POLITECHNIKA WROCŁAWSKA **Wydział Informatyki i Telekomunikacji**	
Algorytmy i złożoność obliczeniowa	
Rok akademicki:	2024/2025
Autor projektu:	Daniel Gościński
Nr indeksu:	280878
Temat projektu:	BADANIE EFEKTYWNOŚCI ALGORYTMÓW SORTOWANIA

Oświadczenie: Przekazując to sprawozdanie do oceny prowadzącemu zajęcia Autor oświadcza, że zostało ono przygotowane samodzielnie, bez udziału osób trzecich oraz że żadna jego część nie jest plagiatem

I. Spis Treści

II. Wstęp teoretyczny

Problem sortowania to jeden z podstawowych problemów informatyki. Polega on na uporządkowaniu określonego zbioru danych według zadanych cech. Jednym ze szczególnych przypadków sortowania jest sortowanie liczb rosnąco lub malejąco, który umożliwia łatwiejszą dla człowieka prezentację danych lub łatwiejszą dalszą ich modyfikację. Ponieważ istnieje wiele algorytmów sortowania, utworzono klasyfikację, umożliwiające łatwiejsze określanie cech algorytmów sortowania:

- Złożoność obliczeniowa: określana dla najlepszej, najgorszej i uśrednionej kombinacji elementów w tablicy w zależności od ilości elementów. Określa się poprzez notację duże O. Za dobrą złożoność uznaje się złożoność średnią O(nlogn) oraz pesymistyczną $O(n^2)$.
- Zużycie pamięci: algorytm potrzebuje pamięci do wykonania swoich operacji. Wyróżniamy takie warianty jak O(1), gdzie algorytm zużywa stałą ilość pamięci bez względu na rozmiar danych n, O(n) gdzie wykorzystana pamięć będzie proporcjonalna do ilości elementów, oraz $O(n^2)$, w którym dla każdej komórki n zostanie zużyte n komórek pamięci.
- Stabilność: stabilny algorytm sortowania to taki, który nie zamienia miejscami takich samych elementów (czyli przykładowo dwie takie same liczby w różnych miejscach tablicy po posortowaniu nie zostaną skrzyżowane)

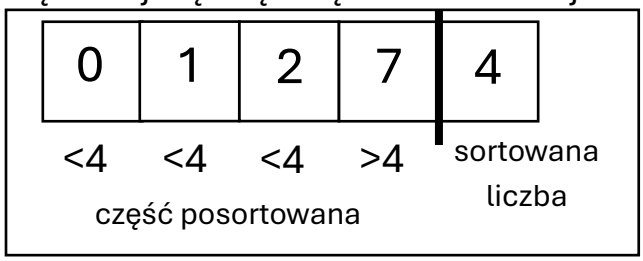
W tym projekcie zbadano algorytm sortowania przez wstawianie. Opisywany jest on jako stabilny algorytm o złożoności optymistycznej O(n), pesymistycznej i średniej $O(n^2)$. Ponieważ wykonuje on operacje na sortowanej tablicy, zużywa on stałą ilość pamięci (O(1)), potrzebując jedynie miejsce na przechowanie zamienianej wartości. Badane jest 5 wariantów algorytmu sortowania, gdzie w czterech z nich zmieniany jest kierunek sortowania oraz kierunek wstawiania liczby (więcej o tym w sekcji III). Piąty algorytm wykorzystuje przeszukiwanie binarne w celu ustalenia ostatecznej pozycji liczby.

III. Opis działania algorytmu

W celu zrozumienia sposobu działania algorytmu należy wyobrazić sobie podział tablicy na dwa segmenty: część zawierającą posortowane już liczby oraz część nieposortowaną. W ten sposób uzyskujemy pierwszą zależność.

Rys 1. Podział tablicy w klasycznym sortowaniu przez wstawianie

Następną zależność, jaką możemy zaobserwować w celu lepszego zrozumienia sortowania jest to, jak dzieli się sekcja posortowana. Kiedy bierzemy i-tą liczbę, możemy podzielić liczby posortowane na te większe od wybranej liczby, oraz te mniejsze od wybranej liczby. Zależność ta pozwala nam w łatwy sposób uwarunkować, jak długo chcemy przesuwać liczbę w zbiorze (sprowadza się to do prostego „przesuwaj tak długo aż nie napotkasz liczbę mniejszą/większą od sortowanej liczby).



