Badanie złożoności obliczeniowej różnych algorytmów sortowania

Daniel Śliwowski

nr ideksu: 241166, Termin: ŚR 11:15, Data: 30.03.2019,
Prowadzący: Dr inż. Łukasz Jeleń

1 Cel badania

Celem badania jest eksperymentalne sprawdzenie złożoności obliczeniowej trzech algorytmów sortowania: sortowania szybkigo, przez scalanie oraz introspektywnego, a następnie porówanananie otrzymanych wyników do złożoności obliczeniowej otrzymanej analitycznie. Każdy z alorytmów zostal przetestowany dla kilku zbiorów danych różniących się ilością liczb oraz procentem posortowania. Tablice zawierają liczby całkowite i sortowane są w kolejności rosnącej.

2 Sortowanie przez scalanie

Sortowanie przez scalanie jest algorytmem typu dziel i zwyciężaj. Zakłada podział zbioru danych na dwie identyczne w długości części i posortowania ich za pomocą scalania. Zaimplementowany algorytm ma postać:

Data: Nieposortowania Tablica A
Result: Posortowana w rosnącej kolejności tablica A
begin
    $n1 \longleftarrow n2 \longleftarrow 0$
    $size \longleftarrow$ rozmiar talbicy A
    if $size \leqslant 1$ **then**
        return
    end
    if $size \ \% \ 2 \neq 0$ **then**
        $n1 \longleftarrow size/2$
        $n1 \longleftarrow n1 + 1$
    else
        $n1 \longleftarrow n2 \longleftarrow size/2$
    end
    $A_1 \longleftarrow$ tablica o rozmiarze n1
    $A_2 \longleftarrow$ tablica o rozmiarze n2
    $i \longleftarrow 0$
    $j \longleftarrow n1$
    for $i < n1$ **do**
        $A_1[i] \longleftarrow A[i]$
        $i \longleftarrow i + 1$
    end
    for $j < size$ **do**
        $A_2[j - n1] \longleftarrow A[j]$
        $j \longleftarrow j + 1$
    end
    MergSort(A_1)
    MergSort(A_2)
    Merge(A, A_1, A_2)
end

Algorytm 1: MergSort(A)

Funkcja Merge(A, A_1, A_2) ma postać:
Data: A - tablica do której łączymy, A_1 i A_2 - tablice które łaczymy
Result: A - Tablica otrzymana w wyniku połączenia A_1 i A_2

```
begin
    i ⟵ j ⟵ k ⟵ 0
    while i < A1.size() and j < A1.size() do
        if A1[i] < A2[j] then
            A[k] ⟵ A1[i]
            i ⟵ i + 1
        else
            A[k] ⟵ A2[j]
            j ⟵ j + 1
        end
        k ⟵ k + 1
    end
    while i < A1.size() do
        A[k] ⟵ A1[i]
        i ⟵ i + 1
        k ⟵ k + 1
    end
    while j < A2.size() do
        A[k] ⟵ A2[j]
        j ⟵ j + 1
        k ⟵ k + 1
    end
end
```

Algorytm 2: Merge(A, A_1, A_2)

Przewidywana złożoność obliczeniowa w przypadku średnim i najgorszym wynosi $O(nlog(n))$. Wynika to z tego, że głęgokość rekurencji wynosi $log_2(n)$ i na każdym poziomie złożoność scalania wynosi około $O(n)$.

