Porównanie algorytmów sortujących

Anna Bonikowska

4 stycznia 2024

1 Wstęp

W dziedzinie informatyki, skuteczność algorytmów sortujących stanowi istotny obszar badań, mający fundamentalne znaczenie dla efektywnego przetwarzania danych. W niniejszym artykule porównam wybrane algorytmy sortujące, by ocenić ich złożoność czasową i pamięciową. Dzięki zrozumieniu różnic pomiędzy nimi możemy lepiej stosować te algorytmy do dpowiadających im sytuacji.

2 Omówienie algorytmów

2.1 BUBBLE-SORT

Działanie tego algorytmu polega na porównywaniu sąsiadujących lementów listy. Ma on złozoność $O(n^2)$. Jest to najgorsza złożoność wśród przedstawionych algorytmów. Oprócz łatwej implementacji nie ma żadnych zalet.

```
def BUBBLE_SORT(a):

    rozmiar= a.shape[0]

    for i in range(rozmiar):
        for j in range(rozmiar - i -1):
            if a[j]>a[j+1]:

                pom=a[j]
                a[j]=a[j+1]
                a[j+1]=pom
    return a
```

2.2 INSERTION-SORT

Sposób działania tego algorytmu jest porównywalny do sortowania kart. Na początku sortuje się początek tablicy po czym następny element wstawia się w posortowaną cześć analogicznie robi się z pozostałymi elementami.

```
def INSERTION_SORT(a):


    rozmiar= a.shape[0]

    for j in range(1,rozmiar):
        key= a[j]
        i=j-1

        while i>-1 and a[i]>key:
            a[i+1]=a[i]
            i-=1
```

```
            a[i+1]=key

        return a
```

2.3 MERGE-SORT

Jest to algorytn rekurencyjny, czyli sam siebie wywołuje. Dzieli on tablice na coraz mniejsze kawałki aż do momentu kiedy są one długości 1(tablice jednoelementowe są posortowane). Po czym je łączy tak aby wyjsciowa była posortowana. Poniżej przedstawiłam głóną funkcję tego algorytmu.

```
def MERGE_SORT2(A,p,k):
    if p<k:
        s= np.floor((p+k)/2)
        MERGE_SORT2(A,p,s)
        MERGE_SORT2(A,s+1,k)
        MERGE(A,p,s,k)
```

To jest główny fragment przykładowej funkcji merge. Polega na kolejnym dodawaniu elementów by był zachowany porządek.

```
while i < n1 and j < n2:
    if L[i] <= R[j]:
        A[x] = L[i]
        i += 1
    else:
        A[x] = R[j]
        j += 1
    x += 1


while i < n1:
    A[x] = L[i]
    i += 1
    x += 1


while j < n2:
    A[x] = R[j]
    j += 1
    x += 1
```

2.4 MERGE-SORT21

Ten algorytm działania w podobny sposób jak poprzedni z tą różnicą, że nie dzieli tablicy na dwie równe części tylko dzieli je w stosunky 2:1. Jest on odrobinę szybszy od poprzednika.

```
def MERGE_SORT21(A,p,k):
    if p<k and (k-p)>3:
        s=p+ np.floor((k-p)*(2/3))
        MERGE_SORT221(A,p,s)
        MERGE_SORT221(A,s+1,k)
        MERGE(A,p,s,k)
    elif p<k:
        k=int(k)
        p=int(p)
```

```
    B=np.ones(k-p+1)
    for i in range(0,k-p+1):
        B[i]=A[i+p]
    BUBBLE_SORT(B)
    for i in range(0,k-p+1):
        A[i+p]=B[i]
```

2.5 HEAPSORT

Jest to algorytm rekurencyjny. Przedstawia on tablicę w formie kopca binarnego. Na początek tworzy on kopiec i nadaje mu odpowiednie właściwości, czyli rodzic musi być większy od dzieci które są pod nim. Za to jest odpowiedzialna funkcja build_heap. Następnie zostaje zamieniony pierwszy element z ostanim i przywra się własność kopca. Następnie zamienia się pierwszy element z drugim od końca . Podobnie robi się z pozostałymi.

