Algorytmy i złożoność obliczeniowa

Nr projektu: **1**

Temat projektu: **Analiza wydajności wybranych algorytmów sortowania w zależności od typu danych i stopnia wstępnego uporządkowania zestawów wejściowych**

Wydział Informatyki i Telekomunikacji i złożoność obliczeniowa – project - [W04ITE-SI0064G]	
Grupa zajęciowa: 3, wtorek, 7:30	Data oddania sprawozdania: 27.04.2025
Imie, nazwisko i numer indeksu wykonującego projekt: Andrii Kytrysh, 275770	Ocena:

Spis treści

1 Cel projektu

Celem danego projektu jest zapoznanie się z algorytmami sortowania przez implementacjie i zbadanie działania algorytmów sortowania w przybliżonych do rzeczywistości warunkach. W tym zbadać i przeanalizować wpływ rozkładu oraz typu danych wejściowych (int, float, double i td.) na czas sortowania. Celem projektu zarówno jest napisanie implementacji algorytmów oraz dodatkowego oprogramowania do obsługi badań czasu wykonania algorytmów, zapisywania wyników sortowania, odczytu danych wejściowych z pliku, generowania danych wejściowych o różnym rozkładzie, różnej długości, zapisu wyniku posortowanych danych do pliku.

2 Wstęp

W tym projekcie będziemy badali różne algorytmy na rosnącym ciągu ilości danych wejściowych, co jest przybliżeniem do rzeczywistych przypadków wykorzystania algorytmów sortowania w praktyce. Wykonamy analizę otrzymanych danych i porównamy je z teorią. Wykonamy wizualizacje otrzymanych przez badania wyników. Każde badanie zawiera w sobie 100 powtórzeń dla każdego przypadku i typu algorytmu. Wynikiem serii powtórzeń jest średni, minimalny oraz maksymalny czas pracy algorytmu oraz mediana czasu. Wszystkie te informacji będą wykorzystane do analizy wyników oraz zbudowania wykresów. Na samym początku omówimy algorytmy które będziemy implementować.

2.1 Sortowanie przez wstawianie

Sortowanie przez wstawianie to prosty algorytm, który buduje posortowaną część tablicy element po elemencie. W każdej iteracji bierze kolejny element z nieposortowanej części i wstawia go w odpowiednie miejsce w części już uporządkowanej, przesuwając większe elementy o jedno miejsce w prawo. Jego złożoność czasowa to $O(n^2)$ w pesymistycznym i średnim przypadku, natomiast w najlepszym — gdy dane są już wstępnie posortowane — działa w czasie liniowym $O(n)$. Ze względu na prostotę jest wydajny dla niewielkich zbiorów lub niemal posortowanych danych. Dla dużych tablic staje się niepraktyczny.

2.2 Sortowanie binarne przez wstawianie

Sortowanie binarne przez wstawianie to wariant klasycznego sortowania przez wstawianie, w którym do znalezienia miejsca wstawienia wykorzystuje się wyszukiwanie binarne zamiast liniowego skanowania. Dzięki temu liczba porównań maleje do $O(\log n)$ na wstawienie, co daje łącznie $O(n \log n)$ porównań, jednak liczba przesunięć elementów pozostaje wciąż $O(n^2)$. Algorytm zachowuje stabilność i prostotę implementacji podobnie jak zwykłe wstawianie, ale może być nieco szybszy.

2.3 Sortowanie szybkie

Sortowanie szybkie (Quicksort) to algorytm typu „dziel i zwyciężaj”. Wybiera element zwany pivotem, dzieli tablicę na dwie części: mniejsze i większe od pivota, a następnie rekurencyjnie sortuje obie części. Średnia złożoność czasowa to $O(n \log n)$, natomiast w najgorszym przypadku — np. przy ciągłym wyborze skrajnego elementu jako pivot — stanowi $O(n^2)$. W praktyce, przy losowym

doborze pivota (w projekcie zaimplementowano losowy wybór pivota) lub bardziej zaawansowanych strategiach wyboru, Quicksort jest bardzo szybki i efektywny pamięciowo. Algorytm posiada sortowanie w miejscu. Nie jest stabilny.

