2024

Badanie efektywności wybranych algorytmów sortowania ze względu na złożoność obliczeniową

ROBERT PYTEL (272931)

Spis treści

Sortowanie w informatyce

Sortowanie jest jednym z podstawowych problemów rozpatrywanych w informatyce. Polega ono na uporządkowaniu elementów listy według określonych wytycznych czy kryteriów. Zagadnienie to było badane oraz eksplorowane od początku ery komputerowej. Algorytmy sortowania mogą wydawać się szczególnie atrakcyjne ze względu na złożoność rozwiązania, pomimo pozornie prostego sformułowania problemu, jakim wydaje się być uporządkowanie elementów w określonej kolejności. Jednym z prostszych algorytmów sortowania, którego rozwiązanie sprawia wrażenie bardzo intuicyjnego, jest sortowanie bąbelkowe. Według niektórych źródeł, wzmianki na temat analizy efektywności tego algorytmu można datować na okolice roku 1956. Na przestrzeni kilkudziesięciu lat opracowano wiele różnych algorytmów zarówno tych mniej jak i bardziej efektywnych. W tej pracy zostały opisane następujące algorytmy sortowania: sortowanie przez wstawianie, sortowanie przez kopcowanie, sortowanie Shella oraz sortowanie szybkie.

Tak jak w przypadku innych dziedzin algorytmów informatyki do pomiaru złożoności obliczeniowej algorytmów sortowania używa się miary asymptotycznego tempa wzrostu, lepiej znanej jako notacji dużego O. Matematyczna definicja tej miary prezentuje się następująco:

Mówimy, że f jest co najwyżej rzędu g, gdy istnieją takie $n_0 > 0$*, oraz* $c > 0$*, że:*

$$\forall\, n \geq n_0 : f(n) \leq c * g(n)$$

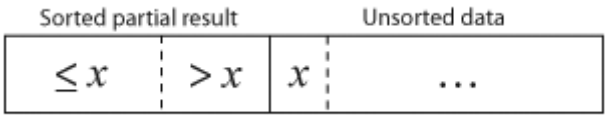
Można to zinterpretować w następujący sposób. Dla wszystkich argumentów, które są większe bądź równe n_0, funkcja f(n) przyjmuje wartości nie większe niż wartości funkcji g(n) przemnożone przez pewną stałą c. Jaki można wysunąć z tego wniosek? Funkcja f(n) nie rośnie szybciej niż funkcja g(n).

Sortowanie przez wstawianie

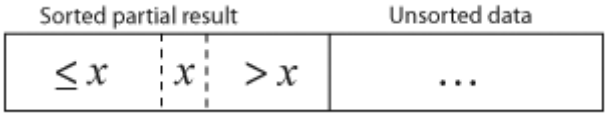
Opis algorytmu

Z angielskiego *Insertion sort,* to jeden z prostszych algorytmów sortowania, którego użycie w kontekście komputerów można datować na rok 1945 i przypisuje Konradowi Zuse, niemieckiemu inżynierowi, pionierowi informatyki. Sortowanie przez wstawianie jest algorytmem iteracyjnym, który wykorzystuje operację porównywania wartości sąsiadujących liczb. Okazuje się, że schemat sortowania przez wstawianie opiera się na tej samej idei wykorzystywanej w pewnej grze karcianej. Mowa tutaj o brydżu, w którym zawodnik w celu uporządkowania kart posiadanych w ręce dokonuje sekwencji zamian zbliżonej do tej w opisywanym algorytmie.

Główna idea algorytmu polega na podziale tablicy na dwie części: częściowo uporządkowaną oraz nieposortowaną. W każdym kroku algorytmu pojedynczy element trafia z części nieposortowanej do części uporządkowanej, poprzez proces znalezienia odpowiedniego miejsca dla przenoszonego elementu w tablicy częściowo posortowanej. Czynności te powtarzamy do momentu, w którym wszystkie elementy z części nieposortowanej zostaną przeniesione do części uporządkowanej. Poniższe ilustracje obrazują wyżej przedstawioną sytuację:



Rysunek 1 Tablica przed umieszczeniem elementu x w części posortowanej
https://upload.wikimedia.org/wikipedia/commons/3/32/Insertionsort-before.png



Rysunek 2 Tablica po wykonaniu procesu wstawiania elementu x do części uporządkowanej
https://upload.wikimedia.org/wikipedia/commons/d/d9/Insertionsort-after.png

Pseudokod implementujący wyżej przedstawiony schemat postępowania prezentuje się następująco:

```
i ← 1
while i < length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
    i ← i + 1
end while
```

Jedną z własności powyższego algorytmu jest fakt, że po każdej iteracji pętli zewnętrznej pierwsze *i* elementów tablicy jest posortowane.

Analiza złożoności obliczeniowej

W przypadku sortowania przez wstawianie najgorszym przypadkiem początkowego ustawienia tablicy są dane posortowane malejąco, największy element na pierwszym miejscu, najmniejszy na ostatnim. Mając do czynienia z takim scenariuszem, proces znalezienia optymalnego miejsca w części uporządkowanej tablicy będzie wymagał wykonania *i* operacji zamian. To oznacza, że algorytm będzie miał złożoność kwadratową, w notacji dużego O przedstawianą jako $O(n^2)$.

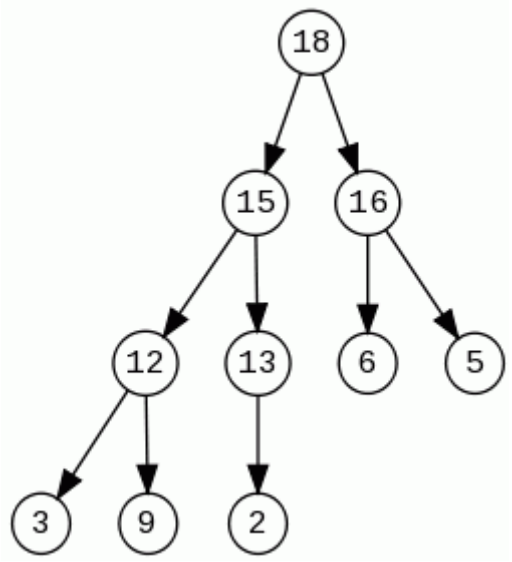
Jeżeli mowa o złożoności obliczeniowej dla przypadku średniego to również wynosi ona $O(n^2)$, co sprawia, że algorytm sortowania przez wstawianie nie jest efektywnym algorytmem w przypadku tablic o dużych rozmiarach. Jednakże w przypadku tablic o małej ilości elementów jest on bardzo efektywny. Z tego względu, w niektórych implementacjach jednego z najbardziej wydajnych algorytmów sortowania, jakim jest sortowanie szybkie, w przypadku gdy tablica wejściowa jest małych rozmiarów, używany jest schemat sortowania przez wstawianie.

Poza tym algorytm sortowania przez wstawianie cechuje się następującymi własnościami: jest wykonywany w miejscu (ang. *In place*) co oznacza, że dodatkowa pamięć wymagana do algorytmu jest stała, jest to algorytm stabilny, co oznacza, że elementy posiadające te same wartości będą miały taką samą kolejność w tablicy wynikowej jak w tablicy wejściowej.

Sortowanie przez kopcowanie

Opis algorytmu

Kopiec (ang. *heap*) to struktura danych oparta na drzewie, w której występuje pewna relacja między węzłami. Otóż drzewo jest kopcem w momencie gdy wartość węzła rodzica jest nie mniejsza od wartości węzłów potomków. W przypadku sortowania przez kopcowanie, kopiec musi spełniać dodatkowo własność prawie pełności: warunek ten jest spełniony w momencie gdy liście występują na ostatnim i ewentualnie przedostatnim poziomie w drzewie oraz liście na ostatnim poziomie są spójnie ułożone od strony lewej do prawej.



Rysunek 3 Przykład kopca zupełnego
https://lh3.googleusercontent.com/proxy/KIvN2gyx5Aas_Xx7RkWDlRm_6ECBvn2lhG6JnptxpHIJOSqYSZkoW8QywOezYqY5-sH4kjsH3SOD4W4ZAN0WKpOopDZ9vjTCEg7LtqAfVA

Jednym ze sposobów reprezentacji kopca jest umieszczenie go w tablicy. Dzięki temu rozpatrywanie wzajemnych relacji między elementami kopca staje się bardzo proste. W przypadku tablicy indeksowanej od zera można zastosować następujące wzory:

$$p = \left\lfloor \frac{i-1}{2} \right\rfloor$$
$$l = 2p + 1$$
$$r = 2p + 2$$
$$gdzie:$$
$$p - indeks\ rodzica$$
$$i - indeks\ wierzchołka$$
$$l - indeks\ lewego\ potomka$$
$$r - indeks\ prawego\ potomka$$

Skoro została już wyjaśniona struktura kopca, należy zastanowić się w jaki sposób może ona zostać użyta do posortowania danych. Własnością, która umożliwia dokonanie tej operacji, jest relacja mniejszości, która jak już zostało wcześniej wspomniane określa, że wartość rodzica nie może być mniejsza od wartości potomka. Dzięki temu w korzeniu kopca mamy węzeł, który posiada największą wartość w całym drzewie. Algorytm sortowania (ang. *heapsort*) przez kopcowanie w początkowej fazie zakłada utworzenie maksymalnego kopca na tablicy z danymi

wejściowymi. W tym celu można wykorzystać algorytm Floyda do tworzenia kopca. Po utworzeniu kopca w korzeniu znajduje się wartość maksymalna w całej tablicy. Następnym krokiem jest zamiana korzenia wraz z ostatnim elementem tablicy oraz zmniejszenie rozmiaru kopca o jeden. Dokonana zamiana mogła zaburzyć relacje mniejszości kopca, zatem trzeba dokonać tak zwanej naprawy kopca w dół, czyli procesu, który ma za zadanie przywrócenie własności kopca. Po pomyślnie wykonanej naprawie w korzeniu powinna znajdować się druga największa wartość w danych do posortowania. Wymienione wyżej kroki powtarzamy aż do momentu, w którym struktura kopca obejmuje tylko jeden element.

