Spis treści

QuickSort i okolice

Projekt I na Algorytmy i Struktury danych

Kacper Sołtysiak

416853

Omówienie projektu

Celem projektu było porównanie czasu działania trzech klasycznych algorytmów sortujących: BubbleSort, InsertSort oraz QuickSort. Pomiarów dokonywano na listach, których długości tworzyły ciąg arytmetyczny: od 100 do 1000, ze stałym krokiem 100. Takie podejście umożliwiło obserwację, jak czas sortowania zmienia się wraz ze wzrostem rozmiaru danych.
W ramach eksperymentów przeprowadzono:

- pomiary czasu sortowania losowych danych każdym z algorytmów,
- test wpływu wstępnego uporządkowania danych – zarówno rosnącego (czyli „StepSort”), jak i malejącego (czyli odwrotnie posortowanego),
- analizę działania sortowania na danych częściowo uporządkowanych („StepSort”),
- porównanie wyników przy różnych zakresach wartości wejściowych: od 10 do 100000,
- oraz – zgodnie z wymaganiami projektu – test wersji QuickSorta wspomaganej InsertSortem dla krótkich podtablic (w tym przypadku: o długości 10 lub mniejszej).

Wszystkie testy zostały przeprowadzone w języku Python, a wyniki przedstawiono w formie wykresów i zestawień, które pozwalają łatwo zauważyć różnice w efektywności algorytmów. Choć InsertSort i BubbleSort mają swój spokojny, dydaktyczny urok, to QuickSort – szczególnie wspomagany InsertSortem – pokazuje pełnię swojej wydajności w praktyce.

Funkcje sortujące

Tak jak już wspomniano wcześniej, badania mieliśmy przeprowadzić dla trzech podstawowych funkcji podstawowych, tj;

- InsertSort
- BubbleSort
- QuickSort

Badania przeprowadzić również mieliśmy za pomocą funkcji hybrydowych, tj; InsertStepSort, który w kodzie nazwany został „StepSort” oraz QuickInsertSort który w kodzie ma nazwę: „QuickSortHybrid”. Do kodu została również dodana funkcja Partition, wspomagająca QuickSorta dzieląc tablice na dwie części względem losowanego pivota.

InsertSort

Funkcja ta jest prostym algorytmem sortującym, który wstawia kolejne elementy w odpowiednie miejsce w posortowanej części listy. Dobrze działa dla małych danych

```
8  def InsertSort(_list):          #Funkcja sortujaca przez wstawianie
9      for i in range(1, len(_list)):       #iteracja od drugiego elementu
10         temp = _list[i]         #chwilowe zapamietanie biezacego elementu
11         j = i - 1
12         while j >= 0 and _list[j] > temp:       #przesuwamy wieksze elementy w prawo
13             _list[j + 1] = _list[j]
14             j -= 1
15         _list[j + 1] = temp         #wstawiamy element na wlasciwe miejsce
```

BubbleSort

Klasyczny algorytm wielokrotnie przechodzący przez listę, zamieniając sąsiednie elementy miejscami w przypadku jeśli leża w złej kolejności. Wolny lecz stosunkowo prosty.

```
17 def BubbleSort(_list):          #funckja sortowania babelkowego
18     n = len(_list)              #dlugosc listy
19     for i in range(n):
20         for j in range(0, n - i - 1):       #porownanie elementow sasiadujacych
21             if _list[j] > _list[j + 1]:     #jesli sa w zlej kolejnosci
22                 _list[j], _list[j + 1], = _list[j + 1], _list[j]        #zamiana
```

QuickSort

Algorytm który z tej trójki jest najszybszy, lecz stosunkowo najcięższy. Algorytm dzieli listę na części mniejsze i większe od swojego pivota, czyli wybranego najczęściej losowo elementu, a następnie sortuje je rekurencyjnie aż do skutku.

```
36 def QuickSortR(_list, _start, _stop):       #rekurencyjna wersja QuickSorta
37     if _start < _stop:
38         p = Partition(_list, _start, _stop)     #podzial
39         QuickSortR(_list, _start, p - 1)        #sortowanie lewej czesci
40         QuickSortR(_list, p + 1, _stop)         #sortowanie prawej czesci
41
42 def QuickSort(_list):        #Funkcja startowa QuickSorta
43     QuickSortR(_list, 0, len(_list) - 1)
```

Poniżej również funkcja Partition która wspomaga QuickSorta w dzieleniu do pivota.

```
def Partition(_list, _start, _stop):                # funkcja podzialu do QuickSorta
    pivot_index = random.randint(_start, _stop)     #losowy pivot
    _list[pivot_index], _list[_stop] = _list[_stop], _list[pivot_index]     #liczba przebiegow
    pivot = _list[_stop]        #zapamietanie pivota
    i = _start
    for j in range(_start, _stop):          #iteracja po liscie
        if _list[j] < pivot or (_list[j] == pivot and random.randint(0, 2) == 1):           #1/3 szansy na przesuniecie
            _list[i], _list[j] = _list[j], _list[i]     #zamiana
            i += 1
    _list[i], _list[_stop] = _list[_stop], _list[i]
    return i            #zwrot pozycji pivota
```

QuickInsertSort

W drugim punkcie naszego projektu, do badań dołożyć mamy wersję QuickSorta która na krótkich dystansach wspomagana jest InsertSortem. Funkcja ta zdecydowanie radzi sobie prześwietnie, gdyż InsertSort oraz BubbleSort radzą sobie słabo przy większej ilości danych, natomiast QuickSort bywa nieefektywny przy małej ilości ze względu na swoją rekurencję. QuickInsertSort natomiast łączy atrybuty obu algorytmów stając się najlepszym z nich.

```
def QuickSortHybridR(_list, _start, _stop, threshold=10):       #quicksort + insertsort(dla malych przedzialow)
    if _stop - _start + 1 <= threshold:         #jesli maly fragment (w tym przypadku <10)
        for i in range(_start + 1, _stop + 1):          #sortowanie przez wstawianie (InsertSort)
            temp = _list[i]
            j = i - 1
            while j >= _start and _list[j] > temp:
                _list[j + 1] = _list[j]
                j -= 1
            _list[j + 1] = temp
    elif _start < _stop:                        # w przeciwnym razie zwykly QuickSort
        p = Partition(_list, _start, _stop)
        QuickSortHybridR(_list, _start, p - 1, threshold)
        QuickSortHybridR(_list, p + 1, _stop, threshold)

def QuickSortHybrid(_list):         #Funkcja startowa QuickInsertSorta
    QuickSortHybridR(_list, 0, len(_list) - 1)
```

