

Dokumentacja projektu  
*"Algorytmy Sortowania w Informatyce"*

Yuliya Zviarko

Wydział Fizyki i Informatyki Stosowanej  
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Styczeń 2025

## Spis treści

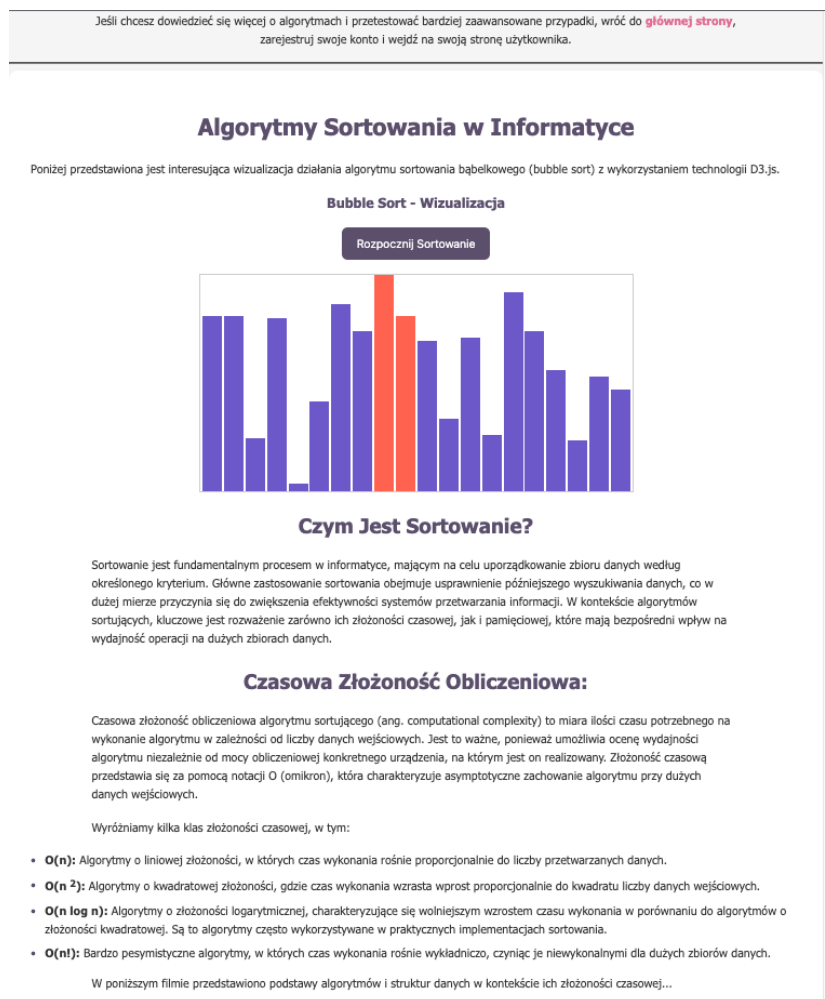
1	Zawartość merytoryczna projektu	3
2	Zawartość funkcjonalna projektu	4
3	Język HTML5 i style CSS	11
4	Grafika w projekcie	11
5	JavaScript w projekcie	11
6	Część serwerowa aplikacji	11
7	Ciekawa technologia w projekcie	11

# 1 Zawartość merytoryczna projektu

Celem projektu było zaprojektowanie i wdrożenie kilku popularnych algorytmów sortowania. Skupiłam się na trzech metodach: sortowaniu bąbelkowym, sortowaniu przez wybieranie oraz algorytmie Quicksort. Każdy z tych algorytmów został zaimplementowany w różnych wersjach i przetestowany na danych o różnej wielkości. Głównym celem badań było sprawdzenie wydajności tych algorytmów pod względem czasu działania. Sprawdzono również, kiedy najlepiej je stosować, biorąc pod uwagę rodzaj sortowanych danych. Dodatkowo, przeprowadzono porównanie tych algorytmów w celu ustalenia, który z nich jest najskuteczniejszy w różnych sytuacjach.

