Rzeszów 07.02.2020r.



Przedmiot:

Algorytmika Praktyczna

Temat:

Porównanie wybranych algorytmów sortowania

wykonawca:

Adam Arciszewski 53698

7IIZ

prowadzący:

dr inż. Łukasz Piątek

Spis treści

[1. Wstęp 3](#_Toc32026609)

[1.1. Cel 3](#_Toc32026610)

[2. Algorytmy sortowania 4](#_Toc32026611)

[2.1. Sortowanie bąbelkowe 4](#_Toc32026612)

[2.1.1. Implementacja 4](#_Toc32026613)

[2.1.2. Schemat blokowy 6](#_Toc32026614)

[2.1.3. Złożoność obliczeniowa 6](#_Toc32026615)

[2.2. Sortowanie przez scalanie 7](#_Toc32026616)

[2.2.1. Implementacja 7](#_Toc32026617)

[2.2.2. Schemat blokowy 9](#_Toc32026618)

[2.2.3. Złożoność obliczeniowa 9](#_Toc32026619)

[2.3. Sortowanie szybkie 9](#_Toc32026620)

[2.3.1. Implementacja 10](#_Toc32026621)

[2.3.2. Schemat blokowy 11](#_Toc32026622)

[2.3.3. Złożoność obliczeniowa 11](#_Toc32026623)

[3. Testy algorytmów 12](#_Toc32026624)

[3.1. Implementacja programu testującego: 12](#_Toc32026625)

[3.2. Sortowanie bąbelkowe 12](#_Toc32026626)

[3.3. Sortowanie bąbelkowe 13](#_Toc32026627)

[3.4. Sortowanie przez scalanie 13](#_Toc32026628)

[3.5. Sortowanie szybkie 13](#_Toc32026629)

[4. Wnioski 14](#_Toc32026630)

[4.1. Sortowanie bąbelkowe 14](#_Toc32026631)

[4.2. Sortowanie przez scalanie 14](#_Toc32026632)

[4.3. Sortowanie szybkie 14](#_Toc32026633)

# Wstęp

Projekt realizowany będzie w oparciu o program napisany w języku programowania C# (.NET Core). Program będzie składał się z:

* Implementacji wybranych algorytmów sortowania
* Testach jednostkowych wybranych algorytmów
* Programu sprawdzającego wydajność wybranych algorytmów.

Kod źródłowy programu jest dostępny w publicznym repozytorium GitHub pod adresem:

<https://github.com/Adasq01/AlgorytmikaPraktyczna>

## Cel

Celem projektu jest implementacja oraz porównanie działania wybranych algorytmów sortowania: sortowania bąbelkowego, sortowania przez scalanie oraz sortowania szybkiego.

# Algorytmy sortowania

## Sortowanie bąbelkowe

Sortowanie bąbelkowe polega na porównaniu dwóch kolejnych elementów i zamianie ich w kolejności odpowiedniej do kierunku sortowania. Sortowanie kończy się gdy nie dokonano już żadnej zmiany.

### Implementacja

Implementacja znajduje się w projekcie SortowanieDanych, plik BubbleSorter.cs

/// <summary>

/// Sortowanie bąbelkowe tablicy int

/// </summary>

public static int[] BubbleSort(this int[] array, SortDirection sortDirection =

SortDirection.Ascending)

{

// 1 operacja +

if (array.Length < 2)

return array;

// 1 operacja +

var n = array.Length;

// n-1 \* (operacje wewnątrz pętli)

do

{

//n-1 \* (operacje wewnatrz petli)

for (var i = 0; i < n - 1; i++)

{

// 2 operacje

if (sortDirection == SortDirection.Ascending

? array[i] < array[i + 1]

: array[i] > array[i + 1]) continue;

//3 operacje

var tmp = array[i];

array[i] = array[i + 1];

array[i + 1] = tmp;

//MAX 5 operacji

}

// + 1 operacja

n--;

}

while (n > 1);

// f(n) = 1 + 1 +(n-1)\*((n-1)\*5+1) = 2 + 5(n-1)^2 + (n-1)

// Złożoność obliczeniowa = O(n^2)