Rys 2. Podział segmentu posortowanego względem sortowanej liczby

Znając te dwie zależności można przystąpić do zrozumienia działania algorytmu, najpierw wytłumaczony zostanie podstawowy algorytm sortowania przez wstawianie, co umożliwi łatwiejsze zrozumienie jego wariantów. W celu ułatwienia tłumaczenia uznajemy, że sortujemy liczby rosnąco.

```
Dane:
n – ilość elementów w tablicy
tablica[0,..,n-1]- tablica do posortowania


dla i od 1 do n - 1
        j ← tablica[i]
        j ← i - 1
        dopóki j ≥ 0 oraz x < tablica[j]
                tablica[j+1] ← tablica[j]
                j ← j-1
        tablica[j+1] ← x
```

Kod 1. Pseudokod algorytmu sortowania przez wstawianie

Sortowanie zaczynamy od drugiego elementu tablicy, ponieważ zgodnie z pierwszą obserwacją wszystkie elementy wcześniejsze są już posortowane. Następnie sprawdzamy, czy nasza liczba jest mniejsza od liczby poprzedniej. Jeśli jest, to przesuwamy porównywaną liczbę w prawo. Czynność powtarzamy tak długo, aż nie znajdziemy się albo na początku tablicy, albo nie znajdziemy liczby mniejszej niż sortowana liczba. W tym przypadku wstawiamy liczbę na ustalone miejsce. Sekwencję czynności powtarzamy dla każdej liczby tak długo, aż nie dojdziemy do końca tablicy.

Po zapoznaniu się z metodą działania algorytmu można łatwiej zrozumieć pozostałe warianty algorytmu.

Modyfikacje algorytmu polegają na zmianie, z której strony tablicy wykonujemy sortowanie oraz z której strony części posortowanej wykonujemy przeszukiwanie. Analizując powyższy algorytm można wywnioskować, że podstawowy wariant sortowania sortuje od lewej strony tablicy, oraz przeszukuje część posortowaną od prawej strony. Z tego powodu podstawowy wariant sortowania określany będzie w reszcie dokumentu jako sortLP, gdzie pierwsza litera oznacza, od której strony tablicy wykonujemy sortowanie, a druga litera oznacza, od której strony części posortowanej wykonujemy przeszukiwanie.

Pierwszą zmianę algorytmu, jaki możemy zrobić jest sortowanie od końca tablicy. W ten sposób powstanie drugi wariant sortPL.

```
Dane:
n – ilość elementów w tablicy
tablica[0,..,n-1]- tablica do posortowania

dla i od n-2 do 0
        x ← tablica[i]
        j ← j + 1

        dopóki j < n oraz x > tablica[j]
                tablica[j-1] ← tablica[j]
                j ← j + 1
        tablica[j-1] ← x
```

Kod 2. Pseudokod sortPL, zmodyfikowanego algorytmu sortLP

Tak zmodyfikowany kod tworzy część posortowaną po prawej strony tablicy, i z tego też powodu zaczynamy sortowanie od przedostatniego elementu (ostatni element traktowany jest jako liczba posortowana). Zmienia się również strona, w którą przesuwamy liczby, aby zrobić miejsce. Ponieważ liczby przesuwane są w stronę części nieposortowanej, algorytm ten przesuwa liczby w lewą stronę. Pomimo tych zmian algorytm ten jest lustrzanym odbiciem klasycznego wariantu, w wyniku czego czas do posortowania, jak i ilość porównań i przesunięć powinna być podobna lub taka sama.

Posiadając obie wersje sortowania jesteśmy w stanie wykonać ostatnią możliwą zmianę, czyli zmienić stronę, z której przeszukujemy część posortowaną. W ten sposób powstają algorytmy sortPP i sortLL

```
Dane:
n – ilość elementów w tablicy
tablica[0,..,n-1]- tablica do posortowania

Dla i od n-2 do 0
        x ← tablica[i]
        wstaw ← n-1

        dopóki wstaw > i oraz x < tablica[wstaw]
                wstaw ← wstaw-1

        dla j od i do wstaw-1
                tablica[j] ← tablica[j+1]

        tablica[wstaw] ← x
```

Kod 3. Pseudokod sortPP, zmodyfikowanego algorytmu sortPL

Widoczną różnicą algorytmu od sortPL jest pętla while, która przechodząc po elementach liczb posortowanych zaczynając od prawej strony (ostatni element tablicy) szuka, na którą pozycję należy wstawić. W wyniku zmian osobno sprawdzamy, na której pozycji sortowana liczba powinna się znaleźć oraz osobno wykonujemy przesuwanie liczb.

Analogicznie do powyższej zmiany możemy zmienić sortLP, aby przeszukiwał on część posortowaną z prawej strony, tworząc w ten sposób sortLL.

```
Dane:
n – ilość elementów w tablicy
tablica[0,..,n-1]- tablica do posortowania

dla i od 1 do n-1
        x ← tablica[i]
        wstaw ← 0

        dopóki wstaw < i oraz tablica[wstaw] ≤ x
                        wstaw ← wstaw + 1

        dla j od i do wstaw + 1 [krok: j--]
                        tablica[j] ← tablica[j-1]

        tablica[wstaw] ← x
```

Kod 4. Pseudokod sortLL, zmodyfikowanego algorytmu sortLP

Różnica między algorytmami jest analogiczna do poprzedniej zmiany. Ponieważ przeszukujemy posortowane liczby od lewej strony, dodatkową pętlą szukamy, na której pozycji sortowana liczba powinna się znaleźć, a następnie przesuwamy wszystkie elementy od miejsca wstawienia do końca sekcji posortowanej w prawo.