## 2 Zawartość funkcjonalna projektu

Na początkowej stronie projektu znajduje się ogólna prezentacja, która jest dostępna dla wszystkich użytkowników. Logowanie nie jest wymagane. Użytkownicy mogą uruchomić prostą animację napisaną w D3.js, która ilustruje, jak działa proces sortowania tablicy.



Rysunek 1: Strona główna dla wszystkich użytkowników.

Dodatkowo, dostępny jest filmik z informacjami wstępnymi na temat algorytmów sortowania oraz złożoności obliczeniowej.

### Czasowa Złożoność Obliczeniowa:

Czasowa złożoność obliczeniowa algorytmu sortującego (ang. computational complexity) to miara ilości czasu potrzebnego na wykonanie algorytmu w zależności od liczby danych wejściowych. Jest to ważne, ponieważ umożliwia ocenę wydajności algorytmu niezależnie od mocy obliczeniowej konkretnego urządzenia, na którym jest on realizowany. Złożoność czasową przedstawia się za pomocą notacji  $O$  (omikron), która charakteryzuje asymptotyczne zachowanie algorytmu przy dużych danych wejściowych.

Wyróżniamy kilka klas złożoności czasowej, w tym:

- $O(n)$ :** Algorytmy o liniowej złożoności, w których czas wykonania rośnie proporcjonalnie do liczby przetwarzanych danych.
- $O(n^2)$ :** Algorytmy o kwadratowej złożoności, gdzie czas wykonania wzrasta wprost proporcjonalnie do kwadratu liczby danych wejściowych.
- $O(n \log n)$ :** Algorytmy o złożoności logarytmicznej, charakteryzujące się wolniejszym wzrostem czasu wykonania w porównaniu do algorytmów o złożoności kwadratowej. Są to algorytmy często wykorzystywane w praktycznych implementacjach sortowania.
- $O(n!)$ :** Bardzo pesymistyczne algorytmy, w których czas wykonania rośnie wykładniczo, czyniąc je niewykonalnymi dla dużych zbiorów danych.

W poniższym filmie przedstawiono podstawy algorytmów i struktur danych w kontekście ich złożoności czasowej...

### Przykład:

Założmy, że algorytm o złożoności  $O(n^2)$  sortuje 100 elementów w 1 sekundę. Wówczas, sortowanie 1000 elementów zajmie już 100 sekund, co obrazuje wykładniczy wzrost czasu wykonania wraz ze wzrostem liczby danych.

Lp.	n	Czas obliczeń
1.	100	1 sekunda
2.	1,000	100 sekund = 1 minuta 40 sekund
3.	10,000	10,000 sekund = 2 godziny 46 minut 40 sekund
4.	100,000	1,000,000 sekund = 11 dni 13 godzin 46 minut 40 sekund

Rysunek 2: Filmik na stronie głównej.

Na stronie znajduje się także interaktywny element w postaci mini gry z patyczkami, gdzie użytkownicy mogą losowo ustawiać wysokość patyczków i sortować je za pomocą przeciągania.

```

1 <style>
2   /* Stylizacja gry z patyczkami */
3   #sticks-container {
4     display: flex;
5     justify-content: space-around;
6     flex-wrap: wrap;
7     gap: 10px;

```

```

8     }
9
10    .stick {
11        width: 10px;
12        background-color: #4CAF50;
13    }
14 </style>
15
16 <h3>Mini gra z patyczkami:</h3>
17 <div id="sticks-container"></div>
18
19 <script src="https://cdnjs.cloudflare.com/ajax/libs/Sortable/
20 1.14.0/Sortable.min.js"></script>
21
22 <script>
23     // Generowanie losowych patyczek w
24     function generateSticks() {
25         const container =
26             document.getElementById('sticks-container');
27         const numSticks = 50; // Liczba patyczek w
28         const heights = [];
29
30         for (let i = 0; i < numSticks; i++) {
31             const height = Math.floor(Math.random() * 300) +
32                 20; // Wysoko losowana między 20 a 300 px
33             heights.push(height);
34
35             const stick = document.createElement('div');
36             stick.classList.add('stick');
37             stick.style.height = height + 'px';
38             container.appendChild(stick);
39         }
40     }
41
42     // Inicjalizacja Sortable.js
43     window.onload = function() {
44         generateSticks();
45
46         const container =
47             document.getElementById('sticks-container');
48         new Sortable(container, {
49             group: 'sticks', // Grupa, aby umożliwić
50                 przeciąganie między różnymi kontenerami
51             animation: 150, // Animacja przy przesuwaniu
52             dragClass: 'dragging', // Klasa dodawana podczas
53                 przeciągania
54         });
55     };
56 </script>

