Sprawozdanie z projektu

Student: Piotr Najda, w65550

Prowadzący: prof. dr hab. inż. Władysław Homenda

Kod przedmiotu: 2IID/2021-GW01

Algorytmy i struktury danych

WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
z siedzibą w Rzeszowie

Spis treści

1. Analiza empiryczna złożoności algorytmów sortowania i wyszukiwania

1.1. Definicje problemów

- Problem wyszukiwania - czy dany element należy do danego zbioru elementów?
- Problem sortowania - dany jest ciąg elementów, które potrafimy porównać w sensie relacji większości (mniejszości). Ustawić elementy tego ciągu w kolejności niemalejącej (posortowane).

Źródło definicji: [9]

1.2. Opis problemów

Dla algorytmów wyszukiwania liniowego i binarnego, sortowania bąbelkowego, przez wstawianie i scalanie: jaka jest liczba wykonań operacji dominujących dla danego algorytmu wyszukiwania/sortowania, dla danych wejściowych:

- posortowanych,
- posortowanych odwrotnie,
- w kolejności losowej,

przy rozmiarach danych wejściowych:

- n, 2n, 3n, ...

Przedstawię opisy realizacji części źródeł programu odpowiedzialnych za:

- analizowane algorytmy
- empiryczną analizę złożoności algorytmów

Zaprezentuję także wnioski i wykresy zależności liczby wykonań operacji dominujących od rozmiaru zadania 'n' i typu danych wejściowych, w porównaniu z wykresem funkcji, która określa rząd złożoności danego algorytmu.

1.3. Instrukcja uruchomienia programu i opis plików źródłowych

```
- Folder "AISD Projekt"
    - Folder "Debug"
        - "AISD.exe" -- program wykonywalny (wersja Release)
    - Folder "AISD" - pliki źródłowe ".cpp" i nagłówkowe ".h" projektu
        - "main.cpp" -- główny plik programu
        - "sortingAlgorithms" -- algorytmy sortujace
        - "searchAlgorithms" -- algorytmy wyszukiwania
        - "GenerateArray" -- tworzy i kopiuje tablice o różnych typach danych wejściowych
        - "GraphPoint" -- tworzy punkty wykresu z pojedynczego wykonania danego algorytmu
    - "AISD.sln" -- otworzy projekt w Visual Studio 2019
        - uruchomienie: Ctrl + F5
    - Folder "wynikiProgramuAISD"
        - Program "utworzWykresyZWynikow.exe" -- generuje interaktywne wykresy z pliku
        ↪  "wynikiAnalizy.csv". UWAGA: program otworzy domyślną przeglądarkę z wygenerowanymi
        ↪  wykresami.
        - "utworzWykresyZWynikow.py" -- plik źródłowy programu powyżej (do wglądu)
        - "wynikiAnalizy.csv" - przykładowy wynik uruchomienia programu "AISD.exe" lub
        ↪  zbudowaniu i uruchomieniu projektu "AISD.sln", z wyborem w menu domyślnej ilości
        ↪  liczby tablic równej 100.
    - Folder "sprawozdanie"
        - Sprawozdanie w PDF
        - Pliki źródłowe sprawozdania (do wglądu)
```

1.4. Wyszukiwanie liniowe

1.4.1. Opis realizacji algorytmu

```
1  int linearSearch(int* arr, int n, int find) {
2      for (int i = 0; i < n; i++) {
3          // Miejsce dodania operacji dominujacej
4          if (arr[i] == find) return i;
5      }
6      return -1;
7  }
```

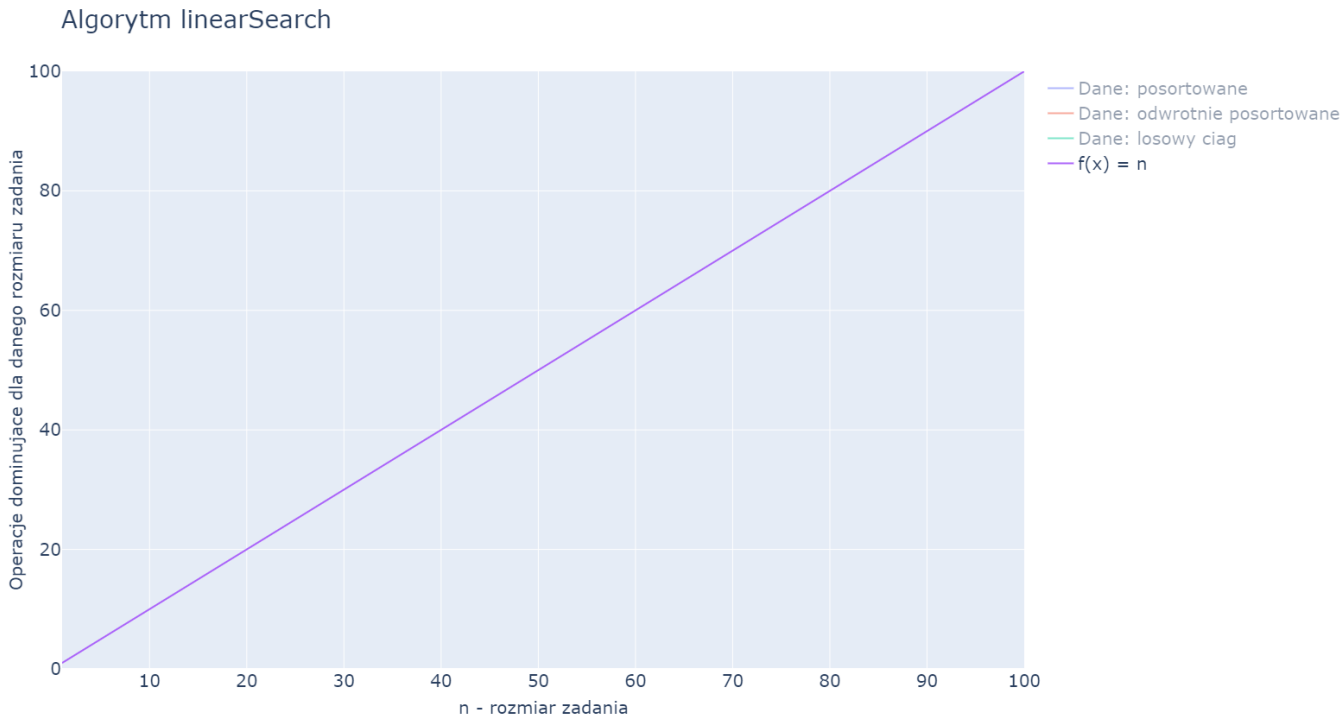
Problem: wyszukiwanie
Algorytm: wyszukiwanie liniowe
Dane: ciąg (int* arr) n (int n) liczb posortowanych/posortowanych odwrotnie/ustawionych w kolejności losowej, liczba do wyszukania (int find)
Metoda i opis:
Linijki 2 - 4: Dla kolejnych wartości indeksu "i = 0, 1, 2... n - 1": jeśli liczba do wyszukania kryje się w komórce tablicy "arr[i]", zwróć indeks tej komórki równy "i"
Linijka 5: Wyjście z pętli w linijkach 2 - 4 jest równoważne z brakiem obecności wyszukiwanej liczby w tablicy, zwróć więc wartość "-1", która to zasygnalizuje.
Wynik:
Otrzymujemy indeks tablicy pod którym kryje się wyszukiwany element lub w przypadku jego nieobecności, liczbę "-1"
Źródło wykorzystane do opisu: [9]

1.4.2. Opis realizacji empirycznej analizy złożoności algorytmu

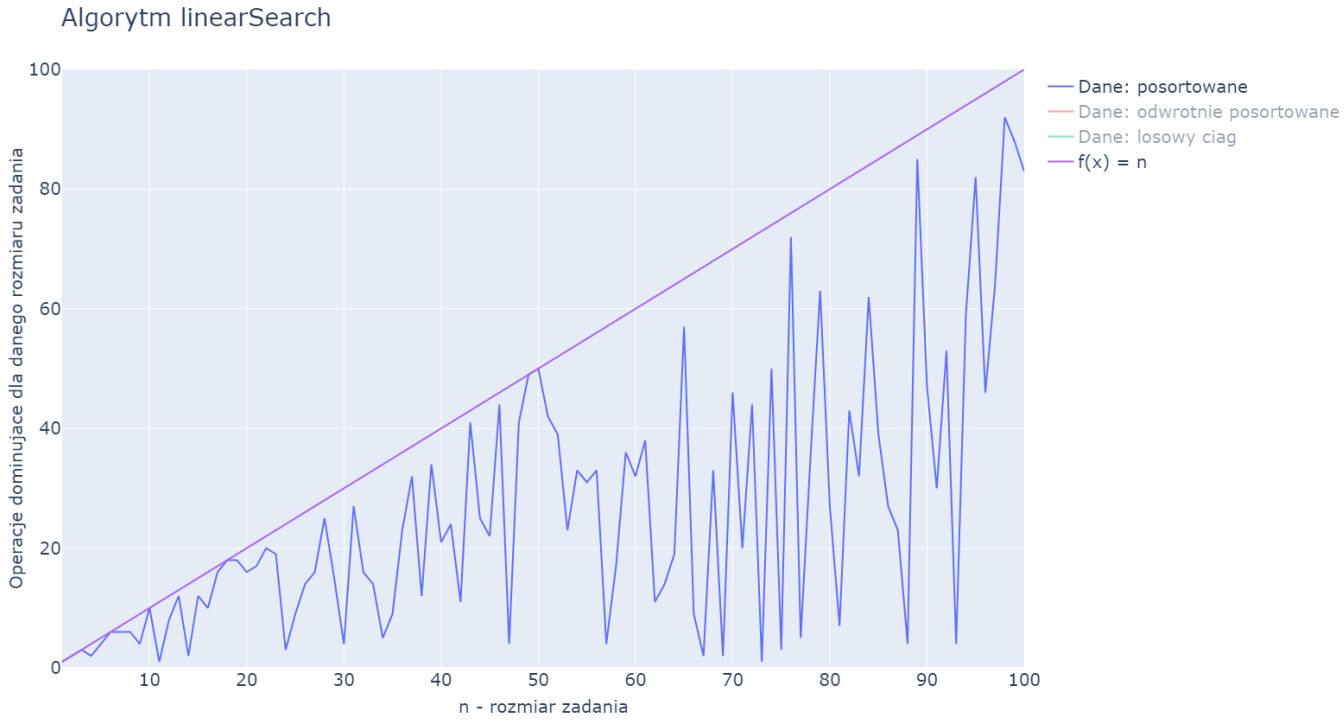
Jako punkt zliczania operacji dominującej wybrałem sprawdzenie warunku "i < n" pętli "for". Jest to pętla przechodząca przez tablice, aż nie znajdzie się wyszukiwana liczba lub nie skończy się tablica, a więc jej wykonanie następuje aż do samego końca działania algorytmu.

1.4.3. Prezentacja wyników

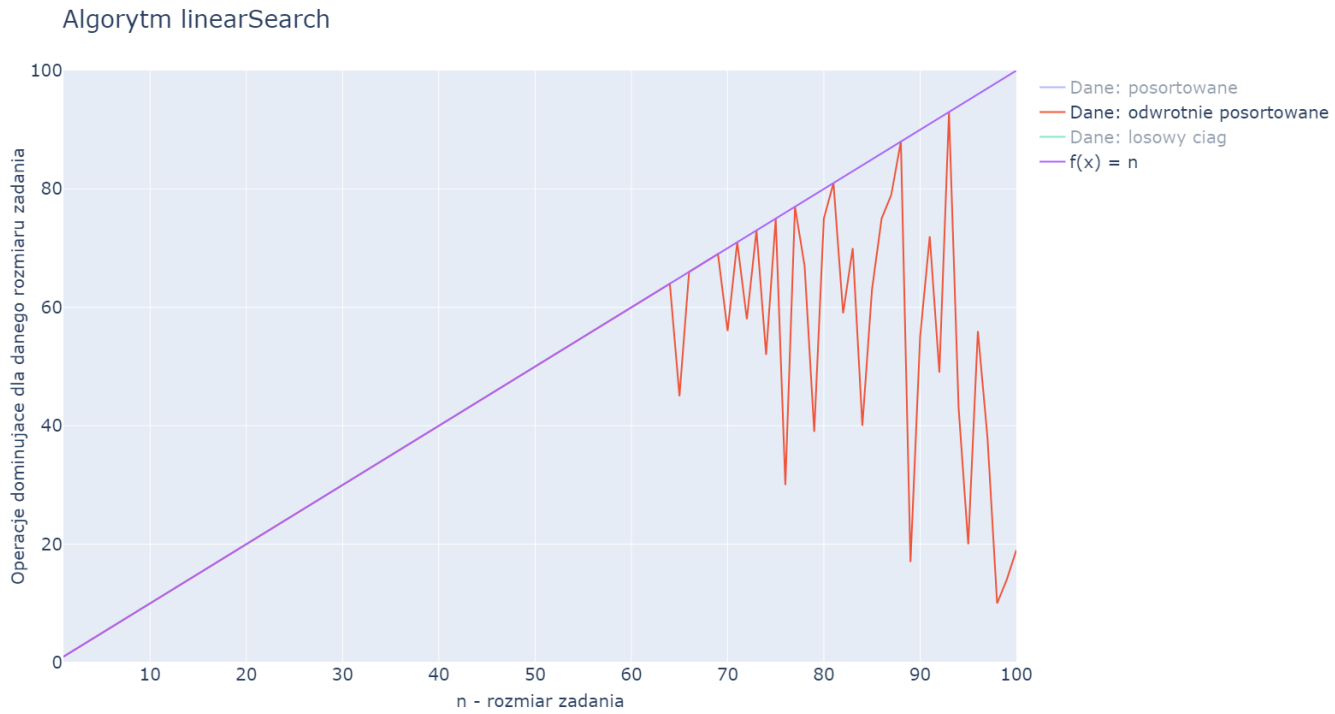
Zaprezentuje teraz wykresy zależności operacji dominujących od rozmiaru zadania dla algorytmu wyszukiwania liniowego, przy różnych typach danych: posortowanych, posortowanych odwrotnie, ciągach losowych, o rozmiarach od stałego "n = 1", do "100n". Proces powtórzę dla reszty algorytmów.
Liczby, które wyszukuję dla algorytmu wyszukiwania liniowego i binarnego są takie same dla danych rozmiarów zadania. Skale osi Y liczby operacji dominujących również są takie same dla tych dwóch algorytmów, dla wygody porównania.



