AiSD — zadanie pierwsze

Gliwiński Jarosław Marek
Kruczyński Konrad Marek
Grupa dziekańska I5

27 marca 2008

1 Wstęp

Naszym zadaniem była implementacja i porównanie efektywności pięciu algorytmów sortowania:

1. Selection sort (sortowanie przez wybór)
2. Insertion sort (sortowanie przez wstawianie)
3. Heapsort (sortowanie przez kopcowanie)
4. Shell sort (sortowanie Shella)
5. Quicksort (sortowanie szybkie)

Sortowanymi obiektami były ciągi liczb naturalnych. Czasy działania każdego (za wyjątkiem sortowania szybkiego) z algorytmów zostały przetestowane dla pięciu przypadków ciągów liczb:

1. rosnący
2. malejący
3. stały
4. A-kształtny
5. losowy

W drugiej części zadania porównywane były różne algorytmy dla tych samych danych. Użyte zostały te same wyniki, ale w inny sposób przedstawiono je na wykresach (tj. na danym wykresie znajdowały się wyniki dla różnych algortymów i wybranych danych wejściowych).

Proces pomiaru dla sortowania szybkiego przebiegał inaczej. Testowane były trzy wersje tego algorytmu, różniące się sposobem wyboru elementu rozdzielającego. Elementem tym był odpowiednio:

1. prawy skrajny
2. środkowy
3. środkowy względem wartości z trzech losowych (mediana)

Warto wspomnieć, że przy trzeciej metodzie element był określany za pomocą sortowania bąbelkowego, dzięki czemu dokonywano jednocześnie częściowego posortowania tablicy.

Dla każdej z tych trzech metod przygotowano cztery rodzaje ciągów danych:

1. malejący
2. stały
3. A-kształtny
4. losowy

Komentarza wymaga również sortowanie Shella. Z kilku możliwych wartości przyrostów zdecydowaliśmy się na algorytm Knutha $(3h_{i+1} + 1)$, ponieważ jest to najlepsza spośród metod charakteryzujących się trywialnymi wzorami. Porównywalna byłaby tzw. metoda Hibbarda $(2_{i+1} - 1)$ również działająca w czasie $O(n^{3/2})$. By uzyskać niższe wartości wykładnika należałoby posługiwać sie już bardziej złożonymi zależnościami.

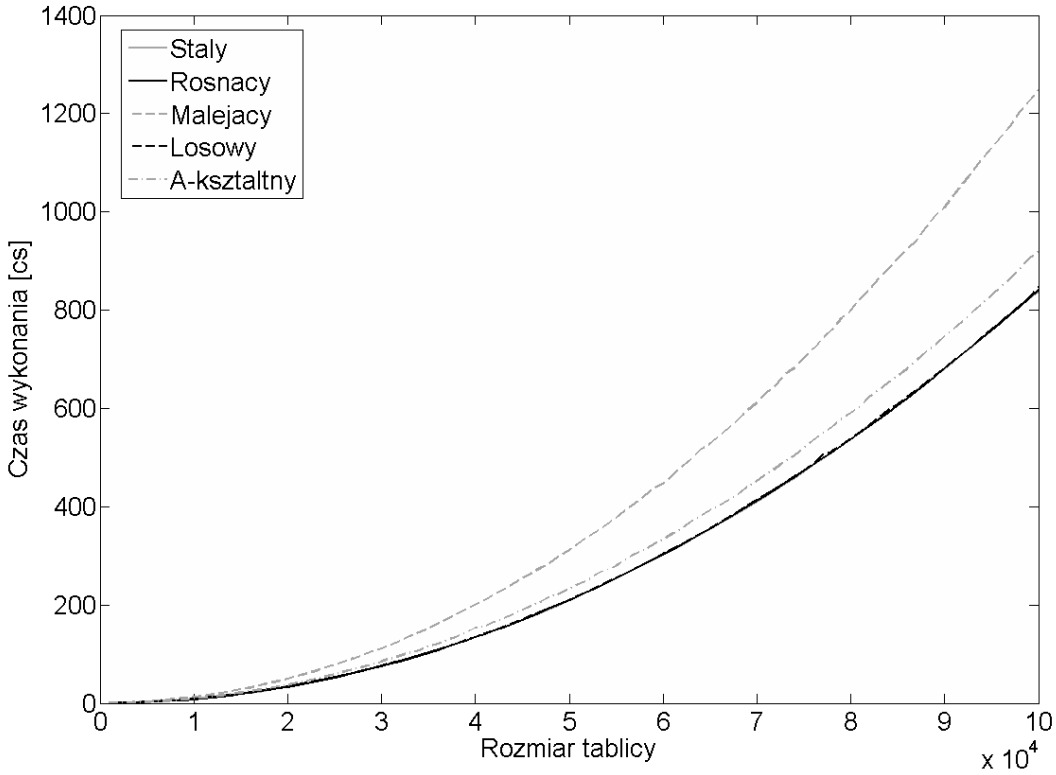
Na koniec warto wspomnieć o sposobie pomiaru czasu. Systemem operacyjnym, w jakim testowaliśmy algorytmy, był Windows XP. Zgodnie z informacjami, jakie udało się nam uzyskać, najdokładniejszą metodą pomiaru czasu jest użycie funkcji pochodzących z API Windows: *QueryPerformanceFrequency* oraz *QueryPerformanceCounter*. Aby zminimalizować fluktuacje spowodowane innymi wątkami, jakie mógł wykonywać w tym czasie procesor, przydzieliliśmy naszemu programowi priorytet czasu rzeczywistego.