```

## Listing 1: Mini gra z patyczkami

3.	10,000	10,000 sekund = 2 godziny 46 minut 40 sekund
4.	100,000	1,000,000 sekund = 11 dni 13 godzin 46 minut 40 sekund
5.	1,000,000	100,000,000 sekund = 3 lata 2 miesiące 9 godzin 46 minut 40 sekund
6.	10,000,000	$1 \times 10^{10}$ sekund = 317 lat 1 miesiąc 4 dni 17 godzin 46 minut 40 sekund

Przykład ten uwiadamia problem z wydajnością algorytmów o wyższej złożoności, co jest kluczowe przy obróbce dużych zbiorów danych.

**Złożoność Pamięciowa:**

Oprócz złożoności czasowej, równie istotną kwestią jest złożoność pamięciowa (ang. memory complexity). Określa ona ilość zasobów pamięciowych wymaganych przez algorytm w zależności od liczby przetwarzanych danych. Dobre algorytmy sortujące charakteryzują się niską złożonością pamięciową, co ma duże znaczenie przy dużych zbiorach danych, gdzie brak pamięci RAM może uniemożliwić przetwarzanie danych.

**Algorytmy Sortujące/Nie Sortujące w miejscu:**

Algorytmy sortujące dzielą się na dwie główne grupy:

- **Algorytmy sortujące w miejscu (in place):** Algorytmy, które wymagają stałej liczby dodatkowych struktur danych, niezależnie od liczby elementów. Złożoność pamięciowa w takich algorytmach jest  $O(1)$ .
- **Algorytmy nie sortujące w miejscu:** Algorytmy, które wymagają dynamicznego przydzielania pamięci w zależności od liczby danych wejściowych. Zwykle są one szybsze, ale mają wyższą złożoność pamięciową.

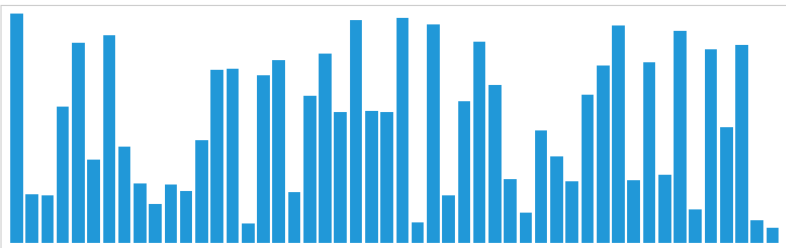
**Algorytmy Stabilne i Niestabilne:**

Algorytmy sortujące dzielimy także na stabilne i niestabilne:

- **Algorytmy stabilne:** Zachowują kolejność elementów równych w zbiorze, co jest istotne w przypadku sortowania danych, gdzie kolejność powtarzających się elementów ma znaczenie, np. w bazach danych.
- **Algorytmy niestabilne:** Nie zachowują kolejności elementów równych, co może prowadzić do zmiany ich względnego położenia w zbiorze po posortowaniu.

Wybór odpowiedniego algorytmu sortującego zależy od konkretnego przypadku zastosowania, wymaganej wydajności oraz dostępnych zasobów.

**Mini gra z patyczkami:**



Rysunek 3: Mini gra na stronie głównej.

