Politechnika Wrocławska

ALGORYTMY I ZŁOŻONOŚĆ OBLICZENIOWA

Sprawozdanie projektowe

ANALIZA ALGORYTMÓW SORTOWANIA W JĘZYKU C++

Oleksandr Nedosiek 275978

Projekt, grupa №7

Wtorek TP 9:15

Prowadzący:

mgr inż. Damian Mroziński

Wrocław, 4 maja 2025

Spis treści

1. Cel projektu
2. Wstęp teoretyczny
3. Plan badań
4. Przebieg oraz analiza badań
5. Wnioski
6. Literatura

1. Cel projektu

Celem projektu było zaimplementowanie i przeprowadzenie analizy efektywności oraz złożoności niektórych algorytmów sortowania – sortowanie przez wstawianie, przez kopcowanie, szybkie sortowanie oraz sortowanie Shella. Zakłada się, że sortowanie to uporządkowanie elementów rosnąco.

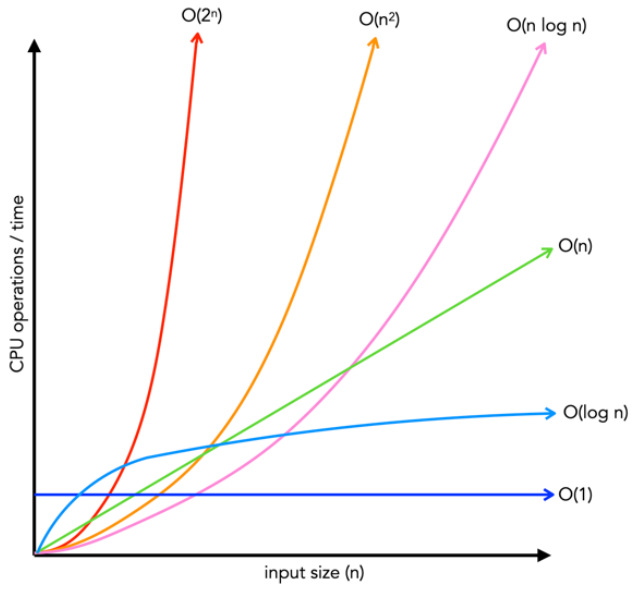
Dodatkowymi celami było nauczenie się podstawowego skryptowania, dopasowania programu do wymagań/istniejących klas oraz mądrego planowania projektu.

Założenia projektowe zostały umieszczone przez prowadzącego w opisie projektu na e-Portalu.

2. Wstęp teoretyczny

2.1. Złożoność czasowa

Złożoność czasowa algorytmów określa, jak szybko rośnie liczba operacji wymaganych do rozwiązania problemu w zależności od rozmiaru danych wejściowych. Wyraża się ją za pomocą notacji dużego O (np. $O(n^2)$, $O(\log n)$). Złożoność czasowa pomaga porównać algorytmy w celu wybrania najbardziej efektywnego dla danego przypadku. Wyróżnia się złożoność w najlepszym, średnim oraz najgorszym (pesymistycznym) przypadkach, co pozwala lepiej oszacować zachowanie algorytmu w różnych sytuacjach.



Rysunek 1. Wykres zależności czasu od liczby elementów różnych złożoności czasowych.

2.2. Wybrane algorytmy

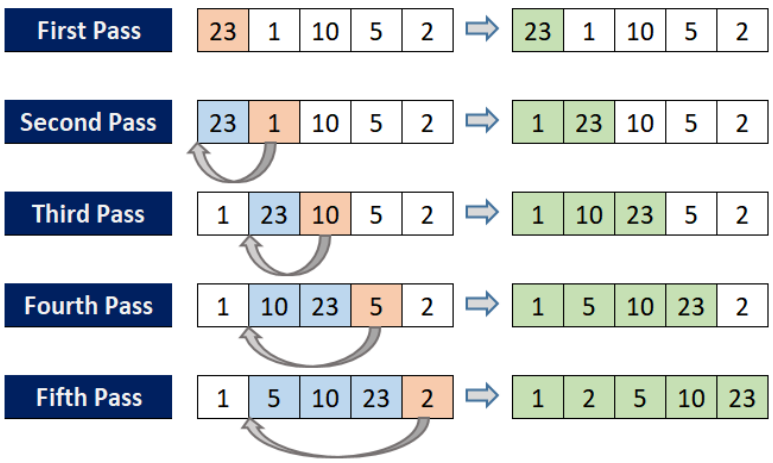
- Sortowanie przez wstawianie (Insertion Sort),
- Sortowanie przez kopcowanie (Heap Sort – wersja podstawowa oraz własna wersja „pijanego studenta”),
- Sortowanie Shella (wybór wielkości odstępów Shella 1959 r. (Shell Sort), Franka Lazarusa 1960 r. (Shell Sort FL)),
- Sortowanie szybkie (Quick Sort - wybór pivota: skrajny lewy lub prawy, środkowy i losowy).

Nazwa algorytmu sortowania	Średnia złożoność obliczeniowa	Pesymistyczna złożoność obliczeniowa
Insertion Sort (przez wstawianie)	$O(n^2)$	$O(n^2)$
Heap Sort (przez kopcowanie)	$O(n \log n)$	$O(n \log n)$
Quick Sort (szybkie)	$O(n \log n)$	$O(n^2)$
Shell Sort (Shella)	$O(n^{\frac{3}{2}})$	$O(n^2)$
Shell Sort FL (Shella Franka Lazarusa)	$O(n \log^2 n)$	$O(n^{\frac{3}{2}})$

Tabela 1. Złożoność obliczeniowa algorytmów.

2.2.1. Sortowanie przez wstawianie (Insertion Sort)

Algorytm sortowania przez wstawianie polega na przejściu przez elementy tablicy od lewej do prawej i wstawianiu każdego elementu w odpowiednie miejsce w już posortowanej części tablicy. Zaletami danego algorytmu są stabilność, łatwa implementacja algorytmu, bezproblemowe działanie z różnymi typami danych oraz odbywa się ono „w miejscu”, czyli nie są używane żadne tablice pomocnicze. Natomiast posiada on także dość istotne wady, takie jak wrażliwość na błędy oraz złożoność pesymistyczna, wynosząca $O(n^2)$ (natomiast dla małych zbiorów nadaję się idealnie). Rysunek 2 pokazuje przykład działania danego sortowania.



Rysunek 2. Graficzny przykład działania sortowania przez wstawianie.

2.2.2. Sortowanie przez kopcowanie (Heap Sort)

Sortowanie przez kopcowanie to algorytm sortowania oparty na takiej strukturze danych, jak kopiec (kopiec maksymalny – korzeń jest elementem największym). Najpierw buduje się odpowiedni kopiec z danych wejściowych, a następnie korzeń jest usuwany i ustawiany na końcu tablicy. Kolejnym krokiem jest naprawianie właściwości kopca. Cały ten proces powtarzany jest do usunięcia ostatniego elementu z kopca. Dany algorytm pochłania mało pamięci i jest na ogół dość szybkim algorytmem sortowania (złożoność czasowa jest taka sama dla każdej ilości elementów wejściowych), natomiast jest to algorytm niestabilny.