2.4 Sortowanie przez kopcowanie

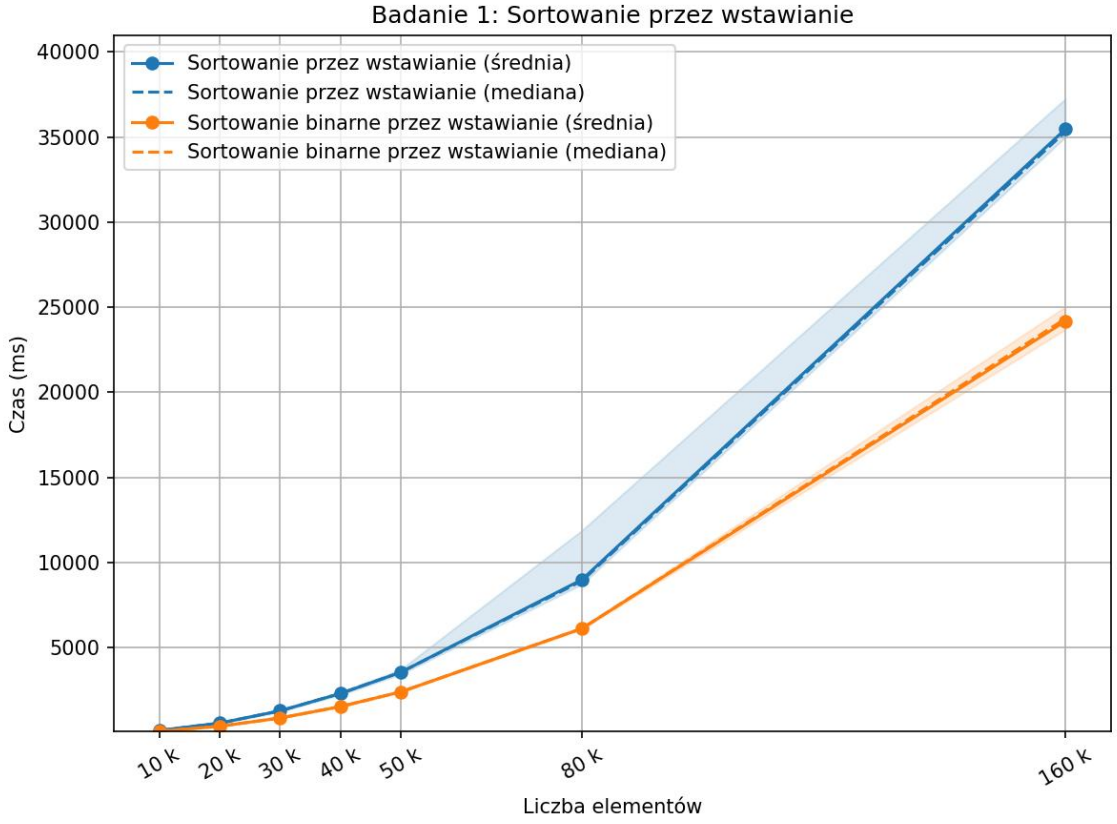
Sortowanie przez kopcowanie (Heapsort) również wykorzystuje ideę „dziel i zwyciężaj". Najpierw buduje z tablicy strukturę kopca (pełne drzewo binarne) maksymalnego w czasie $O(n)$, a następnie wielokrotnie usuwa korzeń (największy element) i odbudowuje kopiec, co daje kolejne $O(n \log n)$. Całkowita złożoność to zawsze $O(n \log n)$, niezależnie od początkowego rozkładu danych. Algorytm jest nie stabilny. Sortowanie jest w miejscu. Algorytm jest bardzo praktyczny w sytuacjach w których wymagane jest gwarantowane zachowanie tej samej złożoności działania w najgorszym przypadku.