Ogólna lista kroków dla algorytmu sortowania przez kopcowanie może zostać zapisana w następujący sposób:

```
Krok 1: utwórz kopiec
Krok 2: zamień korzeń wraz z ostatnim elementem i zmniejsz wielkość
kopca
Krok 3: napraw kopiec
Krok 4: powtarzaj krok 2 dopóki kopiec zawiera elementy
```

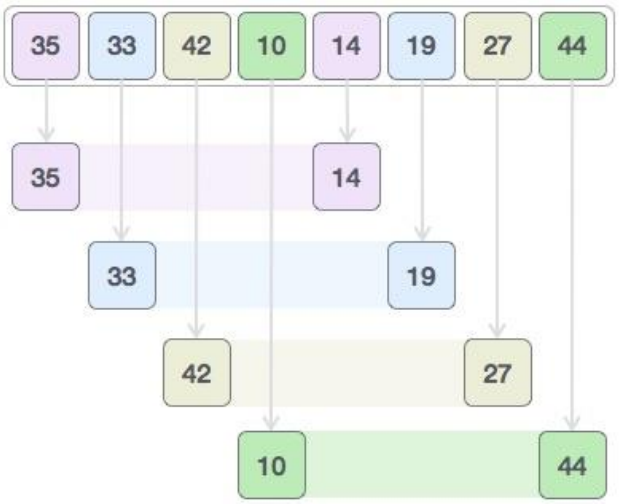
Analiza złożoności obliczeniowej

Na podstawie powyższego opisu krokowego możliwa jest ocena pesymistycznej złożoności obliczeniowej algorytmu. Jeżeli literą n określimy wielkość kopca to operacja naprawy kopca może dokonać maksymalnie $\log_2 n$ zamian. Naprawy kopca będziemy dokonywać n razy, zatem finalna złożoność wynosi $\mathrm{n} * \log_2 n$, którą przy pominięciu stałych można wyrazić jako $O(n \log n)$. Średnia złożoność obliczeniowa algorytmu wynosi również $O(n \log n)$. Algorytm sortowania przez kopcowanie jest wykonywany w miejscu oraz jest niestabilny. W kwestii wydajności konkuruje on z algorytmem szybkiego sortowania, gdzie przewagę zyskuje przy pesymistycznej złożoność obliczeniowa, która dla *quicksorta* wynosi $O(n^2)$. Dzięki swojej stabilności, sortowanie z użyciem kopca wykorzystywane jest w systemach czasu rzeczywistego oraz w systemach wbudowanych.

Sortowanie Shella

Opis algorytmu

Algorytm sortowania opracowany przez Donalda Shella w 1959 roku jest zmodyfikowaną wersją algorytmu sortowania przez wstawianie. Podstawową wadą *insertion sort* jest jego pesymistyczna złożoność obliczeniowa w przypadku gdy element o małej wartości znajduje się na końcu tablicy. Musi wtedy zostać przeprowadzona seria wielu operacji zamiany i porównań. Aby zapobiec tego typu sytuacjom algorytm sortowania Shella wprowadza operację sortowania par liczb oddalonych od siebie o daną odległość. Wraz z kolejnymi iteracjami pętli odległość ta jest zmniejszana w wyniku czego sortowana jest coraz to większa ilość elementów oddalonych od siebie o mniejszą liczbę pozycji. Gdy luka osiągnie wartość jeden dochodzimy do sytuacji, w której wykonujemy zwykły algorytm sortowania przez wstawianie. Jednak seria przeprowadzonych wcześniej operacji częściowych sortowań zbilansowała rozłożenie liczb w tablicy, dzięki czemu algorytm sortowania przez wstawianie będzie działał o wiele wydajniej.



Rysunek 4 Idea sortowania Shella
https://www.tutorialspoint.com/data_structures_algorithms/images/shell_sort_gap_4.jpg?w=144

Pseudokod dla wyżej opisanego schematu prezentuje się następująco. Potwierdza to pokrewieństwo algorytmu sortowania Shella wraz z algorytmem sortowania przez wstawianie.

```
for gap in gaps:
        i = gap
        while i < length(A)
                j ← i
                while j >= gap and A[j-gap] > A[j]
                        swap A[j] and A[j-gap]
                        j ← j - gap
                end while
                i ← i + 1
        end while
```

Kluczową częścią algorytmu jest tablica *gaps* zawierająca kolejne wartości odstępów używanych do porównywania liczb zaczynając od dużych odległości a kończąc na wartości jeden. Dobór wartości kolejnych odstępów ma diametralny wpływ na złożoność obliczeniową algorytmu.

Analiza złożoności obliczeniowej

Złożoność obliczeniowa algorytmu sortowania Shella jest niejednoznaczna i zależy od dobranych wartości odległości. Twórca algorytmu jak i jego następcy próbowali opracować takie sekwencje liczb, dla których algorytm osiągał najlepsze rezultaty. Początkowa sekwencja opracowana przez Donalda Shella miała mieć postać $\left\lfloor \frac{N}{2^K} \right\rfloor$, co oznaczało iteracyjne dzielenie długości tablicy na pół. Okazuje się jednak, że sekwencja ta osiąga fatalne rezultaty i dla pesymistycznego przypadku wynosi $O(n^2)$. Na przestrzeni lat naukowcy badali różne sekwencje liczb w celu polepszenia kwadratowej złożoności obliczeniowej. Jak się okazało pozytywne rezultaty były możliwe do osiągnięcia. Dwie sekwencje uwzględnione w programie projektu to: sekwencja Knutha oraz sekwencja Sedgewicka. Pierwsza z nich opracowana w 1973 charakteryzuje się ciągiem:

$$\frac{3^k - 1}{2}, gdzie$$
$$obliczona\ wartość\ nie\ może\ być\ większa\ niż\ \left\lceil \frac{N}{3} \right\rceil,$$
$$N\ to\ długość\ tablicy$$
$$k\ to\ numer\ wyrazu$$

Pesymistyczna złożoność obliczeniowa wynosi w tym przypadku $O\left(n^{3/2}\right)$.

Kolejną sekwencją, która osiąga lepsze rezultaty jest sekwencja Sedgewicka opracowana w 1982 roku. Jej pierwszym wyrazem jest liczba 1, natomiast kolejne wartości można obliczyć na podstawie wzoru:

$$4^k + 3 * 2^{k-1} + 1, gdzie$$
$$k\ to\ numer\ wyrazu$$

Pesymistyczna złożoność obliczeniowa dla tej sekwencji wynosi $O\left(n^{4/3}\right)$.

Sortowanie szybkie

Opis algorytmu

Z angielskiego *quicksort,* to jeden z najbardziej popularnych i efektywnych algorytmów sortowania. Został opracowany przez Tonyego Hoare, brytyjskiego informatyka, w 1959 roku, natomiast datę publikacji przypisuje się na rok 1961. Algorytm szybkiego sortowania wykorzystuje w swoim działaniu jeden z paradygmatów projektowania algorytmów a mianowicie *dziel i zwyciężaj*. Metoda ta zakłada podział struktury danych na coraz to mniejsze części i wykonywanie określonego procesu na mniejszych porcjach. W przypadku algorytmu sortowania szybkiego wymyślono następujący schemat działania. Mając do dyspozycji tablicę liczb należy wybrać jeden z jej elementów, który będzie określany mianem *pivota*. Liczba ta będzie dzieliła zbiór na dwa podzbiory w taki sposób, że liczby mniejsze od *pivota* znajdą się po jego lewej stronie, natomiast liczby większe od *pivota* po jego prawej stronie. Proces przemieszczania elementów tablicy względem wyznaczonej liczby nosi nazwę partycjonowania. Po pomyślnie przeprowadzonym partycjonowaniu pivot jest ustawiony w takim miejscu, które dzieli tablicę na dwie podtablice tak jak zostało to wyżej opisane. Kolejnym krokiem algorytmu szybkiego sortowania jest powtórzenie operacji partycjonowania dla obu podtablic. Operacja ta jest zaimplementowana w sposób rekurencyjny tak długo, aż rozmiar podtablicy będzie większy od jedynki.

Pseudokod dla algorytmu szybkiego sortowania prezentuje się w następujący sposób:

```
algorithm quicksort(A, lo, hi) is
  if lo >= 0 && hi >= 0 && lo < hi then
    p := partition(A, lo, hi)
    quicksort(A, lo, p)
    quicksort(A, p + 1, hi)
```

Funkcja *partition* może przyjmować różne implementacje. Popularnymi wariantami są schematy partycjonowania Lomuto oraz Hoare. W programie projektowym został zaimplementowany drugi wariant, który wykorzystuje dwa wskaźniki. Idea polega na ustawieniu lewego wskaźnika na początku tablicy, natomiast prawego na końcu. Lewy wskaźnik szuka elementów większych od wartości *pivota* natomiast prawy wskaźnik poszukuje elementów mniejszych od wartości *pivota*. Jeżeli wskaźniki znajdą owe liczby a wskaźnik lewy nie minął jeszcze wskaźnika prawego to oznacza, że liczby wskazywane przez wskaźniki można zamienić miejscami. Operacja wykonywana jest do momentu, w którym wskaźniki miną siebie, jeden z nich będzie wyznaczał wtedy miejsce, w którym ma finalnie wylądować *pivot*. Algorytm sortowania szybkiego jest wykonywany w miejscu, oraz w większości implementacji jest niestabilny.

Analiza złożoności obliczeniowej

Jeżeli mowa o najbardziej pesymistycznym przypadku, algorytm sortowania szybkiego przyjmuje złożoność rzędu $O(n^2)$. Sytuacja ta ma miejsce kiedy główna tablica dzielona jest na podtablice, w których jedna z nich ma tylko jeden element. Może to nastąpić w wyniku doboru złego pivota np. liczby najmniejszej lub największej. Inne przypadki obejmują konkretne implementacje funkcji partycjonowania. Przykładem gdy algorytm osiąga złożoność kwadratową jest scenariusz dla schematu partycjonowania Lomuto, gdy wszystkie elementy w tablicy są sobie równe. Złożoność dla przypadku średniego algorytmu sortowania szybkiego wynosi $O(n\ logn)$.

Plan eksperymentu

Założenia początkowe

Celem przeprowadzonego doświadczenia jest sprawdzenie czasów działania wymienionych na wstępie algorytmów sortowania. Maszyna użyta w tym celu jest wyposażona w procesor Intel Core i5-3320M, którego częstotliwość taktowania wynosi 2.6 GHz. Kod aplikacji został napisany w języku programowania C++ oraz skompilowany dla 64 bitowej architektury procesora. Najważniejszym parametrem rozpatrywanym w przypadku sortowania jest rozmiar tablicy, która zawiera dane wejściowe. W związku z tym wyznaczono eksperymentalnie siedem wartości, dla których pomiary czasów sortowania dla wyżej wspomnianego komputera były mierzone w milisekundach: 15000, 30000, 50000, 100000, 200000, 500000, 1000000. Pomiary wykonane dla wszystkich algorytmów korzystały z tablic zawierających 4 bajtowe liczby całkowite. W przypadku algorytmu sortowania przez kopcowanie zbadano również wpływ typu danych sortowanych elementów na czas wykonania operacji. Pomiary przeprowadzono dla trzech typów danych: *integer, float* oraz *char*.