Po zalogowaniu użytkownik uzyskuje dostęp do pełnej funkcjonalności aplikacji,

### Wizualne Porównanie Algorytmów Sortowania w Informatyce

#### Rejestracja

#### Logowanie

#### Użytkownicy

ID: 678e1a7c94fc211b663534e4 - pawel\_sipko

ID: 678e221294fc211b66353508 - laba\_adam

ID: 678e261fd4d21619b6cca530 - test

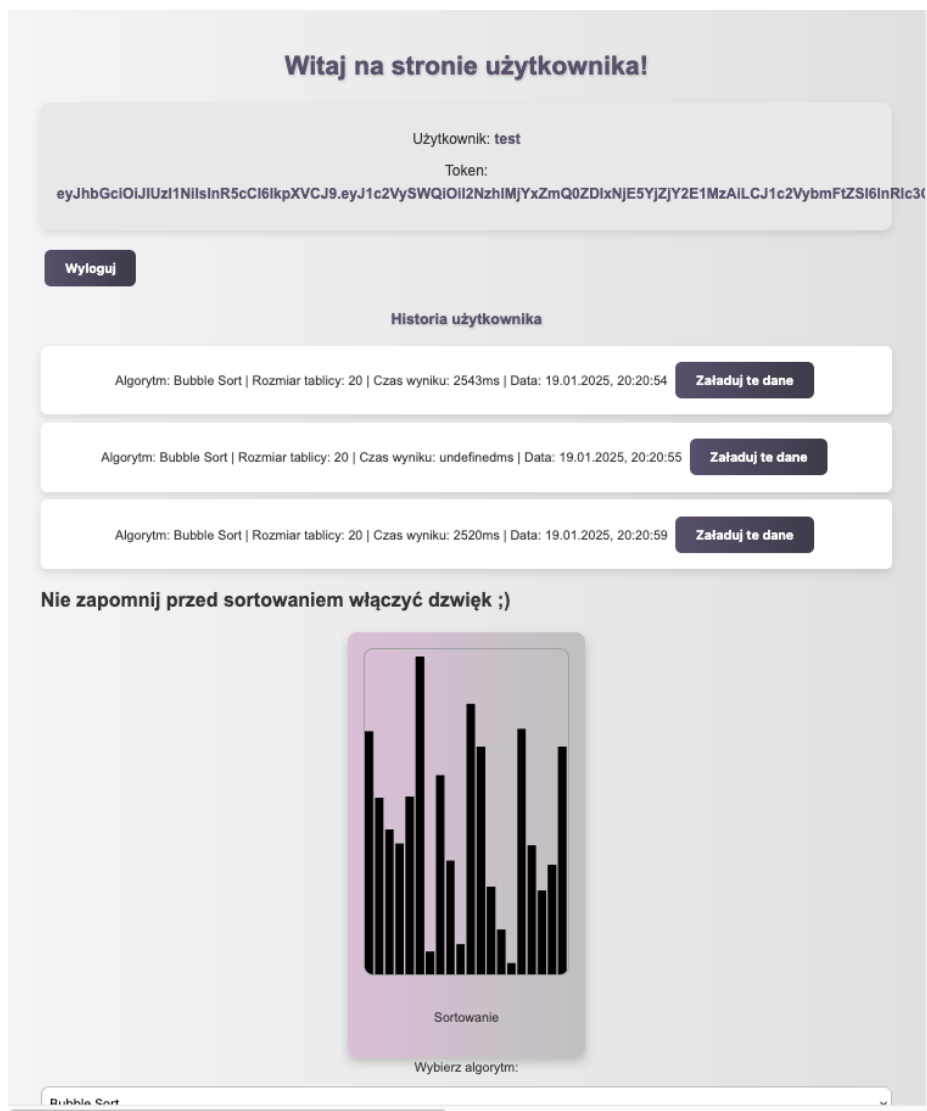
ID: 678e2852d4d21619b6cca53e - julia\_zwierko

#### Aktualizuj użytkownika

Rysunek 4: Panel Rejestracji/Logowania się.

w tym historii poprzednich uruchomień algorytmów. Można tam zobaczyć, jak działały algorytmy przy różnych rozmiarach tablic oraz jakie czasy wykonania zostały osiągnięte.





Rysunek 5: Strona zalogowanego użytkownika

W tym panelu użytkownik ma możliwość wyboru jednego z trzech algorytmów sortowania: sortowania bąbelkowego, sortowania przez wybieranie lub algorytmu Quicksort. Dodatkowo, użytkownik może określić rozmiar tablicy, którą chce posortować.

W projekcie zastosowano również ciekawe dodatki, takie jak dźwięki wydawane podczas sortowania, które dodają interaktywności i atrakcyjności wizualnej. Dźwięki zostały zaimplementowane w języku JavaScript przy pomocy `<script>` i dostosowane do różnych etapów procesu sortowania. Na koniec, użytkownik może zapoznać się z implementacją wybranych algorytmów w różnych językach programowania, takich jak C++, JavaScript, czy Java.