Jest to główna funkcja tego alborytmu.

```
def HEAPSORT(A):
    build_heap(A)
    n=len(A)
    for i in range(len(A)-1, 0 ,-1):
        k=A[i]
        A[i]=A[0]
        A[0]=k
        n-=1
        heapify(A,n,0)
```

Budowanie kopca.

```
def build_heap(A):
    heap_sizeA=len(A)

    for i in range(heap_sizeA//2 -1, -1, -1):

        heapify(A,heap_sizeA,i)
```

Ta funkcja jest odpowiedzialbna za przywracanie własności kopca. Sprawdza czy rodzic jest większy od dzieci i jeśli tak nie jest to dokonuje odpowiednie zamiany . Następnie sprawdza czy rodzic w zamienionym miejscu jest wiekszy od swoich nowych dzieci jęśli nie znowu jest robiona zamiana i znowu się sprawdza czy rodzic jest wiekszy od dzieci itd..

```
def heapify(A ,n, i):
    l=2*i +1
    r=2*i +2
    heap_size_A= n-1

    if l<=heap_size_A and A[l]>A[i]:
        largest=l
    else:
        largest=i

    if r<=heap_size_A and A[r]>A[largest]:
        largest=r
```

```
    if largest!=i:
        k=A[i]
        A[i]=A[largest]
        A[largest]=k

        heapify(A,n,largest)
```

2.6 HEAPSORT3

Ten algorytm działa podobnie jak HEAPSORT z tą rożnicą, że rodzic ma trójkę dzieci zamiast dwójki z którą jest porównywany.

Jest to główna funkcja tego alborytmu.

```
def HEAPSORT3(A):
    build_heap3(A)
    n=len(A)
    for i in range(len(A)-1, 0 ,-1):
        k=A[i]
        A[i]=A[0]
        A[0]=k
        n-=1
        heapify3(A,n,0)
```

Budowanie kopca.

```
def build_heap3(A):
    heap_sizeA=len(A)

    for i in range(heap_sizeA//3 , -1, -1):

        heapify3(A,heap_sizeA,i)
```

Ta funkcja jest odpowiedzialbna za przywracanie własności kopca. Sprawdza czy rodzic jest większy od dzieci i jeśli tak nie jest to dokonuje odpowiednie zamiany . Następnie sprawdza czy rodzic w zamienionym miejscu jest wiekszy od swoich nowych dzieci jęśli nie znowu jest robiona zamiana i znowu się sprawdza czy rodzic jest wiekszy od dzieci itd..

```
def heapify3(A ,n, i):
    l=3*i +1
    c=3*i +2
    r=3*i +3
    heap_size_A= n-1

    if l<=heap_size_A and A[l]>A[i]:
        largest=l
    else:
        largest=i

    if c<=heap_size_A and A[c]>A[largest]:
        largest=c
```

```
    if r<=heap_size_A and A[r]>A[largest]:
        largest=r

    if largest!=i:
        k=A[i]
        A[i]=A[largest]
        A[largest]=k

        heapify3(A,n,largest)
```

2.7 QUICKSORT

Quicksort to algorytm sortowania, który operuje na zasadzie dziel i zwyciężaj. Proces ten rozpoczyna się od wyboru elementu pivot, a następnie dzieli tablicę na dwie części , jedną z elementami mniejszymi od pivota i drugą z elementami większymi. Procedura ta jest rekurencyjnie powtarzana dla obu podtablic, co prowadzi do efektywnego uporządkowania całej tablicy.