2 Pomiary dla czetrech pierwszych algorytmów – porównanie efektywności dla różnych danych

2.1 Procedura testowa

W przypadku danych wyznaczanych w sposób (przynajmniej częściowo) losowy każdy pomiar powtórzono dziesięciokrotnie, natomiast dla pozostałych danych – trzykrotnie. Dla każdej takie próby zmierzono względne odchylenie standardowe czasu sortowania. Wybraliśmy względne, ponieważ siłą rzeczy wartości bezwzględne były znacząco mniejsze dla mniejszej ilości danych. Następnie odchylenie jest uśredniane oraz wyznaczane jest ochylenie średniej odchylenia. Pomiary wykonywane były w zakresie od 1000 do 100000 wartości wejściowych z krokiem 1000.

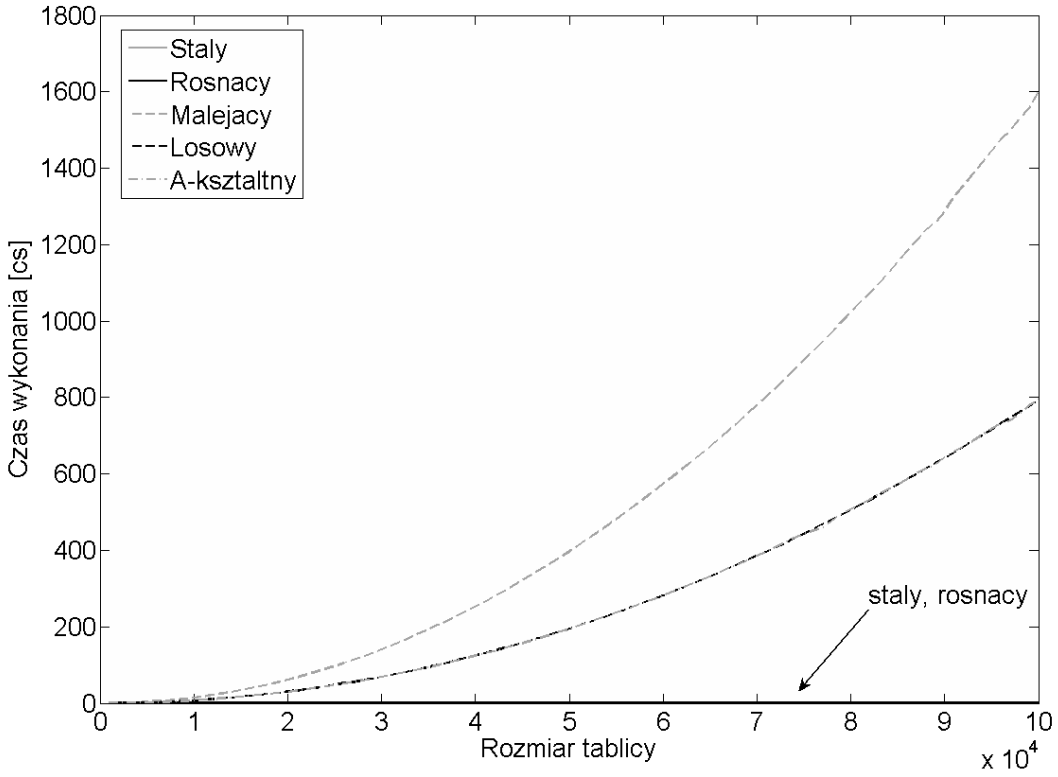
2.2 Selection sort



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$0,033 \pm 0,005$
Rosnące	$0,18 \pm 0,1$
Malejące	$0,62 \pm 0,06$
Losowe	$0,14 \pm 0,024$
A-kształtne	$1,68 \pm 0,06$

