Střední průmyslová škola a Vyšší odborná škola, Liberec, příspěvková organizace

POROVNÁNÍ ŘADÍCÍCH ALGORITMŮ

Ročníkový projekt

Autor **bohuslav.cermak.023@pslib.cz;vit.dobrovsky.023@pslib.cz**

Obor **Informační Technologie**

Vedoucí práce **Ing. Tomáš Erlebach**

Školní rok **2024/2025**

Počet stran **4**

Počet slov **2528**



Anotace

Práce se zaměřuje na analýzu a porovnání dvou řadících algoritmů – Bublinkového řazení (Bubble Sort) a Rychlého řazení (Quick Sort). Cílem je zhodnotit jejich efektivitu vzhledem k velikosti a typu datových sad, s důrazem na časovou složitost a paměťovou náročnost.

Vychází z teoretických principů třídění, přičemž Bublinkové řazení je jednoduché, ale neefektivní u větších datových sad, zatímco Rychlé řazení je efektivnější díky metodě dělení dat.

Práce přináší výsledky testování na různých datových sadách, které ukazují rozdíly v výkonu obou algoritmů. Tyto výsledky pomohou při výběru vhodného algoritmu podle konkrétních potřeb.

Čestné prohlášení

Prohlašujeme, že jsme předkládaný ročníkový projekt vypracovali sami a uvedli jsme veškerou použitou literaturu a bibliografické citace.

V Liberci dne 31.03.2025

bohuslav.cermak.023@pslib.cz;vit.dobrovsky.023@pslib.cz

Obsah

[Úvod 1](#_Toc194269583)

[1 Analýza řadících algoritmů 2](#_Toc194269584)

[1.1 Rozčlenění algoritmu 2](#_Toc194269585)

[1.1.1 Vnější řazení 2](#_Toc194269586)

[1.1.2 Vnitřní řazení 2](#_Toc194269587)

[1.2 Parametry 3](#_Toc194269588)

[2 Teoretické základy řazení 5](#_Toc194269589)

[2.1 Bublinkové řazení (Bubble Sort) 5](#_Toc194269590)

[2.1.1 Princip použitého algoritmu 5](#_Toc194269591)

[2.1.2 Charakteristiky 6](#_Toc194269592)

[2.2 Rychlé řazení (Quick Sort) 6](#_Toc194269593)

[2.2.1 Princip použitého algoritmu 6](#_Toc194269594)

[2.2.2 Charakteristiky 8](#_Toc194269595)

[3 Testování výkonu 9](#_Toc194269596)

[3.1 Výběr datových sad 9](#_Toc194269597)

[3.1.1 Typy a velikosti testovacích datových sad 9](#_Toc194269598)

[3.1.2 Princip použitého algoritmu 9](#_Toc194269599)

[3.2 Měření výkonu 10](#_Toc194269600)

[3.2.1 Princip použitého algoritmu 10](#_Toc194269601)

[4 Výsledky a analýza 11](#_Toc194269602)

[4.1 Porovnání časové složitosti 11](#_Toc194269603)

[4.1.1 Benchmark 11](#_Toc194269604)

[4.1.2 Graf 12](#_Toc194269605)

[4.1.3 Princip použitého algoritmu 13](#_Toc194269606)

[4.2 Porovnání paměťové náročnosti 14](#_Toc194269607)

[4.2.1 Bubble sort 14](#_Toc194269608)

[4.2.2 Quick sort 14](#_Toc194269609)

[5 Závěr 15](#_Toc194269610)

[5.1 Výsledky testování 15](#_Toc194269611)

[5.2 Doporučení pro použití 15](#_Toc194269612)

[5.3 Celkové shrnutí 15](#_Toc194269613)

[Seznam zkratek a odborných výrazů 16](#_Toc194269614)

[Seznam obrázků 17](#_Toc194269615)

[Použité zdroje 18](#_Toc194269616)

[A. Seznam přiložených souborů I](#_Toc194269617)