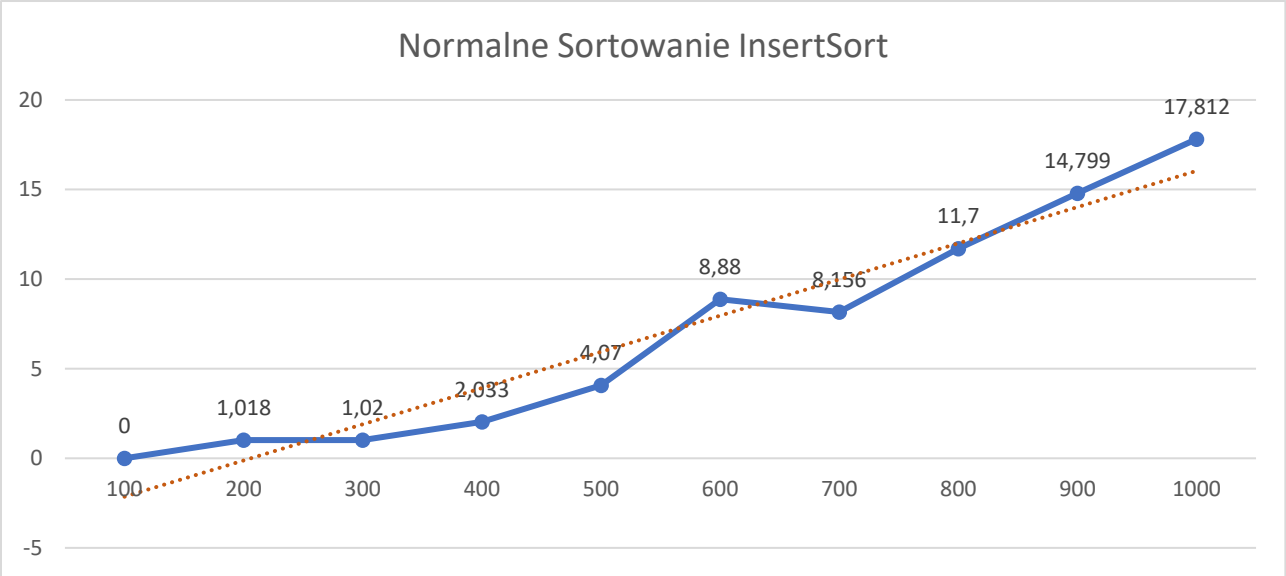
Badania porównawcze czasu sortowania

Badania przeprowadzono na 10 listach o długościach od 100 do 1000, o skoku równym 100. Stanowi więc to ciąg arytmetyczny o n = 10, a1 = 100, r=100. Zakres z którego losowano liczby stanowił zakres (0, 10000). Testy przeprowadzono za pomocą funkcji „pomiar_czasu”, która w moim kodzie wygląda następująco:

```
def pomiar_czasu(func, data):           #pomiara czasu dzialania funkcji
    start = time.time()     #poczatek czasu
    func(data)          #wywolanie
    end = time.time()       #koniec czasu
    return (end - start) * 1000  # ms           #zwrot czasu w milisekundach
```

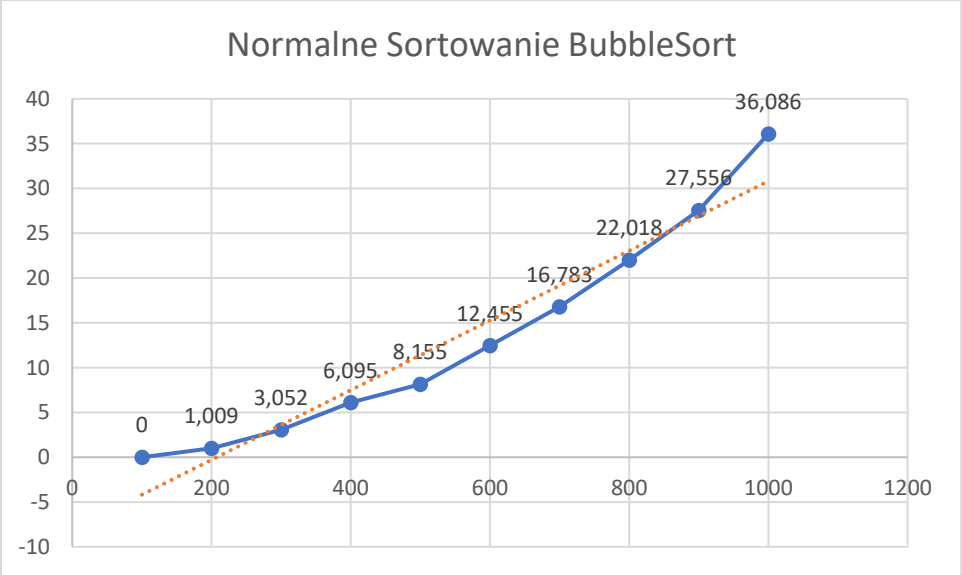
InsertSort

Jak widać na poniższym wykresie, czas sortowania przy 100 elementach listy był natychmiastowy, natomiast przy 1000 elementach wynosił już 17,8 ms. Jak na dziesięciokrotny skok w ilości elementów do przesortowania to dość diametralna różnica. Miedzy wyrazami naszego ciągu arytmetycznego (pomiędzy danymi dwoma sąsiadującymi ilościami elementów) różnica jest niewielka. Linia trendu natomiast wyraźnie wskazuje na wzrost czasu sortowania wraz ze wzrostem ilości elementów listy.



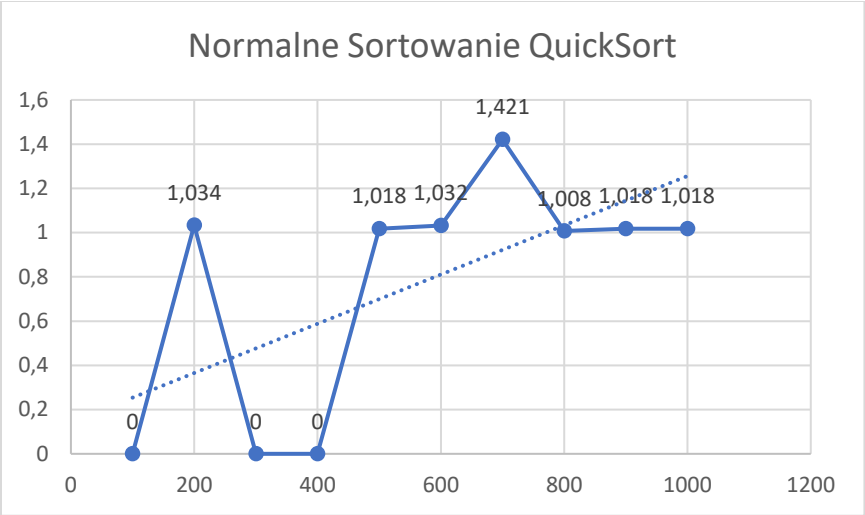
BubbleSort

Przy bąbelkowym sortowaniu, linia trendu również wskazuje mocny wzrost czasu analogicznie do przyrostu ilości elementów do przesortowania. Czas pierwszego sortowania również jest na tyle mały, że zbliżony do zera przy wartościach z 3 cyframi po przecinku, natomiast ostatni sort trwa już aż 36 ms.