Czas sortowania danych jest również zależny od początkowego ułożenia elementów tablicy. W związku z tym program umożliwia wybranie czterech różnych stanów początkowych: tablica posortowana rosnąco, tablica posortowana malejąco, 33% elementów posortowanych, 66% elementów posortowanych oraz tablica z losowym ułożeniem elementów. W pierwszych dwóch przypadkach użyto algorytmu sortowania szybkiego w celu uporządkowania elementów, natomiast w przypadku trzecim oraz czwartym użyto zmodyfikowanego algorytmu sortowania bąbelkowego, który w zależności od zadanego parametru, posortuje tylko określoną liczbę elementów.

Kolejnym aspektem jest sposób dokonywania pomiaru czasu sortowania elementów. Pojedynczy pomiar jest obarczony dużym błędem i daje mało wiarygodny wynik. Ze względu na to pomiary zostały wykonane stukrotnie, dodatkowo generując nowy zestaw danych dla każdej iteracji. Finalny wynik, który został uwzględniony w sprawozdaniu, jest średnią arytmetyczną stu pomiarów. Do zmierzenia czasu wykorzystano C++ o nazwie std::chrono::high_resolution_clock. Pomiar obejmuje tylko i wyłącznie czas działania algorytmu sortowania.

Przebieg badania

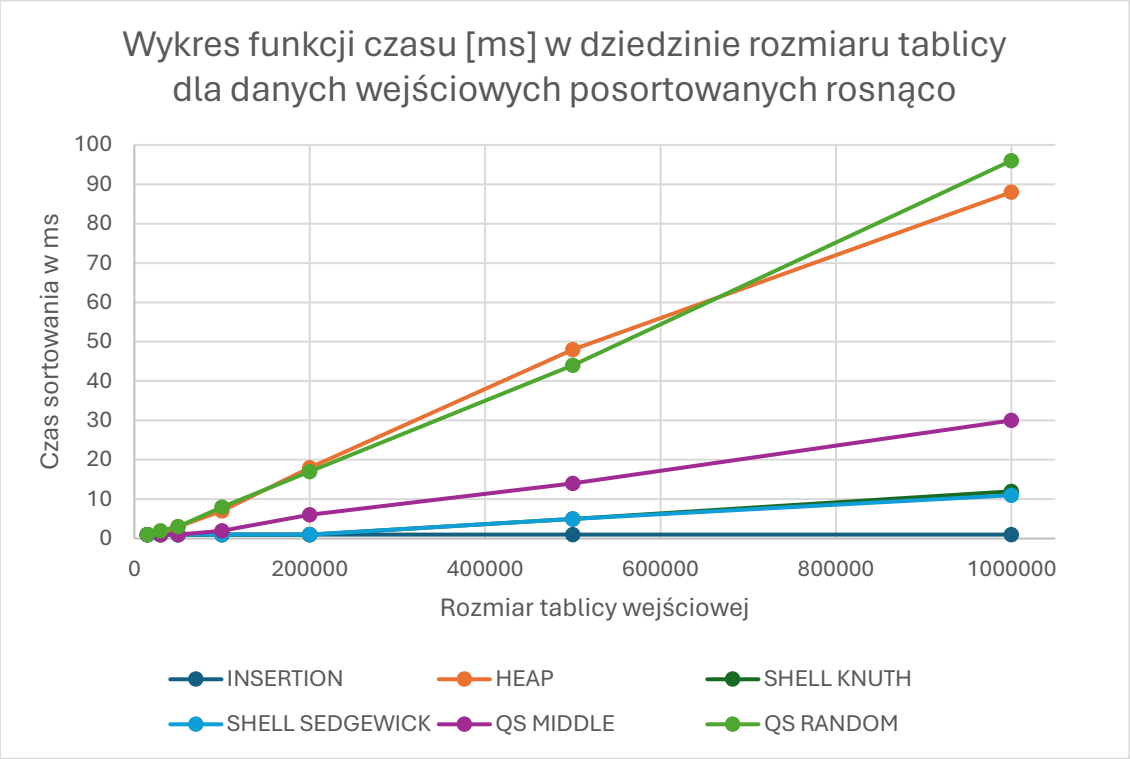
Program umożliwia wygenerowanie danych określonego typu oraz o określonym rozmiarze. Ręczne wykonanie 100 pomiarów jest w związku z tym możliwe, jednakże zajęcie to byłoby bardzo mozolne i czasochłonne. W związku z tym do menu programu dodano jedną z funkcji, która umożliwia programowe wykonanie testów wydajnościowych. W pierwszym kroku funkcja ta wykonuje 100 pomiarów czasu sortowania wybranego algorytmu. Następnie funkcja oblicza średnią arytmetyczną wykonanych pomiarów i zwraca uzyskany wynik użytkownikowi w postaci komunikatu konsolowego. Operacje te wykonywane są dla siedmiu reprezentatywnych wielkości tablic wymienionych w poprzedniej części rozdziału.

Pomiary zostały przeprowadzone dla pięciu możliwych ułożeń początkowych danych w tablicy, dla wszystkich ośmiu wariantów algorytmów sortowania. Wykonanie niektórych z nich zajęło bardzo dużo czasu obliczeniowego ze względu na pesymistyczną charakterystykę danych wejściowych np. dane posortowane malejąco dla algorytmu sortowania przez wstawianie, dane posortowane rosnąco dla algorytmu sortowania szybkiego. Tabele oraz wykresy zawierające wyniki pomiarów zostały przedstawione oraz omówione poniżej.

Dane wejściowe posortowane rosnąco

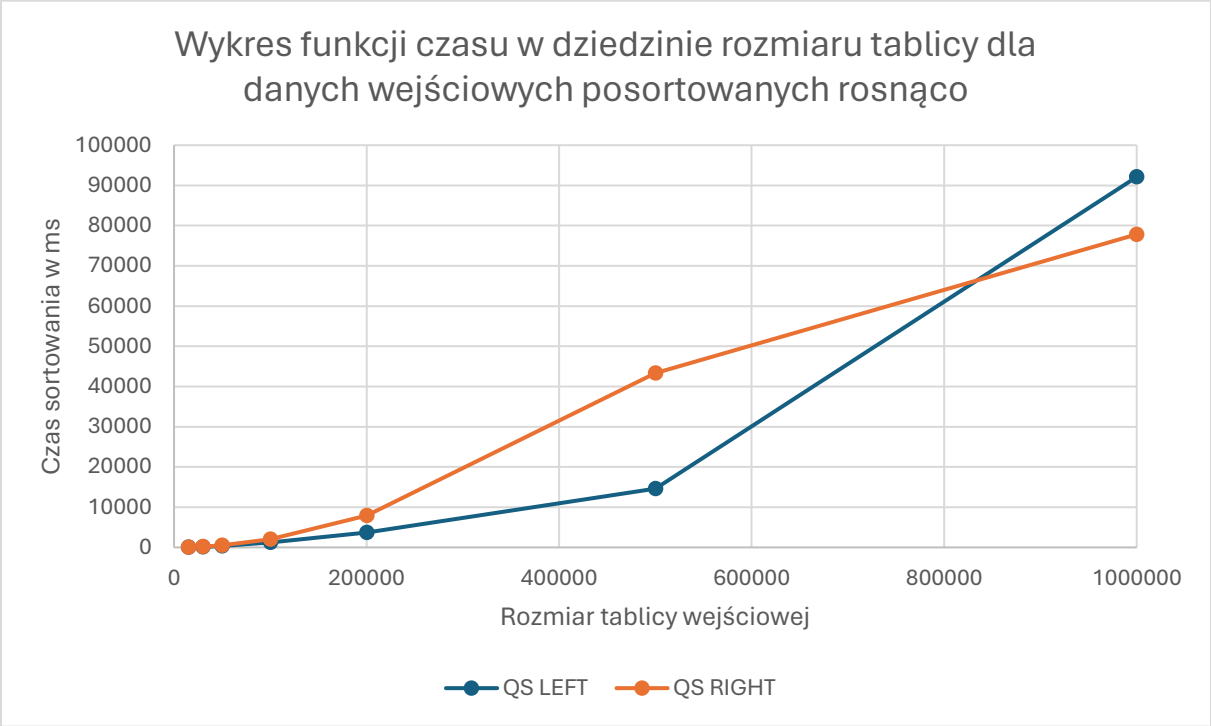
Tabela 1 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego algorytmu sortowania. Dane wejściowe posortowane rosnąco

	Insertion	Heap	Shell Knuth Sequence	Shell Sedgewick Sequence	Quicksort Left Pivot	Quicksort Right Pivot	Quicksort Middle Pivot	Quicksort Random Pivot
15000	1	1	1	1	42	48	1	1
30000	1	1	1	1	158	200	1	2
50000	1	3	1	1	387	538	1	3
100000	1	7	1	1	1233	2060	2	8
200000	1	18	1	1	3680	7922	6	17
500000	1	48	5	5	14577	43373	14	44
1000000	1	88	12	11	92106	77830	30	96



Powyższa tabela zawiera wyniki czasów sortowania w zależności od rozmiaru tablicy oraz wybranego algorytmu sortowania. W tym przypadku dane wejściowe zostały posortowane rosnąco. Najgorzej wypadły dwa warianty sortowania szybkiego: z pierwszym oraz ostatnim elementem jako pivot. Dla obu tych przypadków złożoność osiągała wartość złożoności pesymistycznej a mianowicie $O(n^2)$. W przypadku gdy rozmiar tablicy wejściowej wynosi milion elementów, średni czas potrzebny na posortowanie elementów przekraczał minutę. Ze względu na swoje dramatyczne rezultaty, wyniki obu algorytmów nie zostały uwzględnione na wykresie ze względu na diametralne różnice w otrzymanych wartościach. Najbardziej efektywne okazały się algorytmy z rodziny sortowania przez wstawianie, dla których rosnący porządek danych wejściowych daje najlepsze wyniki i złożoności rzędu liniowych. Niestety wykres nie

odzwierciedla średnich charakterystyk poszczególnych algorytmów sortowania, ale służy wizualizacji różnic pomiędzy algorytmami.