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

W funkcji partition elementy tablicy są tak zamieniane miejscami by podzielić je na część mniejszych i większych od pivota.

```
def partition(A,p,k):
    x=A[k]
    i=p-1

    for j in range(p,k):

        if A[j]<=x:
            i+=1

            d=A[i]
            A[i]=A[j]
            A[j]=d

    d=A[i+1]
    A[i+1]=A[k]
    A[k]=d

    return i+1
```

2.8 QUICKSORT3

Quicksort3 działa w podobny sposób jak algorytm Quicksort z tą różnica ,że tablica jest dzielona na 3 części wzgledem dwóch pivotów a nie jednego.

```
def quicksort3(A,p,k):
    if 2<=(k-p):
```

```
        l=partition3(A,p,k)
        a=l[0]
        b=l[1]
        quicksort3(A,p,a-1)
        quicksort3(A,a+1,b-1)
        quicksort3(A,b+1,k)
    elif k>p:
        if A[p]>A[k]:
            d=A[k]
            A[k]=A[p]
            A[p]=d
```

Funkcja artition3 działa annnnnngcznie do funkcji partition z poprzedniego algorytmu. Najpierw dzieli elementy tablicy według pierwszego pivota a następnie według drugiego. Dlatetego kod tej funkcji est dwa razy dłuższy od poprzedniego.

```
def partition3(A,p,k):


if A[k-1]>A[k]:
    d=A[k]
    A[k]=A[k-1]
    A[k-1]=d

x=A[k-1]
y=A[k]



i=p-1

for j in range(p,k-1):

    if A[j]<=x:
        i+=1

        d=A[i]
        A[i]=A[j]
        A[j]=d

d=A[i+1]
A[i+1]=A[k-1]
A[k-1]=d

a=i+1

if k-a>=2:
    i=a

    for j in range(a+1,k):

        if A[j]<=y:
            i+=1

            d=A[i]
            A[i]=A[j]
```

```
            A[j]=d

        d=A[i+1]
        A[i+1]=A[k]
        A[k]=d

        b=i+1

    else:
        b=k

    return [a,b]
```

2.9 COUNTINGSORT

Ten algorytm działa w całkowicie inny sposób od wcześniejszych. Działanie poprzednich algorytmów opierało sie głównie na porównywaniu elementów tablicy z sobą. Głównym pomysłem countingsort jest liczenie ile razy pojawiła się dana liczba i zapisywanie tych wyników w tablicy. Następnie liczby wypisuje się tyle razy ile wystąpiła dana liczba w kolejności rosnącej.

```
def COUNTINGSORT(A,k):
    C=np.zeros(k)
    B=np.zeros(len(A))

    for i in range(0,k):
        C[i]=0

    for i in range(0,len(A)):
        C[int(A[i])]+=1

    for i in range(1,k):
        C[i]+=C[i-1]

    for i in range(len(A)-1,-1,-1):
        print(int(C[int(A[i])])-1)
        B[int(C[int(A[i])])-1]=A[i]
        C[A[i]]-=1

    for i in range(0, len(A)):

        A[i]=B[i]
```

2.10 COUNTINGSORT

Ten algorytm działa w całkowicie inny sposób od wcześniejszych. Działanie poprzednich algorytmów opierało sie głównie na porównywaniu elementów tablicy z sobą. Głównym pomysłem countingsort jest liczenie ile razy pojawiła się dana liczba i zapisywanie tych wyników w tablicy. Następnie liczby wypisuje się tyle razy ile wystąpiła dana liczba w kolejności rosnącej. Jest to algorytm stabilny, czyli zachowuje kolejność elementów w tablicy jeśli posiadają taką samą wartosc według której sa sotowane.

```
def COUNTINGSORT(A,k):
    C=np.zeros(k)
    B=np.zeros(len(A))

    for i in range(0,k):
        C[i]=0
```

```
    for i in range(0,len(A)):
        C[int(A[i])]+=1

    for i in range(1,k):
        C[i]+=C[i-1]

    for i in range(len(A)-1,-1,-1):
        print(int(C[int(A[i])])-1)
        B[int(C[int(A[i])])-1]=A[i]
        C[A[i]]-=1

    for i in range(0, len(A)):

        A[i]=B[i]
```