Úvod

Existuje mnoho různých algoritmů pro řazení dat, každý s výhodami a nevýhodami podle konkrétních požadavků na výkon. Cílem této práce je porovnat dva klasické řadící algoritmy – Bublinkové řazení (*Bubble Sort*) a Rychlé řazení (*Quick Sort*). Tyto algoritmy byly vybrány pro jejich rozdílné charakteristiky, které umožní srovnání různých přístupů k řazení. (1) (2)

V práci bude využit benchmarking, metoda pro měření a porovnávání výkonu algoritmů za stejných podmínek. K testování budou použity různé datové sady, včetně náhodně seřazených a již předem uspořádaných souborů. Benchmarking poskytne přehled o časové složitosti a paměťové náročnosti obou algoritmů při různých velikostech dat. (3)

Cílem je porovnat efektivitu algoritmů a zjistit, jak jejich výkon závisí na velikosti a typu dat. Výsledky analýzy poslouží jako návod pro výběr vhodného algoritmu podle potřebných požadavků na rychlost a paměť.

# Analýza řadících algoritmů

Řadicí algoritmus uspořádává datové záznamy do požadovaného pořadí. Řazení je běžná úloha, důležitá pro mnoho dalších algoritmů. Vývoj efektivních algoritmů pro tuto úlohu vedl k vytvoření různých metod řazení. (1)

## Rozčlenění algoritmu

Algoritmy řazení se dělí na dvě hlavní skupiny: vnitřní a vnější.

Vnější řadicí algoritmy jsou určeny pro situace, kdy množství dat přesahuje kapacitu paměti počítače. Tyto algoritmy pracují s daty uloženými na pevném disku, kde nejsou data uložena v paměti, ale jsou postupně načítána, zpracována a opět ukládána. Vnější řazení se běžně používá pro třídění velkých databází nebo souborů.

Vnější řazení se obvykle dělí na různé kategorie podle metod, které využívají k efektivnímu třídění dat na disku.

### Vnější řazení

* **Přímé slučování**  
  Tento způsob řazení spočívá v opakovaném sloučení menších částí dat do větších seřazených bloků, dokud není celý soubor seřazen. Tento algoritmus se často používá při práci s velkými soubory uloženými na pevném disku. **(Direct Merging)**
* **Přirozené slučování**  
  Tento algoritmus využívá přirozených sekvencí v datech, kdy se nejprve identifikují již seřazené podsekvence a ty se následně sloučí. Tato metoda může být efektivní, pokud jsou data již částečně seřazena. **(Natural Merging)**
* **Vyvážené slučování**  
  Při vyváženém slučování se data dělí na rovnoměrné bloky, které se následně sloučí v optimálním pořadí. Tento způsob minimalizuje počet potřebných operací a zajišťuje efektivitu při třídění velkých souborů. **(Balanced Merging)**

### Vnitřní řazení

* **Řazení výběrem**

Tento algoritmus najde nejmenší (nebo největší) prvek v ne-seřazené části seznamu a vymění jej s prvkem na začátku této části. Tento proces opakuje, dokud není celý seznam seřazen. **(Selection Sort)**

* Řazení výměnou

Algoritmus porovnává sousední prvky a v případě potřeby je vymění, aby se seznam postupně dostal do seřazeného stavu. Nejznámějším příkladem je Bublinkové řazení. (Bubble Sort)

* **Řazení vkládáním**  
  Tento algoritmus postupně vkládá jednotlivé prvky na správné místo ve seřazené části seznamu. Tento proces je podobný tomu, jakým způsobem si člověk řadí karty při hře. **(Insertion Sort)**

## Parametry

**Časová složitost**:

* + Nejlepší případ: O(n log n) – pro vyvážené slučování, O(n) pro přirozené slučování (pokud jsou data již částečně seřazena).
  + **Nejhorší případ**:

O(n log n) pro oba typy.

**Paměťová složitost**:

O(n) – Slučování vyžaduje pomocnou paměť pro uchovávání dočasných dat při sloučení.