Wszystkie algorytmy wykorzystują liniowe przeszukiwanie w celu znalezienia pozycji, na której powinna znaleźć się sortowana liczba. Ostatnią zmianą, jaką możemy wykonać jest zmiana sposobu wyszukiwania na wyszukiwanie binarne. Zmianę to zaimplementujemy do algorytmu sortLP, a z powodu na swoją odmienność od reszty nazwana zostanie binSort.

```
Dane:
n – ilość elementów w tablicy
tablica[0,..,n-1]- tablica do posortowania

funkcja wyszukiwanieBinarne(tablica, x, lewy, prawy)

        dopóki  lewy ≤ prawy
                środek ← (lewy + prawy) / 2
                jeżeli tablica[środek] < x
                        lewy ← środek + 1
                w przeciwnym wypadku
                        prawy ← środek -1
        zwróć lewy

funkcja binSort
        dla i od 1 do n – 1
                x ← tablica[i]
                wstaw ← wyszukiwanieBinarne(tablica,x,0,i-1)
        dla j od i do wstaw+1 [krok: j--]
                tablica[j] ← tablica[j – 1]
        tablica[wstaw] ← x
```

Kod 5. Pseudokod binSort, zmodyfikowanego algorytmu sortLP

Algorytm ten wykorzystuje dodatkową funkcję, która wykonuje wyszukiwanie binarne w części posortowanej tablicy. Z tego też powodu parametrami podawanymi do funkcji jako lewa i prawa strona tablicy jest 0 oraz i-1, ponieważ są to granice sekcji posortowanej.

IV. Plan eksperymentów

W celu zbadania algorytmów wykonane zostały 2 niezależne eksperymenty.

Celem pierwszego testu jest zmierzenie czasów potrzebnych do posortowania rosnąco tablicy o określonym rozmiarze. W tym celu utworzono funkcję testującą i mierzącą czasy. Badanie rozpoczęto tablicą zawierającą losowe wartości z przedziału <1;50 000> o rozmiarze 10 000. Następnie zwiększano rozmiar tablicy o 10 000 i wykonywano ponownie pomiary czasów sortowań. Badanie zakończono na rozmiarze tablicy 100 000. W celu uśrednienia wyników dla każdego rozmiaru testy wykonywano 100 razy, za każdym razem generując losowo nową zawartość tablicy.

Drugim badaniem (niezależnym od poprzedniego) było badanie ilości porównań i przesunięć potrzebnych do posortowania tablicy zawierającej wartości posortowane malejąco, rosnąco oraz zawierające wartości losowe. Badanie to wykonano na dwóch tablicach o rozmiarach 1000 oraz 10 000. Wyniki tablicy z losowymi wartościami uśredniono wynikami 100 pomiarów, każdy na nowej zawartości tablicy.

V. Warunki badawcze

Badania zostały wykonane na komputerze o następujących podzespołach, następującej wersji systemu operacyjnego. Podano poniżej również wersję wykorzystanego kompilatora C++.

Procesor	AMD Ryzen 5 3600
Karta graficzna	NVIDIA GeForce GTX 1660 SUPER (6GB)
RAM	Crucial 16GB (2x8GB) 3600MHz CL16 Ballistix
Płyta główna	MSI B450-A PRO MAX (MS-7B86)
Windows	10 OS Build 19045.5753
Kompilator	g++ Wersja: (Rev3, Built by MSYS2 project) 13.2.0

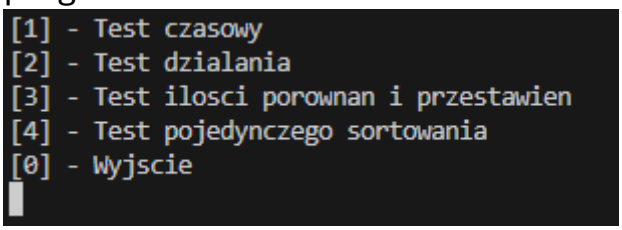
VI. Opis programu

Uwaga: Program tworzy i modyfikuje pliki tekstowe o nazwach „sortowania.txt” oraz „wyniki.txt”. Przed uruchomieniem programu zaleca się sprawdzenie, czy pliki takie nie istnieją w folderze w którym umieszczony jest program, oraz czy nie ma na tych plikach danych. Uruchomienie programu usunie wszystkie informacje znajdujące się w tych plikach.

Program kompilowany był przy pomocy polecania konsoli

g++ .\main.cpp -o .\main.exe -g

Po uruchomieniu programu w konsoli wyświetli się lista możliwych trybów programu.