2.11 RADIXSORT

Radixsort jest modyfikacją algorytmu countingsort. Sortuje on liczby według cyfr znajdujących sie na pozycjach. W każdym przejściu sortuje elementy na podstawie jednej konkretnej pozycji , a następnie przchodzi do sortowania według wyższej pozycji, aż do momentu kiedy wszystkie pozycje zostana uwzględnione.

```
def COUNTINGSORT(A,k):
    C=np.zeros(k)
    B=np.zeros(len(A))

    for i in range(0,k):
        C[i]=0

    for i in range(0,len(A)):
        C[int(A[i])]+=1

    for i in range(1,k):
        C[i]+=C[i-1]

    for i in range(len(A)-1,-1,-1):
        print(int(C[int(A[i])])-1)
        B[int(C[int(A[i])])-1]=A[i]
        C[A[i]]-=1

    for i in range(0, len(A)):

        A[i]=B[i]
```

2.12 BUCKETSORT

Bucketsort sortuje liczby z przedziału [0,1). Najpierw rozdziela elementy do list na podstawie ich wartości , a następnie sortuje listy wybranym algorytmem. Po posortowaniu list łączy je z sobą i zwraca jako wynik.

```
def BUCKETSORT(A,n):
    B= []
    a=len(A)
    for i in range(0,n):
        B.append([])

    for i in range(0,len(A)):
```

```
        B[math.floor((n)*A[i])].append(A[i])

    for i in range(0,n):
        INSERTION_SORT(B[i])

    C=[]

    for i in range(0,n):
        C+=B[i]

    return C
```

3 Testy

3.1 wnioski

- Ulepsszenie, które zostało użute w BUCKETSORT jest bardzo wydajne, ponieważ dzieki niemu działa on nieporównywalnie szybciej niż INSERTIONSORT , jest w nim używany do sortowania mniejszych tablic.
- Mimo, że w niekorzystnych przypadkack czas działania QUICKSORT może wynieść O(n^2), to jest to najszybszy algorytm, który prztestowałam (nie licząc BUCKETSORT)
- Najmniej wydajnym algorytmem okazał się być BUBBLESORT
- Mimo tego, że COUNTINGSORT ma złożoność asymptotyczną O(n+m), to dla małych tablic nie opłaca się go stosować, jego czas działania jest zbyt duży. Natomiast wydaje się, że wielkość danych nie ma wpływu nie ma jego czas działania

3.2 tabelki i wykresy

nazwa algorytmu	10	100	1000	10000
HEAPSORT	1,41 e-4	1,57 e-3	2.33 e-2	3,11 e-1
HEAPSORT3	1,08 e-4	1,25 e-3	1,81 e-2	2,38 e-1
QUICKSORT	7,26 e-5	5,27 e-4	7,63 e-3	9,69 e-2
QUICKSORT3	7,33 e-5	6,39 e-4	6,96 e-3	9,01 e-2
MERGE_SORT	1,48 e-4	1,4- e-3	1,72 e-2	2,01 e-1
MERGE_SORT21	1,38 e-4	1,25 e-3	1,70 e-2	1.97 e-1
INSERTION_SORT	3,38 e-5	1,91 e-3	1,26 e-1	1,25 e1
BUBBLE_SORT	2,84 e-3	2,84 e-3	2,78 e-1	2,77 e1
BUCKETSORT	2,86 e-4	3,08 e-4	2,02 e-3	1,00 e-1
RADIXSORT	4,21 e-4	2,92 e-3	2,23 e-2	2,127 e-1
COUNTINGSORT	5,88 e2	5,87 e2	5,86 e2	

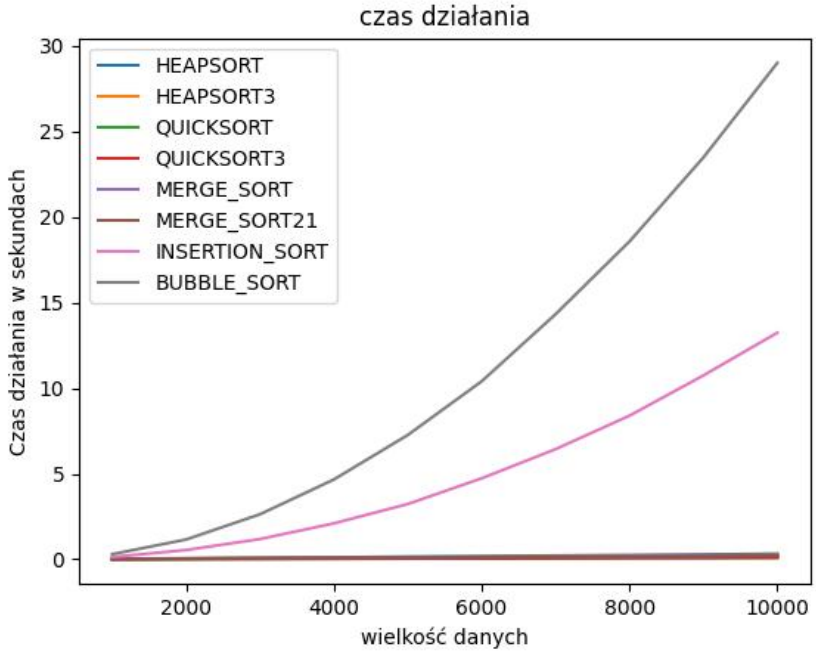
Rysunek 1: Wyniki testów czasu działania algorytmów w zależności od wielkości danych

