

POLITECHNIKA ŁÓDZKA

SZYBKIE ALGORYTMY

Techniki zwiększenie efektywności algorytmów

Prowadzący zajęcia:

prof. dr hab. Mykhaylo YATSYMIRSKYY

Autor:

Filip RYNKIEWICZ



Politechnika
Łódzka

2016-11-16

1 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*.

2 Przyjęte własność

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

Jako pesymistyczny przypadek przyjmuję się sytuacje w której algorytm ma posortować tablicę 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.

3 Algorytm sortowania przez wstawianie

Dla dwóch elementów

Początkowym krokiem algorytmu jest posortowanie par (x, y) w zbiorze V , gdzie każda para $(x, y) \in V$, tak aby pierwsza liczba x była zawsze liczba mniejsza od liczby y . Indeksy liczby x jest zawsze o jeden mniejszy od indeksu liczby y w zbiorze.

Kolejnym krokiem tego sortowania będzie wybranie pary (x', y') . Pary wybierane są poprzez przesuwanie od indeksu 2 zbioru V , ponieważ zakładamy że pierwsza para jest posortowana, zawsze o 2 indeksy. Każde przejście zaczyna się od elementu $V[i + 2]$ ¹.

Po wybraniu pary (x', y') następuje porównywanie elementu y' z elementem z , który poprzedza wybrana parę oraz indeks $z \in |V|$. Dopóki $z > y'$ wykonuje

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

się przesuniecie 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 .

Oszacowanie złożoności

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.
- Druga pętla *for*(11 linijka) zostanie wykonana $\frac{n-2}{2}$ razy.
- Pierwsza zagnieżdżona pętla *while*(16 linijka) zostanie wykonana 0 razy.
- Druga zagnieżdżona pętla *while*(23 linijka) zostanie wykonana 0 razy.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane 0 iteracji(34 linijka).

$$T \approx \frac{n}{2} + \frac{n-2}{2} = n - 1 \quad (1)$$

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.

Pesymistyczna

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

- Pierwsza pętla *for*(4 linijka) zostanie wykonana dokładnie $\frac{n}{2}$ razy.
- Druga pętla *for*(11 linijka) zostanie wykonana dokładnie $\frac{n-2}{2}$ razy.
- Pierwsza zagnieżdżona pętla *while*(16 linijka) zostanie wykonana $\approx \frac{n}{2}$ razy.

- Druga zagnieżdżona pętla *while*(23 linijka) zostanie wykonana $\approx \frac{n}{2}$ razy.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane $n - 2$ iteracji(34 linijka).

$$T \approx \frac{n}{2} + \frac{n-2}{2} \cdot (n) + (n-2) = \frac{n^2}{2} + \frac{n}{2} - 2$$

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

Ponieważ obie pętle *while* korzystają z tego samego iteratora, zostało przyjęte że obie w sumie wykonają się n razy, więc średnio każda wykona się $\frac{n}{2}$.

Średnia

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

- Pierwsza pętla *for*(4 linijka) zostanie wykonana dokładnie $\frac{n}{2}$ razy.
- Druga pętla *for*(11 linijka) zostanie wykonana dokładnie $\frac{n-2}{2}$ razy.
- Pierwsza zagnieżdżona pętla *while*(16 linijka) zostanie wykonana $\approx \frac{2}{n}$ razy.
- Druga zagnieżdżona pętla *while*(23 linijka) zostanie wykonana $\approx \frac{2}{n}$ razy.
- Jeżeli tablica jest o rozmiarze nieparzystym dodatkowo zostanie wykonane $n - 2$ iteracji(34 linijka).

$$T \approx \frac{n}{2} + \frac{n-2}{2} \cdot (n) + (n-2) = \frac{n^2}{2} + \frac{n}{2} - 2$$