Bubble Sort

Wybierz rozmiar tablicy:

20

Zacznij Sortowanie Resetuj Tablice Zatrzymaj Algorytm

Czas Bubble Sort: - ms  
Czas Selection Sort: - ms  
Czas Quick Sort: - ms

### Podgląd Algorytmów Sortowania

Wybierz algorytm:

Bubble Sort

Wybierz język:

Java

### Podgląd kodu:

```
public static void bubbleSort(int[] arr) {  
    int n = arr.length;  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            if (arr[j] > arr[j + 1]) {  
                int temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
            }  
        }  
    }  
}
```

Rysunek 6: Enter Caption

Implementacje te zostały zamieszczone w formie kodu źródłowego, dzięki czemu użytkownicy mogą poznać szczegóły techniczne oraz różnice między wersjami algorytmów w różnych językach.

### 3 Język HTML5 i style CSS

Projekt opiera się na języku HTML5, który pozwala na tworzenie interaktywnych elementów strony internetowej. Wykorzystano również CSS do stylizacji elementów, takich jak kontener z patyczkami czy formularze. Stylizacja została zaprojektowana w sposób przyjazny dla użytkownika, zapewniając czytelność i intuicyjność interfejsu.

### 4 Grafika w projekcie

Grafika na stronie zalogowanego użytkownika została opracowana z wykorzystaniem elementu `<canvas>` HTML5. Dzięki temu możliwe było stworzenie interaktywnej wizualizacji algorytmów sortowania.

### 5 JavaScript w projekcie

W projekcie użyto kilku kluczowych funkcji JavaScript:

- `getElementById` i `innerHTML` – do manipulacji elementami DOM, umożliwiając pobieranie i wyświetlanie danych na stronie.
- `addEventListener("click")` – do obsługi zdarzeń,
- `window.onload` – do wywołania funkcji po załadowaniu strony, np. generowania elementów wizualnych z funkcji `generateSticks()`.

Te technologie umożliwiają dynamiczne i interaktywne działanie aplikacji.

### 6 Część serwerowa aplikacji

W projekcie użyto Node.js do implementacji części serwerowej. Każdy użytkownik otrzymuje token JWT po zalogowaniu, co pozwala na autentykację i autoryzację dostępu do zasobów. Dane użytkowników są przechowywane w bazie danych MongoDB.

### 7 Ciekawa technologia w projekcie

W projekcie zastosowano bibliotekę D3.js do rysowania wizualizacji algorytmu sortowania bąbelkowego na stronie głównej.

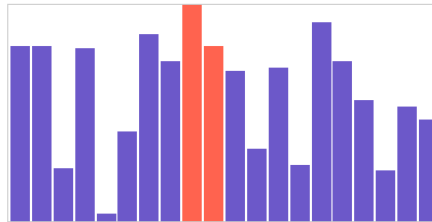
Jeśli chcesz dowiedzieć się więcej o algorytmach i przetestować bardziej zaawansowane przypadki, wróć do [głównej strony](#), zarejestruj swoje konto i wejdź na swoją stronę użytkownika.

## Algorytmy Sortowania w Informatyce

Poniżej przedstawiona jest interesująca wizualizacja działania algorytmu sortowania bąbelkowego (bubble sort) z wykorzystaniem technologii D3.js.

### Bubble Sort - Wizualizacja

Rozpocznij Sortowanie



### Czym Jest Sortowanie?

Sortowanie jest fundamentalnym procesem w informatyce, mającym na celu uporządkowanie zbioru danych według określonego kryterium. Główne zastosowanie sortowania obejmuje usprawnienie późniejszego wyszukiwania danych, co w dużej mierze przyczynia się do zwiększenia efektywności systemów przetwarzania informacji. W kontekście algorytmów sortujących, kluczowe jest rozważenie zarówno ich złożoności czasowej, jak i pamięciowej, które mają bezpośredni wpływ na wydajność operacji na dużych zbiorach danych.