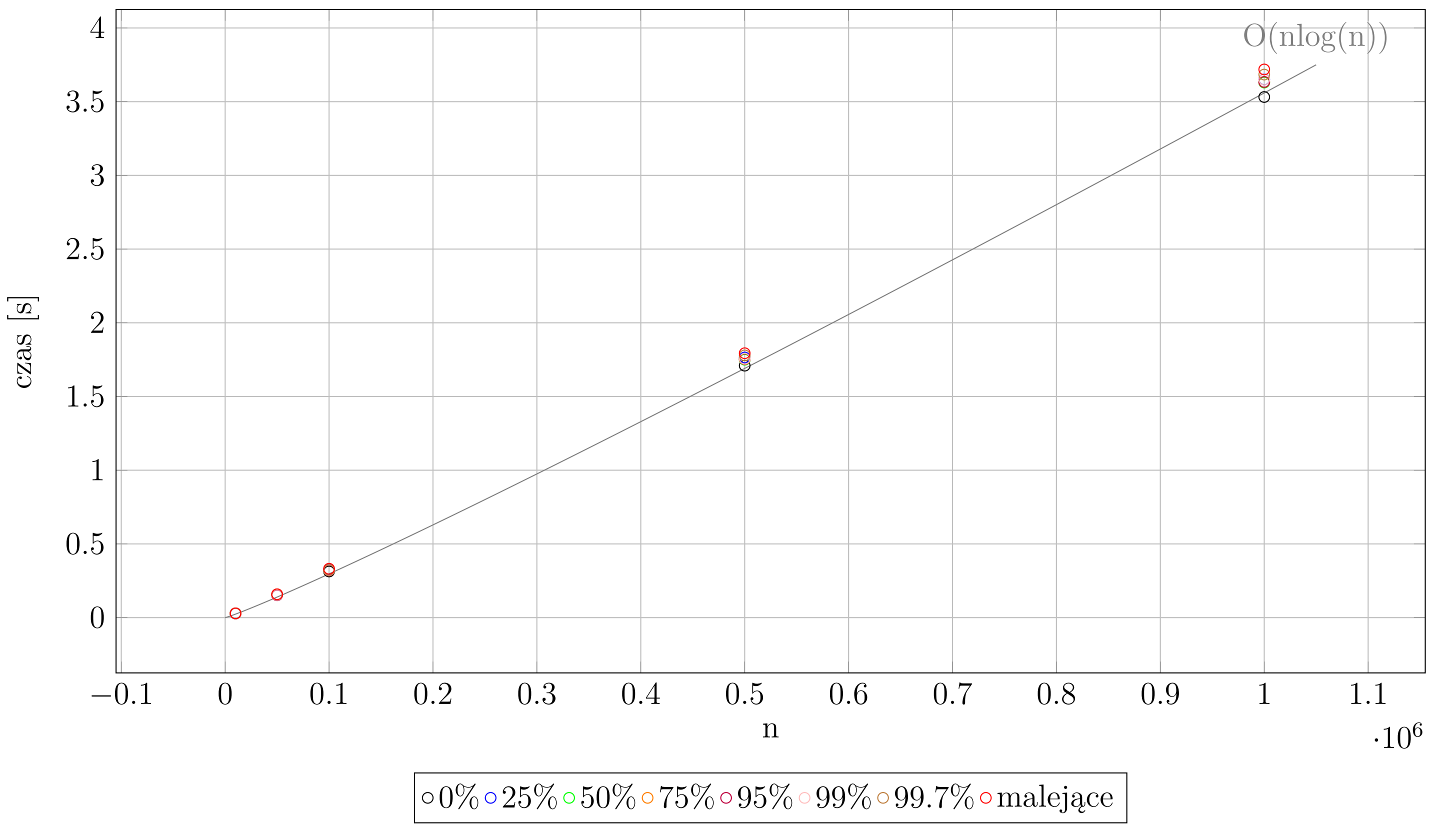
Wyniki badań przedstawiają się następująco:

Tabela 1: Tabela pokazująca zależność czasu sortowania od ilości elementów i procentu posortowania

