Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

Projektowanie Algorytmów i Metody Sztucznej Inteligencji

Projekt 2

Zadanie na ocenę bdb

Prowadzący:
Dr inż. Łukasz Jeleń

Wykonała:
Zuzanna Mejer, 259382

Wrocław, 6 maja 2022r.

Spis treści

1 Wprowadzenie **2**

2 Opis badanych algorytmów i ich złożoność obliczeniowa **2**
2.1 Sortowanie przez scalanie 2
2.2 Sortowanie szybkie 2
2.3 Sortowanie introspektywne 2
2.4 Porównanie złożoności obliczeniowych wybranych algorytmów 2

3 Implementacja algorytmów sortowania **3**
3.1 Sortowanie przez scalanie 3
3.2 Sortowanie szybkie 5
3.3 Sortowanie introspektywne 6

4 Zadanie 1 - przeszukanie i przefiltrowanie danych **8**
4.1 Krótki opis 8
4.2 Analiza złożoności 9

5 Analiza złożoności algorytmów sortowań **11**
5.1 Przebieg eksperymentów 11
5.2 Sortowanie przez scalanie 12
5.3 Sortowanie szybkie 13
5.4 Sortowanie introspektywne 14

6 Średnia wartość i mediana **15**

7 Podsumowanie i wnioski **15**

8 Bibliografia **16**

1. Wprowadzenie

Zadanie miało na celu zapoznanie się z algorytmami sortowania oraz przeprowadzenie analizy efektywności wybranych i zaimplementowanych sortowań. Z wymienionych algorytmów wybrałam sortowania: przez scalanie, szybkie oraz introspektywne.

2. Opis badanych algorytmów i ich złożoność obliczeniowa

2.1. Sortowanie przez scalanie

Jest to rekurencyjny algorytm sortowania danych, stosujący metodę „dziel i zwyciężaj". W algorytmie wyróżnia się trzy podstawowe kroki: podział danych wejściowych na 2 rozłączne podzbiory; rekurencyjnie zastosowanie sortowania dla każdego podzbioru, aż do uzyskania struktur jednoelementowych; scalenie posortowanych podzbiorów w jeden zbiór. Całkowita złożoność obliczeniowa dla sortowania przez scalanie wynosi $O(n \cdot logn)$, w związku z czym zastosowanie tego sortowania okaże się wydajniejsze dla bardzo dużych tablic.

2.2. Sortowanie szybkie

Również jest to algorytm sortowania danych stosujący metodę „dziel i zwyciężaj", nie wykorzystuje on jednak dodatkowych podtablic. Istnieje wiele implementacji sortowania szybkiego, jednak generalna idea jest taka, że wybierany jest jeden element w sortowanej strukturze, który nazywany jest piwotem. Może być to element środkowy, pierwszy, ostatni bądź losowy, przy czym należy pamiętać, że w przypadkach, kiedy piwot jest ciągle maksymalny lub minimalny, występuje najgorsza złożoność obliczeniowa $O(n^2)$. Przy optymalnych wyborach piwotu, złożoność wynosi $O(n \cdot logn)$.

2.3. Sortowanie introspektywne

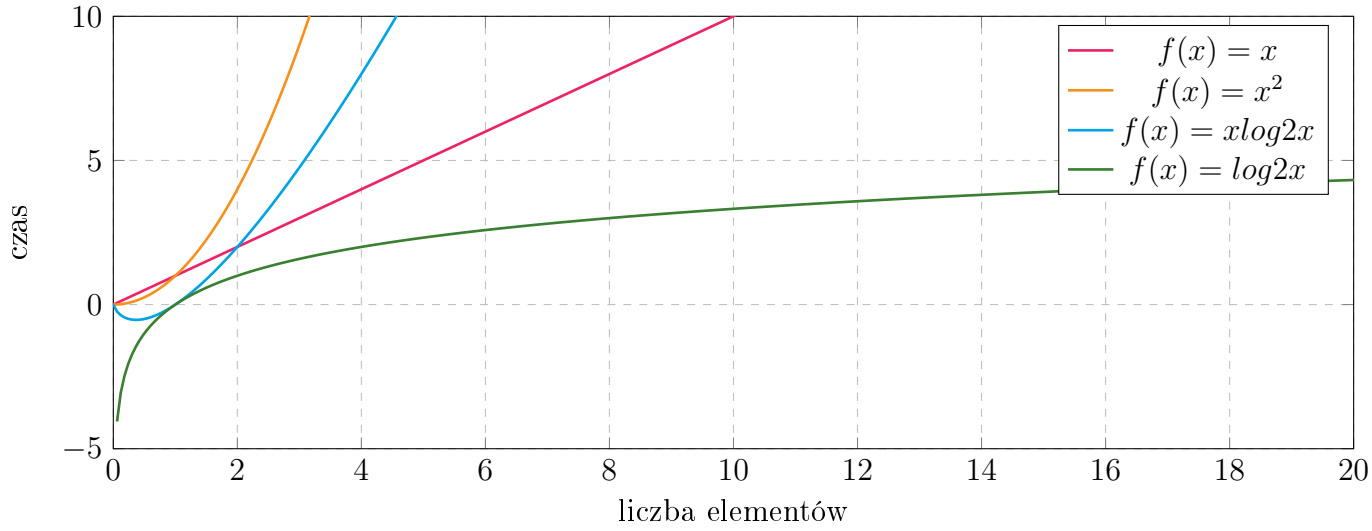
Jest to odmiana sortowania hybrydowego, które opiera się na spostrzeżeniu, że niewydajne jest wywoływanie ogromnej liczby rekurencji dla małych tablic w algorytmie sortowania szybkiego. Głównym założeniem algorytmu sortowania introspektywnego jest zatem wyeliminowanie problemu złożoności $O(n^2)$ występującej w najgorszym przypadku sortowania szybkiego. Sortowanie introspektywne jest połączeniem sortowania szybkiego i sortowania przez kopcowanie, które jest traktowane jako pomocnicze. Tym samym złożoność obliczeniowa wynosi $O(n \cdot logn)$.