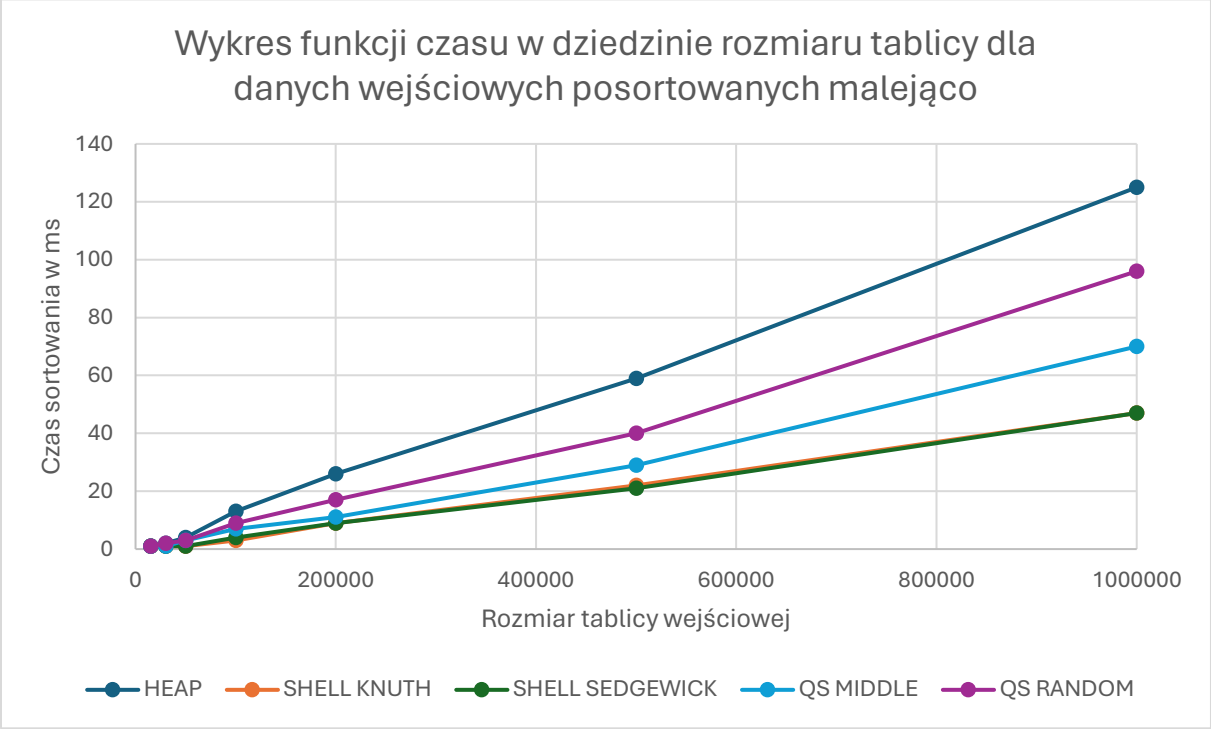


Powyższy wykres przedstawia wyniki algorytmów sortowania szybkiego w wariantach ze skrajnymi pivotami. Jak widzimy oczekiwana złożoność kwadratowa jest zachowana tylko między niektórymi punktami. Dwukrotne zwiększenie rozmiaru tablicy powinno w rezultacie dać czterokrotnie dłuższy czas sortowania, jednak w praktyce nie otrzymano takich wyników. Można to szczególnie zauważyć dla prawego skrajnego pivota dla wyników sortowań 500000 i 1000000 elementów.

Dane wejściowe posortowane malejąco

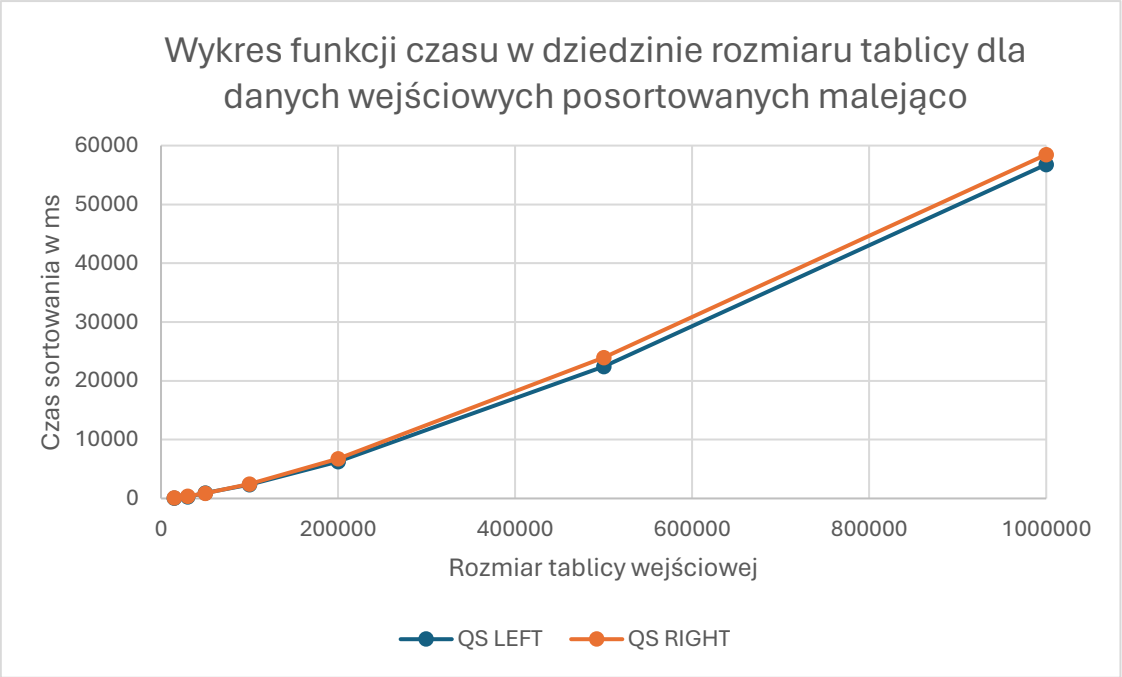
Tabela 2 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego algorytmu sortowania. Dane wejściowe posortowane malejąco

	Insertion	Heap	Shell Knuth Sequence	Shell Sedgewick Sequence	Quicksort Left Pivot	Quicksort Right Pivot	Quicksort Middle Pivot	Quicksort Random Pivot
15000	245	1	1	1	71	96	1	1
30000	1082	2	1	1	279	400	1	2
50000	3776	4	1	1	930	898	3	3
100000	14848	13	3	4	2361	2434	7	9
200000	62489	26	9	9	6291	6752	11	17
500000	216360	59	22	21	22421	23968	29	40
1000000	919996	125	47	47	56785	58465	70	96



Podobnie jak w poprzednim przypadku algorytm sortowania szybkiego w wersjach ze skrajnymi elementami jako pivotami osiąga złożoność kwadratową i fatalne czasy działania. Jednakże w tym przypadku do grupy nieefektywnych rozwiązań dołączył również podstawowy algorytm sortowania przez wstawianie, dla którego dane wejściowe posortowane malejąco są najgorszym możliwym przypadkiem. Czasy wyróżnione kolorem czerwonym zostały wyznaczone na podstawie serii 20 pomiarów ze względu na duży nakład czasowy jednego sortowania, co z kolei sprawia, że wynik jest obarczony dużym błędem. Odstępstwo od normy pomiarowej wydaje się być uzasadnione, ze względu na fakt, że czasy osiągają te kilka minut co w porównaniu z pozostałymi algorytmami czyni je bezużytecznymi w praktycznym zastosowaniu. Z racji dużych różnic w porównaniu do pozostałych algorytmów wartości wynikowe nie zostały uwzględnione na wykresie. Przechodząc do bardziej efektywnych rozwiązań można zauważyć, że najlepiej spisały się dwa warianty algorytmu Shella. Drugie miejsce zajęły algorytmy sortowania szybkiego ze środkowym oraz losowym elementem jako pivot. Na trzecim miejscu podium znajduje się

algorytm sortowania przez kopcowanie. Wynik wydaje się być zaskakujący ze względu na fakt, że algorytmy Shella mają złożoności $O(n^{3/2})$ oraz $O(n^{4/3})$ i osiągnęły lepsze rezultaty niż algorytmy sortowania szybkiego oraz przez kopcowanie, których średnia złożoność jest rzędu $O(n\ logn)$.

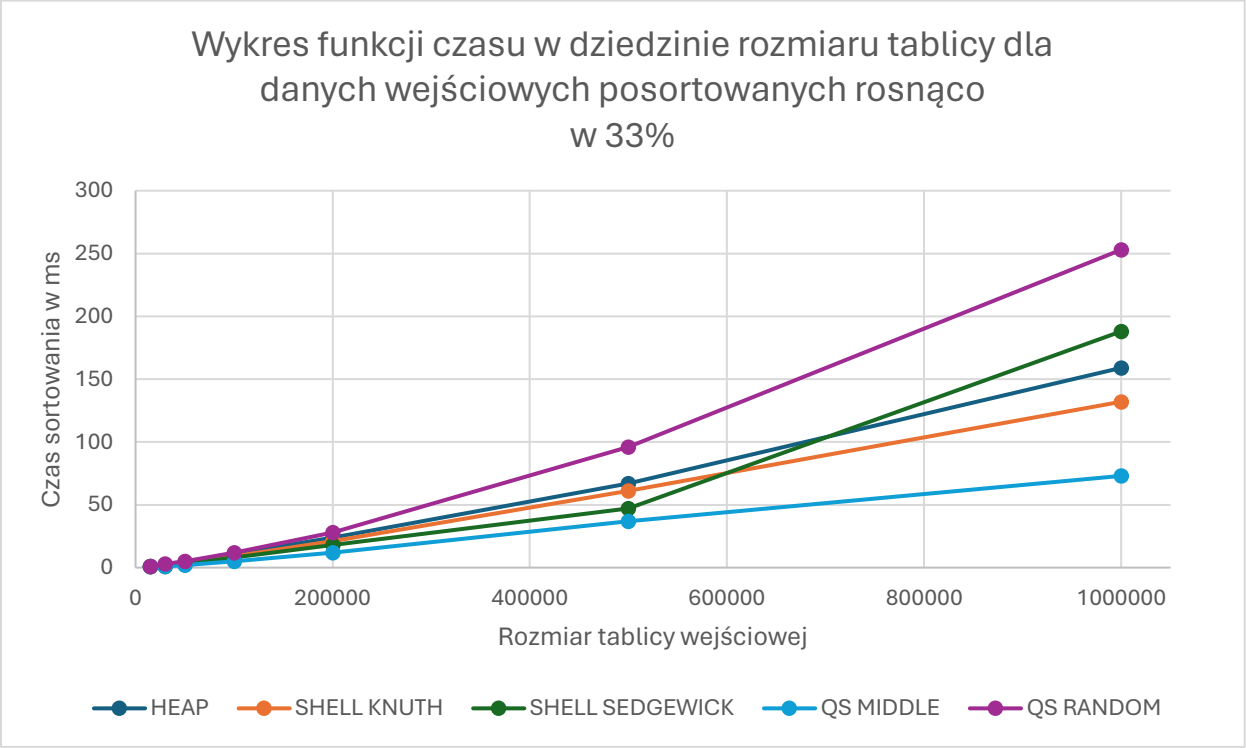


Powyższy wykres przedstawia wyniki algorytmów sortowania szybkiego w wariantach ze skrajnymi pivotami. Podobnie jak w poprzednim przypadku, kiedy elementy tablicy były posortowane rosnąco, można zauważyć, że oczekiwana złożoność kwadratowa jest zachowana tylko między niektórymi punktami. Jednak nie widzimy tak niespodziewanych i silnych zmian kierunku jak w poprzednim scenariuszu. Duże znaczenie może mieć tutaj błąd pomiarowy lub mała ilość próbek jeżeli mowa o rozmiarze danych do posortowania.