2.2.3. Sortowanie Shella (Shell Sort, Shell F. Lazarus Sort)

Algorytm sortowania Shella to usprawniona wersja sortowania przez wstawiania, główna idea którego polega na porównanie elementów oddalonych od siebie o określony krok (w przeciwieństwie do sortowania przez wstawianie, gdzie porównywane są sąsiednie elementy), który z czasem się zmniejsza. Dzięki temu sposobowi elementy przemieszczają się szybciej w kierunku właściwej pozycji. Wadą tego algorytmu jest to, że nie jest on stabilny, oraz dla większych zbiorów danych wejściowych czas pesymistyczny nadal jest taki sam, jak dla sortowania przez wstawianie.

Sortowanie Shella z ciągiem Franka Lazarusa to jedna z optymalizacji klasycznej wersji algorytmu Shella. Różni się ona doborem odstępów, czyli sposobem, w jaki wyznacza się kolejne odległości między porównywanymi elementami.

Wyraz ogólny ciągu $(k \geqslant 1)$	Konkretne odstępy	Rząd złożoności pesymistycznej	Autor i rok publikacji
$\lfloor N/2^k \rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta(N^2)$ [gdy $N = 2^p$]	Shell, 1959[1]
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank, Lazarus, 1960[2]

Rysunek 3. Wzory obliczania odległości w sortowaniu Shella.

2.2.4. Sortowanie szybkie (Quick Sort)

Sortowanie szybkie to jeden z najbardziej efektywnych sposobów porządkowania danych. Wykorzystuje on technikę „dziel i zwyciężaj”, co oznacza, że dzieli problem na mniejsze części, rozwiązuje je osobno, a potem łączy wyniki w jeden ogólny. Na początku wybieramy jeden element z tablicy nazywany pivotem, a następnie dzielimy wszystkie pozostałe elementy na 2 grupy: mniejsze od pivota i większe bądź równe. Teraz sortujemy obie grupy w ten sam sposób – wybieramy pivot i dzielimy. Proces powtarzamy, aż zostaną same jednoelementowe tablice. Na sam koniec łączymy wszystkie części w jedną, posortowaną całość. Dany algorytm jest łatwy do zaimplementowania, świetnie współgra z różnymi typami danych, sortuje „w miejscu”, no i oczywiście – jest bardzo szybki (stąd i pochodzi nazwa ;)). Niestety, algorytm jest niestabilny, a w sytuacji pesymistycznej złożoność wynosi $O(n^2)$.

3. Plan badań

Aby zbadać dane algorytmu sortowania należało zaimplementować je w języku C++, gdzie badane były liczba elementów wejściowych, początkowy rozkład elementów oraz typy danych. Każdy eksperyment był przeprowadzony 100 razy w celu miarodajności badań, a wyniki uśrednione. Obliczone zostały również mediana, minimum, maximum oraz odchylenie standardowe dla każdego przypadku.

3.1. Badanie 1

Badanie 1 polegało na porównaniu czasu działania algorytmów (oraz szczegółowych przypadków algorytmu, jeśli takie istnieją) względem liczebności zbioru sortowanego. Liczebność zbioru dla każdego algorytmu była taka: 10, 20, 30, 40 oraz 50-tysięcy elementów. Algorytmy, analizowane w danym badaniu były takie: sortowanie przez wstawianie, sortowanie przez kopcowanie, sortowanie Shella dla 2 przypadków odstępu – Shella oraz Franka Lazarusa, a także sortowanie szybkie dla 4 przypadków wyboru pivota – skrajny prawy, skrajny lewy, środkowy oraz losowy.

Ponadto, trochę różniącym się przypadkiem było badanie sortowania jednego algorytmu w wersji „pijanego studenta" dla 5 przypadków parametru „pijaństwa". Dla tego przypadku wybrałem sortowanie przez kopcowanie na 10-tysięcach elementów.

3.2. Badanie 2

Badanie 2 polegało na zbadaniu czasu działania algorytmów względem początkowego rozkładu elementów w zbiorze sortowanym. Przypadki rozkładu były takie: zbiór losowy (niesortowany), posortowany rosnąco, posortowany malejąco, posortowany częściowo – 33% oraz 66% sortowanych elementów (33% oraz 66% elementów posortowanych, znajdujących się na początku zbioru sortowanego). Należało zbadać poprzednie algorytmy w wersji „podstawowej" (dla sortowania przez wstawianie oraz sortowania przez kopcowanie, gdzie to była jedyna opcja), a dla sortowania Shella, sortowania szybkiego oraz sortowania przez kopcowanie w wersji pijanego studenta należało wybrać zwycięską opcję pośród wszystkich z Badania 1. Dla sortowania Shella był to odstęp F. Lazarusa, dla sortowania szybkiego – wybranie lewego pivota, a dla sortowania przez kopcowanie w wersji pijanego studenta – wybranie parametru pijaństwa o wartości 50%. Dodatkowo, należało wybrać liczebność zbioru sortowanego, i decyzja padła na 10-tysięczny zbiór.

3.3. Badanie 3

Badanie 3 miało na celu analizę czasu działania algorytmów pod wpływem typu danych w zbiorze. Typy danych były następujące: int, float oraz double. Należało zbadać „zwycięskie" opcję – sortowanie Shella z odstępem F. Lazarusa i sortowanie szybkie z wyborem lewego pivota, a także jeden wybrany algorytm – sortowanie przez kopcowanie w wersji podstawowej. Jako parametry podstawowe wybrana została liczebność zbioru o wartości 10-tysięcy oraz początkowy rozkład elementów w wersji losowej.

4. Przebieg oraz analiza badań

4.1. Badanie 1

4.1.1. Sortowanie szybkie – wybór pivota

W tym rozdziale zbadamy 4 opcje wybranego pivota w sortowaniu szybkim w celu wyłonienia „zwycięskiej" opcji. Poniżej przedstawione są tabele wyników badania oraz wykresy, porównujący wszystkie opcje.

Quick Sort (Lewy pivot)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	0,78	1,99	3,48	5,00	6,14
Odchylenie standardowe [ms]	1,62	3,01	3,44	1,39	1,41
Mediana [ms]	0	0	2	5	6
Minimum [ms]	0	0	0	2	3
Maximum [ms]	8	10	10	8	9

Tabela 2. Wyniki badań lewego pivota w Quick Sorcie.