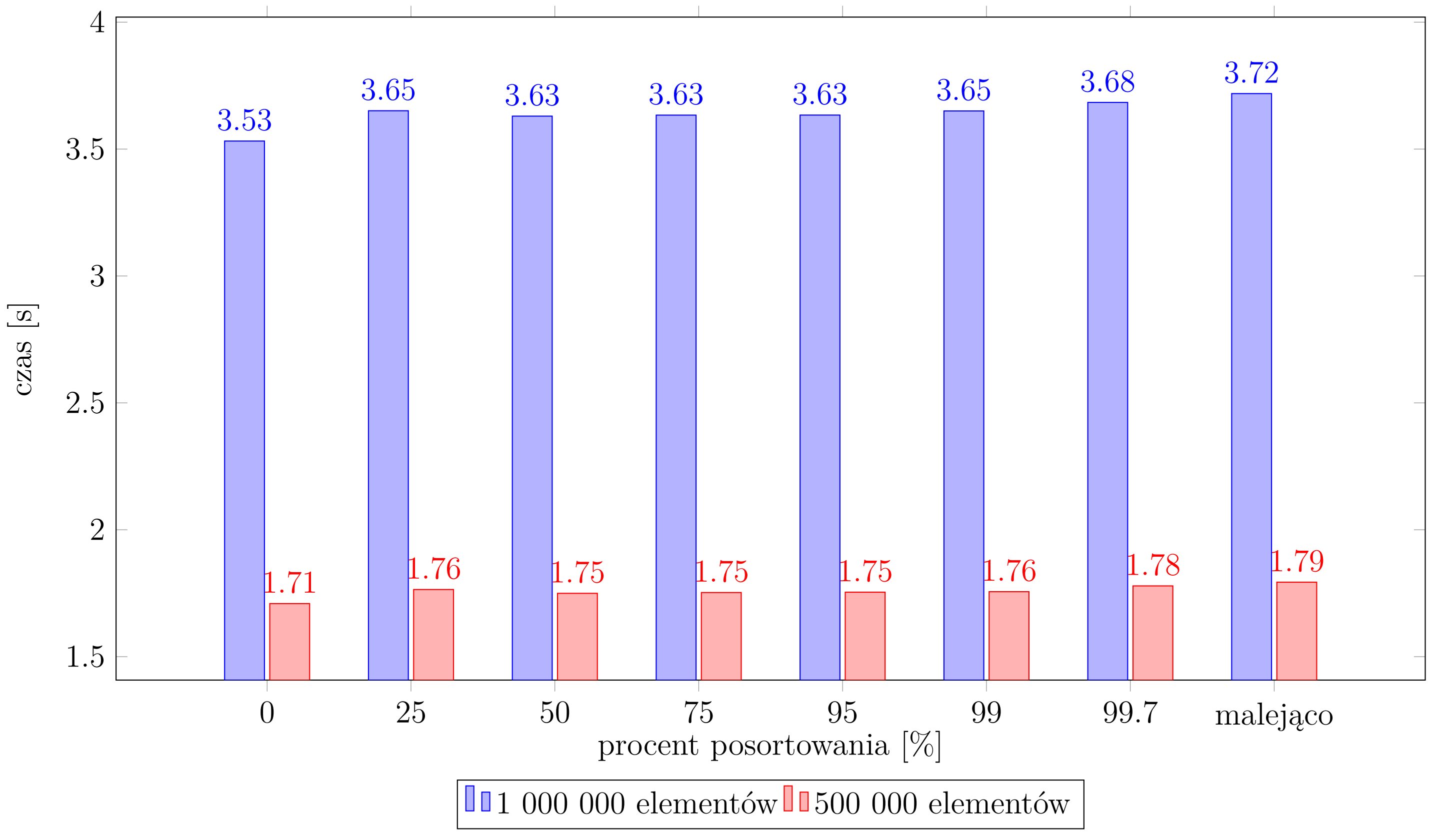
n	0%	25%	50%	75%	95%	99%	99,7%	malejąco
10000	0,028464	0,028315	0,028334	0,028463	0,028084	0,027907	0,028703	0,028895
50000	0,15326	0,15454	0,155507	0,154498	0,151632	0,154504	0,157731	0,158489
100000	0,313051	0,324961	0,320817	0,320735	0,320147	0,322416	0,323021	0,33154
500000	1,709182	1,764593	1,749607	1,752676	1,754125	1,756129	1,778895	1,793538
1000000	3,531617	3,650977	3,629768	3,633724	3,63393	3,650196	3,683775	3,718621

Jak widać z Wyk. 1 złożoność obliczeniowa sortowania przez scalanie, dla każdego przypadku, wpisuje się w krzywą obrazującą złożoność $O(nlog(n))$. Na podstawie Wyk. 2 można stwierdzić, że wraz z wzrostem procętu posortowania rośnie czas działania algorytmu. Wyjątkiem jest punkt 25% gdzie potrzebny czas jest wyższy od sąsiadujących wartości. Najmniej optymalnym przypadkiem jest tablica posotrowana w odwrotnej kolejności (w badanym przypadku malejącej).

