

POLITECHNIKA ŁÓDZKA

SZYBKIE ALGORYTMY

---

# Techniki zwiększenie efektywności algorytmów

---

*Prowadzący zajęcia:*

prof. dr hab. Mykhaylo YATSYMIRSKYY

*Autor:*

Filip RYNKIEWICZ



Politechnika  
Łódzka

2016-11-24

# Spis treści

<b>1</b>	<b>Repozytorium</b>	<b>1</b>
<b>2</b>	<b>Informacje o sprzęcie testowym</b>	<b>1</b>
<b>3</b>	<b>Przyjęte własność</b>	<b>1</b>
<b>4</b>	<b>Algorytm sortowania przez wstawianie</b>	<b>2</b>
4.1	Dla dwóch elementów . . . . .	2
4.1.1	Oszacowanie złożoności . . . . .	2
4.1.1.1	Optymistyczna . . . . .	2
4.1.1.2	Pesymistyczna . . . . .	3
4.1.1.3	Średnia . . . . .	3
4.1.2	Kod . . . . .	4
4.2	Dla trzech elementów . . . . .	5
4.2.1	Oszacowanie złożoności . . . . .	5
4.2.1.1	Optymistyczna . . . . .	5
4.2.1.2	Pesymistyczna . . . . .	5
4.2.1.3	Średnia . . . . .	6
4.2.2	Kod . . . . .	6
4.3	Wyniki . . . . .	8
<b>5</b>	<b>Algorytm sortowania bąbelkowego</b>	<b>10</b>
5.1	Dla dwóch elementów . . . . .	10
5.2	Oszacowanie złożoności . . . . .	10
5.2.0.1	Optymistyczna . . . . .	10
5.2.0.2	Pesymistyczna . . . . .	11
5.2.0.3	Średnia . . . . .	11
5.2.1	Kod . . . . .	11
5.3	Dla trzech elementów . . . . .	12
5.3.1	Oszacowanie złożoności . . . . .	12
5.3.1.1	Optymistyczna . . . . .	12
5.3.1.2	Pesymistyczna . . . . .	13
5.3.1.3	Średnia . . . . .	13
5.3.2	Kod . . . . .	14
5.4	Wyniki . . . . .	15

<b>6</b>	<b>Algorytm sortowania przez wybieranie</b>	<b>17</b>
6.1	Dla dwóch elementów . . . . .	17
6.1.1	Oszacowanie złożoności . . . . .	17
6.1.1.1	Optymistyczna . . . . .	17
6.1.1.2	Pesymistyczna . . . . .	17
6.1.1.3	Średnia . . . . .	18
6.1.2	Kod . . . . .	18
6.2	Wyniki . . . . .	19

## 1 Repozytorium

Cały kod oraz sprawozdanie jest dostępne na repozytorium: <https://github.com/MenosGrandes/Doktoranckie>

## 2 Informacje o sprzęcie testowym

Do pomiaru czasu zostały wykorzystane funkcje *QueryPerformanceCounter* oraz *QueryPerformanceFrequency* z biblioteki **windows.h**. Wszystkie testy zostały wykonane na maszynie z systemem Windows 8.1Pro 64, z procesorem Intel(R) Core(TM) i7-5700HQ CPU @ 2.70GHz. Testy zostały przeprowadzone na kompilatorze *mingw32-c++.exe (GCC) 5.3.0* z flagą *-std=gnu++11* zapewniającą wsparcie *C++11*.

## 3 Przyjęte własność

Jako optymistyczny przypadek została przyjęta sytuacja w której algorytm ma za zadanie posortować już posortowaną tablice.

Jako pesymistyczny przypadek przyjmuję się sytuacje w której algorytm ma posortować tablice posortowaną w odwrotnym kierunku niż pożądaną.

Jako średni przypadek zostało przyjęta sytuacja w której tablica zawiera w sobie elementy losowe.

Każde porównanie algorytmów zostało przeprowadzone na tych samych danych wejściowych. Wszystkie algorytmy, te zmodyfikowane oraz te dla par, zostały napisane przez autora.

## 4 Algorytm sortowania przez wstawianie

### 4.1 Dla dwóch elementów

Każde iteracja zaczyna się od elementu  $V[i + 2]$ <sup>1</sup>.

Pierwszym krokiem tego sortowania będzie wybranie pary  $(x', y')$ . Po wybraniu pary  $(x', y')$ , jest ona wstępnie sortowana rosnąco, potem następuje porównywanie elementu  $y'$  z elementem  $z$ , który poprzedza wybraną parę oraz indeks  $z \in |V|$ . Dopóki  $z > y'$  wykonuje się przesunięcie całej pary przed liczbę  $z$ . Za każdym razem liczba  $z$  jest liczbą poprzedzającą liczbę  $x'$ . Jeżeli  $z < y'$  algorytm przechodzi do porównania  $z > x'$ . Jeżeli zostanie spełniony ten warunek liczba mniejsza z pary zostaje przestawiona przed liczbą  $z$ .

#### 4.1.1 Oszacowanie złożoności

##### 4.1.1.1 Optymistyczna

W tym przypadku złożoność obliczeniowa będzie wynosiła  $O(n)$ . Wynika to z faktu że :

- Pierwsza pętla *for* (4 linijka) zostanie wykonana dokładnie  $\frac{n}{2}$  razy.
- Porównanie elementów w parze (6 linijka), i posortowanie jej zawsze zajmie 1 porównanie.
- Pierwsza zagnieżdżona pętla *while* (13 linijka) zostanie wykonana 0 razy.
- Druga zagnieżdżona pętla *while* (20 linijka) zostanie wykonana 0 razy.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane 0 iteracji (31 linijka).

$$T \approx \frac{n}{2} \tag{1}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

Zerowa ilość przejść w przypadku wszystkich pętli *while* wynika z faktu że zawarte w nich instrukcje zawsze będą warunkiem kończącym pętlę, zatem nigdy nie zostaną wykonane.