Quick Sort (Prawy pivot)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,03	2,31	3,11	4,94	6,82
Odchylenie standardowe [ms]	2,06	2,86	3,46	3,56	1,62
Mediana [ms]	0	1	1	6	7
Minimum [ms]	0	0	0	0	3
Maximum [ms]	8	9	10	12	10

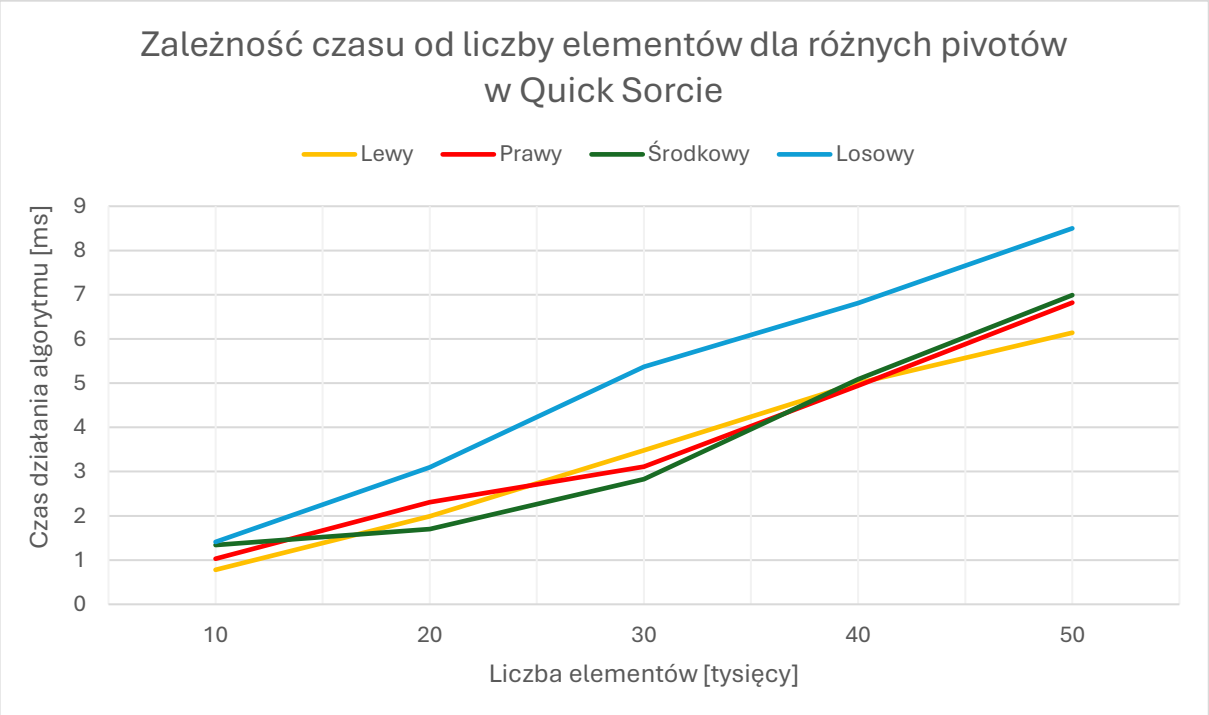
Tabela 3. Wynik badań prawego pivota w Quick Sorcie.

Quick Sort (Środkowy pivot)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,34	1,70	2,83	5,09	6,99
Odchylenie standardowe [ms]	2,31	2,63	3,34	1,17	3,07
Mediana [ms]	0	0	1	5	8
Minimum [ms]	0	0	0	2	0
Maximum [ms]	8	9	10	7	15

Tabela 4. Wyniki badań środkowego pivota w Quick Sorcie.

Quick Sort (Losowy pivot)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,41	3,10	5,37	6,81	8,5
Odchylenie standardowe [ms]	2,41	3,19	3,45	1,65	3,81
Mediana [ms]	0	2	7	7	9
Minimum [ms]	0	0	0	3	0
Maximum [ms]	9	10	11	10	17

Tabela 5. Wyniki badań losowego pivota w Quick Sorcie.



Wykres 1. Zależność czasu działania algorytmu od liczby elementów w zbiorze sortowanym dla różnych opcji pivotu w Quick Sorcie.

Analizując powyższe tabele i wykres możemy stwierdzić, iż wybór pivota na podstawie badań względem liczebności nie jest miarodajny, ponieważ każda z opcji daje zbliżony wynik. Aczkolwiek, żeby wyłonić „zwycięzcę" brałem do uwagi stabilność oraz czas działania algorytmu dla większego zbioru, i w taki sposób wybrany został **lewy** pivot. Losowy pivot miał ogólnie większy czas działania, niż reszta opcji, a pivoty prawy i środkowy miałe prawie identyczne czasy do lewego, lecz na zbiorze 50-tysięcy czasy działania były większe. Ponadto, lewy pivot był najbardziej stabilny, bez skoków czasu w zależności od liczebności. W kolejnych badań zauważymy, że ogólny wybór lewego pivota jednak nie jest najlepszą opcją.

4.1.2. Sortowanie Shella – wybór odstępu

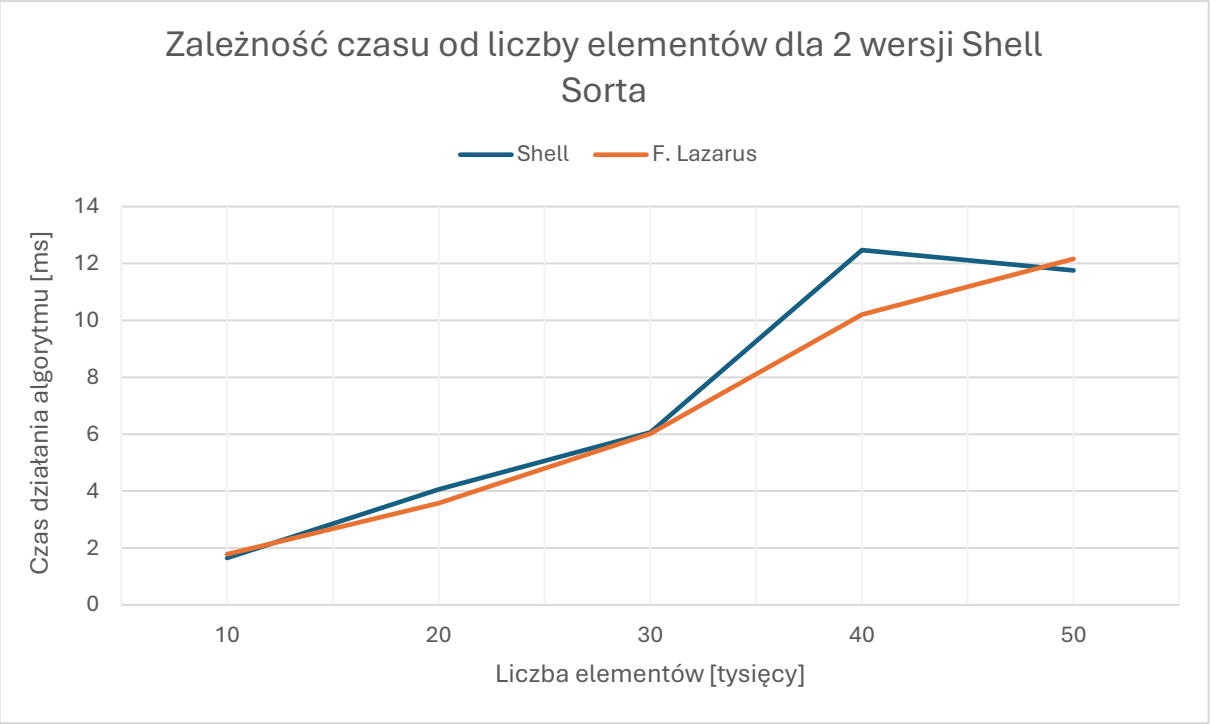
W tym rozdziale zbadamy 2 opcje wybranych odstępów w sortowaniu Shella w celu wyłonienia „zwycięskiej" opcji. Poniżej przedstawione są tabele wyników badania oraz wykresy, porównujący te opcje.