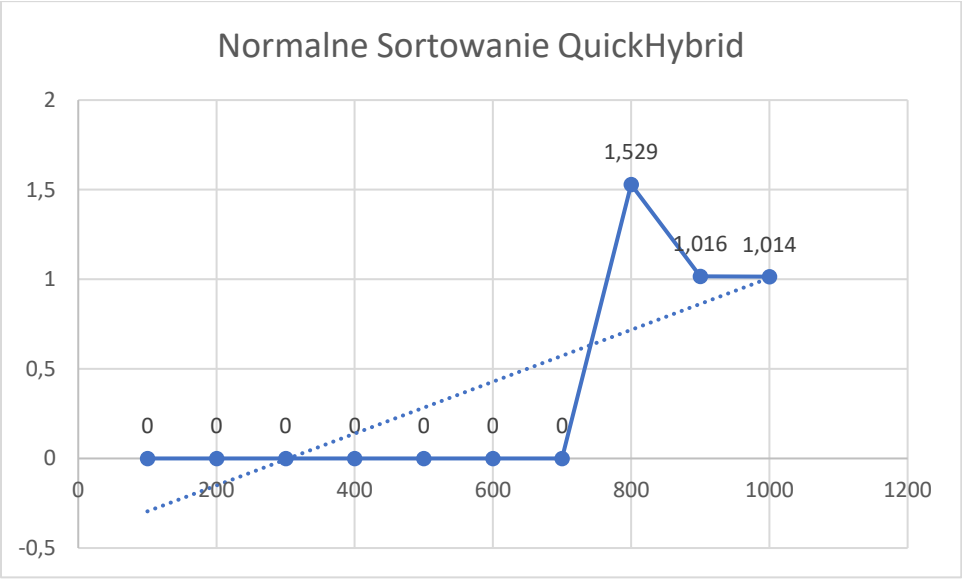
QuickSort

Jak widać poniżej, algorytm ten radzi sobie na tyle dobrze z takim zakresem, że badania są niemiarodajne. Przy niektórych testach czas wynosi 0, a przy reszcie zbliżony jest do 1ms. Wywnioskować można z tego, iż algorytm QuickSort jest najlepszy z całej trójki. Sortuje na tyle szybko, że jest to niemalże natychmiastowe.

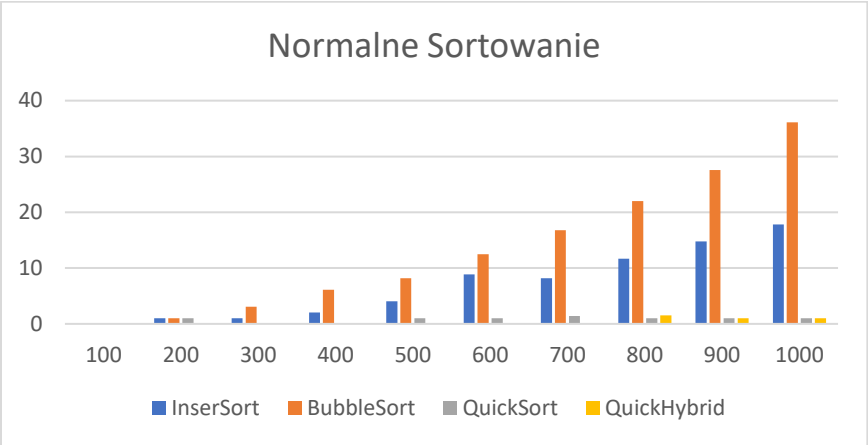


QuickInsertSort

Jak wspomniałem wcześniej, algorytm ten stanowczno najlepiej radzi sobie z całej gromady. Problemy pojawiają się dopiero przy górnych granicach ilości elementów, poniżej 800 sortuje je natychmiastowo. W dalszym etapie czas sortowań nie zbliża się nawet do 2ms, co tylko utwierdza nas w tym przekonaniu.



Porównanie algorytmów



Normalne sortowanie

n	InserSort	BubbleSort	QuickSort	QuickHybrid
100	0	0	0	0
200	1,018	1,009	1,034	0
300	1,02	3,052	0	0
400	2,033	6,095	0	0
500	4,07	8,155	1,018	0
600	8,88	12,455	1,032	0
700	8,156	16,783	1,421	0
800	11,7	22,018	1,008	1,529
900	14,799	27,556	1,018	1,016
1000	17,812	36,086	1,018	1,014

Wstępne posortowanie

Kolejnym etapem naszego projektu, miało być sprawdzenie jak radzą sobie nasze algorytmy z tablicami które spełniają pewne zadane wymogi. Na pierwszy ogień sprawdzić mamy czas sortowań ówcześnie posortowanych już tablic. Efekt ten uzyskamy sortując kopię tablicy funkcją StepSort, która sortuje tablice co 10 elementów InsertSortem. Następnie posortowaną już w ten sposób tablicę wrzucimy innym algorytmom do posortowania. Pod koniec porównamy czas który im to zajął.

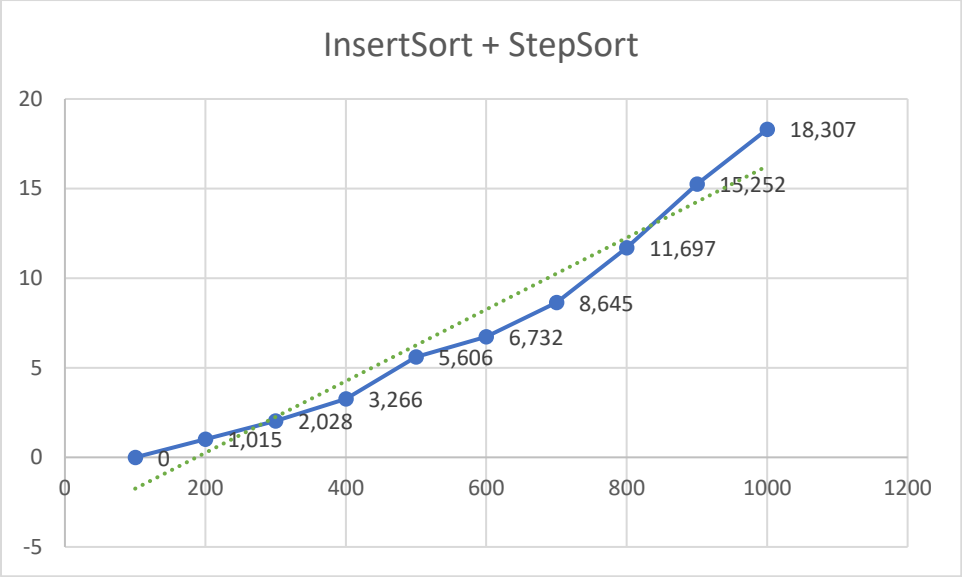
```
113     # b) Po StepSorcie
114     stepsort_wyniki = []        #lista na wyniki po stepsorcie
115     for n in dlugosci:
116         lista = [random.randint(0, 10000) for i in range(n)]        #losowanie listy w przedziale
117         base = lista.copy()     #kopia listy
118         StepSort(base)      #stepne sortowanie co 10
119         t1 = pomiar_czasu(InsertSort, base.copy())
120         t2 = pomiar_czasu(BubbleSort, base.copy())
121         t3 = pomiar_czasu(QuickSort, base.copy())
122         t4 = pomiar_czasu(QuickSortHybrid, base.copy())
```

StepSort - Funkcja sortująca tablicę co step elementów za pomocą InsertSorta.

```
65
66  def StepSort(_list, step=10):                #wstepne sortowanie co 10 elementow
67      for i in range(0, len(_list), step):
68          end = min(i + step, len(_list))      #ustalam koniec fragmentu
69          InsertSort(_list[i:end])             #sortujemy ten fragment przez wstawianie
```