---

<sup>1</sup>Indeks  $i + 2$  ze zbioru  $V$ , gdzie  $i$  jest kolejna iteracja algorytmu.

#### 4.1.1.2 Pesymistyczna

Złożoność obliczeniowa będzie wynosiła  $O(n^2)$ .

- Pierwsza pętla *for*(4 linijka) zostanie wykonana dokładnie  $\frac{n}{2}$  razy.
- Porównanie elementów w parze(6 linijka), i posortowanie jej zawsze zajmie 1 porównania.
- Pierwsza zagnieżdżona pętla *while*(13 linijka) zostanie wykonana  $\frac{n}{2} - 1$  razy.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane  $n$  iteracji(31 linijka).

$$T \approx \frac{n^2}{4}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

Ponieważ obie pętle *while* korzystają z tego samego iteratora zawsze zostanie wywołana tylko pierwsza pętla, w tym przypadku.

#### 4.1.1.3 Średnia

Złożoność obliczeniowa będzie wynosiła  $O(n^2)$ .

- Pierwsza pętla *for*(4 linijka) zostanie wykonana dokładnie  $\frac{n}{2}$  razy.
- Porównanie elementów w parze(6 linijka), i posortowanie jej zawsze zajmie 1 porównania.
- Pierwsza oraz druga pętla *while*(13 i 20 linijka) zostaną wykonane łącznie  $\approx \frac{n-1}{2}$  razy, ponieważ zależne są od tego samego iteratora.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane  $n$  iteracji(31 linijka).

$$T \approx \frac{n}{2} \left(1 + \frac{n-1}{2}\right) + n$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 4.1.2 Kod

Algorytm 1: Sortowanie przez wstawianie dla par

```
1 void sort(std::vector<int> &toSort)
2 {
3     const int sizeOfArray=toSort.size()-(toSort.size()%2);
4     for(int i=0; i<sizeOfArray; i+=2)
5     {
6         if(toSort[i] > toSort[i+1])
7         {
8             std::swap(toSort[i],toSort[i+1]);
9         }
10        const int pom1 = toSort[i];
11        const int pom2 = toSort[i+1];
12        int j = i-1;
13        while(j>=0 && toSort[j]>pom2)
14        {
15            toSort[j+2] = toSort[j];
16            j--;
17        }
18        toSort[j+2] = pom2;
19        toSort[j+1] = pom1;
20        while(j>=0 && toSort[j]>pom1)
21        {
22            toSort[j+1] = toSort[j];
23            --j;
24        }
25        toSort[j+1] = pom1;
26    }
27    if(toSort.size()%2==1)
28    {
29        const int pom = toSort[toSort.size()-1];
30        int k = toSort.size()-2;
31        while(k>=0 && toSort[k]>pom)
32        {
33            toSort[k+1] = toSort[k];
34            --k;
35        }
36        toSort[k+1] = pom;
37    }
38 }
```

## 4.2 Dla trzech elementów

Każde iteracja zaczyna się od elementu  $V[i + 3]^2$ .

Pierwszym krokiem tego sortowania będzie wybranie trojek  $(x', y', z)$ . Po wybraniu trojki, jest ona wstępnie sortowana rosnąco. Wybierz liczbę  $w$  która zawsze jest elementem poprzedzającym najmniejszy element z trojki  $(x', y', z')$ . Następnie zostaną wykonane następujące akcje :

- Dopóki  $w > z'$  przesun całą trojkę przed  $w$ , wybierz nowe  $w$ . Jeżeli warunek zostanie niespełniony przejdź do następnego kroku.
- Dopóki  $w > y'$  przesun parę  $(x', y')$  przed  $w$ , wybierz nowe  $w$ . Jeżeli warunek zostanie niespełniony przejdź do następnego kroku.
- Dopóki  $w > x'$  przesun  $x'$  przed  $w$ , wybierz nowe  $w$ . Jeżeli warunek zostanie niespełniony przejdź do kolejnej iteracji.

### 4.2.1 Oszacowanie złożoności

#### 4.2.1.1 Optymistyczna

Złożoność obliczeniowa będzie wynosiła  $O(n)$ .

- Jeżeli tablica będzie o rozmiarze nieparzystym to dodatkowo pętla *for*(6 linijka) wykona się  $n\%3$  razy.
- Pierwszy *for* zostanie wykonany  $\frac{n-(n\%3)}{3}$  razy.

$$T \approx n\%3 + \frac{n - (n\%3)}{3}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

#### 4.2.1.2 Pesymistyczna

Złożoność obliczeniowa będzie wynosiła  $O(n^2)$ .

- Jeżeli tablica będzie o rozmiarze nieparzystym to dodatkowo pętla *for*(6 linijka) wykona się  $n\%3$  razy.

---

<sup>2</sup>Indeks  $i + 3$  ze zbioru  $V$ , gdzie  $i$  jest kolejna iteracja algorytmu.

- Pierwszy *for* zostanie wykonany  $\frac{n-n\%3}{3}$  razy.
- Instrukcja warunkowa *else*(27 linijka) zostanie wykonana 1 raz.
- Pętla *while*(linijka 46) zostanie wykonana  $\frac{n-n\%3}{3} + 1$  razy.

$$T \approx n\%3 + \frac{n - (n\%3)}{3} \cdot \left(\frac{n - n\%3}{3} + 2\right)$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

#### 4.2.1.3 Średnia

Złożoność obliczeniowa będzie wynosiła  $O(n^2)$ .

