, minimální hodnotu v poli, maximální hodnotu v poli a dobu běhu třídícího algoritmu.

2.3.3 Interaktivita a ovládání

Uživatel bude mít možnost třídění zastavit nebo přizpůsobit rychlost, s jakou se vizualizace třídění odehrává, pokud je zapnuta. Tyto prvky budou umístěny v ovládacím panelu v hlavním okně.

2.3.4 Export výsledků a import dat

Uživatel si bude mít možnost po dokončení třídícího procesu vyexportovat výsledky do souboru CSV, které budou obsahovat nesetříděné a setříděné pole, počet prvků, počet výměn, počet porovnání a celkový čas třídění. Také bude mít možnost importovat vlastní data ze souborů CSV, které obsahují celá čísla a následně možnost je třídit.

2.4 Implementace

2.4.1 Třídy

Třída SortingAlgorithm

Tato třída je abstraktní základ pro všechny třídící algoritmy v aplikace. Je navržena tak, aby umožňovala snadné přidání nových algoritmů bez nutnosti měnit obsah ostatní části aplikace. Každý konkrétní algoritmus z této třídy dědí a implementuje její abstraktní metody.

Metoda `Sort(int[] array)` je abstraktní metoda určená k přepsání v odvozených třídách, která obsahuje logiku konkrétního třídícího algoritmu.

Metoda `GetStatistics()` vrátí statistiky třídění, jako jsou počet porovnání, počet výměn a dobu trvání třídění.

Metoda `Reset()` resetuje statistiky, aby bylo možné použít znovu instanci algoritmu bez předchozích dat.

Metody `StartTimer()`, `StopTimer()` a `GetTimer()` jsou metody pro správu časovače, který měří dobu třídění.

Metody `IncreaseSwap()`, `IncreaseComparisons()` slouží jako pomocné metody pro inkrementaci počtu výměn a porovnání.

Událost `OnSwap` je událost vyvolaná při každé výměně prvků v poli. Tato událost je používána pro vizualizaci procesu třídění. Vyvolání události je zajištěno chráněnou metodou `RaiseOnSwap(int[] array, int indexA, int indexB)`.

Vlastnost `Name` je abstraktní vlastnost, která je implementována v odvozených třídách pro identifikaci algoritmu.

Vlastnost `AnimationDelay` je časové zpoždění používané při vizualizaci třídění.

Vlastnost `IsVisualized` určuje, zda má být třídění vizualizováno.

Třída SortingStatistics

Třída `SortingStatistics` slouží k uchování statistických dat o třídění, jako jsou počet provedených výměn, počet provedených porovnání a také celková doba trvání třídění.

Obsahuje atribut `Comparisons` pro uchování počtu porovnání, `Swaps` pro uchování počtu výměn a `Duration` pro uchování celkové doby třídění.

Třída NumberGenerator

Třída `NumberGenerator` je služba v aplikace určená ke generování náhodných číselných polí, které jsou použity v rámci třídění. Poskytuje metody pro vytvoření celočíselných polí s nastavitelným rozsahem hodnot a počtem prvků.

Metoda `GenerateRandomArray(int count, int minValue, int maxValue)` generuje pole celých čísel s definovaným počtem prvků, minimální a maximální hodnotou. Pokud jsou parametry zadány nesprávně, tak metoda vyvolá výjimku.

Třída SortAlgorithmFactory

Třída `SortAlgorithmFactory` slouží jako bod pro získávání instancí třídících algoritmů v aplikaci. Je postavena na návrhovém vzoru Factory, který je používán pro vytváření objektů bez specifikace konkrétních tříd objektů. Je implementována jako statická třída, což znamená, že její metody a vlastnosti jsou přístupné globálně bez nutnosti vytváření instance třídy.

Metoda `GetAlgorithm(string name)` přijímá název algoritmu jako řetězec a vrací instanci odpovídajícího algoritmu. Využívá příkaz switch, aby rozhodla, který objekt vrátit na základě zadaného názvu. Podporuje všech 15 algoritmů popsaných v teoretické části.

Třída DataExport

Třída `DataExport` slouží k exportování dat třídění a statistik do souboru CSV. Je navržena jako utilitní třída, což znamená, že její metody jsou statické a může být přístupná bez nutnosti instance třídy. Obsahuje metodu `ExportToCsv(int[] sortedData, int[] unsortedData, SortingStatistics stats)`, která nejprve otevírá dialog pro uložení souboru pomocí SaveFileDialog, který uživatelům umožňuje zvolit umístění a název souboru pro export. Po potvrzení umístění metoda vytvoří obsah CSV souboru. Nakonec je soubor CSV uložen do vybraného umístění a zobrazí se informační okno s potvrzením exportu.

Další Třídy

`BitonicSort`, `BubbleSort`, `BucketSort`, `CocktailSort`, `CombSort`, `CountingSort`, `HeapSort`, `InsertionSort`, `MergeSort`, `PigeonholeSort`, `QuickSort`, `RadixSort`, `SelectionSort`, `ShellSort`, `TimSort` jsou třídy, které dědí z třídy `SortingAlgorithm` a obsahují vlastní implmentaci metody `Sort()` v závislosti na algoritmu.

2.4.2 Ukázka třídy BubbleSort

Třída `BubbleSort` je konkrétní implementací abstraktní třídy `SortingAlgorithm` pro třídící algoritmus Bubble Sort. Tato třída definuje vlastní implementaci abstaktních metod a vlastností definovaných v základní třídě.

```
public class BubbleSort : SortingAlgorithm
    {

        public override string Name => "Bubble Sort";

        public override async Task Sort(int[] array)
        {

                StartTimer();
                bool swapped;
                for (int i = 0; i < array.Length - 1; i++)
                {
                    swapped = false;
                    for (int j = 0; j < array.Length - i - 1; j++)
                    {
                        IncreaseComparisons();
                        if (array[j] > array[j + 1])
                        {
                            int temp = array[j];
                            array[j] = array[j + 1];
                            array[j + 1] = temp;
                            swapped = true;
                            IncreaseSwaps();
                            GetTimer();
                            if (stopRequested)
                            {
                                return;
                            }
                            if (IsVisualized)
                            {
                                RaiseOnSwap(array, j, j + 1);
                                await Task.Delay(AnimationDelay);
                            }

                        }
                    }
                    if (!swapped) break;
                }
                StopTimer();

        }
```

```
        public override SortingStatistics GetStatistics()
        {
            return Statistics;
        }

        public override void Reset()
        {
            Statistics.Reset();
        }
    }
```

2.4.3 Třída MainWindow

Třída `MainWindow` obsahuje všechny metody a event handlery, které řídí interakci uživatele s grafickými komponentami okna a zajišťují funkčnost třídění.