Shell Sort (wersja podstawowa)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,65	4,06	6,06	12,47	11,76
Odchylenie standardowe [ms]	2,65	3,36	3,46	4,98	4,42
Mediana [ms]	0	4	7,5	14	10,5
Minimum [ms]	0	0	0	0	0
Maximum [ms]	9	10	15	19	20

Tabela 6. Wyniki badań odstępu podstawowego w Shell Sorcie.

Shell Sort (odstęp Franka Lazarusa)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,78	3,58	6,02	10,2	12,16
Odchylenie standardowe [ms]	2,73	3,75	3,42	3,91	4,60
Mediana [ms]	0	2	7	9	12,5
Minimum [ms]	0	0	0	0	0
Maximum [ms]	10	13	12	18	24

Tabela 7. Wyniki badań odstępu Franka Lazarusa w Shell Sorcie.



Wykres 2. Zależność czasu działania algorytmu od liczby elementów w zbiorze sortowanym dla różnych opcji odstępu w Shell Sorcie.

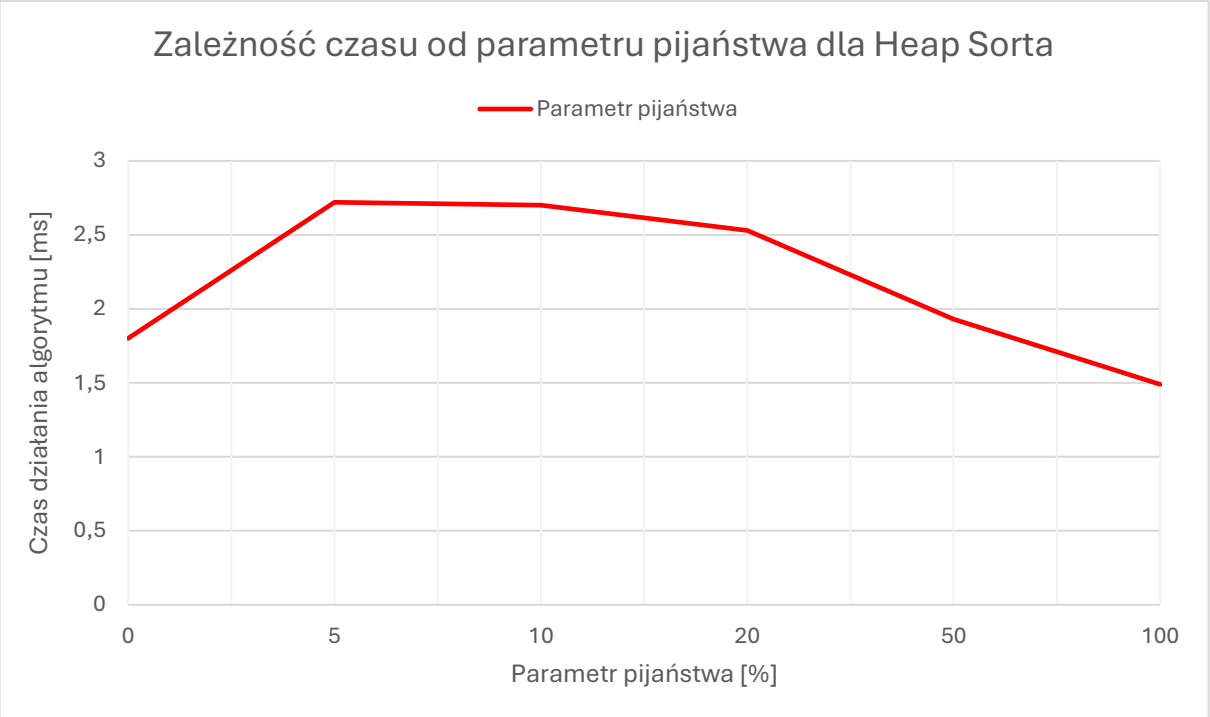
Porównując wyniki sortowania Shella z zastosowaniem klasycznych odstępów (Shella) do wyników uzyskanych przy implementacji Shella z odstępami Franka Lazarusa, widać różnicę w wydajności danych opcji. Dla prawie każdego badanego przypadku czasy działania algorytmu w wersji Lazarusa są nieco mniejsze, niż w wersji klasycznej, przy czym przypadki, gdzie wyniki są zbliżone, rożnica jest bardzo mała. Odstępy Lazarusa sugerują jednak bardziej stabilną zależność, co i daje mu przewagę nad odstępem klasycznym.

4.1.3. Sortowanie przez kopcowanie w wersji „pijanego studenta” – wybór parametru „pijaństwa”

W tym rozdziale zbadamy 5 opcji wybranego parametru pijaństwa w sortowaniu przez kopcowanie w celu wyłonienia „zwycięskiej” opcji. Poniżej przedstawione są tabela wyników badania oraz wykresy, porównujący wszystkie opcje. Dany rozdział różni się od wcześniejszych, ponieważ badana jest tutaj nie liczebność zbioru, lecz parametr „pijaństwa”, a liczebność zbioru została wybrana samodzielnie i wynosi 10 000 elementów.

Heap Sort (wersja „pijanego studenta”)						
Parametr pijaństwa [%]	0	5	10	20	50	100
Średnia [ms]	1,80	2,72	2,70	2,53	1,93	1,49
Odchylenie standardowe [ms]	2,52	0,47	0,63	0,56	0,50	0,50
Mediana [ms]	0	3	3	2,5	2	1
Minimum [ms]	0	2	2	2	1	1
Maximum [ms]	8	4	4	4	3	2

Tabela 8. Wyniki badań parametru pijaństwa dla wersji "pijanego studenta" Heap Sorta.



Wykres 3. Zależność czasu działania algorytmu od parametru pijaństwa dla Heap Sorta w wersji "pijanego studenta".