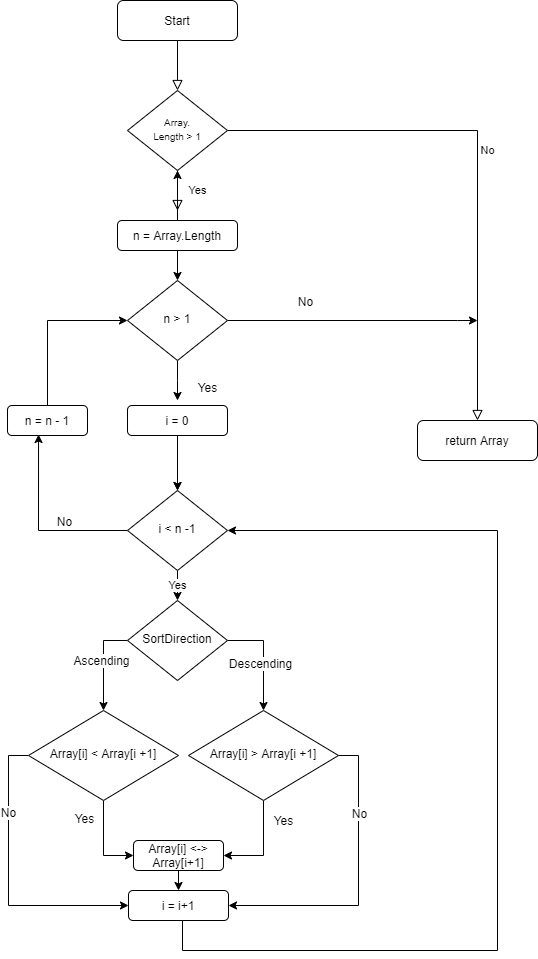
return array;

}

Implementacja algorytmu sortowania bąbelkowego sprowadza się do wykorzystania dwóch pętli (jedna pętla zagnieżdżona). Pierwszy warunek sprawdza, czy tablica nie jest pusta lub jednoelementowa (taka tablica z zasady jest już posortowana). Następnie jest wykonywana pętla do… while n-1 razy, gdzie n = liczba elementów w tablicy. W ciele tej pętli wykonywana jest kolejna pętla (for), która również jest wykonywana n-1 razy. W owym zagnieżdżeniu, procedura wykonuje porównanie dwóch kolejnych elementów, zaczynając od pierwszego (index zerowy). W zależności od kierunku sortowania (rosnąco lub malejąco), jeśli element jest większy lub mniejszy od następnego elementu, następuje zamiana owych elementów w tablicy (lub nie).

Następnie całość jest powtarzana zgodnie z pętlą while.

### Schemat blokowy



### Złożoność obliczeniowa

## Sortowanie przez scalanie

Sortowanie przez scalanie wykorzystuje metodę „dziel i zwyciężaj”. Opiera się on na definicji, że dwie posortowane tablice, można scalić porównując kolejno elementy obu tablic zaczynając od pierwszego i wrzucając do tablicy „scalonej” element zgodny z kierunkiem sortowania, po czym zostaje zwiększony licznik tablicy z której element został zabrany i następuje dalsze porównywanie. Takie założenie jest możliwe ze względu na zasadę, że tablica jednoelementowa jest już posortowana, a więc tablica X elementową można dzielić na pół i sortować, aż do momentu uzyskania posortowanych jednoelementowych tablic.

### Implementacja

Implementacja znajduje się w projekcie SortowanieDanych, plik MergeSorter.cs

public static int[] MergeSort(this int[] array, SortDirection sortDirection =

SortDirection.**Ascending**)

{

return array.Sort(0, array.Length - 1, sortDirection);

}

static int[] Sort(this int[] array, int left, int right, SortDirection direction)

{

var helperArray = new int[array.Length];

if (left >= right) return array;

//podziel sortowana tablice na pol

array.Sort(left, (left + right) / 2, direction);

array.Sort((left + right) / 2 + 1, right, direction);

//scal posortowane tablice

array.Merge(helperArray, left, right, direction);

return array;

}

static void Merge(this int[] array, int[] helperArray, int left, int right, SortDirection direction)

{

//Skopiuj wartosci do tablicy pomocniczej

for (var **i** = left; **i** <= right; **i**++)

{

helperArray[**i**] = array[**i**];

}

//Scalaj tablice

var **p** = left;

var **q** = (left + right) / 2 + 1;

var **r** = left;

while (**p** <= (left + right) / 2 && **q** <= right)

{

if (direction == SortDirection.**Descending**

? helperArray[**p**] > helperArray[**q**]