Atributy: `loadedData` – pole pro ukládání načtených nebo vygenerovaných dat pro třídění; `numbersCount` – počet prvků v načtených datech; `numbersGenerator` – instance třídy pro generování náhodného pole celých čísel; `sorter` – referenční proměnná pro aktuálně vybraný třídící algoritmus.

Metody

`BtnSort_Click` – tato metoda je vyvolána po kliknutí na tlačítko Sort. Inicializuje třídící proces pro vybraný algoritmus, nastavuje parametry třídění a zahajuje vizualizaci třídění. Také spravuje dostupnost ostatních ovládacích prvků během třídění. Při zapnuté vizualizaci je nastaven fixní počet prvků na hodnotu 70 pro lepší přehlednost.

`BtnStop_Click` – Slouží k zastavení běžícího třídění po stisknutí tlačítka Stop.

`BtnLoad_Click` – Umožňuje načítat vlastní soubory ve formátu CSV. Po načtení aktualizuje uživatelské rozhraní pro zobrazení načtených dat. Pokud obsahuje soubor více než 70 prvků, tak se zavolá metoda `chkPerfomance_Checked` a přejde se do režimu performance.

`UpdateStatistics` – Aktualizuje textová pole s aktuálními statistikami třídění.

`VisualizeSwap` a `DrawArray` – Slouží k zajištění vizualizace třídění na Canvasu, kde je každý prvek reprezentován jako obdélník.

`AnimationSpeedSlider_ValueChanged` – Metoda, která se spustí při změně hodnoty slideru pro rychlost animace a upravuje delay mezi jednotlivými kroky třídění.

`chkPerformance_Checked` a `chkPerformance_Unchecked` – Jsou event handlery pro zaškrtávání pole Performance, které skrývají nebo odkrývají Canvas.

chkVisualisation_Checked a chkVisualisation_Unchecked – spravují, podobně jako performance zobrazení Canvasu.

BtnGenerateRandomArray – Metoda slouží k vygenerování náhodného pole celých čísel. Čte hodnoty z uživatelských vstupů pro minimální a maximální hodnotu a počet prvků z pole count. Vytvořené pole se uloží do proměnné loadedData a tlačítko pro spuštění všech se aktivuje.

BtnCompareSelected_Click – Tato metoda je navržena pro hromadné porovnávání výkonu různých algoritmů na stejném poli. Nejdříve zkontroluje, zda je pole loadedData dostupné pro třídění. Pokud ano, tak vytvoří kopii těchto dat, aby měl každý algoritmus stejný dataset. Pro každý zaškrtnutý algoritmus se inicializuje odpovídající instance pomocí SortAlgorithmFactory. Každý algoritmus poté shromažďuje statistiky do seznamu results. Po dokončení se otevře okno CompareAlgorithms, kde se zobrazí srovnání algoritmů.

Okno MainWindow.xaml

Okno MainWindow slouží jako hlavní uživatelské rozhraní pro aplikaci pro vizualizaci algoritmů. Okno je nastaveno na pevnou velikost 1800x900 pixelů bez možnosti změny velikosti. Okno používá grid layout se třemi sloupci. Levý a pravý sloupec jsou pevně nastaveny na šířku 200 pixelů, zatímco prostřední sloupec zabírá zbytek prostoru. V levé části se nachází ComboBox, který umožňuje uživatelům vybrat z různých třídících algoritmů, kde je každý algoritmus reprezentován jako položka v rozevíracím seznamu. Dále se zde nachází tlačítka Sort, Stop a Load Data. Tlačítko Sort spustí třídění pro vybraný algoritmus, tlačítko Stop pozastaví třídící proces a ukončí ho a tlačítko Load slouží k načtení vlastní CSV souborů s celými čísly. Pod tlačítky se nachází ComboBox pro nastavení intervalu vizualizace (po kolika prohození se aktualizuje Canvas). V TextBoxech „Range of Values“ si uživatel může před Stisknutím tlačítka Sort nastavit rozemezí hodnot, které budou náhodně vygenerovány při spuštění třídění. CheckBox performance nám skryje komponentu Canvas a v pozadí se nenastaví žádný delay pro vizualizaci, takže výsledky nám poskytnou přesnější údaj o době třídění. CheckBox Visualization nám naopak Canvas zobrazí, aby bylo možné pozorovat třídění, kterému může uživatel nastavit za pomocí Slideru delay(ms) třídění. Pod Sliderem se nachází CheckBox pro třídění stejného pole pomocí více algoritmy najednou.

