1 Krótkie wprowadzenie teoretyczne

1.1 Quicksort

Quicksort, czyli sortowanie szybkie, działa na zasadzie wybierania kotwicy, czyli dowolnej liczby, i umieszczanie jej na właściwym miejscu przerzucając wszystkie wartości większe od niej na prawo, a mniejsze na lewo metodą zamian. Uzyskujemy w ten sposób dwie partycje, na których rekurencyjnie wywołujemy quicksort, aż nie uzyskamy partycji z jednym elementem, która nie wymaga sortowania.

W mojej implementacji kotwica wybierana jest ze środka partycji.

Teoretyczna średnia złożoność obliczeniowa: $O(n \log n)$

1.2 Shell sort

Shell sort to algorytm sortowania danych, które wykorzystuje fakt, że częściowo posortowane ciągi łatwiej się sortuje przez niektóre algorytmy. W tym celu używa ciągów odstępów, od których zależy złożoność algorytmu. Na każdy element ciągu wykonywana jest seria porównań i zamian liczb oddalonych o wielkość przerwy.

Wybrane ciągi odstępów: ciąg Shella ($\left\lfloor \frac{n}{2^k} \right\rfloor$) oraz ciąg Franka-Lazarusa ($2 \left\lfloor \frac{n}{2^{k+1}} \right\rfloor$), gdzie n to rozmiar sortowanej tablicy, a k to numer iteracji.

Teoretyczna złożoność obliczeniowa (Shell): $\theta(n^2)$

Teoretyczna złożoność obliczeniowa (Frank-Lazarus): $\theta(n^{\frac{3}{2}})$

1.3 Merge sort

Merge sort to rekurencyjny algorytm sortowania danych, stosujący metodę dziel i zwyciężaj. Polega on na podzieleniu danych na dwie równe części i wywołanie merge sort'a dla każdej z nich, dopóki nie zostanie jeden element w każdej części. Następnie scalamy posortowane podciągi w ciąg posortowany.

Teoretyczna złożoność obliczeniowa: $O(n \log n)$

2 Opis schematu i przebiegu badań

2.1 Sposób generowania ciągów

Ciągi są generowane dla każdej kombinacji długość-typ ciągu. Jeżeli wykryjemy, że ciągi dla danej kombinacji są już wygenerowane, możemy pominąć zapisywanie danej kombinacji. Ponieważ generacja i zapisywanie trochę trwa, dla danej kombinacji rozdzielam zapisywanie i generację na wszystkie rdzenie procesora.

Konkretna generacja ciągów odbywa się poprzez losowanie danej ilości liczb w przedziale $< int.Min; int.Max)$, a następnie posortowanie ich Quicksortem, jeśli jest taka potrzeba (w zależności od typu ciągu).

```
private static void GenerateAndSaveAll(AppData appData, Logger logger)
{
    foreach (SequenceLength sequenceLength in Enum.GetValues(typeof(SequenceLength)))
    {
        foreach (SequenceType sequenceType in Enum.GetValues(typeof(SequenceType)))
        {
            if (appData.AreNumbersAlreadySaved(sequenceLength, sequenceType))
            {
                logger.Log($"{sequenceLength.Description()} {sequenceType.Description()} - already written - skipping.")
                continue;
            }

            //Partition generation and saving into all cores
            //MaxDegreeOfParallelism is needed, because we perform many IO operations
            Parallel.For(0, 100, new ParallelOptions {MaxDegreeOfParallelism = Environment.ProcessorCount}, async i =>
                {
                    int[] arr = GenerateNumbers(sequenceLength, sequenceType);
                    await appData.SaveNumbers(string.Join(' ', arr), i, sequenceLength, sequenceType);
                }
            );
            logger.Log($"{sequenceLength.Description()} {sequenceType.Description()} - writing finished.");
        }
    }
}
```

