Paweł Drakoniewicz

Algorytmy sortowania QuickSort

Celem tego projektu jest porównanie różnych implementacji algorytmu Quicksort. W badaniu skupiamy się na trzech wariantach: QuickSort z Partition w wersji Lomuto z losowaniem pivota, QuickSort z Partition w wersji Hoare oraz QuickSort z Partition w wersji Dutch flag.

Quicksort to jeden z najpopularniejszych i najbardziej wydajnych algorytmów sortowania stosowanych w informatyce. Polega on na dzieleniu zbioru danych na mniejsze podzbiory, sortowaniu ich rekurencyjnie, a następnie łączeniu w celu uzyskania posortowanego wyniku.

W ramach tego badania, testujemy czas działania każdej z wymienionych implementacji Quicksorta. Nasze testy będą oparte na tablicach z losowymi liczbami, których długości będą stanowiły ciąg arytmetyczny. Dodatkowo, zbadamy wpływ małego zakresu losowanych liczb na wydajność każdego z algorytmów.

Analiza czasu działania algorytmów dla różnych rozmiarów danych oraz różnych zakresów wartości w tablicach pozwoli nam na ocenę ich skuteczności w praktycznych zastosowaniach. Przeprowadzone testy pozwolą nam również zidentyfikować, który z wariantów Quicksort może być najbardziej odpowiedni dla konkretnych scenariuszy i rodzajów danych.

Oto kody trzech różnych wariantów algorytmu Quicksort:

- **Quicksort z Partition w wersji Lomuto z losowaniem pivota**

```
def PartitionLomutoRand(arr, left, right):
    pivot_index = random.randint(left, right)  # Losowy wybór indeksu pivota
    arr[pivot_index], arr[right] = arr[right], arr[pivot_index]  # Zamiana pivota z ostatnim elementem
    pivot = arr[right]  # Pivot to ostatni element
    j = left  # Indeks dla elementów mniejszych niż pivot
    for i in range(left, right):
        if arr[i] <= pivot:
            arr[j], arr[i] = arr[i], arr[j]  # Zamiana elementów mniejszych niż pivot z elementami większymi niż pivot
            j += 1
    arr[j ], arr[right] = arr[right], arr[j]  # Umieszczenie pivota na właściwej pozycji
    return j  # Zwraca indeks pivota


def QuickSortLomutoRand(arr, left, right):
    if left < right:
        pivot_index = PartitionLomutoRand(arr, left, right)
        QuickSortLomutoRand(arr, left, pivot_index-1)
        QuickSortLomutoRand(arr, pivot_index + 1, right)
```

Funkcja **PartitionLomutoRand(arr, left, right)** dokonuje partycjonowania tablicy arr z użyciem schematu Lomuto z losowaniem pivota. Jest to proces dzielenia tablicy na dwie części: jedną z elementami mniejszymi lub równymi pivotowi i drugą z elementami większymi od pivota. Zwraca indeks pivota.

- **Quicksort z Partition w wersji Hoare**

```
def PartitionHoare(arr, left, right):
    pivot = arr[left] # Wybór pierwszego elementu jako pivota
    i = left - 1 # Inicjalizacja wskaźnika i na indeks leżący przed lewą stroną tablicy
    j = right + 1# Inicjalizacja wskaźnika j na indeks leżący po prawej stronie tablicy
    while True:
        i += 1# Zwiększenie indeksu i, aby przesunąć się w kierunku końca tablicy
        while (arr[i] < pivot):
            i += 1 # Przesuwanie wskaźnika i w prawo, aż znajdzie element większy lub równy pivotowi
        j -= 1 # Zmniejszenie indeksu j, aby przesunąć się w kierunku początku tablicy
        while (arr[j] > pivot):
            j -= 1 # Przesuwanie wskaźnika j w lewo, aż znajdzie element mniejszy lub równy pivotowi
        if (i >= j): # Jeśli wskaźnik i przekroczył wskaźnik j, zakończ pętlę
            return j  # Zwróć indeks j jako indeks pivota po partycjonowaniu
        arr[i], arr[j] = arr[j], arr[i] # Zamień miejscami elementy na pozycjach i i j, aby umieścić je po odpowiednich stronach pivota
def QuickSortHoare(arr, left, right):
    if left < right:
        pivot_index = PartitionHoare(arr, left, right)
        QuickSortHoare(arr, left, pivot_index)
        QuickSortHoare(arr, pivot_index + 1, right)
```