2.4. Porównanie złożoności obliczeniowych wybranych algorytmów

Poniższa tabela zestawia oczekiwane i najgorsze przypadki złożoności wybranych algorytmów sortowania. Poniżej dodano także poglądowy wykres funkcji, na którym widać, że dla małej liczby danych sortowanie o złożoności kwadratowej będzie wydajniejsze niż dla logarytmicznej i przeciwnie dla dużej liczby elementów do posortowania.

Tab. 1: Porównanie oczekiwanych i najgorszych przypadków złożoności obliczeniowej dla wybranych algorytmów sortowania

	sortowanie		
	przez scalanie	szybkie	introspektywne
typowa złożoność	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$
najgorszy przypadek złożoności	$O(nlogn)$	$O(n^2)$	$O(nlogn)$



Rys. 1: Poglądowe wykresy funkcji możliwych złożoności obliczeniowych

3. Implementacja algorytmów sortowania

3.1. Sortowanie przez scalanie

Poniżej przedstawiona została implementacja sortowania przez scalanie. Składa się ona z 2 funkcji: *merge* oraz *mergesort*. Funkcja *merge* tworzy dwie podtablice o odpowiedniej liczbie komórek, do których zapisuje odpowiednie elementy. Następnie w tej funkcji odbywa się sortowanie, czyli „wkładanie" odpowiednich elementów do wyjściowej tablicy, dopóki w obydwu podtablicach coś jest. Jeżeli któraś z podtablic stanie się pusta, następuje „wkładanie" pozostałych elementów drugiej podtablicy do wyjściowej tablicy. Funkcja *mergesort* zawiera w sobie oprócz kroku podstawowego, rekurencyjne wywołania samej siebie dla 2 podtablic i wywołanie wcześniej wymienionej funkcji *merge* do scalania powstałych podtablic.

```
1    void merge(float array[], int left, int middle, int right)
2    {
3        int sub_array1 = middle - left + 1;
4        int sub_array2 = right - middle;
5
6        float *left_array = new float[sub_array1];
7        float *right_array = new float[sub_array2];
8
9        for (int i = 0; i < sub_array1; i++)
```

```
10          {
11              left_array[i] = array[left + i];
12          }
13
14          for (int i = 0; i < sub_array2; i++)
15          {
16              right_array[i] = array[middle + 1 + i];
17          }
18
19          int index_sub_array1 = 0;
20          int index_sub_array2 = 0;
21          int index_merged_arrays = left;
22
23          while (index_sub_array1 < sub_array1
24                  && index_sub_array2 < sub_array2)
25          {
26              if (left_array[index_sub_array1]
27                  <= right_array[index_sub_array2] )
28              {
29                  array[index_merged_arrays]
30                  = left_array[index_sub_array1];
31                  index_sub_array1++;
32              }
33
34              else
35              {
36                  array[index_merged_arrays]
37                      = right_array[index_sub_array2];
38                  index_sub_array2++;
39              }
40              index_merged_arrays++;
41          }
42
43
44          while (index_sub_array1 < sub_array1)
45          {
46              array[index_merged_arrays] = left_array[index_sub_array1];
47              index_sub_array1++;
48              index_merged_arrays++;
49          }
50
51
52          while (index_sub_array2 < sub_array2)
53          {
54              array[index_merged_arrays] = right_array[index_sub_array2];
55              index_sub_array2++;
56              index_merged_arrays++;
```

```
57          }
58      }
59
60      void merge_sort(float array[], int const begin, int const end)
61      {
62          // krok podstawowy
63          if (begin >= end)
64          {
65              return;
66          }
67
68          int middle = begin + (end - begin) / 2;
69          merge_sort(array, begin, middle);
70          merge_sort(array, middle + 1, end);
71          merge(array, begin, middle, end);
72      }
```

