Politechnika Łódzka

Szybkie algorytmy

Techniki zwiększenie efektywności algorytmów

Prowadzący zajęcia:

prof. dr hab. Mykhaylo Yatsymirskyy

Autor:

Filip Rynkiewicz

2016-11-15

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.

2 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.
Pierwszym krokiem tego sortowania będzie wybranie pary (x', y'). Pary wybierane są poprzez przesuwanie od indeksu 2 zbioru V, ponieważ zakładamy ze 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 się przesuniecie całej pary przed liczbę z. Za każdym razem liczba z jest liczba poprzedzająca liczbę x'. Jeżeli $z < y'$ algorytm przechodzi do porównania $z > x'$. Jeżeli zostanie spełniony ten warunek liczba mniejsza z pary zostaje przestawiona przed liczba z.

Oszacowanie zlozonosci

Optymistyczna

Pesymistyczna

Srednia

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

Kod

,

```
void sort(std::vector<int> &toSort)
{
 const int sizeOfArray=toSort.size()-(toSort.size()%2);
 for(int i=0; i<sizeOfArray; i+=2)
     {
      if(toSort[i] > toSort[i+1])
         {
          std::swap(toSort[i],toSort[i+1]);
         }
     }
 for(int i=2; i<sizeOfArray; i+=2)
    {
      const int pom1 = toSort[i];
      const int pom2 = toSort[i+1];
      int j = i-1;
          while(j>=0 && toSort[j]>pom2)
           {
            toSort[j+2] =  toSort[j];
            j--;
           }
          toSort[j+2] = pom2;
          toSort[j+1] = pom1;
          while(j>=0 && toSort[j]>pom1)
           {
            toSort[j+1] = toSort[j];
            --j;
           }
          toSort[j+1] = pom1;
        }
        if(toSort.size()%2==1)
           {
            const int pom = toSort[toSort.size()-1];
            int k = toSort.size()-2;
            while(k>=0 && toSort[k]>pom)
             {
              toSort[k+1] = toSort[k];
              --k;
                 }
           toSort[k+1] = pom;
            }
}
```

Wyniki

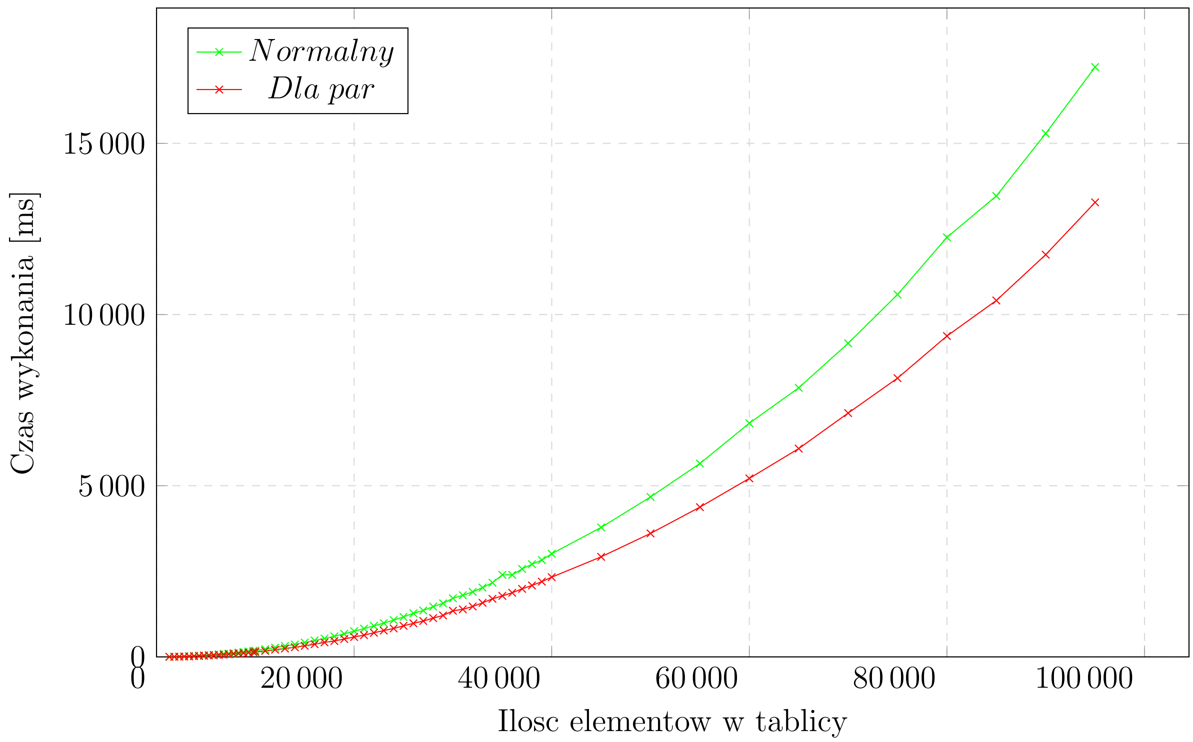

Figure 1: Wykres dla losowych elementów tablicy

3 Algorytm sortowania bąbelkowego

3.1 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 ze mamy porównywać dwie liczby, zostało przyjęte ze 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 kazdej liczby z poprzedzajacej x' zamien ze soba te elementy jezeli