Wykres 1: Zależność czasu o ilości elementów



Wykres 2: Zależności czasu sortowania od procentu posortowania



3 Sortowanie szybkie

Sortowanie szybkie jest również algorytmem typu dziel i zwyciężaj. Zakłada wybór elementu rozdzielającego i podział tablicy na jego podstawie. Elementy mniejsze od rozdzielającego są przed nim, a większe za nim. W najprosztrzej formie elementem rozdzelającym jest ostatni element tablicy. W tym badaniu został użyta tzw. metoda mediany z trzech. Zaimplmentowany algortm ma postać:

Data: A - nieposortowana tablica, l i r - indeksy pomiedzy którymi sotrowana jest tablica
Result: Tablica posortowana pomiedzy indeksami l i r
begin
    split $\longleftarrow$ 0
    if $l < r$ **then**
        split $\longleftarrow$ Partition(A, l, r)
        QuickSort(A, l, split)
        QuickSort(A, split+1, r)
    end
end

Algorytm 3: QuickSort(A, l, r)

Metoda Partition(A, l, r) ma postać:

Data: A - tablica, l i r - indeksy pomiedzy którymi dokonuje się podziału
Result: Indeks punktu podziału tablicy
begin
    pivot $\longleftarrow$ PickPivot(A, l, r)
    $i \longleftarrow l - 1$
    $j \longleftarrow r + 1$
    while *true* **do**
        while $A[i] < pivot$ **do**
            $i \longleftarrow i + 1$
        end
        while $A[j] > pivot$ **do**
            $j \longleftarrow j - 1$
        end
        if $i \geqslant j$ **then**
            return j
        end
        A.Swap(i,j)
    end
end

Algorytm 4: Partition(A, l, r)

Metoda PickPivot(A, l, r) zrealizowana jest jako wybór mediany z pierwszego, środkowego i ostatniego elementu i ma złożoność obliczeniową O(1).

Optymistycznym przypadkiem dla sortowania szybkiego jest sytuacja, w której element rozdzielający jest zawsze medianą sortowanej tablicy. Wówczas zoastaje ona podzilona na dwie identyczne w długości części, zatem głebokość drzewa rekurencji będzie wynosila $log_2(n)$. Aby podzielić tablice należy dokonać n porównań, czyli całkowita złożoność obliczeniowa wynosi $O(nlog(n))$.

Dla przypadku pesymistycznego element rodzielający jest zawsze elementem maksymalnym (bądź minimalnym, w zależności od rodzaju uporządkowania) zbioru. Wtedy zostaje on podzielony na zbiory o $n-1$, 1 i 0 elementów, dla na każdym etapie należy dokonać n porównan, zatem złożoność obliczeniowa jest równa $O(n^2)$.