3.2. Sortowanie szybkie

Poniżej przedstawiona została implementacja sortowania szybkiego. Wybrana implementacja ustawia 2 pomocnicze zmienne - i przed tablicą oraz j za tablicą. Następnie wyznacza piwot, który w celu uniknięcia najgorszego przypadku złożoności obliczeniowej, zostaje ustawiony po środku tablicy. Następnie odbywa się przechodzenie po tablicy i porównywanie elementów z piwotem, oraz, w określonych przypadkach, zamiana miejscami elementów znajdujących się na prawo i na lewo od piwotu. Funkcja *quicksort* także wywołuje się rekurencyjnie.

```
1       void quicksort(float array[], int left, int right)
2       {
3               if(right <= left) return;
4
5               int i = left - 1;
6           int j = right + 1;
7           int pivot = array[(left+right)/2];
8
9               while(1)
10          {
11                  while(pivot > array[++i]);
12                  while(pivot < array[--j]);
13
14                  if( i <= j)
15                          std::swap(array[i], array[j]);
16                  else
17                          break;
18              }
19
20              if(j > left)
21                  quicksort(array, left, j);
22              if(i < right)
```

```
23                  quicksort(array, i, right);
24      }
```

3.3. Sortowanie introspektywne

Poniżej przedstawiona została implementacja sortowania introspektywnego. Składa się ona z 5 funkcji: *heapify, heap_sort, insertion_sort, partition* oraz *intro_sort*.

- Funkcje *heapify, heap_sort* są algorytmami sortowania przez kopcowanie. Algorytm sortowania przez kopcowanie składa się z dwóch faz. W pierwszej sortowane elementy reorganizowane są w celu utworzenia kopca. Zatem znajdowany jest największy element w kopcu i ustawiany jako „ojciec". W drugiej fazie dokonywane jest właściwe sortowanie - budowany jest poprawny kopiec, następnie zamieniany jest „ojciec" z „najmłodszym synem" (przeniesienie największej wartości na koniec tablicy) i wywoływana jest znowu funkcja kopcowania, już dla zredukowanej tablicy.

- Funkcja *insertion_sort* odpowiada za algorytm sortowania przez wstawianie. W tym algorytmie następuje przejście po elementach tablicy i porównanie obecnej wartości elementu z wartością elementu poprzedniego. W zależności od wartości tych elementów - są one albo zamieniane, kontynuując porównanie zamienianej wartości aż do początku tablicy, albo są zostawiane na swoich pozycjach i algorytm wykonuje się dla dalszych elementów tablicy.

- Funkcja *partition* jest funkcją, która dzieli tablicę wejściową tak, jak w algorytmie sortowania szybkiego. Wybierany jest piwot - w tym przypadku jest to element na końcu tablicy, następnie wartości elementów od początku tablicy są porównywane z piwotem i odpowiednio przemieniane.

- Funkcja *intro_sort* wykorzystuje badanie głębokości rekurencji - w zależności od rozmiaru tablicy, wywołane zostaje sortowanie szybkie, przez kopcowanie, lub przez wstawianie. Sortowanie szybkie zostaje wywołane dla małej liczby danych (ze względu na złożoność kwadratową), jednak dla najmniejszych tablic wywołane zostaje sortowanie przez wstawianie, a dla największych, po przekroczeniu progu $2 \cdot log2(size)$ - sortowanie przez kopcowanie.

```
1   void heapify(float array[], int n, int i)
2   {
3       int largest = i;
4       int l = 2 * i + 1;
5       int r = 2 * i + 2;
6
7
8       if (l < n && array[l] > array[largest])
9           largest = l;
10
11
12      if (r < n && array[r] > array[largest])
13          largest = r;
14
15
```

```
16          if (largest != i)
17          {
18              std::swap(array[i], array[largest]);
19              heapify(array, n, largest);
20          }
21      }
22
23
24
25      void heap_sort(float array[], int n)
26      {
27          for (int i = n / 2 - 1; i >= 0; i--)
28              heapify(array, n, i);
29
30
31          for (int i = n - 1; i > 0; i--)
32          {
33              std :: swap(array[0], array[i]);
34              heapify(array, i, 0);
35          }
36      }
37
38
39
40      void insertion_sort (float array[], int N)
41      {
42          int i, j;
43          float temp;
44          for (i=1; i<N; ++i)
45          {
46              temp=array[i];
47              for (j=i; j>0 && temp<array[j-1]; --j)
48                array[j]=array[j-1];
49              array[j]=temp;
50          }
51      }
52
53
54
55      int partition (float data[], int left , int right )
56      {
57          int pivot = data[right];
58          int temp;
59          int i = left;
60          for (int j = left ; j < right; ++j)
61          {
62              if (data[j] <= pivot)
```

```
63              {
64                  std::swap(data[j], data[i]);
65                  i++;
66              }
67          }
68
69          data[right] = data[i];
70          data[i] = pivot;
71
72          return i;
73      }
74
75
76
77      void intro_sort(float array[] , int size)
78      {
79          int partition_size = partition(array , 0, size - 1);
80              if ( partition_size < 16)
81              {
82                  insertion_sort(array , size );
83              }
84              else if (partition_size > (2 * std::log( size )))
85              {
86                  heap_sort(array, size );
87              }
88              else
89              {
90                  quicksort(array, 0, size -1);
91              }
92      }
```