Badając parametry pijaństwa dla sortowania przez kopcowanie w wersji „pijanego studenta” można zauważyć kilka ciekawych rzeczy, których nawet, szczerze mówiąc, się nie

spodziewałem. Implementacja „pijanego studenta” polegała na tym, że wybrany parametr oznaczał procent szansy na losowe użycie metody HEAPIFY (przywracanie własności kopca). Czyli dla parametru pijaństwa 50% - 50% szans na to, że w losowym momencie zostanie użyta metoda HEAPIFY dla losowej wartości. Losowe użycia metody zwykle bardzo utrudniają działanie programu, natomiast z tego, co możemy zobaczyć – wcale nie zawsze tak jest. Parametr 0% został wybrany ze zwykłej metody sortowania przez kopcowanie, chociaż wyniki metody z wersją „pijanego studenta” były prawie identyczne. Gdy parametr wynosi 0%, nic się nie dzieję, a więc czas działania algorytmu jest „zwykły”, natomiast gdy ten parametr jest różny od 0%, dzieją się ciekawe rzeczy. Im **większy** jest parametr – tym **szybciej** działa algorytm! Niesamowite! (badania należy przeprowadzić w wersji rzeczywistej, żeby upewnić się, czy tak rzeczywiście jest w wersji pijanego studenta). Właśnie po przeprowadzeniu badań dla 0, 5, 10, 15 i 20% zauważyłem, że takie coś ma miejsce, zdecydowałem się zmienić parametry pijaństwa i byłem bardzo zaskoczony.

Tutaj nie ma za bardzo opcji do wyłonienia zwycięscy, a więc zdecydowałem się wybrać ten „najciekawszy” parametr, czyli 50%. Albo pijany, albo nie. Wyniki dalszych badań mogą pójść w obie strony, a więc wybór jest taki.

4.1.4. Porównanie reszty algorytmów i opcji zwycięskich niektórych algorytmów

W tym rozdziale zweryfikujemy, który algorytm jest najlepszy do wykorzystania względem liczebności zbioru sortowanego. Badania sprawdzają algorytmy sortowania przez wstawianie, przez kopcowanie, sortowanie Shella w wersji Franka Lazarusa oraz sortowanie szybkie z wyborem lewego pivotu. Poniżej znajdują się tabele wyników badań każdego algorytmu oraz wykres, porównujący te algorytmy.

Insertion Sort					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	45,92	211,02	451,18	892,51	1438,65
Odchylenie standardowe [ms]	12,14	35,30	61,90	94,20	112,05
Mediana [ms]	36,5	216	461	909	1445,5
Minimum [ms]	34	139	287	554	869
Maximum [ms]	70	284	565	1122	1597

Tabela 9. Wyniki badań dla Insertion Sort.

Heap Sort					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,80	2,40	4,88	7,17	9,10
Odchylenie standardowe [ms]	2,53	3,20	3,44	3,30	3,51
Mediana [ms]	0	1	7	8	9
Minimum [ms]	0	0	0	0	0
Maximum [ms]	8	10	10	18	18

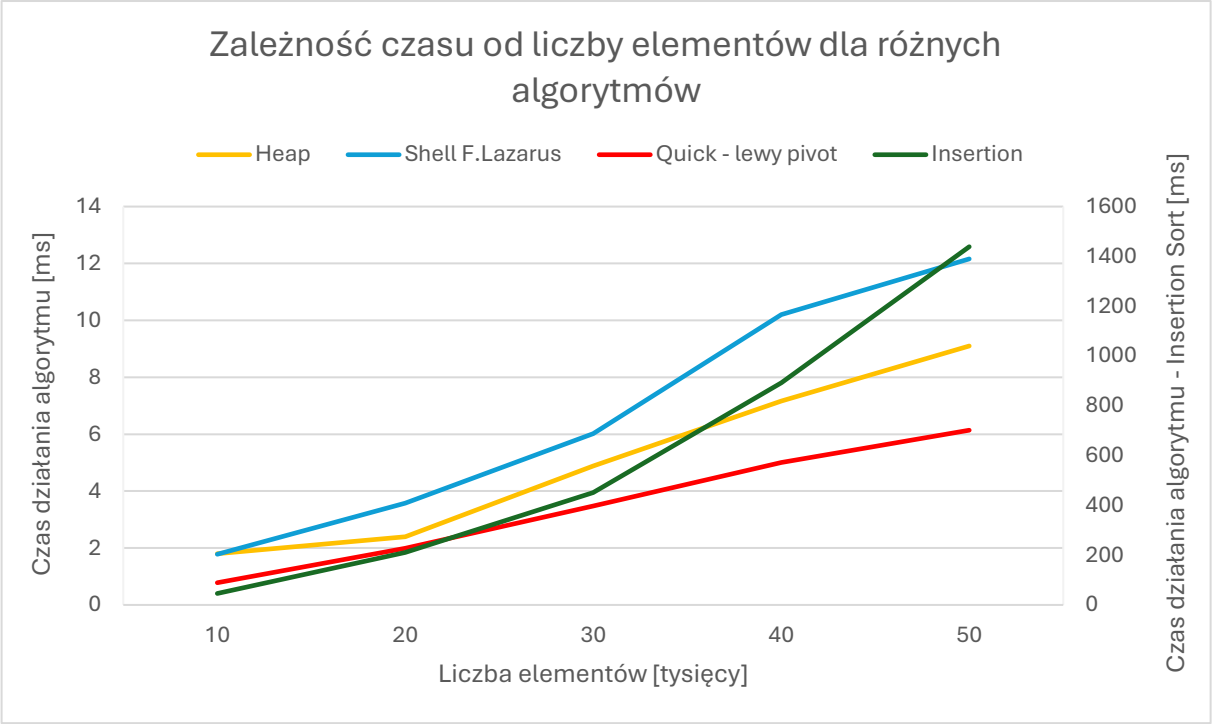
Tabela 10. Wyniki badań dla Heap Sorta.

Shell Sort (Odstęp Franka Lazarusa)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	1,78	3,58	6,02	10,2	12,16
Odchylenie standardowe [ms]	2,73	3,75	3,42	3,91	4,60
Mediana [ms]	0	2	7	9	12,5
Minimum [ms]	0	0	0	0	0
Maximum [ms]	10	13	12	18	24

Tabela 11. Wyniki badań dla Shell Sorta z odstępem F. Lazarusa.

Quick Sort (Lewy pivot)					
Liczebność [n]	10 000	20 000	30 000	40 000	50 000
Średnia [ms]	0,78	1,99	3,48	5,00	6,14
Odchylenie standardowe [ms]	1,62	3,01	3,44	1,40	1,41
Mediana [ms]	0	0	2	5	6
Minimum [ms]	0	0	0	2	3
Maximum [ms]	8	10	10	8	9

Tabela 12. Wyniki badań dla Quick Sorta z lewym pivotem.



Wykres 4. Zależność czasu działania algorytmu od liczby elementów dla badanych algorytmów.

