Gymnázium, Golianova 68, Nitra

Ročníková práca: Triedenie lineárnych štruktúr

Jaroslav Rojík

IV.AJ

2023/2024

Úvod

Triedenie je základná operácia v informatike a zohráva kľúčovú úlohu v rôznych aplikáciách, od jednoduchých úloh správy údajov až po zložité algoritmické problémy. V jeho podstate triedenie znamená usporiadanie kolekcie položiek podľa určitých kritérií.

V tejto ročníkovej práci sa budem zaoberať triediacimi algoritmami. Vysvetlím, čo sú to triediace algoritmy, prečo sú dôležité a aké sú ich základné vlastnosti. Taktiež sa pozriem na niektoré z najznámejších triediacich algoritmov a porovnám ich efektívnosť a zložitosť.

Dôvod tvorby projektu

Triediace algoritmy sú základnou súčasťou počítačovej vedy a majú široké uplatnenie v rôznych oblastiach informatiky. Vytvorenie projektu o metódach triedenia umožňuje hlbšie porozumenie základným algoritmom a dátovým štruktúram, čím sa zvýšia zručnosti pri riešení problémov. Poskytuje praktické skúsenosti s implementáciou a testovaním rôznych triediacich algoritmov, čo vedie ku komplexnému pochopeniu ich účinnosti a vhodnosti pre rôzne scenáre.

Metodika práce

1. Vytvorenie testovacej funkcie
2. Vytvorenie jednotlivých triediacich algoritmov
3. Vytvorenie vizualizačnej funkcie
4. Vizualizácia jednotlivých triediacich algoritmov
5. Vytvorenie porovnávacej funkcie
6. Porovnanie jednotlivých triediacich algoritmov

Stanovenie cieľa

Cieľom práce je porovnať a analyzovať niektoré z najznámejších triediacich algoritmov a zistiť, ktorý z nich je najefektívnejší a najvhodnejší pre rôzne typy dát a veľkosti problémov.

Stanovenie hypotézy

Pri porovnávaní triediacich algoritmov sa často skúma ich časová a priestorová zložitosť. Napríklad, Bubblesort má časovú zložitosť O(n^2), čo sa pri veľkých poliach neškáluje dobre. Quicksort má priemernú časovú zložitosť O(n log n), ale najhorší prípad má časovú zložitosť O(n^2). Platí aj to, že

Mergesort má časovú zložitosť O(n log n) pre všetky prípady, ale vyžaduje viac pamäte. Na základe týchto informácií sa dá predpokladať, že Mergesort a Quicksort budú najefektívnejšie.

Dôležitosť triedenia

Význam triediacich algoritmov vyplýva z ich všadeprítomného použitia v každodennom živote a ich nevyhnutnej úlohy v počítačovom programovaní a vývoji softvéru. Tu sú niektoré kľúčové dôvody, prečo je triedenie dôležité:

1. Organizácia dát: Triedenie umožňuje organizáciu dát v štruktúrovanej forme, čím uľahčuje ich rýchlejšie vyhľadávanie, získavanie a analýzu.
2. Získavanie informácií: Triedené dáta uľahčujú rýchlejšie vyhľadávanie a získavanie informácii. Vyhľadávanie v triedenom zozname je efektívnejšie, pretože môžeme použiť vyhľadávacie algoritmy ako napríklad binárne hľadanie.

Rozsah triediacich algoritmov

Triediace algoritmy sa líšia v zložitosti, efektívnosti a vhodnosti pre rôzne typy dát a veľkosti problémov. Môžu byť klasifikované podľa ich základných princípov, ako aj podľa ich časovej a priestorovej zložitosti.

Existujú rôzne triediace algoritmy, od jednoduchých a intuitívnych metód ako Bubble sort a Insert sort po sofistikovanejšie metódy ako Merge sort, Quick sort atď.

Typy triediacich algoritmov

Porovnávacie triediace algoritmy sú najbežnejším typom a pracujú porovnávaním jednotlivých prvkov vstupnej kolekcie a ich postupným presúvaním do správneho poradia. Tu sú niektoré z najznámejších porovnávacích triediacich algoritmov na ktoré sa pozriem v tejto práci:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Testovacia funkcia

```
import random

generated_data = [random.randint(1, 100) for _ in range(30)]


def test(func):
  data = generated_data.copy()
```

```
  print(data)
  func(data)
  print(data)
```

Bubble Sort

Bubble Sort je jednoduchý algoritmus, ktorý opakovane prechádza cez zoznam a porovnáva susedné prvky. Ak sú v nesprávnom poradí, vymenia sa. Tento proces sa opakuje, kým nie sú všetky prvky usporiadané.

Tento algoritmus má časovú zložitosť O(n^2). Je vhodný pre malé zoznamy ale nie je efektívny pre veľké množstvo dát.

```
def bubble_sort(arr):
  n = len(arr)
  # swapped = False
  for i in range(n - 1):
    for j in range(0, n - i - 1):
      if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
    #     swapped = True
    # if not swapped:
    #   break

test(bubble_sort)
```

```
[12, 30, 32, 99, 22, 48, 9, 75, 87, 55, 81, 33, 90, 70, 70, 78, 46, 99, 9, 17, 9
9, 60, 68, 75, 60, 77, 80, 70, 70, 95]
[9, 9, 12, 17, 22, 30, 32, 33, 46, 48, 55, 60, 60, 68, 70, 70, 70, 70, 75, 75, 7
7, 78, 80, 81, 87, 90, 95, 99, 99, 99]
```

Selection Sort

Selection Sort funguje tak, že v každom kroku nájde najmenší nepremiestnený prvok a presunie ho na správnu pozíciu. Tento proces sa opakuje, kým nie je celý zoznam usporiadaný.

Časová zložitosť tohto algoritmu je tiež O(n^2), ale môže byť v praxi efektívnejší ako bublinkové triedenie pre veľké zoznamy, pretože má menej výmen.

```
def selection_sort(arr):
  for step in range(len(arr)):
    min_index = step

    for i in range(step + 1, len(arr)):
      if arr[i] < arr[min_index]:
        min_index = i
    arr[step], arr[min_index] = arr[min_index], arr[step]

test(selection_sort)
```

```
[12, 30, 32, 99, 22, 48, 9, 75, 87, 55, 81, 33, 90, 70, 70, 78, 46, 99, 9, 17, 9
9, 60, 68, 75, 60, 77, 80, 70, 70, 95]
[9, 9, 12, 17, 22, 30, 32, 33, 46, 48, 55, 60, 60, 68, 70, 70, 70, 70, 75, 75, 7
7, 78, 80, 81, 87, 90, 95, 99, 99, 99]
```

Insertion Sort

Insertion Sort funguje tak, že postupne vkladá prvky z neusporiadanej časti zoznamu do už usporiadanej časti zoznamu tak, aby výsledný zoznam bol usporiadaný. Tento proces sa opakuje, kým nie sú všetky prvky vložené do usporiadanej časti.