- Jeżeli tablica będzie o rozmiarze nieparzystym to dodatkowo pętla *for*(6 linijka) wykona się  $n\%3$  razy.
- Pierwszy *for* zostanie wykonany  $\frac{n-n\%3}{3}$  razy.
- Średnio wszystkie instrukcje warunkowe wykonają się 3 razy
- Wszystkie pętle *while* wykonają się  $\frac{n-n\%3}{3}$  razy, ponieważ operują na tym samym iteratorze.

$$T \approx n\%3 + \frac{n - (n\%3)}{3} \cdot \left(\frac{n - n\%3}{3} + 1\right)$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

#### 4.2.2 Kod

Algorytm 2: Sortowanie przez wstawianie dla trójek

```

1 void sort(std::vector<int>&toSort)
2 {
3     const int arrayDivider = (toSort.size()%3);
4     if(toSort.size()%3!=0)
5     {
6         for(int i=1; i<arrayDivider; i++)

```



```

7   {
8       const int pom = toSort[i];
9       int j = i-1;
10      while(j>=0 && toSort[j]>pom)
11      {
12          toSort[j+1] = toSort[j];
13          --j;
14      }
15      toSort[j+1] = pom;
16  }
17  }
18  for(int i=arrayDivider; i<toSort.size(); i+=3)
19  {
20      if(toSort[i] < toSort[i+1])
21      {
22          if(toSort[i+2]<toSort[i])
23          {
24              std::swap(toSort[i],toSort[i+2]);
25          }
26      }
27      else
28      {
29          if(toSort[i+1]<toSort[i+2])
30          {
31              std::swap(toSort[i],toSort[i+1]);
32          }
33          else
34          {
35              std::swap(toSort[i],toSort[i+2]);
36          }
37      }
38      if(toSort[i+2]<toSort[i+1])
39      {
40          std::swap(toSort[i+1],toSort[i+2]);
41      }
42      const int pom1 = toSort[i];
43      const int pom2 = toSort[i+1];
44      const int pom3 = toSort[i+2];
45      int j = i-1;
46      while(j>=0 && toSort[j]>pom3)
47      {
48          toSort[j+3] = toSort[j];
49          j--;
50      }
51      toSort[j+3] = pom3;

```

```

52     toSort[j+2] = pom2;
53     toSort[j+1] = pom1;
54     while(j>=0 && toSort[j]>pom2)
55     {
56         toSort[j+2] = toSort[j];
57         j--;
58     }
59     toSort[j+2] = pom2;
60     toSort[j+1] = pom1;
61     while(j>=0 && toSort[j]>pom1)
62     {
63         toSort[j+1] = toSort[j];
64         --j;
65     }
66     toSort[j+1] = pom1;
67 }
68 };

```

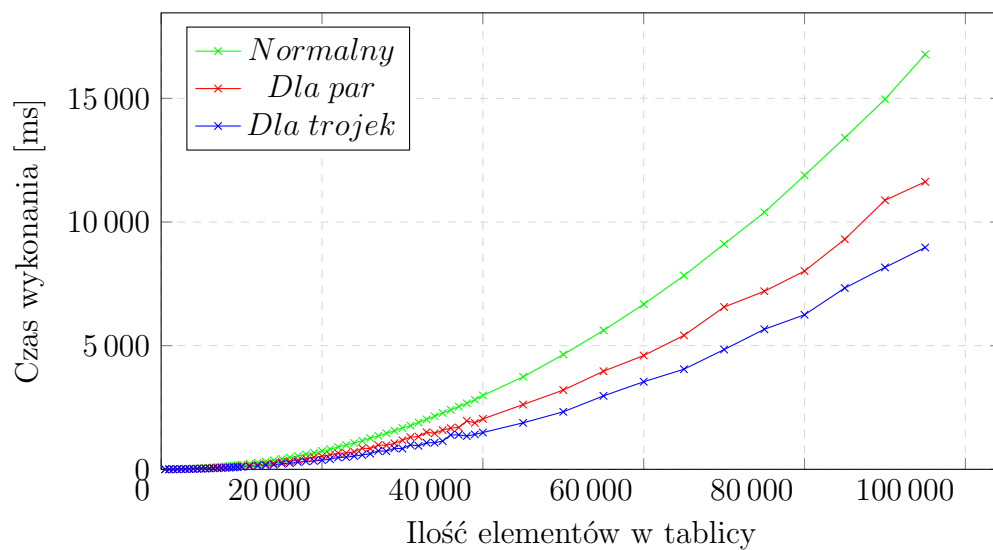
### 4.3 Wyniki

Rys. 1 przedstawia wyniki dla tablicy z losowymi elementami. Na wykresie można zauważyć przyspieszenie względem standardowego algorytmu. Przyspieszenie to wynosi  $\approx 34\%$ . Opisana wcześniej złożoność dla tego przypadku jest taka sama jak dla standardowej implementacji. Jednakże fakt że algorytm bierze pod uwagę parę i najpierw sortuje po największym elemencie a potem po najmniejszym zmniejsza znacząco ilość porównań. Doprowadza to do tego że teoretyczne i maksymalne przyspieszenie powinno wynieść  $\approx 50\%$  przy tym pomyśle, uwzględniając idealną implementację. Dla trojek przyspieszenie wynosi  $\approx 50\%$ . Ponieważ nałożone są dodatkowe instrukcje warunkowe do posortowania trzech elementów, oraz jedna dodatkowa pętla, oszacowanie teoretyczne jest o wiele większe niż wyniki praktyczne.