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

Kod

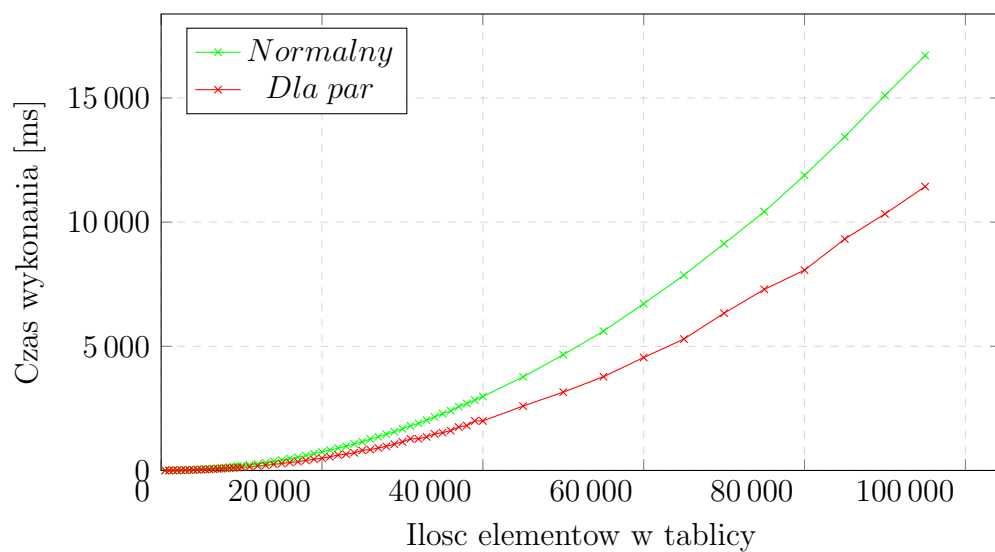
```
,
1 void sort(std::vector<int> &toSort)
2 {
3     const int sizeofArray=toSort.size()-(toSort.size()%2);
4     for(int i=0; i<sizeofArray; i+=2)
5     {
6         if(toSort[i] > toSort[i+1])
7         {
8             std::swap(toSort[i],toSort[i+1]);
9         }
10    }
11    for(int i=2; i<sizeofArray; i+=2)
12    {
13        const int pom1 = toSort[i];
14        const int pom2 = toSort[i+1];
15        int j = i-1;
16        while(j>=0 && toSort[j]>pom2)
17        {
18            toSort[j+2] = toSort[j];
19            j--;
20        }
21        toSort[j+2] = pom2;
22        toSort[j+1] = pom1;
23        while(j>=0 && toSort[j]>pom1)
24        {
25            toSort[j+1] = toSort[j];
26            --j;
27        }
28        toSort[j+1] = pom1;
29    }
30    if(toSort.size()%2==1)
31    {
32        const int pom = toSort[toSort.size()-1];
33        int k = toSort.size()-2;
34        while(k>=0 && toSort[k]>pom)
35        {
36            toSort[k+1] = toSort[k];
37            --k;
38        }
39        toSort[k+1] = pom;
40    }
41 }
```

Wyniki

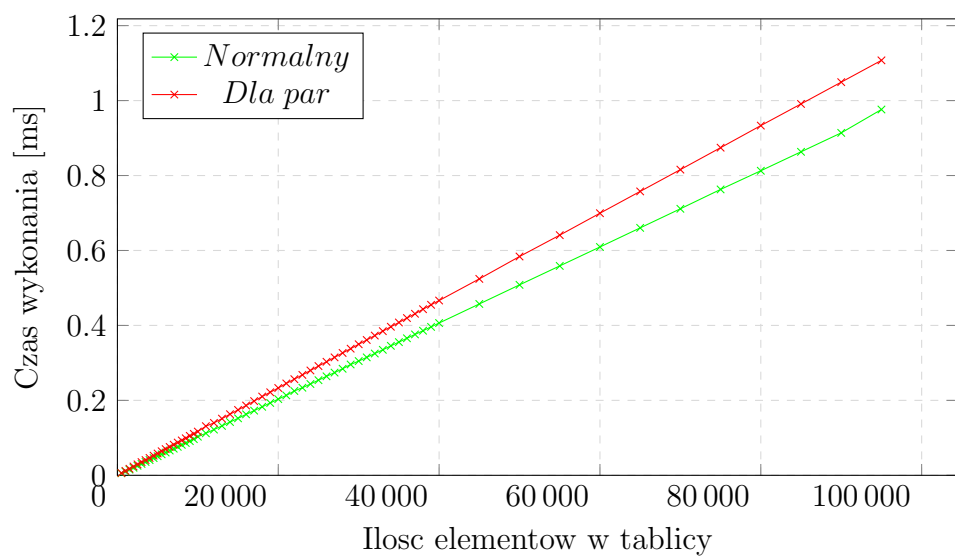
Rys. 1 przedstawia wyniki dla tablicy z losowymi elementami. Na wykresie można zauważyć przyspieszenie względem standardowego algorytmu. Przyspieszenie to wynosi $\approx 34\%$. Opisana wcześniej złożoność dla tego przypadku jest taka sama jak dla standardowej implementacji. Jednakże fakt że algorytm 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 pomysle, uwzględniając idealną implementację.