Dane wejściowe posortowane częściowo: 33% początkowych elementów posortowane rosnąco

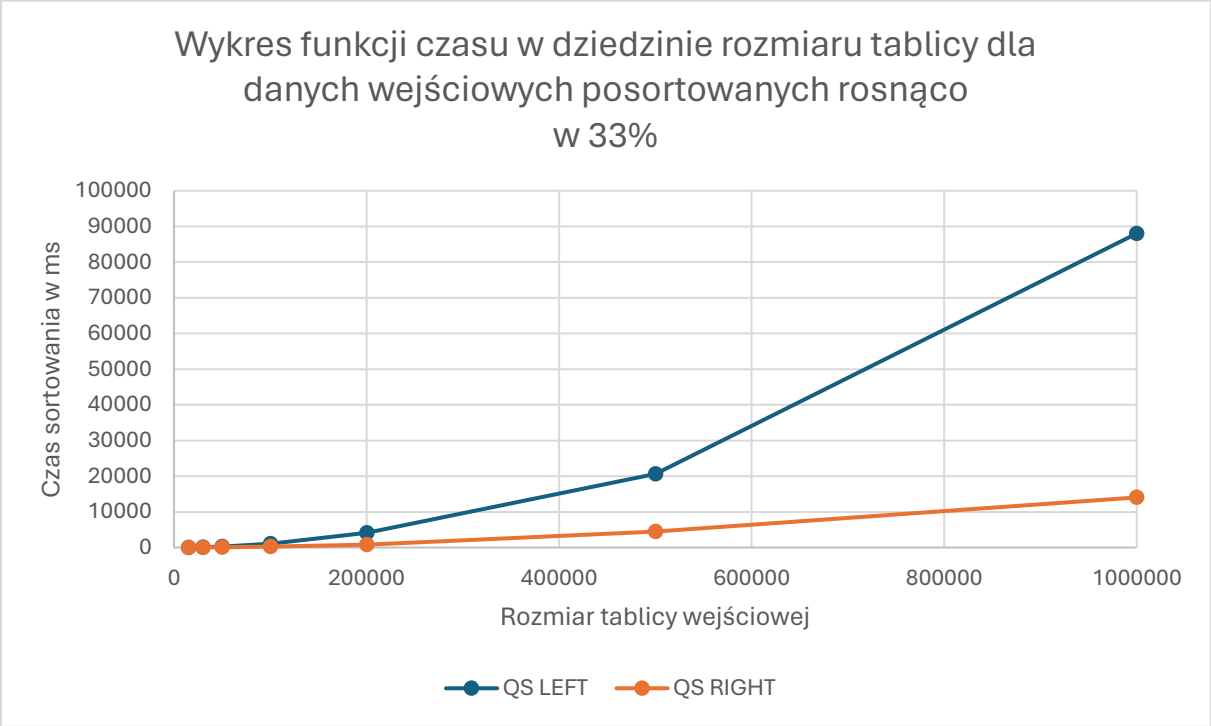
Tabela 3 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego algorytmu sortowania. Dane wejściowe posortowane częściowo: 33% początkowych elementów posortowane rosnąco.

	Insertion	Heap	Shell Knuth Sequence	Shell Sedgewick Sequence	Quicksort Left Pivot	Quicksort Right Pivot	Quicksort Middle Pivot	Quicksort Random Pivot
15000	41	1	1	1	33	5	1	1
30000	173	2	2	1	145	21	1	3
50000	472	4	3	3	344	63	2	5
100000	2280	11	9	8	1223	283	5	12
200000	9165	24	21	18	5032	788	12	28
500000	56119	67	61	47	20635	4519	37	96
1000000	212496	159	132	188	88040	14042	73	253



W przypadku częściowo posortowanych danych okazuje się, że zarówno algorytm sortowania przez wstawianie oraz algorytmy szybkiego sortowania ze skrajnymi elementami jako pivoty, nie radzą sobie najlepiej. Dla algorytmu sortowania przez wstawianie wyraźnie widać zarys złożoności kwadratowej: dwukrotne zwiększenie ilości danych spowoduje czterokrotne wydłużenie czasu sortowania. Przez swoje ogromne czasy sortowania, pomiary dla algorytmu sortowania przez wstawianie nie zostały uwzględnione na żadnym z wykresów. Ciekawą sytuację można zaobserwować dla quicksortów ze skrajnymi pivotami. Implementacja ze skrajnie prawym elementem wypada lepiej niż ta z początkowym elementem, natomiast charakterystyki obu zbiegają do kwadratowej. Na podstawie danych zamieszczonych w tabeli można zauważyć, że pozostałe algorytmy biją o rzędy wielkości czasy sortowania trzech pozostałych. Dla algorytmów sortowania przez kopcowanie oraz szybkiego z losowym pivotem charakterystyka najbardziej przypomina zbliżoną do $O(n\,logn)$. Po raz kolejny można zauważyć, że dwie implementacje

algorytmu Shella uzyskują lepsze wyniki niż algorytm sortowania szybkiego z losowym pivotem, co jest niezgodne z teoretyczną złożonością algorytmów.

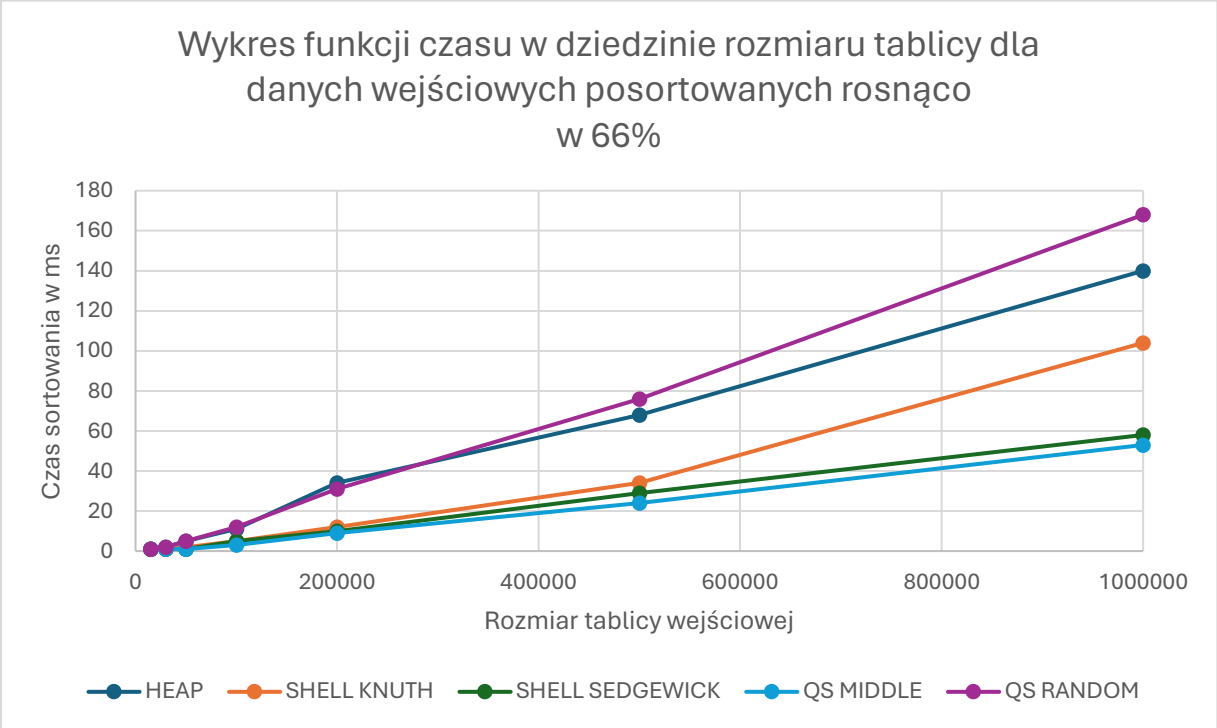


Powyższy wykres przedstawia czasy sortowań dla algorytmu sortowania szybkiego w wariantach ze skrajnymi pivotami. Jak można zauważyć, czasy sortowania dla skrajnie prawego elementu są lepsze niż w przypadku skrajnie lewego elementu. Ciężko w tym momencie stwierdzić czy wynika to z błędów pomiarowych czy istnieje logiczne uzasadnienie zaistniałej sytuacji.

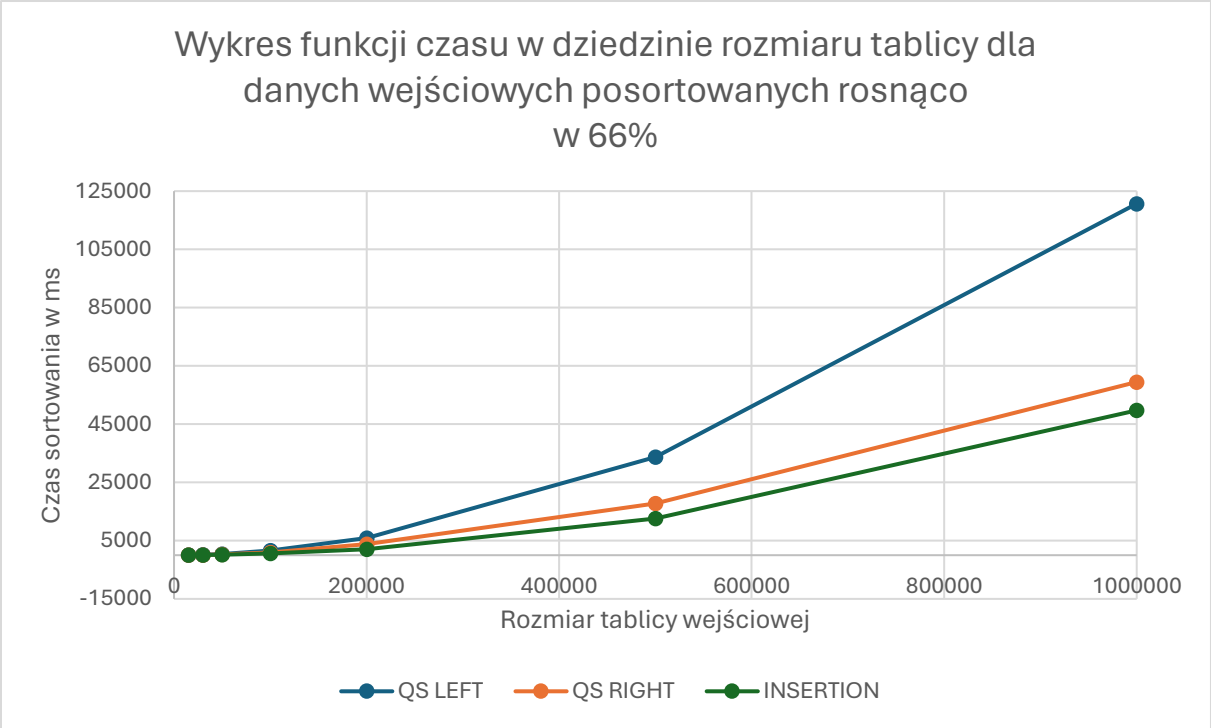
Dane wejściowe posortowane częściowo: 66% początkowych elementów posortowane rosnąco

Tabela 4 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego algorytmu sortowania. Dane wejściowe posortowane częściowo: 66% początkowych elementów posortowane rosnąco.