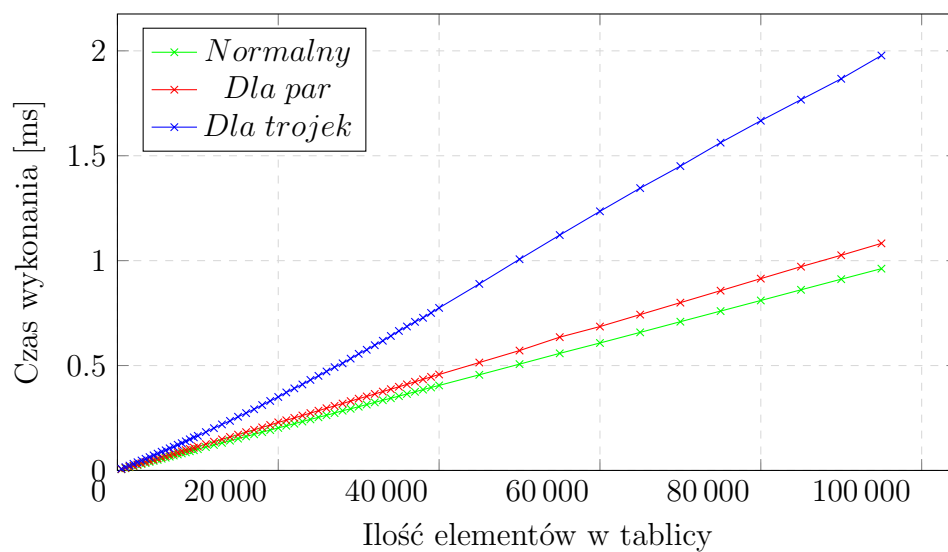
Rys. 2 przedstawia wyniki dla tablicy która została już posortowana. Jak można zauważyć implementacja standardowa jest szybsza niż zaimplementowana tutaj. Ponieważ ilość porównań została zwiększona, lecz te porównania zostają tylko wykonane raz i ominięte, algorytmy dla dwójki i trojki są znacznie wolniejsze w tym przypadku. Jednakże biorąc pod uwagę liniowość wszystkich trzech funkcji różnica może być pominięta.

Rys. 3 przedstawia wyniki dla pesymistycznego ułożenia tablicy. W tym wypadku implementacja algorytmu dla par działa o  $\approx 50\%$  szybciej od standardowej implementacji. Wynika to z faktu że, tak samo jak w średnim przypadku, iteracja pętla odpowiedzialna za przestawianie par wynosi  $\frac{n-2}{2}$  przez

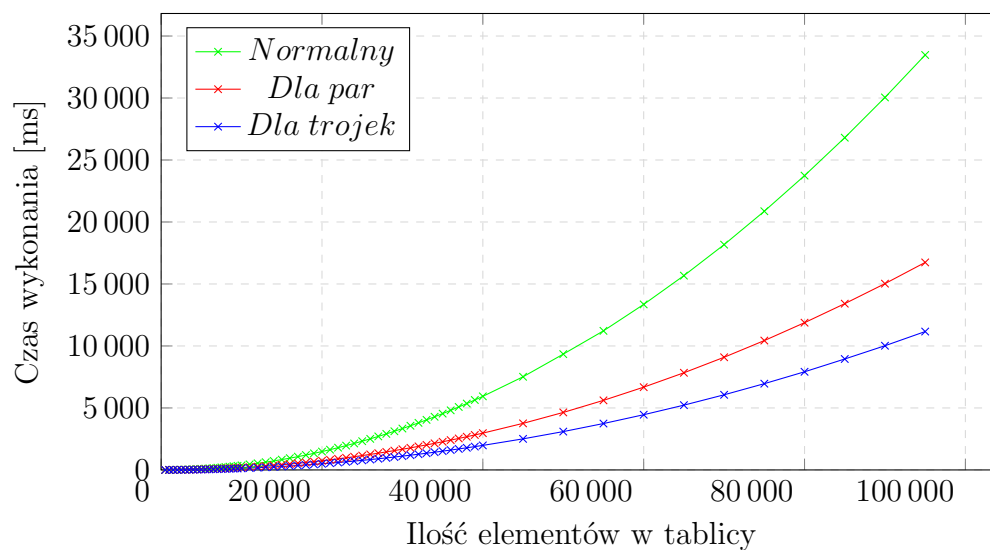
co faktycznie algorytm przyspiesza do  $\approx 150\%$  szybkości algorytmu podstawowego. Wersja dla trzech elementów jest szybsza o 68% względem wersji podstawowej. Jednakże teoretycznie algorytm powinien znacznie bardziej przyspieszyć, nie dzieje się tak ponieważ zostały nałożone dodatkowe instrukcje warunkowe oraz dodatkowe zmienne przez co procesor musi wykonać kilkanaście dodatkowych operacji.



Rys. 1: Wykres dla tablicy ze elementami losowymi



Rys. 2: Wykres dla tabeli z najlepszym rozkładem elementów



Rys. 3: Wykres dla tabeli z najgorszym rozkładem elementów

## 5 Algorytm sortowania bąbelkowego

### 5.1 Dla dwóch elementów

Podstawowa wersja tego algorytmu polega na porównywaniu ze sobą dwóch kolejnych elementów  $(x, y) \in V$  i zmianie ich kolejności, mając tylko jeden bąbelek który wypływa na początek lub na koniec zbioru.

Zakładając że mamy porównywać dwie liczby, zostało przyjęte że są dwa bąbelki. Jeden który idzie na początek zbioru  $V$  oraz drugi który idzie na koniec zbioru  $V$ .

Dla każdej pary  $(x', y') \in V$  składającej się z kolejnych elementów ze zbioru  $V$ :

- Posortuj parę  $(x', y')$  rosnąco
- Dla każdej liczby  $z$  poprzedzającej  $x'$  zamień ze sobą te elementy jeżeli

$$x' < z$$

- Dla każdej liczby  $w$  następującej  $y'$  zamień ze sobą te elementy jeżeli

$$y' > w$$

### 5.2 Oszacowanie złożoności

#### 5.2.0.1 Optymistyczna

Najlepszy przypadek tego algorytmu ma złożoność obliczeniową  $O(n)$ .

- Pierwsza pętla(3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 1$

$$T \approx n - 1$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 5.2.0.2 Pesymistyczna

Najgorszy przypadek algorytmu sortowania bąbelkowego dla par ma złożoność obliczeniową  $O(n^2)$ .

- Pierwsza pętla(3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 1$
- Pętla *while*(10 linijka) w najgorszym przypadku wykona się  $n$  razy
- Pętla *while*(15 linijka) w najgorszym przypadku wykona się  $n$  razy

$$T \approx (n - 1) \cdot (2n)$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 5.2.0.3 Średnia

Dla losowych elementów w tablicy sortowanie to ma złożoność obliczeniową  $O(n^2)$ .