4. Zadanie 1 - przeszukanie i przefiltrowanie danych

4.1. Krótki opis

Plik udostępniony do sortowania był okrojoną bazą filmów „IMDb Largest Review Dataset" ze strony kaggle.com. Plik zawierał tytuły filmów oraz przypisane im oceny. Niektóre pola z ocenami były puste, zatem przed wykonaniem zadań związanych z sortowaniem, należało wykonać przeszukanie i usunięcie wpisów bez ocen. Do wykonania tego zadania, zastosowano gotową strukturę z biblioteki STL: *std::vector*. Mimo chęci wykonania sortowań na strukturze dwuelementowej: *std::vector< std::pair <std::string, float> >*, przechowującej i tytuł filmu, i ocenę, komputery, na których wykonywałam testy złożoności obliczeniowej, nie były w stanie wykonać sortowań dla maksymalnej liczby elementów z pliku. Podsumowując, wykonane zostało przeszukiwanie, wykorzystujące strukturę jednoelementową, a następnie sortowane były jedynie oceny filmów. Poniżej przedstawiono algorytm przeszukiwania struktury i usuwania pól z pustymi ocenami:

```
1       for (int i = 0; i< structure.size(); ++i)
```

```
2       {
3           if ( structure[i].second.empty() )
4           {
5               structure.erase(structure.begin() + i--);
6           }
7       }
```

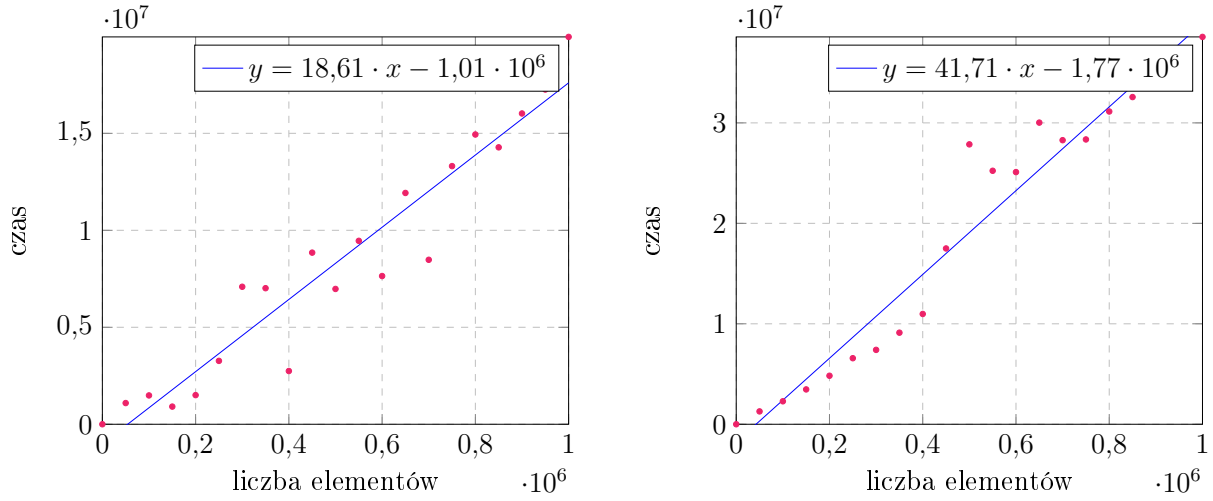
4.2. Analiza złożoności

Kluczową rolę w kodzie odgrywają 2 funkcje: *empty* oraz *erase*. Funkcja *empty* ma złożoność obliczeniową stałą dla jednego elementu, jednak zaimplementowana jak w powyższy sposób, wykona się dla n elementów, zatem jej złożoność w tym przypadku powinna być liniowa $O(n)$. Funkcja *erase* ma oczekiwaną złożoność obliczeniową także liniową $O(n)$. Przeprowadzone zostały testy dla różnych danych w pliku i zmierzone zostały czasy działania obydwu funkcji. Wyniki przedstawia poniższa tabela.

Tab. 2: Czas działania funkcji *empty* i *erase* dla różnej liczby elementów

liczba elementów	czas działania empty [ns]	czas działania erase [ns]
0	381	734
50000	1092756	1280059
100000	1486812	2285586
150000	910546	3468263
200000	1502636	4828635
250000	3267864	6575286
300000	7089495	7404888
350000	7015680	9116657
400000	2745055	10978655
450000	8849012	17499953
500000	6976789	27862306
550000	9457213	25233448
600000	7641937	25103366
650000	11925197	30027679
700000	8479812	28277875
750000	13310083	28351194
800000	14939825	31133886
850000	14277732	32570166
900000	16023509	34344044
950000	17258645	35836178
1000000	19971461	38559105

Na podstawie tabeli 2 wygenerowane zostały charakterystyki działania obydwu funkcji dla różnej liczby danych w pliku. Jak pokazują poniższe wykresy, obydwie funkcje przypominają oczekiwaną charakterystykę liniową. Zatem, funkcje *empty* oraz *erase* w przedstawionej implementacji, mają liniowe złożoności obliczeniowe $O(n)$.