**Stabilita algoritmu**:

* + **Stabilní**: Přirozené a vyvážené slučování jsou stabilní algoritmy, což znamená, že zachovávají pořadí prvků se stejnou hodnotou.

**Paralelizace**:

* + **Paralelní výpočty**: Algoritmy slučování mohou být paralelizovány, zejména při slučování bloků, což může výrazně zrychlit jejich výkon.

**Závislost na vstupních datech**:

* + **Přirozené slučování**: Efektivnější, pokud jsou data již částečně seřazena, což může snížit dobu potřebnou pro třídění.
  + **Vyvážené slučování**: Vyžaduje více operací než přirozené, ale je stabilní a předvídatelné bez ohledu na vstupní data.

**Praktické využití**:

* + **Přirozené slučování**: Efektivní, pokud máte částečně seřazená data.
  + **Vyvážené slučování**: Efektivní pro velké soubory a v situacích, kdy je třeba udržet stabilitu při sloučení dat.

# Teoretické základy řazení

V této kapitole porovnáme dva řadící algoritmy – Bublinkové řazení (Bubble Sort) a Rychlé řazení (Quick Sort). Zaměříme se na několik klíčových parametrů, které ovlivňují jejich efektivitu při třídění dat.

## Bublinkové řazení (Bubble Sort)

Bublinkové řazení funguje tak, že prochází seznam a porovnává sousední prvky. Pokud jsou v nesprávném pořadí, vymění se. Tento proces se opakuje, dokud není seznam seřazený. (1)

### Princip použitého algoritmu

Kód implementuje bublinkové řazení, kde se porovnávají sousední prvky a vyměňují, pokud jsou v nesprávném pořadí. Tento proces probíhá opakovaně, dokud není seznam seřazen.

public static IEnumerable<T> Sort(IEnumerable<T> collection)

{

var list = collection.ToList();

for (int i = 0; i < list.Count - 1; i++)

{

for (int j = 0; j < list.Count - i - 1; j++)

{

if (list[j].CompareTo(list[j + 1]) > 0)

{

(list[j], list[j + 1]) = (list[j + 1], list[j]);

}

}

}

return list;

}

### Charakteristiky

|  |  |
| --- | --- |
| **Parametr** | **Popis** |
| **Časová složitost** | O(n²) (průměr, nejhorší), O(n) (nejlepší výsledek) |
| **Paměťová náročnost** | O(1) – konstantní množství paměti |
| **Stabilita** | Ano – zachovává pořadí stejné hodnoty |
| **Rychlost** | Pomalé pro velké datové sady |
| **Jednoduchost implementace** | Velmi jednoduché |
| **Použití** | Vhodné pro malé datové sady |

(1)

## Rychlé řazení (Quick Sort)

Rychlé řazení funguje na principu "rozděl a panuj". Nejprve se vybere **pivot**, prvek, který rozdělí seznam na dvě části. Jedna část obsahuje prvky menší než pivot, druhá větší. Tento proces se opakuje rekurzivně pro každou část, dokud nejsou všechny části seřazeny. (2)

**Efektivita**: Tento algoritmus je velmi efektivní pro velké datové sady díky složitosti **O(n log n)**, což je výrazně lepší než u některých jiných algoritmů, jako je Bublinkové řazení. (2)

### Princip použitého algoritmu

Kód rozdělí pole na části pomocí pivotu a rekurzivně seřadí levé a pravé části pole. Proces pokračuje, dokud jsou všechny části seřazeny.

public static T[] Sort(T[] arr)

{

T[] result = new T[arr.Length];

Array.Copy(arr, result, arr.Length);

QuickSortAlgorithm(result, 0, result.Length - 1);

return result;

}

Tento kód rozdělí pole na části pomocí pivotu a rekurzivně seřadí levé a pravé části. Tento proces pokračuje, dokud jsou všechny části správně seřazeny.

private static void QuickSortAlgorithm(T[] arr, int low, int high)