Rys. 2 przedstawia wyniki dla tablicy która została już posortowana. Jak można zauważyć implementacja standardowa jest szybsza niż zaimplementowana tutaj. Jednakże różnica w czasie działania obu funkcji wynosi około $0.1ms$, więc można uznać że są takie same. Ponieważ jedyną akcją która jest zależna od rozmiaru tablicy jest pętla przestawiająca pary oraz pętla przeszukująca tablice co dwa elementy złożoność obliczeniowa będzie liniowa $O(n)$.

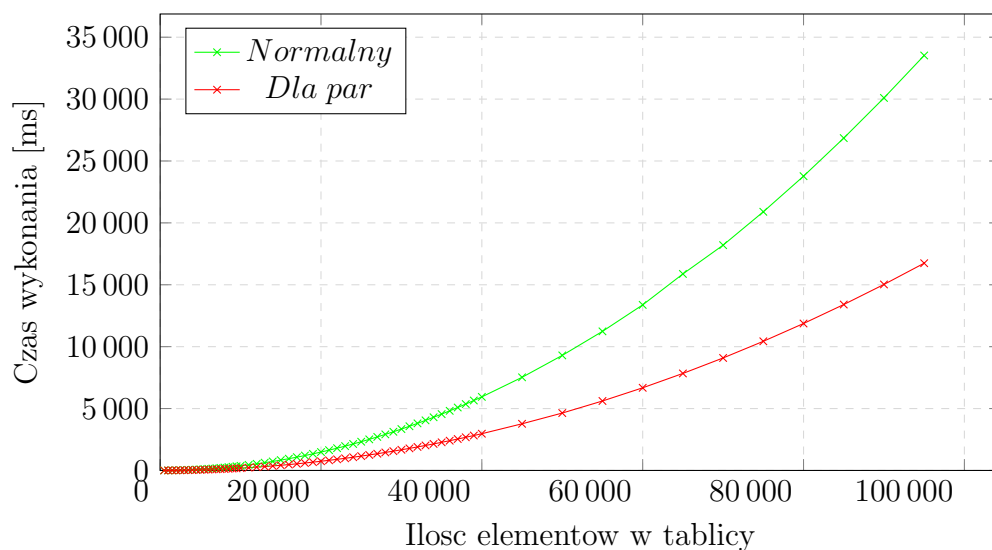
Rys. 3 przedstawia wyniki dla pesymistycznego ułożenia tablicy. W tym wypadku implementacja algorytmu dla par działa o $\approx 50\%$ szybciej od standardowej implementacji. Wynika to z faktu że, tak samo jak w średnim przypadku, iteracja pętla odpowiedzialna za przestawianie par wynosi $\frac{n-2}{2}$ przez co faktycznie algorytm przyspiesza do $\approx 150\%$ szybkości algorytmu podstawowego.