Funkcja **PartitionHoare(arr, left, right)** dokonuje partycjonowania tablicy arr z użyciem schematu Hoare. Rozpoczyna od wyboru pierwszego elementu tablicy jako pivota. Następnie, używając dwóch wskaźników, przesuwa się przez tablicę, inkrementując wskaźnik *i*, aby znaleźć element większy lub równy pivotowi oraz dekrementując wskaźnik *j*, aby znaleźć element mniejszy lub równy pivotowi. Gdy wskaźnik *i* przekroczy wskaźnik *j*, zamienia miejscami elementy na ich odpowiednich pozycjach względem pivota. Na końcu zwraca indeks *j* jako indeks pivota po partycjonowaniu.

- **Quicksort z Partition w wersji Dutch flag.**

```
def PartitionDutchFlag(arr, left, right, pivot_index):
    pivot = arr[pivot_index]  # Ustalenie wartości pivota
    smaller = left  # Indeks obszaru elementów mniejszych od pivota
    equal = left    # Indeks obszaru elementów równych pivotowi
    larger = right # Indeks obszaru elementów większych od pivota

    while equal <= larger:
        if arr[equal] < pivot:  # Jeśli element jest mniejszy od pivota
            arr[smaller], arr[equal] = arr[equal], arr[smaller]  # Zamień miejscami aktualny element z elementem o indeksie smaller
            smaller += 1  # Zwiększ obszar mniejszych elementów
            equal += 1    # Przejdź do następnego elementu
        elif arr[equal] == pivot:  # Jeśli element jest równy pivotowi
            equal += 1  # Przejdź do następnego elementu
        else:  # Jeśli element jest większy od pivota
            arr[equal], arr[larger] = arr[larger], arr[equal]  # Zamień miejscami aktualny element z elementem o indeksie larger
            larger -= 1  # Zmniejsz obszar większych elementów
    return smaller, larger  # Zwróć granice obszarów mniejszych i większych od pivota

def QuickSortDutchFlag(arr, left, right):
    if left < right:
        pivot_index = random.randint(left, right)
        smaller, larger = PartitionDutchFlag(arr, left, right, pivot_index)
        QuickSortDutchFlag(arr, left, smaller - 1)
        QuickSortDutchFlag(arr, larger + 1, right)
```

Funkcja **PartitionDutchFlag(arr, left, right, pivot_index)** dokonuje partycjonowania tablicy arr z użyciem schematu Dutch flag. W tym schemacie pivot jest określany na podstawie indeksu przekazanego jako argument funkcji. Podczas partycjonowania, elementy tablicy są grupowane na trzy kategorie: mniejsze od pivota, równe pivotowi oraz większe od pivota. Po zakończeniu procesu, funkcja zwraca granice obszarów zawierających elementy mniejsze i większe od pivota.

Funkcje **QuickSortLomutoRand(arr, left, right), QuickSortHoare(arr, left, right) i QuickSortDutchFlag(arr, left, right)** to implementacje algorytmu Quicksort dla odpowiednio Lomuto z losowaniem pivota, Hoare i Dutch flag. Każda z nich rekurencyjnie sortuje podtablice od lewej do prawej strony tablicy.