{

if (low < high)

{

int pivotIndex = Partition(arr, low, high);

QuickSortAlgorithm(arr, low, pivotIndex - 1);

QuickSortAlgorithm(arr, pivotIndex + 1, high);

}

}

Tento kód rozdělí pole na dvě části podle pivota, přičemž prvky menší než pivot jsou umístěny vlevo a větší vpravo, a vrátí index pivota pro další rekurzivní dělení.

private static int Partition(T[] arr, int low, int high)

{

T pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++)

{

if (arr[j].CompareTo(pivot) <= 0)

{

i++;

Swap(arr, i, j);

}

}

Swap(arr, i + 1, high);

return i + 1;

}

Tento kód vymění hodnoty dvou prvků v poli na pozicích i a j pomocí dočasné proměnné.

private static void Swap(T[] arr, int i, int j)

{

T temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

### Charakteristiky

|  |  |
| --- | --- |
| **Parametr** | **Popis** |
| **Časová složitost** | O(n log n) (průměr, nejlepší), O(n²) (nejhorší) |
| **Paměťová náročnost** | O(log n) – požaduje stack pro rekurzi |
| **Stabilita** | Ne – nezachovává pořadí stejné hodnoty |
| **Rychlost** | Relativně složitější než Bubble Sort |
| **Jednoduchost implementace** | Vhodné pro velké datové sady |
| **Použití** | Vhodné pro malé datové sady |

(2)

# Testování výkonu

V této kapitole se zaměříme na testování Bublinkového (Bubble Sort) a Rychlého řazení (Quick Sort) na různých datových sadách, včetně analýzy jejich výkonu na různých typech dat (náhodně seřazená, již seřazená).

## Výběr datových sad

### Typy a velikosti testovacích datových sad

* 1. Náhodná drobná (10)
  2. Srovnaná drobná (10)
  3. Náhodná malá (100)
  4. Srovnaná malá (100)
  5. Náhodná střední (1000)
  6. Srovnaná střední (1000)
  7. Náhodná velká (10000)
  8. Srovnaná velká (10000)
  9. Náhodná velmi velká (100000)
  10. Srovnaná velmi velká (100000)
  11. Náhodná gigantická (1000000)
  12. Srovnaná gigantická (1000000)

### Princip použitého algoritmu

//Little

Element[] randomLittle = CollectionFactory.GenerateShuffledArray(10);

Element[] sortedLittle = CollectionFactory.GenerateOrderedArray(10);

//Low

Element[] randomLow = CollectionFactory.GenerateShuffledArray(100);

Element[] sortedLow = CollectionFactory.GenerateOrderedArray(100);

//Medium

Element[] randomMedium = CollectionFactory.GenerateShuffledArray(1000);

Element[] sortedMedium = CollectionFactory.GenerateOrderedArray(1000);

//Large

Element[] randomLarge = CollectionFactory.GenerateShuffledArray(10000);

Element[] sortedLarge = CollectionFactory.GenerateOrderedArray(10000);

//Extra large

Element[] randomExtraLarge = CollectionFactory.GenerateShuffledArray(100000);

Element[] sortedExtraLarge = CollectionFactory.GenerateOrderedArray(100000);

//Gigantic

Element[] randomGigantic = CollectionFactory.GenerateShuffledArray(1000000);

Element[] sortedGigantic = CollectionFactory.GenerateOrderedArray(1000000);

## Měření výkonu

Měření výkonu obou řadících algoritmů bylo provedeno pomocí nástroje **BenchmarkDotNet**, který nám umožnil porovnat časovou složitost a paměťovou náročnost při práci s různými datovými sadami. (3)

### Princip použitého algoritmu

//Little

[Benchmark]

public void RandomLittleBubbleSort()

{

BubbleSort<Element>.Sort(randomLittle);

}

# Výsledky a analýza

V této kapitole budou prezentovány výsledky měření a analýza výkonu obou algoritmů.