3 Wpływ rozmiaru zbioru na wydajność algorytmów sortowania

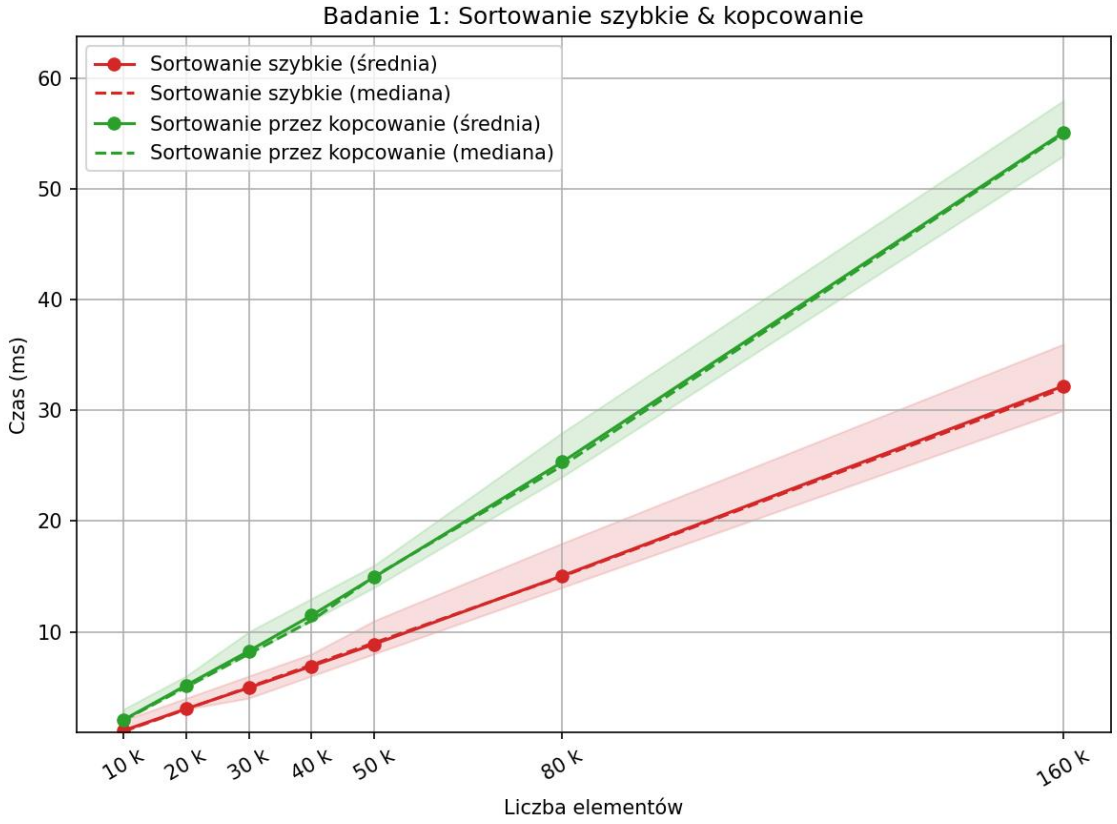
W tym rozdziale przedstawimy wyniki badania 1, w którym zmienialiśmy rozmiar zbioru danych wejściowych i mierzyliśmy czas działania czterech algorytmów sortowania: prostego przez wstawianie, binarnego przez wstawianie, przez kopcowanie oraz szybkiego. Pomiarom towarzyszyły zarówno obliczenia pojedynczych przebiegów (dla każdej wielkości instancji wykonywaliśmy 100 powtórzeń i zapisywaliśmy każdy wynik), jak i statystyki zbiorcze (średnia, mediana oraz wartości minimalne i maksymalne).

Algorytmy sortowania prez wstawianie oraz przez kopcowanie i sortowanie szybkie zostało wyświetlono na dwóch różnych wykresach z powodu wiekszej czytelności wykresu. Sprawa w tym, że ostatnie wymienione dwa typy sortowania mają znacznie lepszą złożoność obliczeniową przez co na wykresie razem z sortowaniem przez wstawianie będziemy obserwować linie ciągłą.

Na poniższym wykresie przedstawiono zależnośc czasu od ilości elementów dla sortowanie przez wstawianie zwykłe i binarne. Jest widać również granicy minimalnych oraz maksymalnych wartości czasu wykonywania sortowania. Dodatkowo na wykresie widać linie median dla obu algorytmów.



Podobny wykres został zrobiony i dla danych o czasie sortowania algorytmami szybkiego sortowania i przez kopcowanie