Rysunek 1: Generacja i zapisywanie ciągów

```
private static int[] GenerateNumbers(SequenceLength sequenceLength, SequenceType sequenceType)
{
    var rng = new RandomNumbersGenerator(MinGeneratedValue, MaxGeneratedValue);
    int[] arr = rng.Generate((int) sequenceLength);
    switch (sequenceType)
    {
        case SequenceType.HalfSorted:
            int[] firstHalf = arr.Take(arr.Length / 2).ToArray();
            int[] secondHalf = arr.Skip(arr.Length / 2).ToArray();
            SortingAlgorithms[0].Sort(ref firstHalf);
            arr = firstHalf.Concat(secondHalf).ToArray();
            break;
        case SequenceType.Sorted:
        case SequenceType.ReverseSorted:
            SortingAlgorithms[0].Sort(ref arr);
            break;
    }

    if (sequenceType == SequenceType.ReverseSorted) arr = arr.Reverse().ToArray();
    return arr;
}
```

Rysunek 2: Generacja konkretnego ciągu

2.2 Obciążenie systemu i działania na celu wyrównania

Ponieważ ten program jest konsolowy, mogę zminimalizować obciążenie systemu uruchamiając symulację na Linuxie bez środowiska graficznego. Aby dać każdemu algorytmowi taką samą szansę, mierzenie czasu sortowania odbywa się synchronicznie, na jednym rdzeniu procesora.

2.3 Sposób pomiaru czasu i organizacja funkcji testowych

Na każdy algorytm, na każdą długość ciągu, na każdy typ ciągu tworzę tablicę czasów dla każdego ze 100 wygenerowanych ciągów danej długości-typu. Następnie na każdy wygenerowany ciąg wczytuję go do pamięci, a następnie mierzę czas sortowania tego ciągu i zapisuję go do tablicy. Po wypełnieniu tablicy czasów zapisuję wyniki na dysku oddzielone spacją w pliku dla danego algorytmu, długości ciągu i typu ciągu.

```
foreach (ISortingAlgorithm algorithm in SortingAlgorithms)
{
    foreach (SequenceLength sequenceLength in Enum.GetValues(typeof(SequenceLength)))
    {
        foreach (SequenceType sequenceType in Enum.GetValues(typeof(SequenceType)))
        {
            Console.WriteLine($"Simulating {algorithm.Name} - {sequenceLength} - {sequenceType}");
            var times = new double[100];
            for (int i = 0; i < 100; ++i)
            {
                int[] numbers = appData.GetNumbers(i, sequenceLength, sequenceType);
                double timeInSeconds = TimeMeasurer.Measure(() => algorithm.Sort(ref numbers)).TotalSeconds;
                times[i] = timeInSeconds;
            }

            appData.SaveResult(string.Join(' ', times), algorithm, sequenceLength, sequenceType).Wait();
        }
    }
}
```

Rysunek 3: Organizacja funkcji

```
public static class TimeMeasurer
{
    private static readonly Stopwatch Sw = new Stopwatch();

    public static TimeSpan Measure(Action a)
    {
        Sw.Reset();
        Sw.Start();
        a();
        Sw.Stop();

        return Sw.Elapsed;
    }

}
```

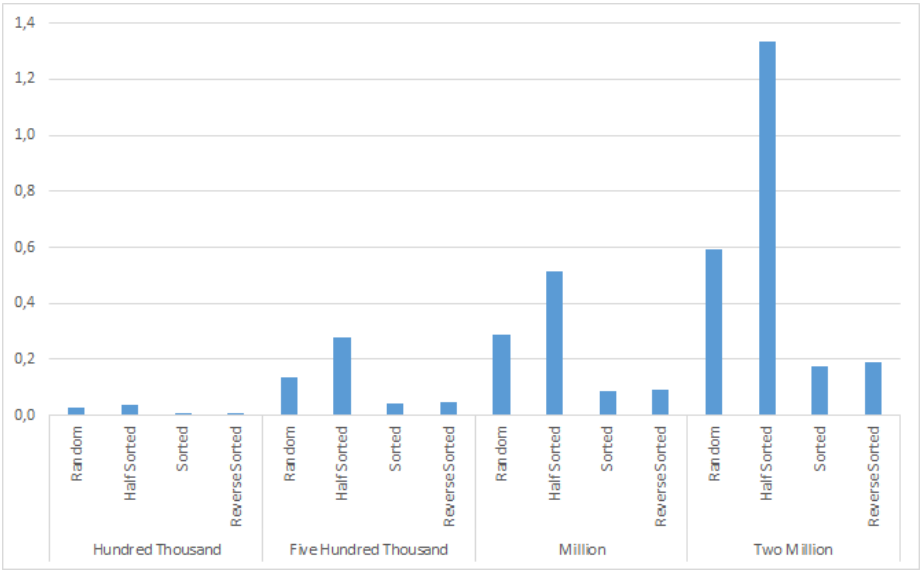
Rysunek 4: Klasa odpowiadająca za mierzenie czasu

