Algorytmy sortujące

Stanisław Olek

Spis treści

1 Wstęp

W tym artykule omówimy różne algorytmy sortujące, w tym insertion-sort, bubble-sort, merge-sort, heap-sort, quicksort, radixsort, countingsort i bucketsort.

2 Insertion Sort

Insertion Sort to prosty algorytm sortujący, który działa poprzez podział tablicy na część posortowaną i nieposortowaną. Następnie wybiera elementy z nieposortowanej części i umieszcza je na odpowiedniej pozycji w posortowanej części. roces ten jest kontynuowany, dopóki cała tablica nie zostanie posortowana.

2.1 Kod

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

3 Bubble Sort

Bubble Sort to algorytm sortujący, który działa poprzez wielokrotne przechodzenie przez tablicę i zamianę miejscami sąsiednie elementy, jeśli są one w złej kolejności.

3.1 Kod

```
def bubble_sort(A: np.ndarray) -> np.ndarray:
    for i in range(0,len(A)-1):
        for j in range(0, len(A)-i-1):
            if A[j] > A[j+1]:
                A[j], A[j+1] = A[j+1], A[j]
    return A
```

4 Merge Sort

Merge Sort to algorytm sortujący, który działa na zasadzie "dziel i zwyciężaj", dzieląc tablicę na dwie połowy, a następnie sortując każdą z nich, a na końcu łącząc je w jedną posortowaną tablicę.

4.1 Kod

```
def merge(A, p, s, k):
    L = A[p-1:s]
    R = A[s:k]
    L.append(np.inf)
    R.append(np.inf)
    i = 0
    j = 0
    for l in range(p-1, k):
        if L[i] <= R[j]:
            A[l] = L[i]
            i = i + 1
        else:
            A[l] = R[j]
            j = j + 1
```

```
def merge_sort(A, p, k):
    if p < k:
        s = (p+k)//2
        merge_sort(A, p, s)
        merge_sort(A, s+1, k)
        merge(A, p, s, k)
    return A
```

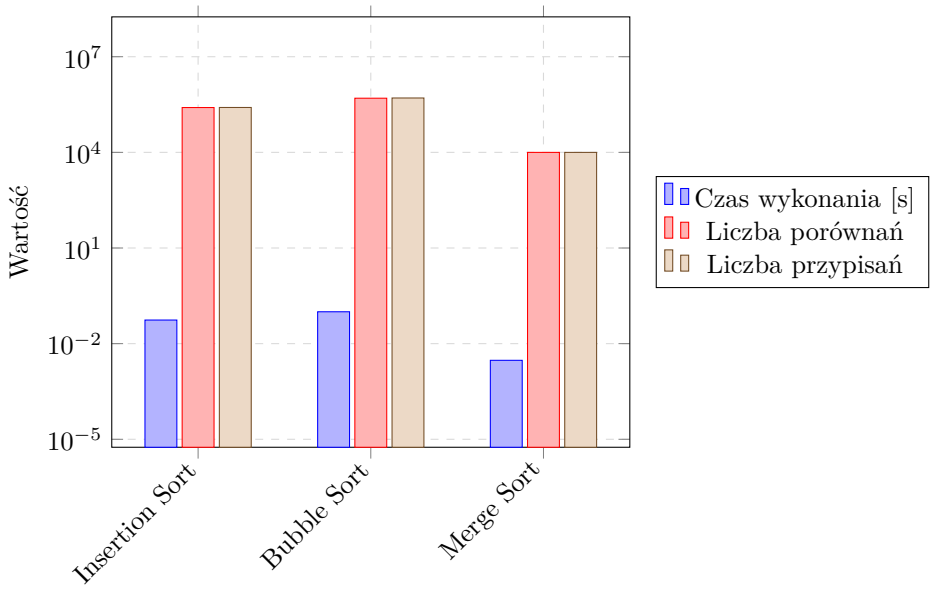
5 Wyniki testów

5.1 Tabela

Algorytm	Rozmiar danych	Czas wykonania	Liczba porównań	Liczba przypisań
Insertion Sort	1000	0.055s	254831	255835
Bubble Sort	1000	0.1s	499500	507674
Merge Sort	1000	0.003s	9976	9976

Tabela 1: Porównanie algorytmów sortujących: Insertion Sort, Bubble Sort i Merge Sort. Testy przeprowadzono dla losowych tablic o podanym rozmiarze.