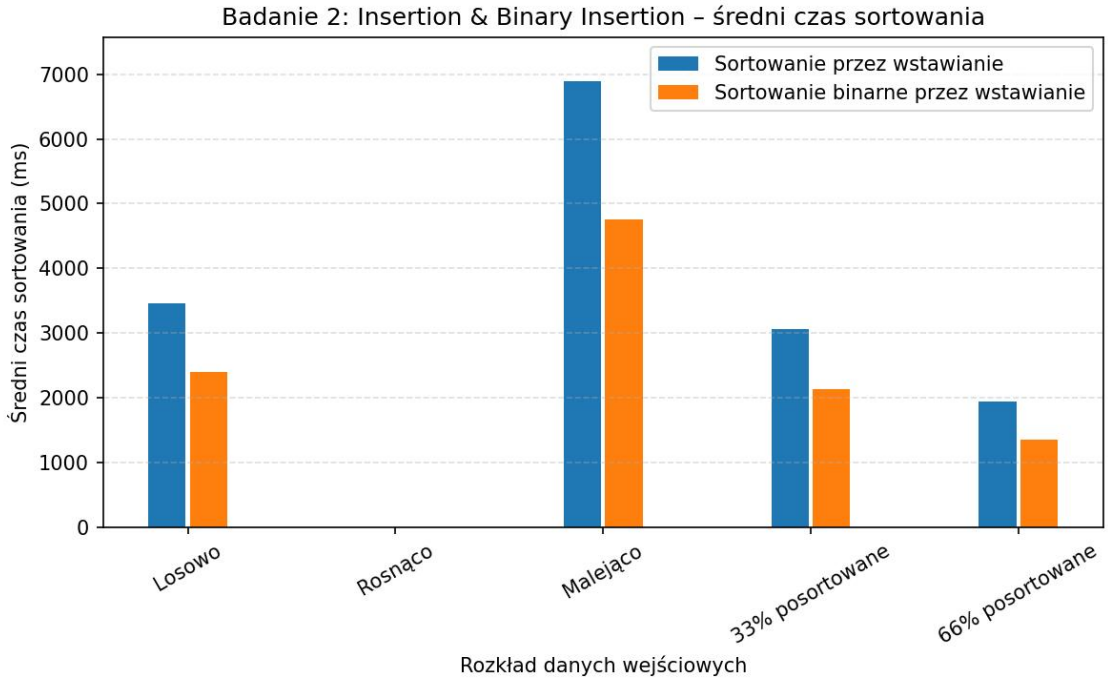


Na wykresach widać bardzo wyraźnie kwadratowy wzrost czasu działania obu wariantów sortowania przez wstawianie. Dla $N = 10\,\text{k}$ czas jest w granicach kilku setek milisekund, ale już przy $N = 80\,\text{k}$ osiągają rzędy $10^4\,\text{ms}$, a przy $N = 160\,\text{k}$ – więcej niż $3.5 \times 10^4\,\text{ms}$. Wariant binarny (pomarańczowy) systematycznie jest nieco szybszy od prostego (niebieskiego), ponieważ przy pomocy wyszukiwania binarnego skraca się czas wyszukiwania miejsca wstawienia elementu do $O(logN^2)$, choć sama operacja przesuwania ciągu w tablicy nadal wymaga $O(N^2)$ przesunięć.

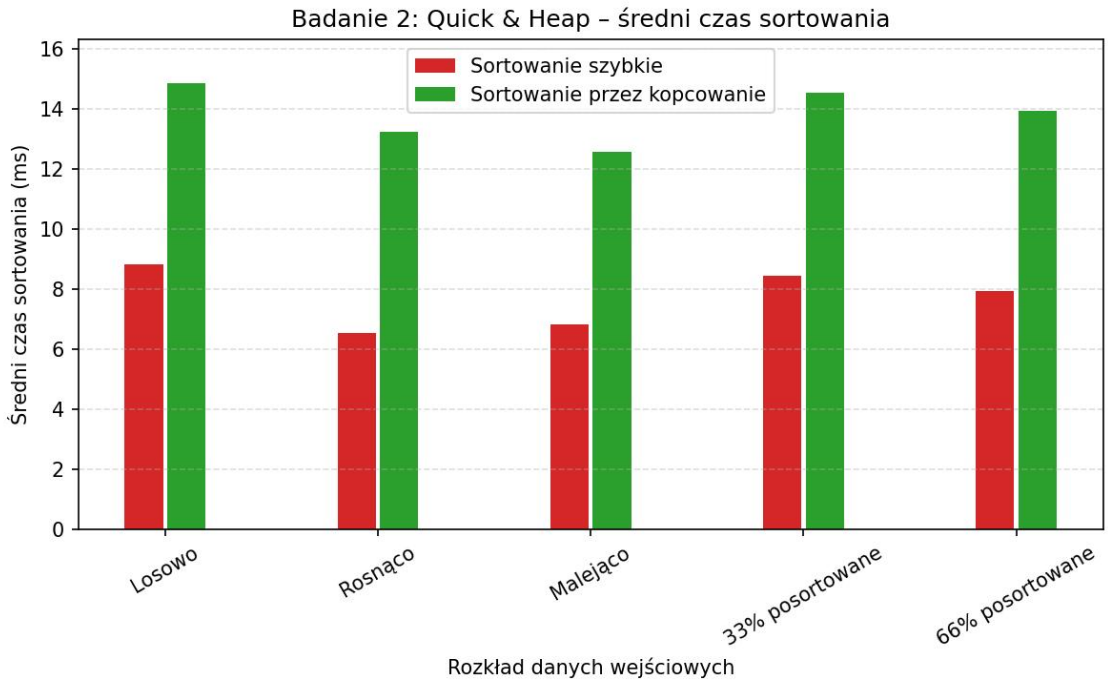
Sortowanie szybkie (czerwone punkty) oraz kopcowanie (zielone punkty) rosną znacznie wolniej – niemal liniowo na wykresach liniowo–logarytmicznych. Dla $N = 160\,\text{k}$ średni czas Quick Sort to około 32 ms, a Heap Sort – około 55 ms. Różnica między nimi wynika głównie z faktu, że Quick Sort w praktyce jest szybki przy losowym pivocie, ale może ulegać degeneracji przy niekorzystnym rozkładzie danych. Kopcowanie w praktyce daje stabilniejszy, choć trochę wolniejszy czas działania.