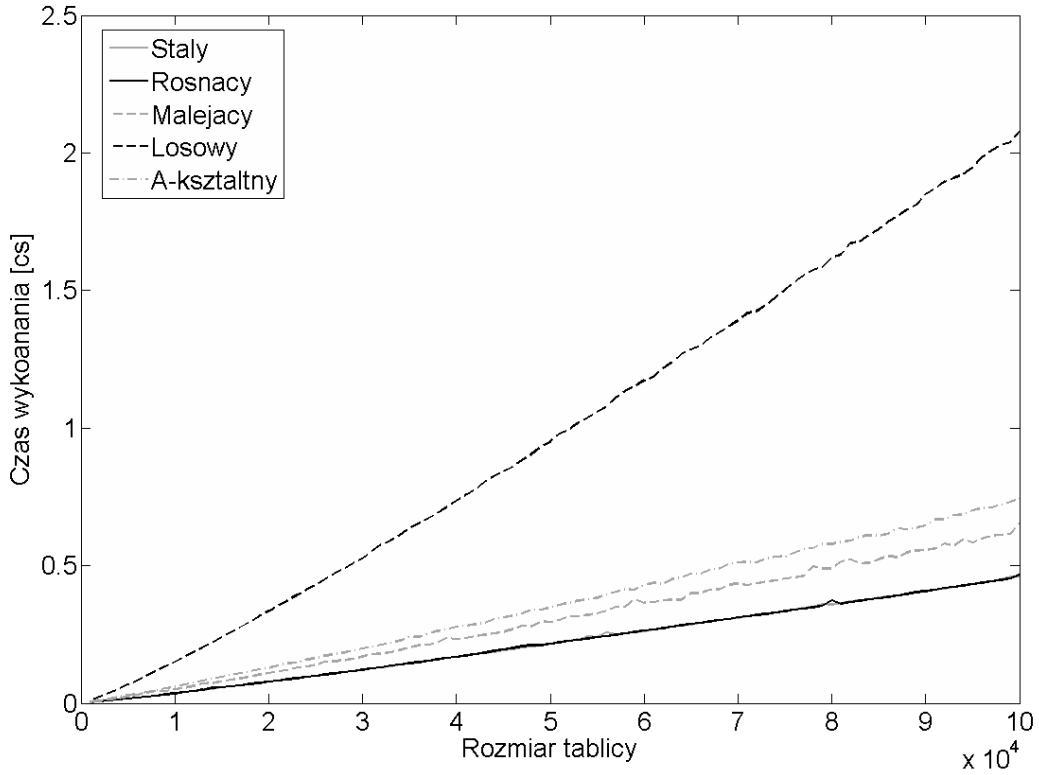
2.3 Insertion sort



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$66,9 \pm 0,4$
Rosnące	$67,9 \pm 0,3$
Malejące	$0,081 \pm 0,013$
Losowe	$0,47 \pm 0,04$
A-kształtne	$0,34 \pm 0,035$