- Pierwsza pętla(3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 1$
- Pętla *while*(10 linijka) w tym przypadku wykona się  $\frac{2}{n}$  razy
- Pętla *while*(15 linijka) w tym przypadku wykona się  $\frac{2}{n}$  razy

$$T \approx (n - 1) \cdot \left(\frac{2}{n} + \frac{2}{n}\right) = (n - 1)n$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 5.2.1 Kod

Algorytm 3: Sortowanie bąbelkowe dla par

```
1 void sort(std::vector<int> &toSort)
2 {
3     for(int i= 0; i<(toSort.size()-1); i++)
4     {
5         int minElem=i,maxElem=i+1;
6         if(toSort[minElem]>toSort[maxElem])
```

```

7      {
8          std::swap(toSort[minElem], toSort[maxElem]);
9      }
10     while(minElem > 0 && toSort[minElem] < toSort[minElem-1])
11     {
12         std::swap(toSort[minElem], toSort[minElem-1]);
13         minElem--;
14     }
15     while(maxElem < (toSort.size()-1) && toSort[maxElem] > toSort[
        ↪ maxElem+1])
16     {
17         std::swap(toSort[maxElem], toSort[maxElem+1]);
18         maxElem++;
19     }
20 }
21 }

```

### 5.3 Dla trzech elementów

Pierwszym krokiem tego algorytmu jest pobranie trzech sąsiednich elementów  $(x, y, z) \in V$  i posortowanie ich w kolejności rosnącej tworząc trojkę  $(x', y', z')$ . Liczba  $w$  jest zawsze liczba poprzedzająca liczbę  $x'$ , a liczba  $g$  jest liczba poprzedzająca  $z'$ .

- Dopóki  $y' < w$  przed liczbę  $w$  wstaw parę  $(x', y')$
- Dopóki  $x' < w$  przed liczbę  $w$  wstaw  $x'$
- Dopóki  $z' > g$  przed liczbę  $g$  wstaw  $z'$

#### 5.3.1 Oszacowanie złożoności

##### 5.3.1.1 Optymistyczna

Najlepszy przypadek tego algorytmu ma złożoność obliczeniową  $O(n)$ .

- Pierwsza pętla (3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 2$

$$T \approx n - 2$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 5.3.1.2 Pesymistyczna

Najlepszy przypadek tego algorytmu ma złożoność obliczeniową  $O(n^2)$ .

- Pierwsza pętla(3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 2$
- Jeżeli rozmiar tablicy jest liczbą parzystą to :
  - Pierwsza pętla *while*(28 linijka) wykona się  $\frac{(\frac{n}{2}-1)^2+(\frac{n}{2}-1)}{2}$  razy
  - Druga pętla *while*(35 linijka) wykona się  $\frac{n}{2}$  razy
  - Trzecia pętla *while*(40 linijka) wykona się  $(\frac{n}{2})^2$  razy
- Jeżeli rozmiar tablicy jest liczbą nieparzystą to :
  - Pierwsza pętla *while*(28 linijka) wykona się  $\approx (\frac{n}{2})^2$ , z zaokrągleniem w dół
  - Druga pętla *while*(35 linijka) wykona się 0 razy
  - Trzecia pętla *while*(40 linijka) wykona się  $(\frac{n}{2})^2$  razy, z zaokrągleniem w dół

$$T_1 \approx \frac{(\frac{n}{2}-1)^2 + (\frac{n}{2}-1)}{2} + \frac{n}{2} + (\frac{n}{2})^2$$
$$T_2 \approx (\frac{n}{2})^2 + (\frac{n}{2})^2$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 5.3.1.3 Średnia

- Pierwsza pętla(3 linijka) *for* jest zależna od rozmiaru tablicy i zostanie wykonana dokładnie  $n - 2$
- Ponieważ wszystkie pętle *while* posiadają dwa warunki a jeden z nich jest dla nich wspólny to średnio połowa wartości dla najgorszego przypadku.

$$T_1 \approx \frac{1}{2} \cdot \left( \frac{(\frac{n}{2}-1)^2 + (\frac{n}{2}-1)}{2} + \frac{n}{2} + (\frac{n}{2})^2 \right)$$



### 5.3.2 Kod

Algorytm 4: Sortowanie bąbelkowe dla trzech elementów

```
1 void sort(std::vector<int>&toSort)
2 {
3     for(int i= 0; i<toSort.size()-2; i++)
4     {
5         if(toSort[i] < toSort[i+1])
6         {
7             if(toSort[i+2]<toSort[i])
8             {
9                 std::swap(toSort[i],toSort[i+2]);
10            }
11        }
12        else
13        {
14            if(toSort[i+1]<toSort[i+2])
15            {
16                std::swap(toSort[i],toSort[i+1]);
17            }
18            else
19            {
20                std::swap(toSort[i],toSort[i+2]);
21            }
22        }
23        if(toSort[i+2]<toSort[i+1])
24        {
25            std::swap(toSort[i+1],toSort[i+2]);
26        }
27        int minElem=i,minElem2=i+1,maxElem=i+2;
28        while(minElem2>1 && toSort[minElem2]<toSort[minElem-1])
29        {
30            std::swap(toSort[minElem],toSort[minElem-1]);
31            std::swap(toSort[minElem2],toSort[minElem2-1]);
32            minElem--;
33            minElem2--;
34        }
35        while(minElem>0 && toSort[minElem]<toSort[minElem-1])
36        {
37            std::swap(toSort[minElem],toSort[minElem-1]);
38            minElem--;
39        }
40        while(maxElem<(toSort.size()-1) && toSort[maxElem]>toSort[
    ↪ maxElem+1])
```

```

41     {
42         std::swap(toSort[maxElem], toSort[maxElem+1]);
43         maxElem++;
44     }
45 }
46 }

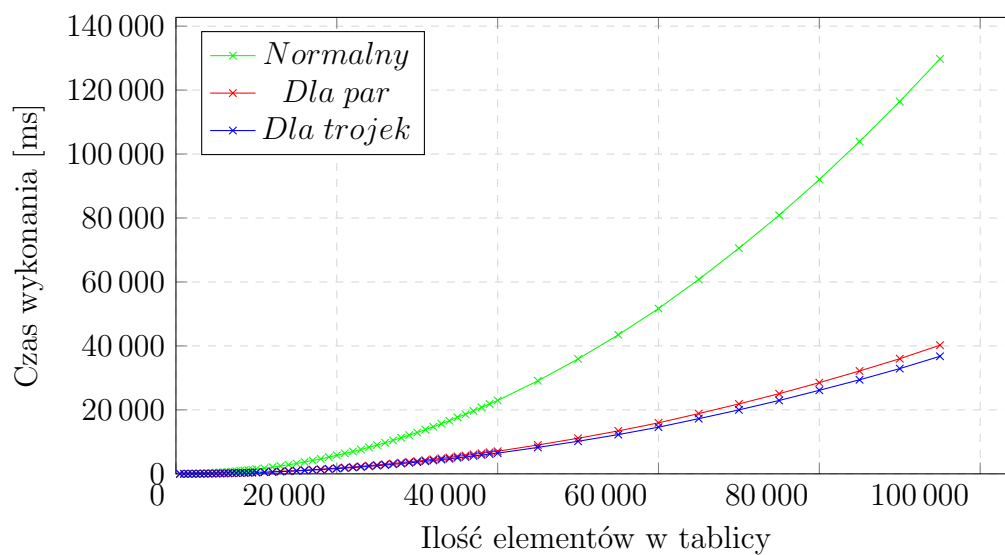
```