$$x' < z$$

- Dla kazdej liczby w nastepujacej y' zamien ze soba te elementy jezeli

$$y' > w$$

Kod

Algorytm 1: Sortowanie babelkowe dla par

```
void sort(std::vector<int> &toSort)
{
 for(int i= 0; i<(toSort.size()-1); i++)//(1)
 {
  int minElem=i,maxElem=i+1;//(2)
  if(toSort[minElem]>toSort[maxElem])//(3)
   {
    std::swap(toSort[minElem],toSort[maxElem]);//(4)
   }
  while(minElem>0 && toSort[minElem]<toSort[minElem-1])//(5)
   {
    std::swap(toSort[minElem],toSort[minElem-1]);//(6)
    minElem--;//(7)
   }
  while(maxElem<(toSort.size()-1) && toSort[maxElem]>toSort[
     maxElem+1])//(8)
   {
    std::swap(toSort[maxElem],toSort[maxElem+1]);//(9)
    maxElem++;//(10)
   }
 }
}
```

Oszacowanie złożoności

Każdy element który wpływa na złożoność algorytmu został zaznaczony za pomocą numeru, zaznaczonego na zielono, na Algorytmie 1.

Optymistyczna

Jako optymistyczny, czyli najlepszy przypadek, zostalo przyjete wtedy kiedy tablica ktora algorytm ma posortowac jest juz posortowana w dobry kierunku.

(1)

$$\sum_{i=0}^{n-1} i \tag{1}$$

Pesymistyczna

Najgorszy przypadek dla tego sortowania została przyjęta sytuacji kiedy tablice która mamy posortować jest posortowana w odwrotna stronę, niż ten algorytm ma posortować.

(1) Petla for wykona sie zawsze **n** razy, wiec zlozonosc wynosi **n**

(2) Inicjalizacja dwoch zmiennych oraz jedno dodanie zajmuje zawsze 3 akcje, zlozonosc wynosi 3

(3) Sprawdzenie dwoch zmiennych oraz dwa operatory indeksowe zajma 3 akcje, zlozonosc wynosi 3

(4) Zamiana dwoch elementow zajmie 3 akcje, a dwa operatory indeksowania zajma dwie akcje, zlozonosc wynosi 5