Range of Values:
1 500
Count:
500
Generate unsorted array
Compare Multiple Algorithms
Bubble sort
Selection sort
Insertion sort
Merge sort
Quick sort
Heap sort
Counting sort
Radix sort
Bucket sort
Shell sort
Comb sort
Cocktail sort
Timsort
Bitonic sort
Pigeonhole sort
Compare selected

Obrázek 31 - Porovnání více algoritmů menu Zdroj: vlastní

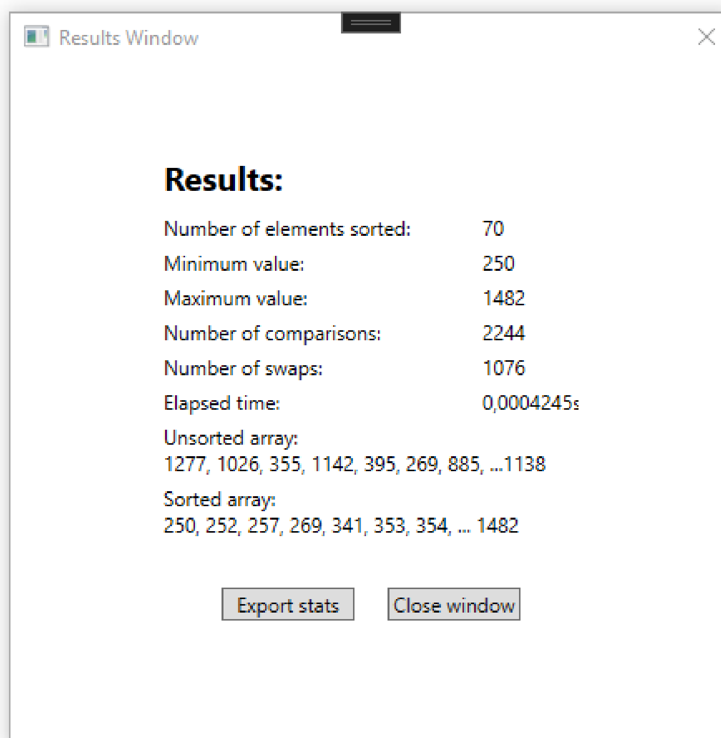
Zde musí uživatel nejprve nastavit rozmezí generovaných hodnot a jejich počet, poté za pomocí tlačítka „Generate unsorted array“ je vygeneruje. Dále musí zaškrtnout algoritmy jimiž chce pole třídit a následně už stačí jen pomocí tlačítka „Compare selected“ spustit proces třídění. Uprostřed okna se nachází Canvas pro vizualizaci třídění, kde jednotlivé prvky jsou zobrazovány jako barevné obdélníky, jejichž výška odpovídá procentuálně hodnotě prvku. V pravé části hlavního okna jsou zobrazeny statistiky třídění, které se za běhu aktualizují.

Obrázek 32 - Main Window Zdroj: vlastní

2.4.4 Třída ResultsWindow

Třída `ResultsWindow` je používána k zobrazení statistik třídění a výsledků třídících operací po dokončení třídění. Konstruktor přijímá tři parametry: sortedDate, UnsortedData a stats, které reprezentují seřazené pole, neseřazené pole a statistiky třídění. Pro inicializaci se volá metoda InitializeComponent() a následně se volá metoda `DisplayResults()`, která používá data a statistiky pro vyplnění textových polí ve formuláři. Tlačítko pro zavření okna má přiřazenou událost `CloseWindow_Click` a tlačítko pro export výsledků má přiřazenou událost `Export_Click`, která vyvolá export dat a statistik do CSV formátu pomocí metody `DataExport.ExportToCsv`.

Okno ResultsWindow.xaml



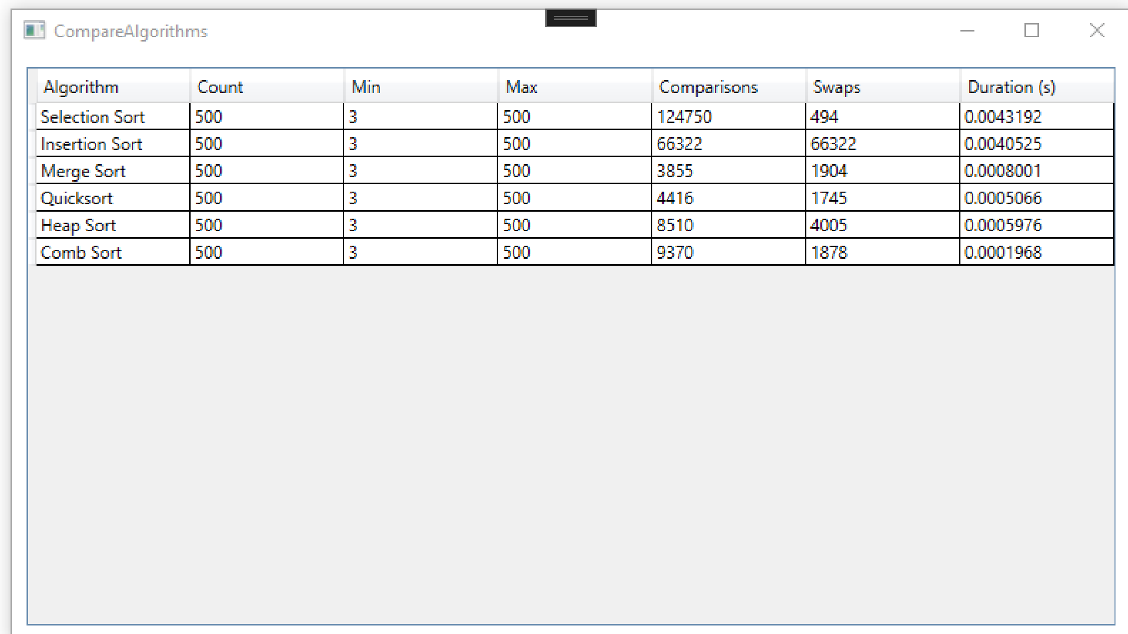
Obrázek 33 - Okno results Zdroj: vlastní

2.4.5 Třída CompareAlgorithms

Třída `CompareAlgorithms` slouží pro zobrazení výsledků třídění vybraných třídících algoritmů. Konstruktor volá pro inicializaci metodu `InitializeComponent()`. Třída obsahuje veřejnou metodu `SetResults(List<SortingResult> results)`, která umožňuje nastavit zdroj dat pro DataGrid zvaný resultsDataGrid. Ten zobrazuje seznam

výsledků třídění, kde každý výsledek obsahuje informace jako název algoritmu, počet prvků, minimální a maximální hodnotu, počet porovnání a výměn a celkovou dobu trvání třídění. Třída `SortingResults` je vnořená třída, která se používá pro ukládání a zobrazení statistických informací o jednotlivých algoritmech.