nazwa algorytmu	10	100	1000	10000
HEAPSORT	1,5 e2	3,41 e3	5,37 e4	7,37 e5
HEAPSORT3	1,19 e2	2,39 e3	3,63 e4	4,87 e5
QUICKSORT	3,8 e1	7,34 e2	1,32 e4	1,65 e5
QUICKSORT3	5,1 e1	8,02 e2	1,26 e4	1,67 e5
MERGE_SORT	1,31 e2	2,31 e3	3,29 e4	4,29 e5
MERGE_SORT21	8,5 e1	1,87 e3	2,97 e4	4,02 e5
INSERTION_SORT	5,8 e1	5,15 e3	5,11 e5	5,00 e7
BUBBLE_SORT	4,5 e1	4,95 e3	4,99 e5	4,99 e7

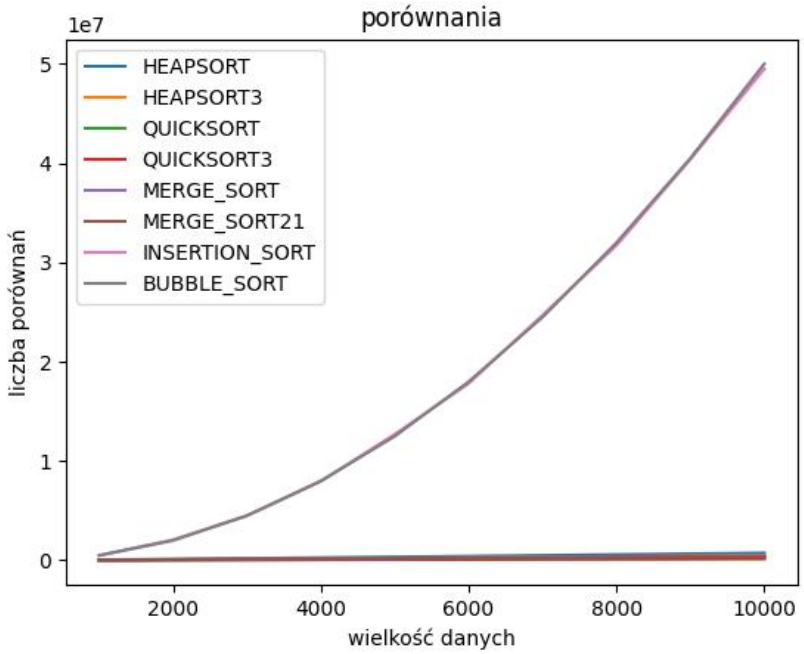
Rysunek 2: Wyniki testów ilości porównań podczas działania algorytmów w zależności od wielkości danych

nazwa algorytmu	10	100	1000	10000
HEAPSORT	2,32 e2	4,56 e3	7,05 e4	9,53 e5
HEAPSORT3	2,32 e2	3,92 e3	5,68 e4	7,53 e5
QUICKSORT	6,20 e1	1,55 e3	3,23 e4	3,41 e5
QUICKSORT3	1,10 e2	1,64 e3	1,96 e4	2,65 e5
MERGE_SORT	2,09 e2	3,48 e3	4,78 e4	6,14 e5
MERGE_SORT21	1,39 e2	3,39 e3	4,73 e4	6,17 e5
INSERTION_SORT	3,90 e1	2,67 e3	2,46 e5	2,50 e7
BUBBLE_SORT	6,60 e1	7,35 e3	7,41 e5	7,55 e7