## Porovnání časové složitosti

Porovnání těchto dvou algoritmů se zaměří na různé velikosti datových sad, od malých (10 až 100 prvků) až po velké (až 1 milion prvků), aby ukázalo, jak se liší jejich časová složitost. Výsledky budou podpořeny grafy, které zobrazí, jak časová složitost roste u každého algoritmu s rostoucí velikostí datové sady. Na základě těchto dat bude možné vybrat vhodný algoritmus pro specifické případy a analyzovat vliv velikosti dat na výkon.

### Benchmark

#### Little – data (10)

Obsah obrázku snímek obrazovky, text, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 1 little-tests

#### Low – data (100)

Obsah obrázku snímek obrazovky, text, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 2 low-tests

#### Medium – data (1,000)

Obsah obrázku snímek obrazovky, text

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 3 medium-tests

#### Large – data (10,000)

Obsah obrázku text, snímek obrazovky, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 4 large-tests

#### Extra large - data (100,000)

Obsah obrázku snímek obrazovky, text, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.

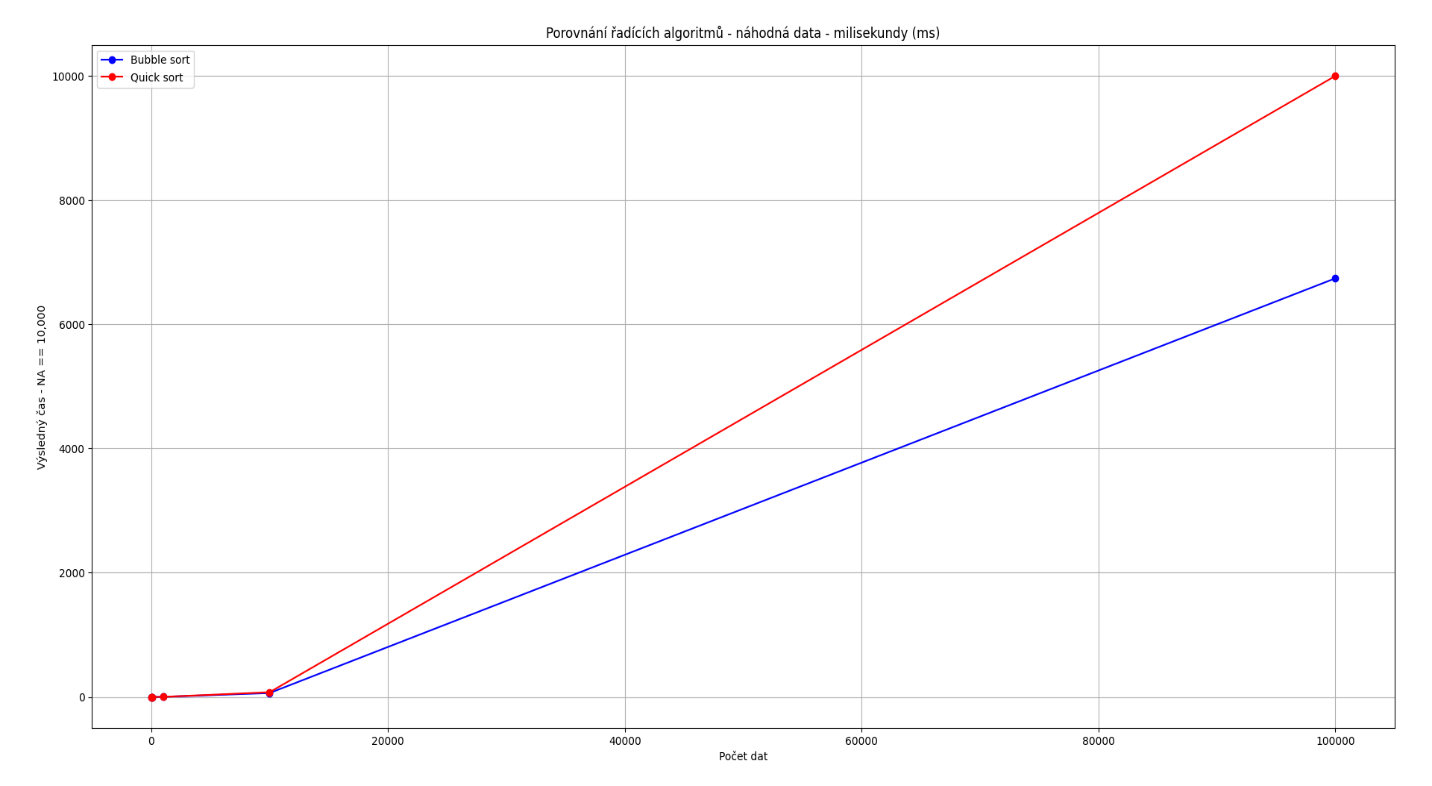
Obrázek 5 extra-large-tests

#### Gigantic – data (1,000,000)

**NA**

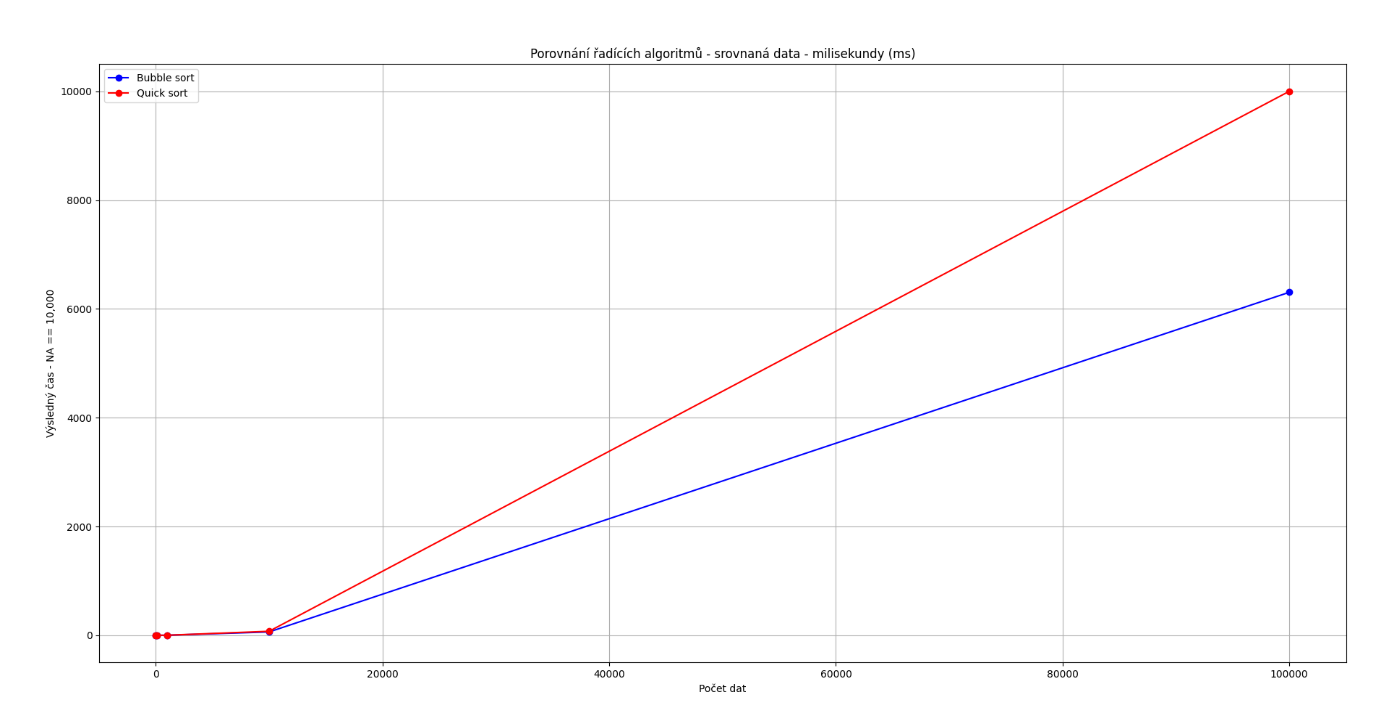
### Graf

#### Náhodná data



Obrázek 6 graph\_random-data

#### Srovnaná data



Obrázek 7 graph\_sorted-data

### Princip použitého algoritmu

import matplotlib.pyplot as plt