Okno CompareAlgorithms.xaml



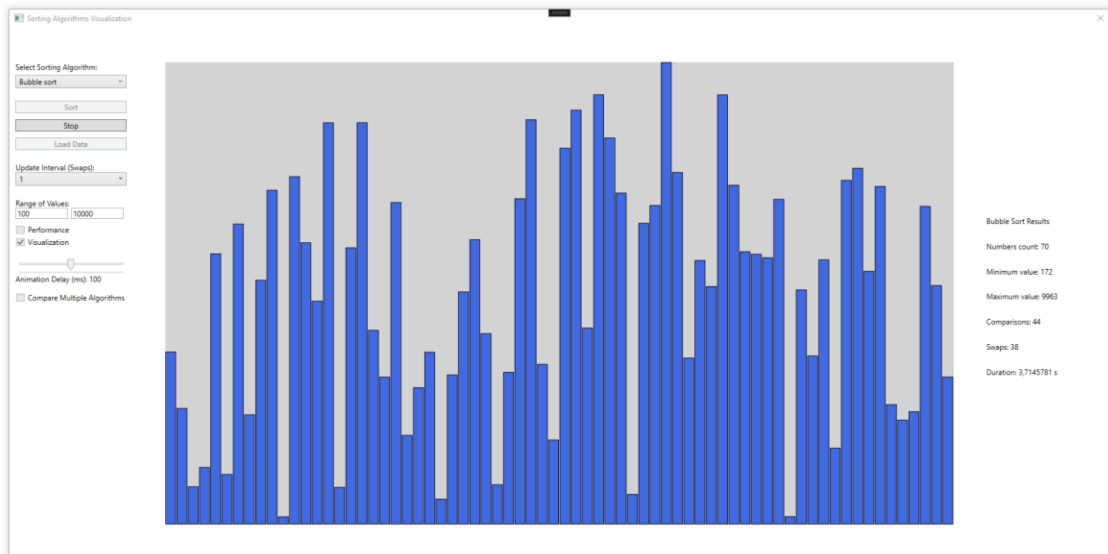
Algorithm	Count	Min	Max	Comparisons	Swaps	Duration (s)
Selection Sort	500	3	500	124750	494	0.0043192
Insertion Sort	500	3	500	66322	66322	0.0040525
Merge Sort	500	3	500	3855	1904	0.0008001
Quicksort	500	3	500	4416	1745	0.0005066
Heap Sort	500	3	500	8510	4005	0.0005976
Comb Sort	500	3	500	9370	1878	0.0001968

Obrázek 34 - Okno CompareAlgorithms Zdroj: vlastní

2.5 Ukázka aplikace

2.5.1 Vizualizace

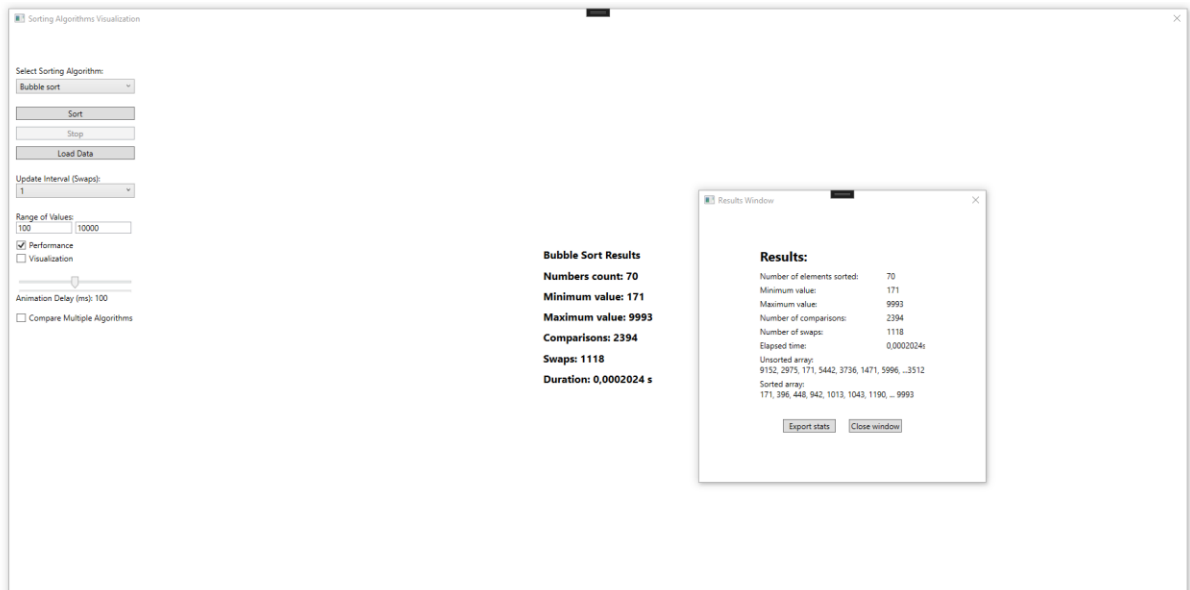
Vizualizace, konkrétně grafické zobrazení průběhu třídění lze jednoduše zapnout tlačítkem Sort. Nad tlačítkem Sort si nejdříve vybereme, jaký algoritmus chceme vizualizovat. Během třídění není možné znovu tlačítko Sort stisknout, až po do dokončení. To samé platí pro tlačítko Load Data, tudíž nelze během procesu třídění nahrávat vlastní soubory. Vždy při zapnuté vizualizaci je maximální počet prvků v Canvasu 70, a pokud tuto hranici překročíme i nahráním souboru CSV, tak se automaticky aplikace přepne do režimu performance. Pokud chceme nastavit rozsahy generovaných hodnot, musíme tak učinit vždy před spuštěním třídění, a povolené hodnoty jsou pouze celá čísla. Při zapnuté vizualizaci máme aktuální statistiky třídění na levé straně okna, která se v průběhu třídění aktualizují.