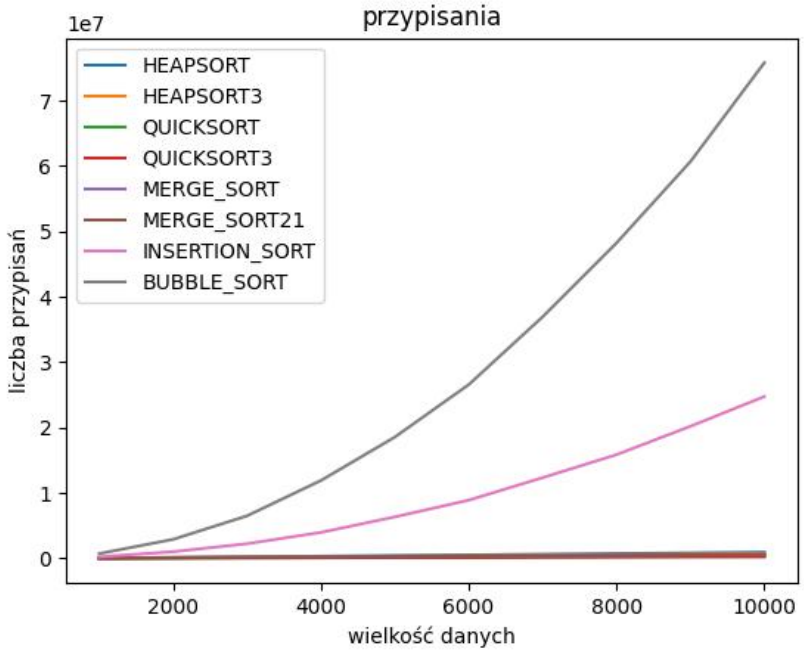
Rysunek 3: Wyniki testów ilości przypisań podczas działania algorytmów w zależności od wielkości danych



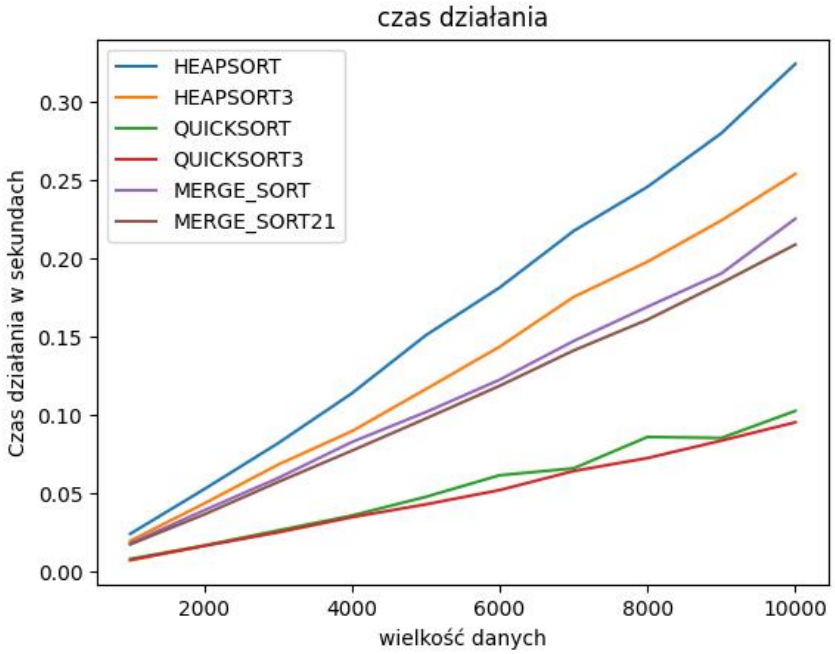
Rysunek 4: Wyniki testów czasu działania algorytmów w zależności od wielkości danych