3 Prezentacja wyników

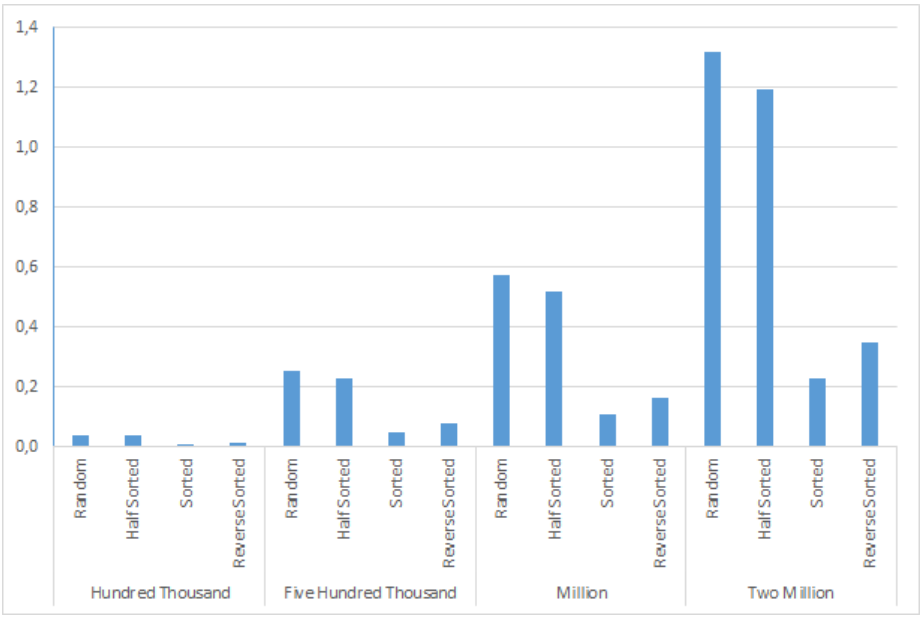
Wykresy przedstawiają średni czas posortowania danego losowego ciągu w sekundach.

	Hundred Thousand				Five Hundred Thousand				Million				Two Million			
	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted
avg	0.0254	0.0357	0.0079	0.0086	0.1367	0.2763	0.0412	0.0455	0.2872	0.5164	0.0852	0.0924	0.5943	1.3322	0.1757	0.1894
standard deviation	0.0006	0.0184	0.0003	0.0003	0.0028	0.2260	0.0008	0.0015	0.0055	0.3960	0.0028	0.0016	0.0115	1.2380	0.0034	0.0046