5.2 Wykres



Rysunek 1: Porównanie algorytmów sortujących: Insertion Sort, Bubble Sort i Merge Sort. Testy przeprowadzono dla losowych tablic rozmiaru 1000.

6 Heap Sort

Heap Sort to algorytm sortujący, który działa na zasadzie porównań, opierając się na strukturze danych zwaną kopcem binarnym. Najpierw przekształca tablicę w kopiec, a następnie usuwa kolejno korzeń kopca i zastępuje go ostatnim węzłem, po czym przywraca własność kopca. Proces ten jest powtarzany, dopóki kopiec nie zostanie zredukowany do jednego elementu.

6.1 Kod

```
    def heapsort(A):
    build_heap(A)
    rozmiar_kopca = len(A)
    for i in range(rozmiar_kopca-1, 0, -1):
        A[0], A[i] = A[i], A[0]
        rozmiar_kopca = rozmiar_kopca - 1
        heapify(A, 0, rozmiar_kopca)
    return A

def build_heap(A):
    rozmiar_kopca = len(A)
    for i in range(rozmiar_kopca//2-1, -1, -1):
        heapify(A, i, rozmiar_kopca)
    return A

def heapify(A, i, rozmiar_kopca):
    l = 2*i + 1
    r = 2*i + 2
    if l < rozmiar_kopca and A[l] > A[i]:
        najwiekszy = l
    else:
        najwiekszy = i
    if r < rozmiar_kopca and A[r] > A[najwiekszy]:
        najwiekszy = r
    if najwiekszy != i:
        A[i], A[najwiekszy] = A[najwiekszy], A[i]
        heapify(A, najwiekszy, rozmiar_kopca)
    return A
```

7 Quick Sort

Quick Sort to algorytm sortujący działający na zasadzie "dziel i zwyciężaj", wybierając element jako pivot i partycjonując tablicę wokół wybranego pivotu. Pivot jest umieszczany na odpowiedniej pozycji w posortowanej tablicy, a wszystkie mniejsze elementy są umieszczane po lewej stronie pivotu, a większe po prawej. Proces ten jest powtarzany rekurencyjnie dla każdej z podtablic utworzonych przez partycjonowanie.

7.1 Kod

```
    def quicksort(A, p, k):
    if p < k:
        s = partition(A, p, k)
        quicksort(A, p, s-1)
        quicksort(A, s+1, k)
    return A

def partition(A, p, k):
    x = A[k]
    granica = p - 1
    for j in range(p, k):
        if A[j] <= x:
            granica += 1
            A[j], A[granica] = A[granica], A[j]
    A[granica+1], A[k] = A[k], A[granica+1]
    return granica + 1
```