Sposób testowania i prezentacji wyników działania trzech różnych wariantów algorytmu Quicksort na losowo generowanych danych:

- Tablice testowe będą generowane przy użyciu funkcji **rand_list**, która losuje liczby z podanego zakresu

  ```
  import random
  def rand_list(length, start=1, stop=1000000):
      rlist = []
      for i in range(length):
          rlist.append(random.randint(start, stop))
      return rlist
  ```

- Czas wykonania algorytmów będzie mierzony za pomocą funkcji **ileczasu**, która zwraca czas wykonania funkcji sortującej dla danej tablicy

  ```
  import time
  def ileczasu(func, *args):
      start= time.time()
      func(*args)
      end= time.time()
      return end - start
  ```

- Testy zostaną przeprowadzone dla różnych długości tablic, z krokiem co 500 elementów. Dla każdej długości tablicy zostaną zmierzone czasy wykonania algorytmów QuickSort w trzech wariantach

```
zakres_liczb = input('Podaj zakres losowanych liczb w postaci początek:koniec
')
start,stop = zakres_liczb.split(':')
start = int(start)
stop = int(stop)
długości_tablic=[]
czasyQuickSortLomutoRand=[]
czasyQuickSortHoare=[]
czasyQuickSortDutchFlag=[]

# Testowanie czasu wykonania dla QuickSortLomuto
for i in range(500,9001, 500):
    długości_tablic.append(i)
    arr = rand_list(i,start,stop)
    time_quicksort_lomuto = ileczasu(QuickSortLomutoRand, arr, 0, len(arr)-1)
    czasyQuickSortLomutoRand.append(time_quicksort_lomuto)
    print(f"Czas wykonania QuickSortLomuto dla tablicy o długości {len(arr)}:
{time_quicksort_lomuto}")

# Testowanie czasu wykonania dla QuickSortHoare
for i in range(500,9001, 500):
    arr = rand_list(i,start,stop)
    time_quicksort_hoare = ileczasu(QuickSortHoare, arr, 0, len(arr)-1)
    czasyQuickSortHoare.append(time_quicksort_hoare)
    print(f"Czas wykonania QuickSortHoare dla tablicy o długości {len(arr)}:
{time_quicksort_hoare}")

# Testowanie czasu wykonania dla QuickSortDutchFlag
for i in range(500,9001, 500):
    arr = rand_list(i,start,stop)
    time_quicksort_dutch_flag = ileczasu(QuickSortDutchFlag, arr, 0, len(arr)-1)
    czasyQuickSortDutchFlag.append(time_quicksort_dutch_flag)
    print(f"Czas wykonania QuickSortDutchFlag dla tablicy o długości {len(arr)}:
{time_quicksort_dutch_flag}")
```

- Wyniki będą prezentowane na wykresie, gdzie oś X będzie przedstawiała długość tablicy, a oś Y czas wykonania algorytmu. Każdy wariant algorytmu Quicksort będzie oznaczony innym kolorem na wykresie, co umożliwi porównanie ich efektywności.

  ```
  import matplotlib.pyplot as plt
  plt.plot(długości_tablic, czasyQuickSortLomutoRand, marker='o', linestyle='-', label='QuickSortLomutoRand')
  plt.plot(długości_tablic, czasyQuickSortHoare, marker='o', linestyle='-', label='QuickSortHoare')
  plt.plot(długości_tablic, czasyQuickSortDutchFlag, marker='o', linestyle='-', label='QuickSortDutchFlag')
  plt.title(f'Porównanie czasu wykonania trzech wariantów QuickSort na liczbach z zakresu: {zakres_liczb}')
  plt.xlabel('Długość tablicy')
  plt.ylabel('Czas wykonania (s)')
  plt.autoscale(axis='y')
  plt.legend()
  plt.grid(True)
  plt.show()
  ```