InsertSort

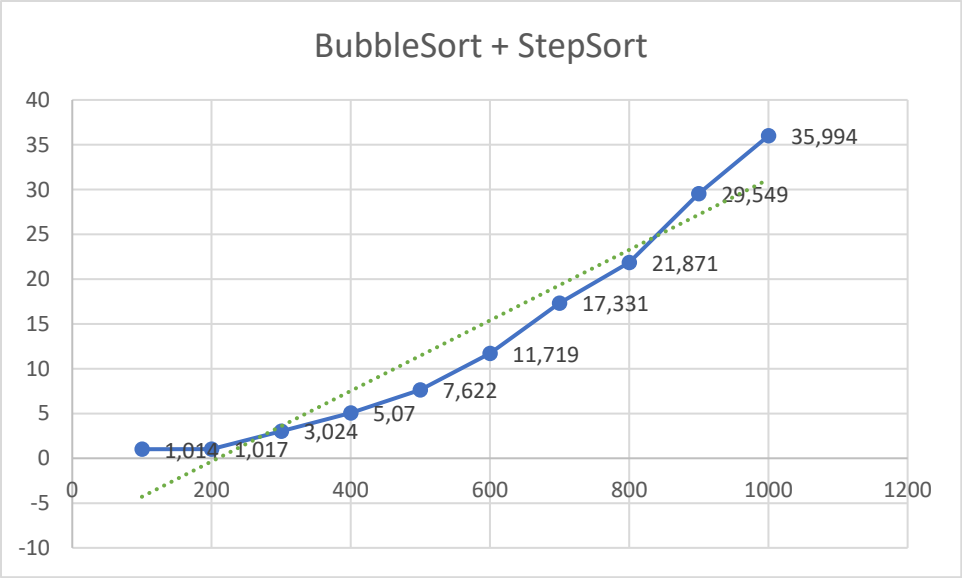
```
119          t1 = pomiar_czasu(InsertSort, base.copy())
```



Zakres (0, 10000) dla 10 tablic (100, 200, ..., 1000)

BubbleSort

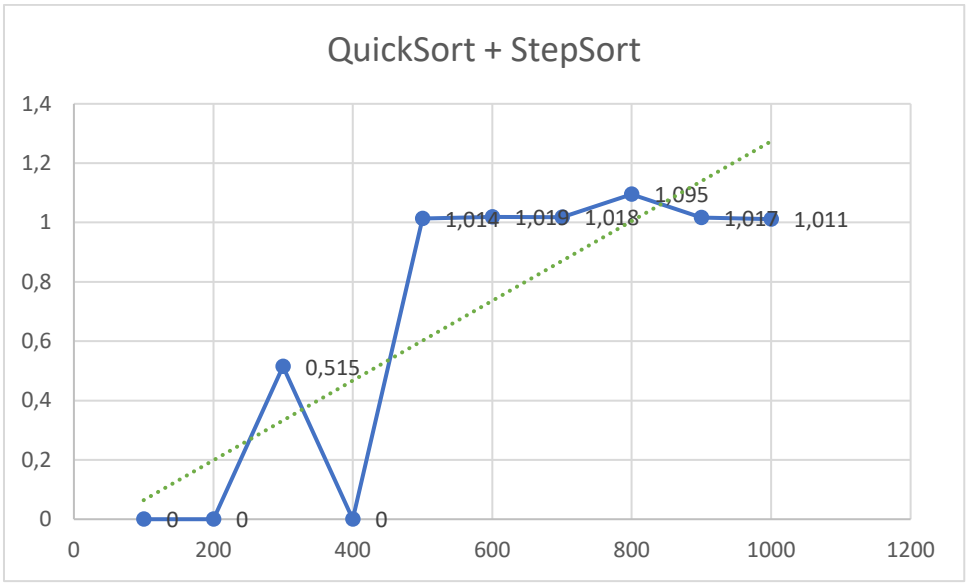
```
120          t2 = pomiar_czasu(BubbleSort, base.copy())
```



Zakres (0, 10000) dla 10 tablic (100, 200, ..., 1000)

QuickSort

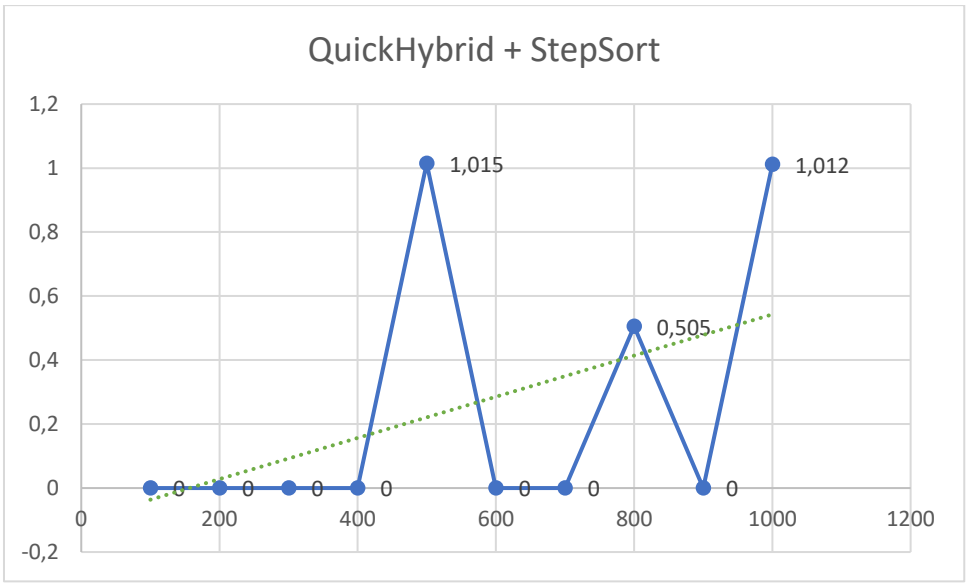
```
121          t3 = pomiar_czasu(QuickSort, base.copy())
```



Zakres (0, 10000) dla 10 tablic (100, 200, …, 1000)

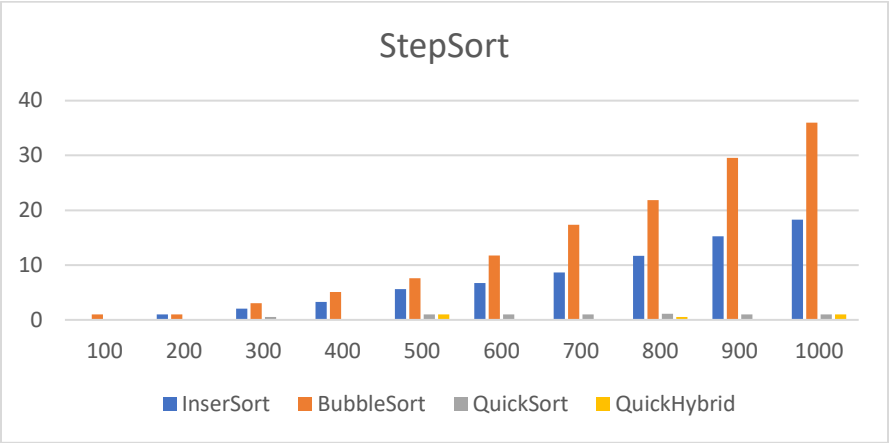
QuickInsertSort

```
122          t4 = pomiar_czasu(QuickSortHybrid, base.copy())
```



Zakres (0, 10000) dla 10 tablic (100, 200, …, 1000)

Porównanie algorytmów



StepSort				
n	InserSort	BubbleSort	QuickSort	QuickHybrid
100	0	1,014	0	0
200	1,015	1,017	0	0
300	2,028	3,024	0,515	0
400	3,266	5,07	0	0
500	5,606	7,622	1,014	1,015
600	6,732	11,719	1,019	0
700	8,645	17,331	1,018	0
800	11,697	21,871	1,095	0,505
900	15,252	29,549	1,017	0
1000	18,307	35,994	1,011	1,012