Tento algoritmus má tiež časovú zložitosť O(n^2), ale je efektívnejší pre zoznamy, ktoré sú už čiastočne usporiadané.

```
def insertion_sort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    j = i - 1
    while j >= 0 and key < arr[j]:
      arr[j + 1] = arr[j]
      j -= 1
    arr[j + 1] = key

test(insertion_sort)
```

```
[12, 30, 32, 99, 22, 48, 9, 75, 87, 55, 81, 33, 90, 70, 70, 78, 46, 99, 9, 17, 9
9, 60, 68, 75, 60, 77, 80, 70, 70, 95]
[9, 9, 12, 17, 22, 30, 32, 33, 46, 48, 55, 60, 60, 68, 70, 70, 70, 70, 75, 75, 7
7, 78, 80, 81, 87, 90, 95, 99, 99, 99]
```

Merge Sort

Merge Sort je efektívny algoritmus, ktorý využíva princíp rozdelenia a zlúčenia. Postupne delí zoznam na polovice, triedi ich a potom zlúči do výsledného usporiadaného zoznamu.

Tento algoritmus má časovú zložitosť O(n log n), čo ho robí efektívnym pre veľké zoznamy. Jeho hlavnou nevýhodou je potreba dodatočnej pamäte pre zlúčenie podzoznamov.

```
def merge_sort(arr):
  if len(arr) > 1:
    mid = len(arr) // 2
    L = arr[:mid]
    R = arr[mid:]
    merge_sort(L)
    merge_sort(R)
    i = j = k = 0
    while i < len(L) and j < len(R):
      if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
      else:
        arr[k] = R[j]
        j += 1
      k += 1
    while i < len(L):
      arr[k] = L[i]
      i += 1
      k += 1
    while j < len(R):
      arr[k] = R[j]
      j += 1
      k += 1

test(merge_sort)
```

```
[12, 30, 32, 99, 22, 48, 9, 75, 87, 55, 81, 33, 90, 70, 70, 78, 46, 99, 9, 17, 9
9, 60, 68, 75, 60, 77, 80, 70, 70, 95]
[9, 9, 12, 17, 22, 30, 32, 33, 46, 48, 55, 60, 60, 68, 70, 70, 70, 70, 75, 75, 7
7, 78, 80, 81, 87, 90, 95, 99, 99, 99]
```

Quick Sort

Quick Sort je založené na princípe rozdelenia a zjednotenia podľa pivotu. Zoznam je rozdelený na dve podmnožiny na základe pivotu a potom sú obe podmnožiny triedené rekurzívne.

Tento algoritmus má priemernú časovú zložitosť O(n log n) a je často jedným z najrýchlejších triediacich algoritmov. Jeho efektivita však môže klesať v prípade, že je zoznam veľmi nevyvážený.

```
def quick_sort(array, low, high):
  if low < high:
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
      if array[j] <= pivot:
        i += 1
        array[i], array[j] = array[j], array[i]

    array[i + 1], array[high] = array[high], array[i + 1]
    pi = i + 1
    quick_sort(array, low, pi - 1)
    quick_sort(array, pi + 1, high)

test(lambda arr: quick_sort(arr, 0, len(arr) - 1))
```

```
[12, 30, 32, 99, 22, 48, 9, 75, 87, 55, 81, 33, 90, 70, 70, 78, 46, 99, 9, 17, 9
9, 60, 68, 75, 60, 77, 80, 70, 70, 95]
[9, 9, 12, 17, 22, 30, 32, 33, 46, 48, 55, 60, 60, 68, 70, 70, 70, 70, 75, 75, 7
7, 78, 80, 81, 87, 90, 95, 99, 99, 99]
```

Vizualizácia triediacich algoritmov

Vizualizácia triediacich algoritmov je užitočný nástroj na pochopenie ich fungovania a porovnanie ich efektivity.

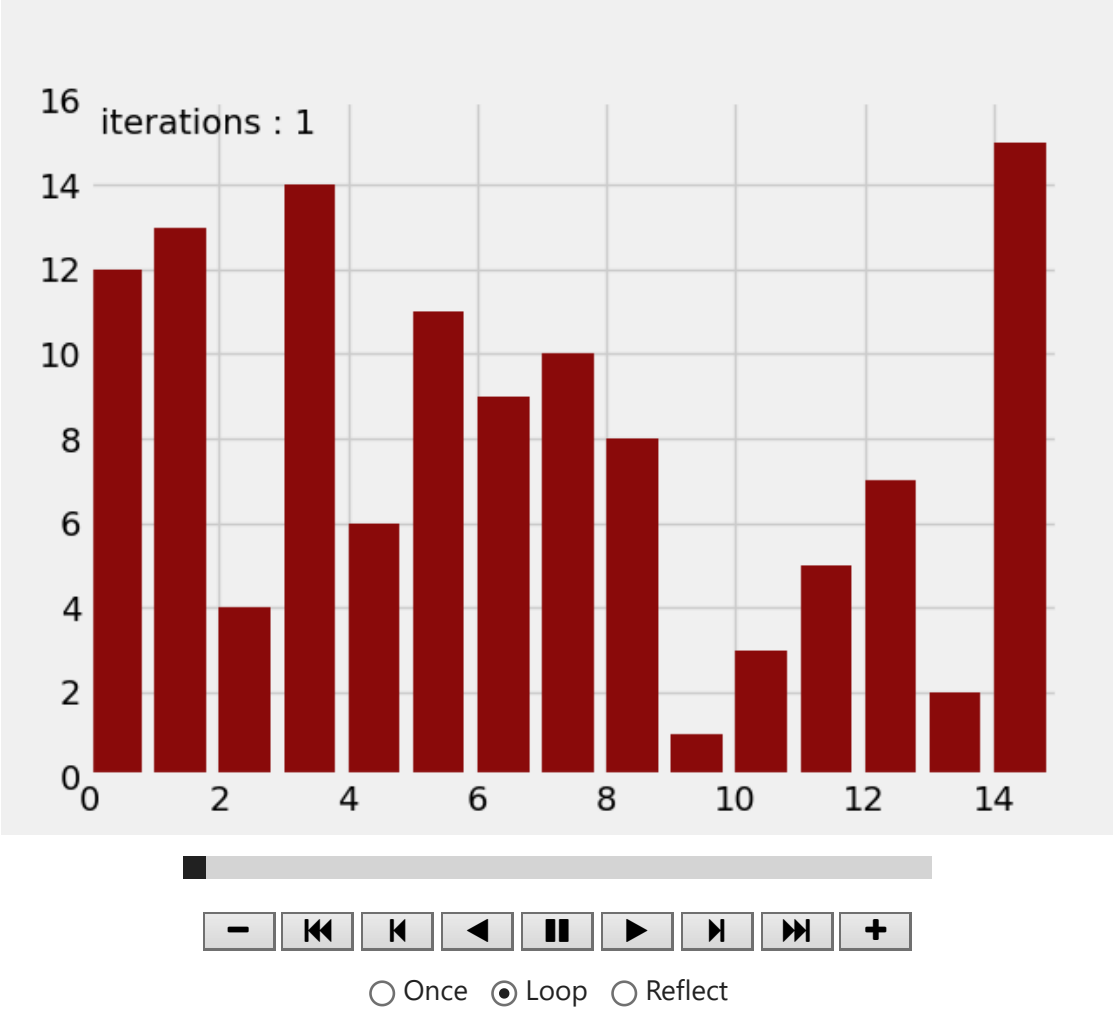
```
from matplotlib.animation import FuncAnimation
import matplotlib.pyplot as plt
import matplotlib as mp
import random

n = 15
delay = 100
arr = [i for i in range(1, n + 1)]
random.shuffle(arr)

def show(generator):
  data = []
  for i in generator(arr.copy()):
    data.append(i.copy())

  plt.style.use('fivethirtyeight')
  plt.rcParams['animation.html'] = 'jshtml'
  # plt.rcParams["figure.dpi"] = 150
```

```
  # plt.ioff()

  fig, ax = plt.subplots()
  rects = ax.bar(range(n), arr, align='edge', color=(0.54, 0.04, 0.04))

  ax.set_xlim(0, n)
  ax.set_ylim(0, int(1.1 * n))

  text = ax.text(0.01, 0.95, '', transform=ax.transAxes)

  iteration = [0]

  def animate(A, rects, iteration):
    for rect, val in zip(rects, A):
      rect.set_height(val)

    text.set_text(f'iterations : {iteration[0]}')
    iteration[0] += 1

  return FuncAnimation(
    fig,
    func=animate,
    fargs=(rects, iteration),
    frames=data,
    interval=delay,
  )
```