Obrázek 35 - Ukázka vizualizace Zdroj: vlastní

2.5.2 Bez vizualizace

Pokud uživatel zaškrtne checkbox „performance“, tak se nám plocha pro vykreslování skryje a na její místo se nám přesunou statistiky třídění. V tomto módu se při třídění zanedbává update interval a delay, který slouží pro zpoždění animace třídění. Po skončení procesu třídění se nám vždy zobrazí dialogové okno s výsledky daného třídění.



Obrázek 36 - Ukázka bez vizualizace Zdroj: vlastní

2.5.3 Hromadné porovnání algoritmů

Pokud chce uživatel porovnávat více algoritmů najednou na stejném setu dat, tak musí zaškrtnout políčko „Compare Multiple Algorithms“. To nám skryje nepotřebné komponenty pro tento režim a zobrazí nám další zaškrtávací políčka pro výběr jednotlivých algoritmů. Uživatel musí nejdříve dle vlastní potřeby nastavit rozsahy a počet prvků, které následně vygeneruje pomocí tlačítka „Generate unsorted array“. Poté, co je pole vygenerováno, se nám odblokuje tlačítko „Compare selected“. Při velkém množství dat bude chvíli trvat, než se nám zobrazí dialogové okno, které obsahuje tabulku s naměřenými hodnotami a názvy algoritmů. Výsledky uživatel může řadit podle jednotlivých sloupců. Hodnoty swaps u nekomparativních algoritmů jsou brány jako přiřazení do pole než jako samotná výměna prvků. Po každém třídění je nutno znovu vygenerovat pole čísel.

CompareAlgorithms

Algorithm	Count	Min	Max	Comparisons	Swaps	Duration (s)
Bubble Sort	5000	101	9999	12493222	6297530	0.4438893
Selection Sort	5000	101	9999	12497500	4993	0.4710566
Insertion Sort	5000	101	9999	6297530	6297530	0.3652021
Merge Sort	5000	101	9999	55238	27133	0.0106003
Quicksort	5000	101	9999	70047	33711	0.0074281
Heap Sort	5000	101	9999	119184	57092	0.0124405
Counting Sort	5000	101	9999	0	5000	0.0002855
Radix Sort	5000	101	9999	0	20000	0.0023478
Bucket Sort	5000	101	9999	5576	10576	0.0008256
Shell Sort	5000	101	9999	61788	61788	0.004283
Comb Sort	5000	101	9999	148381	28451	0.0031382
Cocktail Sort	5000	101	9999	9499924	6297530	0.3755257
Timsort	5000	101	9999	77315	77279	0.0058606
Bitonic Sort	5000	101	9999	372736	114795	0.0435153
Pigeonhole Sort	5000	101	9999	0	5000	0.0094904

Range of Values:
100
10000
Count:
5000
Generate unsorted array
Compare Multiple Algorithms
Bubble sort
Selection sort
Insertion sort
Merge sort
Quick sort
Heap sort
Counting sort
Radix sort
Bucket sort
Shell sort
Comb sort
Cocktail sort
Timsort
Bitonic sort
Pigeonhole sort
Compare selected

Obrázek 37 - Hromadné třídění Zdroj: vlastní

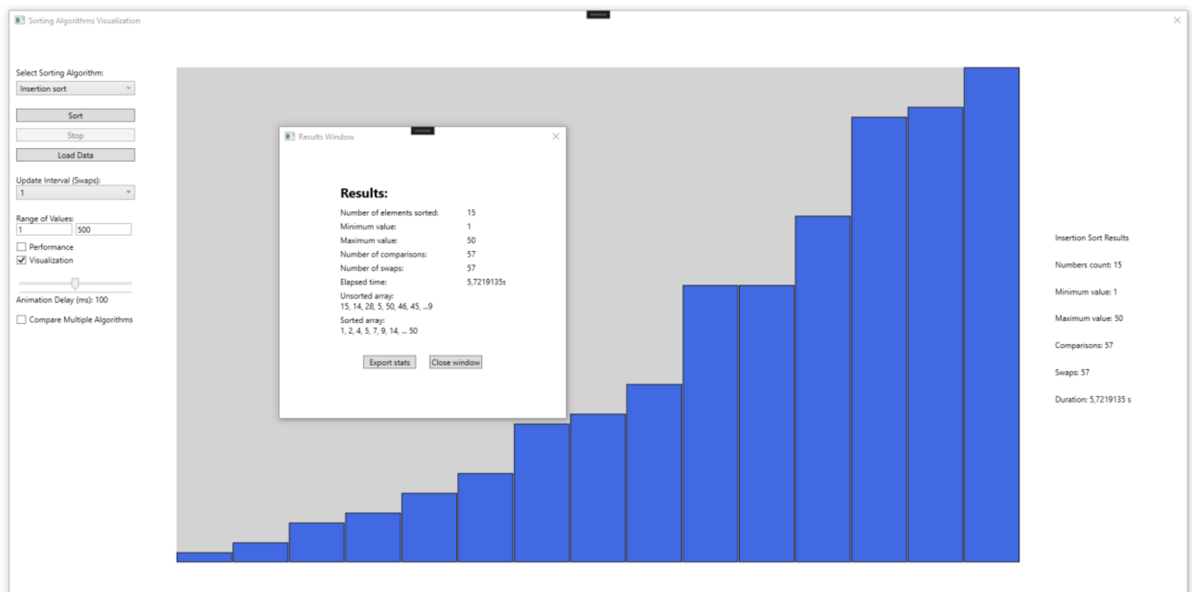
2.5.4 Import vlastních dat

Pro import vlastních dat uživatel jednoduše stiskne tlačítko Load Data. Je potřeba, aby data byla uložena ve správném formátu v CSV souboru. Já pro demonstraci zvolil soubor následujícími daty:

	A
1	15
2	14
3	28
4	5
5	50
6	46
7	45
8	1
9	2
10	18
11	4
12	35
13	7
14	28
15	9
16	

Obrázek 38 - Testovací csv data Zdroj: vlastní

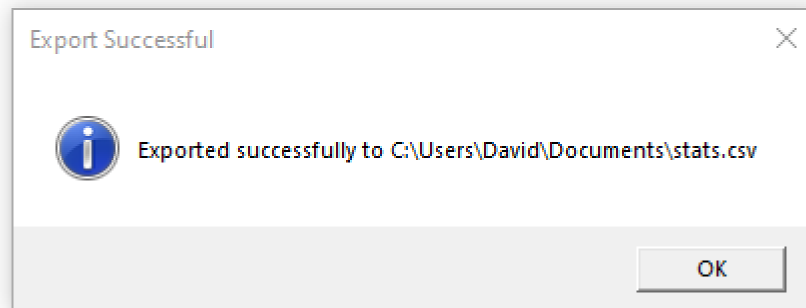
Pokud jsou data úspěšně importována, zobrazí se nám v canvasu neseřazená. Pokud ovšem soubor obsahuje více než 70 hodnot, tak se automaticky program přepne do režimu performance, kvůli zachování přehlednosti. Následně stačí už jen stlačit tlačítko Sort a importovaná data budou seřazena.



Obrázek 39 - Import dat Zdroj: vlastní

2.5.5 Export výsledků

Po každém úspěšném třídění si uživatel může výsledky nechat vyexportovat do souboru CSV pomocí tlačítka „export stats“.



Obrázek 40 - Export dat Zdroj: vlastní

	A	B	C	D	E	F	G	H	I	J
1	Statistic,Value									
2	Number of elements sorted,70									
3	Number of comparisons,2400									
4	Number of swaps,1342									
5	Elapsed time,3,0173446s									
6	Minimum value,5									
7	Maximum value,491									
8	Unsorted array,69, 407, 168, 478, 17, 188, 168, 58, 319, 363, 140, 8, 216, 441, 48, 341, 213, 395, 376, 407, 1									
9	Sorted array,5, 6, 8, 17, 17, 18, 24, 33, 38, 44, 47, 48, 54, 54, 58, 69, 85, 91, 96, 103, 104, 110, 117, 124, 126,									
10										
11										
12										

Obrázek 41 - Exportovaný soubor Zdroj: vlastní

2.5.6 Naměřené hodnoty

První test

V prvním testu se zaměříme na srovnání výkonnosti všech dostupných algoritmů. Celkový počet prvků pro třídění byl stanoven na 1000, s náhodně vygenerovanými hodnotami v rozmezí od 1 do 10 000. U Counting, Bucket, Radix a Pigeonhole sortů je swap počítáno jako přiřazení do pole. Výsledky jsou seřazeny od nejrychlejšího po nejpomalejší.

Tabulka 2 - Porovnání algoritmů 1. Test

Algorithm	Count	Min	Max	Comparisons	Swaps	Time (s)
Counting Sort	1000	8	9995	0	1000	8,56E-05
Bucket Sort	1000	8	9995	224	1224	0,0002947
Shell Sort	1000	8	9995	7468	7468	0,0005509
Comb Sort	1000	8	9995	22709	4409	0,0005675
Radix Sort	1000	8	9995	0	4000	0,00092
Pigeonhole Sort	1000	8	9995	0	1000	0,0011984
Quicksort	1000	8	9995	10997	4409	0,0020891
Timsort	1000	8	9995	12816	12797	0,0020917
Heap Sort	1000	8	9995	19136	9068	0,0021563
Merge Sort	1000	8	9995	8718	4291	0,002741
Bitonic Sort	1000	8	9995	28160	13817	0,0114988
Insertion Sort	1000	8	9995	244813	244813	0,0139886
Cocktail Sort	1000	8	9995	375747	244813	0,0148902
Bubble Sort	1000	8	9995	496944	244813	0,0183886
Selection Sort	1000	8	9995	499500	995	0,0188989

Z výsledků je zřejmé, že algoritmy, které využívají pokročilejší metody rozdělení dat, jako jsou Radix sort a Bucket sort dosáhly výrazně lepších časů třídění ve srovnání s tradičními algoritmy jako je Bubble sort a Selection sort.

Druhý test

Ve druhém testu budeme postupovat obdobně a použijeme stejné algoritmy, jen změníme rozsahy hodnot. Celkový počet prvků nastavíme na 10 000 s rozsahy od 1 000 do 100 000. Výsledky jsou opět seřazeny od nejrychlejšího po nejpomalejší.

Tabulka 3 - Porovnání algoritmů 2. Test

Algorithm	Count	Min	Max	Comparisons	Swaps	Time (s)
Counting Sort	10000	1003	99982	0	10000	0,0009392
Bucket Sort	10000	1003	99982	2325	12325	0,0022004
Radix Sort	10000	1003	99982	0	50000	0,005091
Comb Sort	10000	1003	99982	336733	63800	0,007013
Shell Sort	10000	1003	99982	156285	156285	0,0098662
Timsort	10000	1003	99982	162157	162137	0,010763
Quicksort	10000	1003	99982	152465	70867	0,0136463
Merge Sort	10000	1003	99982	120380	59075	0,0193064
Heap Sort	10000	1003	99982	258378	124189	0,025313
Pigeonhole Sort	10000	1003	99982	0	10000	0,041003
Bitonic Sort	10000	1003	99982	860160	266154	0,0942881
Insertion Sort	10000	1003	99982	25051654	25051654	1,4177071
Cocktail Sort	10000	1003	99982	37487499	25051654	1,4943814
Bubble Sort	10000	1003	99982	49965354	25051654	1,7628241
Selection Sort	10000	1003	99982	49995000	9993	1,7817108

Třetí test

Ve třetím testu bylo třídění prováděno na 10 000 000 prvcích v rozmezí 1 000 – 1 000 000 000. Bylo vybráno pouze sedm nejrychlejších algoritmů z předchozího testu, protože algoritmy jako například Selection sort nebo Bubble sort by byly na tento set neefektivní.