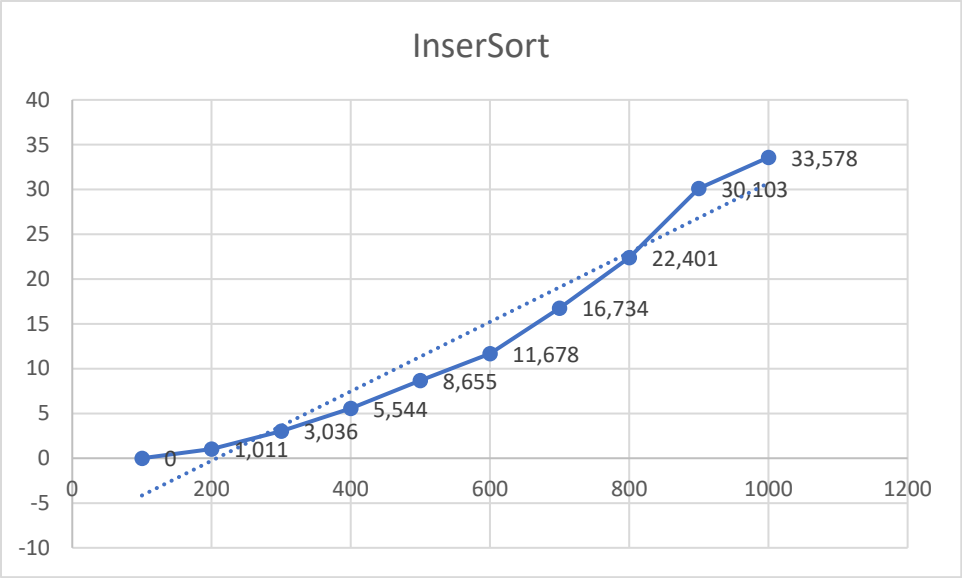
Jak widać powyżej, BubbleSort radzi sobie najgorzej z całej czwórki. QuickSortHybrid jest znakomity, zwykły QuickSort nieco wolniejszy natomiast InsertSort pozostaje z lekka w tyle.

Wstępne posortowanie w odwrotnym kierunku

W drugim podpunkcie zadania 1, posortować kopie tablic mamy wstępnie w odwrotnym kierunku co dla niektórych algorytmów takich jak InsertSort czy BubbleSort – może okazać się wyjątkowo niekorzystne. Algorytmy te źle radzą sobie z listami uporządkowanymi rosnąco w złej kolejności, ponieważ każdy element musi być przesunięty na początek, co generuje bardzo dużo operacji porównań i przestawień. Sprawdzimy zatem, jak znacząco pogarsza się ich czas działania w tej sytuacji i porównamy go z czasami sortowania przez QuickSort oraz QuickSortHybrid, które dzięki rekurencyjnemu dzieleniu listy powinny poradzić sobie lepiej nawet w niekorzystnym przypadku.

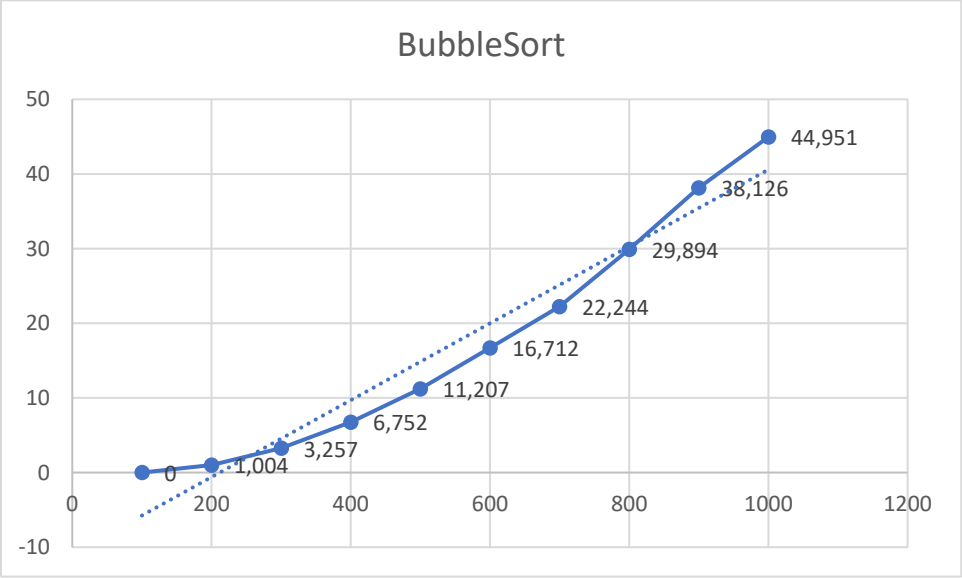
```
126     # c) Lista posortowana odwrotnie
127     reverse_wyniki = []     #lista na wyniki dla odwrotnego sortowania
128     for n in dlugosci:
129         lista = sorted([random.randint(0, 10000) for i in range(n)], reverse=True)  #losowanie odwrotnej listy
130         t1 = pomiar_czasu(InsertSort, lista.copy())
131         t2 = pomiar_czasu(BubbleSort, lista.copy())
132         t3 = pomiar_czasu(QuickSort, lista.copy())
133         t4 = pomiar_czasu(QuickSortHybrid, lista.copy())
```

InsertSort



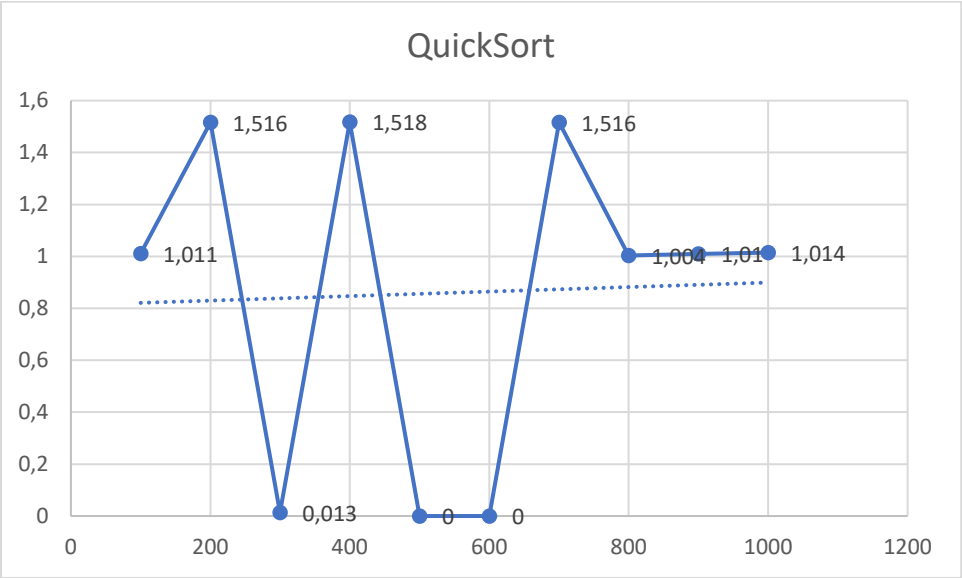
Po powyższym wykresie łatwo wywnioskować iż InsertSort jest bardzo wrażliwy na tablice posortowane w odwrotnym kierunku. Zaczynając od 1,011ms przy 100 elementach dochodzimy aż do 33,578ms przy 1000 elementach co znaczy iż czas jego pracy drastycznie rośnie.