Vizualizácia Bubble Sortu

```
def bubblesort(arr):
  n = len(arr)
  for i in range(n - 1):
    swapped = False
    for j in range(0, n - i - 1):
      if arr[j] > arr[j + 1]:
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
        swapped = True
        yield arr
    if swapped == False:
      break

show(bubblesort)
```

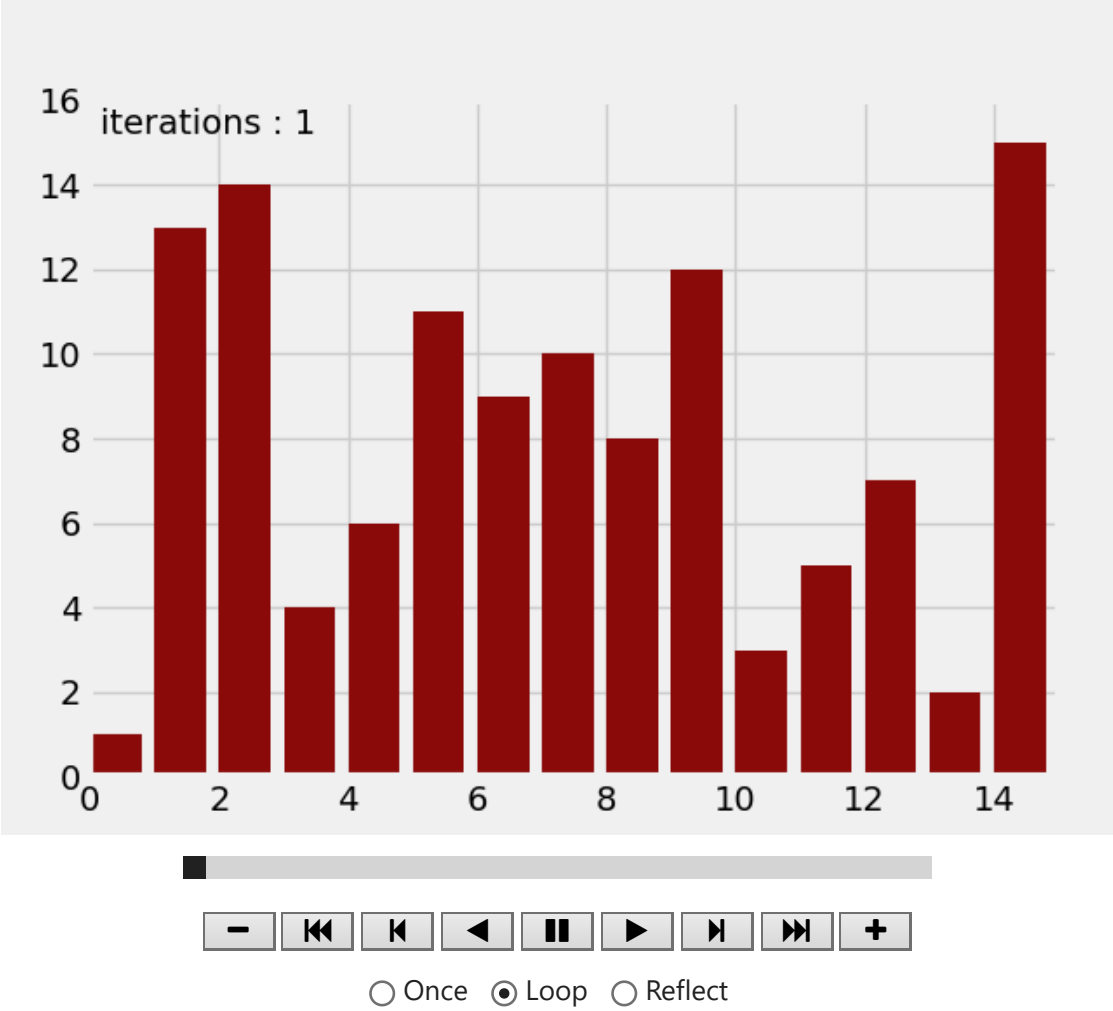
Out[ ]:



Vizualizácia Selection Sortu

```
def selectionsort(arr):
  for step in range(len(arr)):
    min_idx = step

    for i in range(step + 1, len(arr)):
      if arr[i] < arr[min_idx]:
        min_idx = i
    (arr[step], arr[min_idx]) = (arr[min_idx], arr[step])
    yield arr

show(selectionsort)
```