Wyjaśnienie działania poszczególnych programów:

1. Tryb czasowy: tryb w którym wykonane zostało pierwsze badanie. Po wybraniu programu użytkownik poproszony będzie o wprowadzenie danych:
   1.1 Dolny zakres losowania
   1.2 Górny zakres losowania
   1.3 Początkowy rozmiar tablicy
   1.4 Rozmiar powiększenia tablicy (wielkość o którą początkowy rozmiar tablicy będzie zwiększany z każdą próbą)
   1.5 Maksymalny rozmiar tablicy

   Po podaniu powyższych parametrów program zapyta się o potwierdzenie ustawień. Zatwierdzenie ich przeprowadzi test czasu. Opis działania został przedstawiony w sekcji VII. Wyniki badania zapisane zostaną w pliku „wyniki.txt”. W formie tekstowej zapisane są w formacie Rozmiar tablicy;Czas sortLP;Czas sortPL;Czas sortLL;Czas sortPP;Czas binSort. Zaleca się wprowadzenie wyników do arkuszu kalkulacyjnego używając jako separatora średnika.
2. Test działania: tryb wypisujący do pliku „sortowania.txt” wszystkie przesunięcia jakie wykonuje program w trakcie sortowania, oraz zliczoną ilość porównań i przesunięć. Wyświetlone one zostaną w formacie:
   -Tablica nieposortowana

-Nazwa użytego sortowania
-Lista przesunięć jakie zostały wykonane (linie kreskowane oddzielają każdą iterację)
-Ilość porównań i przesunięć
-Tablica posortowana
Wyniki powtórzone są dla każdego z sortowań.
Wymagane dane do wprowadzenia to:

2.1 Rozmiar tablicy
2.2 Zakres dolny losowania
2.3 Zakres górny losowania

Program ten pozwala sprawdzić poprawność działania każdego z wykorzystanych algorytmów. Z powodu dużej ilości iteracji jakie wykonuje każde sortowanie zalecane jest używanie małego rozmiaru tablicy.

3. Test ilości porównań i przesunięć: tryb wypisujący do pliku „sortowania.txt" ilość porównań i przesunięć jakie wykonuje każdy algorytm podczas sortowania. Opis działania został przedstawiony w sekcji VII

   Wymagane dane do wprowadzenia to:

   3.1 Rozmiar tablicy
   3.2 Zakres dolny losowania
   3.3 Zakres górny losowania

4. Test pojedynczego sortowania: wykonuje on działanie identyczne do drugiego testu, jednak wykonywany jest on dla pojedynczego, wybranego sortowania.

VII. Opis implementacji

W sekcji tej opisane zostaną metody wykonywania wszystkich testów, które można wybrać w programie. Opisy oraz wprowadzane do testów dane przedstawione zostały w sekcji VII.

1. Test Czasowy: Test wykonywany jest w pętli zaczynającej się od rozmiaru początkowego tablicy aż do rozmiaru maksymalnego, zwiększając się o podaną wielkość. Iterator tej pętli służy jako wskazania rozmiaru tablicy podczas generowania tablic. W zagnieżdżonej pętli iterującej od zera do ilości prób-1 wykonywane są następujące kroki
   - Wypełnienie tablicy źródłowej
   - Skopiowanie tablicy źródłowej (nowa tablica będzie sortowana)
   -Wykonanie sortowania odpowiednim sortowaniem oraz zmierzenie czasu
   - Zapisanie wyniku
   - Skopiowanie tablicy źródłowej

   Cykl powtarzany jest dla każdego sortowania. Po zakończeniu wewnętrznej pętli wyniki czasowe podzielone przez ilość prób zapisywane są w celu późniejszego zapisu do pliku.
2. Test Działania

   Funkcja testu tworzy 2 tablice, generuje jedną losowymi wartościami oraz otwiera plik do zapisu. Następnie w pętli wykonuje następujące czynności
   - Zerowanie ilości porównań i przesunięć
   - Uruchomienie funkcji sortującej z dodatkowymi zliczaniem porównań i przesunięć oraz wypisywanie do pliku

   Każde następne wykonanie pętli zmienia rodzaj sortowania w celu zbadania wszystkich implementacji.
3. Test Ilości porównań i przesunięć

   Test podzielony jest na 3 części. Części wykonują następujące czynności:

   3.1 Sortowanie tablicy posortowanej rosnąco:
   - wypełnienie tablicy źródłowej
   -skopiowanie tablicy
   -wypisanie do pliku nazwy sortowania

-wyzerowanie liczników
-wywołanie funkcji sortowania
-wypisanie do pliku ilości porównań i przesunięć
Z pominięciem pierwszej czynności wszystko wykonywane jest dla każdego