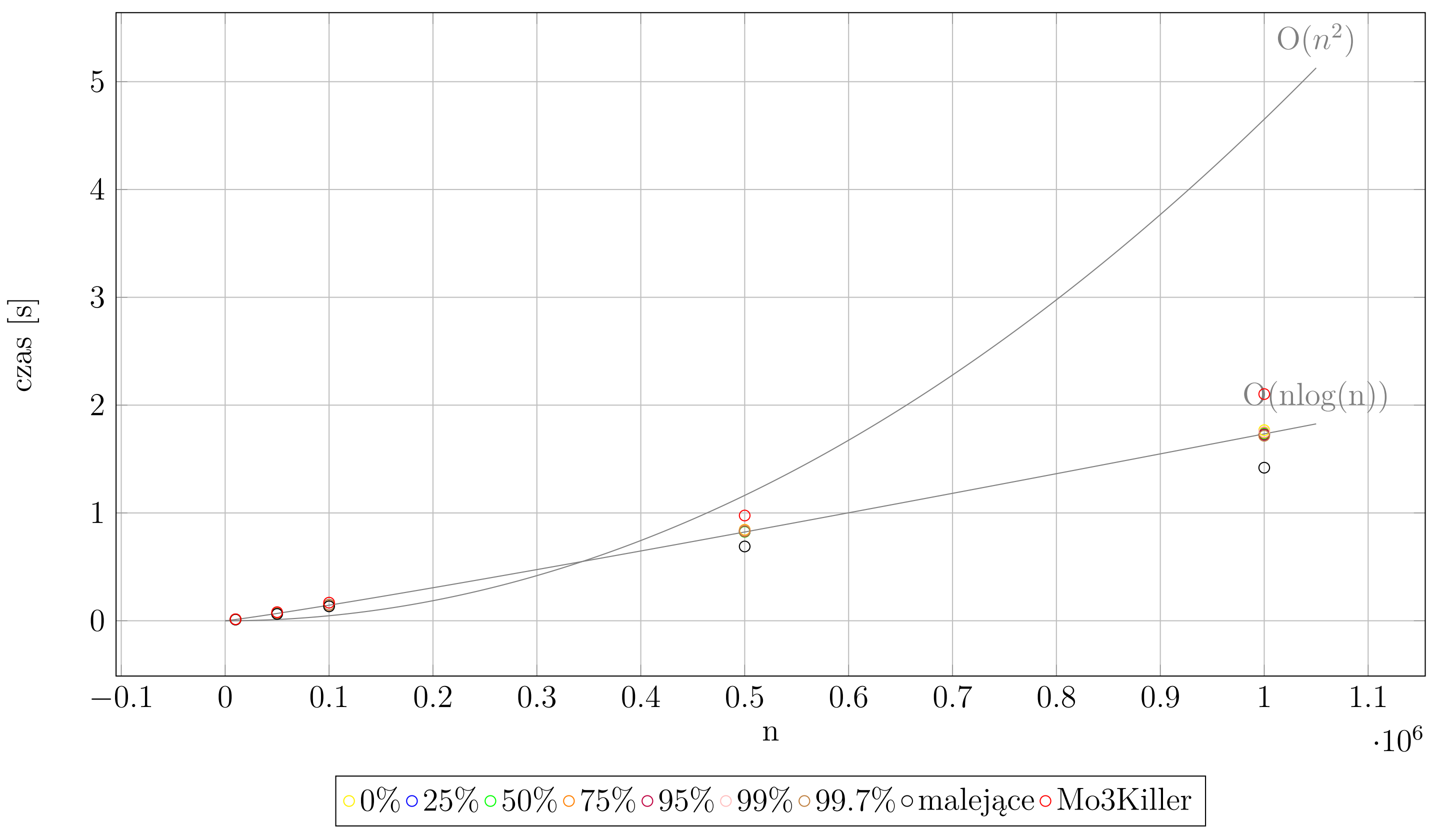
Pokazuje to przewage wyboru elementu rozdzielającego przez medianę z terzech, ponieważ eliminuje najgorszy przypadek, gdy tablica jest odwrotnie posortowana. Dla tej metody najgorszy przypadek wysępuje wówczas, gdy za każdym razem zostanie wybrany element drugi największy (bądź najmniejszy). Złożoność obliczeniowa wtedy rownież bedzie wynosić $O(n^2)$.

Wyniki badań sortowania szybkiego wyglądają następująco:

Tabela 2: Czas sortowania w sekundach w zależności od ilości elementów i procent posortowania