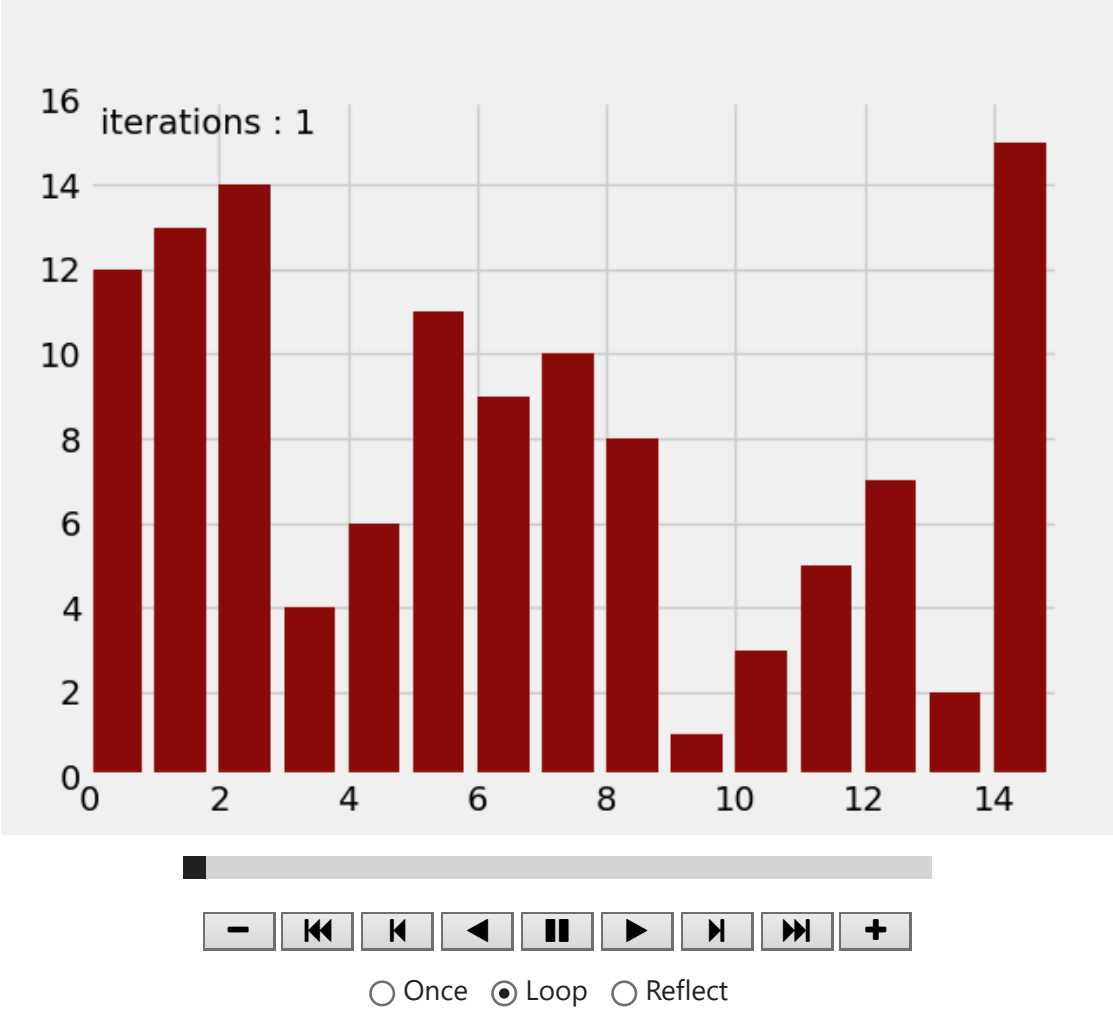
Out[ ]:



Vizualizácia Insertion Sortu

```
def insertionsort(arr):
  for j in range(1, len(arr)):
    key = arr[j]
    i = j - 1

    while i >= 0 and arr[i] > key:
      arr[i + 1] = arr[i]
      i -= 1
      yield arr
    arr[i + 1] = key
    yield arr

show(insertionsort)
```

Out[ ]:

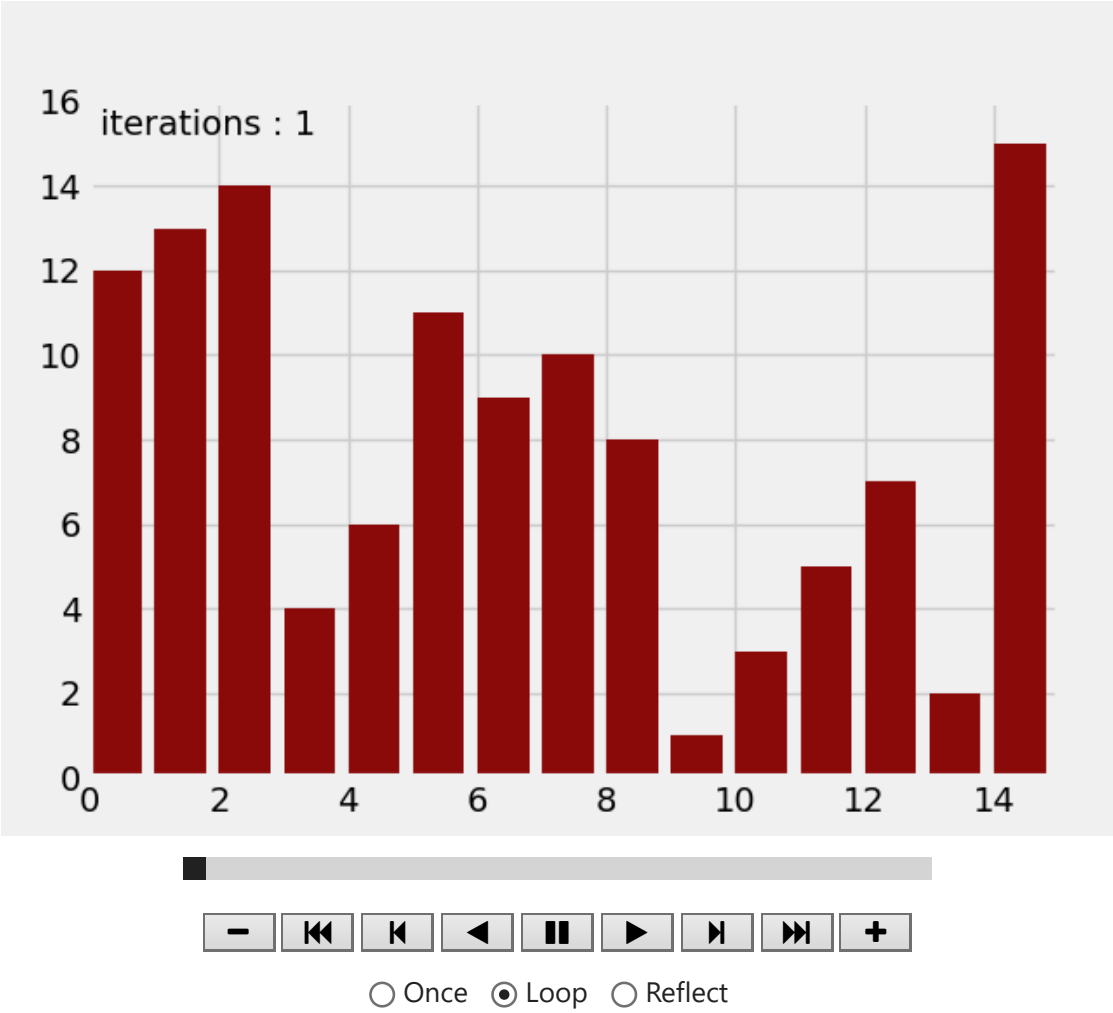


Vizualizácia Merge Sortu

```
def mergesort(A, start, end):
  if end <= start:
    return

  mid = start + ((end - start + 1) // 2) - 1

  yield from mergesort(A, start, mid)
  yield from mergesort(A, mid + 1, end)
  yield from merge(A, start, mid, end)


def merge(A, start, mid, end):
  merged = []
  leftIdx = start
  rightIdx = mid + 1

  while leftIdx <= mid and rightIdx <= end:
    if A[leftIdx] < A[rightIdx]:
      merged.append(A[leftIdx])
      leftIdx += 1
    else:
      merged.append(A[rightIdx])
      rightIdx += 1

  while leftIdx <= mid:
    merged.append(A[leftIdx])
    leftIdx += 1
```

```
  while rightIdx <= end:
    merged.append(A[rightIdx])
    rightIdx += 1

  for i in range(len(merged)):
    A[start + i] = merged[i]
    yield A


show(lambda a: mergesort(a, 0, len(a) - 1))
```