: helperArray[**p**] < helperArray[**q**])

{

array[**r**] = helperArray[**p**];

**r**++;

**p**++;

}

else

{

array[**r**] = helperArray[**q**];

**r**++;

**q**++;

}

}

//Przepisz koncowke

while (**p** <= (left + right) / 2)

{

array[**r**] = helperArray[**p**];

**r**++;

**p**++;

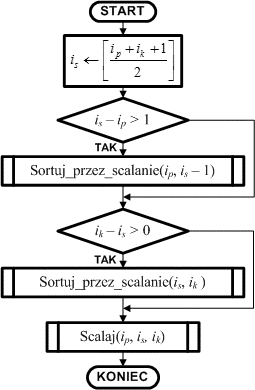
}

}

}

Do implementacji metody sortowania poprzez scalanie wykorzystana została rekurencyjna metoda Sort. Czyli metoda, która w swoim ciele wywołuje samą siebie. Aby nie doszło do pętli nieskończonej, metoda musi dążyć do skończonego warunku, który przerwie rekurencję.

### Schemat blokowy



### Złożoność obliczeniowa

## Sortowanie szybkie

Sortowanie szybkie, tak jak sortowanie przez scalanie, opiera się o metodę „dziel i zwyciężaj”. Zbiór danych zostaje podzielony na dwa zbiory i każdy z nich jest sortowany niezależnie. Do podziału wykorzystywany jest element rozdzielający (element środkowy tablicy), następnie elementy z lewej i prawej strony są z nim porównywane i przenoszone do odpowiedniej tablicy (lewa – prawa) zgodnie z kierunkiem sortowania. Potem następuje rekurencja dla obu stron, która kończy się aż do podziału tablicy na jednoelementową (nie wymagającej sortowania).

### Implementacja

Implementacja znajduje się w projekcie SortowanieDanych, plik QuickSorter.cs

public static void QuickSort(this int[] array, SortDirection sortDirection = SortDirection.**Ascending**)

{

array.Sort(0, array.Length - 1, sortDirection);

}

static void Sort(this int[] array, int left, int right, SortDirection direction)

{

var **i** = left;

var **j** = right;

//wartość środkowa

var pivot = array[(left + right) / 2];

while (**i** < **j**)

{

while (direction == SortDirection.**Ascending**

? array[**i**] < pivot

: array[**i**] > pivot) **i**++;

while (direction == SortDirection.**Ascending**

? array[**j**] > pivot

: array[**j**] < pivot) **j**--;

if (**i** > **j**) continue;

var tmp = array[**i**];

array[**i**++] = array[**j**];

array[**j**--] = tmp;

}

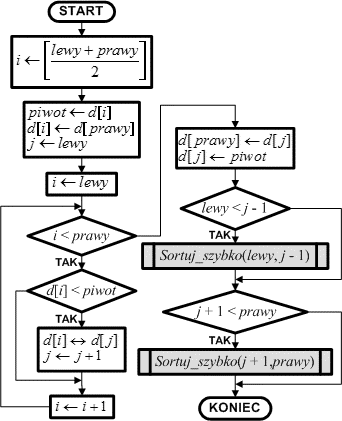
if (left < **j**) Sort(array, left, **j**, direction);

if (**i** < right) Sort(array, **i**, right, direction);

}

Implementacja sortowania szybkiego również opiera się o wykorzystanie rekurencji. Tablica z każdym wywołaniem funkcji rekurencyjnej jest dzielona na coraz to mniejsze połowy, a następnie elementy są porównywane ze zmienną pivot (element środkowy po podzieleniu tabeli) i zamieniane jeśli spełniony zostaje warunek dla odpowiedniego kierunku sortowania.

### Schemat blokowy



### Złożoność obliczeniowa

W przypadku algorytmu sortowania szybkiego, złożoność obliczeniową tego algorytmu należy rozpatrywać w dwóch wariantach:

Optymistyczna:

Oraz pesymistyczny:

Wynika to z faktu określania elementu środkowego, jeśli uda nam się trafić w medianę sortowanego fragmentu tablicy, przypadek będzie optymistyczny. Jednak jeśli trafimy na element najmniejszy (lub największy), to zachodzi przypadek pesymistyczny.

Można zoptymalizować algorytm poprzez wykorzystanie algorytmu wyszukiwania mediany, wówczas pesymistyczna złożoność będzie równa optymistycznej.