def create\_your\_graph(x\_array: list[any], y1\_array: list[any], y2\_array: list[any], graph\_title: str = "Porovnání řadících algoritmů", y1\_title: str = "Bubble sort", y2\_title: str = "Quick sort", x\_title: str = "Počet dat", y\_title: str = "Výsledný čas - NA == 10,000") -> None:

plt.plot(x\_array, y1\_array, marker = "o", linestyle = "-", color = "b", label = y1\_title)

plt.plot(x\_array, y2\_array, marker = "o", linestyle = "-", color = "r", label = y2\_title)

plt.xlabel(x\_title)

plt.ylabel(y\_title)

plt.title(graph\_title)

plt.legend()

plt.grid()

plt.show()

#create\_your\_graph([10, 100, 1000, 10000, 100000], [0.00010597, 0.007002, 0.6364, 62.97, 6741], [0.00013150, 0.008637, 0.7658, 75.7, 10000], "Porovnání řadících algoritmů - náhodná data - milisekundy (ms)")

#create\_your\_graph([10, 100, 1000, 10000, 100000], [0.00009895, 0.006824, 0.6265, 61.75, 6305], [0.00012966, 0.008429, 0.7472, 73.93, 10000], "Porovnání řadících algoritmů - srovnaná data - milisekundy (ms)")

## Porovnání paměťové náročnosti

### Bubble sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Množství dat | 10 | 100 | 1,000 | 10000 | 100,000 | 1,000,000 |
| Paměť (Kb) | 0.211 | 1.62 | 15.68 | 156.37, 156.38 | 1564.32 | NA |

### Quick sort

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Množství dat | 10 | 100 | 1,000 | 10000 | 100,000 | 1,000,000 |
| Paměť (Kb) | 0.1797 | 1.59 | 15.65 | 156.36 | NA | NA |