UWAGA! Dla algorytmu Insertion Sort parametry osi OY są zaznaczone po prawej stronie wykresu z powodu zbyt dużego czasu działania danego algorytmu w celu lepszego skalowania wykresu i lepszego widzenia różnicy pomiędzy resztą algorytmów.

Analizując powyższe tablice oraz wykres możemy potwierdzić niektóre teoretyczne stwierdzenia/fakty dla każdego z badanych algorytmów.

Quick Sort potwierdza swoją nazwę, ponieważ działa najszybciej niemalże dla każdej liczebności zbioru, najlepiej widać to dla większych zbiorów, gdzie różnica wynosi ponad 3 ms. Jest on także bardzo stabilny, nie ma żadnych skoków czasu.

Insertion Sort bardzo się różni od reszty algorytmów, ponieważ przykładowo iloraz czasu działania tego algorytmu dla 50-tysięcznego wzoru a czasu działania Quick Sorta wynosi około 250 (1500/6 = 250). Można też zauważyć, iż dla mniejszych zbiorów – 10k i 20 k – jest to najszybszy algorytm, co potwierdza jego efektywność dla małych zbiorów elementów.

Shell Sort w wersji odstępu Franka Lazarusa, jako ulepszona wersja Insertion Sorta, zdecydowanie działa lepiej i szybciej, a nawet może się porównywać z innymi algorytmami sortowania. Aczkolwiek jest trochę wolniejszy od reszty, i różnica ta jest większa im większy jest zbiór elementów. A także jest bardzo niestabilny, ciągle skoki góra/dół/

Heap Sort – to Heap Sort. Porównywalnie stabilny, lecz działa gorzej dla mniejszych zbiorów. Dla zbioru o 10k jest najwolniejszym pośród badanych (nie licząc Insertion Sorta), natomiast dla większych zbiorów czas jest ciągle o 2 ms większy, niż ma nasz „zwycięzca” Quick Sort.

4.2. Badanie 2

W tym badaniu skupiamy się nie na liczebności zbioru sortowanego (będzie on taki sam dla każdego algorytmu – 10 000 elementów), lecz na początkowym rozkładzie elementów. Badamy 4 algorytmy z punktu 4.1.2. (sortowanie przez wstawiania, sortowanie przez kopcowanie, sortowanie Shella z odstępem Franka Lazarusa, sortowanie szybkie z wyborem lewego pivota) oraz sortowanie przez kopcowanie w wersji „pijanego studenta" z parametrem pijaństwa o wartości 50%. Tablice z wynikami badań oraz diagram, który porównuje te algorytmy przedstawiono niżej.

Insertion Sort					
Początkowy rozkład tablicy	Losowa	Malejąca	Częściowo – 33%	Częściowo – 66%	Rosnąca
Średnia [ms]	50,2	119,07	44,33	27,61	0,08
Odchylenie standardowe [ms]	17,02	30,33	16,19	9,23	0,80
Mediana [ms]	40	133	34	25	0
Minimum [ms]	32	64	24	15	0
Maximum [ms]	90	169	83	54	8

Tabela 13. Wyniki badań dla Insertion Sorta.

Heap Sort (Wersja podstawowa)					
Początkowy rozkład tablicy	Losowa	Malejąca	Częściowo – 33%	Częściowo – 66%	Rosnąca
Średnia [ms]	1,08	1,30	1,27	1,08	1,37
Odchylenie standardowe [ms]	2,00	2,53	2,64	2,16	2,63
Mediana [ms]	0	0	0	0	0
Minimum [ms]	0	0	0	0	0
Maximum [ms]	8	9	9	8	9

Tabela 14. Wyniki badań dla wersji podstawowej Heap Sorta.

Shell Sort (Odstęp Franka Lazarusa)					
Początkowy rozkład tablicy	Losowa	Malejąca	Częściowo – 33%	Częściowo – 66%	Rosnąca
Średnia [ms]	1,51	0,57	1,39	1,35	0,33
Odchylenie standardowe [ms]	2,55	1,70	2,60	2,61	1,14
Mediana [ms]	0	0	0	0	0
Minimum [ms]	0	0	0	0	0
Maximum [ms]	10	9	10	9	8

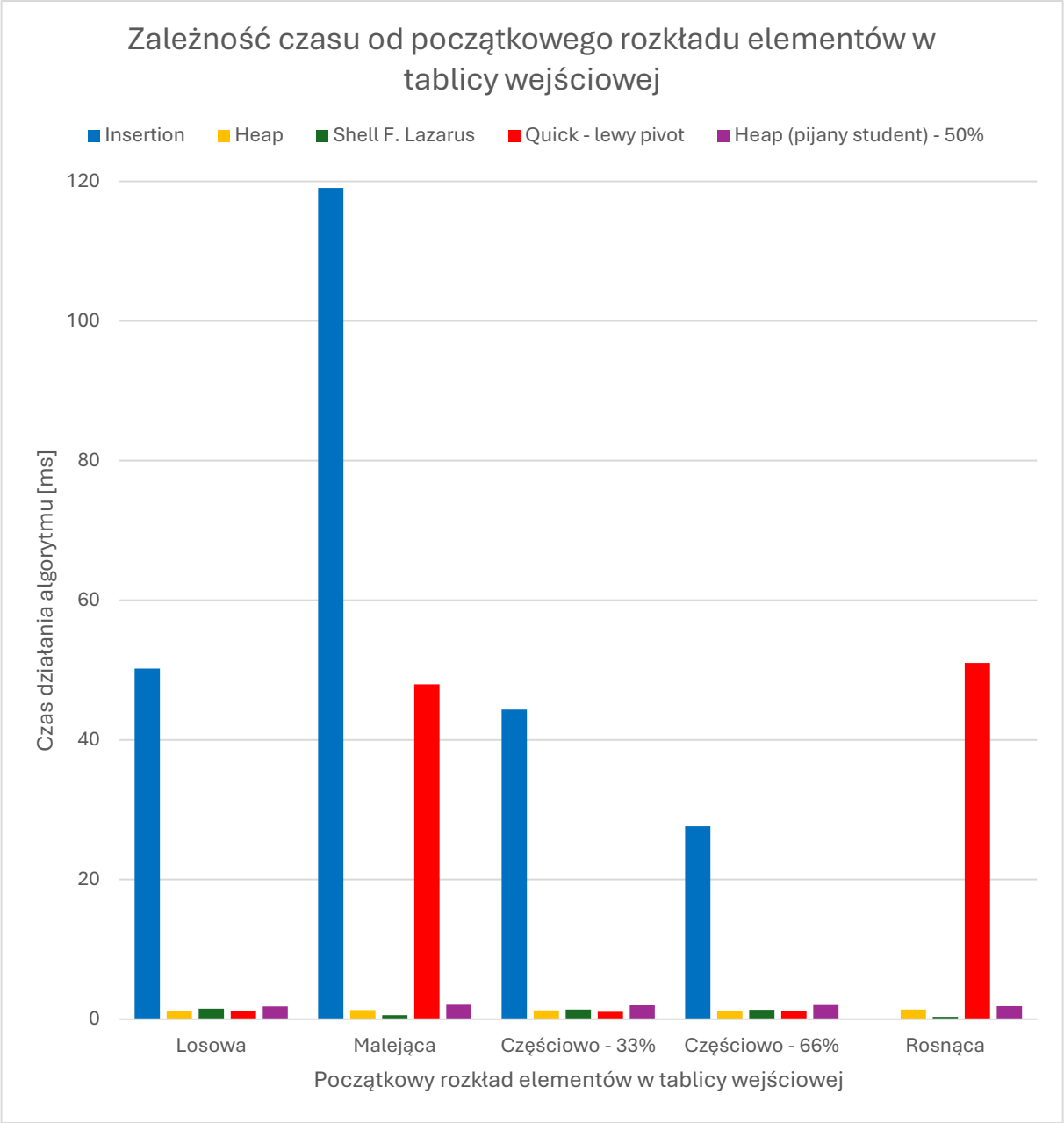
Tabela 15. Wyniki badań dla Shell Sorta z odstępem F. Lazarusa.