Out[ ]:

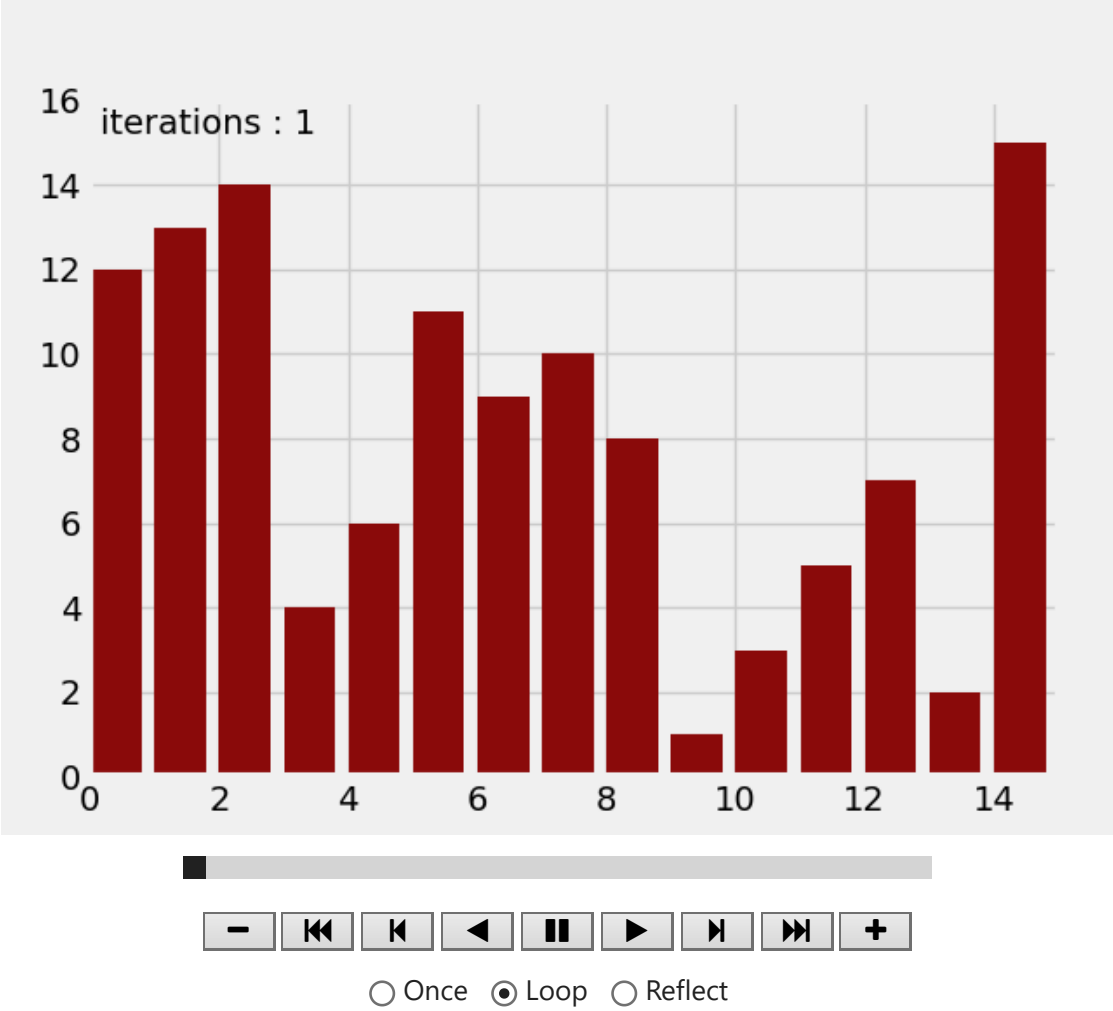


Vizualizácia Quick Sortu

```
def quicksort(a, l, r):
  if l >= r:
    return
  x = a[l]
  j = l
  for i in range(l + 1, r + 1):
    if a[i] <= x:
      j += 1
      a[j], a[i] = a[i], a[j]
    yield a
  a[l], a[j] = a[j], a[l]
  yield a

  yield from quicksort(a, l, j - 1)
  yield from quicksort(a, j + 1, r)
```

```
show(lambda a: quicksort(a, 0, len(a) - 1))
```

Out[ ]:



Benchmarking a porovnanie

Benchmarking je dôležitý proces pri hodnotení výkonnosti triediacich algoritmov. Pomáha porovnať rôzne algoritmy podľa ich efektivity a vhodnosti pre konkrétne použitie.

Úvod do benchmarkingu

Benchmarking zahŕňa vykonávanie rôznych experimentov s triediacimi algoritmami a zaznamenávanie ich výkonnosti podľa rôznych metrík, ako sú čas trvania triedenia, pamäťové nároky a stabilita algoritmu.

Návrh benchmarkových experimentov

Dôležitou súčasťou benchmarkingu je správne navrhnutie experimentov. To zahŕňa vytvorenie rôznych testovacích sád s rôznymi typmi vstupných dát, ako aj rôznymi veľkosťami vstupov.

Implementácia benchmarkingu

Po navrhnutí experimentov je potrebné implementovať benchmarkingový kód, ktorý spustí rôzne triediace algoritmy na testovacích dátach a zaznamená ich výsledky.

Porovnanie výsledkov

Po vykonaní benchmarkingových experimentov je čas na porovnanie výsledkov a zhodnotenie výkonnosti jednotlivých algoritmov. Pri porovnávaní algoritmov by sa malo zohľadniť niekoľko faktorov, vrátane:

- Čas triedenia.
- Pamäťové nároky.
- Stabilita a robustnosť.