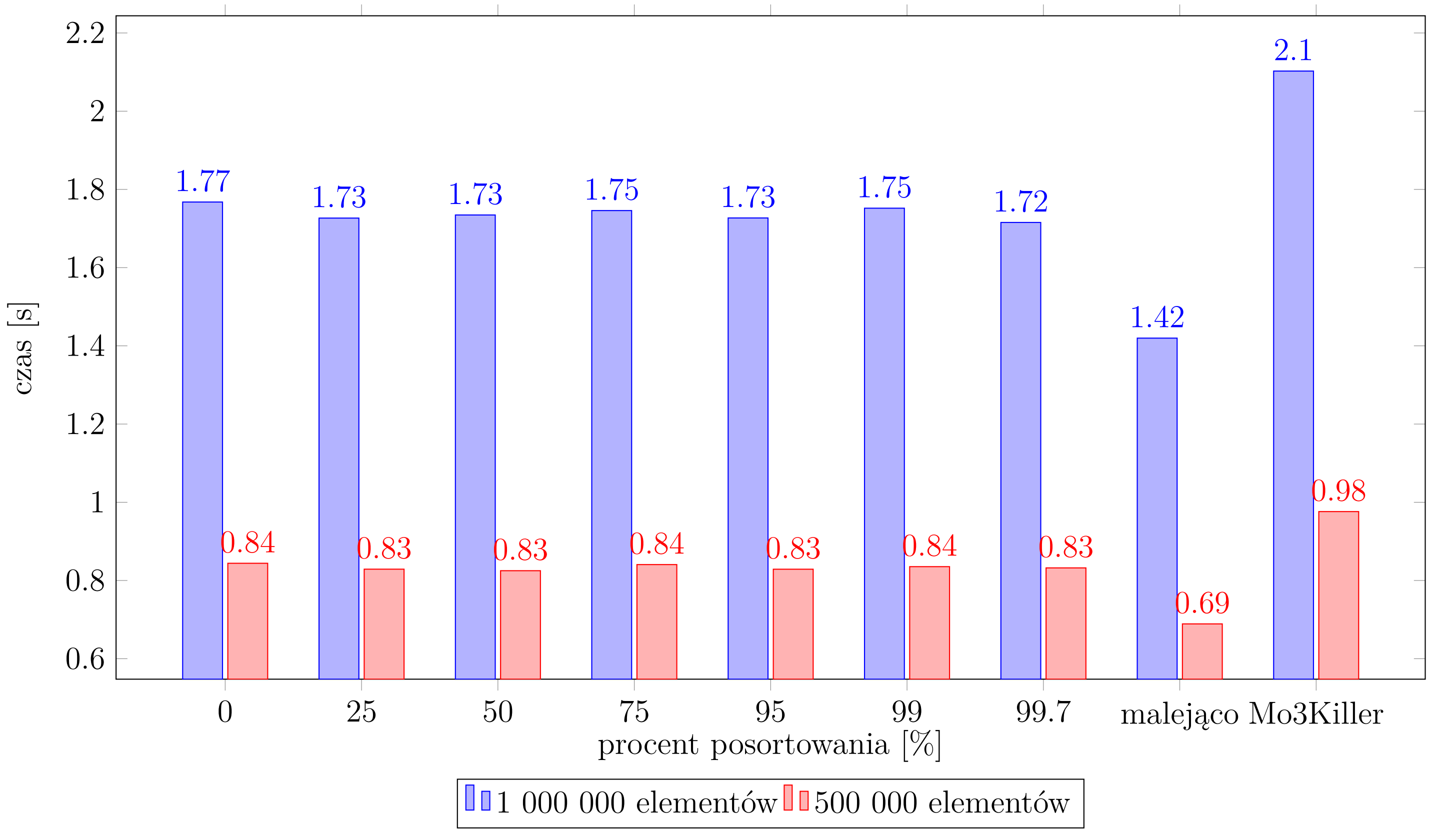
n	0	25	50	75	95	99	99.7	malejąco	Mo3Killer
10000	0,012801	0,012261	0,012977	0,012603	0,012777	0,012679	0,012329	0,01115	0,01395
50000	0,070224	0,070163	0,070319	0,069897	0,070019	0,069708	0,069189	0,06311	0,079639
100000	0,14694	0,148651	0,147161	0,146958	0,146834	0,14683	0,147731	0,133493	0,167912
500000	0,844003	0,828923	0,825011	0,840674	0,828817	0,835374	0,832195	0,688994	0,976188
1000000	1,767726	1,726656	1,73458	1,745884	1,726974	1,751946	1,715451	1,419692	2,102569

Wykres 3: Zależność czasu o ilości elementów



Na Wyk. 3 widać, że czas poszczególnych pomiarów dla różnych procentów posortowań układa się w zależność $O(nlog(n))$. Jendynie zastaw danych mających na celu wymuszenie najgorszego przypadku (Median of 3 killer) nie zachowuje się zgodnie z przewidywaniami. Wyk. 2. pokazuje, że nie ma zależności pomiedzy procentem posortowania, a czasem sortowania. Wartość dla odwróconego zbioru danych jest niższa od pozostałych i wynika to najprawdopodobniej z faktu, że a tym zbiorze nie było powtarzających się liczb. Najgorszym przypadkiem była tablica z sekwencją liczb Median od 3 Killer, jednak czas jest znacznie niższy od przewidywanego.