BubbleSort



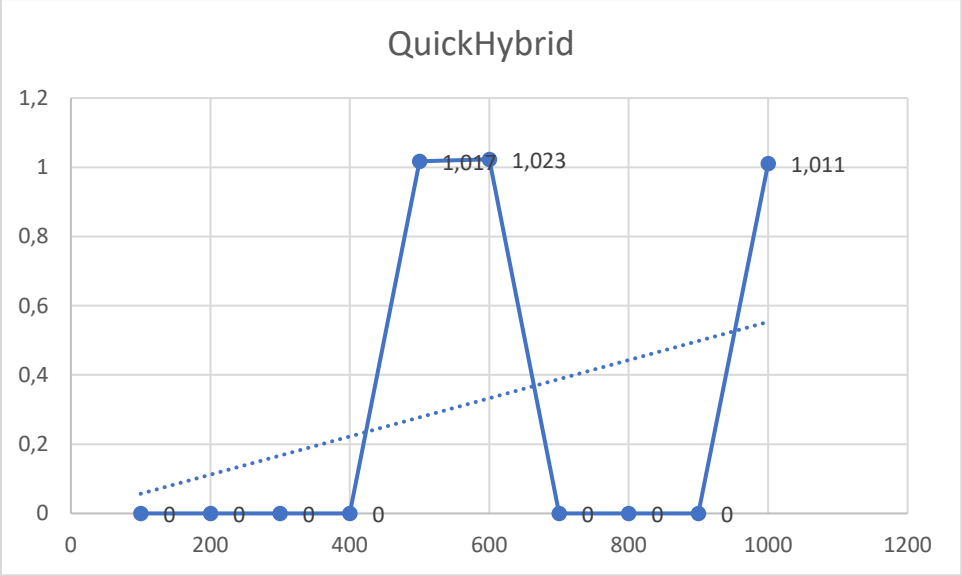
Tutaj sytuacja powtarza się jak z jego poprzednikiem. BubbleSort nie cierpli tablic posortowanych w ten sposób. Zaczynając od niemalże 1ms dochodzi aż do 45ms! Okropnie długi czas jak na jedynie 1000 elementów.

QuickSort



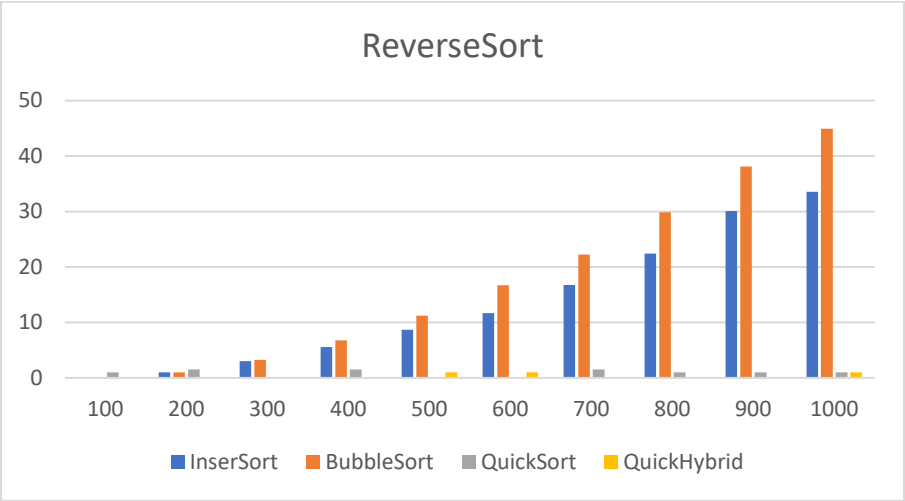
Przy tak niekorzystnym ustawieniu danych, QuickSort daje sobie świetnie radę. Jego czasy pracy nie wybiegają poza 1,5ms. Nawet przy największej tablicy, 1000 elementowej jego czas pracy to niemalże 1ms.

QuickSortHybrid



Jak już wiemy, hybryda ta działa świetnie przy małych jak i dużych tablicach. Czas pracy nieprzekraczający 1ms przy takiej różnorodności tablic. Świetny wynik.

Porównanie algorytmów



RverseSort

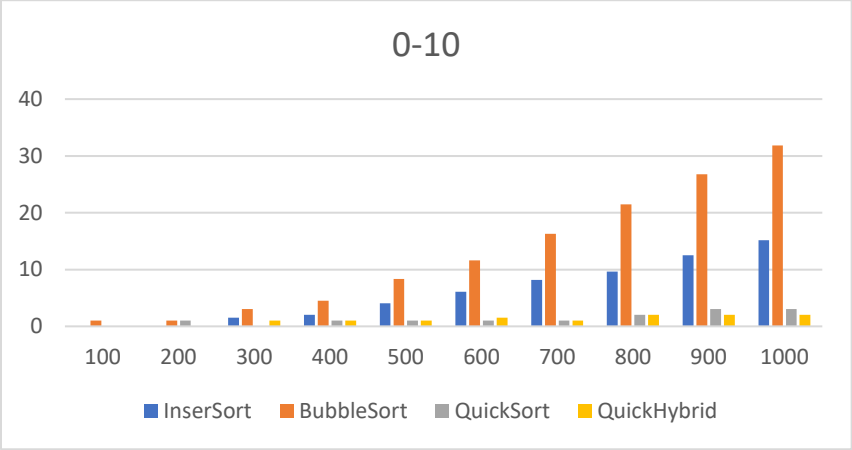
n	InserSort	BubbleSort	QuickSort	QuickHybrid
100	0	0	1,011	0
200	1,011	1,004	1,516	0
300	3,036	3,257	0,013	0
400	5,544	6,752	1,518	0
500	8,655	11,207	0	1,017
600	11,678	16,712	0	1,023
700	16,734	22,244	1,516	0
800	22,401	29,894	1,004	0
900	30,103	38,126	1,01	0
1000	33,578	44,951	1,014	1,011

W przypadku odwrotnie posortowanych danych klasyczne algorytmy takie jak InsertSort i BubbleSort wypadają bardzo słabo, osiągając czasy kilkadziesiąt razy dłuższe niż nowoczesne podejścia. QuickSort utrzymuje dobrą wydajność, ale to QuickSortHybrid zdecydowanie dominuje, łącząc zalety dwóch podejść i eliminując ich słabości. Wybór odpowiedniego algorytmu ma więc kluczowe znaczenie w zależności od rodzaju danych wejściowych.