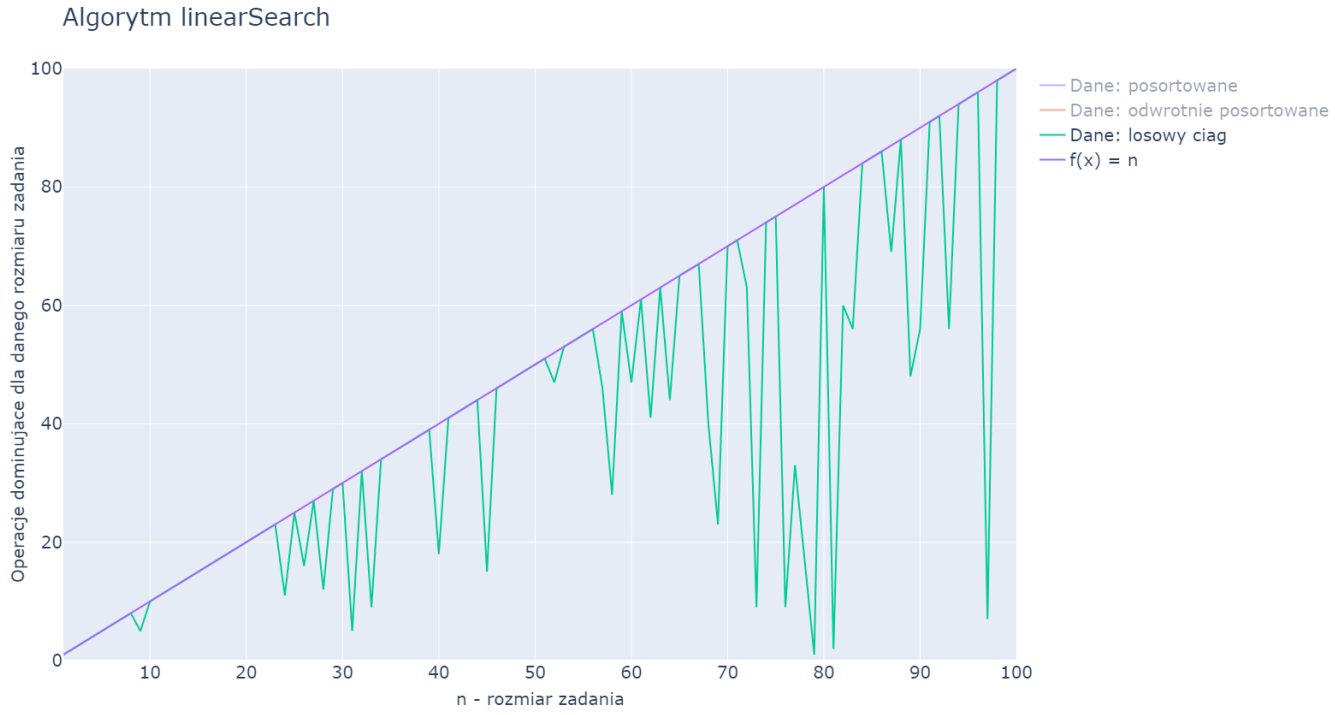
Rysunek 1: Teoretyczna złożoność wyszukiwania liniowego jest rzędu $\theta(n)$



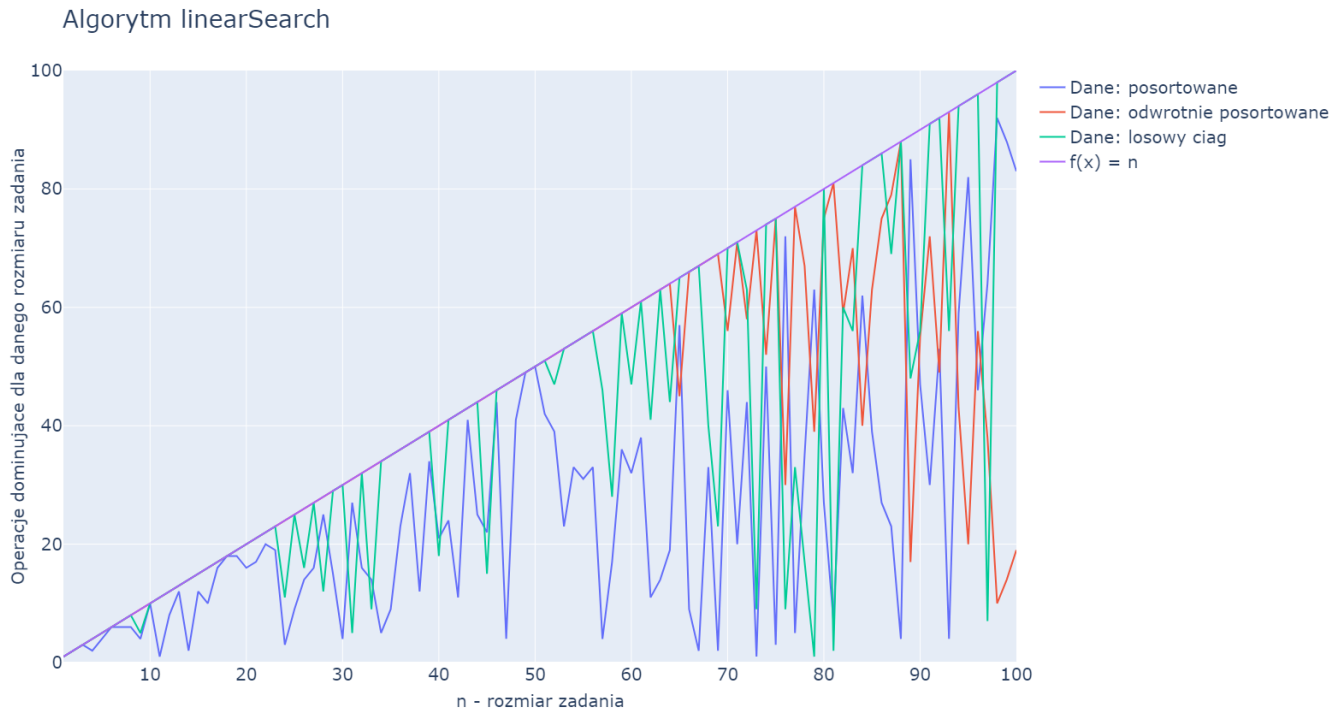
Rysunek 2: Tablice posortowane



Rysunek 3: Tablice posortowane odwrotnie - Dla kilkudziesięciu pierwszych elementów algorytm nie był w stanie znaleźć danej liczby w tablicy (bo nie istniała w niej) lub znajdywała się ona na samym końcu - wtedy liczba operacji dominujących jest rzędu n.



Rysunek 4: Tablice ciągów losowych



Rysunek 5: Zestawienie wszystkich typów danych

1.4.4. Wnioski

Liczba wykonań operacji dominującej w tym algorytmie zależy od długości tablicy i od ustawienia wyszukiwanego elementu w tej tablicy. [9]

W optymistycznym wypadku złożoność jest rzędu $\theta(1)$, co oznacza, że jest niezależna od liczby danych wejściowych. [1] W przypadku wyszukiwania liniowego jest tak, gdy wyszukiwany element znajduje się w pierwszej komórce tablicy.

W pesymistycznym wypadku wyszukiwany element znajduje się na ostatnim miejscu tablicy lub nie ma go w niej. Wtedy musimy przejść przez całą tablicę, więc złożoność jest liniowa / rzędu n, czyli jest proporcjonalna do długości tablicy. [9]

Wykorzystanie algorytmu może być praktyczne w przypadku, gdy tablica posiada niewiele elementów lub gdy chcemy wykonać małą liczbę wyszukań na nieposortowanej tablicy. [2]

1.5. Wyszukiwanie binarne

1.5.1. Opis realizacji algorytmu

```
1 int binarySearch(int* arr, int l, int r, int find){
2     while (l <= r) {
3         int m = (l + r) / 2;
4         if (arr[m] == find) return m;
5         if (arr[m] < find) l = m + 1;
6         else r = m - 1;
7     }
8     return -1;
9 }
```

Źródło algorytmu wyszukiwania binarnego: [3]

Problem: wyszukiwanie

Algorytm: wyszukiwanie binarne

Dane: posortowany ciąg (int* arr) n liczb, liczba do wyszukania (int find)

Metoda i opis:

Algorytm wyszukiwania binarnego działa poprawnie tylko dla posortowanej tablicy, gdyż opiera się on na założeniu, że środkowa wartość tablicy jest jej medianą [4]. W przeciwnym razie nie możemy skutecznie stwierdzić czy wyszukiwana liczba może znajdywać się na prawo lub na lewo.

Algorytm wyszukiwania binarnego jest ułożony za pomocą metody "dziel i rządź", która dzieli wejściowy problem (tablica liczb, którą przeszukujemy) na mniejsze podproblemy/prostsze podzadania (lewa połowa, prawa połowa, lewa połowa lewej połowy, itd.), które rozwiązuje rekurencyjnie (szukamy liczby w środku danej podtablicy, aż ją znajdziemy lub dojdziemy do pustego zakresu wyszukiwania – zadaniem elementarnym jest tutaj sprawdzenie czy w danej podtablicy znajduje się szukana liczba).

Źródła informacji na temat metody "dziel i rządź": [8], [9]

Algorytm zaimplementowałem iteracyjnie, ponieważ jest on nieskomplikowany i przejrzysty również w takiej formie.

"l" i "r" to zakres indeksów w tablicy, między którymi będziemy wyszukiwać daną liczbę.

W linijce 2 pętla "while" sprawdza czy tablica nie jest pusta. Jeśli jest, zwracamy, że liczby nie znaleziono ("-1").

Jeśli jednak nie jest pusta, najpierw sprawdzamy czy element po środku tablicy jest wyszukiwanym elementem, jeśli jest, to go zwracamy. Jeśli nie, następuje zmniejszenie problemu o połowę: z uwagi na to, że dane wejściowe są posortowane i wartość po środku reprezentuje medianę, sprawdzamy teraz prawą stronę tablicy jeśli element po środku jest mniejszy od szukanego, lub lewą w przeciwnym wypadku – powtarzamy cały proces aż do znalezienia elementu lub do momentu gdy nie ma już gdzie szukać.