Implementácia

Importy

```
import random
import time
from statistics import mean
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
```

Nastavenie testovacích dát

```
velkosti_testovacich_sad = [x for x in range(0, 3200, 200)]
pocet_opakovani_pre_jednu_velkost = 1
```

Vytvorenie dataframe-u a testovacích dát

```
df = pd.DataFrame()
df['Velkost sady'] = velkosti_testovacich_sad
test_arrays = {i: [random.randint(1, 5000) for _ in range(i)] for i in velkosti_testovacich
```

Visualizačná funkcia

```
def visualise(name):
  global df
  # plt.figure(figsize=(10, 7))
  plt.plot(df['Velkost sady'], df[name], label=name)
  plt.legend()
  plt.title('Porovnanie časových náročností rôznych triedianích algoritmov')
  plt.xlabel('Veľkosť sady (n)')
  plt.ylabel('Čas (s)')
  plt.show()
```

Benchmarkovacia funkcia

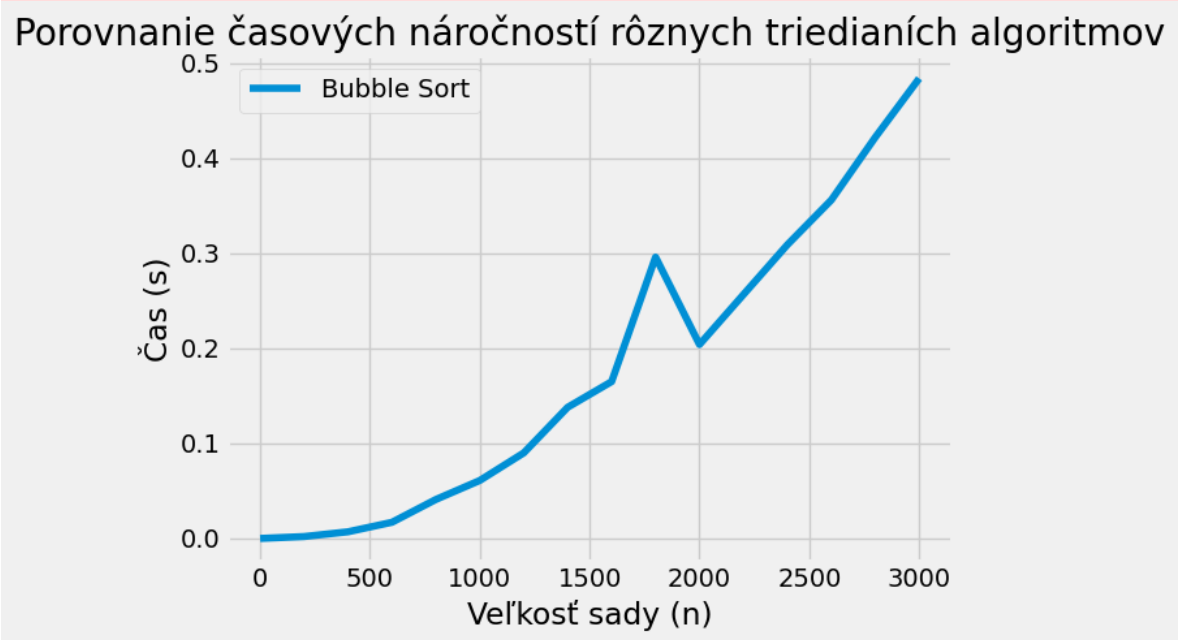
```
def benchmark(func, name):
  global velkosti_testovacich_sad, test_arrays, df

  times = []
  temp = []
  print(f'Benchmark pre {name}')
  for i in tqdm(velkosti_testovacich_sad, desc='Prebieha benchmarkovanie:'):
```

```
    for _ in range(pocet_opakovani_pre_jednu_velkost):
      start = time.time()
      func(test_arrays[i].copy())
      end = time.time()
      temp.append(end - start)
    times.append(mean(temp))
    temp = []
  df[name] = times
  visualise(name)
```

Benchmark Bubble Sortu

```
benchmark(bubble_sort, 'Bubble Sort')
```

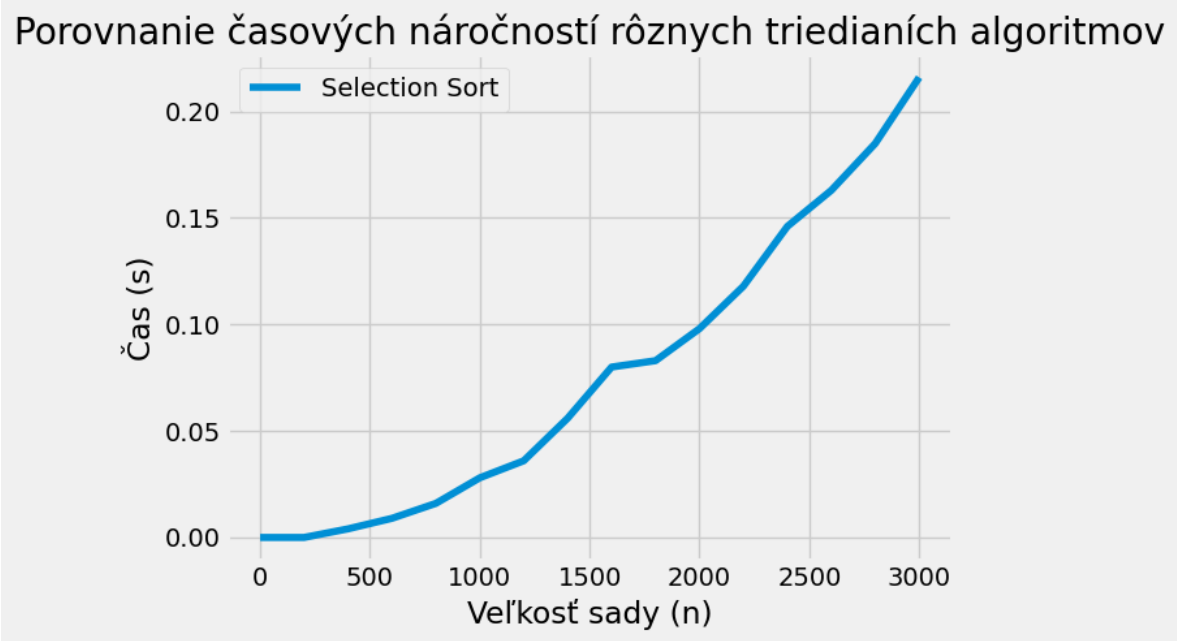
```
Benchmark pre Bubble Sort
Prebieha benchmarkovanie:: 100%|██████████| 16/16 [00:02<00:00,  5.61it/s]
```