## 5.4 Wyniki

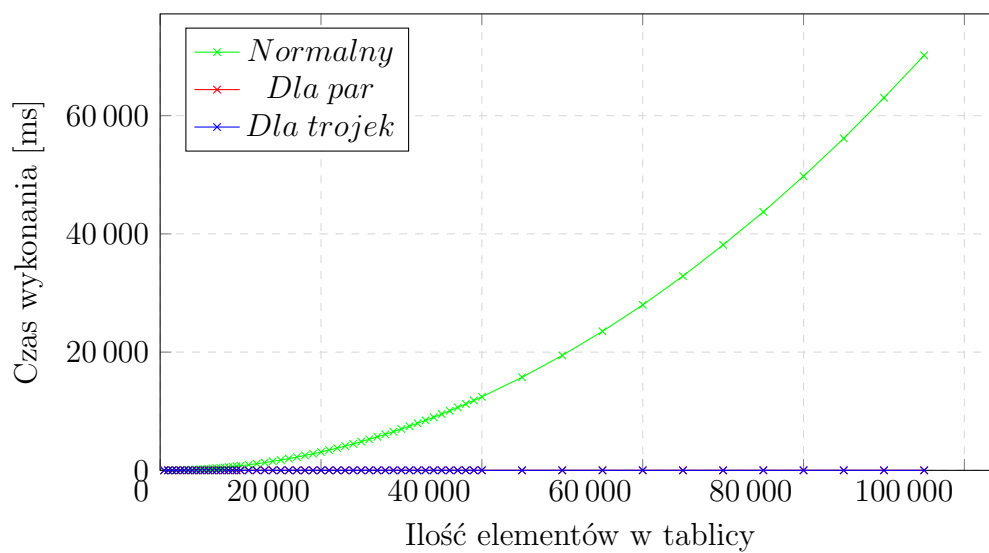
Rys. 4 przedstawia efekt porównania działania obu algorytmów dla tablic z losowymi elementami. Z przedstawionego wykresu wyraźnie widać przyspieszenie. Maksymalne przyspieszenie dla tego algorytmu wynosi  $\approx 69\%$ . Dzięki zastosowaniu wewnętrznych pętli *while* które sterują zachowaniem bąbelków można bardzo szybko przyspieszyć algorytm. Dzięki temu że średnio każda pętla wykonuje się połowę rozmiaru tablicy. Wiece teoretyczne przyspieszenie powinno być co najmniej 50%. Dla trojek przyspieszenie wynosi średnio 74%, ponieważ przesuwanie całej pary skutkuje usunięciem jednego porównania. Dodatkowo przyspieszenie zależne jest od tego czy rozmiar tablicy jest liczba parzysta czy nieparzysta, ponieważ w przypadku parzystych zostaje wykonane mniej porównań.

Rys. 5 reprezentuje graficznie otrzymane wyniki z porównania algorytmu sortowania bąbelkowego. Ponieważ każda z funkcji ma złożoność bliska  $O(n)$ , wiece przyspieszenie wynosi  $\approx 99\%$ .

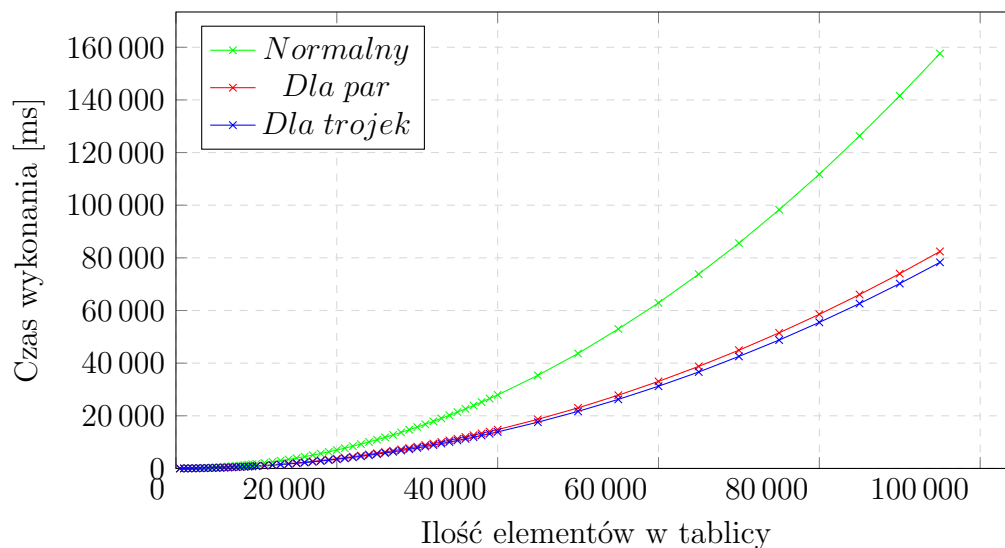
Rys. 6 przedstawia wyniki dla tablicy posortowanej w druga stronę. Przyspieszenie algorytmu wynosi w tym wypadku  $\approx 52\%$ . Ponieważ używamy dwóch bąbelków, w porównaniu do użycia jednego, zmniejsza czas wykonania o 50%. Dla trojek przyspieszenie wynosi  $\approx 55\%$ . Ponieważ dodatkowa para bąbelków w dol jednocześnie przenosi parę powstaje kilka porównań mniej.