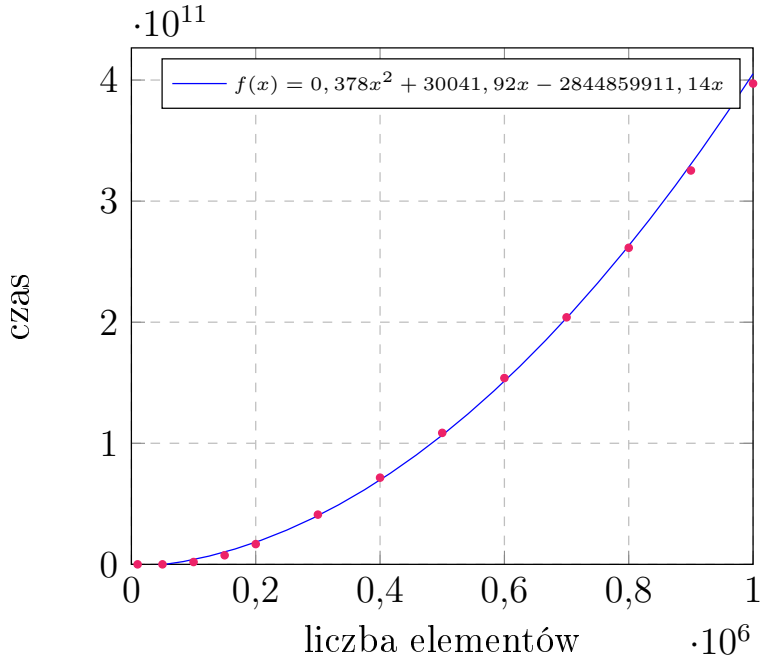
(a) empty (b) erase

Rys. 2: Złożoności obliczeniowe funkcji *empty* i *erase* w przedstawionej implementacji

Złożoności obydwu funkcji są liniowe, zatem całość - przeszukanie i przefiltrowanie danych powinno mieć złożoność kwadratową: $n \cdot n = n^2$. W poniższej tabeli przedstawiono zebrane pomiary działania całego algorytmu.

Tab. 3: Pomiary czasu działania całego algorytmu przeszukiwania i usuwania wybranych pól dla różnej liczby danych

liczba elementów	czas [ns]
10000	99010
50000	2233885
100000	1916298094
150000	7481184919
200000	16730176118
300000	41073112025
400000	71606117902
500000	108592553455
600000	153867195664
700000	204008003164
800000	261449177263
900000	325313418247
1000000	397092989768
1010294	426561982926



Rys. 3: Złożoność obliczeniowa całego algorytmu przeszukiwania i usuwania wybranych elementów

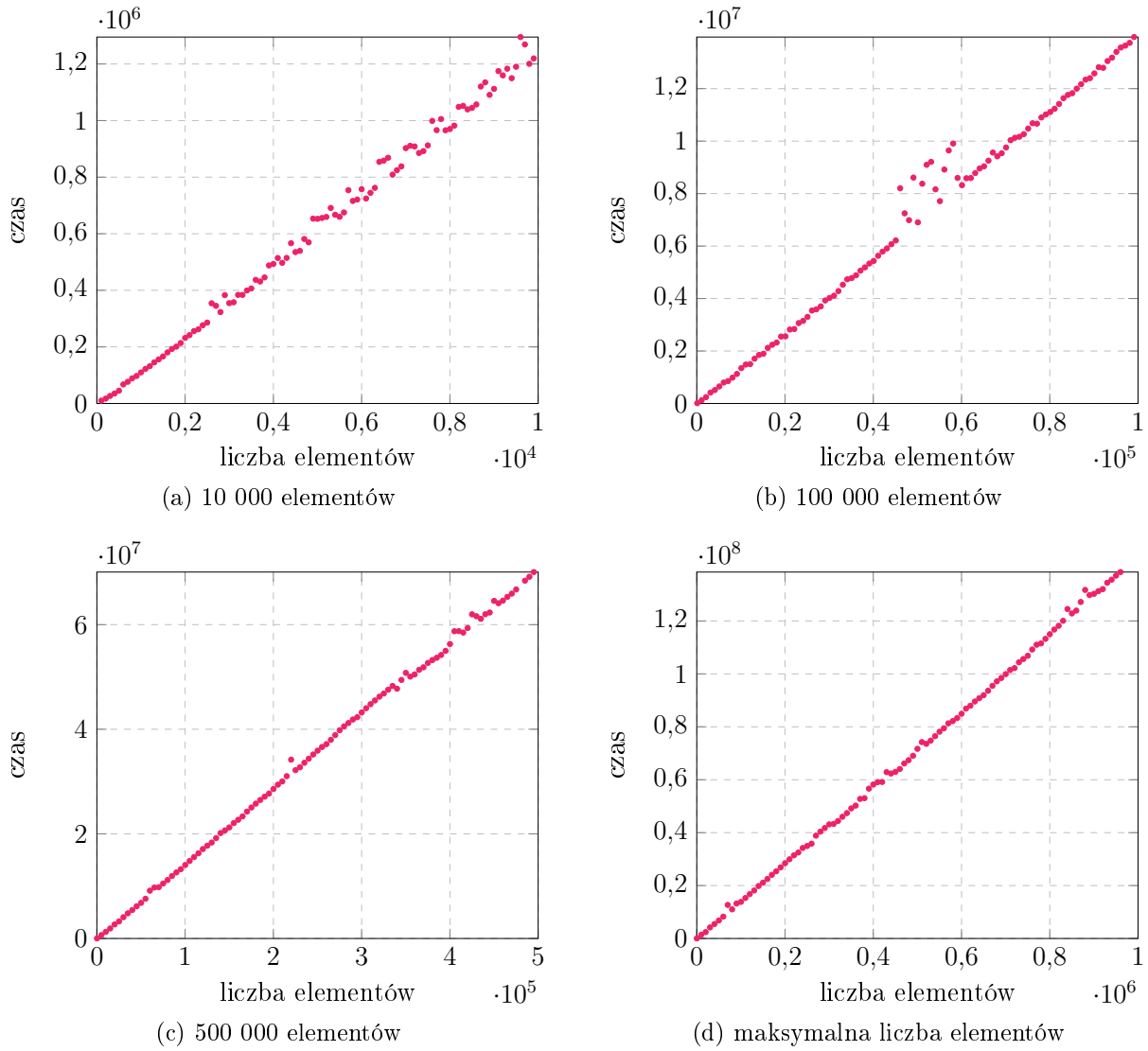
Jak widać na powyższym wykresie, przeszukanie i przefiltrowanie danych ma kwadratową złożoność obliczeniową $O(n^2)$. Dla ponad miliona danych, nie jest to optymalna złożoność. Łączny czas wykonywania przeszukiwania i usuwania wybranych pól zajęła: **426561982926 ns**, czyli około **7,10 min**.