Tabela 1: Wartości średnie i odchylenie standardowe dla Quicksort'a



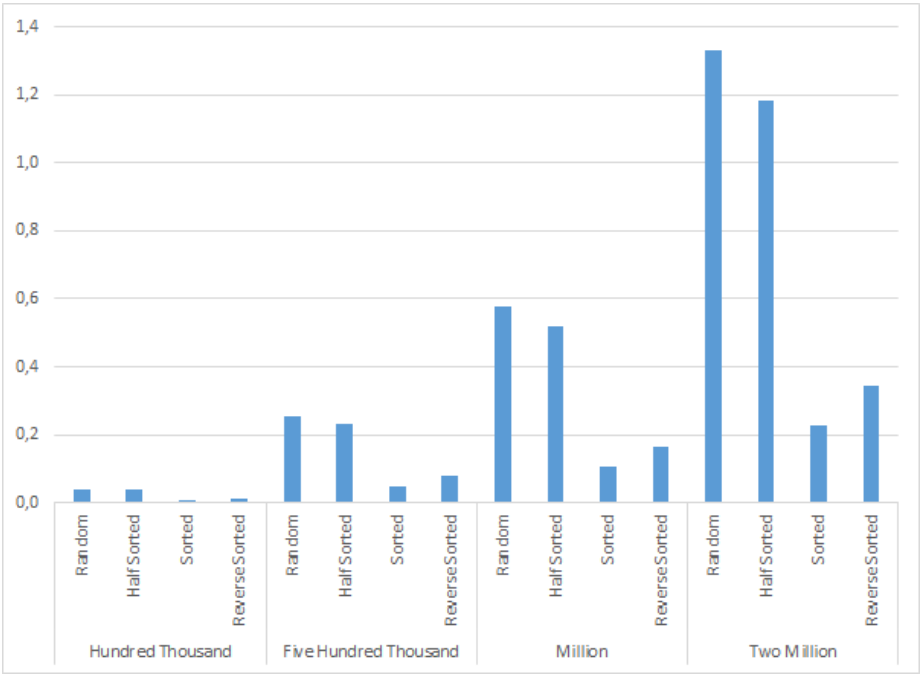
Rysunek 5: Quicksort



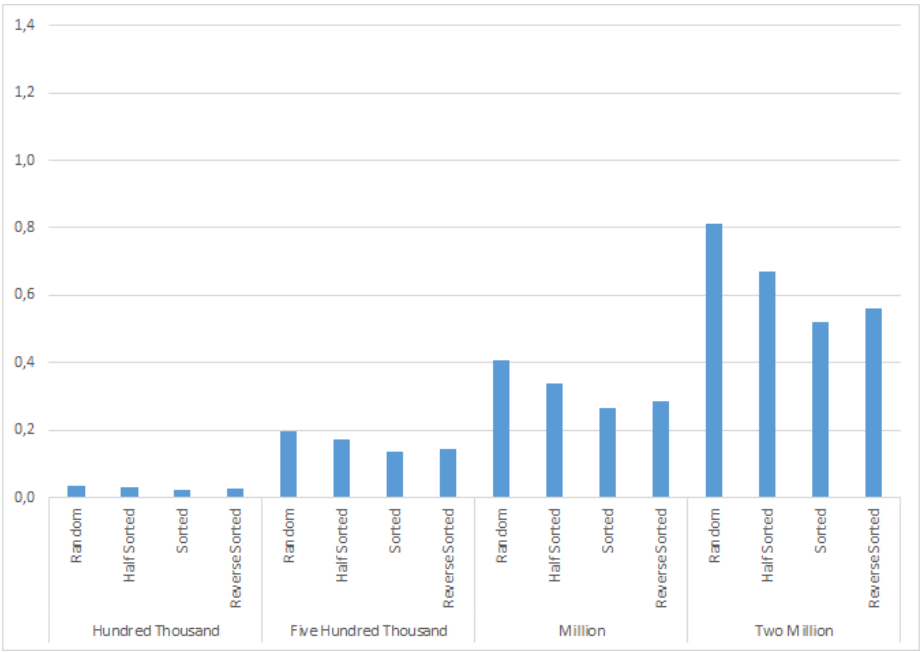
Rysunek 6: Shell sort (Shell)

	Hundred Thousand				Five Hundred Thousand				Million				Two Million			
	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted
avg	0.0407	0.0374	0.0089	0.0139	0.2510	0.2285	0.0501	0.0773	0.5725	0.5185	0.1066	0.1612	1.3172	1.1954	0.2284	0.3464
standard deviation	0.0018	0.0012	0.0003	0.0003	0.0047	0.0036	0.0013	0.0032	0.0144	0.0148	0.0053	0.0054	0.0411	0.0345	0.0050	0.0078

Tabela 2: Wartości średnie i odchylenie standardowe dla Shell sort'a (Shell)