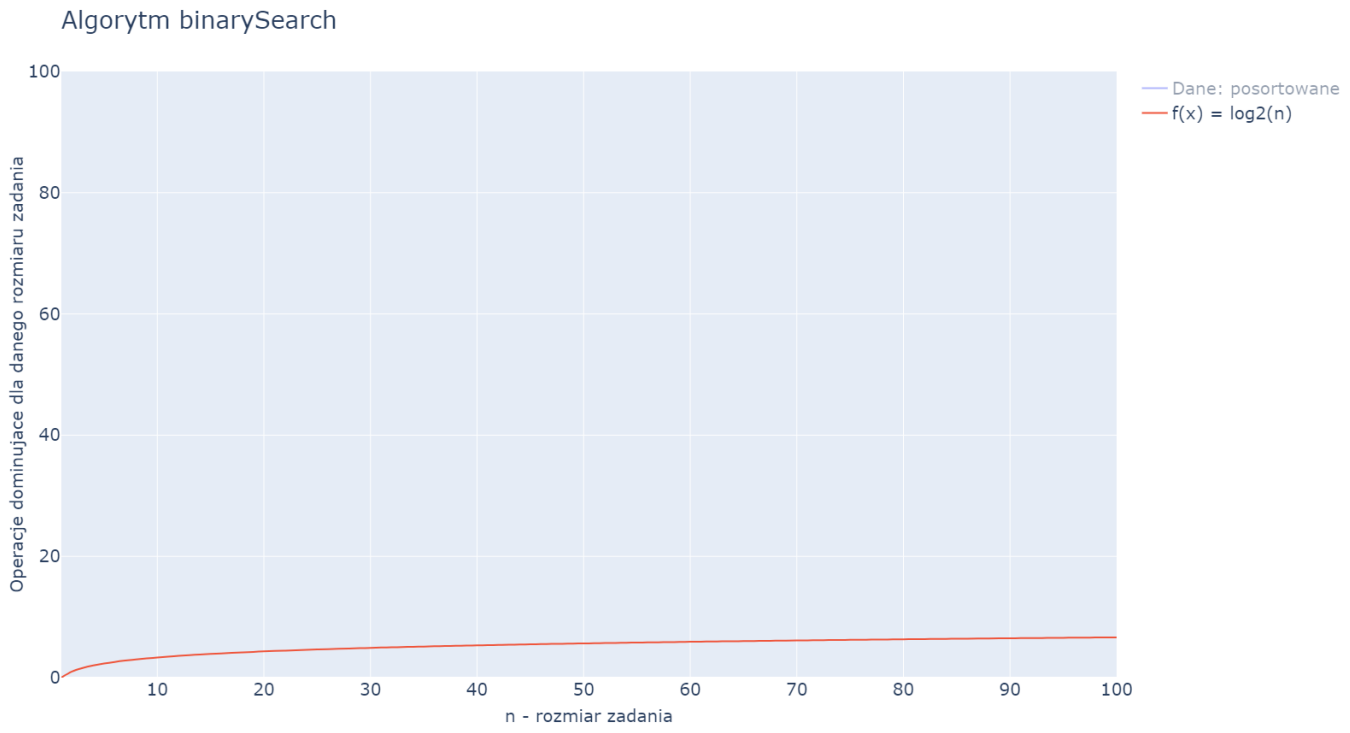
```
Przykładowe wywołanie funkcji w pseudokodzie

arr = [1, 2, 3, 4, 5, 7]
szukana liczba = 6
- binarySearch(tablica, 0, 5, 6)
    - tablica nie jest pusta (l < r)
    - srodek = (0 + 5) / 2 = 2-gi element czyli 3
    - arr[srodek] != szukana liczba
    - arr[srodek] jest mniejszy niz szukana liczba -> bedziemy przeszukiwac prawa polowe
    ↪  tablicy, l = srodek + 1 czyli 3-ci element, "4"
        - l < r
        - m = 4, arr[m] = 5
        - arr[m] != find
        - arr[m] < find -> bedziemy szukac na prawo, l = 5, r nadal jest rowne 5
            - l <= r
            - m = 5, arr[m] = 7
            - arr[m] != find
            - arr[m] > find -> bedziemy przeszukiwac lewa polowe tablicy, r = 4
                -- lewy indeks wiekszy od prawego -> liczby nie znaleziono

Podział na podproblemy:
- Wejściowy: [1, 2, 3, 4, 5, 7] --  arr[m] = 3; 6 > arr[m]
    - Prawa połowa tablicy [4, 5, 7] -- arr[m] = 5; 6 > arr[m]
        - [7]  -- arr[m] = 7; 6 < arr[m],
            - nie ma więcej podtablic do sprawdzenia
```

Wynik:
Otrzymujemy indeks tablicy pod którym kryje się wyszukiwany element lub w przypadku jego nieobecności, liczba "-1"

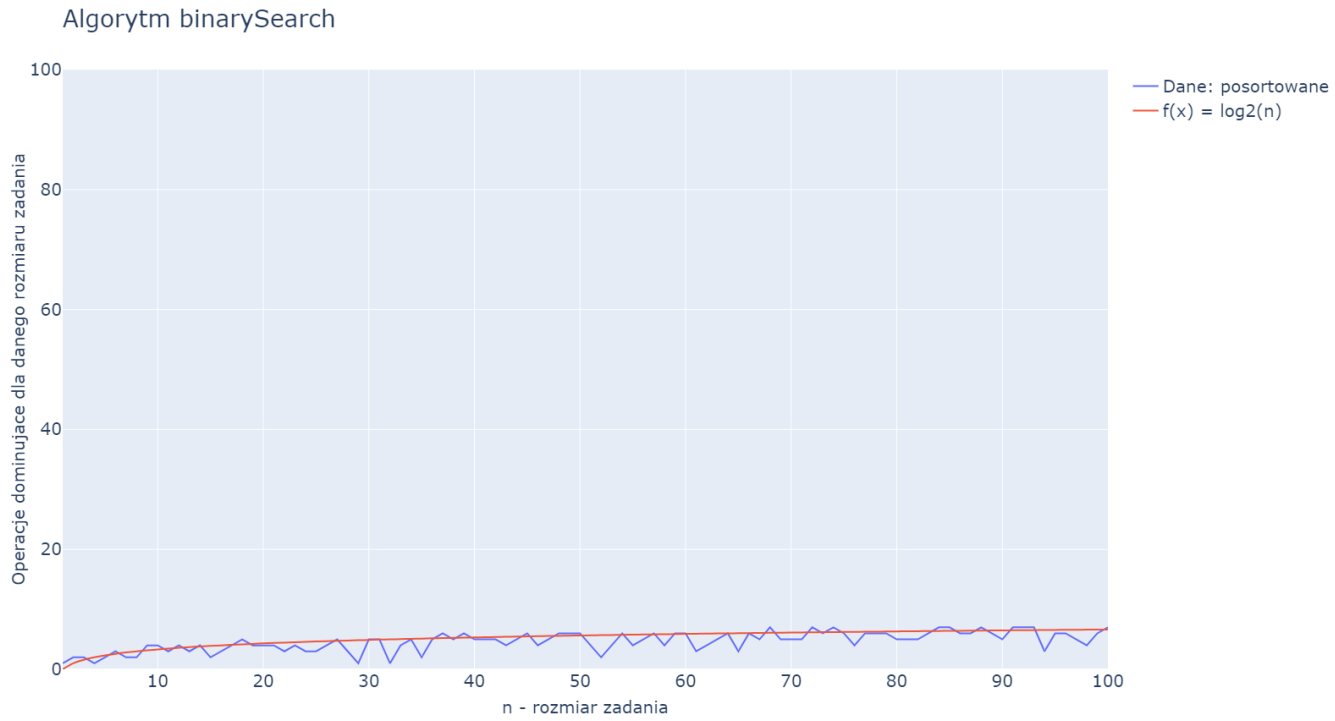
1.5.2. Opis realizacji empirycznej analizy złożoności algorytmu

Jako punkt zliczania operacji dominującej wybrałem sprawdzanie warunku "l <= r", który jest tożsamy z próbą rozwiązania danego podproblemu, a więc podzieleniem poprzedniego podproblemu (lub problemu wejściowego) na pół.

1.5.3. Prezentacja wyników



Rysunek 6: Teoretyczna złożoność wyszukiwania binarnego jest rzędu $\theta(log2(n))$



Rysunek 7: Tablice posortowane

1.5.4. Wnioski

Zachowując tę samą skalę osi X i Y przy wyszukiwaniu liniowym i binarnym, te same typy danych wejściowych, te same dane w tablicach, jak i te same wyszukiwane liczby, widać, że wyszukiwanie binarne jest szybszym rozwiązaniem. Wymaga ono jednak posortowanej tablicy, co może wiązać się z dodatkowym procesowaniem danych wejściowych.

Dokonana przeze mnie empiryczna analiza złożoności wskazuje, że liczba operacji dominujących dla danego rozmiaru zadania oscyluje wokół wykresu funkcji ”log2(n)”.

1.6. Sortowanie bąbelkowe

1.6.1. Opis realizacji algorytmu

```
1  void bubbleSort(int* arr, int n) {
2      for (int i = 0; i < n - 1; i++) {
3          bool swap = false;
4          for (int j = 0; j < n - 1 - i; j++) {
5              // Miejsce dodania operacji dominujacej
6              if (arr[j] > arr[j + 1]) {
7                  // Zamien element wiekszy i mniejszy ze soba
8                  int pomocnicza = arr[j];
9                  arr[j] = arr[j + 1];
10                 arr[j + 1] = pomocnicza;
11                 swap = true;
12             }
13         }
14         // Jesli nie dokonalismy zadnej zamiany w petli wewnetrznej, tablica jest posortowana,
           ↪  wiec przestan sortowac
15         if (swap == false) break;
16     }
17 }
```

Źródło algorytmu sortowania bąbelkowego: [5]

Problem: sortowanie

Algorytm: sortowanie bąbelkowe

Dane: tablica do posortowania (int* arr) o rozmiarze n (int n) zawierająca liczby posortowane/posortowane odwrotnie/ustawione losowo

Metoda i opis:

Zmodyfikowana wersja algorytmu ma na celu niższą złożoność dla ciągów już posortowanych.

Zewnętrzna pętla for iteruje od pierwszego elementu aż do przedostatniego (z uwagi na to, że w wewnętrznej pętli odwołujemy się do indeksu "j + 1"). Każde wykonanie pętli zewnętrznej jest tożsame z kolejnym największym elementem tablicy na odpowiednim miejscu w tablicy.

Wewnętrzna pętla dokonuje przeniesienia największego elementu na odpowiednie miejsce porównując każde dwa elementy obok siebie i zamieniając je miejscem, jeśli jeden jest większy od drugiego – robi tak dla nieposortowanej części tablicy, która rozpina się od indeksu "0" do indeksu "n - 1 - i". W pierwszej iteracji pętli zewnętrznej, "n - 1 - i" jest równe ostatniemu indeksowi, bo cała tablica jest jeszcze nieposortowana. W drugiej iteracji, "n - 1 - i" jest równe "n - 2", czyli tablica jest nieposortowana do indeksu "n - 2", czyli największy element jest na swoim miejscu.

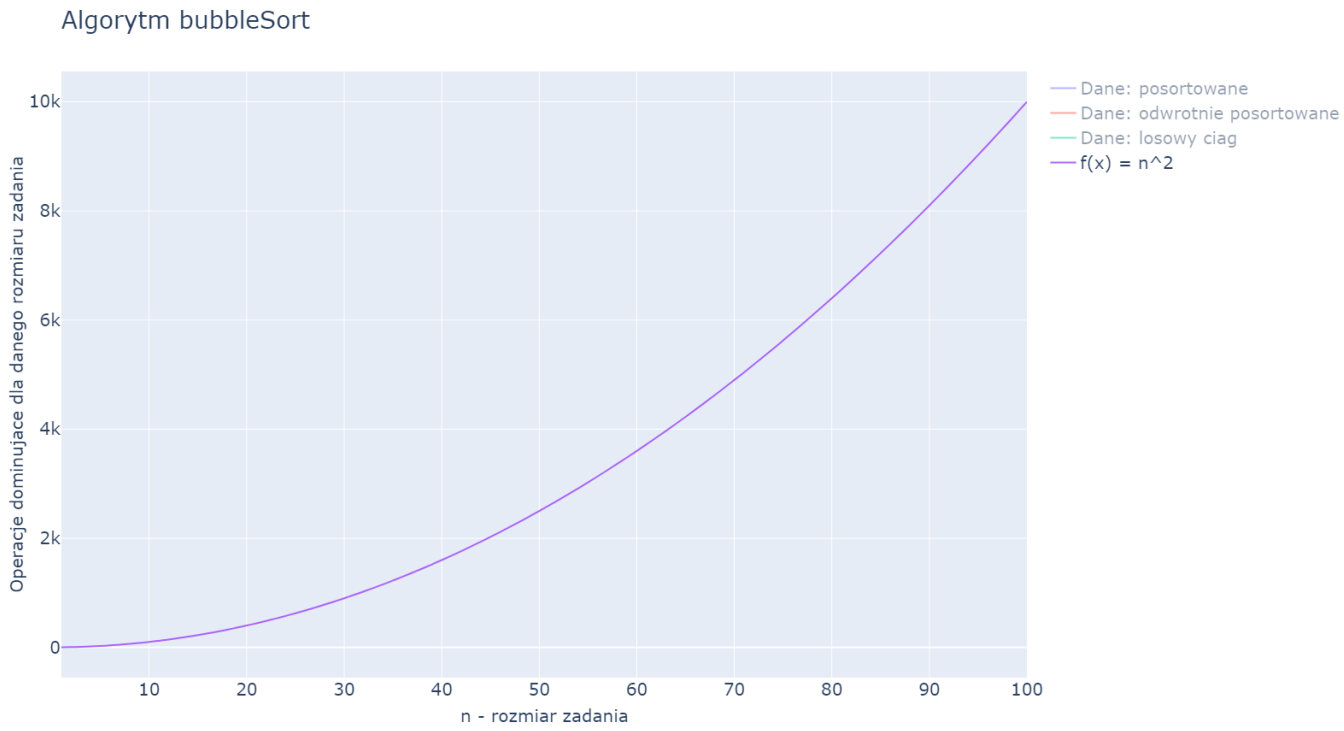
1.6.2. Opis realizacji empirycznej analizy złożoności algorytmu

Jako miejsce zliczania operacji dominującej wybrałem operacje porównania "arr[j] > arr[j + 1]", miejsce to również zlicza operacje porównania w warunku wykonania wewnętrznej pętli "for".