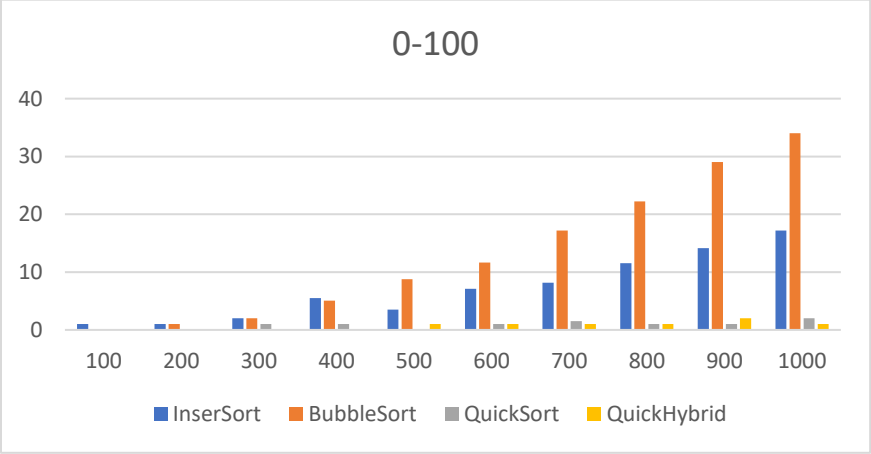
Porównanie czasów sortowań przy różnych zakresach z którego losowane są liczby.

W kolejnym, ostatnim już podpunkcie, przyszło nam sprawdzić, czy zakres losowanych liczb ma jakiś wpływ na działanie naszych algorytmów sortujących.
Wcześniej nasze dane były generowane losowo, ale zawsze w tym samym, domyślnym zakresie.
Teraz zamierzamy to zmienić – sprawdzimy, co się stanie, gdy liczby będą losowane z dużo mniejszego zakresu (1-10) a następnie z bardzo szerokiego (1-10000).
Zobaczymy, czy to jakoś zmienia czasy działania — może niektóre algorytmy lepiej sobie radzą, gdy jest dużo powtórzeń, albo odwrotnie – gdy każdy element jest inny.

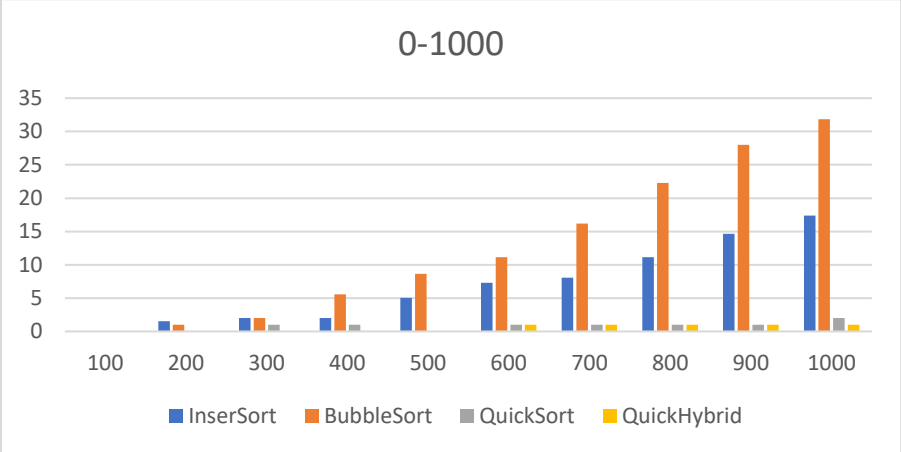
Zakres 1-10



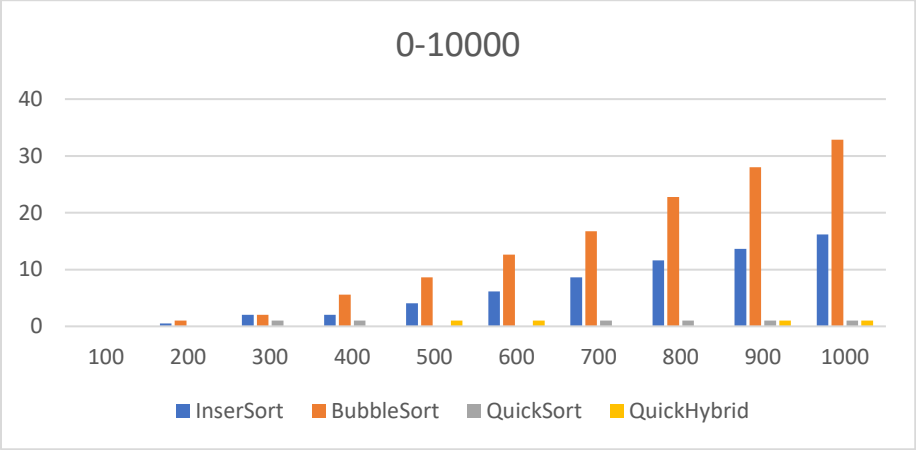
Zakres 1-100



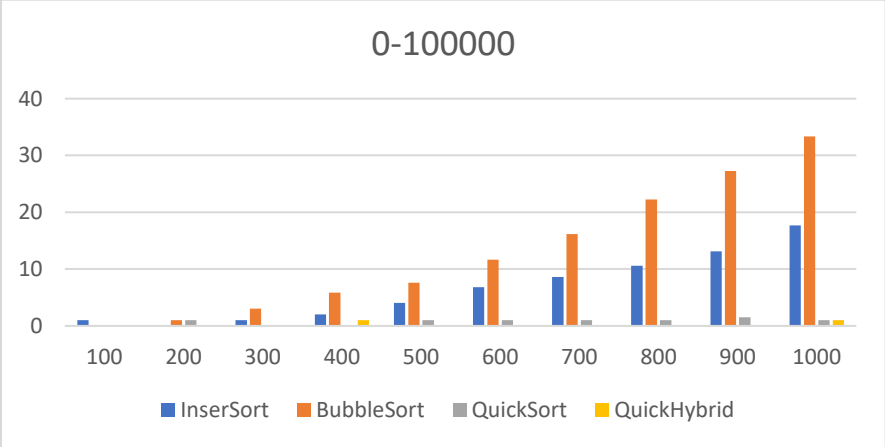
Zakres 1-1000



Zakres 1-10000



Zakres 1-100000



Wnioski

InsertSort oraz BubbleSort, wykazują ogromną wrażliwość na zakres danych, im większy, tym więcej unikalnych liczb i więcej operacji do wykonania, co drastycznie przedłuża czas pracy tych algorytmów. Dla małych zakresów radziły sobie świetnie, gdyż wiele liczb się powtarzało. Natomiast QuickSort i QuickHybrid niezależnie od zakresu danych, są mega optymalne i odporne na bałagan w danych wejściowych. Poradziły sobie świetnie w każdym przypadku. Nastomiast widać jednak lekką przewagę naszej hybrydy ze względu na możliwość użycia InsertSorta przy małych tablicach. Osiągnał on najlepsze czasy w każdym z zakresów.
Mówiąc w prost – zmiana zakresów miała największy wpływ na proste algorytmy sortujące, które nie są zoptymalizowane pod kątem różnorodności danych.

Podsumowanie

- QuickSort i QuickHybrid są zdecydowanie najszybsze spośród testowanych algorytmów. Czas ich działania jest bardzo niski, nawet przy dużych zbiorach danych.
- InsertSort i BubbleSort radzą sobie wyłącznie z małymi lub wstępnie uporządkowanymi danymi – przy większych listach ich czas działania rośnie gwałtownie.
- Wstępne sortowanie rosnące (StepSort):
  - znacząco przyspiesza działanie InsertSorta i BubbleSorta,
  - nie daje istotnych korzyści przy QuickSorcie – jest zbyt szybki, żeby to miało sens,
  - QuickHybrid zyskuje minimalnie.