4 Wpływ wstępnego rozkładu danych na czas sortowania

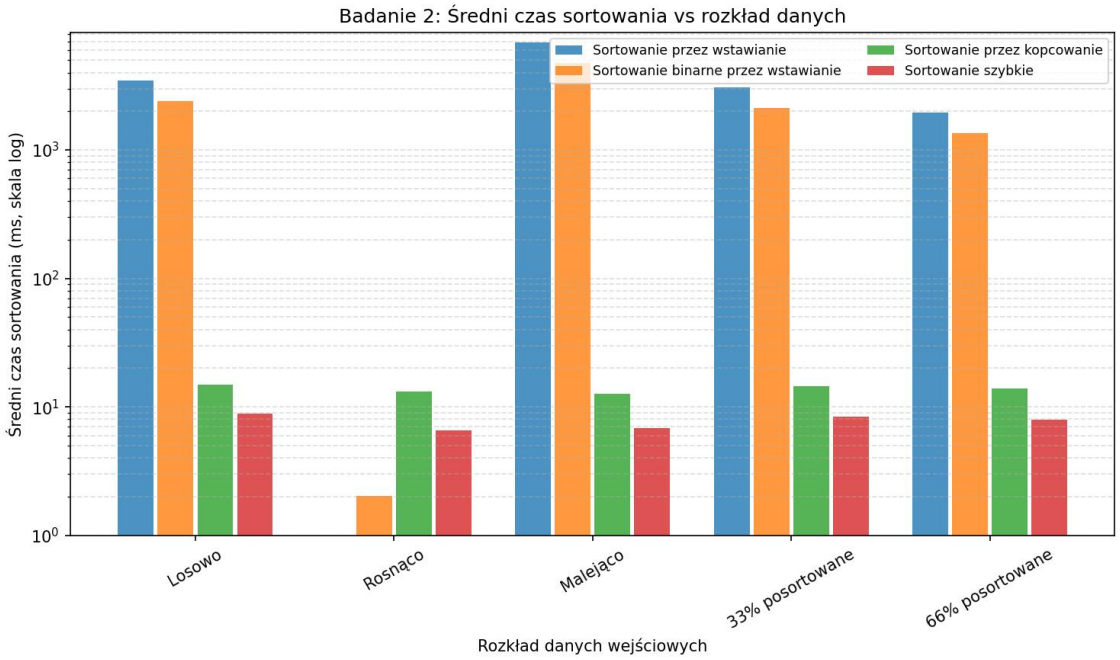
W badaniu drugim oceniono wpływ rodzaju początkowego uporządkowania zbioru na średni czas sortowania czterech algorytmów, wykonując dla każdego rozkładu (losowy, rosnący, malejący oraz częściowo wstępnie posortowany w 33% i 66%) 100 pomiarów i obliczając wartości średnie. Dane wejściowe zawierały 50000 elementów



Na rysunku widać, że sortowanie przez wstawianie osiąga najwyższy czas dla danych posortowanych malejąco (ok. 6900 ms), w tym przypadku algorytm musi dosłownie obrócić liste elementów i przy kazdym sprawdzeniu następujący element będzie mniejszy od poprzedniego. Znacznie niższy czas dla częściowo posortowanych (ok. 3050 ms przy 33%) i minimalny (rzędu pojedynczych milisekund) dla już posortowanych rosnąco. W tym przypadku algorytm wykonuje n przejść po zbiorze danych wejściowych i tak jak zbiór już jest posortowany, nic nie przestawia. Natomiast wersja binarna systematycznie redukuje czasy o około 30–35% w związku z binarnym wyszukaniem miejsca wstawienia elementu co znaczy że przy każdym sprawdzeniu jest odcinana połowa zbioru.