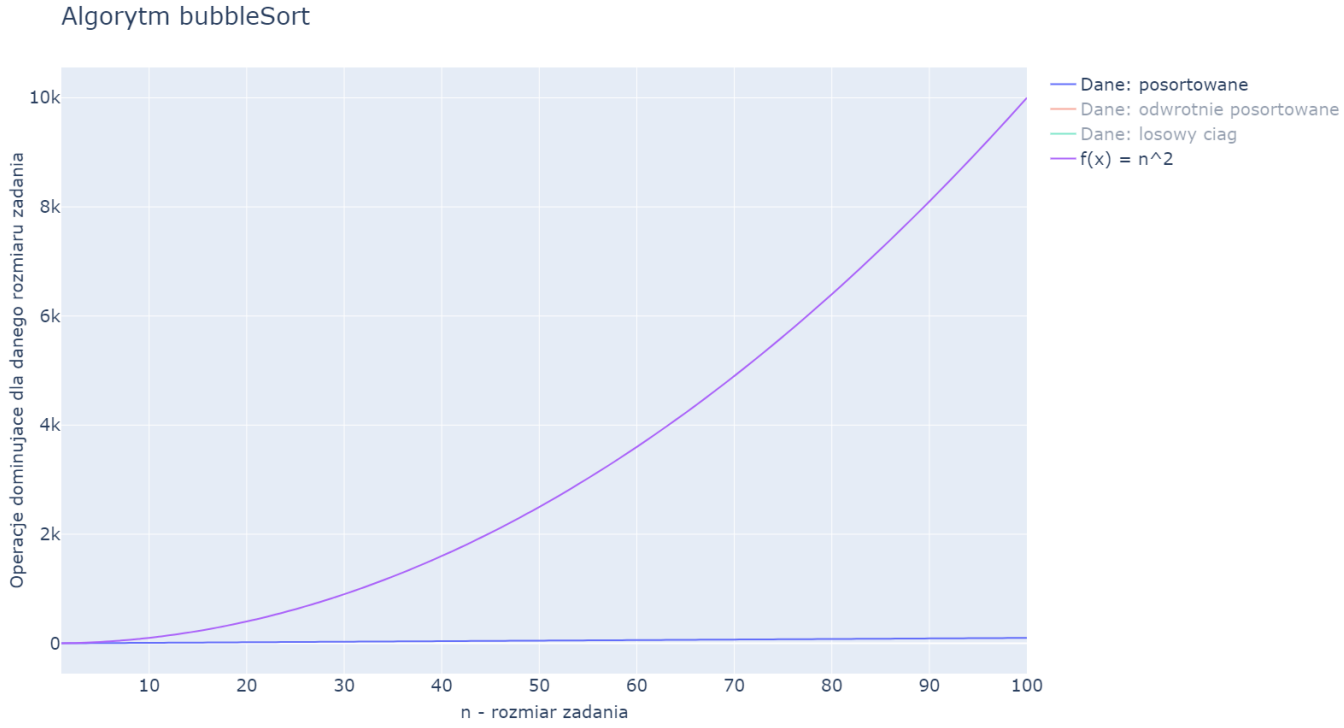
Zewnętrzna pętla for w najgorszym wypadku wykona się n razy, gdy musimy posortować n elementów. Wewnętrzna pętla w tym wypadku wykona się (n - 1) + (n - 2) + (n - 3) + (n - 4), ... + 1 razy, co jest równe $\frac{(n-1)n}{2}$ [9], więc złożoność pesymistyczna jest rzędu $\theta(n^2)$.

Zamiana elementu większego i mniejszego ma stałą złożoność.

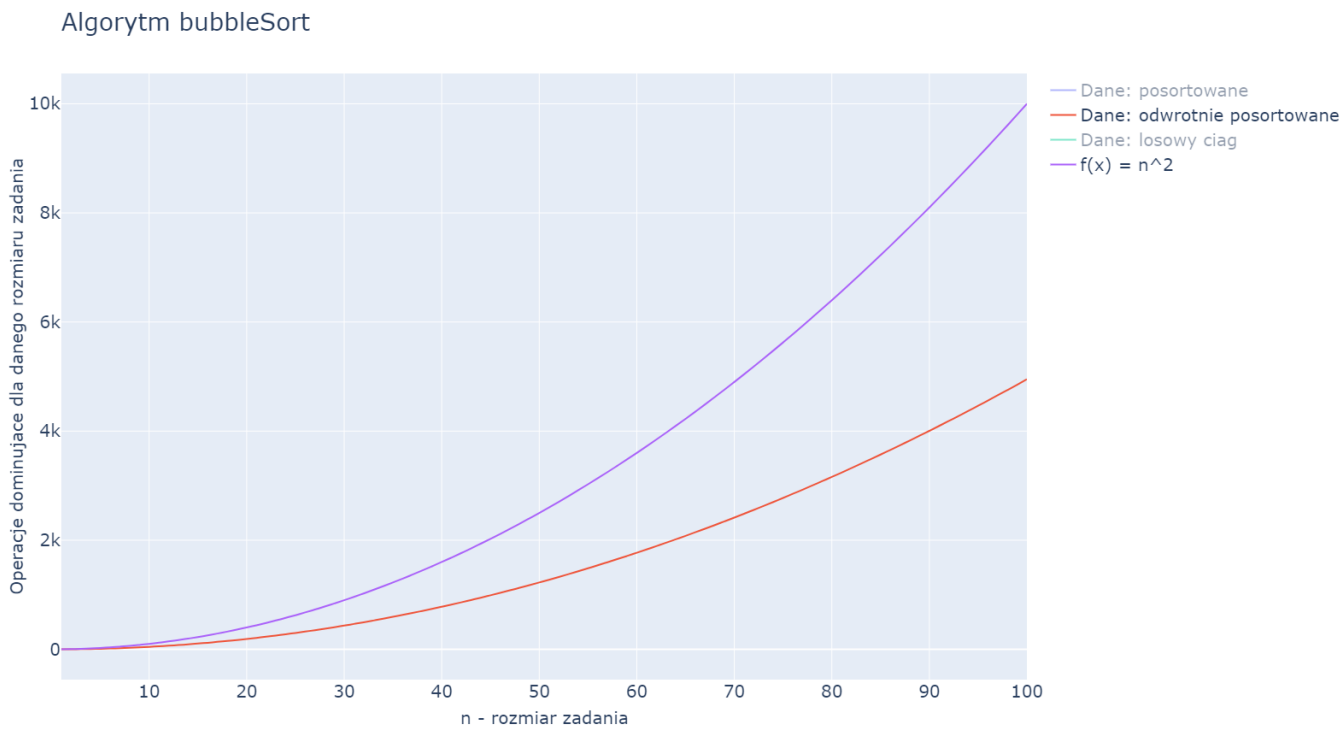
1.6.3. Prezentacja wyników



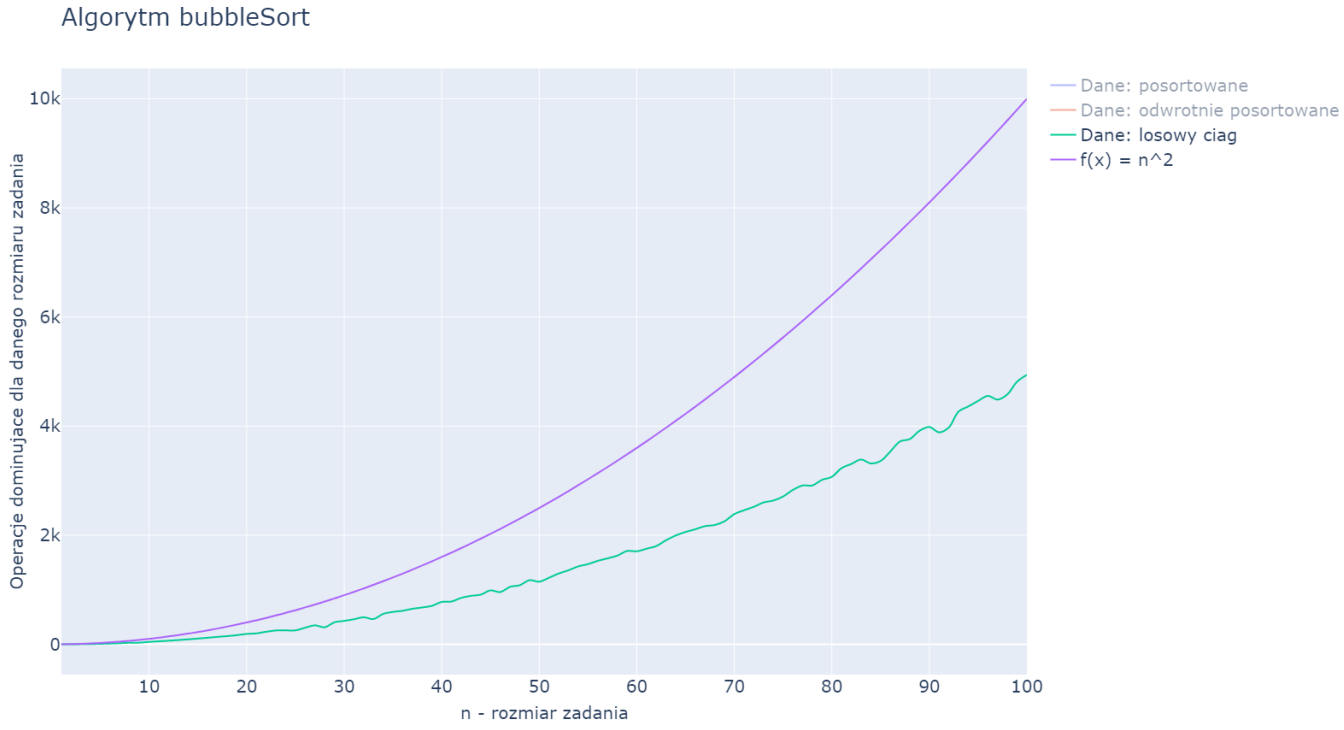
Rysunek 8: Teoretyczna złożoność bubble sortu jest rzędu $\theta(n^2)$



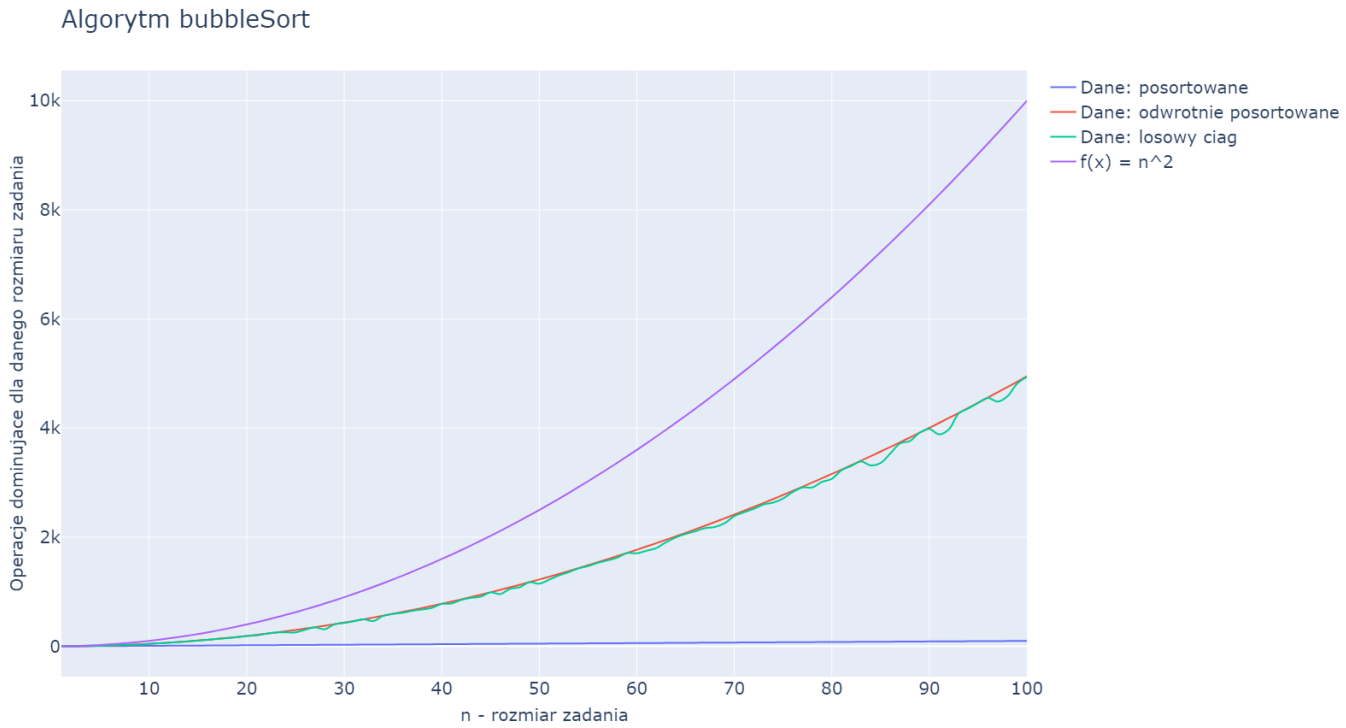
Rysunek 9: Tablice posortowane



Rysunek 10: Tablice posortowane odwrotnie



Rysunek 11: Tablice ciągów losowych



Rysunek 12: Zestawienie wszystkich typów danych

1.6.4. Wnioski

Złożoność optymistyczna: korzystam ze zmodyfikowanej wersji algorytmu bąbelkowego: jeśli wewnętrzna pętla for nie dokona ani jednej zamiany to znaczy, że tablica wejściowa jest już posortowana i algorytm może zakończyć sortowanie - wtedy złożoność jest rzędu $\theta(n)$