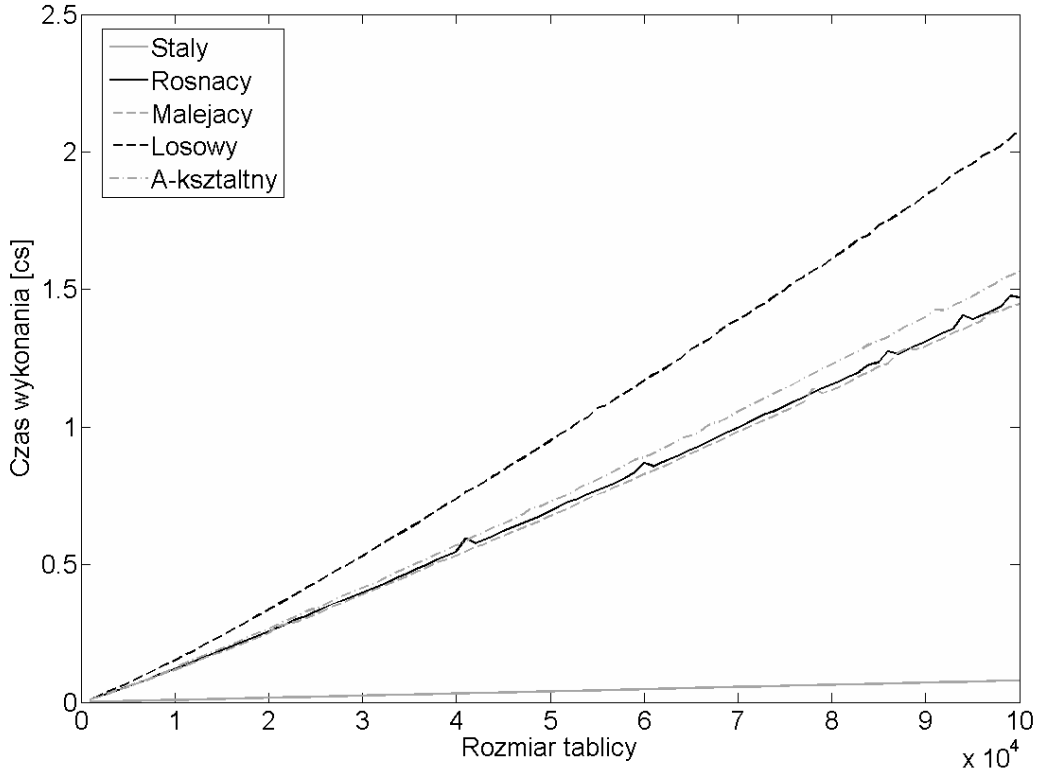
2.4 Shell sort



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$0,31 \pm 0,09$
Rosnące	$0,44 \pm 0,14$
Malejące	$0,70 \pm 0,22$
Losowe	$1,02 \pm 0,09$
A-kształtne	$1,36 \pm 0,09$

2.5 Heapsort



Odchylenia wyników w próbie przedstawiają się:

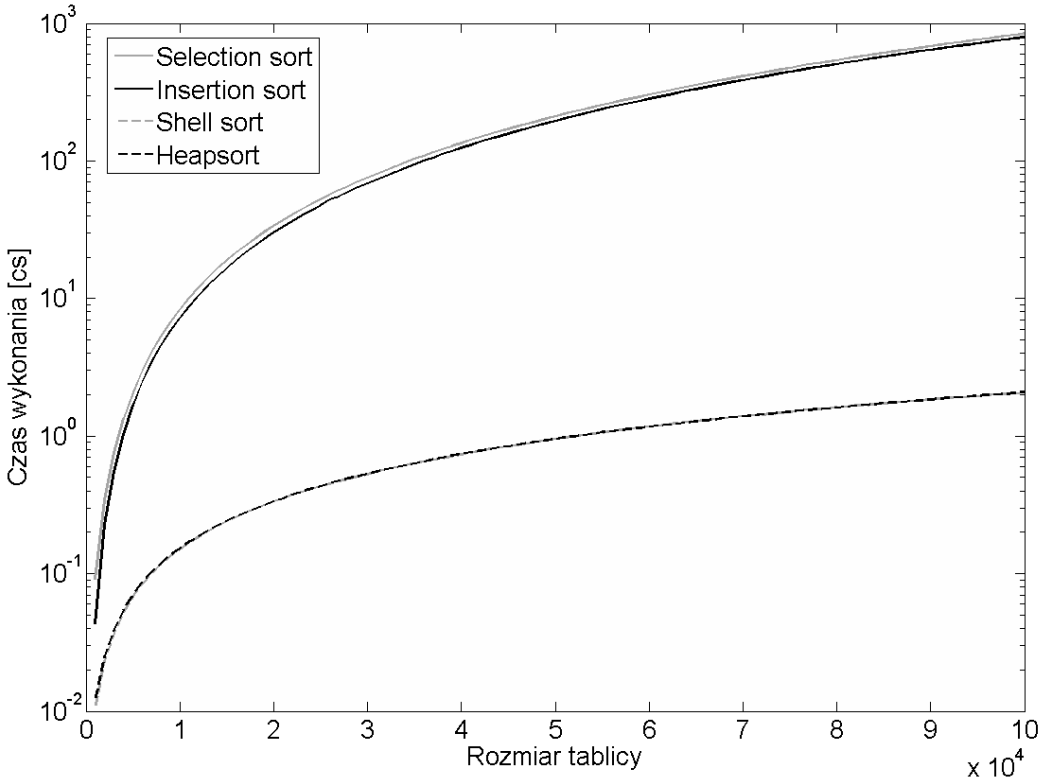
Dane	Odchylenie względne [%]
Stałe	$0,34 \pm 0,08$
Rosnące	$0,38 \pm 0,11$
Malejące	$0,38 \pm 0,09$
Losowe	$0,52 \pm 0,06$
A-kształtne	$0,76 \pm 0,04$

3 Pomiary dla czterech pierwszych algorytmów – porównanie samych algorytmów

3.1 Procedura testowa

Jest ona analogiczna do poprzedniej, acz tym razem zastosowano wykresy ze skalą logarytmiczną.