# Testy algorytmów

Testy zostały wykonane wykorzystując prosty program konsolowy, który dla wybranego algorytmu sortowania, będzie tworzył tablice o różnej wielkości, losowo przypisanych elementów, a następnie wykonywał operację sortowania na tym zbiorze.

Testy oczywiście były wykonywane na tej samej jednej maszynie.

## Implementacja programu testującego:

Program znajduje się w projekcie AlgorytmikaPraktyczna (Program.cs).

private static void TestSort(int repeat, Action<int[]> sort)

{

var **n** = 100;

for (var **i** = 1; **i** <= repeat; **i**++)

{

var intArray = new int[**n**];

var random = new Random(1001);

for (var **j** = 0; **j** < **n**; **j**++)

{

intArray[**j**] = random.Next();

}

var watch = new Stopwatch();

watch.Restart();

sort(intArray);

watch.Stop();

Console.WriteLine($"Sortowanie {**n**} elementów zajęło: {watch.Elapsed.TotalSeconds:F} sekund, {watch.Elapsed.TotalMilliseconds:F} milisekund))");

**n** \*= 2;

}

}

## Sortowanie bąbelkowe

Przykład użycia metody TestSort

Console.WriteLine("Sortowanie bąbelkowe.\n");

TestSort(**n**, array => array.BubbleSort());

Console.WriteLine("Zakończono. Wybierzk kolejne dzialanie.\n");

Console.WriteLine("Sortowanie przez scalanie.\n");

TestSort(**n**, array => array.MergeSort());

Console.WriteLine("Zakończono. Wybierzk kolejne dzialanie.\n");

Console.WriteLine("Sortowanie szybkie.\n");

TestSort(**n**, array => array.QuickSort());

Console.WriteLine("Zakończono. Wybierzk kolejne dzialanie.\n");

Metoda testująca przyjmuje początkową wartość zbioru 100 elementowego, tworzy zbiór inicjując wartości jego elementów przy użyciu klasy Random(). W konstruktorze klasy Random podana jest wartość int Seed, dzięki której za każdym razem otrzymujemy ten sam losowy zbiór (aby badanie różnych metod sortowania odbywało się zawsze na tym samym zbiorze). Ustawiany jest zegar liczący ile czasu trwało sortowanie. Po zakończeniu sortowania operacja jest powtarzana podaną przez użytkownika liczbę razy, z każdą powtórka liczba elementów w zbiorze wzrasta dwukrotnie.

Testy były powtarzane dla tych samych oraz różnych seedów w celu wyciągnięcia średnich czasów wykonywania.