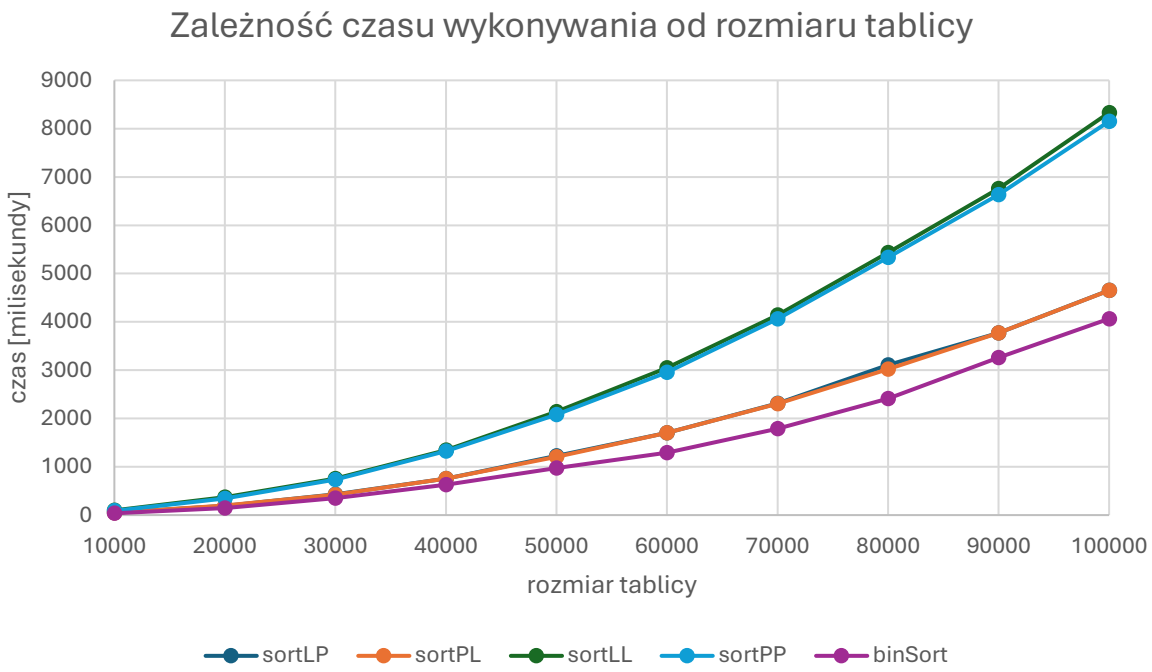
3.2 Sortowanie na tablicy posortowanej malejąco
-posortowanie tablicy źródłowej malejąco
-skopiowanie tablicy
-wypisanie do pliku nazwy sortowania
-wyzerowanie liczników
-wywołanie funkcji sortowania
-wypisanie do pliku ilości porównań i przesunięć
Z pominięciem pierwszej czynności wszystko wykonywane jest dla każdego algorytmu

3.3 Sortowanie na tablicy posortowanej malejąco
-posortowanie tablicy źródłowej malejąco
-skopiowanie tablicy
-wypisanie do pliku nazwy sortowania
-wyzerowanie liczników
-wywołanie funkcji sortowania
-wypisanie do pliku ilości porównań i przesunięć
Z pominięciem pierwszej czynności wszystko wykonywane jest dla każdego algorytmu

IX. Tabele i wykresy

Po wykonaniu badań opisanych w sekcji IV uzyskano wyniki, które w celu wizualizacji oraz łatwiejszej interpretacji umieszczono w tabelach oraz wykresach. Interpretacja wyników znajduje się w sekcji IX.

Badanie 1: Test czasowy

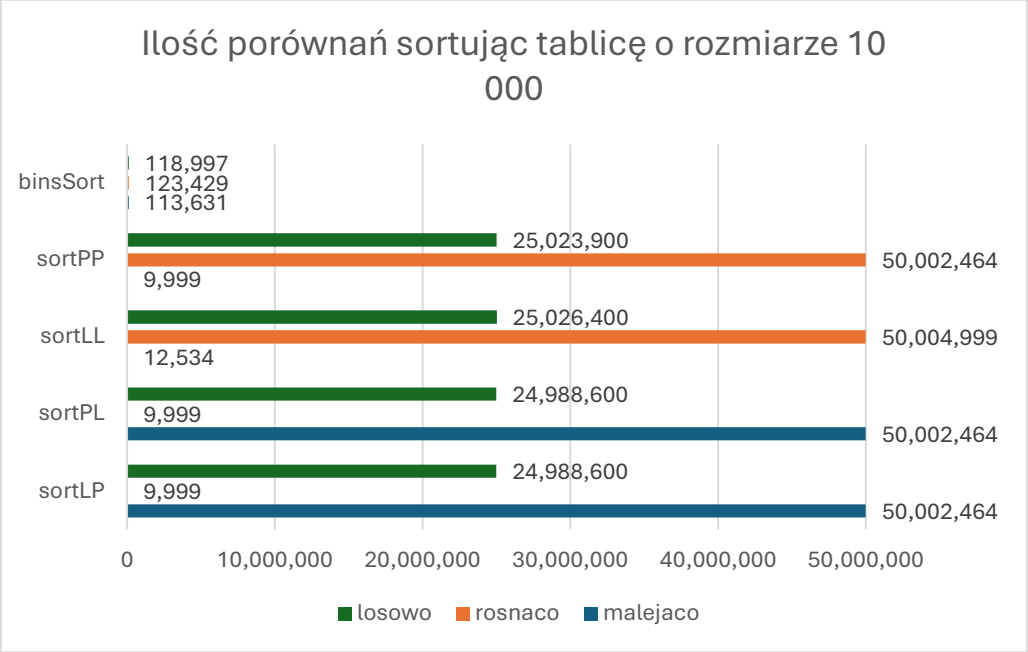


Wykres 1. Wyniki testu czasu (w milisekundach) potrzebnego do posortowania tabeli zawierającej losowo wygenerowaną zawartość