Rys. 4: Wykres dla tablicy ze elementami losowymi



Rys. 5: Wykres dla tablicy z najlepszym rozkładem elementów



Rys. 6: Wykres dla tablicy z najgorszym rozkładem elementów

## 6 Algorytm sortowania przez wybieranie

### 6.1 Dla dwóch elementów

Podstawowa wersja algorytmu zakłada wybieranie najmniejszej lub największej wartości z zbioru. Ten pomysł wybiera największą oraz najmniejszą wartość.

- Pobierz element który znajduje się na początku, *begin*, zbioru  $V$ .
- Pobierz element który znajduje się na końcu, *end*, zbioru  $V$ .
- Zakładając że pobieramy element najmniejszy *min* oraz element największy *max* ze zbioru  $V$ , zamień *min* z *begin* a *max* z *end*.
- Ustaw początek zbioru *begin* na element z indeksem o jeden większy niż *begin*, a koniec na element z indeksem o jeden mniejszym niż *end*.

#### 6.1.1 Oszacowanie złożoności

**6.1.1.1 Optymistyczna** Najlepszy przypadek tego algorytmu ma złożoność obliczeniową  $O(n^2)$ .

- Pętla *while*(11 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ przy każdej iteracji usuwane są 2 elementy.
- Pętla *for*(14 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ zawsze wykonuje się tyle samo razy co poprzednia.

$$T \approx \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

**6.1.1.2 Pesymistyczna** Najgorszy przypadek algorytmu sortowania przez wybieranie dla par ma złożoność obliczeniową  $O(n^2)$ .

- Pętla *while*(11 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ przy każdej iteracji usuwane są 2 elementy.
- Pętla *for*(14 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ zawsze wykonuje się tyle samo razy co poprzednia.

$$T \approx \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

**6.1.1.3 Średnia** Dla losowych elementów w tablicy sortowanie to ma złożoność obliczeniową  $O(n^2)$ .

- Pętla *while*(11 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ przy każdej iteracji usuwane są 2 elementy.
- Pętla *for*(14 linijka) wykonuje się zawsze  $\frac{n}{2}$ , ponieważ zawsze wykonuje się tyle samo razy co poprzednia.

$$T \approx \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$$

gdzie  $T$  to oszacowana ilość operacji a  $n$  rozmiar tablicy do posortowania.

### 6.1.2 Kod

Algorytm 5: Sortowanie przez wybieranie dla par

```
1 void sort(std::vector<int> &toSort)
2 {
3     int vectorSize=0;
4     if(toSort.size()%2!=0)
5     {
6         vectorSize++;
7         std::iter_swap((std::min_element(toSort.begin(),toSort.end
8             ↪ )),toSort.begin());
9     }
10    std::vector<int>::iterator _begin = toSort.begin()+
11        ↪ vectorSize;
12    std::vector<int>::iterator _end = toSort.end() - 1;
13    while (_begin < _end)
14    {
15        std::vector<int>::iterator it=_begin,_min=it,_max=it;
16        for (it = _begin; it <= _end; ++it)
17        {
18            if ((*it) < (*_min))
19            {
20                _min = it;
21            }
22            else if ((*it) > (*_max))
23            {
24                _max = it;
25            }
26        }
27        std::iter_swap(_min,_begin);
28        if(_begin==_max)
29        {
30            _max=_min;
31        }
32        std::iter_swap(_max,_end);
33        ++_begin;
34        --_end;
35    }
36 }
```

## 6.2 Wyniki

Jak widać na powyższych założeniach teoretycznych każdy z przykładów ma taką samą złożoność obliczeniową. Wynika to z faktu że każda pętla musi

być wykonana zawsze i wymaganą ilość razy.

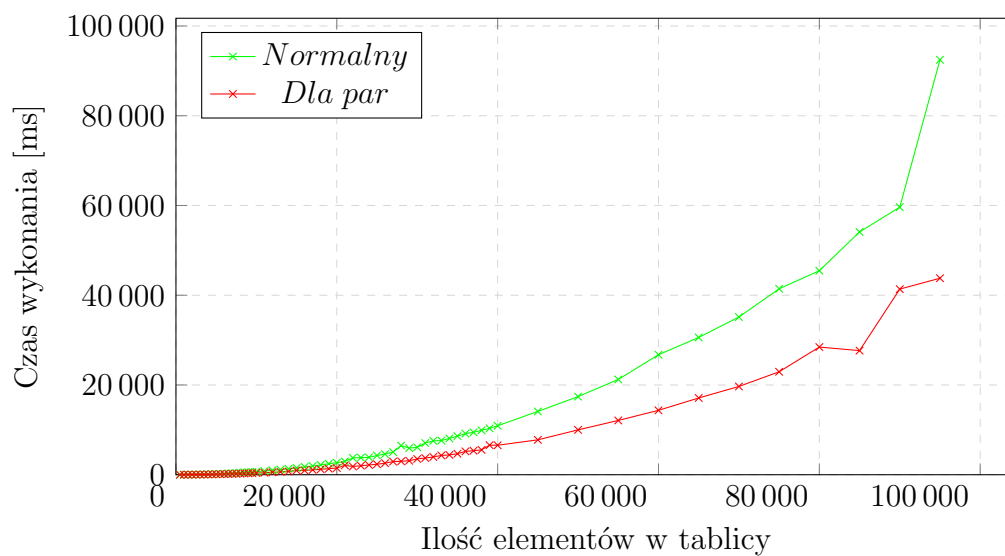
Rys.7 przedstawia porównanie szybkości wykonywania się obliczeń dla tablicy z danymi losowymi. Przyspieszenie wynosi  $\approx 40\%$ . Ponieważ średnie teoretyczne przyspieszenie powinno wynosi  $\approx 50$ , widać że nie zgadza się to z wynikami. Dzieje się tak ponieważ wykorzystuje jedna dodatkowa instrukcje warunkowa która spowalnia algorytm.