Quick Sort (Lewy pivot)					
Początkowy rozkład tablicy	Losowa	Malejąca	Częściowo – 33%	Częściowo – 66%	Rosnąca
Średnia [ms]	1,22	47,97	1,07	1,16	51,03
Odchylenie standardowe [ms]	2,40	10,75	2,26	2,23	6,21
Mediana [ms]	0	50	0	0	50
Minimum [ms]	0	23	0	0	24
Maximum [ms]	9	69	8	8	73

Tabela 16. Wyniki badań dla Quick Sorta z wyborem lewego pivota.

Heap Sort (wersja „pijanego studenta” – 50% pijaństwa)					
Początkowy rozkład tablicy	Losowa	Malejąca	Częściowo – 33%	Częściowo – 66%	Rosnąca
Średnia [ms]	1,81	2,07	1,97	2,03	1,86
Odchylenie standardowe [ms]	0,66	0,38	0,85	0,17	0,49
Mediana [ms]	2	2	2	2	2
Minimum [ms]	1	1	1	2	1
Maximum [ms]	4	3	4	3	3

Tabela 17. Wyniki badań dla Heap Sorta w wersji "pijanego studenta" z parametrem pijaństwa 50%.



Wykres 5. Diagram zależności czasu od początkowego rozkładu elementów w tablicy wejściowej dla badanych algorytmów.

Porównując powyższe algorytmy widać ogromną różnicę pomiędzy czasem działania poszczególnych algorytmów.

Najbardziej stabilnymi są Heap Sort-y (w obu wersjach), które prawie nie różnią się czasem względem początkowego rozkładu elementów wejściowych (jedyna różnica – wywnioskowana w Badaniu 1 – Heap Sort w wersji pijanej działa odrobinę dłużej, niż w wersji podstawowej.

Reszta algorytmów ma ciekawe przypadki do zbadania względem początkowego rozkładu, a więc skupimy się na nich bardziej. Zaczynając od najbardziej „widocznego" algorytmu – Insertion Sort. Chociaż upewniliśmy się w Badaniu 1, że działa on ogólnie bardzo długo, możemy zauważyć, iż w przypadku tablicy posortowanej rosnąco (default) działa on niemalże 0 ms. Dzieje się tak dlatego, że algorytm przechodzi jeden raz przez cała tablicę, nie znajduję żadnych

„niepasujących” elementów i kończy swoje działanie po tej jednej iteracji. W przypadku tablicy losowej i posortowanej częściowo (33%) wyniki są bardzo podobne, z czego możemy wywnioskować, że małe częściowe sortowanie nie pomaga temu algorytmu. Aczkolwiek częściowe sortowanie (66%) już działa lepiej, bo sortuje prawie 2 razy szybciej, niż w przypadku losowej tablicy. No i najgorszy przypadek – tablica malejąca, sortowanie której odbywa się 2.5 razy wolniej, niż w przypadku tablicy losowej. Algorytm zaczyna działanie od skrajnego lewego elementu, który w przypadku tablicy malejącej jest najmniejszy i musi przejść cały zbiór, aż do końca tablicy, żeby wstawić go tam. I to się powtarza aż nie posortuje wszystkie elementy z „powrotem”.

Shell Sort z odstępem F. Lazarusa, jako ulepszona wersja Insertion Sorta, działa zdecydowanie lepiej i bardzo szybko dla każdej tablicy. Podobnie, jak i Insertion Sort, dla tablicy posortowanej czas działania algorytmu zliża się do 0. Aczkolwiek w przypadku tablicy malejącej – zachowuje się kompletnie inaczej. Działa 3 razy szybciej, niż reszta przypadków, prawdopodobnie dlatego, że wylicza odstęp i „nie powtarza swoich błędów”, jak to robi Insertion Sort. Reszta przypadków ma podobny czas działania.

Quick Sort z lewym pivotem też ma ciekawe zachowanie. W przypadku tablic losowej i posortowanych częściowo działa szybko i w podobnym czasie, aczkolwiek dla już posortowanej tablicy (rosnąco bądź malejąco) – o wiele dłużej – z 30 razy więcej. Dzieje się tak dlatego, że tablica nie jest dzielona na równe części, jak w innych przypadkach, lecz na części o 1 elemencie i n-1, co znacznie utrudnia działanie na zbiorze.

4.3. Badanie 3

W danym badaniu działamy na różnych typach danych. Jako dane podstawowe wybieramy losowy zbiór o liczebności 10 000 elementów. Badamy 3 algorytmy: sortowanie przez kopcowanie, sortowanie Shella z odstępem Franka Lazarusa, sortowanie szybkie z wyborem lewego pivota. Tablice z wynikami badań oraz diagram, który porównuje te algorytmy przedstawiono niżej.