Tabulka 4 - Porovnání algoritmů 3. Test

Algorithm	Count	Min	Max	Comparisons	Swaps	Time (s)
Counting Sort	10000000	1000	10000000	0	10000000	1,9676833
Bucket Sort	10000000	1000	10000000	22495979	32495979	2,9364569
Radix Sort	10000000	1000	10000000	0	80000000	7,276221
Comb Sort	10000000	1000	10000000	656666766	117750797	13,3237226
Tim Sort	10000000	1000	10000000	263538594	263539605	16,5441254
Quick Sort	10000000	1000	10000000	289443537	132635592	17,7051218
Shell Sort	10000000	1000	10000000	858386659	858386659	48,6851209

ZÁVĚR

Cílem této bakalářské práce bylo poskytnout čtenářům ucelený pohled na problematiku třídících algoritmů, zkoumat jejich výkonnost a efektivitu, a demonstrovat tuto problematiku prostřednictvím vizualizační aplikace. Práce byla rozdělena do dvou hlavních částí – teoretické a praktické.

V teoretické části byly popsány základní i pokročilé třídící algoritmy, bylo představeno jejich fungování, principy, výkonnost a časová složitost. Tato analýza pomohla poskytnout lepší porozumění pro každý algoritmus.

Praktická část práce se zaměřovala na vývoj vizualizační aplikace, která byla implementována v jazyce C# s využitím frameworku WPF. Aplikace umožňuje uživatelům interaktivně pracovat s různými třídícími algoritmy, což napomáhá lepšímu pochopení jejich chování a efektivity. Z naměřených hodnot vyplývá, že algoritmy jako Radix sort a Counting sort vykazují výrazně lepší časy na větším objemu dat, zatímco algoritmy jako Bubble sort a Insertion sort jsou vhodné pro menší datové sady.

Výsledky práce naznačují, že výběr třídícího algoritmu by měl být prováděn na základě specifických potřeb aplikace a charakteristik dat, které mají být tříděny. Závěrem, tato práce přispěla k lepšímu pochopení složitosti a významu třídících algoritmů a poskytla nástroj pro jejich vizualizaci.

POUŽITÁ LITERATURA

[1] *Řadící algoritmus*. Online. Wikipedia. 2023. Dostupné z: https://cs.wikipedia.org/wiki/%C5%98adic%C3%AD_algoritmus. [cit. 2024-02-26].

[2] Algoritmus. *Mioweb* [online]. 2020 [cit. 2024-02-26]. Dostupné z: https://www.mioweb.cz/slovnicek/algoritmus/

[3] Řazení. *Wikipedia* [online]. 2023 [cit. 2024-02-26]. Dostupné z: https://cs.wikipedia.org/wiki/%C5%98azen%C3%AD

[4] Časová složitost algoritmů. *ČVUT* [PDF]. 2009 [cit. 2024-02-26]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/y36alg/2009_sumperk_p10.pdf

[5] Úvod do teoretické informatiky. *VŠB - Technická univerzita Ostrava* [PDF]. 2020 [cit. 2024-02-26]. Dostupné z: https://www.cs.vsb.cz/sawa/uti/2020/slides/uti-12-cz.pdf

[6] In-Place Algorithm. *Geeksforgeeks* [online]. 2022 [cit. 2024-02-26]. Dostupné z: https://www.geeksforgeeks.org/in-place-algorithm/

[7] Bubble sort. *Algoritmy* [online]. 2016 [cit. 2024-02-27]. Dostupné z: https://www.algoritmy.net/article/3/Bubble-sort

[8] Bubble_sort. *Algolist* [online]. 2009 [cit. 2024-02-27]. Dostupné z: https://www.algolist.net/Algorithms/Sorting/Bubble_sort

[9] HARTINGER, David. Lekce 2 - Bubblesort. *Itnetwork* [online]. c2024 [cit. 2024-02-27]. Dostupné z: https://www.itnetwork.cz/algoritmy/razeni/algoritmus-bubblesort-probublavani-trideni-cisel

[10] *Bubble_sort-1*. Online. In: Algolist.net. C. Dostupné z: https://www.algolist.net/img/sorts/bubble-sort-1.png. [cit. 2024-04-17].

[11] *Selection-sort-1*. Online. In: Algolist.net. C. Dostupné z: https://www.algolist.net/img/sorts/selection-sort-1.png. [cit. 2024-04-17].

[12] *Insertion-sort-1*. Online. In: Algolist.net. C. Dostupné z: https://www.algolist.net/img/sorts/insertion-sort-1.png. [cit. 2024-04-17].

[13] *Merge-sort-example_0*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/merge-sort-example_0.png. [cit. 2024-04-17].

[14] *Quick-sort-0.1_0*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-0.1_0.png. [cit. 2024-04-17].

[15] *Quick-sort-partition-first-step*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-partition-first-step.png. [cit. 2024-04-17].

[16] *Quick-sort-partition-second-step*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-partition-second-step.png. [cit. 2024-04-17].

[17] *Quick-sort-partition-third-step*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-partition-third-step.png. [cit. 2024-04-17].

[18] *Quick-sort-partition-fifth-step*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-partition-fifth-step.png. [cit. 2024-04-17].

[19] *Quick-sort-partition-sixth-step*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/quick-sort-partition-sixth-step.png. [cit. 2024-04-17].

[20] *1mghTRv.png*. Online. In: Imgur. 2018. Dostupné z: https://i.imgur.com/1mghTRv.png. [cit. 2024-04-17].

[21] *Heap_sort*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/heap_sort.png. [cit. 2024-04-17].

[22] *Counting-sort-0_0*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Counting-sort-0_0.png. [cit. 2024-04-17].

[23] *Counting-sort-1*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Counting-sort-1.png. [cit. 2024-04-17].