	Insertion	Heap	Shell Knuth Sequence	Shell Sedgewick Sequence	Quicksort Left Pivot	Quicksort Right Pivot	Quicksort Middle Pivot	Quicksort Random Pivot
15000	14	1	1	1	48	21	1	1
30000	66	2	1	1	158	94	1	2
50000	155	5	2	1	412	258	1	5
100000	566	11	5	5	1579	982	3	12
200000	2027	34	12	10	5923	3764	9	31
500000	12613	68	34	29	33692	17755	24	76
1000000	49726	140	104	58	120686	59462	53	168



W przypadku gdzie 66% początkowych danych w tablicy jest posortowanych najgorzej poradziły sobie algorytmy sortowania przez wstawianie oraz szybkie ze skrajnymi pivotami. Z powodu swoich dużych czasów pomiarowych zostały uwzględnione na wykresie poniżej. Najlepsze czasy uzyskały algorytmy sortowania szybkiego ze środkowym elementem jako pivot oraz sortowania Shella z wykorzystaniem sekwencji Sedgewicka. Następnymi algorytmami w kolejności najmniejszych czasów sortowania są Shella z wykorzystaniem sekwencji Knutha, sortowania przez kopcowanie oraz sortowania szybkiego z losowym pivotem. Co ciekawe w powyższym przypadku to wartiant Sedgewicka algorytmu Shella uzyskał czasy bardzo przybliżone algorytmowi sortowania szybkiego. Duży wpływ na uzyskany wynik miało 66% posortowanie danych wejściowych, które zredukowało liczbę porównań użytych w algorytmie Shella.

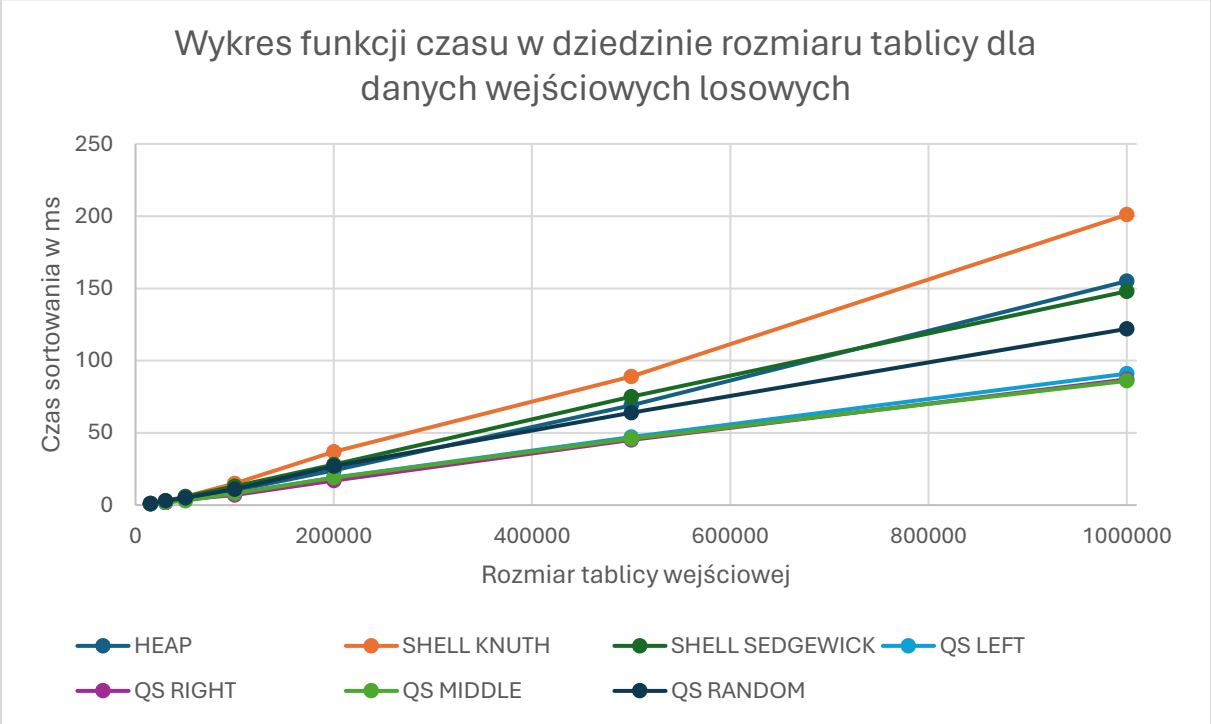


Jeżeli mowa o pozostałych trzech algorytmach to wyniki pomiarowe prezentują się następująco. Dzięki sporemu posortowaniu danych wejściowych, algorytm sortowania przez wstawianie był bardziej efektywny niż algorytmy sortowania szybkiego.

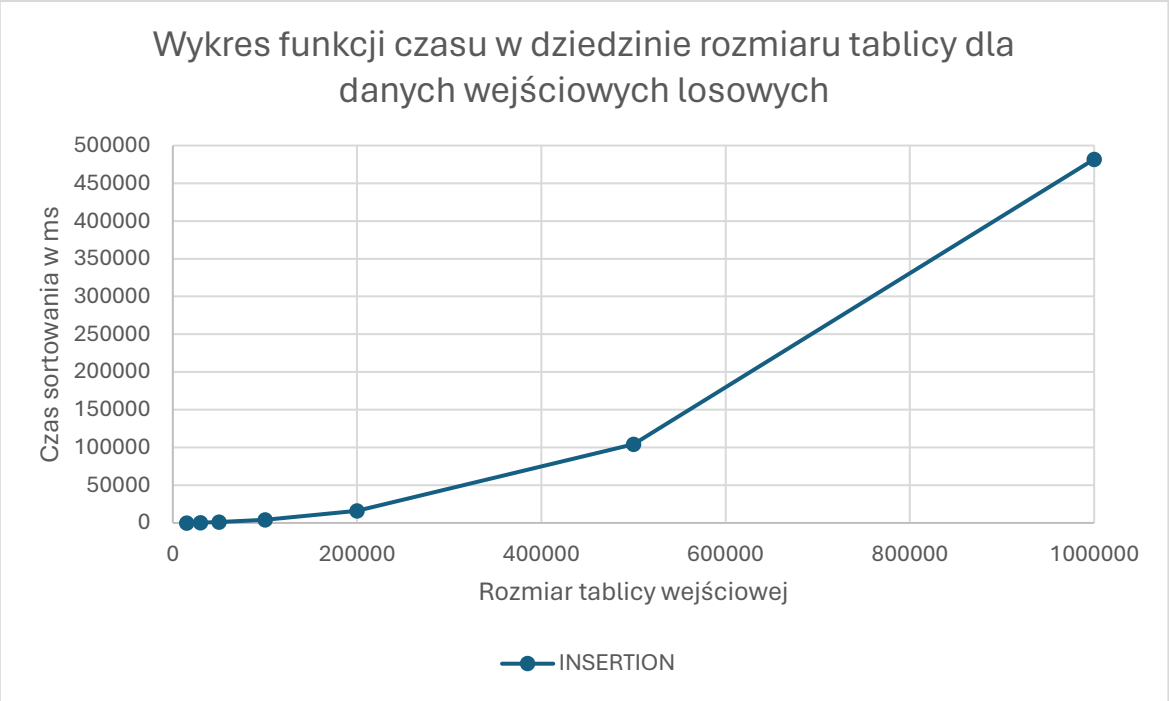
Dane wejściowe w losowej kolejności

Tabela 5 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego algorytmu sortowania. Dane wejściowe w losowej kolejności

	Insertion	Heap	Shell Knuth Sequence	Shell Sedgewick Sequence	Quicksort Left Pivot	Quicksort Right Pivot	Quicksort Middle Pivot	Quicksort Random Pivot
15000	92	1	1	1	1	1	1	1
30000	165	2	3	2	2	2	2	3
50000	1062	5	6	6	5	4	3	5
100000	4071	10	15	13	8	7	8	11
200000	16069	24	37	28	19	17	19	27
500000	104335	69	89	75	47	45	46	64
1000000	481889	155	201	148	91	87	86	122



Wyniki pomiarów dla najważniejszego przypadku, który obejmuje większość scenariuszy życia codziennego, czyli danych wejściowych wygenerowanych losowo prezentuje się w powyższy sposób. W tym przypadku jedyną czarną owcą wśród algorytmów okazało się sortowanie przez wstawianie, które wyraźnie odbiega w górę z czasami sortowania w porównaniu reszty grupy. Największą efektywnością wykazała się grupa algorytmów z rodziny sortowania szybkiego, jednakże zarówno algorytm sortowania przez kopcowanie jak i Shella nieznacznie od niej odbiegały. Jak widzimy na podstawie tabeli oraz wykresu, algorytmy sortowania szybkiego prezentują bardziej tendencję do złożoności liniowej aniżeli teoretycznej $O(n\ logn)$. Może to być spowodowane przez błędy pomiarowe.