Heap Sort (Wersja podstawowa)			
Typ danych	int	float	double
Średnia [ms]	1,34	1,7	1,57
Odchylenie standardowe [ms]	2,35	2,50	2,48
Mediana [ms]	0	0	0
Minimum [ms]	0	0	0
Maximum [ms]	9	9	9

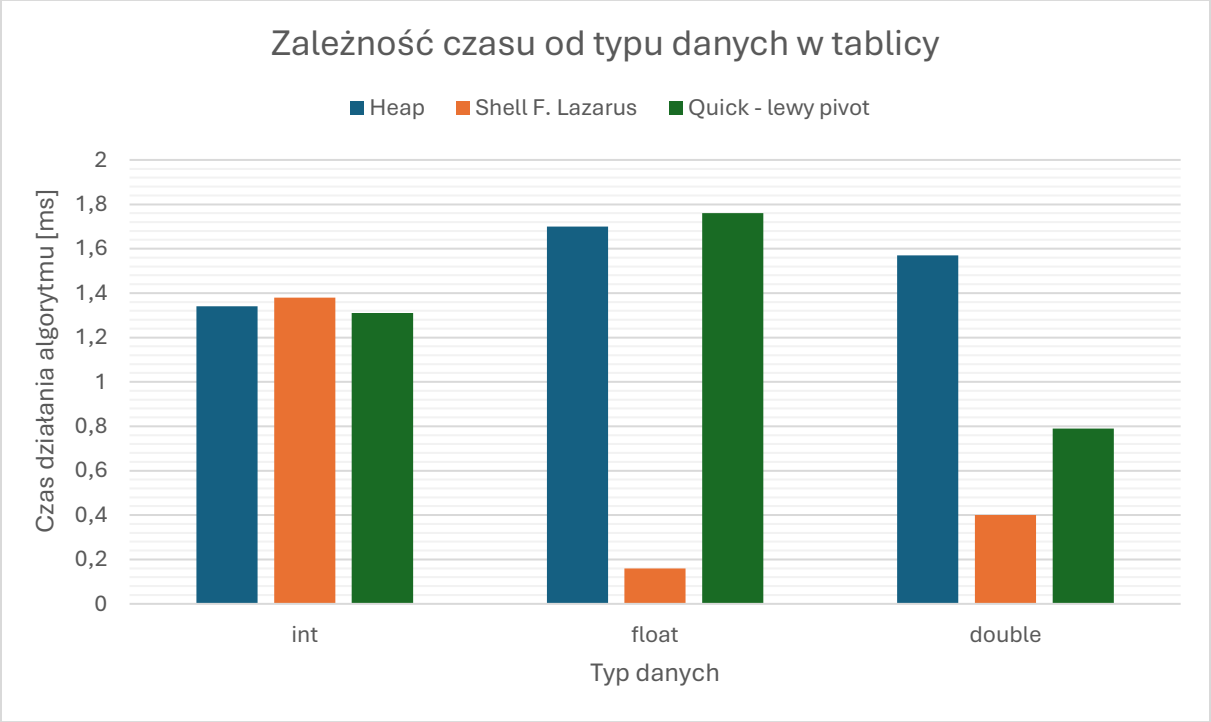
Tabela 18. Wyniki badań dla Heap Sorta.

Shell Sort (Odstęp Franka Lazarusa)			
Typ danych	int	float	double
Średnia [ms]	1,38	0,16	0,4
Odchylenie standardowe [ms]	2,32	0,90	1,34
Mediana [ms]	0	0	0
Minimum [ms]	0	0	0
Maximum [ms]	8	8	8

Tabela 19. Wyniki badań dla Shell Sorta z odstępem Franka Lazarusa.

Quick Sort (Lewy pivot)			
Typ danych	int	float	double
Średnia [ms]	1,31	1,76	0,79
Odchylenie standardowe [ms]	2,55	2,60	1,58
Mediana [ms]	0	0	0
Minimum [ms]	0	0	0
Maximum [ms]	9	9	8

Tabela 20. Wyniki badań dla Quick Sorta z lewym pivotem.



Wykres 6. Zależność czasu działania algorytmu od typu danych w tablicy.

Analizując powyższe dane, możemy stwierdzić, iż Heap Sort działa podobnie dla każdego typu danych, podobnie jak i Quick Sort z lewym pivotem, który jednak działa szybciej dla double i wolniej dla float, co jest trochę dziwne. Aczkolwiek największym zaskoczeniem jest Shell Sort z odstępem F. Lazarusa, ponieważ on działa podobnie dla int-ów jak i reszta algorytmów, ale dla float i double – zdecydowanie szybciej. Być może algorytm Lazarusa inaczej „czyta" implementacje szablonów, niż inne algorytmy. Konkretnej odpowiedzi jednak dlaczego tak się dzieję podać nie umiem.

5. Wnioski

Przeprowadzone badania wykazały znaczące różnice pomiędzy algorytmami. Potwierdziła się większość teoretycznych stwierdzeń na temat badanych algorytmów.

Każdy z algorytmów ma swoje wady i zalety, przez co musimy zawsze oszacować, jakiego algorytmu warto użyć do danego problemu. Quick Sort działa stabilnie dla każdej liczebności danych, aczkolwiek należy odpowiednio wybrać pivota. W przypadku lewego nie warto sortować już posortowanych tablic, w przypadku prawego jest na odwrót. W przypadku środkowego i losowego Quick Sort działa jak powinien, szybko i stabilnie. Insertion Sorta warto używać tylko dla mniejszych zbiorów danych, ewentualnie do sortowania posortowanej rosnąco tablicy, w innych przypadkach proces będzie bardzo wydłużony. Sortowanie Shella w obu przypadkach wykazało dobre wyniki, aczkolwiek dla większych zbiorów odstęp Lazarusa jest lepszy. No i na koniec – Heap Sort. Mój ulubieniec. Działa bardzo stabilnie w każdym przypadku – małe zbiory, duże zbiory, posortowane bądź losowe tablicy, różne typy danych – zawsze szybko da wynik. A jego pijana wersja też nie różni się zbyt dużo, a nawet dla największego parametru pijaństwa działa lepiej, niż wersja oryginalna.

Liczebność zbioru ma bardzo duży wpływ na Insertion Sort, im większy zbiór, tym więcej nieprzespanych nocy spędzi się nad komputerem czekając na wyniki. Początkowy rozkład tablicy ma ogromny wpływ na Quick Sorta (szczególnie na skrajne pivoty) i też na Insertion Sort. Typ danych, jak się okazało, też dość istotnie wpływa na czas działania algorytmów. Jest to oczywiste, że int-y są „latwiejsze" dla komputera, a więc wykonuje się je szybciej, niż float i double, aczkolwiek Shell Sort z odstępem Lazarusa pokazał co innego – float-y i double-y są dla niego lepsze! (być może to zależy od zaimplementowanego programu, a nie samego algorytmu).

Warto też zwrócić uwagę na złożoność obliczeniową. Pomimo tego, że jest ona taka sama dla Quick Sorta i Heap Sorta, złożoność czasowa jest inna, co mogliśmy wywnioskować z naszych badań. Algorytmy rekurencyjne zużyją więcej pamięci, i będą trwać o wiele dłużej z powodu większej ilości przeprowadzonych kroków.

Oszacowanie, którego algorytmu użyć do rozwiązania danego problemu jest niezwykle ważne w celu mniejszego zużycie pamięci i najważniejsze – czasu.

6. Literatura

- Thomas H. Cormen, Wprowadzenie do algorytmów, PWN, 2022
- https://pl.wikipedia.org/wiki/Sortowanie_Shella
- https://www.geeksforgeeks.org/insertion-sort-algorithm/
- https://miro.medium.com/v2/resize:fit:632/1*nk42coSk8TO8GOZdTZ6phA.png