5. Analiza złożoności algorytmów sortowań

5.1. Przebieg eksperymentów

Sortowane były jedynie oceny filmów. Sortowania odbyły się dla 10 000, 100 000, 500 000 oraz maksymalnej ilości danych z pliku po przefiltrowaniu. Dla każdego zestawu danych wykonano po 100 pomiarów.

5.2. Sortowanie przez scalanie



(a) 10 000 elementów

(b) 100 000 elementów

(c) 500 000 elementów

(d) maksymalna liczba elementów

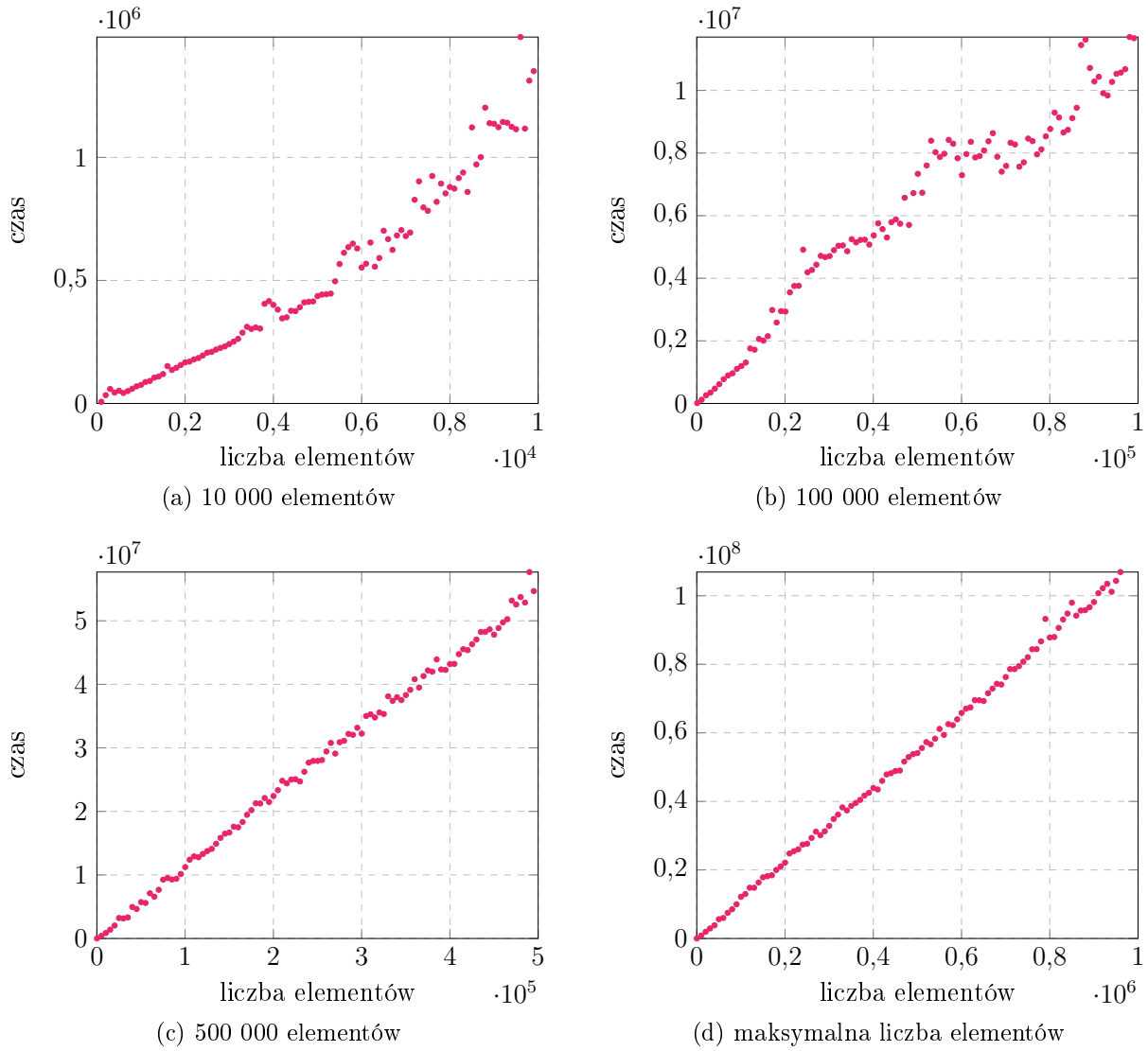
Rys. 4: Sortowanie przez scalanie dla różnej liczby elementów