Rys. 1: Wykres dla tablicy ze elementami losowymi



Rys. 2: Wykres dla tablicy z najlepszym rozkładem elementów



Rys. 3: Wykres dla tablicy z najgorszym rozkładem elementów

4 Algorytm sortowania bąbelkowego

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

$$y' > w$$

Kod

Algorytm 1: Sortowanie babelkowe 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[
16            ↪ maxElem+1])
17        {
18            std::swap(toSort[maxElem],toSort[maxElem+1]);
19            maxElem++;
20        }
21    }
```

Oszacowanie złożoności

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.

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.

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

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

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

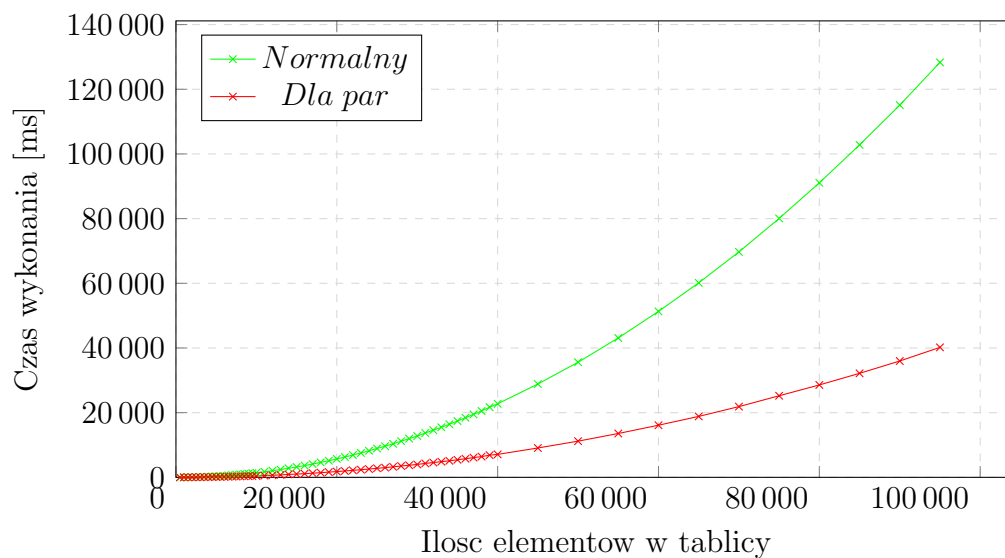