Wykres 4: Zależność czasu sortowania od procentu posortowania



4 Sortowanie introspektywne

Sortowanie introspektywne jest kolejnym algorytmem typu dziel i zwyciężaj. Jego celem jest uzyskanie szybkości sortowania szybkiego przy jednoczesnej eliminacji pesymistycznego przypadku. Osiąga się to za pomocą sortowania przez kopcowanie w momencie gdy głębokośc rekurencji osiągnie odpowiedni poziom, zazwyczaj $2 \cdot \lfloor log_2(n) \rfloor$. Algorytm ma postać:

Data: A - tablica
Result: Posortowana tablica A
begin
  $maxDepth \longleftarrow 2\lfloor log_2(n) \rfloor$
  $IntroSort(A, l, r, maxDepth)$
end

Algorytm 5: Introsort(A)

Gdzie $IntroSort(A, l, r, maxDepth)$:

Metoda $Partition(A, l, r)$ ma taką samą postać jak przy sortowaniu szybkim, a $HeapSort(A, l, r)$ jest sortowaniem przez kopcowanie.

W przypadku optymistycznym algorytm działa identycznie jak sortowanie szybkie, zatem jego złożoność obliczeniowa wynosi $O(nlog(n))$. W przypadku nieoptymistycznym algorytm wywoła $2 \cdot \lfloor log_2(n) \rfloor$ rekurencji sortowania szybkiego i "przełączy" się na sortowanie przez kopcowanie, który w najgorszym przypadku ma złożoność $O(nlog(n))$, zatem całkowita złożoność będzie wynosiła $O(nlog(n))$.

Wyniki badań przedstawiaja się następująco:

Z Wyk. 5 wynika, że zgodnie z przewidiwaniami wyniki układają się w kształt charakterystyki $O(nlog(n))$ dla każdego przypadku. Dla zbioru elementów Median of 3 Killer, różnież czas potrzebny na posortowanie nie ma zależności kwadratowej (kolor czerwony). Na podstawie Wyk.

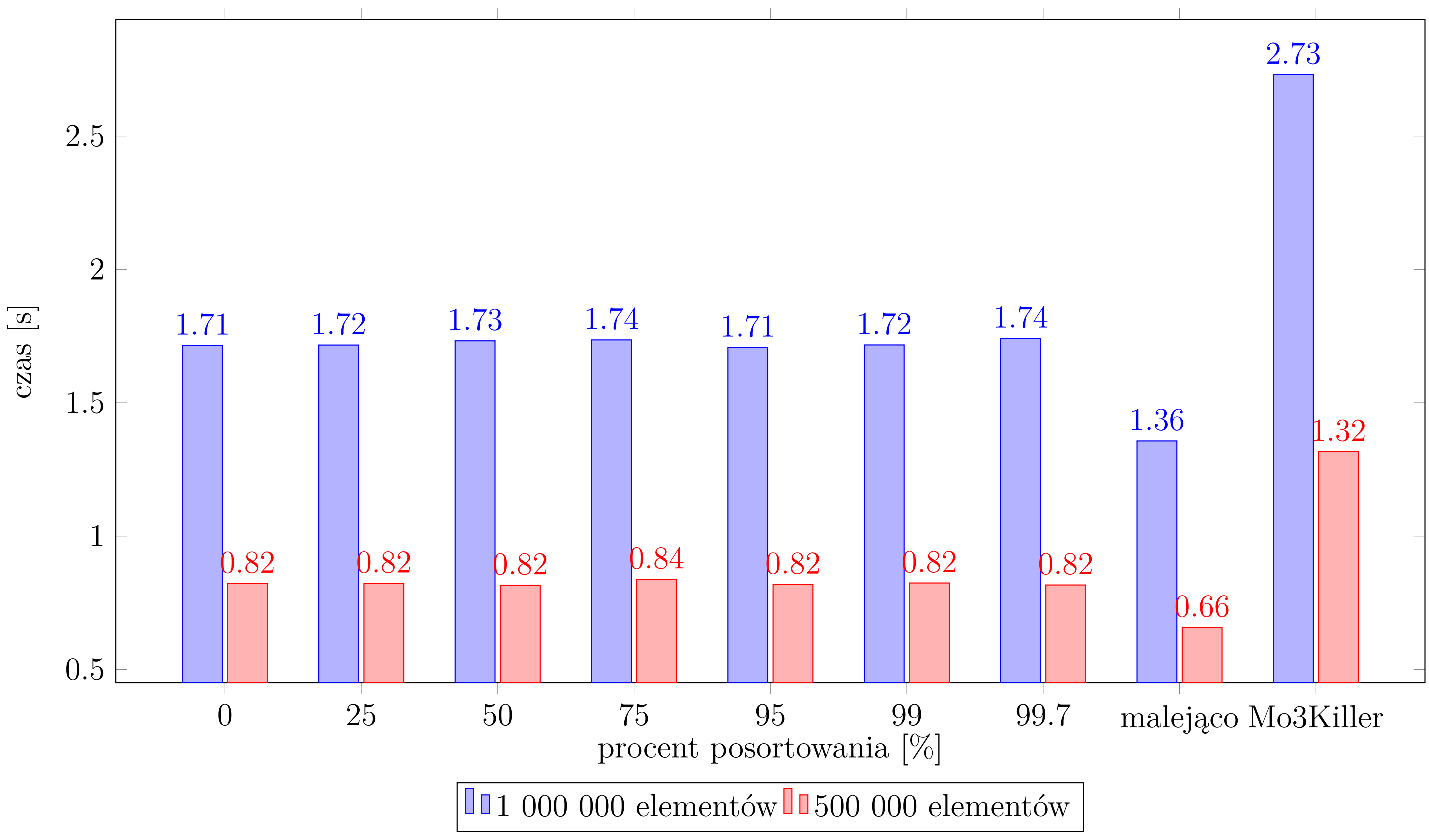
Data: A - tablica
Result: Posortowana tablica A
begin
  $split \longleftarrow 0$
  if $l < r$ **then**
    if $maxDepth \leqslant 0$ **then**
      $HeapSort(A, l, r)$
      return
    end
    $split = Partition(A, l, r)$
    $IntroSort(A, l, split, maxDepth - 1)$
    $IntroSort(A, split + 1, r, maxDepth - 1)$
  end
end