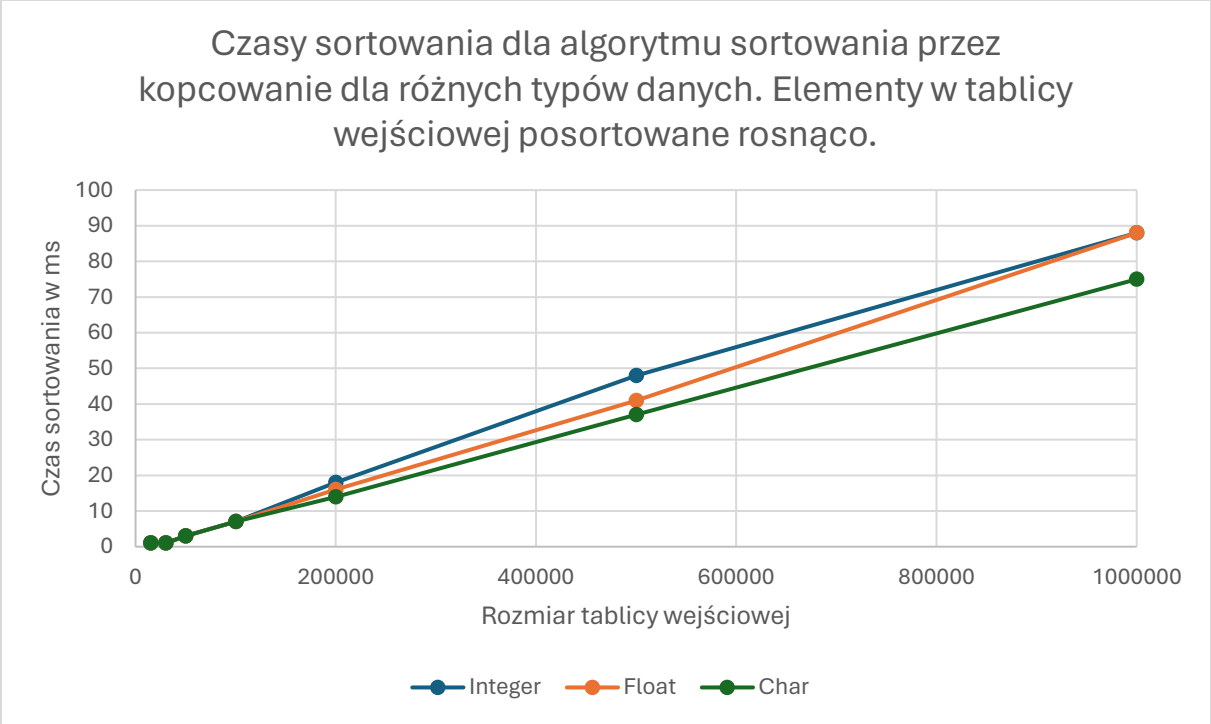
Powyższy wykres reprezentuje zmierzone czasy dla algorytmu sortowania przez wstawianie, który jak już zostało wcześniej wspomniane, wypadł najgorzej ze wszystkich algorytmów. Podobnie jak w poprzednich przypadkach, czerwonym kolorem został oznaczony pomiar wykonany 20 razy ze względu na swoją dużą złożoność czasową.

Wpływ typu danych na czas sortowania na przykładzie algorytmu sortowania przez kopcowanie

Kolejną kwestią poddaną pomiarom jest wpływ typu danych tablicy na czas sortowania. Sytuację rozpatrzono dla trzech typów danych: liczba całkowita 32 bitowa typu *integer*, liczba zmiennoprzecinkowa 32 bitowa typu *float* oraz zmienna 8 bitowa typu *char*. Ze względu na dobre średnie wyniki czasowe pomiary zostały wykonane dla algorytmu sortowania przez kopcowanie. Ponownie jak w przypadku poprzednich pomiarów, badania wykonano dla 5 scenariuszy w zależności od początkowego posortowania danych. Wyniki i rezultaty zostały przedstawione i opisane poniżej.

Tabela 6 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego typu danych dla algorytmu sortowania przez kopcowanie. Dane wejściowe posortowane rosnąco.

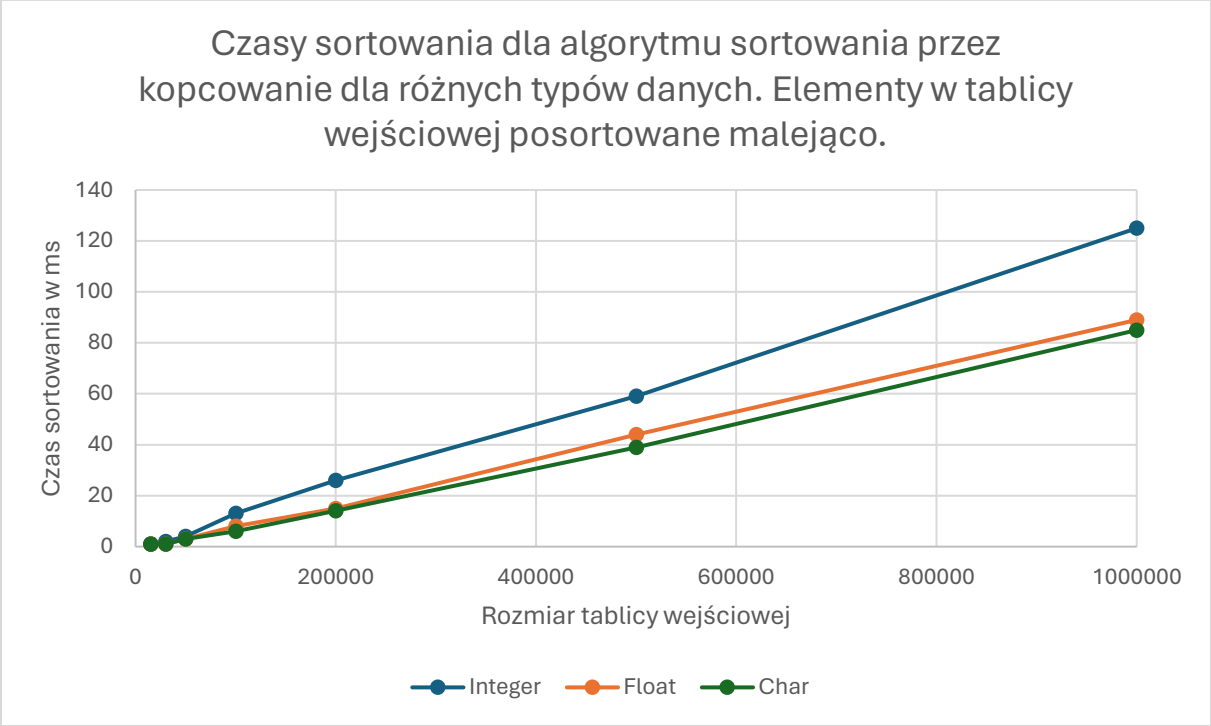
	Integer	Float	Char
15000	1	1	1
30000	1	1	1
50000	3	3	3
100000	7	7	7
200000	18	16	14
500000	48	41	37
1000000	88	88	75



W powyższym przypadku różnice uzyskane dla różnych typów są bardzo niewielkie. Na podstawie powyższych danych można stwierdzić, że w tym przypadku rodzaj typu danych nie miał dużego znaczenia na ostateczny czas sortowania.

Tabela 7 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego typu danych dla algorytmu sortowania przez kopcowanie. Dane wejściowe posortowane malejąco.

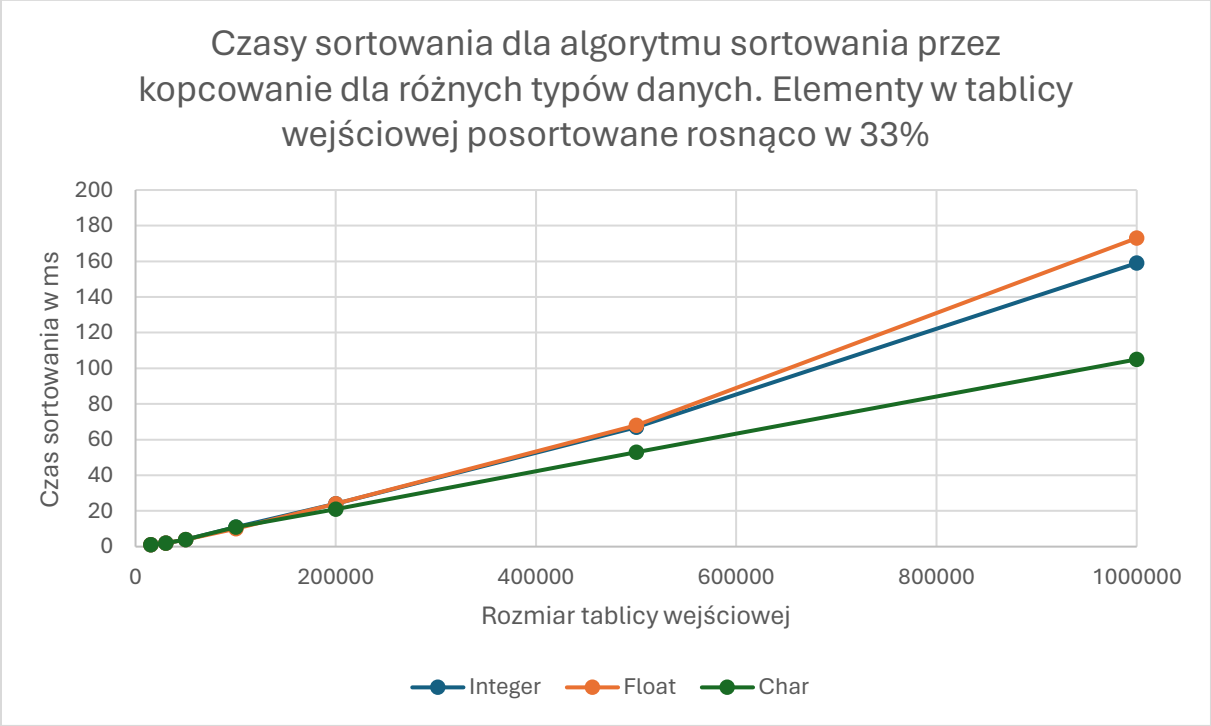
	Integer	Float	Char
15000	1	1	1
30000	2	1	1
50000	4	3	3
100000	13	8	6
200000	26	15	14
500000	59	44	39
1000000	125	89	85



W przykładzie gdy dane są posortowane malejąco można zauważyć rozbieżność dla typu danych reprezentującego liczbę całkowitą. Niestety ciężko stwierdzić dlaczego algorytm zachował się akurat w taki sposób czy jest to tylko kwestia chwilowej niedyspozycji maszyny lub błędów pomiarowych, bowiem różnica między czasami wynosi tylko kilkadziesiąt milisekund.

Tabela 8 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego typu danych dla algorytmu sortowania przez kopcowanie. Dane wejściowe posortowane rosnąco w 33%.