Złożoność pesymistyczna: jeśli tablica jest posortowana odwrotnie, algorytm musi przenieść najmniejszy element z pierwszej pozycji na ostatnią (”n” przeniesień) i musi tak zrobić ”n” razy. Jest ona rzędu $\theta(n^2)$ - na wykresie widnieje funkcja $n^2/2$, złożoność wciąż pozostaje rzędu n^2.

1.7. Sortowanie przez wstawianie

1.7.1. Opis realizacji algorytmu

```
1  void insertSort(int* arr, int n) {
2      for (int i = 1; i < n; i++) {
3          // Miejsce dodania operacji dominujacej
4          int klucz = arr[i];
5          int j = i - 1;
6          while (j >= 0 && arr[j] > klucz) {
7              // Miejsce dodania operacji dominujacej
8              arr[j + 1] = arr[j];
9              j = j - 1;
10         }
11         arr[j + 1] = klucz;
12     }
13 }
```

Źródło algorytmu sortowania przez wstawianie i informacji na jego temat: [10]

Algorytm sortowania przez wstawianie jest podobny do sortowania kart do gry, gdzie zaczynamy z pustą lewą ręką, a następnie wsadzamy do niej na właściwej pozycji odwróconą kartę leżącą na stole. Żeby znaleźć odpowiednią pozycję, porównujemy wziętą kartę ze stołu, z każdą z kart w lewej ręce. Nasza lewa ręka posiada jedynie posortowane karty, a na stole leżą karty do posortowania.

W linijce 2, indeks "i" wskazuje na obecną kartę, którą bierzemy ze stołu i chcemy wsadzić w odpowiednie miejsce w lewej ręce. W linijce 3, zapisujemy tę kartę pod zmienną "klucz".

Indeksy od "0" do "i - 1" wskazują na lewą rękę, czyli część tablicy, która jest posortowana, podczas gdy indeksy od "i" do "n - 1" wskazują na karty na stole, które musimy włożyć w odpowiednie miejsce lewej ręki.

Indeks "j" w pętli for wskazuje na indeks największej z posortowanych kart, na początku ustawiamy go jako "0", bo musimy włożyć jakąś kartę do lewej ręki, żeby zacząć proces sortowania.

Pętla "while", od karty w lewej ręce o największej wartości pod indeksem "j = i - 1" (0 w pierwszej iteracji petli for), sprawdza czy karta klucz ze stołu jest od niej mniejsza "tab[j] > klucz". Jeśli tak, to przesuwa kartę w lewej ręce o jedno miejsce w prawo "tab[j + 1] = tab[j]", a następnie przygotowuje poprzednią kartę z lewej ręki do sprawdzenia "j = j - 1".

Jeśli jednak najwyższa karta karta w lewej ręce nie jest mniejsza, to znaczy, że karta klucz jest na swojej pozycji, bo lewa ręka jest w każdym momencie posortowana.

Pętla while zakończy się również jeśli nie ma już kart w lewej ręce do porównania z kartą klucz, co jest zapisane warunkiem wykonania pętli while "j >= 0".

Po pętli while, linijka 9 umieszcza obecnie sortowaną kartę klucz na odpowiedniej pozycji, tym samym zwiększa się liczba posortowanych kart – na początku petli "for" zmienna "j" będzie większa o "1" niż w poprzedniej iteracji.

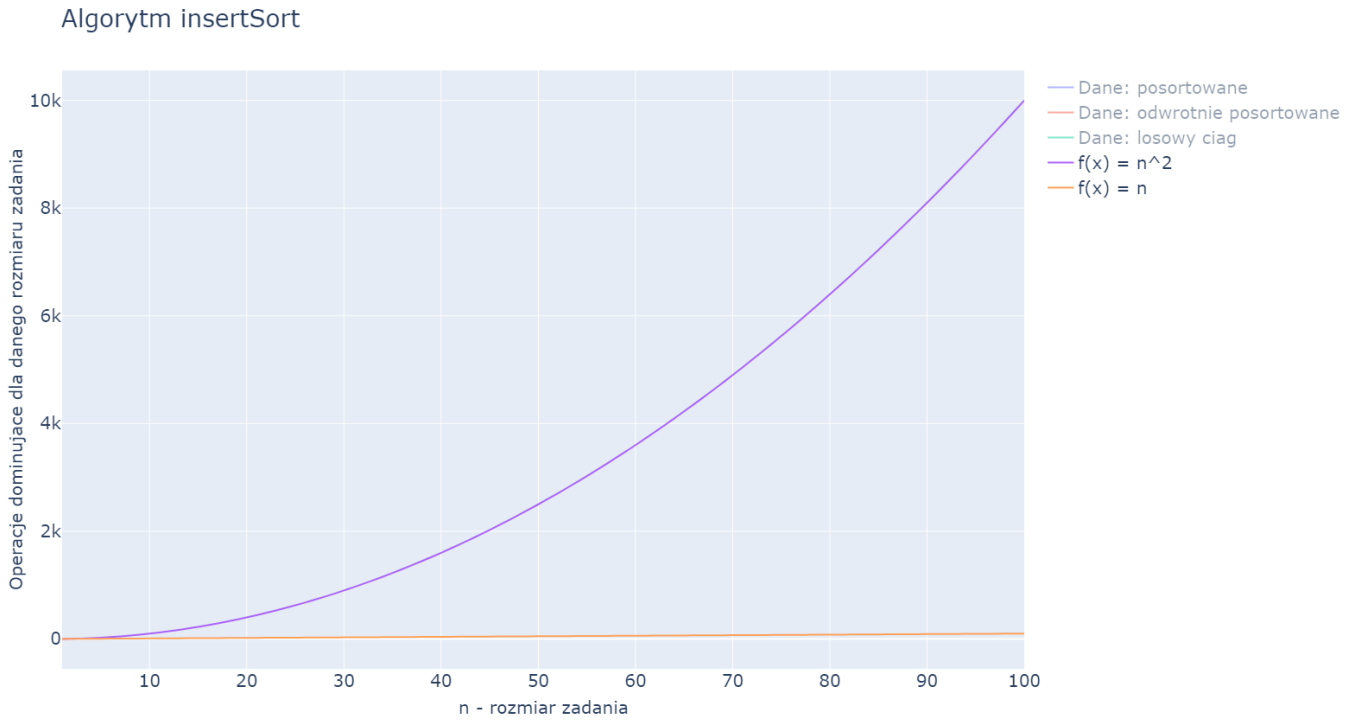
```
Przyklad w pseudokodzie:

arr = [15, 12, 3, 1], insertSort(arr, 4):
- [15, 12, 3, 1] <- zewnetrzne for -- lewa reka = [15]
    - i = 1, klucz = 12, j = 0
    - 0 >= 0 i 15 > 12
        - [15, 15, 3, 1]
        - j = -1
    - [12, 15, 3, 1] -- klucz na wlasciwym miejscu w lewej rece, lewa reka zwieksza sie o 1
- [12, 15, 3, 1] <- zewnetrzne for -- lewa reka = [12, 15]
    - i = 2, klucz = 3, j = 1
    - 1 >= 0 i 15 > 3
        - [12, 15, 15, 1]
        - j = 0
    - 0 >= 0 i 12 > 3
        - [12, 12, 15, 1]
        - j = -1
    - [3, 12, 15, 1]
- [3, 12, 15, 1] - zewnetrzne for -- lewa reka = [3, 12, 15]
- itd...
```

1.7.2. Opis realizacji empirycznej analizy złożoności algorytmu

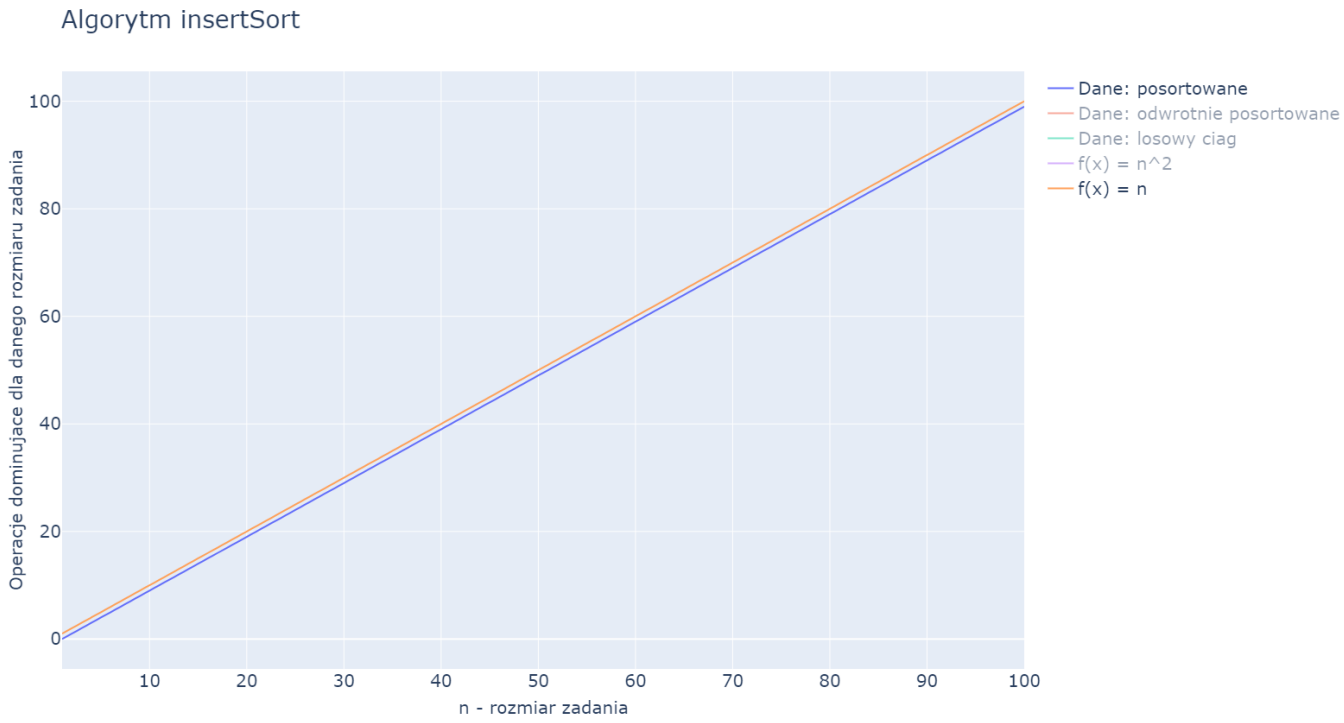
Zewnętrzna pętla for zawsze wykona się "n" razy, bo musimy przejść całą tablicę, nawet jeśli jest posortowana.

Liczba wykonań pętli "while" jest zależna od długości wektora danych i od ustawienia elementów w tym wektorze [9]. Jest to pętla, która porównuje elementy w coraz to większej posortowanej lewej ręce, z zależnym od pętli "for" elementem kluczem. Tak samo jak w przypadku bubble sort, w pesymistycznym wypadku tablicy posortowanej odwrotnie mamy (n - 1) + (n - 2) + (n - 3) + ... + 1 wykonań tej pętli co daje nam $\frac{n^2-n}{2}$, czyli jest rzędu $\theta(n^2)$.