# Závěr

## Výsledky testování

**Časová efektivita:** Bublinkové řazení (Bubble Sort) bylo ve všech testovaných případech rychlejší než Rychlé řazení (Quick Sort). Dokonce projevilo i ve větších data lepší výsledky co se týče splnění než Quick Sort.

**Paměťová náročnost:** Quick Sort byl ve všech případech úspornější na paměť než Bubble Sort.

## Doporučení pro použití

**Kdy použít Bubble Sort?**

* Pro malé datové sady (do 1000 prvků).
* Pokud je potřeba jednoduchý a stabilní algoritmus.
* Pokud není prioritou paměťová efektivita.

**Kdy použít Quick Sort?**

* Na základě tohoto měření Quick Sort nedoporučujeme, protože nedokázal správně seřadit čísla nad 1 000 000 prvků a byl pomalejší než Bubble Sort ve všech testech.
* Pokud je nutné řadit větší množství dat, je vhodné zvážit jiný algoritmus (např. Merge Sort nebo Heap Sort).
* Pokud je důležitá paměťová úspornost, ale zároveň je možné optimalizovat implementaci.

## Celkové shrnutí

Testy ukázaly, že Bubble Sort byl vždy rychlejší než Quick Sort a úspěšně seřadil všechny testované datové sady. Quick Sort měl problémy s efektivitou, zejména při řazení dat nad 1 000 000 prvků, což ani jeden z algoritmů nezvládl. Bubble Sort je spolehlivější, ale není vhodný pro velké objemy dat kvůli nízké efektivitě. Quick Sort by mohl být výhodný pro paměťovou efektivitu, ale v testech neprokázal lepší výkon. Výběr algoritmu závisí na konkrétních požadavcích: pro spolehlivost a stabilitu je lepší Bubble Sort, pro menší datové sady.