[24] *Counting-sort-2*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Counting-sort-2.png. [cit. 2024-04-17].

[25] *Counting-sort-3*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Counting-sort-3.png. [cit. 2024-04-17].

[26] *Counting-sort-4_1*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Counting-sort-4_1.png. [cit. 2024-04-17].

[27] *Radix-sort-0_0*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Radix-sort-0_0.png. [cit. 2024-04-17].

[28] *Bucket_2*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/Bucket_2.png. [cit. 2024-04-17].

[29] *Shell-sort-0.2*. Online. In: Programiz. B. r. Dostupné z: https://www.programiz.com/sites/tutorial2program/files/shell-sort-0.2.png. [cit. 2024-04-17].

[30] *Comb-sort-algorithm2*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/comb-sort-algorithm2.png. [cit. 2024-04-17].

[31] *Comb-sort-algorithm4*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/comb-sort-algorithm4.png. [cit. 2024-04-17].

[32] *Cocktail-sort20*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort20.png. [cit. 2024-04-17].

[33] *Cocktail-sort21*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort21.png. [cit. 2024-04-17].

[34] *Cocktail-sort22*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort22.png. [cit. 2024-04-17].

[35] *Cocktail-sort23*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort23.png. [cit. 2024-04-17].

[36] *Cocktail-sort25*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort25.png. [cit. 2024-04-17].

[37] *Cocktail-sort27*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort27.png. [cit. 2024-04-17].
[38] *Cocktail-sort28*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/cocktail-sort28.png. [cit. 2024-04-17].
[39] *Tim-sort-algorithm1*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/tim-sort-algorithm1.png. [cit. 2024-04-17].
[40] *Tim-sort-algorithm2*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/tim-sort-algorithm2.png. [cit. 2024-04-17].
[41] *Tim-sort-algorithm5*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/tim-sort-algorithm5.png. [cit. 2024-04-17].
[42] *Tim-sort-algorithm6*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/tim-sort-algorithm6.png. [cit. 2024-04-17].
[43] *Bitonic-sort*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/bitonic-sort.png. [cit. 2024-04-17].
[44] *Bitonic-sort2*. Online. In: Javapoint. C2011-2021. Dostupné z: https://static.javatpoint.com/ds/images/bitonic-sort2.png. [cit. 2024-04-17].
[45] *VKhzI.jpg*. Online. In: Riptutorial. B. r. Dostupné z: http://i.stack.imgur.com/VKhzI.jpg. [cit. 2024-04-17].
[46] VIRIUS, Miroslav. *Programování v C#: od základů k profesionálnímu použití*. Knihovna programátora (Grada). Praha: Grada Publishing, 2021. ISBN 978-80-271-1216-6.
[47] MEHLHORN, Kurt a SANDERS, Peter. *Algorithms and data structures: the basic toolbox*. Berlin: Springer, c2010. ISBN 978-364-2096-822.
[48] CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L. a STEIN, Clifford. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press, [2022]. ISBN 978-026-2046-305.
[49] RYANT, Ivan. *Algoritmy a datové struktury objektově*. Vydání první. V Praze: Ivan Ryant, [2017]. ISBN 978-80-270-1660-0.
[50] *Selection sort Algorithm*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/selection-sort. [cit. 2024-04-18].
[51] *Selection Sort*. Online. Algolist. 2009. Dostupné z: https://www.algolist.net/Algorithms/Sorting/Selection_sort. [cit. 2024-04-18].
[52] *Insertion sort Algorithm*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/insertion-sort. [cit. 2024-04-18].
[53] *Insertion sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/insertion-sort. [cit. 2024-04-18].
[54] *Mergesort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/merge-sort. [cit. 2024-04-18].
[55] *Merge sort Algorithm*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/merge-sort. [cit. 2024-04-18].
[56] *Quicksort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/quick-sort. [cit. 2024-04-18].
[57] *Quick sort Algorithm*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/quick-sort. [cit. 2024-04-18].
[58] *Heap sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/heap-sort. [cit. 2024-04-18].
[59] *Counting sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/counting-sort. [cit. 2024-04-18].

[60] *Radix sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/radix-sort. [cit. 2024-04-18].

[61] *Radix sort Algorithm*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/radix-sort. [cit. 2024-04-18].

[62] *Bucket sort algorithm*. Online. Simplilearn. 2023. Dostupné z: https://www.simplilearn.com/tutorials/data-structure-tutorial/bucket-sort-algorithm. [cit. 2024-04-18].

[63] *Bucket sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/bucket-sort. [cit. 2024-04-18].

[64] *Shell sort*. Online. Programiz. B. r. Dostupné z: https://www.programiz.com/dsa/shell-sort. [cit. 2024-04-18].

[65] *Shellsort*. Online. Wikipedia. 2024. Dostupné z: https://en.wikipedia.org/wiki/Shellsort. [cit. 2024-04-18].

[66] *Comb sort*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/comb-sort. [cit. 2024-04-18].

[67] *Cocktail sort*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/cocktail-sort. [cit. 2024-04-18].

[68] *Tim sort*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/tim-sort. [cit. 2024-04-18].

[69] *Efektivní paralelizace algoritmu Timsort*. Bakalářská práce. Praha: Fakulta Informačních technologií ČVUT v Praze, 2018.

[70] *Bitonic sort*. Online. Javatpoint. C2011-2021. Dostupné z: https://www.javatpoint.com/bitonic-sort. [cit. 2024-04-18].

[71] *Pigeonhole sort*. Online. Wikipedia. 2024. Dostupné z: https://en.wikipedia.org/wiki/Pigeonhole_sort. [cit. 2024-04-24].

[72] *Sorting algorithm*. Online. Wikipedia. 2024. Dostupné z: https://en.wikipedia.org/wiki/Sorting_algorithm. [cit. 2024-02-27].

SEZNAM PŘÍLOH

Zip soubor se zdrojovým kódem aplikace