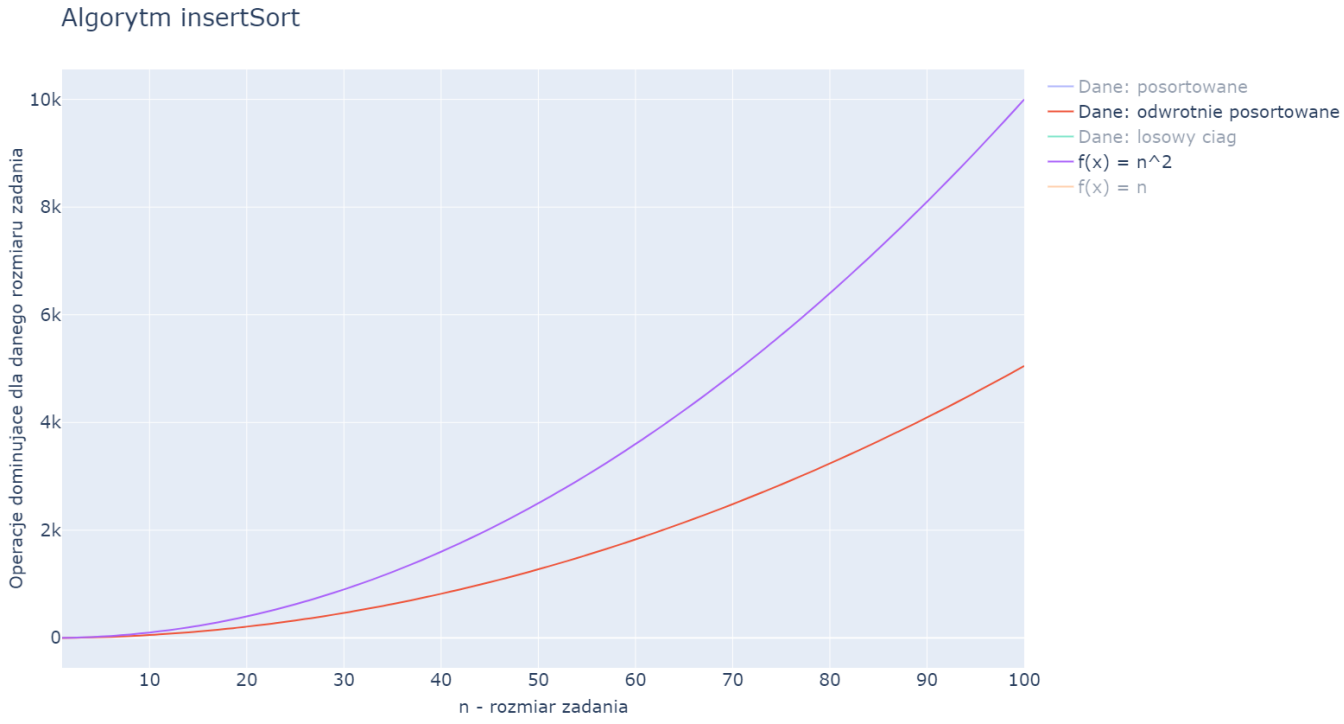
1.7.3. Prezentacja wyników



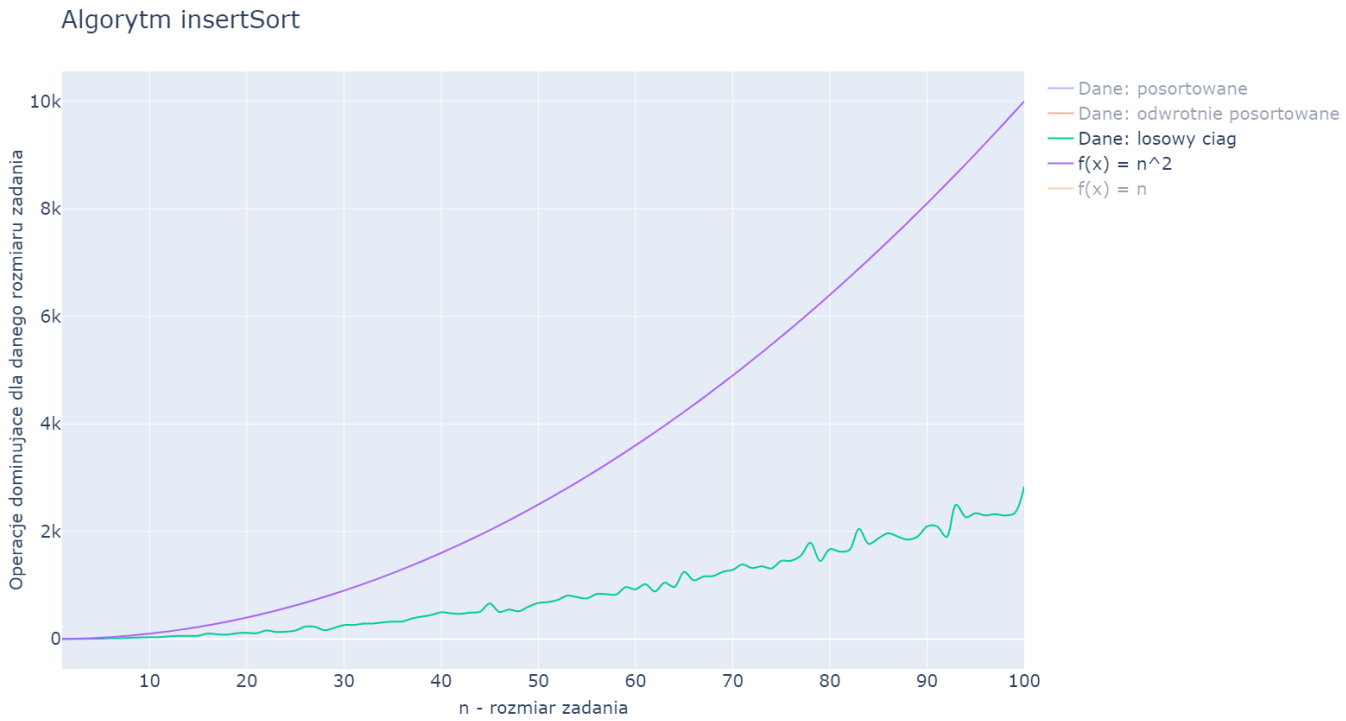
Rysunek 13: Teoretyczna złożoność pesymistyczna insert sortu jest rzędu $\theta(n^2)$, a optymistyczna $\theta(n)$



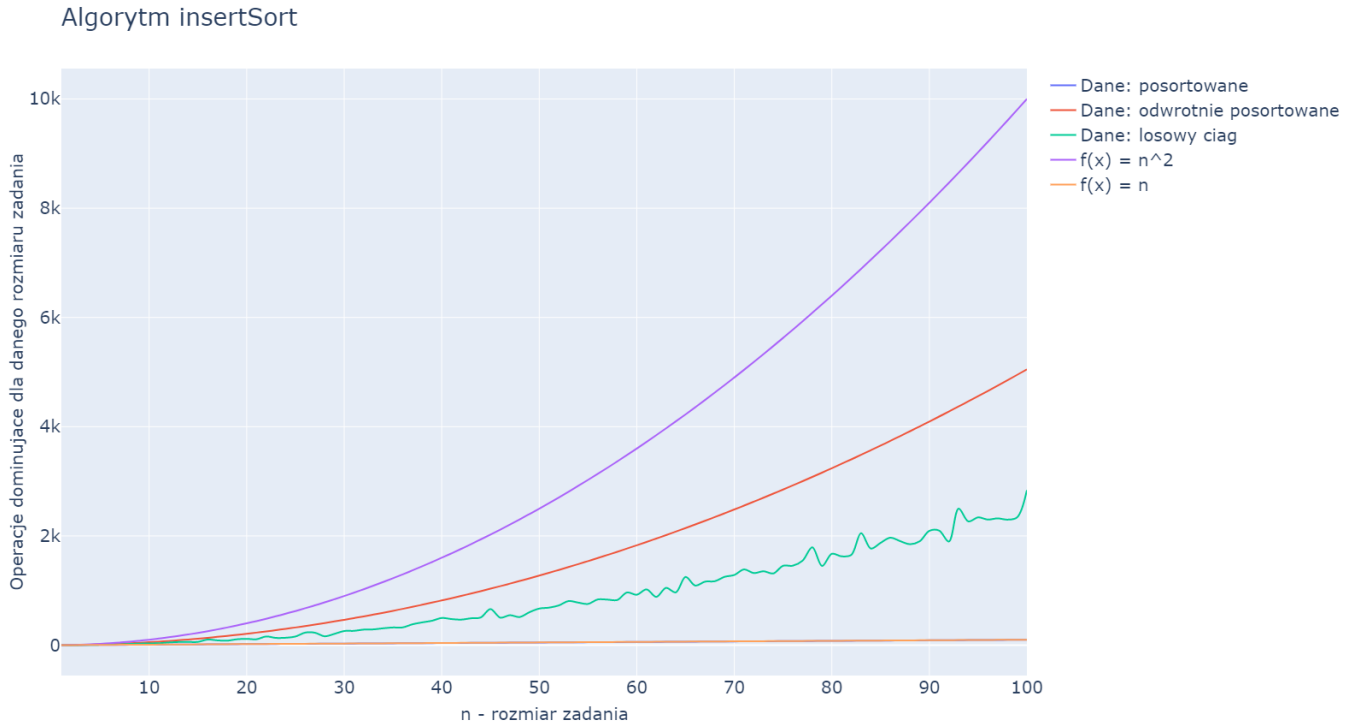
Rysunek 14: Tablice posortowane



Rysunek 15: Tablice posortowane odwrotnie



Rysunek 16: Tablice ciągów losowych



Rysunek 17: Zestawienie wszystkich typów danych

1.7.4. Wnioski

Algorytm sortowania przez wstawianie posiada niską złożoność dla ciągów, które są prawie lub w pełni posortowane, sprawdza się on więc dla ciągów o niskiej ilości elementów.

Empiryczne analizy pokrywają się z teorią, dla tablic posortowanych odwrotnie wykres przybiera postać $n^2/2$, jednak pomijając stałą, złożoność wciąż jest proporcjonalna do n^2.

W przypadku zmodyfikowanej wersji algorytmu bąbelkowego którą zastosowałem, złożoność dla tablic posortowanych jest taka sama dla sortowania bąbelkowego, jak i przez wstawianie. Złożoność obu algorytmów jest podobna, jedynie dla ciągów losowo posortowanych można zauważyć, że insertSort ma troche niższą złożoność, co można zauważyć przez porównanie zestawienia wszystkich typów danych dla insertSort i bubbleSort – niektóre elementy mogą już być na swoim miejscu w insert sorcie, co skraca czas wykonania. Jednak różnica się wydaje być na tyle niezauważalna, że można przyrównać ją do stałej, więc złożoność pozostaje taka sama.

1.8. Sortowanie przez scalanie

1.8.1. Opis realizacji algorytmu

```
1  void mergeSort(int* arr, int l, int r) {
2      if (l < r) { // jesli mamy przynajmniej 2 elementy w tablicy
3          int m = (l + r) / 2; // srodkowy indeks
4          mergeSort(arr, l, m); // posortuj rekurencyjnie lewa czesc wejsciowej tablicy
5          mergeSort(arr, m + 1, r); // posortuj rekurencyjnie prawa czesc wejsciowej tablicy
6          merge(arr, l, m, r); // scal posortowana lewa i prawa czesc nadpisujac wejsciowa tablice
7      }
8  }
9
10 // Scalanie posortowanych tablic
11 void merge(int* arr, int l, int m, int r) {
12     int n1 = m - l + 1; // rozmiar posortowanej lewej podtablicy
13     int n2 = r - m; // rozmiar posortowanej prawej podtablicy
14
15     // Tymczasowe tablice, potrzebujemy ich, bo będziemy nadpisywać główną tablicę
16     int* L = new int[n1];
17     int* R = new int[n2];
18
19     // Wypelnij posortowane tablice odpowiednimi połowami glownej tablicy
20     for (int i = 0; i < n1; i++)
21         L[i] = arr[l + i];
22     for (int j = 0; j < n2; j++)
23         R[j] = arr[m + 1 + j];
24
25     // Iteratory po połówkach
26     int i = 0;
27     int j = 0;
28
29     // Iterator po głównej tablicy, do której będziemy wkładać wartości z prawej i lewej
       ↪ posortowanej tablicy, tak, żeby były one w rosnącej kolejności
30     int k = l;
31
32     // Iteruj dopóki "i" bedzie wskazywac na element z pierwszej polowki, a "j" bedzie wskazywac
       ↪ na element z drugiej polowki
33     while (i < n1 && j < n2) {
34         // 2-gie miejsce dodania operacji dominujacej
35         // Jesli np. 1sza wartosc lewej tablicy jest mniejsza od 1szej wartosci prawej tablicy,
           ↪ nadpisz te wartosc na 1szym miejscu glownej tablicy. Po kazdym przypisaniu przesun
           ↪ sie na kolejny indeks powstajacej tablicy scalonej, a także na kolejny indeks tablicy
           ↪ ktora miesci mniejszy/rowny element
36         arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
37     }
38     // Jesli szybciej dojdziemy do końca prawej tablicy niż lewej, to oznacza, że wstawiliśmy już
       ↪ z niej do głównej tablicy wszystkie elementy i teraz musimy nadpisać resztę głównej
       ↪ tablicy pozostałą częścią lewej tablicy
39     while (i < n1) {
40         // 3-cie miejsce dodania operacji dominujacych -- ma znaczenie zwlaszcza dla tablic
           ↪ posortowanych odwrotnie
41         arr[k++] = L[i++];
42     }
43     // Nie musimy przepisywac juz drugiej polowki, bo jesli juz przepisalismy pierwsza polowke i
       ↪ np. jakis kawalek drugiej polowki, to ten drugi kawalek bedzie juz posortowany, bo
       ↪ elementy przepisywane beda mniejsze od tych, ktore sa w pozostalej czesci drugiej polowki
44 }
```

Źródła do opisów: [9]
Źródło algorytmu sortowania przez scalanie: [6], [9]

Metoda:
Algorytm merge sort tak samo jak binary search korzysta z metody "dziel i rządź".