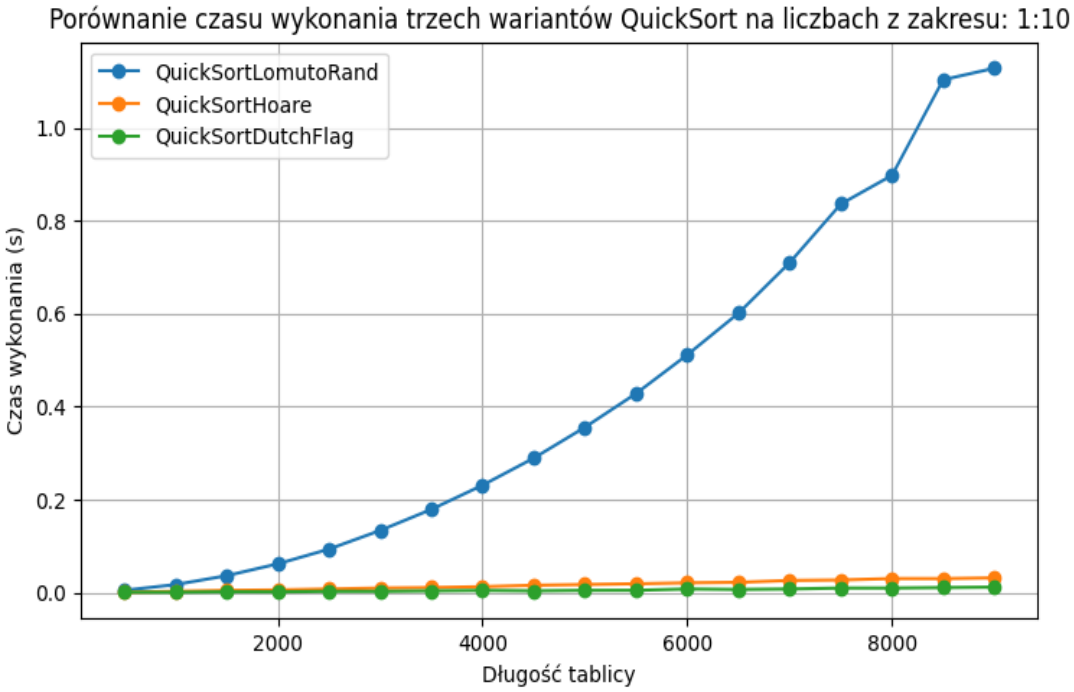
Przeprowadziłem testy wydajnościowe dla trzech różnych zakresów liczb losowych: 1:10, 1:1000 oraz 1:1000000. Pierwszy z tych zakresów charakteryzuje się dużą ilością powtórzeń poszczególnych liczb, co oznacza, że dane są bardziej skondensowane w określonym przedziale. Drugi zakres, 1:1000, reprezentuje sytuację, w której liczby są rozłożone równomiernie na całym przedziale, co skutkuje średnią liczbą powtórzeń. Natomiast trzeci zakres, 1:1000000, obejmuje duży zakres liczb, co powoduje, że powtórzenia pojawiają się bardzo rzadko. Wyniki testów dla tych zakresów prezentują się następująco:

- **Zakres liczb 1:10**

Wyniki z konsoli:

długość tablicy	QuickSortLomutoRand	QuickSortHoare	QuickSortDutchFlag
500	0,005137	0,000999	0,001128
1000	0,017435	0,002998	0,000999
1500	0,036601	0,004997	0,001998
2000	0,062097	0,005996	0,001868
2500	0,093780	0,007994	0,002998
3000	0,133899	0,009995	0,002999
3500	0,179263	0,010993	0,003998
4000	0,231275	0,012993	0,004864
4500	0,289482	0,015990	0,003997
5000	0,355889	0,017598	0,004998
5500	0,428552	0,019248	0,005138
6000	0,511701	0,021241	0,007998
6500	0,601246	0,022262	0,006995
7000	0,710011	0,026397	0,007995
7500	0,836090	0,027390	0,009994
8000	0,897830	0,030399	0,010012
8500	1,103462	0,030268	0,010994
9000	1,128309	0,031980	0,011993

Wykres:

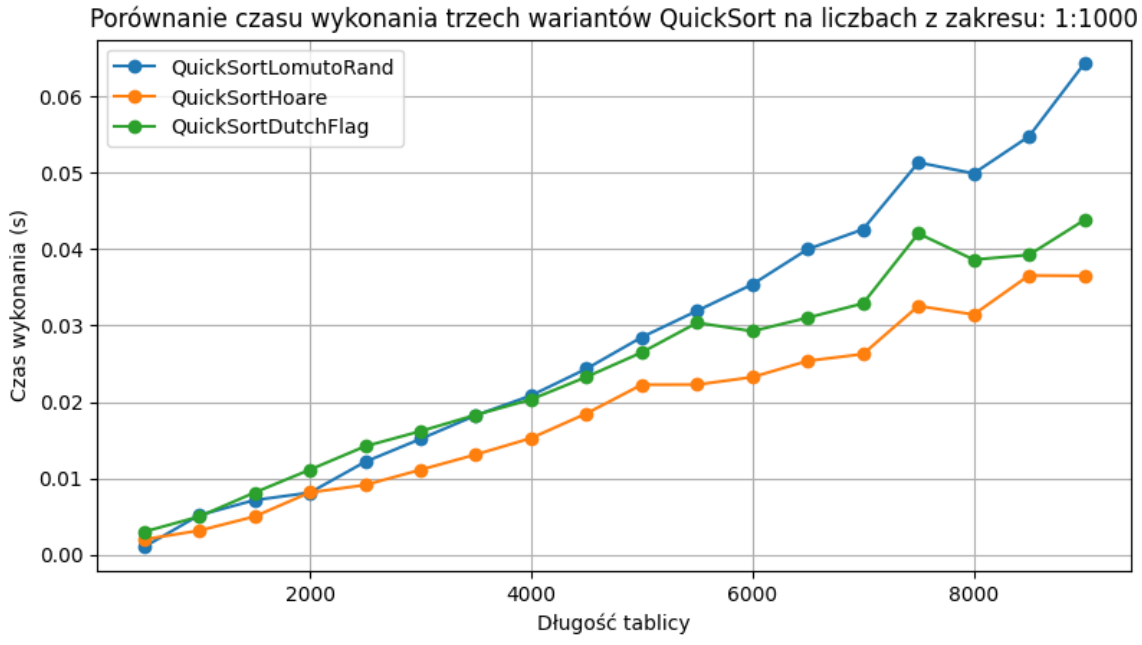


- **Zakres liczb 1:1000**

Wyniki z konsoli:

długość tablicy	QuickSortLomutoRand	QuickSortHoare	QuickSortDutchFlag
500	0,001000	0,001998	0,002997
1000	0,005174	0,003159	0,004998
1500	0,007145	0,004998	0,008132
2000	0,008123	0,008147	0,011129
2500	0,012163	0,009123	0,014198
3000	0,015154	0,011119	0,016171
3500	0,018247	0,013113	0,018307
4000	0,020848	0,015263	0,020330
4500	0,024350	0,018492	0,023282
5000	0,028480	0,022259	0,026487
5500	0,031936	0,022272	0,030372
6000	0,035399	0,023248	0,029267
6500	0,040004	0,025379	0,031034
7000	0,042606	0,026270	0,032900
7500	0,051351	0,032571	0,042057
8000	0,049927	0,031446	0,038635
8500	0,054772	0,036567	0,039258
9000	0,064302	0,036513	0,043813

Wykres:

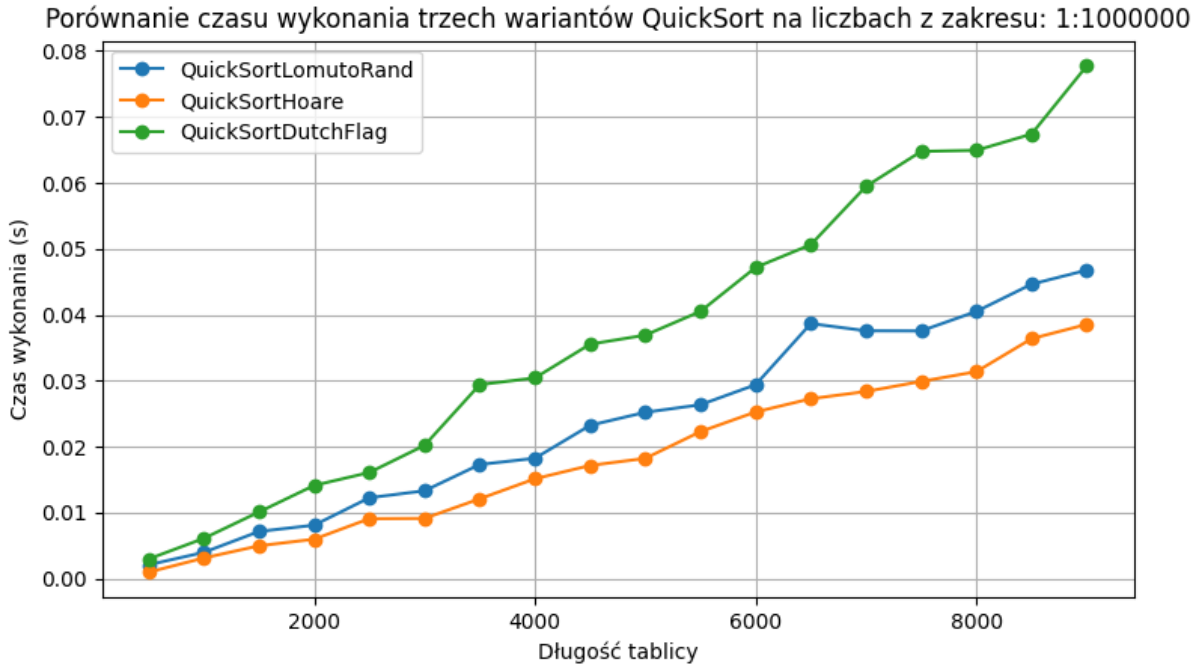


- **Zakres liczb 1:1000000**

Wyniki z konsoli:

długość tablicy	QuickSortLomutoRand	QuickSortHoare	QuickSortDutchFlag
500	0,002116919	0,001000643	0,00299716
1000	0,003999949	0,003147602	0,006122828
1500	0,007171154	0,00499773	0,010123491
2000	0,008125305	0,005998611	0,014148712
2500	0,012279749	0,009111166	0,016112328
3000	0,013338327	0,009150028	0,020221949
3500	0,017320156	0,012120962	0,029425383
4000	0,018262625	0,015154123	0,030440569
4500	0,023258686	0,017177105	0,035559177
5000	0,025233507	0,018248081	0,036925316
5500	0,026376009	0,022304773	0,040528297
6000	0,029412985	0,025286436	0,047188282
6500	0,038680553	0,02728796	0,05064249
7000	0,037589788	0,028385162	0,059477091
7500	0,037569523	0,029913902	0,064775705
8000	0,040523767	0,031392097	0,06494236
8500	0,044647455	0,036388874	0,067405701
9000	0,04677701	0,038561106	0,077730417

Wykres:



Wnioski:

1. **QuickSort z Partition w wersji Dutch flag**:
- Najlepszy przy dużej ilości powtarzających się wartości, ponieważ efektywnie grupuje te wartości, co przyspiesza proces sortowania.

- Jednakże najgorszy przy danych składających się z różnych i dużych liczb, ponieważ może być bardziej kosztowny obliczeniowo ze względu na konieczność porównywania dużej liczby unikalnych wartości.

2. **QuickSort z Partition w wersji Lomuto z losowaniem pivota**:
- Zdecydowanie najmniej wydajny przy małych zakresach liczb z dużą ilością powtórzeń, ponieważ generuje większą liczbę rekurencyjnych wywołań dla tych danych.

- Lepszy od Dutch Flag przy bardzo małej ilości powtórzeń i dużym zakresie.

3. **QuickSort z Partition w wersji Hoare**:
- Wydajniejszy od Lomuto we wszystkich przypadkach, ponieważ dokonuje mniejszej ilości porównań i zamiany elementów, co przekłada się na szybsze sortowanie.

- Sprawdza się dobrze przy każdym zakresie danych, ponieważ zachowuje równie dobrą wydajność zarówno dla dużych jak i małych zakresów liczb, bez względu na ilość powtórzeń.

Wnioski te sugerują, że wybór odpowiedniego wariantu Quicksorta zależy od charakterystyki danych wejściowych. Dla danych z dużą ilością powtórzeń warto rozważyć użycie Dutch Flag. Dla danych z małą ilością powtórzeń i dużym zakresie, Lomuto może być lepszy. Natomiast w ogólnych przypadkach Hoare sprawdza się najlepiej, ponieważ jest bardziej wydajny od Lomuto, niezależnie od charakterystyki danych wejściowych.