### Czasowa Złożoność Obliczeniowa:

Czasowa złożoność obliczeniowa algorytmu sortującego (ang. computational complexity) to miara ilości czasu potrzebnego na wykonanie algorytmu w zależności od liczby danych wejściowych. Jest to ważne, ponieważ umożliwia ocenę wydajności algorytmu niezależnie od mocy obliczeniowej konkretnego urządzenia, na którym jest on realizowany. Złożoność czasową przedstawia się za pomocą notacji  $O$  (omikron), która charakteryzuje asymptotyczne zachowanie algorytmu przy dużych danych wejściowych.

Wyróżniamy kilka klas złożoności czasowej, w tym:

- $O(n)$ : Algorytmy o liniowej złożoności, w których czas wykonania rośnie proporcjonalnie do liczby przetwarzanych danych.
- $O(n^2)$ : Algorytmy o kwadratowej złożoności, gdzie czas wykonania wzrasta wprost proporcjonalnie do kwadratu liczby danych wejściowych.
- $O(n \log n)$ : Algorytmy o złożoności logarytmicznej, charakteryzujące się wolniejszym wzrostem czasu wykonania w porównaniu do algorytmów o złożoności kwadratowej. Są to algorytmy często wykorzystywane w praktycznych implementacjach sortowania.
- $O(n!)$ : Bardzo pesymistyczne algorytmy, w których czas wykonania rośnie wykładniczo, czyniąc je niewykonalnymi dla dużych zbiorów danych.

W poniższym filmie przedstawiono podstawy algorytmów i struktur danych w kontekście ich złożoności czasowej...

Rysunek 7: Technologia "D3.js".

Dzięki D3.js udało się stworzyć dynamiczną animację, która w sposób wizualny przedstawia działanie tego algorytmu:

```
1 <script src="https://d3js.org/d3.v7.min.js"></script>
2 <script>
3   // Generowanie losowych danych
4   const data = Array.from({ length: 20 }, () =>
5     Math.floor(Math.random() * 100) + 1);
6
7   // Ustawienia SVG
8   const svg = d3.select("svg");
9   const width = +svg.attr("width");
10  const height = +svg.attr("height");
11  const barWidth = width / data.length;
```

```

11
12 // Skale
13 const xScale =
    d3.scaleBand().domain(d3.range(data.length)).range([0,
    width]).padding(0.1);
14 const yScale = d3.scaleLinear().domain([0,
    d3.max(data)]).range([0, height]);
15
16 // Rysowanie początkowych słupków
17 svg.selectAll(".bar").data(data).enter().append("rect")
18   .attr("class", "bar")
19   .attr("x", (d, i) => xScale(i))
20   .attr("y", d => height - yScale(d))
21   .attr("width", xScale.bandwidth())
22   .attr("height", d => yScale(d));
23
24 // Funkcja do aktualizacji wizualizacji
25 function update(data, indices = []) {
26   svg.selectAll(".bar").data(data).join("rect")
27     .attr("class", (d, i) => indices.includes(i) ?
28       "bar highlight" : "bar")
29     .attr("x", (d, i) => xScale(i))
30     .attr("y", d => height - yScale(d))
31     .attr("width", xScale.bandwidth())
32     .attr("height", d => yScale(d));
33 }
34
35 // Sortowanie bąbelkowe z wizualizacją
36 async function bubbleSort(data) {
37   for (let i = 0; i < data.length; i++) {
38     for (let j = 0; j < data.length - i - 1; j++) {
39       // Podświetlenie porównywanych elementów
40       update(data, [j, j + 1]);
41       await new Promise(resolve =>
42         setTimeout(resolve, 300));
43
44       if (data[j] > data[j + 1]) {
45         // Zamiana elementów
46         [data[j], data[j + 1]] = [data[j + 1],
47           data[j]];
48         update(data, [j, j + 1]);
49         await new Promise(resolve =>
50           setTimeout(resolve, 300));
51       }
52     }
53   }
54   // Finalna aktualizacja
55   update(data);
56 }

```

```
53
54 // Uruchamianie sortowania po klikni ciu przycisku
55 document.getElementById("start").addEventListener("click",
56     () => bubbleSort(data));
56 </script>
```