Wyniki

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 $\approx 70\%$. 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. Wier teoretyczne przyspieszenie powinno być co najmniej 50%.

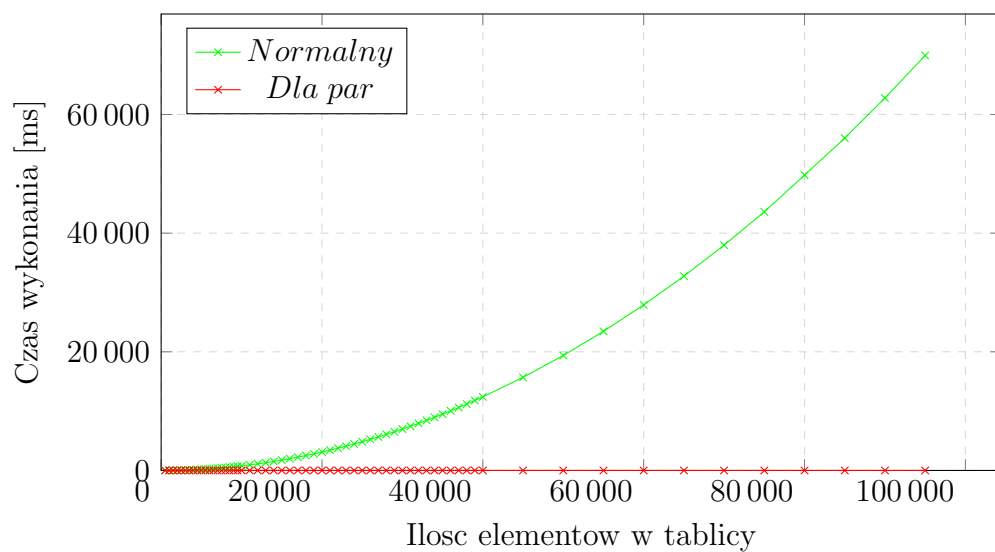
Rys. 5 reprezentuje graficznie otrzymane wyniki z porównania algorytmu sortowania bąbelkowego dla par i normalnego dla posortowanej tablicy. Ponieważ widoczna na wykresie czerwona linia, reprezentująca sortowanie dla par, jest

zawsze bliska zero, można wyciągnąć wniosek. Sortowanie to ma złożoność obliczeniową $O(n)$. Wiek przyspieszenie wynosi $\approx 99\%$

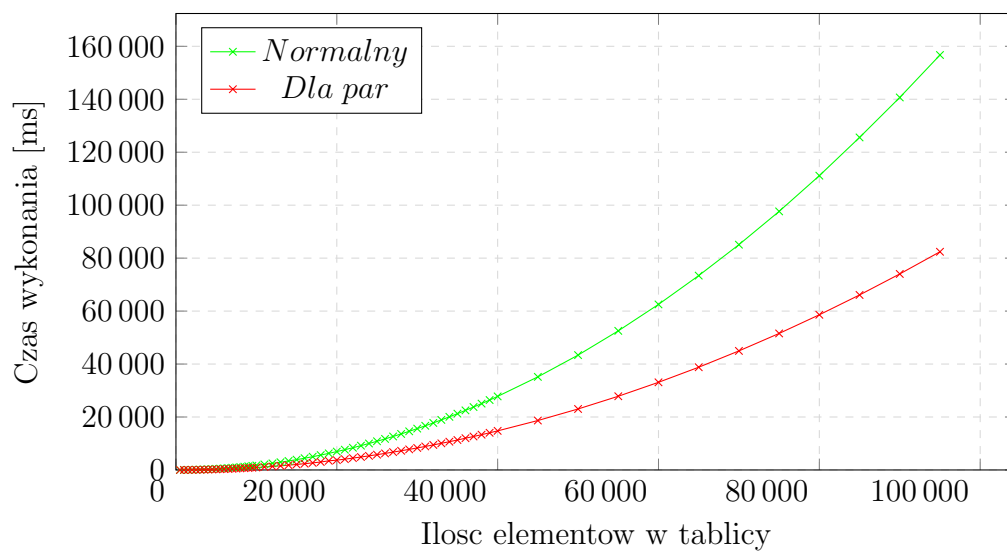
Rys. 6 przedstawia wyniki dla tablicy posortowanej w drugą stronę. Przyspieszenie algorytmu wynosi w tym wypadku $\approx 50\%$. Ponieważ używamy dwóch bąbelków, w porównaniu do użycia jednego, zmniejsza czas wykonania o 50%.