Algorytm 6: Introsort(A, l, r, maxDepth)

Tabela 3: Czas sortowania w sekundach w zależności od ilości elementów i procent posortowania

n	0	25	50	75	95	99	99,7	malejąca	Mo3K
10000	0,012641	0,01255	0,012259	0,012667	0,012224	0,012495	0,01237	0,010972	0,016919
50000	0,070129	0,070003	0,060555	0,069425	0,068965	0,069853	0,069528	0,059145	0,10461
100000	0,147451	0,146459	0,147038	0,148593	0,147288	0,148613	0,145131	0,122599	0,214341
500000	0,82178	0,822635	0,815717	0,837929	0,818479	0,824097	0,816558	0,657414	1,316444
1000000	1,714604	1,716332	1,732167	1,735711	1,707179	1,716727	1,740895	1,356879	2,73031

Wykres 6: Zależność czasu sortowania od procentu posortowania



6, można stwierdzić, że nie ma zależności pomiedzy procentem posortowania tablicy, a czasem działania. Dla tablicy malejącej czas jest niższy prawdopodobnie z powodu, że nie było w niej powtarzających się elementów. Przypadkiem najgorszym był również pesymistyczny przyadek sortowania szybkiego.

5 Wnioski

Eksperymantalna złożonośc obliczeniowa wszystkich algorytmów zgadzała się z przewidywaną złożonością. Jedynym wyjątkiem okazał się pesymistyczny przypadek dla sortowania szybkiego, którego złożoność wyszła $O(nlog(n))$, gdzie powinna byc równa $O(n^2)$. Może to wynikać, z metody wyboru elementu rozdzielającego, jednak dane zostały spreperowane z sposób wymuszający kwadratową wydajność.

Z wszystkich najszybszymi okazały się sortowanie szybkie i intorspektywne. Wydłużony czas działania sortowania przez scalenie może wiązać, się z tworzeniem dwóch tablic i kopiowanie do nich elementów.

Wydajność sortowania szykiego i introsektywnego nie zależały od procętu posortowania tablicy. Dla sortowania przez scalanie zauważa się nieznaczą zależność szybkości sortowania od procentu posortowania. Ciekawy jest ziwększony czas sortowania dla tablicy posortowanej w 25% w stosunku do wartośco 0% i 50%.

Literatura

[1] Michael T. Goodrich, Roberto Tamassia, David Mount. Data Structures and Algorithms in C++ Second Edition.

[2] https://en.wikipedia.org/wiki/Merge_sort

[3] https://en.wikipedia.org/wiki/Quicksort

[4] https://en.wikipedia.org/wiki/Introsort

[5] https://en.wikipedia.org/wiki/Heapsort

Wykres 5: Zależność czasu o ilości elementów