1. **Dziel:** w funkcji "mergeSort", znajdź środkowy indeks "m" tablicy "arr".
2. **Rządź:** rekursywnie posortuj podtablice o indeksach [l, m] i [m + 1, r].
3. **Połącz:** scal ze sobą tablice posortowane w punkcie 2, w jedną połączoną tablicę (funkcja "merge")

Ponieważ korzystamy z rekurencji, potrzebujemy warunku wyjścia (base case). Nie ma potrzeby sortować tablicy mniejszej niż 2 elementy, więc jako warunek wyjścia ustalamy "if (l < r)".

```
Przykładowe wywołanie algorytmu w pseudokodzie:

arr = [14, 7, 3, 12, 9]
n = 5

mergeSort(arr, 0, n - 1):
    Dziel: srodkowy indeks "m" = 2 dzieli nam większą tablicę na dwie podtablicę:
    arr[0...2] i arr[3...n - 1]

    Teraz posortujemy je rekurencyjnie:

    mergeSort(arr, 0, 2); [14, 17, 3]
        l < r = TRUE
        m = 1
        mergeSort(arr, 0, 1); [14, 17]
            l < r = TRUE
            m = 0
            mergeSort(arr, 0, 0) -> return -- tablica jednoelementowa [14]
            mergeSort(arr, 1, 1) -> return -- tablica jednoelementowa [17]
            merge(arr, 0, 0, 1)
                arr[0..1] = [14, 7]
                L = [14]
                R = [7]
                Scalamy obie tablice ze sobą i arr = [7, 14]
        mergeSort(arr, 2, 2) -> return -- tablica jednoelementowa [3]
        merge(arr, 0, 1, 2)
            arr[0..2] = [7, 14, 3]
            L = [7, 14] <- wynik wcześniejszego merge(arr, 0, 1)
            R = [3]
            Scalamy L i R ze sobą: [3, 7, 14] -- pierwsza połowa tablicy została posortowana.
    Teraz zajmiemy się drugą połową tablicy.
    mergeSort(arr, 3, 4); [12, 9]
        l < r = TRUE
        m = 3
        mergeSort(arr, 3, 3) -> return -- tablica jednoelementowa [12]
        mergeSort(arr, 4, 4) -> return -- tablica jednoelementowa [9]
        merge(arr, 3, 3, 4)
            arr[3..4] = [12, 9]
            L = [12]
            R = [9]
            Scalamy obie tablice ze sobą
            wynik: arr = [9, 12]
    Po zakończeniu powyższego pseudokodu podtablice arr[0..2] i arr[3..n - 1] są posortowane.
    Teraz czas scalić je ze sobą:
    merge(arr, 0, 2, n - 1);
        arr[0..2] = [3, 7, 14]
        arr[3..4] = [9, 12]
        Scalamy obie te tablice ze sobą i otrzymujemy posortowaną tablicę [3, 7, 9, 12, 14]
```

1.8.2. Opis realizacji empirycznej analizy złożoności algorytmu

Informacje o analizie złożoności czerpałem stąd: [7]

W swoim programie jako operacje dominujące uznałem:

- podział na mniejszy podproblem, czyli wywołanie funkcji "mergeSort"
- pętle scalające dwa podproblemy w funkcji "merge"

Teraz wytłumaczę dlaczego:

1. Dzielenie większego problemu na mniejsze podproblemy jest rzędu $\theta(1)$, bo musimy tylko wyliczyć indeks "m".
2. Funkcja scalająca "merge" scala elementy w czasie θ(n), bo porównuje wartości dwóch tablic nadpisując tablicę wejściową tak, że powstaje tablica posortowana

Z tej racji rozważymy podpunkty 1 i 2 jako razem zajmujące cn czasu, gdzie c to stała.

Dla pierwszego wywołania funkcji na n-elementowej tablicy "mergeSort" zajmuje dwa razy czas wywołania "mergeSort" dla podproblemów o rozmiarach $n/2 + cn$ dla scalenia rozwiązanych podproblemów.

Pytanie teraz brzmi: jaka jest złożoność wywołań "mergeSort" dla $n/2$ elementów, które z kolei wywołują "mergeSort" dla "n / 4" elementów, "n / 8" elementów i tak dalej, w zależności od wielkości tablicy.

Dla "n" elementów scalanie zajmuje cn czasu.

Dla "n / 2" elementów scalanie zajmuje $cn/2$ czasu
Jednak rozbijając problem o rozmiarze "n" na dwa problemy o rozmiarze "n / 2",
musimy pomnożyć czas jeszcze razy 2, więc mamy $cn/2 * 2 = cn$

Dla 4 "n / 4" elementów scalanie zajmuje $cn/4 * 4 = cn$ czasu.

Złożoność scalania podproblemu możemy więc uogólnić jako cn dla każdego poziomu podproblemu,
aż do (i włącznie z) tablicą 1 elementową.

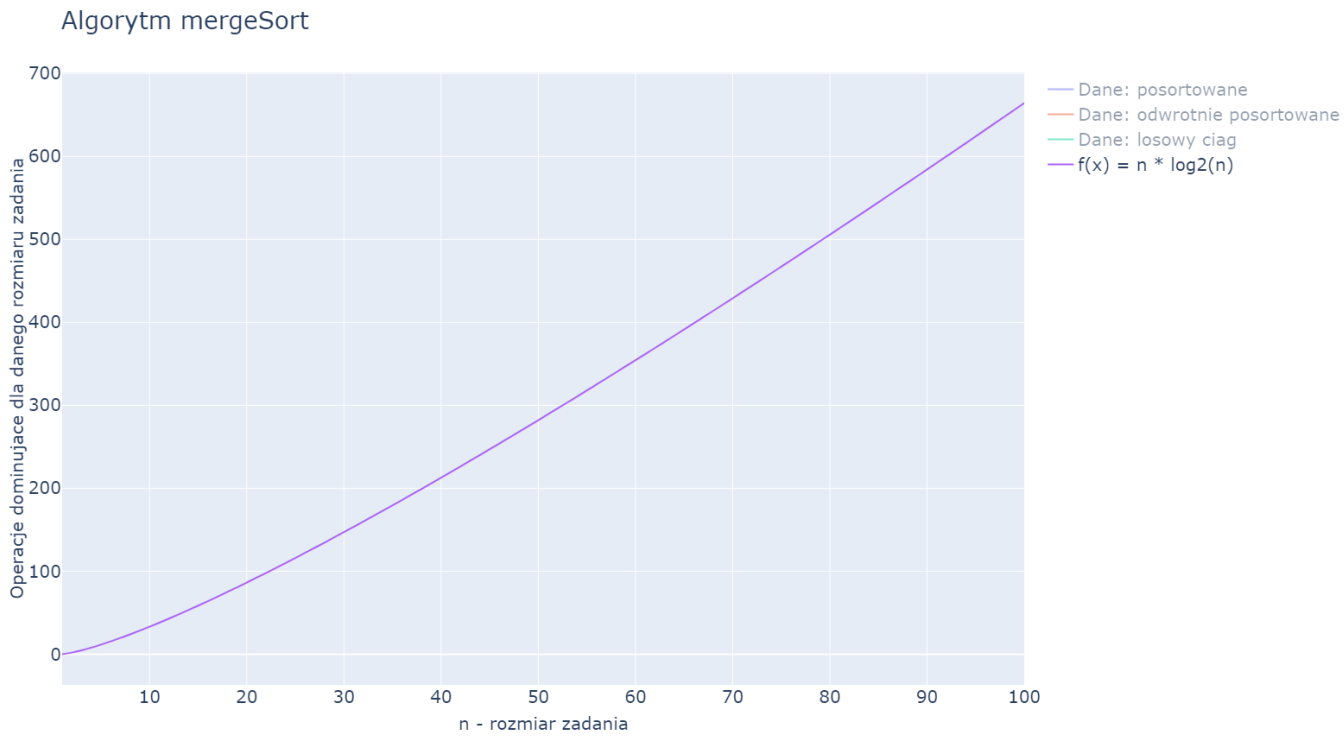
Dla powyższego przykładu z tablicą [14, 7, 3, 12, 9] podział tablicy wygląda tak:

```
- [14, 7, 3, 12, 9] -- n
- [14, 7, 3] -- n / 2
    - [14, 7] -- n / 4
        - [14] -- n / 8
        - [7] -- n / 8
    - [3] -- n / 4
- [12, 9] -- n / 2
    - [12] -- n / 4
    - [9] -- n / 4
```

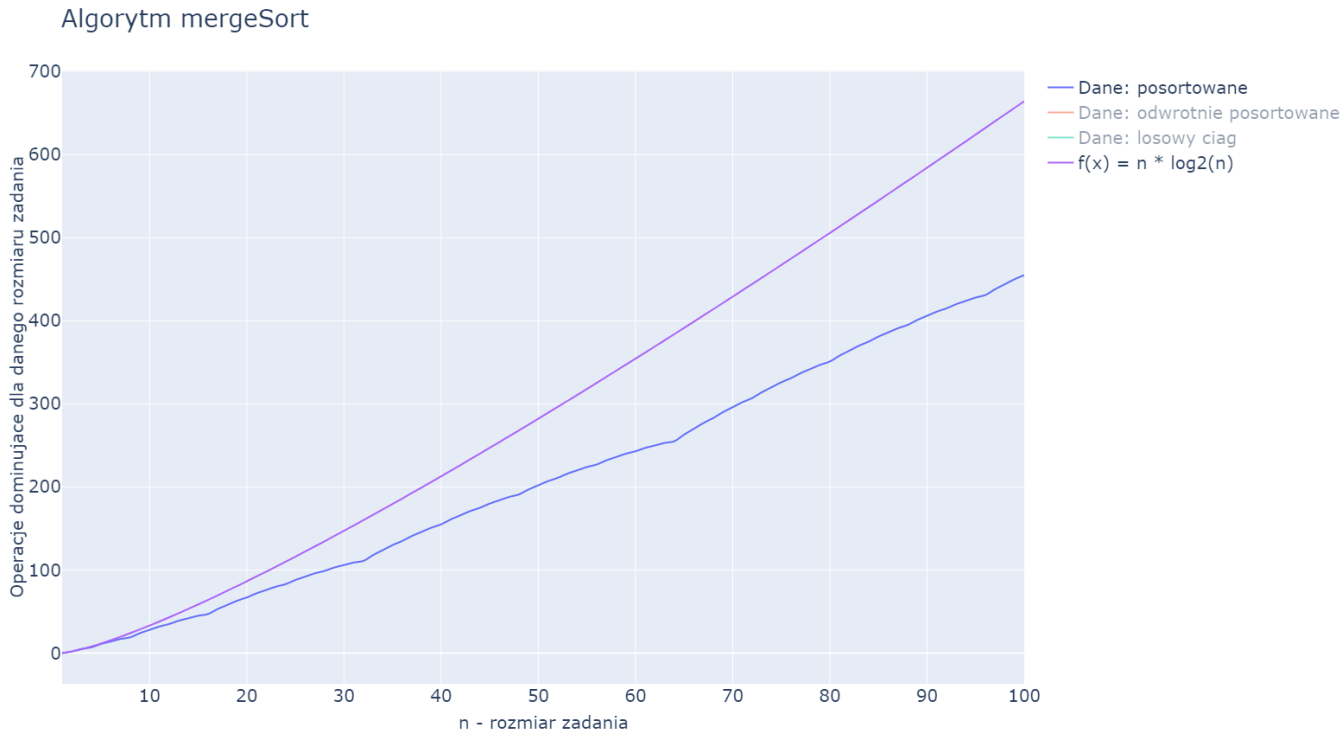
Ponieważ zaokrąglamy rozmiar n / 2 = 5 / 2 z 2.5 do 2,
to otrzymujemy pewne odchylenie, ale nie ma ono znaczenia dla rzędu złożoności.