(5) petla

Srednia

Wyniki

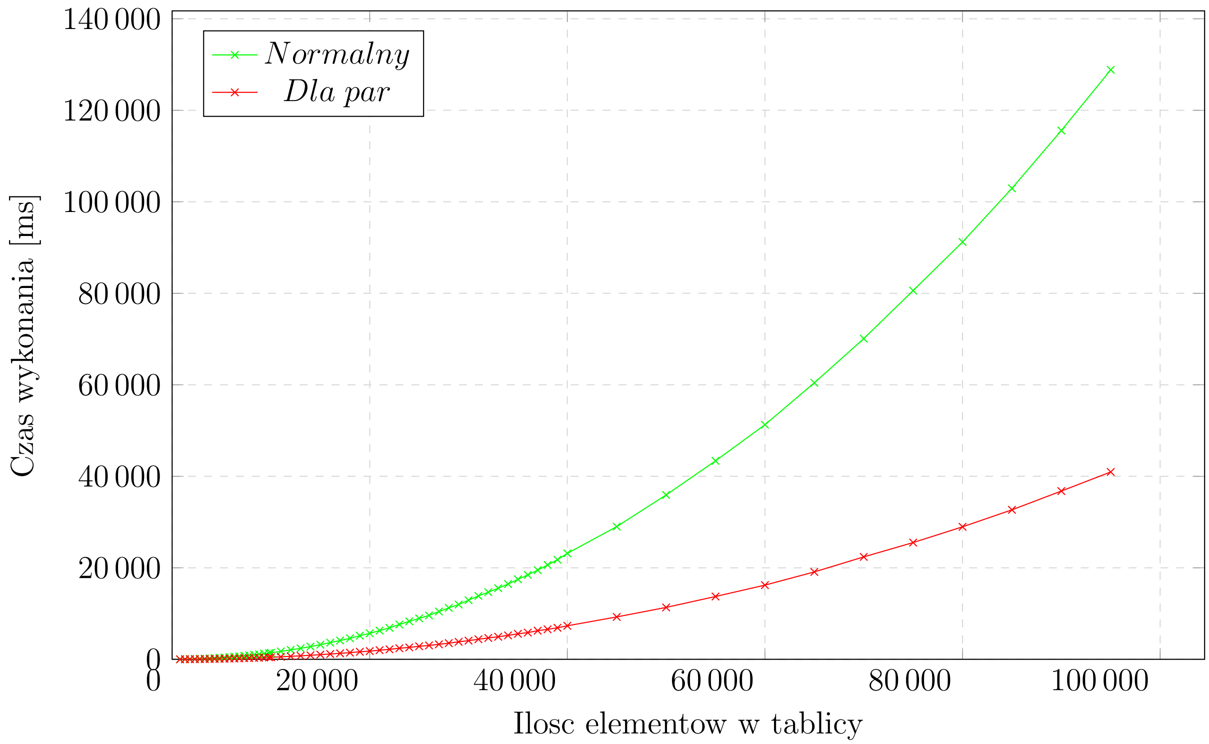

Figure 2: Wykres dla losowych elementow tablicy

4 Algorytm sortowania przez wybieranie

4.1 Dla dwóch elementów

Kod

```
void sort(std::vector<int> &toSort)
{
 int vectorSize=0;
 if(toSort.size()%2!=0)
  {
   vectorSize++;
   std::iter_swap((std::min_element(toSort.begin(),toSort.end
       ())),toSort.begin());
  }
 std::vector<int>::iterator _begin = toSort.begin()+
```

```
    vectorSize;
 std::vector<int>::iterator _end = toSort.end() - 1;
 while (_begin < _end)
  {
   std::vector<int>::iterator it=_begin,_min=it,_max=it;
   for (it = _begin; it <= _end; ++it)
    {
     if ((*it) < (*_min))
      {
           _min = it;
          }
         else if ((*it) > (*_max))
          {
           _max = it;
          }
    }
 std::iter_swap(_min,_begin);
 if(_begin==_max)
  {
   _max=_min;
  }
 std::iter_swap(_max,_end);
 ++_begin;
 --_end;
  }
}
```

Wyniki

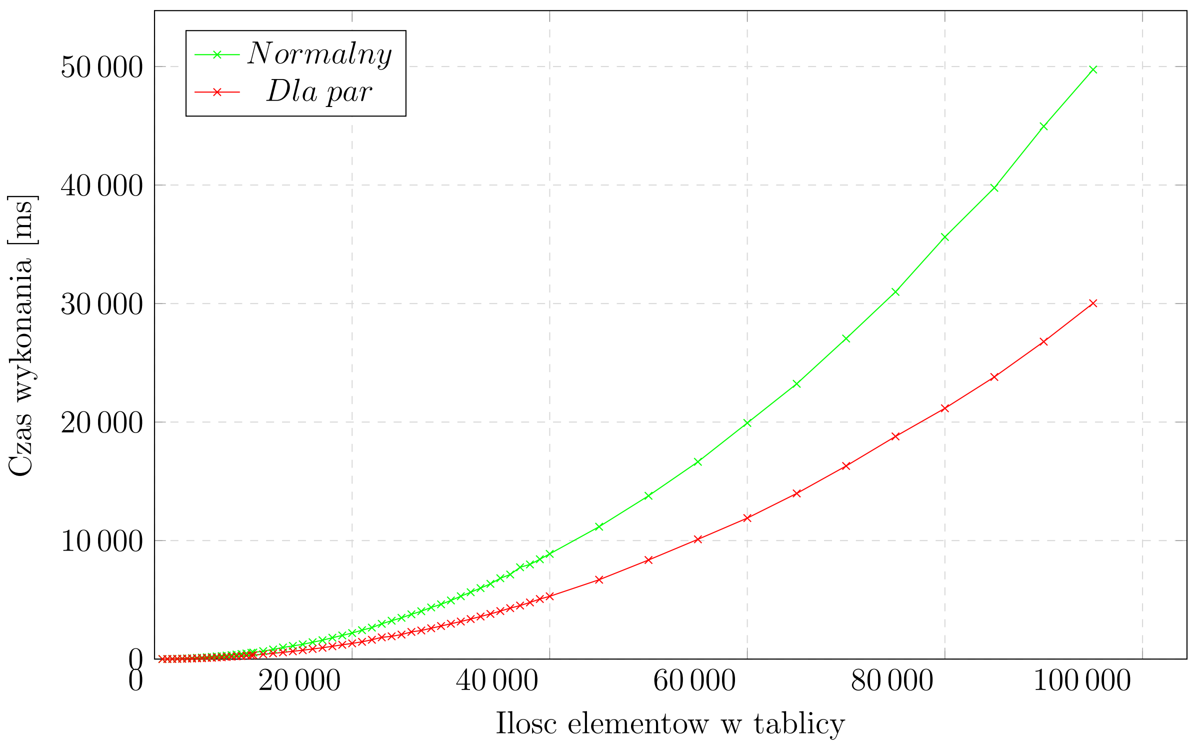

Figure 3: Wykres dla losowych elementów tablicy