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

5 Algorytm sortowania przez wybieranie

Dla dwóch elementów

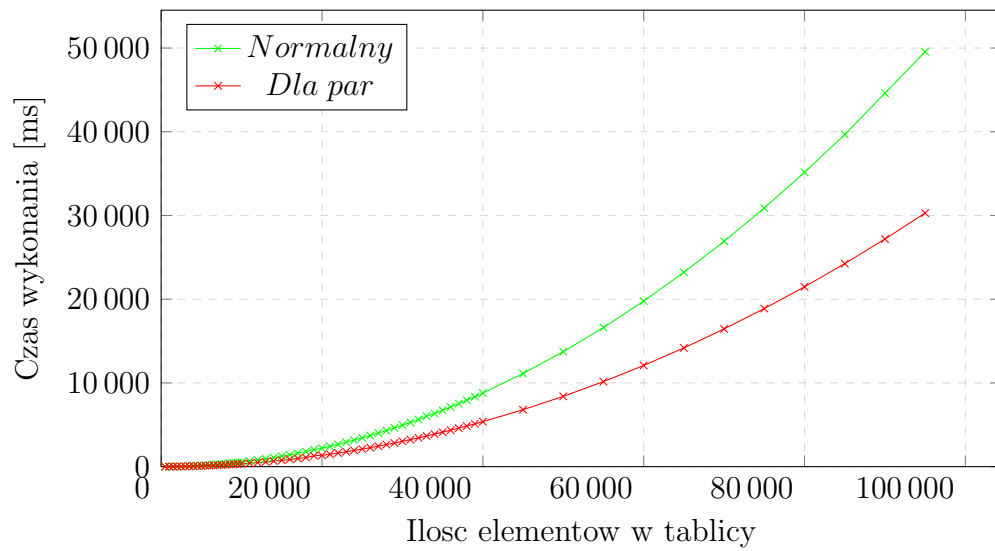
Podstawowa wersja algorytmu zakłada wybieranie najmniejszej lub największej wartości z zbioru. Ten pomysł wybiera największą oraz najmniejszą wartość.

Kod

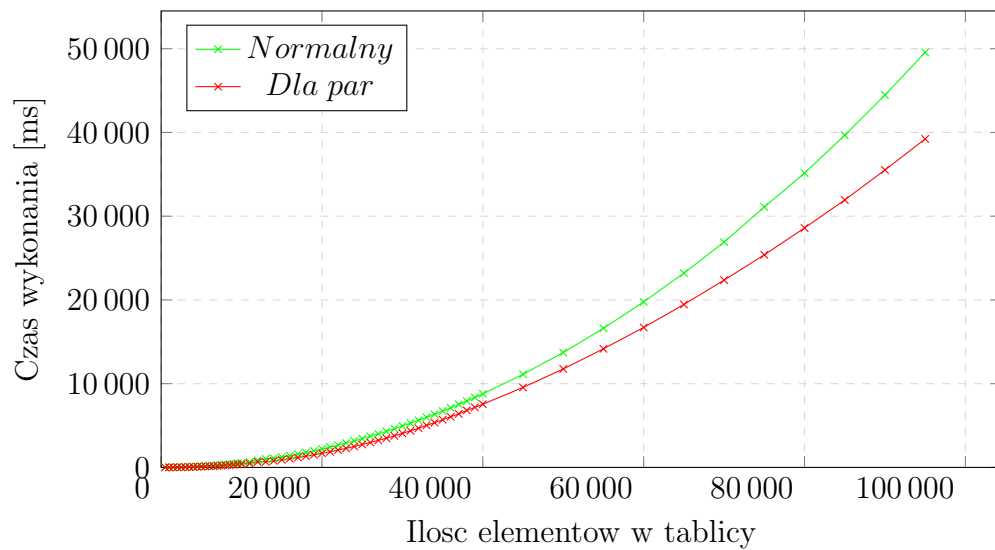
```
1 void sort(std::vector<int> &toSort)
2 {
3     int vectorSize=0;
4     if(toSort.size()%2!=0)
5     {
6         vectorSize++;
7         std::iter_swap((std::min_element(toSort.begin(),toSort.end
8             ↪  ())),toSort.begin());
9     }
10    std::vector<int>::iterator _begin = toSort.begin()+
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    }
```

33 }
34 }

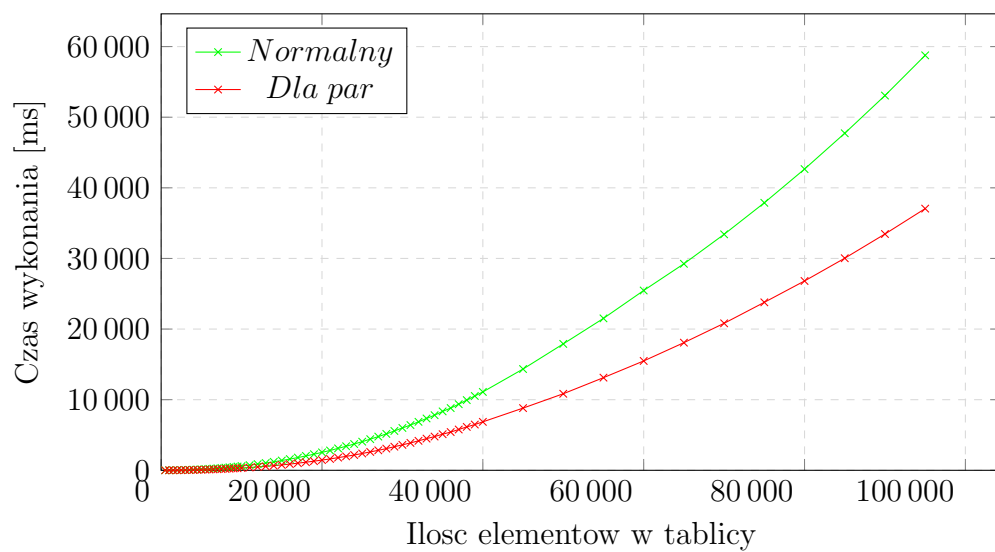
Wyniki



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