Dla każdego podziału czas wynosi cn. Tablicę o rozmiarze "n" możemy podzielić na 2 $log2(n)$ razy.
Więc złożoność, pomijając stałą "c", wynosi $\theta(nlog2(n))$

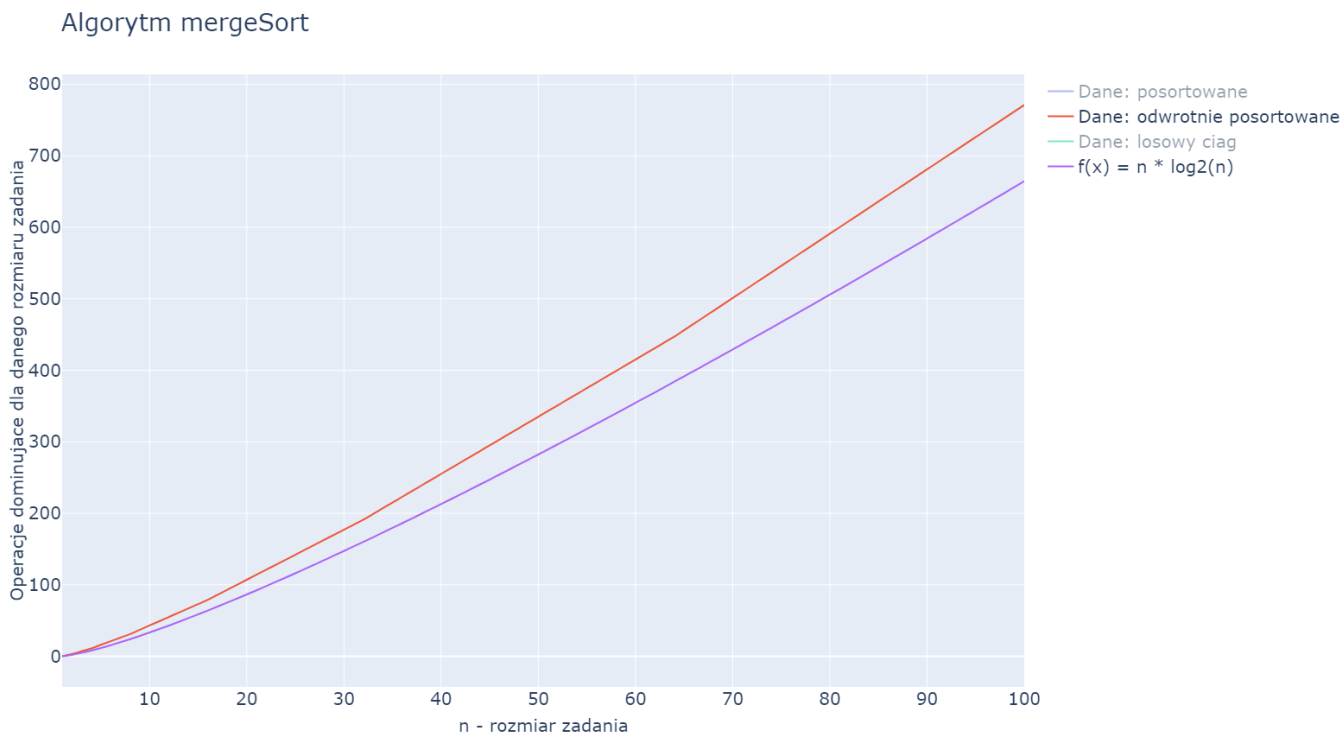
1.8.3. Prezentacja wyników



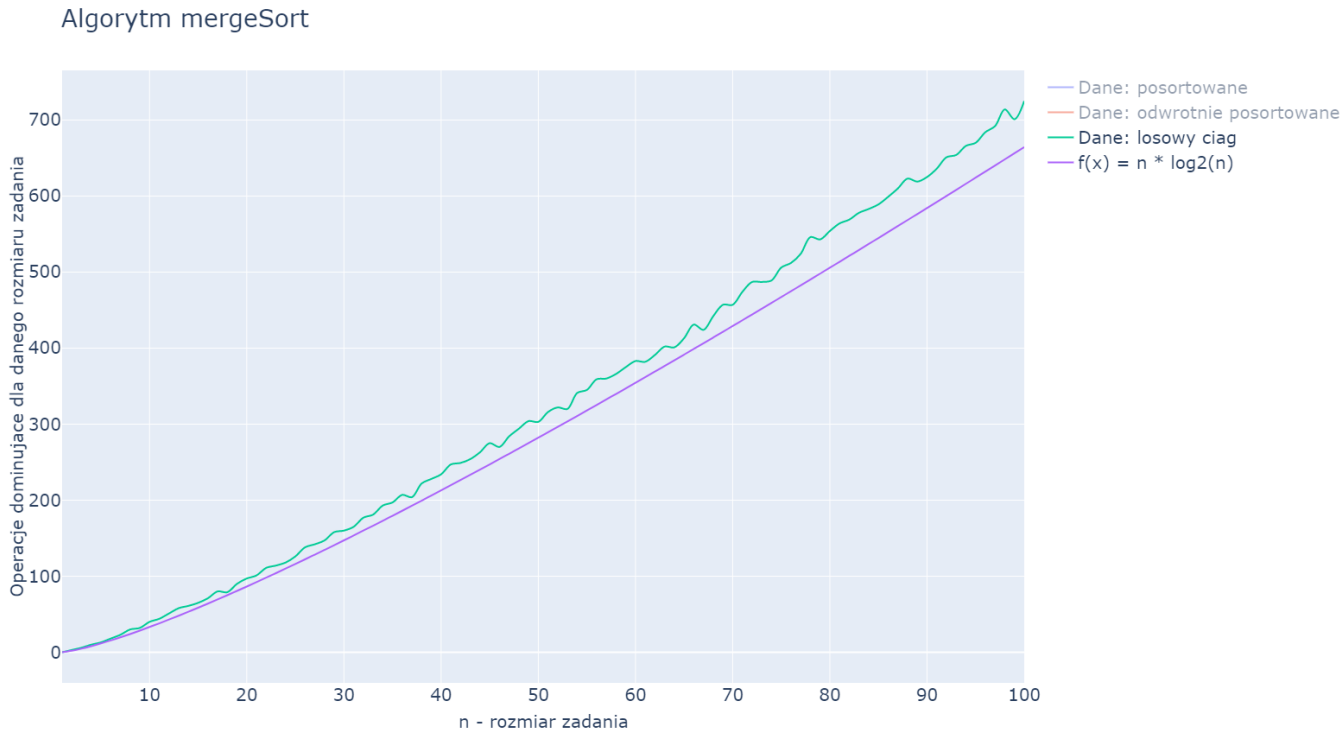
Rysunek 18: Teoretyczna złożoność merge sortu jest rzędu $\theta(nlog2(n))$ dla każdego typu danych wejściowych.



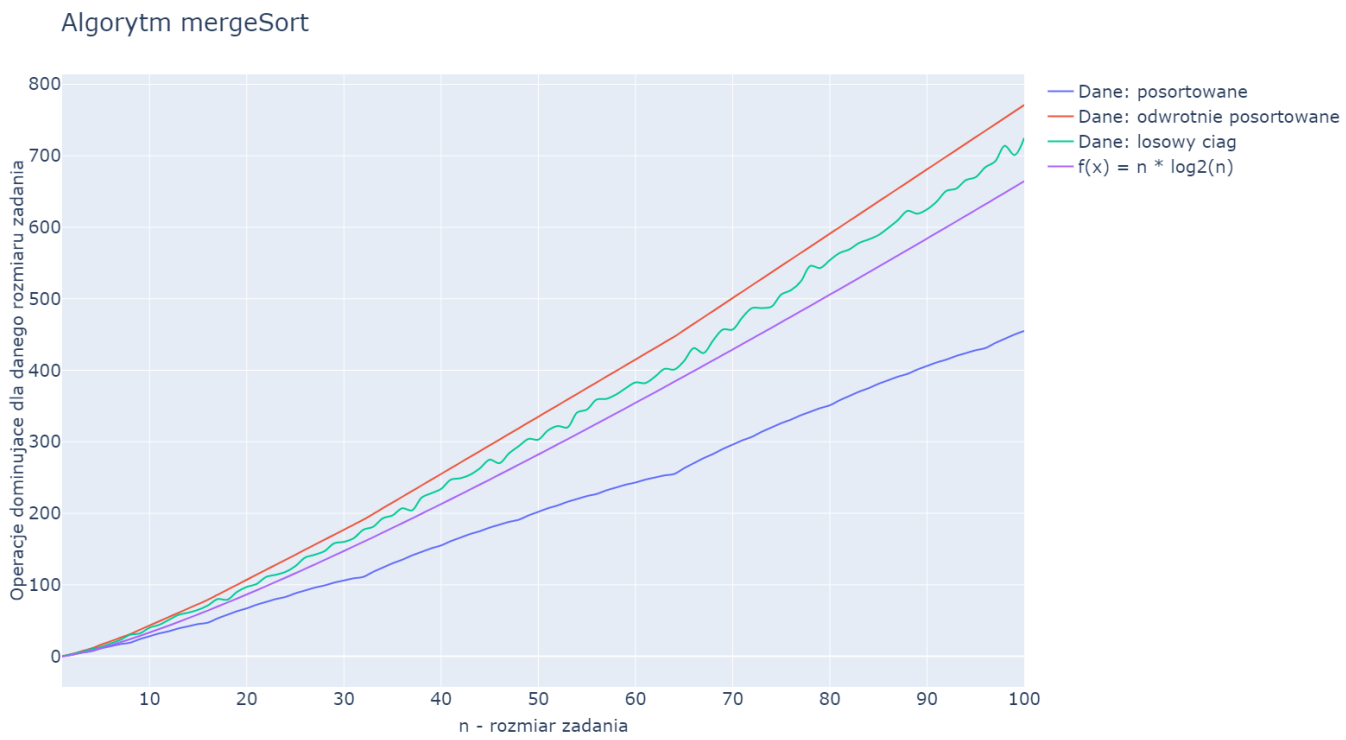
Rysunek 19: Tablice posortowane



Rysunek 20: Tablice posortowane odwrotnie



Rysunek 21: Tablice ciągów losowych



Rysunek 22: Zestawienie wszystkich typów danych

1.8.4. Wnioski

Analiza empiryczna wykazała, że dla każdego typu danych wejściowych złożoność algorytmu oscyluje koło $\theta(nlog2(n))$.

Merge sort gwarantuje stałą złożoność dla każdego rozmiaru i typu posortowania danych wejściowych.

Dla małych rozmiarów danych wejściowych merge sort jest wolniejszy od innych algorytmów. Niezależnie od tego czy dane są posortowane, merge sort zawsze się wykona do końca w przeciwieństwie do insert sortu i wersji bubble sortu, którą zaprezentowałem.

Dodatkowo, merge sort nie jest algorytmem sortowania w miejscu, wymaga "2N + k" komórek pamięci, gdzie k to pewna stała. Jest to więcej więcej niż "k + n" bubble i insert sortu. W obu przypadkach jednak rząd złożoności pamięciowej jest ten sam dla tych trzech algorytmów. [9]

Literatura i źródła

[1] https://www.samouczekprogramisty.pl/podstawy-zlozonosci-obliczeniowej/. [Online; dostęp 20 Maj 2022].

[2] https://en.wikipedia.org/wiki/Linear_search#:~:text=Linear%20search%20is%20usually%20very,to%20use%20a%20faster%20method. [Online; dostęp 20 Maj 2022].

[3] https://www.geeksforgeeks.org/binary-search/. [Online; dostęp 15 Maj 2022].

[4] https://stackoverflow.com/a/27512035. Why does binarySearch need a sorted array? [Online; dostęp 15 Maj 2022].

[5] https://www.geeksforgeeks.org/bubble-sort/. [Online; dostęp 8 Maj 2022].

[6] https://www.geeksforgeeks.org/iterative-merge-sort/. [Online; dostęp 18 Maj 2022].

[7] https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort. [Online; dostęp 18 Maj 2022].

[8] mgr inż. Grażyna Szostak. *Materiały źródłowe z laboratorium*. WSIiZ.

[9] prof. dr hab. inż. Władysław Homenda. *Informacje podane na wykładach*. WSIiZ.

[10] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms, 3rd edition*. MIT Press.