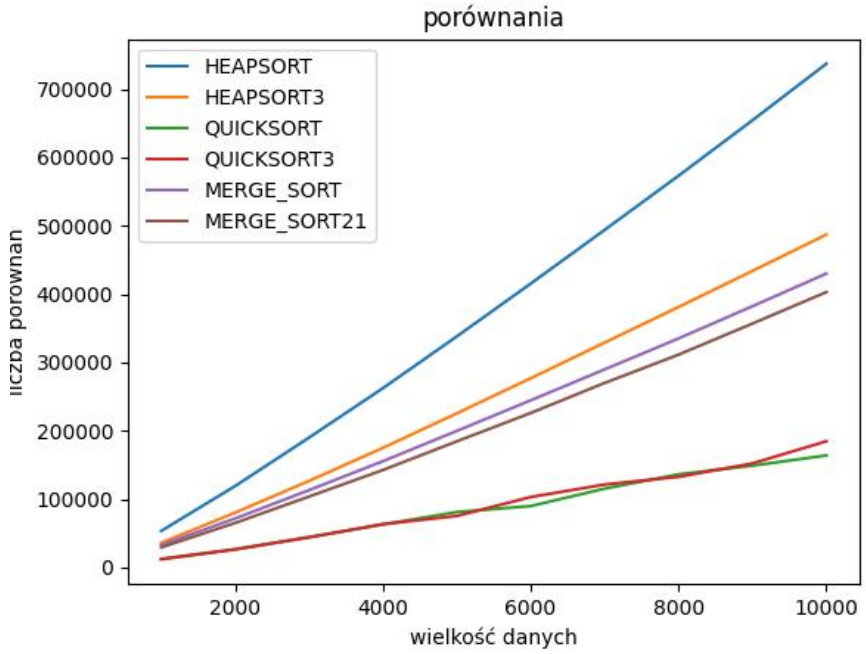
Rysunek 5: Wyniki testów ilości porównań podczas działania algorytmów w zależności od wielkości danych



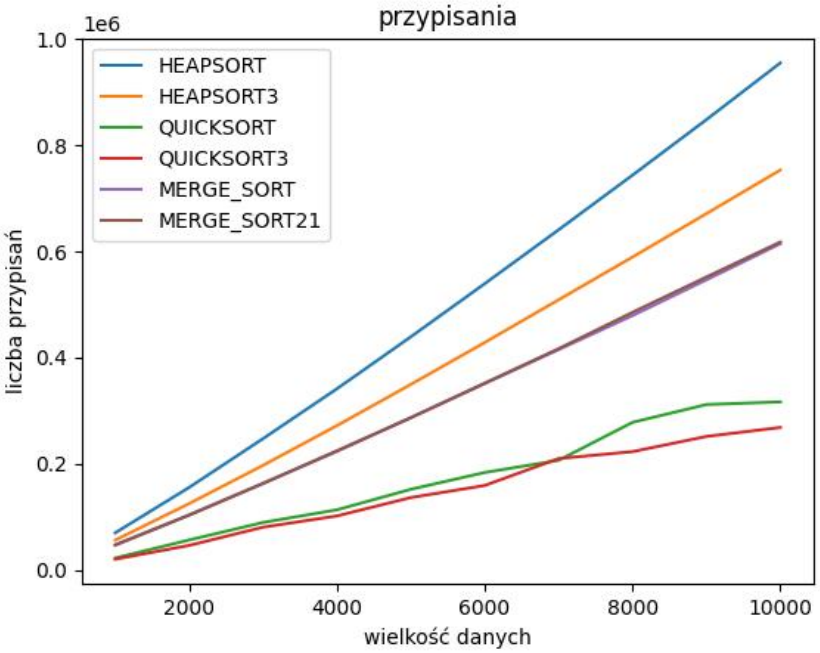
Rysunek 6: Wyniki testów ilości przypisań podczas działania algorytmów w zależności od wielkości danych