Rys.8 przedstawia najgorszy z trzech możliwych wynik gdzie przyspieszenie wynosi  $\approx 35\%$ . Jest to najgorszy wynik ponieważ tablica jest posortowana odwrotnie, przez co w poszukiwaniu najmniejszej i największej liczby występuje dużo więcej porównań niż w sytuacji z losowymi elementami.

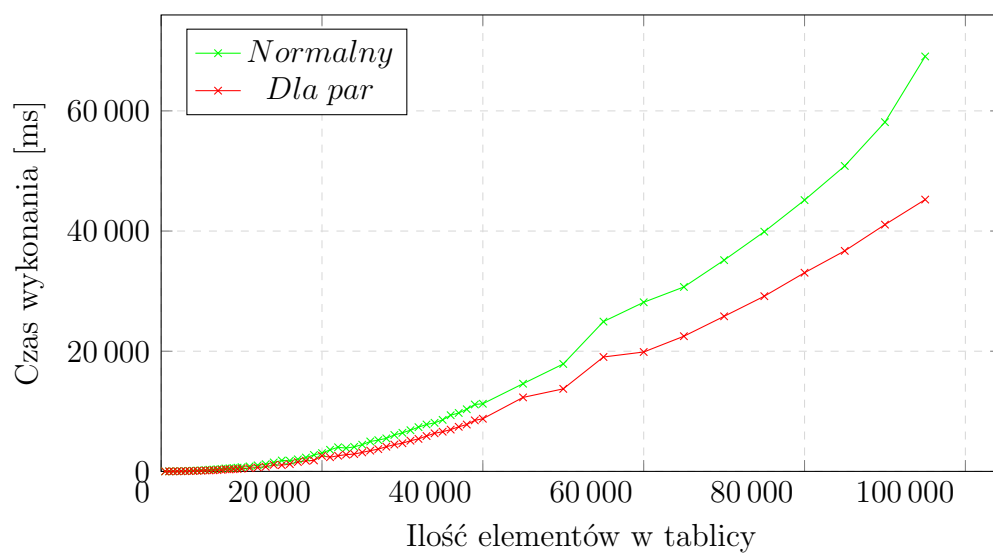
Rys.9 reprezentuje dane pomiarowe na których widać że przyspieszenie wynosi  $\approx 43\%$ .

Ponieważ każde z przypadków sortowań ma taką samą złożoność obliczeniową całe przyspieszenie jest zależne od znalezienia wartości maksymalnej i minimalnej. Ponieważ w algorytmie dla par przeszukujemy tablice wejściową i wybieramy od razu obydwie wartości, to teoretyczne przyspieszenie powinno wynieść  $\approx 50\%$ .

Ponieważ algorytm dla trojek w tym podejściu, wybierania dwóch najmniejszych oraz największej liczby ze zbioru, jest wolniejszy od podstawowego nie zostały przedstawione dla niego wyniki. Ponieważ dodatkowo musimy sprawdzać trzy wartości za każdym razem przeszukując tablice od *begin* to *end* dorzuca to dodatkowo jedna instrukcje warunkowa. Dodatkowo gdy miejsca minimalne oraz maksymalne zostanie znalezione trzeba zrobić dodatkowa zamianę co daje  $3n$  dodatkowych operacji.

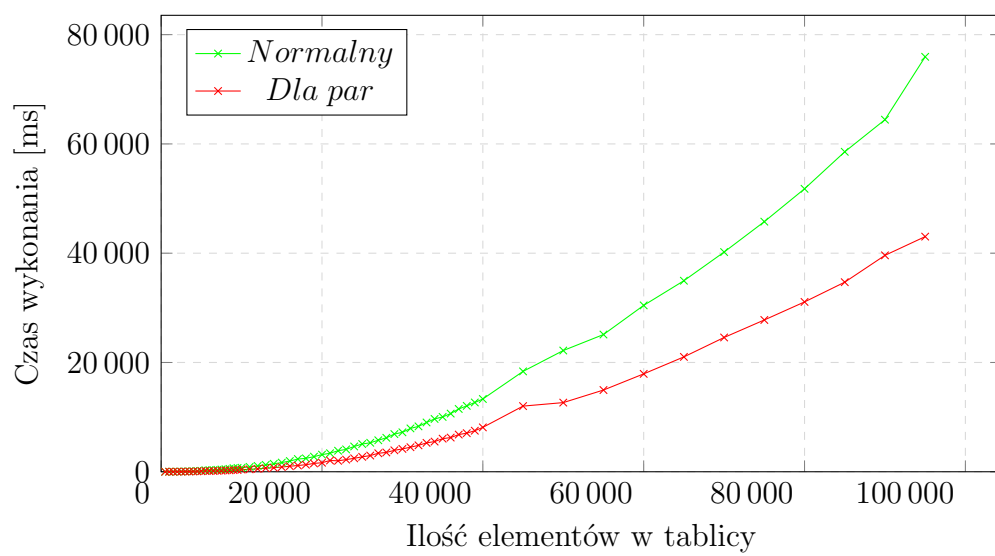


Rys. 7: Wykres dla tablicy ze elementami losowymi



Rys. 8: Wykres dla tablicy z najlepszym rozkładem elementów





Rys. 9: Wykres dla tablicy z najgorszym rozkładem elementów