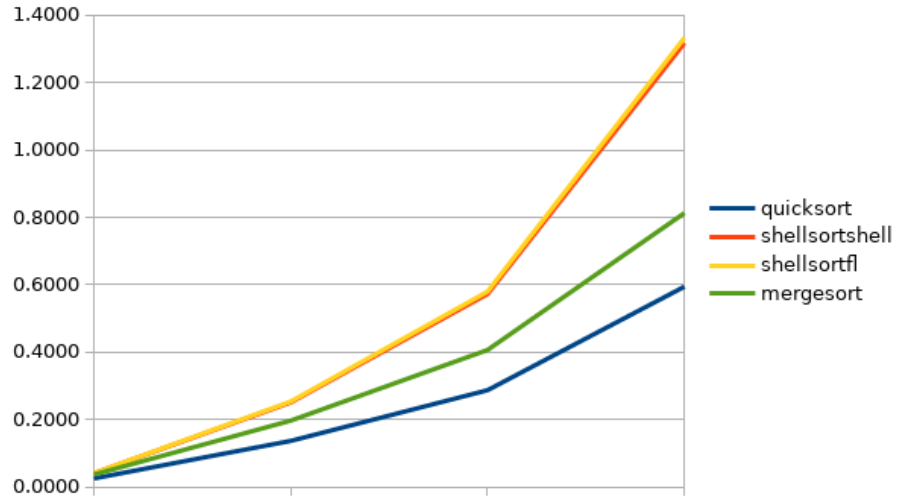
Rysunek 7: Shell sort (Frank-Lazarus)



Rysunek 8: Merge sort

4 Analiza wyników

Można zauważyć relatywnie niskie wartości odchyleń standardowych z wyjątkiem Quicksort'a dla ciągów posortowanego w połowie. Dla takich ciągów Qu-



Rysunek 9: Wykres złożoności sortowań losowych ciągów w zależności od długości ciągu

	Hundred Thousand				Five Hundred Thousand				Million				Two Million			
	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted
avg	0.0410	0.0375	0.0090	0.0138	0.2535	0.2311	0.0500	0.0774	0.5796	0.5179	0.1056	0.1626	1.3325	1.1853	0.2252	0.3447
standard deviation	0.0015	0.0009	0.0002	0.0005	0.0072	0.0050	0.0022	0.0017	0.0169	0.0111	0.0018	0.0032	0.0431	0.0315	0.0050	0.0067

Tabela 3: Wartości średnie i odchylenie standardowe dla Shell sort'a (Frank-Lazarus)

	Hundred Thousand				Five Hundred Thousand				Million				Two Million			
	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted	Random	Half Sorted	Sorted	Reverse Sorted
avg	0.0369	0.0312	0.0249	0.0267	0.1970	0.1725	0.1355	0.1434	0.4066	0.3376	0.2645	0.2856	0.8127	0.6713	0.5196	0.5605
standard deviation	0.0025	0.0022	0.0022	0.0024	0.0095	0.0100	0.0083	0.0078	0.0119	0.0115	0.0121	0.0117	0.0152	0.0140	0.0199	0.0152

Tabela 4: Wartości średnie i odchylenie standardowe dla Merge sort'a

icksort generalnie radzi sobie gorzej, ale nie gorzej niż $O(n^2)$, co jest teoretyczną złożonością pesymistyczną tego algorytmu.

Wykres złożoności wygląda na prawidłowy. Generalnie Quicksort jest nieco szybszy od Merge sort'a, a wybrane implementacje Shell sort'a sprawują się najgorzej z nich. Trochę zastanawia podobieństwo złożoności Shell sort'a dla różnych ciągów odstępów, gdyż teoretycznie ciąg odstępów Franka i Lazarusa powinien się sprawować nieco lepiej.

5 Wnioski

Po analizie danych można się upewnić, że wybór algorytmu sortowania nie zawsze jest prostym zadaniem. Generalnie Quicksort radzi sobie najlepiej, ale jeśli wiemy, że chcemy posortować już częściowo posortowany ciąg, Merge sort mógłby się okazać lepszy do tego zadania.