Benchmark Selection Sortu

```
benchmark(selection_sort, 'Selection Sort')
```

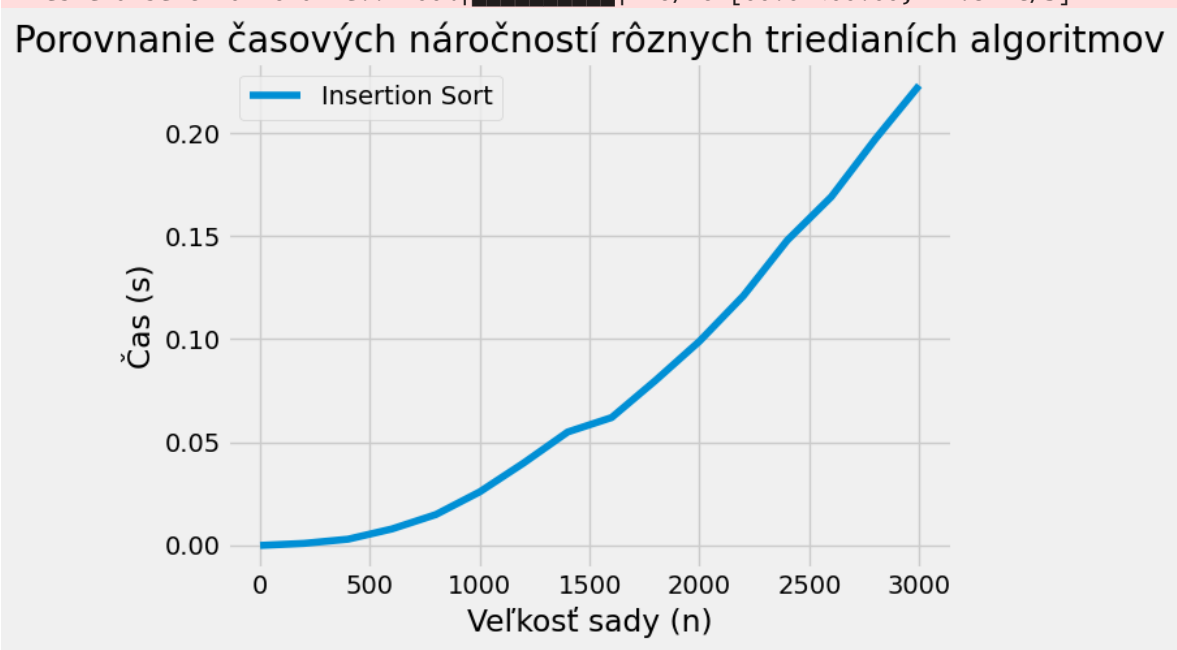
```
Benchmark pre Selection Sort
Prebieha benchmarkovanie:: 100%|██████████| 16/16 [00:01<00:00, 12.90it/s]
```



Benchmark Insertion Sortu

```
benchmark(insertion_sort, 'Insertion Sort')
```

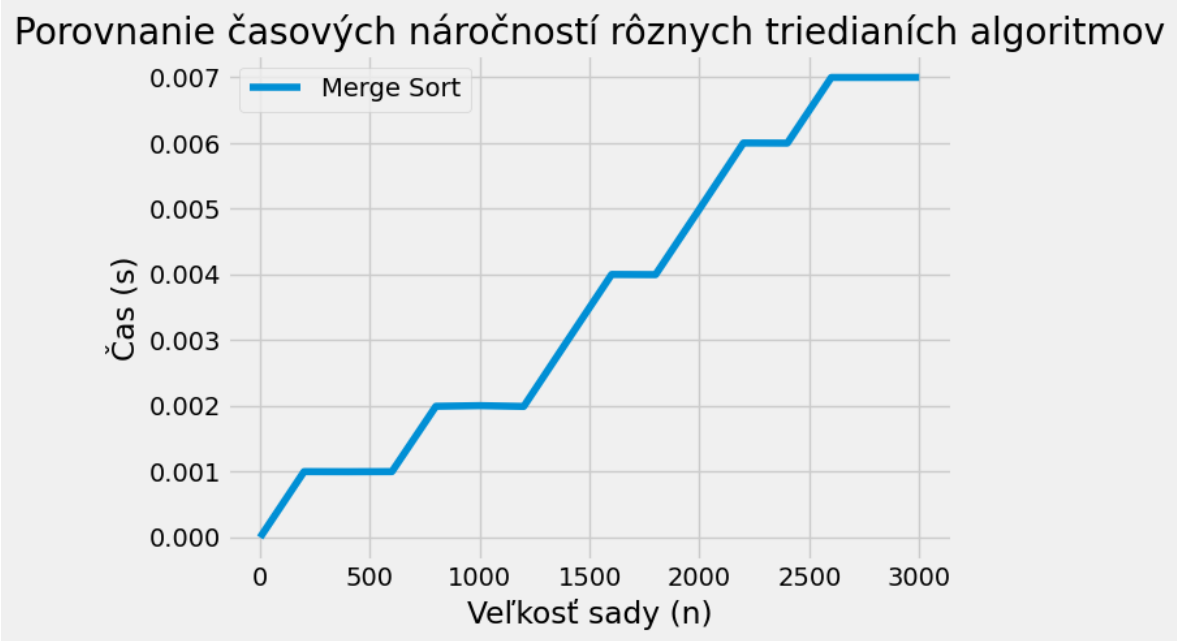
```
Benchmark pre Insertion Sort
Prebieha benchmarkovanie:: 100%|██████████| 16/16 [00:01<00:00, 12.81it/s]
```



Benchmark Merge Sortu

```
benchmark(merge_sort, 'Merge Sort')
```

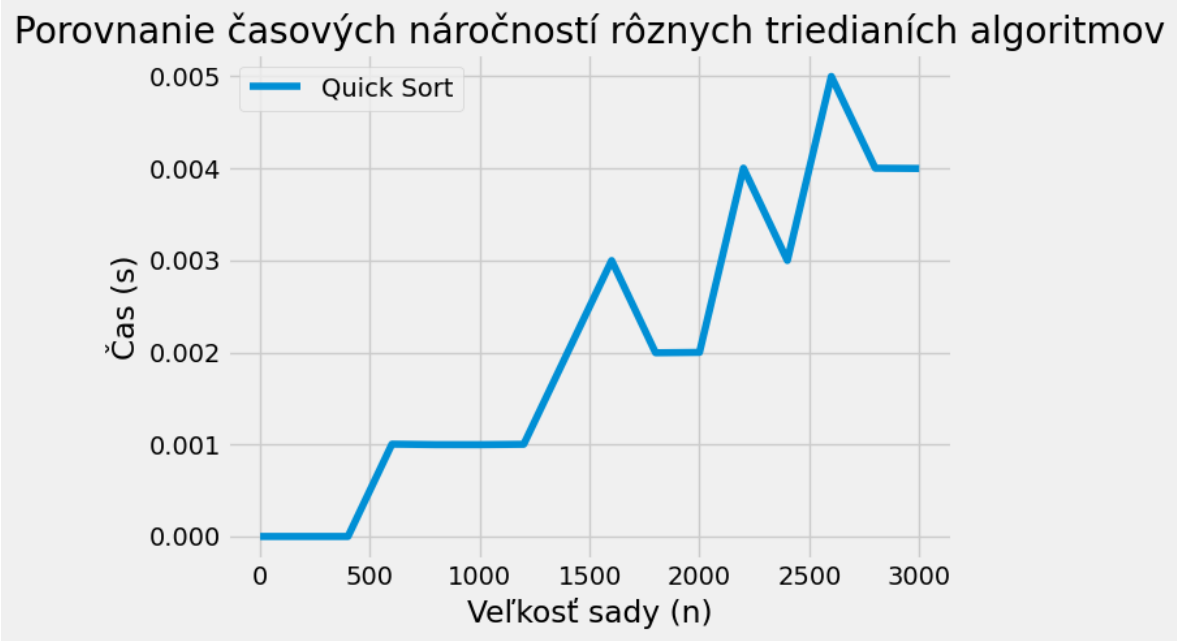
```
Benchmark pre Merge Sort
Prebieha benchmarkovanie:: 100%|██████████| 16/16 [00:00<00:00, 275.87it/s]
```