Rozmiar tablicy	Czas sortLP	Czas sortPL	Czas sortLL	Czas sortPP	Czas binSort
10000	49.6544	47.2423	98.0556	89.0541	37.6362
20000	194.077	195.139	367.311	344.245	147.914
30000	429.915	424.085	754.366	734.944	352.029
40000	755.698	753.304	1344.33	1327.89	626.767
50000	1222.78	1203.52	2139.28	2081.8	974.993
60000	1705.88	1700.34	3046.06	2958.29	1292.43
70000	2315.42	2307.77	4143.54	4061.76	1790.98
80000	3105.6	3021.02	5436.59	5336.91	2410
90000	3769.92	3767.29	6761.55	6630.86	3261.01
100000	4651.73	4649.65	8331.66	8152.4	4064.9

Tabela 1. Wyniki testu czasu (w milisekundach) potrzebnego do posortowania tabeli zawierającej losowo wygenerowaną zawartość

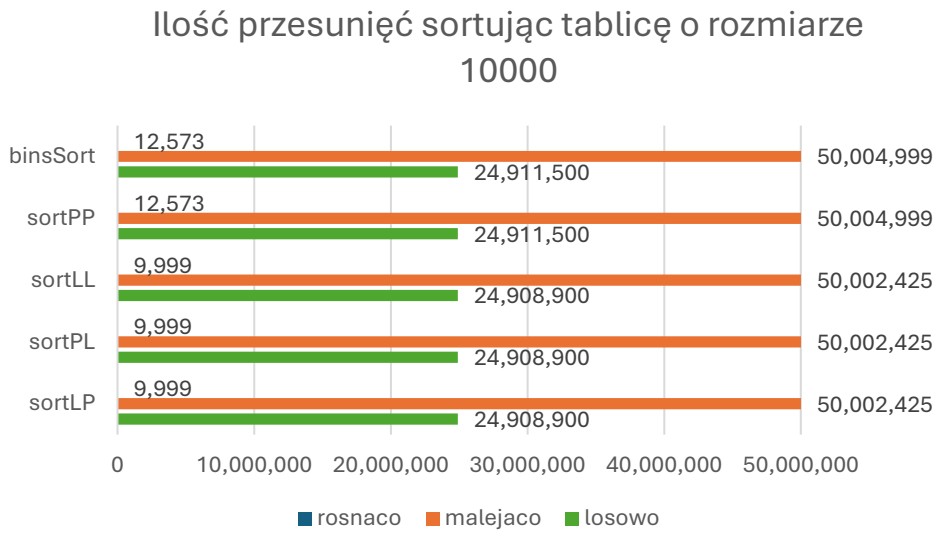
Badanie 2. Pomiar ilości przesunięć i porównań



Wykres 2. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 10 000

Ilość porównań rozmiar 10000			
	malejaco	rosnaco	losowo
sortLP	50002464	9999	24988600
sortPL	50002464	9999	24988600
sortLL	12534	50004999	25026400
sortPP	9999	50002464	25023900
binsSort	113631	123429	118997

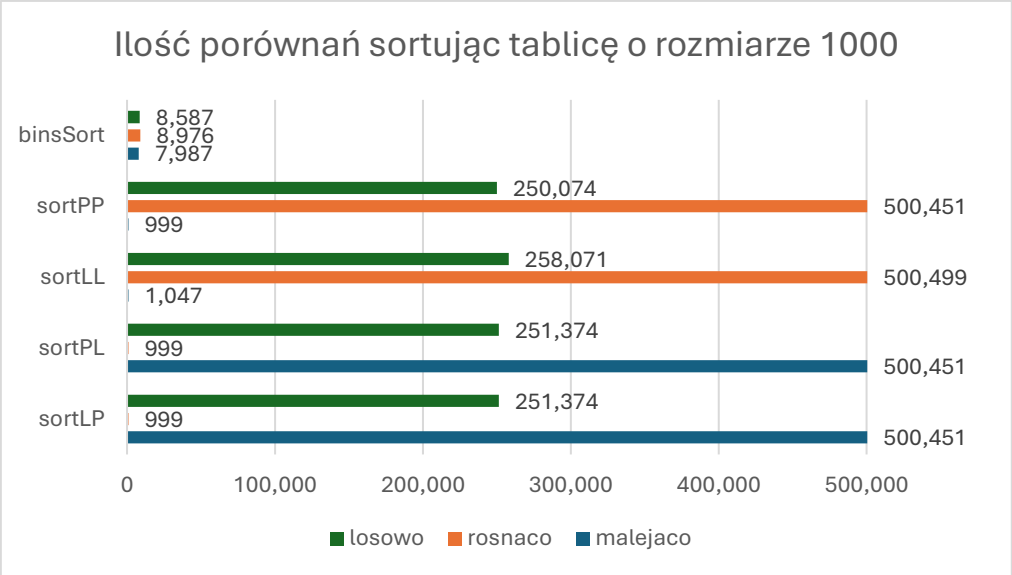
Tabela 2. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 10 000