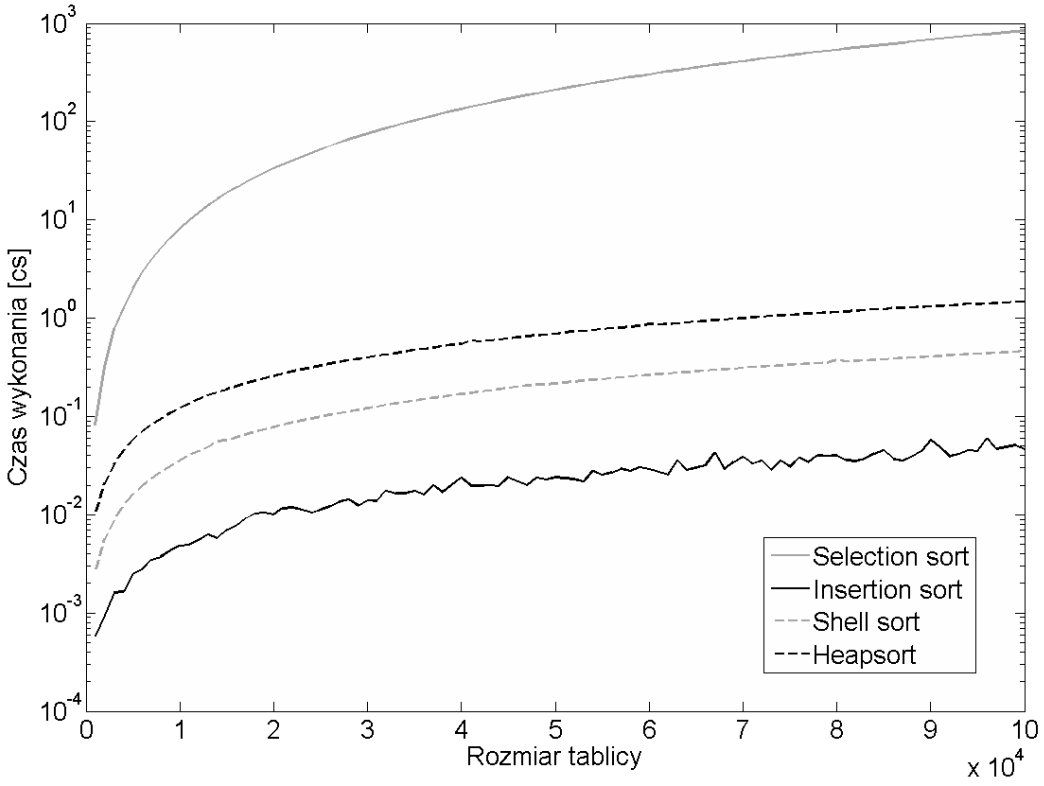
3.2 Dane losowe



Odchylenia wyników w próbie przedstawiają się:

Algorytm	Odchylenie względne [%]
Sortowanie przez wybór	$0,14 \pm 0,04$
Sortowanie przez wstawianie	$0,47 \pm 0,04$
Sortowanie Shella	$1,02 \pm 0,09$
Sortowanie przez kopcowanie	$0,52 \pm 0,06$