Seznam zkratek a odborných výrazů

**Bubble Sort (Bublinkové řazení)** – Jednoduchý řadící algoritmus, který opakovaně prochází seznam a porovnává sousední prvky, přičemž je v případě potřeby mění. (1)

**Quick Sort (Rychlé řazení)** – Efektivní rekurzivní řadící algoritmus, který používá metodu dělení pole podle pivotu a rekurzivně řadí podsekce. (2)

**Benchmarking** – Proces měření a porovnávání výkonu algoritmů nebo systémů za účelem vyhodnocení jejich efektivity. (3)

**Časová složitost** – Měří, jak se čas potřebný k vykonání algoritmu mění v závislosti na velikosti vstupních dat. (1)

**Paměťová složitost** – Měří, jaké množství paměti algoritmus potřebuje v závislosti na velikosti vstupních dat. (1)

**Rekurze** – Technika v programování, kdy funkce volá sama sebe, aby vyřešila podúkoly. (2)

**Pivot** – Prvek v algoritmu Quick Sort, podle kterého se dělí seznam na menší a větší hodnoty, čímž se řadí. (2)

**Složitost O(n²)** – Časová složitost algoritmu, kdy doba běhu roste s kvadrátem velikosti vstupních dat (typické pro Bubble Sort). (1)

**Složitost O(n log n)** – Časová složitost algoritmu, kdy doba běhu roste s násobkem velikosti dat a logaritmické hodnoty, charakteristická pro Quick Sort v průměrném případě. (2)

**Stabilita algoritmu** – Schopnost algoritmu zachovat pořadí stejných prvků v seznamu, pokud mají stejné hodnoty. (1)

**Swap** – Operace, při které dojde k výměně hodnot mezi dvěma položkami v seznamu. (2)

**Array (Pole)** – Datová struktura, která ukládá hodnoty ve formě seznamu s pevně stanovenou velikostí. (2)

**IEnumerable** – Rozhraní v C#, které představuje kolekci, která může být procházená (iterovatelná).

Seznam obrázků

[Obrázek 1 little-tests 11](#_Toc194305791)

[Obrázek 2 low-tests 11](#_Toc194305792)

[Obrázek 3 medium-tests 11](#_Toc194305793)

[Obrázek 4 large-tests 12](#_Toc194305794)

[Obrázek 5 extra-large-tests 12](#_Toc194305795)

[Obrázek 6 graph\_random-data 12](#_Toc194305796)

[Obrázek 7 graph\_sorted-data 13](#_Toc194305797)

Použité zdroje1. **autorů, Kolektiv.** Bublinkové řazení. *Bublinkové řazení - Wikipedie.* [Online] Nadace Wikimedia, 12. 1 2025. [Citace: 30. 3 2025.] https://cs.wikipedia.org/wiki/Bublinkov%C3%A9\_%C5%99azen%C3%AD.

2. —. Rychlé řazení. *Rychlé řazení - Wikipedie.* [Online] Nadace Wikimedia, 9. 6 2024. [Citace: 30. 3 2025.] https://cs.wikipedia.org/wiki/Rychl%C3%A9\_%C5%99azen%C3%AD.

3. —. Benchmarking. *Benchmarking - Wikipedie.* [Online] Nadace Wikimedia, 12. 3 2025. [Citace: 30. 3 2025.] https://cs.wikipedia.org/wiki/Benchmarking.

4. **Matplotlib, The.** Matplotlib 3.10.1 documentation. *Matplotlib — Visualization with Python.* [Online] The Matplotlib, 2012. [Citace: 29. 3 2025.] https://matplotlib.org/stable/.

1. Seznam přiložených souborů

Na přiloženém datovém nosiči se nacházejí následující soubory a složky:

* **2024-p2a-prg-sortingcomparsion-cermakdobrochlap** – složka s kompletní dokumentací a soubory ročníkového projektu