Wykres 3. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 10 000

Ilość przesunięć rozmiar 10000			
	malejaco	rosnaco	losowo
sortLP	50002464	9999	24988600
sortPL	50002464	9999	24988600
sortLL	50002464	9999	24988600
sortPP	50004999	12534	24991100
binsSort	50004999	12534	24991100

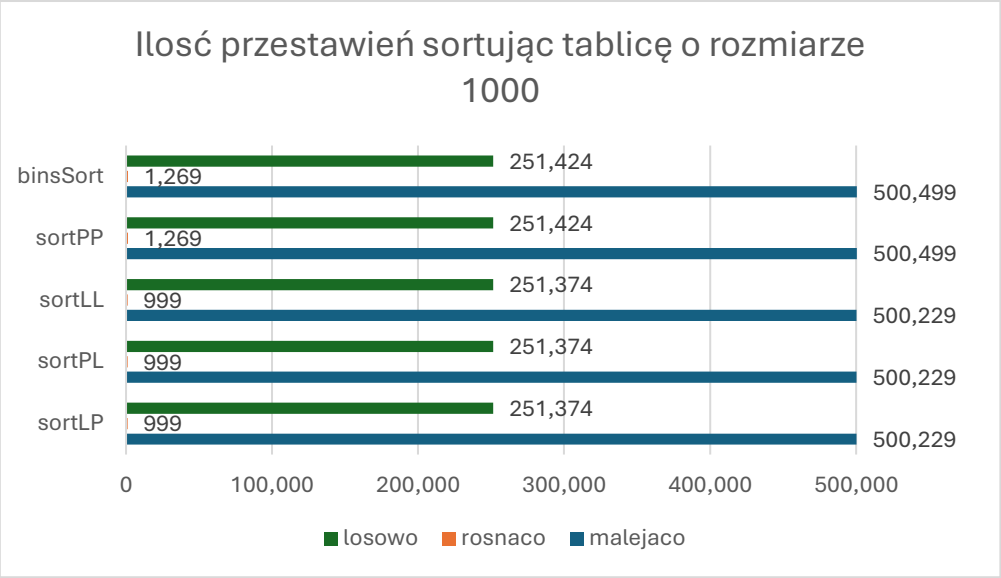
Tabela 3. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 10 000



Wykres 4. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 1000

Ilość porównań rozmiar 1000			
	malejaco	rosnaco	losowo
sortLP	500451	999	251374
sortPL	500451	999	251374
sortLL	1047	500499	258071
sortPP	999	500451	250074
binsSort	7987	8976	8587

Tabela 4. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 1000



Wykres 5. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 1000

Ilość przesunięć rozmiar 1000			
	malejaco	rosnaco	losowo
sortLP	500229	999	251374
sortPL	500229	999	251374
sortLL	500229	999	251374
sortPP	500499	1269	251424
binsSort	500499	1269	251424

Tabela 5. Wynik testu potrzebnej ilości porównań i przesunięć dla rozmiaru tablicy 1000

X. Dyskusja, interpretacja i wnioski

Zgodnie z teorią oczekiwany czas średni wykonywania programu powinien rosnąć kwadratowo wraz ze zwiększaniem się ilości elementów tablicy. Pośród wartości oczekiwanych przesunięć powinniśmy oczekiwać wartości takich jak n-1, $n^2/2$, $n^2/4$.

Analizując uzyskane wyniki badań czasu możemy zauważyć, że wraz ze zwiększeniem się ilości elementów czas potrzebny do posortowania tablicy rośnie kwadratowo. Na wykresie możemy zauważyć widoczny podział algorytmów na 3 sekcje. Pierwsza sekcja składająca się z sortLL i sort PP potrzebują najwięcej czasu, w wyniku czego są najmniej efektywne. Wynikać to może z faktu, że algorytmy zawierają dodatkową pętlę, w wyniku czego czas wykonania znacząco się zwiększa. Drugą sekcją są sortowania LP i PL, są to algorytmy zawierające książkowe metody sortowania przez wstawianie. Ich wykres rośnie łagodniej od poprzedniej sekcji, jednak nadal rośnie ono kwadratowo. Trzecią sekcją jest binSort, który rośnie najmniej z wszystkich sortowań. Wynika to z faktu pominięcia wyszukiwania liniowego na rzecz

sortowania binarnego, który znajduje miejsce wstawienia liczby w mniejszej ilości sprawdzeń. Różnica przeszukiwań staje się widoczna wraz ze zwiększaniem się rozmiaru tablicy.