- Wstępne sortowanie malejące (odwrotne):
  - bardzo pogarsza wydajność InsertSorta i BubbleSorta (nawet kilkukrotnie dłuższe czasy),
  - QuickSort i QuickHybrid są praktycznie niewrażliwe na taki układ danych.
  - QuickSortHybrid (QuickInsertSort) wypada najlepiej w większości testów – łączy szybkość QuickSorta i skuteczność InsertSorta przy małych fragmentach listy.
- Zakres losowanych danych ma znaczenie:
  - dla InsertSorta i BubbleSorta większy zakres oznacza więcej unikalnych wartości →więcej pracy
  - QuickSort i QuickHybrid radzą sobie równie dobrze niezależnie od zakresu (od 1–10 do 1–100000).
- Wnioski praktyczne
  - przy sortowaniu dużych zbiorów lub danych o nieznanym porządku zdecydowanie najlepiej sprawdza się QuickSort lub jego hybryda z InsertSortem
  - proste algorytmy (BubbleSort, InsertSort) warto stosować jedynie dla małych, uporządkowanych list.

Cały kod programu wraz z numerami linii

```
 1  import random
 2  import time
 3
 4  # --------------------
 5  # Funkcje sortujące
 6  # --------------------
 7
 8  def InsertSort(_list):
 9      for i in range(1, len(_list)):
10          temp = _list[i]
11          j = i - 1
12          while j >= 0 and _list[j] > temp:
13              _list[j + 1] = _list[j]
14              j -= 1
15          _list[j + 1] = temp
16
17  def BubbleSort(_list):
18      n = len(_list)
19      for i in range(n):
20          for j in range(0, n - i - 1):
21              if _list[j] > _list[j + 1]:
```

```
22              _list[j], _list[j + 1] = _list[j + 1], _list[j]
23
24  def Partition(_list, _start, _stop):
25      pivot_index = random.randint(_start, _stop)
26      _list[pivot_index], _list[_stop] = _list[_stop], _list[pivot_index]
27      pivot = _list[_stop]
28      i = _start
29      for j in range(_start, _stop):
30          if _list[j] < pivot or (_list[j] == pivot and random.randint(0, 2) == 1):
31              _list[i], _list[j] = _list[j], _list[i]
32              i += 1
33      _list[i], _list[_stop] = _list[_stop], _list[i]
34      return i
35
36  def QuickSortR(_list, _start, _stop):
37      if _start < _stop:
38          p = Partition(_list, _start, _stop)
39          QuickSortR(_list, _start, p - 1)
40          QuickSortR(_list, p + 1, _stop)
41
42  def QuickSort(_list):
43      QuickSortR(_list, 0, len(_list) - 1)
44
45  def QuickSortHybridR(_list, _start, _stop, threshold=10):
46      if _stop - _start + 1 <= threshold:
47          for i in range(_start + 1, _stop + 1):
48              temp = _list[i]
49              j = i - 1
50              while j >= _start and _list[j] > temp:
51                  _list[j + 1] = _list[j]
52                  j -= 1
53              _list[j + 1] = temp
54      elif _start < _stop:
```

```
55          p = Partition(_list, _start, _stop)
56          QuickSortHybridR(_list, _start, p - 1, threshold)
57          QuickSortHybridR(_list, p + 1, _stop, threshold)
58
59  def QuickSortHybrid(_list):
60      QuickSortHybridR(_list, 0, len(_list) - 1)
61
62  # --------------------
63  # StepSort
64  # --------------------
65
66  def StepSort(_list, step=10):
67      for i in range(0, len(_list), step):
68          end = min(i + step, len(_list))
69          InsertSort(_list[i:end])
70
71  # --------------------
72  # Pomiar czasu
73  # --------------------
74
75  def pomiar_czasu(func, data):
76      start = time.time()
77      func(data)
78      end = time.time()
79      return (end - start) * 1000  # ms
80
81  # --------------------
82  # Wydruk tabeli
83  # --------------------
84
85  def print_table(tytul, naglowek, wyniki):
86      print(f"\n\n{tytul}")
87      print("=" * len(tytul))
```

```
    print(naglowek)
    print("-" * len(naglowek))
    for wiersz in wyniki:
        print(wiersz)

# --------------------
# Główna część programu
# --------------------

if __name__ == "__main__":
    dlugosci = list(range(1000, 10001, 1000))

    normalne_wyniki = []
    for n in dlugosci:
        lista = [random.randint(0, 10000) for i in range(n)]
        t1 = pomiar_czasu(InsertSort, lista.copy())
        t2 = pomiar_czasu(BubbleSort, lista.copy())
        t3 = pomiar_czasu(QuickSort, lista.copy())
        t4 = pomiar_czasu(QuickSortHybrid, lista.copy())
        normalne_wyniki.append(f"n = {n:<4} | {t1:9.3f} ms | {t2:9.3f} ms | {t3:9.3f} ms | {t4:9.3f} ms")
    print_table("normalnene sortowanie", "n    | Insert    | Bubble    | Quick    | QuickHybrid", normalne_wyniki)

    stepsort_wyniki = []
    for n in dlugosci:
        lista = [random.randint(0, 10000) for i in range(n)]
        base = lista.copy()
        StepSort(base)
        t1 = pomiar_czasu(InsertSort, base.copy())
        t2 = pomiar_czasu(BubbleSort, base.copy())
        t3 = pomiar_czasu(QuickSort, base.copy())
        t4 = pomiar_czasu(QuickSortHybrid, base.copy())
```

```
119         stepsort_wyniki.append(f"n = {n:<4} | {t1:9.3f} ms | {t2:9.3f} ms | {t3:9.3f} ms | {t4:9.3f} ms")
120     print_table("StepSort (wstępne posortowanie co 10)", "n      | Insert    | Bubble    | Quick     | QuickHybrid", stepsort_wyniki)
121
122     reverse_wyniki = []
123     for n in dlugosci:
124         lista = sorted([random.randint(0, 10000) for i in range(n)], reverse=True)
125         t1 = pomiar_czasu(InsertSort, lista.copy())
126         t2 = pomiar_czasu(BubbleSort, lista.copy())
127         t3 = pomiar_czasu(QuickSort, lista.copy())
128         t4 = pomiar_czasu(QuickSortHybrid, lista.copy())
129         reverse_wyniki.append(f"n = {n:<4} | {t1:9.3f} ms | {t2:9.3f} ms | {t3:9.3f} ms | {t4:9.3f} ms")
130     print_table("Odwrotnie posortowana lista", "n      | Insert    | Bubble    | Quick     | QuickHybrid", reverse_wyniki)
131
132     zakresy = [10, 100, 1000, 10000, 100000]
133     for maks in zakresy:
134         zakresy_wyniki = []
135         for n in dlugosci:
136             lista = [random.randint(0, maks) for i in range(n)]
137             t1 = pomiar_czasu(InsertSort, lista.copy())
138             t2 = pomiar_czasu(BubbleSort, lista.copy())
139             t3 = pomiar_czasu(QuickSort, lista.copy())
140             t4 = pomiar_czasu(QuickSortHybrid, lista.copy())
141             zakresy_wyniki.append(f"n = {n:<4} | {t1:9.3f} ms | {t2:9.3f} ms | {t3:9.3f} ms | {t4:9.3f} ms")
142         print_table(f"range wartości: 0 - {maks}", "n      | Insert    | Bubble    | Quick     | QuickHybrid", zakresy_wyniki)
```