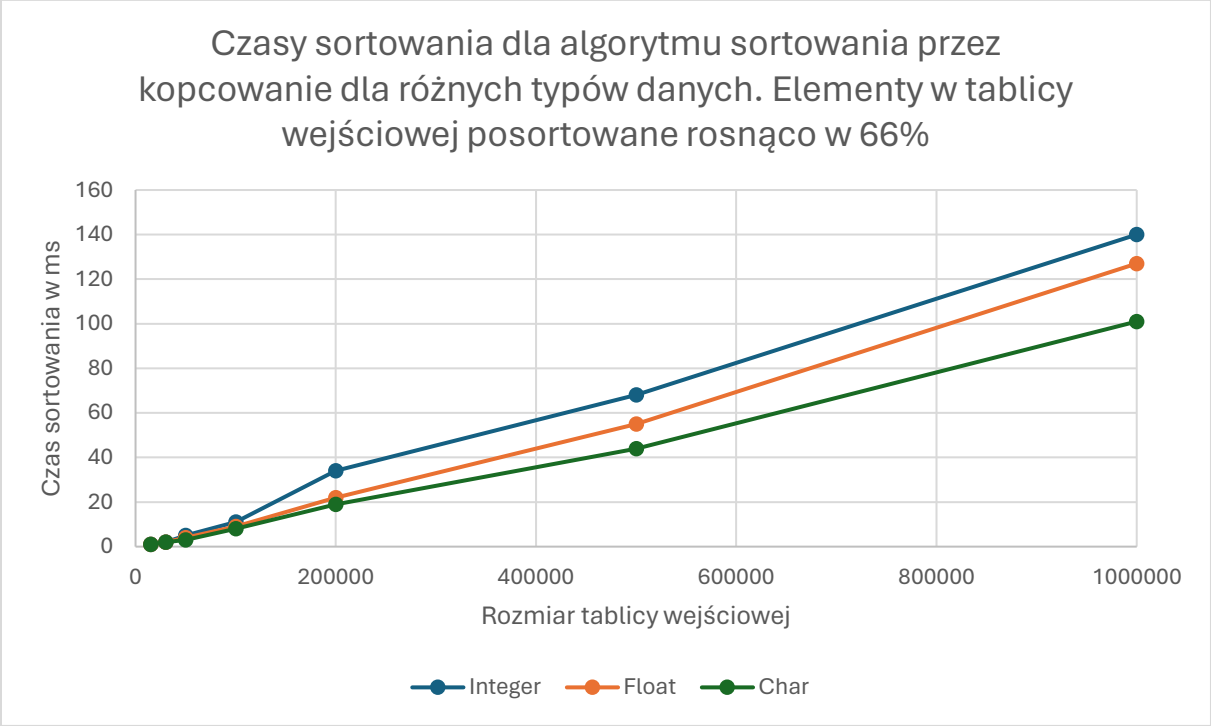
	Integer	Float	Char
15000	1	1	1
30000	2	2	2
50000	4	4	4
100000	11	10	11
200000	24	24	21
500000	67	68	53
1000000	159	173	105



W przypadku gdy dane wejściowe zostały posortowane rosnąco w 33% można zauważyć, że dla typu danych *char* czas sortowań dla rozmiarów tablicy powyżej 500000 był wyraźnie krótszy niż w przypadku *integera* czy *floata*. Podobnie jak w przypadku poprzednich przypadków istnieje duże prawdopodobieństwo, że różnice te wynikają z błędów pomiarowych.

Tabela 9 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego typu danych dla algorytmu sortowania przez kopcowanie. Dane wejściowe posortowane rosnąco w 66%.

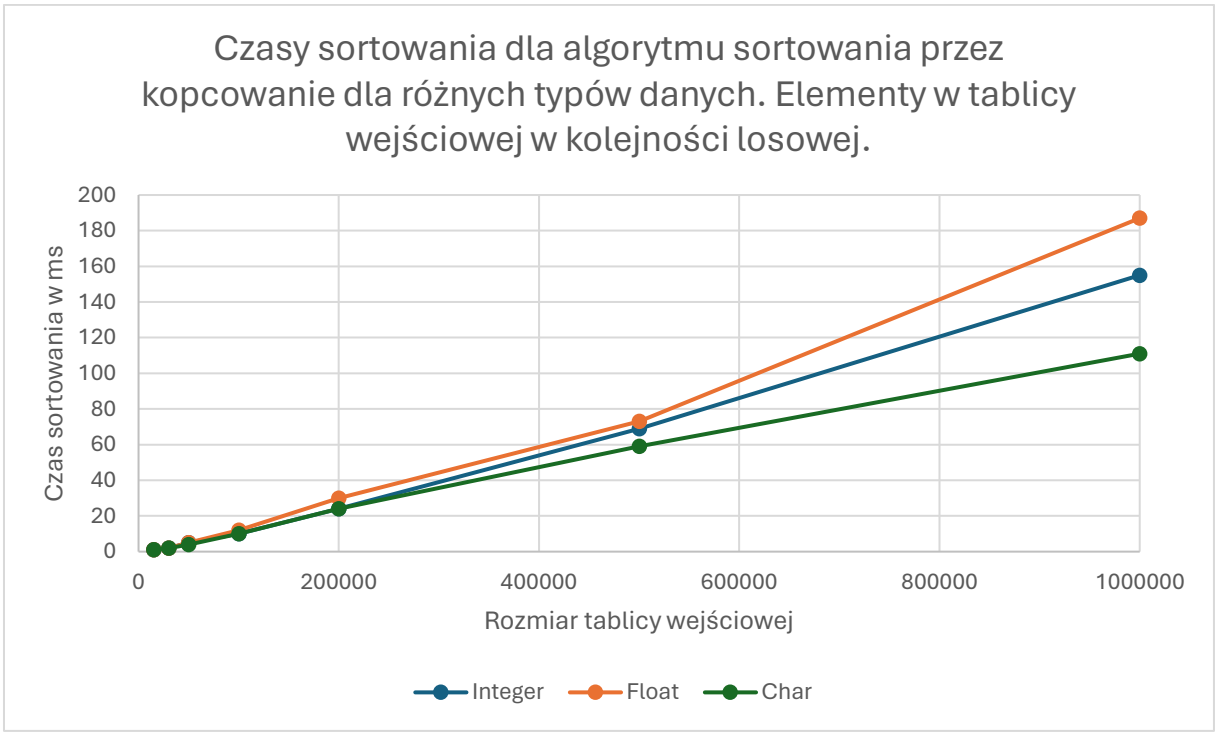
	Integer	Float	Char
15000	1	1	1
30000	2	2	2
50000	5	4	3
100000	11	9	8
200000	34	22	19
500000	68	55	44
1000000	140	127	101



W przypadku gdy dane wejściowe zostały posortowane rosnąco w 66% widzimy, że wystąpiła podobna sytuacja jak w poprzednim przypadku. Czasy sortowań były minimalnie krótsze dla typu danych reprezentującego znak. Jednak wszystkie wartości oscylują wokół pewnego zakresu. Z tego powodu można twierdzić, że różnice występują z powodu błędów pomiarowych.

Tabela 10 Wyniki pomiaru czasu w milisekundach w zależności od rozmiaru tablicy wejściowej i wybranego typu danych dla algorytmu sortowania przez kopcowanie. Dane wejściowe posortowane w kolejności losowej.

	Integer	Float	Char
15000	1	1	1
30000	2	2	2
50000	5	5	4
100000	10	12	10
200000	24	30	24
500000	69	73	59
1000000	155	187	111



W ostatnim przypadku, gdy dane na wejściu nie były posortowane w żaden sposób, widzimy podobną tendencję jak w przypadku dwóch poprzednich scenariuszy. Najlepsze czasy uzyskano dla 8 bitowego typu danych, jednakże różnice między pozostałymi rodzajami nie są ogromne.

Podsumowanie i wnioski

Badania złożoności algorytmów okazały się być bardzo czasochłonnym zadaniem. Oczekiwanie na rezultaty oraz opracowywanie wyników zajęło sporo czasu obliczeniowego, szczególnie dla przypadków, w których algorytmu osiągały pesymistyczne złożoności rzędu kwadratowych. Przyjrzyjmy się ostatecznym wynikom osiągniętym przez poszczególne algorytmy.

Algorytm sortowania przez wstawianie zgodnie z oczekiwaniami jest algorytmem, które osiągał najgorsze rezultaty. Pesymistyczna oraz średnia złożoność kwadratowa czynią ten algorytm niepraktycznym w zastosowaniach, które sortują tablice zawierające wiele elementów. Odstępstwem od wyżej przedstawionego wniosku są dwa scenariusze: dane w tablicy wejściowej posortowane rosnąco oraz przypadek gdy tablica zawiera małe ilości danych np. 20. Te dwie cechy sprawiają, że algorytm sortowania przez wstawianie znajduje zastosowanie w modyfikacjach pozostałych algorytmów sortowania. Mowa wtedy o zastosowaniach hybrydowych, które w zależności od danych wejściowych mogą wybrać czy zostanie użyty np. algorytm sortowania szybkiego, czy liczba danych pozwoli na osiągnięcie lepszych rezultatów przy zastosowaniu właśnie sortowania przez wstawianie.

Kolejnym algorytmem jest sortowanie przez kopcowanie. Uzyskane rezultaty są zgodne z teoretycznymi złożonościami algorytmu. Czasy sortowań niezależne są od początkowego ułożenia danych, co czyni sortowanie przez kopcowanie efektywnym algorytmem.

Algorytmy Shella w wariantach z sekwencjami Knutha oraz Sedgewicka wykazały się również dobrą złożonością obliczeniową. Co dziwne w większości przypadków osiągały one lepsze rezultaty od sortowania przez kopcowanie pomimo tego, że mają gorsze teoretyczne złożoności. Podobnie jak w przypadku poprzednio opisanego algorytmu początkowe ułożenie nie miało negatywnego wpływu na czasy sortowań. W przypadku gdy dane były posortowane rosnąco oba algorytmy osiągały wyśmienite wyniki podobnie jak ich bazowy algorytm sortowania przez wstawianie. Co ciekawe algorytmy Shella osiągnęły najlepsze rezultatów spośród wszystkich algorytmów dla przypadku gdy dane były posortowane malejąco.

Grupa algorytmów sortowania szybkiego osiągnęła wyniki zgodne z oczekiwaniami. W przypadku danych posortowanych rosnąco, zarówno tych w pełni jak i częściowo, warianty sortowania szybkiego ze skrajnymi pivotami osiągnęły fatalne rezultaty. Z tego względu używanie sortowania szybkiego ze środkowym pivotem wydaje się być najlepszym rozwiązaniem, które znacznie ogranicza prawdopodobieństwo osiągnięcia pesymistycznej złożoności kwadratowej. Podobnie jest w przypadku wyboru losowego elementu jako pivot, lecz we wszystkich przypadkach osiągnął on nieznacznie gorsze rezultaty niż wariant ze środkowym elementem.

W przypadku pomiarów czasu algorytmu sortowania przez kopcowanie w zależności od typu danych otrzymujemy niejednoznaczne odpowiedzi. W większości przypadków czasy sortowań są bardzo zbliżone od siebie i różnica pomiędzy poszczególnymi grupami wynosi maksymalnie kilkadziesiąt milisekund. W związku z tym można sądzić, że różnice wynikają jedynie z błędów pomiarowych. Jednak z powodu tego, że tendencja ta powtarza się w wielu scenariuszach, można również wyciągnąć wniosek, że typ danych ma niewielki wpływ na czas sortowania elementów.

Źródła

https://en.wikipedia.org/wiki/Sorting_algorithm

https://en.wikipedia.org/wiki/Insertion_sort

https://hideoushumpbackfreak.com/algorithms/sorting-insertion.html

https://en.wikipedia.org/wiki/Heapsort

https://en.wikipedia.org/wiki/Shellsort

https://en.wikipedia.org/wiki/Quicksort

Wprowadzenie do algorytmów - Cormen