Analizując wyniki drugiego badania możemy zobaczyć liczbę porównań i przestawień algorytmów. Pierwszą rzeczą jaka rzuca się w oczy to fakt różnicy wielkości niektórych wyników, powodując niewidoczność niektórych słupków na wykresach. Rodzaje tablic na jakich wykonywano sortowania są na wykresach (tablica losowa, tablica posortowana rosnąco/malejąco) oznaczone tymi samymi kolorami w celu łatwiejszej interpretacji. Pierwszy wniosek jaki możemy zauważyć to ilość przesunięć pozostaje niezmienna. Dla warunków optymistycznych wynosi on okolice n-1, dla pesymistycznych jest to n2/2, i dla losowych n2/4. Zastanawiający może być wynik n-1 dla tablicy już posortowanej rosnąco, ponieważ teoretycznie nie wykonujemy żadnego przesunięcia. Wynika to ze sposobu implementacji, ponieważ ostatnim krokiem algorytmu jest wstawienie liczby sortowanej, która w tym przypadku wstawiana jest w to samo miejsce gdzie się znajduje. Analizując wyniki porównań również widzimy okolice wartości n-1, n2/2, n2/4, jednak różne metody uzyskują różne wyniki. Dla tablicy malejącej sortowania LL i PP wykonują n-1 porównań, zaś sortowania LP i PL wykonują już ich n2/2. Jest to logiczne ponieważ przeszukując od lewej strony tablicy każda następna liczba będzie mniejsza, i będzie znajdować się na 1 pozycji części posortowanej, do której dotrzemy bardzo szybko z powodu przeszukiwania jej od lewej strony, czyli tam gdzie chcemy się znaleźć. Analogicznie występuje w sortowaniu PP. Przeciwnie jest w sortowaniach LP i PL, gdzie musimy przeszukać całą część posortowaną by dojść do tego samego miejsca w tablicy. Tak samo dzieje się w tablicy posortowanej już rosnąco, gdzie sortowania przeciwne do kierunku przeszukiwania wykonują liniową ilość operacji porównując liczbę sortowaną z tylko jedną liczbą w sekcji posortowanej. Końcowo widzimy, że dla wartości losowych wszystkie sortowania wykonują n2/4. Jedynym wyjątkiem od powyższych wniosków jest sortowanie używające przeszukiwania binarnego. Z powodu używania innych zasad odnajdywania pozycji w tablicy wykonuje on inną ilość porównań od reszty, jednak ilość przesunięć nadal pozostaje niezmienna. Możemy również zastanawiać się na opłacalnością zastosowania wyszukiwania binarnego nad wyszukiwaniem liniowym. Średnia różnica pomiarów między binSort a sortLP wynosi około 19.88%, co może wydawać się znaczącą różnicą, jednak patrząc na fakt, że oba pomiary i tak rosną kwadratowo, a ich rozbieżność zaczyna być zauważalna dopiero w okolicach rozmiarów 40 000 i 50 000

napotykamy nadal ten sam problem sortowania przez wstawianie, czyli jego słabe działanie dla dużych rozmiarów tablic.

Wnioskując z wyników obu badań możemy dojść do wniosku ze algorytmy zostały zaimplementowane poprawnie. Widać również główny problem sortowania przez wstawianie, czyli jego powolne działanie przy dużych rozmiarach tablic. Możemy zauważyć różnice jakie wynikają z implementacji algorytmu. Możemy wysnuć wniosek, że algorytm sortowania przez wstawianie działa najlepiej, kiedy sekcję posortowaną sprawdzamy w stronę przeciwną do kierunku sortowania tablicy.

XI. Podsumowanie

Algorytm sortowania przez wstawianie jest algorytmem stosunkowo łatwym do implementacji, niewymagającym dużej przestrzeni w pamięci do wykonywania swoich operacji. Pomimo jego korzyści widoczne są jego wyraźne wady, a w szczególności jego powolność w działaniu. Mimo to możemy wykorzystywać ten algorytm przy małych rozmiarach tablic, gdzie wykonuje on swoją pracę najlepiej.

Pomimo różnych wersji tego samego sortowania należy uważać na sposób implementacji, ponieważ możemy sztucznie pogorszyć działanie naszego programu.

Podczas wybierania algorytmu sortowania należy zawsze dobierać algorytm, który będzie w stanie działać najlepiej zgodnie z zapotrzebowaniem programu (limity pamięci, limity czasu).

XII. Literatura

Selection of best sorting algorithm. Aditya Dev Mishra & Deepak Garg

Review on Sorting Algorithms A Comparative Study. Khalid Suleiman Al-Kharabsheh & Ibrahim Mahmoud AlTurani & Abdallah Mahmoud Ibrahim AlTurani & Nabeel Imhammed Zanoon