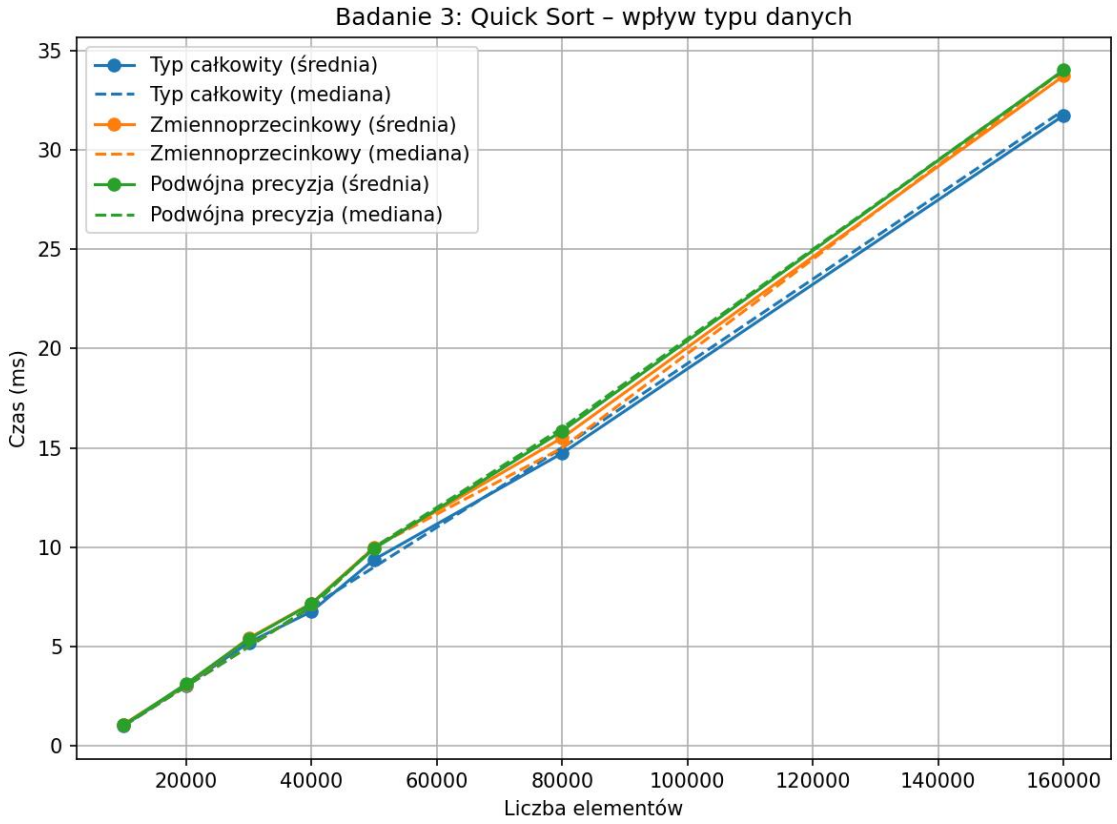
Na powyższym wykresie widać, że sortowanie szybkie jest najszybsze dla danych posortowanych rosnąco i malejąco, ale nieco wolniejszy na danych losowych oraz częściowo posortowanych. Sortowanie przez kopcowanie wykazuje względnie stałą wydajność we wszystkich przypadkach poza rozkładem danych posortowanych malejąco. W tym przypadku tablica już spełnia własność max-kopca (każdy rodzic jest większy od swoich dzieci), więc operacje przesiewania w fazie budowy kopca niemal nic nie robią. Dzięki temu budowa kopca, choć generalnie $O(n)$, dla takiego układu danych przebiega wyjątkowo szybko, z minimalną liczbą porównań i bez zamian.