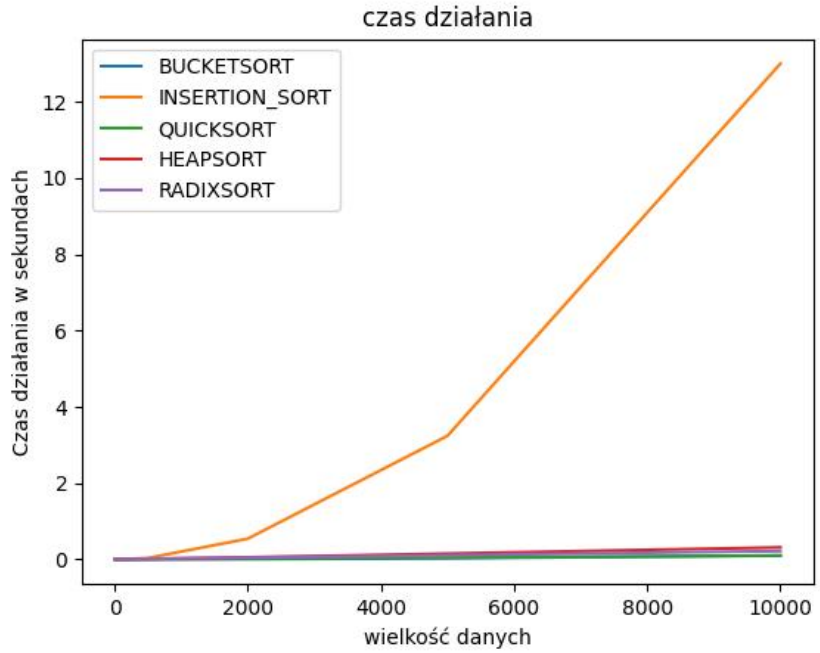
Rysunek 7: Wyniki testów czasu działania algorytmów w zależności od wielkości danych, z pominieciem najgorszychwyników



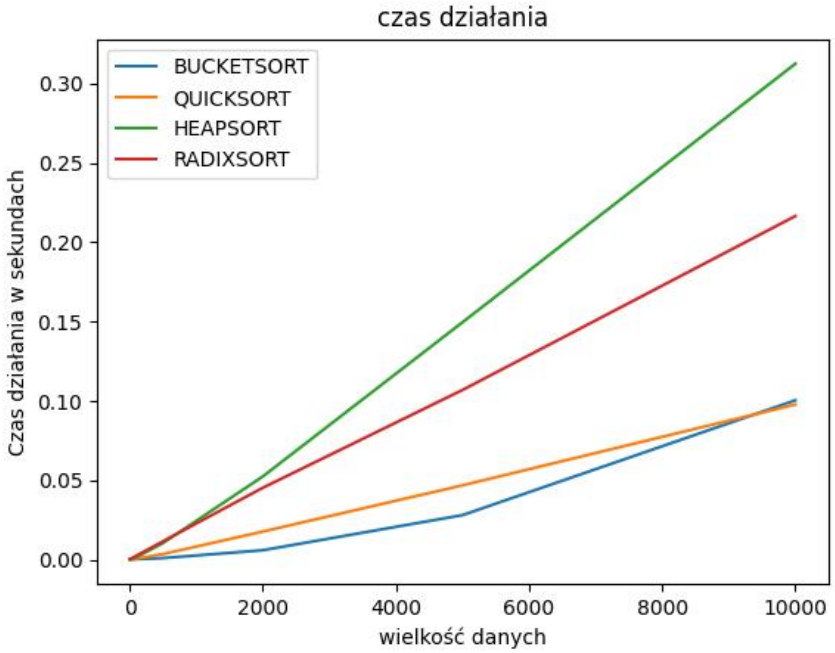
Rysunek 8: Wyniki testów ilości porównań podczas działania algorytmów w zależności od wielkości danych, z pominieciem najgorszychwyników



Rysunek 9: Wyniki testów ilości przypisań podczas działania algorytmów w zależności od wielkości danych, z pominieciem najgorszychwyników



Rysunek 10: Wyniki testów czasu działania algorytmów w zależności od wielkości danych



Rysunek 11: Wyniki testów czasu działania algorytmów w zależności od wielkości danych, z pominięciem INSERTSORT