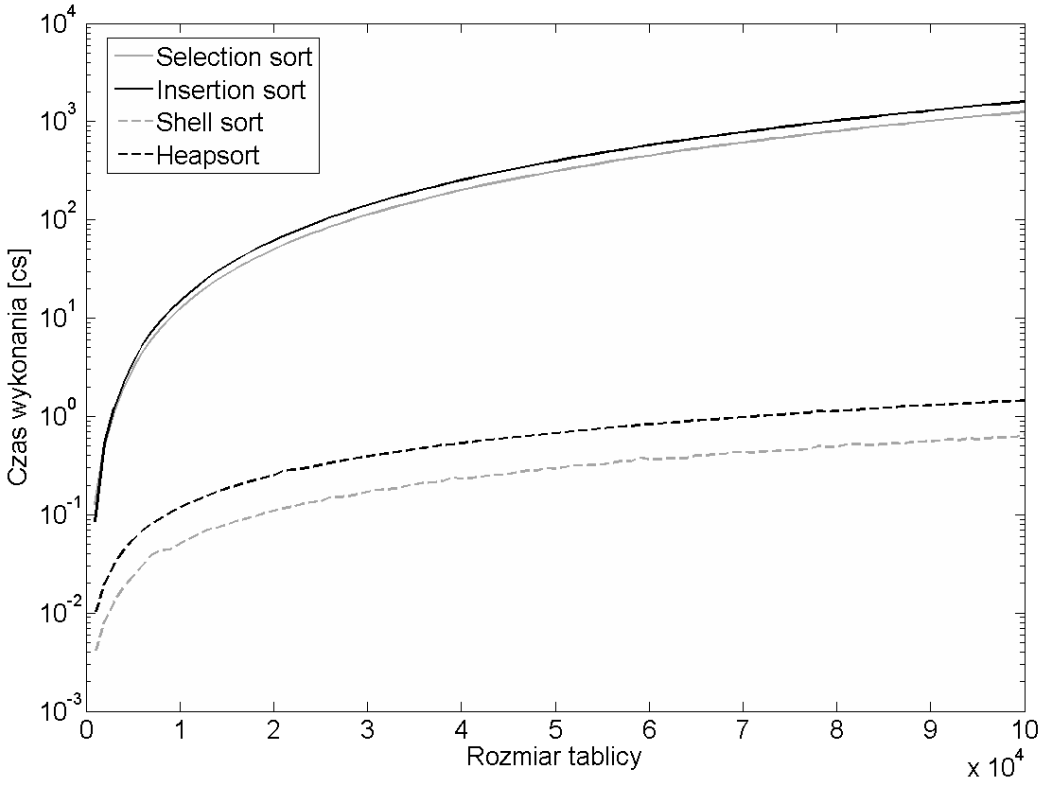
3.3 Dane rosnące



Odchylenia wyników w próbie przedstawiają się:

Algorytm	Odchylenie względne [%]
Sortowanie przez wybór	$0,18 \pm 0,1$
Sortowanie przez wstawianie	$67,9 \pm 0,3$
Sortowanie Shella	$0,44 \pm 0,14$
Sortowanie przez kopcowanie	$0,38 \pm 0,11$

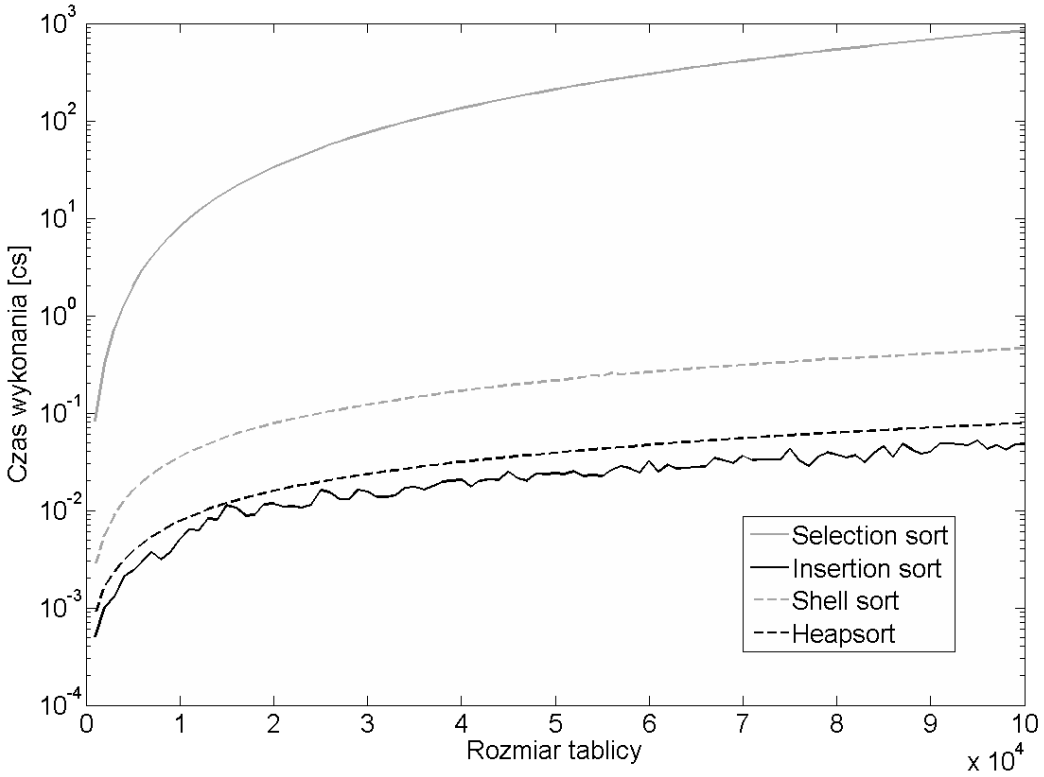
3.4 Dane malejące



Odchylenia wyników w próbie przedstawiają się:

Algorytm	Odchylenie względne [%]
Sortowanie przez wybór	$0,62 \pm 0,06$
Sortowanie przez wstawianie	$0,081 \pm 0,013$
Sortowanie Shella	$0,70 \pm 0,22$
Sortowanie przez kopcowanie	$0,38 \pm 0,09$