## Sortowanie bąbelkowe

Sortowanie 100 elementów zajęło: 0,00 sekund, 0,04 milisekund

Sortowanie 200 elementów zajęło: 0,00 sekund, 0,09 milisekund

Sortowanie 400 elementów zajęło: 0,00 sekund, 0,38 milisekund

Sortowanie 800 elementów zajęło: 0,00 sekund, 2,07 milisekund

Sortowanie 1600 elementów zajęło: 0,01 sekund, 6,87 milisekund

Sortowanie 3200 elementów zajęło: 0,03 sekund, 28,50 milisekund

Sortowanie 6400 elementów zajęło: 0,12 sekund, 122,25 milisekund

Sortowanie 12800 elementów zajęło: 0,52 sekund, 516,47 milisekund

Sortowanie 25600 elementów zajęło: 2,10 sekund, 2100,48 milisekund

Sortowanie 51200 elementów zajęło: 8,32 sekund, 8325,00 milisekund

Sortowanie 102400 elementów zajęło: 33,43 sekund, 33425,12 milisekund

## Sortowanie przez scalanie

Sortowanie 100 elementów zajęło: 0,00 sekund, 0,06 milisekund

Sortowanie 200 elementów zajęło: 0,00 sekund, 0,09 milisekund

Sortowanie 400 elementów zajęło: 0,00 sekund, 0,30 milisekund

Sortowanie 800 elementów zajęło: 0,00 sekund, 0,77 milisekund

Sortowanie 1600 elementów zajęło: 0,00 sekund, 1,32 milisekund

Sortowanie 3200 elementów zajęło: 0,00 sekund, 4,94 milisekund

Sortowanie 6400 elementów zajęło: 0,01 sekund, 14,98 milisekund

Sortowanie 12800 elementów zajęło: 0,07 sekund, 66,76 milisekund

Sortowanie 25600 elementów zajęło: 0,37 sekund, 369,93 milisekund

Sortowanie 51200 elementów zajęło: 1,21 sekund, 1205,60 milisekund

Sortowanie 102400 elementów zajęło: 4,72 sekund, 4723,47 milisekund

Sortowanie 204800 elementów zajęło: 16,61 sekund, 16611,15 milisekund

Sortowanie 409600 elementów zajęło: 67,99 sekund, 67992,91 milisekund

## Sortowanie szybkie

Sortowanie 100 elementów zajęło: 0,00 sekund, 0,03 milisekund

Sortowanie 200 elementów zajęło: 0,00 sekund, 0,04 milisekund

Sortowanie 400 elementów zajęło: 0,00 sekund, 0,05 milisekund

Sortowanie 800 elementów zajęło: 0,00 sekund, 0,11 milisekund

Sortowanie 1600 elementów zajęło: 0,00 sekund, 0,25 milisekund

Sortowanie 3200 elementów zajęło: 0,00 sekund, 0,50 milisekund

Sortowanie 6400 elementów zajęło: 0,00 sekund, 0,83 milisekund

Sortowanie 12800 elementów zajęło: 0,00 sekund, 1,61 milisekund

Sortowanie 25600 elementów zajęło: 0,00 sekund, 3,80 milisekund

Sortowanie 51200 elementów zajęło: 0,01 sekund, 6,85 milisekund

Sortowanie 102400 elementów zajęło: 0,01 sekund, 14,63 milisekund

Sortowanie 204800 elementów zajęło: 0,03 sekund, 30,40 milisekund

Sortowanie 409600 elementów zajęło: 0,06 sekund, 64,65 milisekund

Sortowanie 819200 elementów zajęło: 0,13 sekund, 133,45 milisekund

Sortowanie 1638400 elementów zajęło: 0,28 sekund, 279,00 milisekund

Sortowanie 3276800 elementów zajęło: 0,58 sekund, 575,44 milisekund

Sortowanie 6553600 elementów zajęło: 1,20 sekund, 1197,41 milisekund

Sortowanie 13107200 elementów zajęło: 2,50 sekund, 2498,52 milisekund

Sortowanie 26214400 elementów zajęło: 5,18 sekund, 5184,08 milisekund

Sortowanie 52428800 elementów zajęło: 10,72 sekund, 10724,77 milisekund

# Wnioski

## Sortowanie bąbelkowe

Najwolniejszy algorytm ze wszystkich 3 porównywanych. Jego złożoność logarytmiczna zawsze wynosi . Nadaje się tylko do sortowania niewielkich zbiorów (do kilku tysięcy elementów).

## Sortowanie przez scalanie

Sortowanie przez scalanie miało być idealnym sortowaniem, które ustępowałoby szybkiemu tylko w początkowej fazie, kiedy operacja sortowania była wykonywana na mniejszych zbiorach. Aktualna implementacja sortowania przez scalanie nadawała się do sortowania ok. 100tysiącach elementów, później zaczynała się dłużyć w czasie. Z racji na stałą złożoność logarytmiczną, równą optymistycznemu wariantowi złożoności sortowania szybkiego, zakładam, że może to być problem z implementacją samego sortowania.

## Sortowanie szybkie

Sortowanie szybkie odnosiło sukcesy nawet dla sortowania przy wielomilionowych zbiorach (10 310 720 elementów w 2,5 sekundy). Niestety nie udało się zaimplementować testu, który odtwarzałby wariant pesymistyczny złożoności sortowania szybkiego dla porównania.

Sortowanie szybkie jest sortowaniem popularnie wykorzystywanym w językach programowania, jest on np. wykorzystywany przez .NET’owe LINQ (w przypadku gdy LINQ jest używane w LINQ to Objects).   
W ramach ciekawostki, wykonałem porównanie zaimplementowanego przeze mnie sortowania szybkiego, a sortowaniem w LINQ.

Dla 10 310 720 elementów, sortowanie LINQ potrzebowało 4,59 sekundy, więc o 2 sekundy dłużej niż podstawowa implementacja sortowania szybkiego. Można jednak przypuszczać, że jest to spowodowane szerszym zastosowaniem sortowania LINQ – albowiem przystosowane ono jest do sortowania nie tylko tablic liczb naturalnych, ale także do wszelkiego typu innych struktur, w tym struktur złożonych.