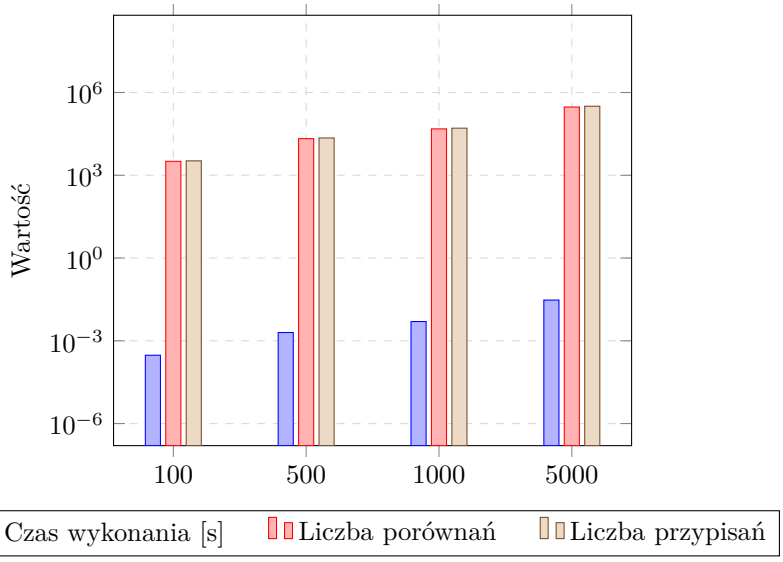
8 Wyniki testów

8.1 Tabela

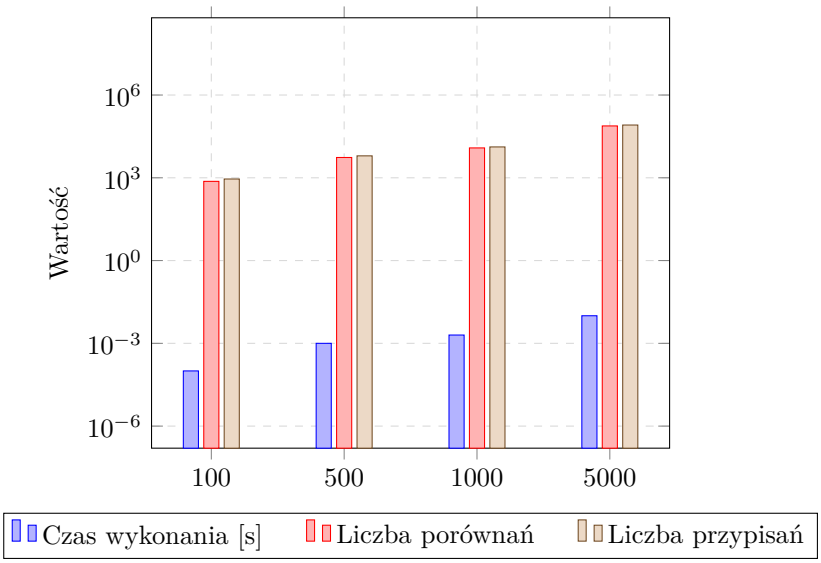
Algorytm	Rozmiar danych	Czas wykonania	Liczba porównań	Liczba przypisań
heapsort_plus	100	0.0003s	3205	3330
	500	0.002s	21285	22499
	1000	0.005s	48030	50958
	5000	0.03s	297805	317882
quicksort_plus	100	0.0001s	743	904
	500	0.001s	5447	6244
	1000	0.002s	12104	13146
	5000	0.01s	75880	81782

Tabela 2: Porównanie algorytmów sortujących: Heap Sort i Quick Sort. Testy przeprowadzono dla losowych tablic o podanych rozmiarach.

8.2 Wykresy



Rysunek 2: Porównanie zależności czasu wykonania, liczby porównań i liczby przypisań od wielkości danych dla algorytmu Heap Sort. Testy przeprowadzono dla losowych tablic o podanych rozmiarach.



Rysunek 3: Porównanie zależności czasu wykonania, liczby porównań i liczby przypisań od wielkości danych dla algorytmu Quick Sort. Testy przeprowadzono dla losowych tablic o podanych rozmiarach.

9 Counting Sort

Counting Sort jest algorytmem sortującym, który sortuje elementy na podstawie liczby wystąpień każdego klucza w tablicy wejściowej.

9.1 Kod

```
def countingsort(A, miejsce, podstawa):
    B = [0] * len(A)
    C = [0] * podstawa
    for i in range(len(A)):
        index = (A[i] // miejsce)
        C[index % podstawa] += 1
    for i in range(1, podstawa):
        C[i] += C[i-1]
    for i in range(len(A)-1, -1, -1):
        index = (A[i] // miejsce)
        B[C[index % podstawa] - 1] = A[i]
        C[index % podstawa] -= 1
    for i in range(len(A)):
        A[i] = B[i]
```

10 Radix Sort

Radix Sort jest algorytmem sortującym, który działa poprzez sortowanie danych na podstawie poszczególnych cyfr kluczy. Radix Sort działa od najmniej znaczącej cyfry do najbardziej znaczącej cyfry. Każda operacja sortowania na cyfrze jest wykonywana za pomocą stabilnego algorytmu sortującego, takiego jak Counting Sort.