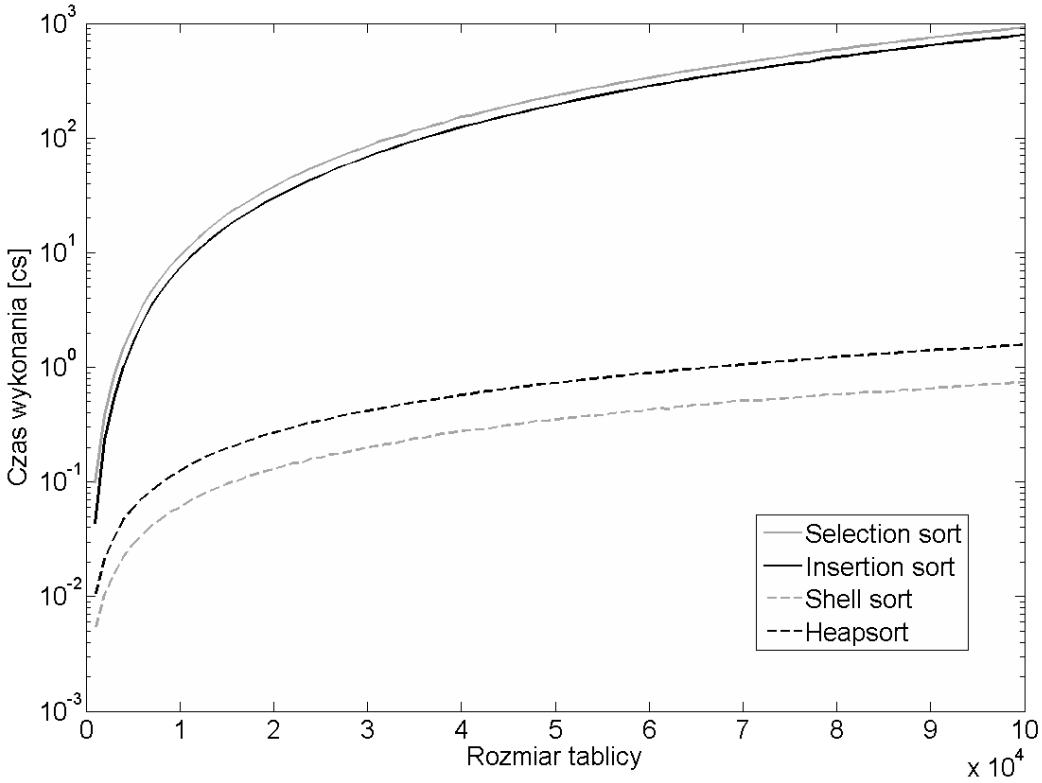
3.5 Dane stałe



Odchylenia wyników w próbie przedstawiają się:

Algorytm	Odchylenie względne [%]
Sortowanie przez wybór	$0,03 \pm 0,005$
Sortowanie przez wstawianie	$66,9 \pm 0,5$
Sortowanie Shella	$0,31 \pm 0,09$
Sortowanie przez kopcowanie	$0,34 \pm 0,08$

3.6 Dane A-kształtne



Odchylenia wyników w próbie przedstawiają się:

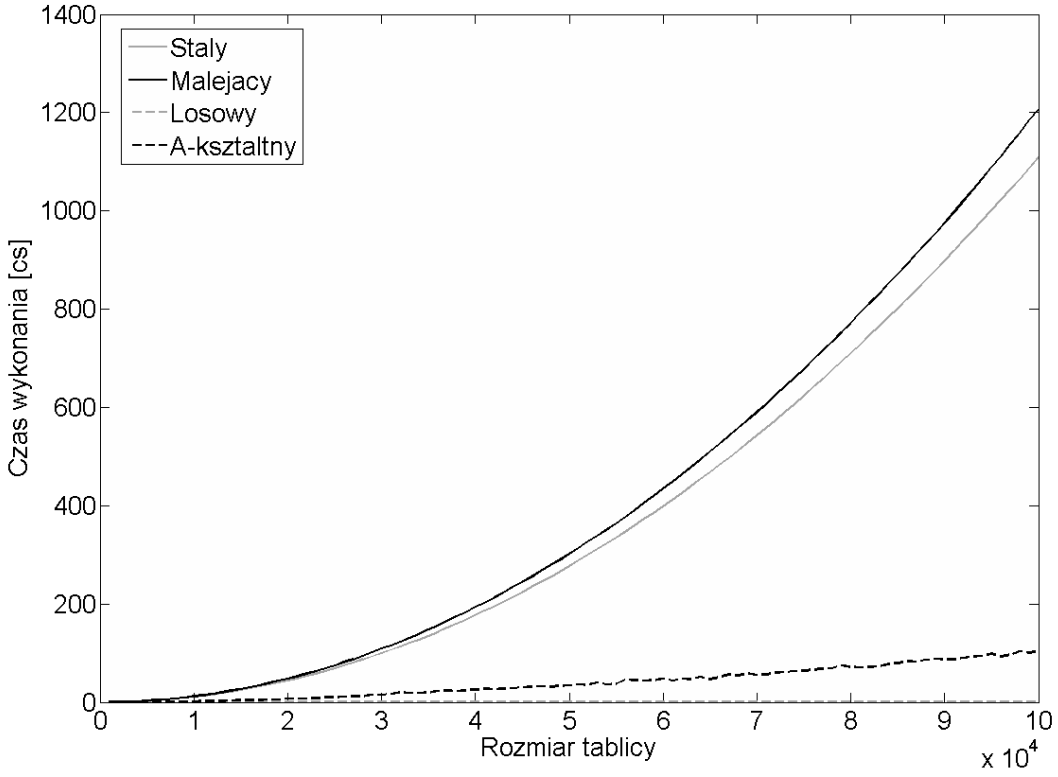
Algorytm	Odchylenie względne [%]
Sortowanie przez wybór	$1,68 \pm 0,06$
Sortowanie przez wstawianie	$0,340 \pm 0,035$
Sortowanie Shella	$1,36 \pm 0,09$
Sortowanie przez kopcowanie	$0,76 \pm 0,04$