Tab. 4: Dokładny i przybliżony czas sortowania przez scalanie

	liczba elementów			
	10 000	100 000	500 000	maksymalna
dokładny czas sortowań [ns]	1295207	13966928	70052081	138481693
przybliżony czas sortowań [ms]	1,29	13,97	70,05	138,48

Algorytm sortowania przez scalanie wykazał złożoność obliczeniową liniową dla każdego zestawu danych. Czas sortowania dla maksymalnej liczby elementów z pliku wyniósł **138,48 ms** $\approx$ **0,14 s**.

5.3. Sortowanie szybkie



(a) 10 000 elementów

(b) 100 000 elementów

(c) 500 000 elementów

(d) maksymalna liczba elementów

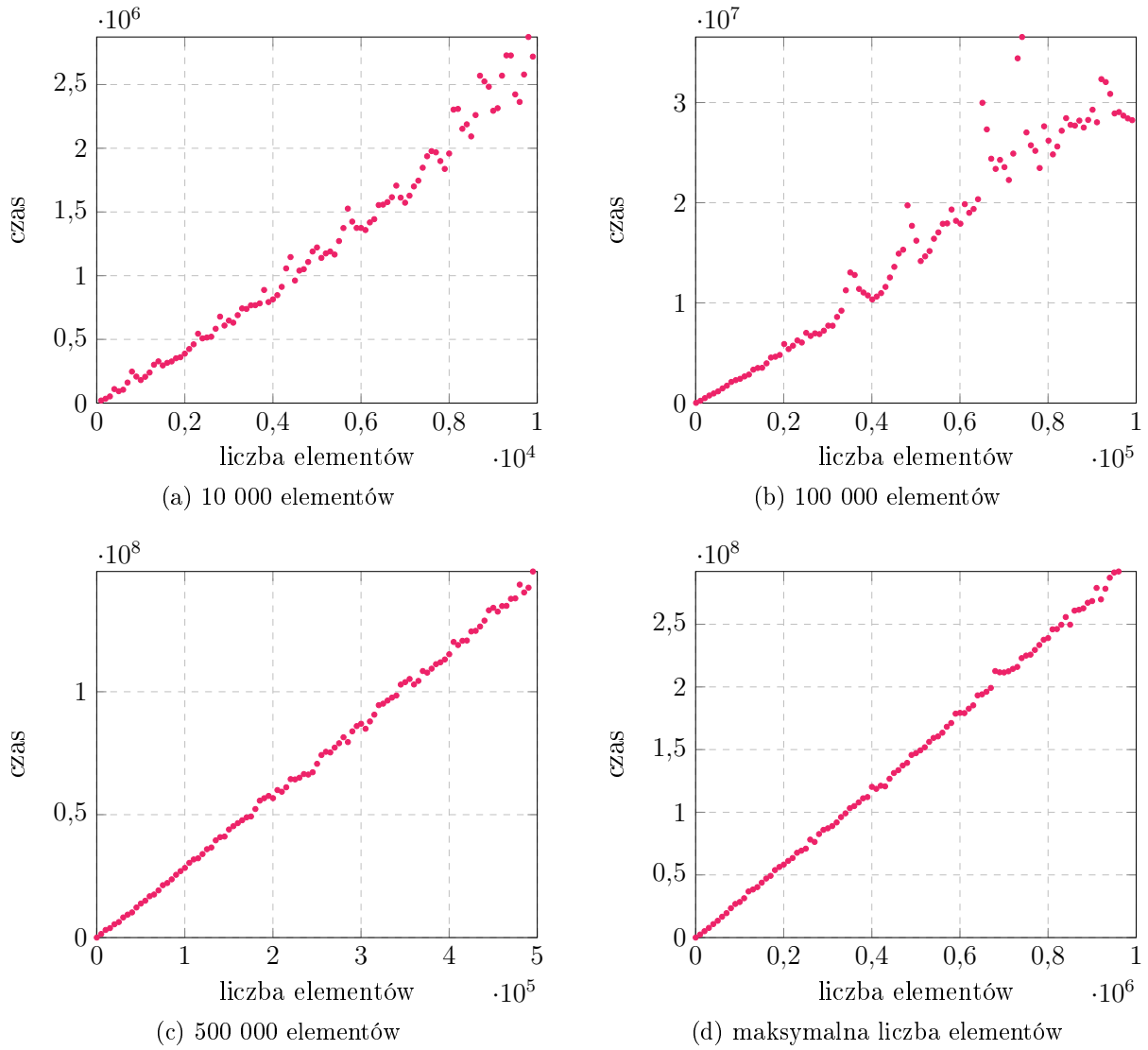
Rys. 5: Sortowanie przez scalanie dla różnej liczby elementów