10.1 Kod

```
def radixsort(A, podstawa):
    max_wart = max(A)
```

```
    miejsce = 1
    while max_wart // miejsce > 0:
        countingsort(A, miejsce, podstawa)
        miejsce *= podstawa
    return A
```

11 Bucket Sort

Algorytm Bucket Sort jest algorytmem sortującym, który dzieli dane na "kubełki", następnie sortuje każdy kubełek osobno za pomocą stabilnego algorytmu sortującego takiego jak Insertion Sort, na końcu zaś łączy je, aby uzyskać posortowaną listę.

11.1 Kod

```
def bucketsort(A):
    n = len(A)
    B = [[] for _ in range(n)]
    for i in range(n):
        index = int(A[i] * n)
        B[index].append(A[i])
    for i in range(n):
        B[i] = insertion_sort(B[i])
    k = 0
    for i in range(n):
        for j in range(len(B[i])):
            A[k] = B[i][j]
            k += 1
    return A
```

12 Wyniki testów

12.1 Tabele

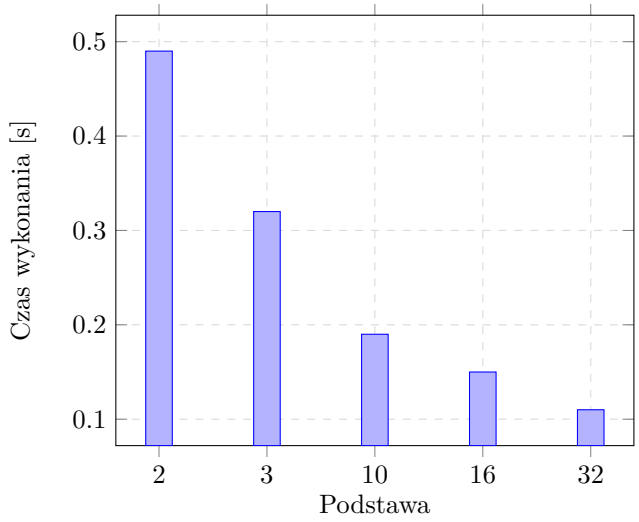
Podstawa	Czas wykonania
2	0.49s
3	0.32s
10	0.19s
16	0.15s
32	0.11s

Tabela 3: Porównanie czasów wykonania algorytmu Radix Sort dla różnych podstaw. Testy przeprowadzono dla losowych tablic rozmiaru 100 000.

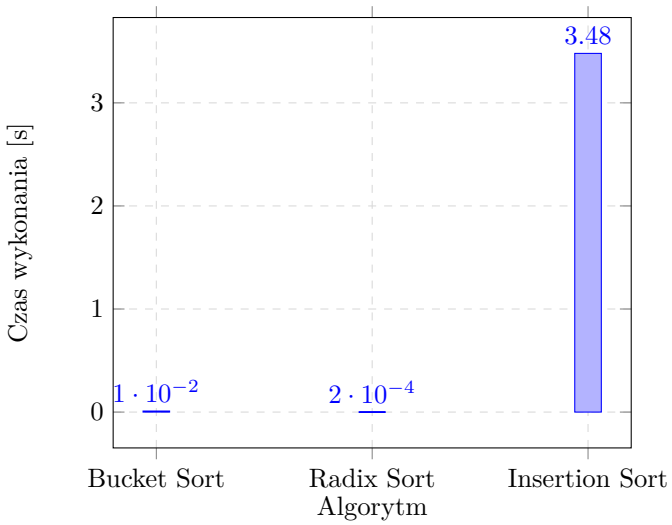
Algorytm	Czas wykonania
Bucket Sort	0.01s
Radix Sort	0.0002s
Insertion Sort	3.48s

Tabela 4: Porównanie czasów wykonania algorytmów sortujących: Bucket Sort, Radix Sort i Insertion Sort. Testy przeprowadzono dla losowych tablic o wartościach z przedziału $[0, 1)$ rozmiaru 10 000.

12.2 Wykresy



Rysunek 4: Porównanie czasów wykonania algorytmu Radix Sort dla różnych podstaw. Testy przeprowadzono dla losowych tablic rozmiaru 100 000.



Rysunek 5: Porównanie czasów wykonania algorytmów sortujących: Bucket Sort, Radix Sort i Insertion Sort. Testy przeprowadzono dla losowych tablic o wartościach z przedziału $[0, 1)$ rozmiaru 10 000.