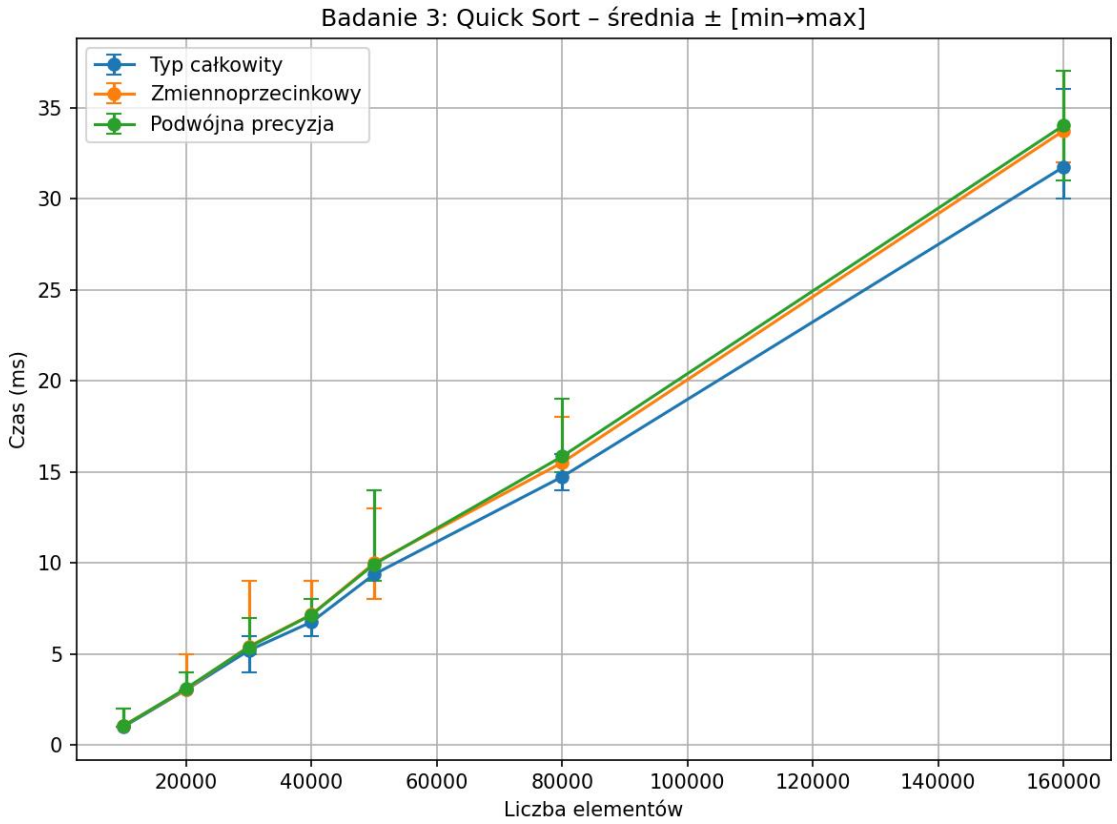
Na rysunku powyżej przedstawiono te same wyniki wraz z sortowaniem przez kopcowanie i szybkim w skali logarytmicznej, co podkreśla, że sortowanie szybkie utrzymuje średnie czasy rzędu 7–9 ms, natomiast sortowanie przez kopcowanie – 12–15 ms, niezależnie od rozkładu danych. Oba te algorytmy są sortowaniami w miejscu, co oznacza, że operują niemal bez dodatkowej pamięci pomocniczej, proporcjonalnej do wielkości wejścia. Sortowanie szybkie wykorzystuje jedynie stos wywołań rekurencyjnych o głębokości $O(\log n)$, a sortowanie przez kopcowanie modyfikuje tablicę bez żadnych struktur pomocniczych, co przekłada się na niskie wykorzystanie pamięci i dobrą efektywność pamięci podręcznej.

5 Wpływ typu danych na efektywność wybranego algorytmu

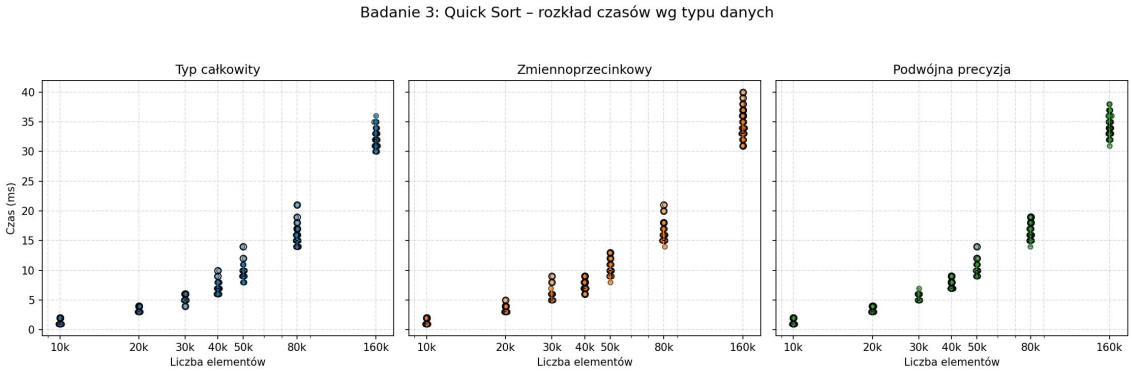
W badaniu trzecim oceniono, jak wybór typu przechowywania elementów (typ całkowity 32-bit, zmiennoprzecinkowy pojedynczej precyzji 32-bit oraz zmiennoprzecinkowy podwójnej precyzji 64-bit) wpływa na wydajność Quick Sorta, wykonując dla każdego typu i każdej wielkości zbioru (10 k, 20 k, 30 k, 40 k, 50 k, 80 k, 160 k) 100 pomiarów czasu i obliczając wartości średnie, mediany oraz zakres min–max.