Tab. 5: Dokładny i przybliżony czas sortowania szybkiego

	liczba elementów			
	10 000	100 000	500 000	maksymalna
dokładny czas sortowań [ns]	1489031	11705054	57671388	106918054
przybliżony czas sortowań [ms]	1,49	11,71	57,67	106,92

Dla małej ilości danych (10 000), algorytm sortowania szybkiego wykazał cechy złożoności kwadratowej, natomiast dla większych ilości danych - liniowej. Czas sortowania dla maksymalnej liczby elementów z pliku wyniósł **106,92 ms ≈ 0,11 s**.

5.4. Sortowanie introspektywne



(a) 10 000 elementów

(b) 100 000 elementów

(c) 500 000 elementów

(d) maksymalna liczba elementów

Rys. 6: Sortowanie introspektywne dla różnej liczby elementów

Tab. 6: Dokładny i przybliżony czas sortowania introspektywnego

	liczba elementów			
	10 000	100 000	500 000	maksymalna
dokładny czas sortowań [ns]	287312	36543660	148902464	292090659
przybliżony czas sortowań [ms]	0,29	36,54	148,90	292,09

Algorytm sortowania introspektywnego wykazał złożoność obliczeniową liniową dla każdego zestawu danych. Czas sortowania dla maksymalnej liczby elementów z pliku wyniósł **292,09 ms ≈ 0,29 s**.

6. Średnia wartość i mediana

Ponadto, dla każdego zestawu danych zostały wyznaczone średnie wartości oraz mediany rankingu, których wartości zostały przedstawione w poniższej tabeli:

Tab. 7: Średnia wartość oraz mediana wyznaczona dla każdego zestawu danych

	liczba elementów			
	10 000	100 000	500 000	maksymalna
średnia wartość	5,46	6,09	6,67	6,64
mediana	5	7	7	7

7. Podsumowanie i wnioski

1. Algorytm napisany do przeszukania i przefiltrowania danych okazał się niewydajny. Jego złożoność obliczeniowa wyniosła $O(n^2)$, co bardzo spowalniało cały program.

2. Łączny czas wykonywania przeszukiwania i usuwania wybranych pól zajęła: 426561982926 ns, czyli około 7,10 min.

3. Żadne z sortowań nie wykazało oczekiwanych złożoności obliczeniowych. Według tabeli 1, typowa złożoność powinna wynieść $O(nlog2n)$. Sortowanie przez scalanie oraz sortowanie introspektywne wykazało złożoność liniową $O(n)$. Sortowanie szybkie dla najmniejszej liczby elementów, wykazało najgorszy przypadek - złożoność kwadratową $O(n^2)$, dla pozostałych zestawów danych - także złożoność liniową $O(n)$.

4. W poniższej tabeli zestawiono czasy sortowań poszczególnych zestawów danych za pomocą zaimplementowanych algorytmów.

Tab. 8: Porównanie czasów sortowań

	liczba elementów			
czas sortowań [ms]	10 000	100 000	500 000	maksymalna
przez scalanie	1,29	13,97	70,05	138,48
szybkie	1,49	11,71	57,67	106,92
introspektywne	0,29	36,54	148,90	292,09

Zdecydowanie najlepszy czas, wyróżniający się dla małej liczby elementów (10 000) osiągnął algorytm sortowania introspektywnego, jednak dla większej liczby danych, zajął on najwięcej czasu. Sortowanie szybkie, zgodnie z oczekiwaniami, zajął najwięcej czasu podczas sortowania małej liczby elementów (10 000), natomiast wykazał się wydajnością dla większych zbiorów elementów. Sortowanie przez scalanie, w porównaniu do pozostałych 2 algorytmów, wypadło po środku.

5. Wyznaczone średnie wartości i mediany dla każdego zestawu danych zostały przetestowane wszystkimi zaimplementowanymi algorytmami sortowań i dodatkowo potwierdzone funkcją *std::sort*.

8. Bibliografia

- M. T. Goodrich, R. Tamassia, D. M. Mount; *Data Structures and Algorithms in C++*
- https://en.cppreference.com/w/cpp/container/vector [dostęp: 6.05.2022]
- https://www.geeksforgeeks.org/introsort-or-introspective-sort/ [dostęp: 6.05.2022]
- https://pl.wikipedia.org/wiki/Sortowanie_introspektywne [dostęp: 6.05.2022]
- http://www.algorytm.edu.pl/algorytmy-maturalne/quick-sort.html [dostęp: 6.05.2022]
- https://www.geeksforgeeks.org/merge-sort/ [dostęp: 6.05.2022]