Benchmark Quick Sortu

```
benchmark(lambda arr: quick_sort(arr, 0, len(arr) - 1), 'Quick Sort')
```

```
Benchmark pre Quick Sort
Prebieha benchmarkovanie:: 100%|██████████| 16/16 [00:00<00:00, 457.20it/s]
```



Uloženie výsledkov do data.csv a ich zobrazenie

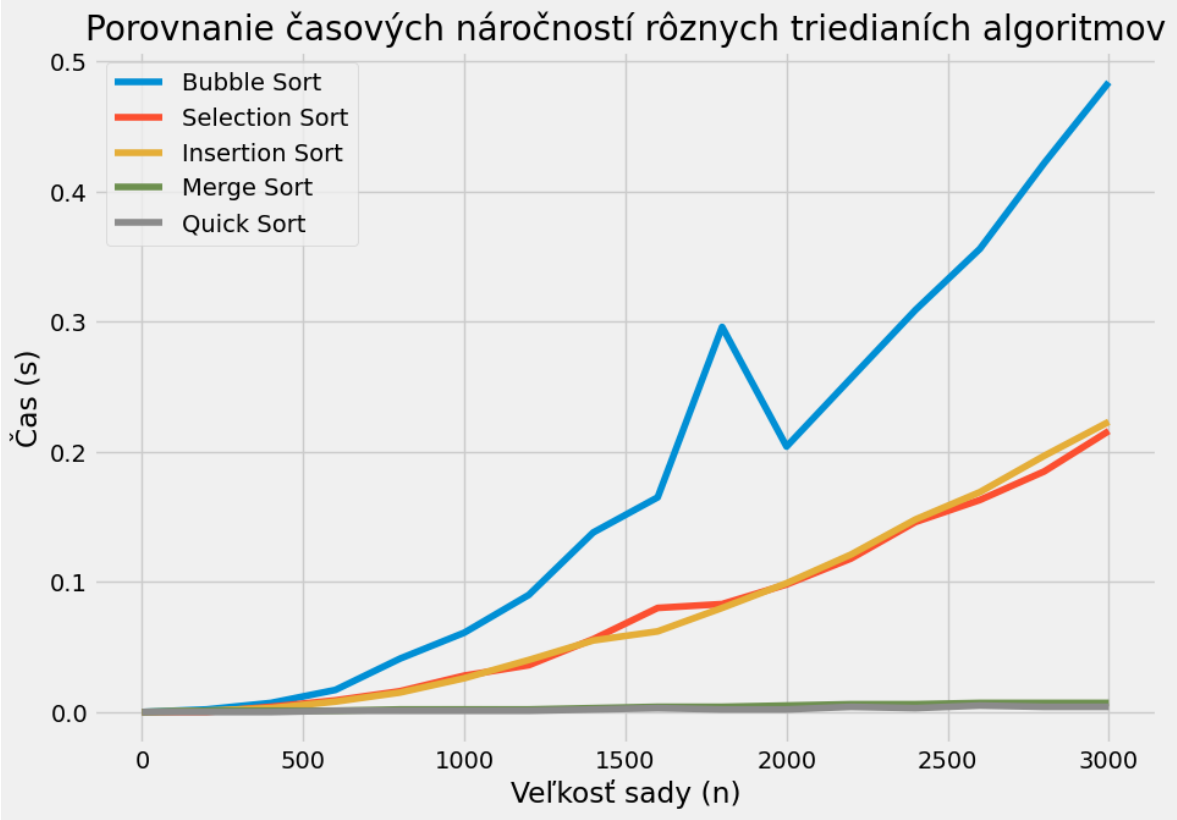
```
# df.to_csv('data.csv', index=False)
pd.set_option('display.precision', 4)
df.head(None)
```

Out[ ]:

	Velkost sady	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort
0	0	0.0000	0.000	0.000	0.000	0.000
1	200	0.0020	0.000	0.001	0.001	0.000
2	400	0.0070	0.004	0.003	0.001	0.000
3	600	0.0170	0.009	0.008	0.001	0.001
4	800	0.0410	0.016	0.015	0.002	0.001
5	1000	0.0610	0.028	0.026	0.002	0.001
6	1200	0.0900	0.036	0.040	0.002	0.001
7	1400	0.1380	0.056	0.055	0.003	0.002
8	1600	0.1650	0.080	0.062	0.004	0.003
9	1800	0.2960	0.083	0.080	0.004	0.002
10	2000	0.2040	0.098	0.099	0.005	0.002
11	2200	0.2566	0.118	0.121	0.006	0.004
12	2400	0.3090	0.146	0.148	0.006	0.003
13	2600	0.3560	0.163	0.169	0.007	0.005
14	2800	0.4220	0.185	0.197	0.007	0.004
15	3000	0.4840	0.216	0.223	0.007	0.004

Ich celkové porovnanie

```
# df = pd.read_csv('data.csv')

plt.figure(figsize=(10, 7))
for i in df.columns[1:]:
  plt.plot(df['Velkost sady'], df[i], label=i)
plt.legend()
plt.title('Porovnanie časových náročností rôznych triedianích algoritmov')
plt.xlabel('Veľkosť sady (n)')
plt.ylabel('Čas (s)')
plt.show()
```



Záver

V tomto projekte sme pracovali s rôznymi triediacimi algoritmami a vizualizovali sme ich fungovanie. Začali sme s implementáciou algoritmov ako Bubble Sort, Selection Sort, Insertion Sort, Merge Sort a Quick Sort. Následne sme vytvorili vizualizácie týchto algoritmov pomocou animácií.

Okrem toho sme tiež vykonali benchmarking týchto algoritmov a porovnali sme ich výkonnosť na rôznych veľkostiach vstupných dát. Výsledky sme zaznamenali do dataframe-u a vizualizovali sme ich pomocou grafu.

Zopakovali/Naučili sme sa základné princípy triediacich algoritmov a ich výkonnosť.A ako ich vizualizovať a porovnávať výsledky benchmarkingu.

Ukazalo sa že najrýchlejší algoritmus je Quicksort a Mergesort ktoré má priemernú časovú zložitosť O(n log n) a sú často jednými z najrýchlejších triediacich algoritmov, čím sa potvrdila naša hypotéza.