Na rysunku powyżej widać niemal liniowy wzrost czasu sortowania wraz ze wzrostem N. Sortowanie typów danych zmiennoprzecinkowych i podwójnej precyzji jest widocznie wolniejsze od sortowania typu całkowitego, co wynika z dodatkowego narzutu operacji na liczbach zmiennoprzecinkowych. Wynika to z faktu, że porównania zmiennoprzecinkowe zgodnie ze standardem muszą rozpakować i porównać zarówno mantysę, jak i wykładnik, a także obsłużyć przypadki specjalne (NaN, nieskończoność), co zajmuje więcej cykli procesora niż proste porównanie liczb całkowitych.



Na powyższym rysunku zaprezentowano te same dane z paskami błędów reprezentującymi wartości minimalne i maksymalne; widać, że nie tylko czasy średnie, ale i amplituda odchyleń pozostaje porównywalna dla wszystkich trzech typów, co potwierdza, że głównym czynnikiem wpływającym na wydajność Quick Sorta jest rozmiar danych, a nie ich reprezentacja.



Powyższy wykres pokazuje, że rozrzut czasów (wąsy i punkty) jest podobny we wszystkich trzech przypadkach – co świadczy o tym, że stabilność algorytmu sortowania szybkiego nie zależy od typu danych.

6 Wnioski

W tym projekcie zostało zrealizowano implementacje czterech algorytmów sortowania: sortowanie przez wstawianie, sortowanie przez wstawianie binarne, sortowanie przez kopcowanie, sortowanie szybkie. Podczas napisania kodu do każdego algorytmu zapoznano się z mechanizmem jego pracy.

Algorytmy sortowania przez wstawianie oraz binarne przez wstawianie wykazują gwałtowny wzrost czasu działania wraz ze wzrostem liczby elementów – dla $n \geqslant 80\,\mathrm{k}$ stają się niepraktyczne. Wariant binarny redukuje liczbę porównań i jest o około 30% szybszy od klasycznej wersji, jednak koszty przesunięć elementów wciąż rosną jak $O(n^2)$.

Algorytmy o złożoności $O(n \log n)$, czyli sortowanie szybkie i przez kopcowanie, charakteryzują się liniowo–logarytmicznym wzrostem czasu działania. Sortowanie szybkie osiąga najniższe czasy średnie (7–9 ms dla $n = 160\,\mathrm{k}$), natomiast sortowanie przez kopcowanie jest nieco wolniejsze (12–15 ms), ale gwarantuje stałe zachowanie złożoności niezależnie od rozkładu danych.

Wpływ początkowego uporządkowania danych jest kluczowy dla algorytmów kwadratowych – najwolniej sortują dane malejąco posortowane, a najszybciej rosnąco posortowane. W przypadku algorytmów $O(n \log n)$ czasy pozostają stabilne, z wyjątkiem przyspieszonej budowy kopca dla danych malejąco posortowanych.

Zmiana typu danych (int32, float32, float64) przekłada się na niewielkie przesunięcie krzywych czasów sortowania szybkiego – operacje na liczbach zmiennoprzecinkowych wymagają dodatkowych cykli procesora i większego ruchu w pamięci. Nie wpływają istotnie na stabilność ani wariancję działania.

Podsumowując, dla dużych zbiorów najlepszym kompromisem pomiędzy szybkością a przewidywalnością jest sortowanie szybkie (z losowym wyborem pivota), natomiast gdy potrzebujemy gwarancji w najgorszym przypadku, lepszym wyborem będzie sortowanie poprzez kopcowanie. Algorytmy kwadratowe nadają się do niewielkich lub posortowanych rosnąco zestawów danych.

W tym projekcie najtrudniejszą częścią były badania. Łącznie czas na wszystkie badania stanowi 12 godzin pracy komputera bez przerwy. Kolejna rzecz która stanowiła problem, sortowanie na pełnym przedziale wartości typu double. Możliwie to może być związane z nie idealną samodzielną implementacją struktury danych wektora. Dla tego przy badaniach zakres badanych wartości typów float oraz double został obniżony. Zakres sortowanych wartości float głównie został obniżony przez redukcje zakresu dla typu double w celach zachowania spójności badań.

Najprostszym było stworzenie struktury programu oraz rozplanowanie klas, je metod prywatnych i publicznych, wymyślenie w jaki sposób będą zapisywane wyniki badań żeby potem kształt danych był użyteczny dla szybkiej analizy danych.

Do stworzenia wykresów oraz automatyzacji procesu badania został napisany skrypt Python. Od samego początku było zaplanowano korzystanie z API programu OriginLab Pro. Program ten jest wiadomy w kołach inżynierów głównie pracujących na co dzień z dużymi ilościami danych. Niestety przez trudną do zrozumienia i niezbyt dostępną dokumentację API danego programu, została podjęta decyzja o skorzystaniu z biblioteki Python numpy oraz matplotlib, które mają podobne narzędzia jak OriginLab Pro.