4 Sortowanie szybkie

4.1 Procedura testowa

Nieznacznie została zmieniona procedura stosowana uprzednio przy porównywaniu algorytmów – dla sortowania ciągu malejącego testy powtórzono dwudziestokrotnie z uwagi na losowy wybór elementu dzielącego. Przyczynę takiego postępowania podamy w podsumowaniu.

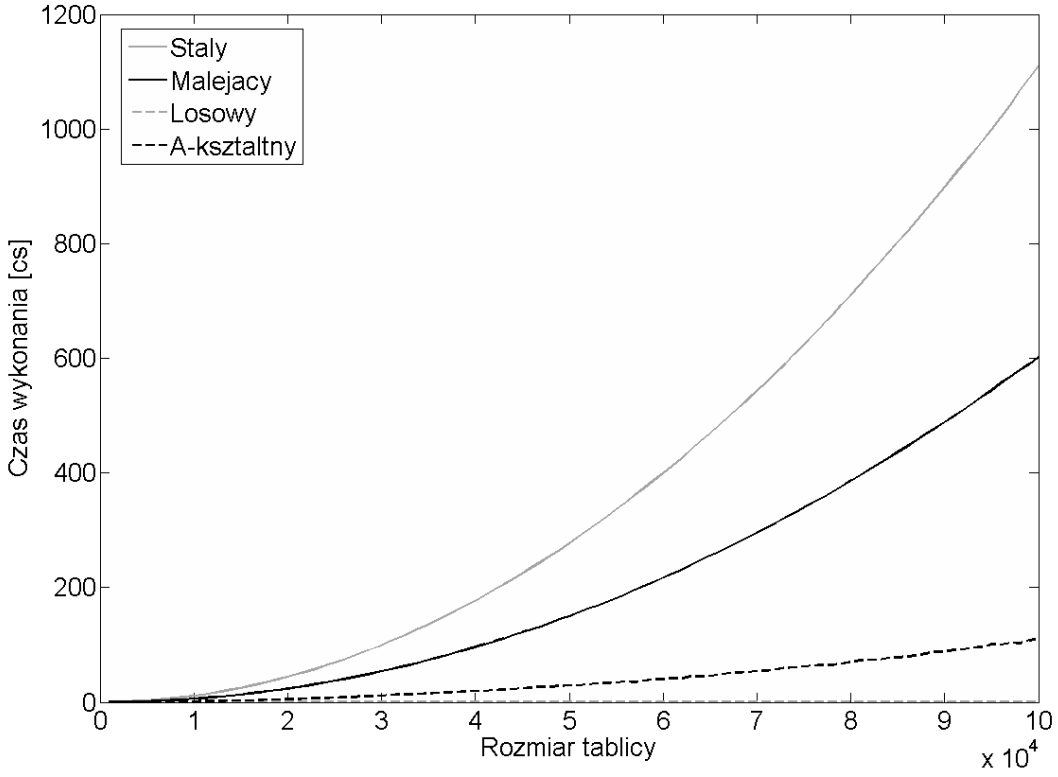
4.2 Element rozdzielający: prawy



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$0,12 \pm 0,01$
Malejące	$0,21 \pm 0,02$
Losowe	$1,02 \pm 0,03$
A-kształtne	$24,9 \pm 0,06$

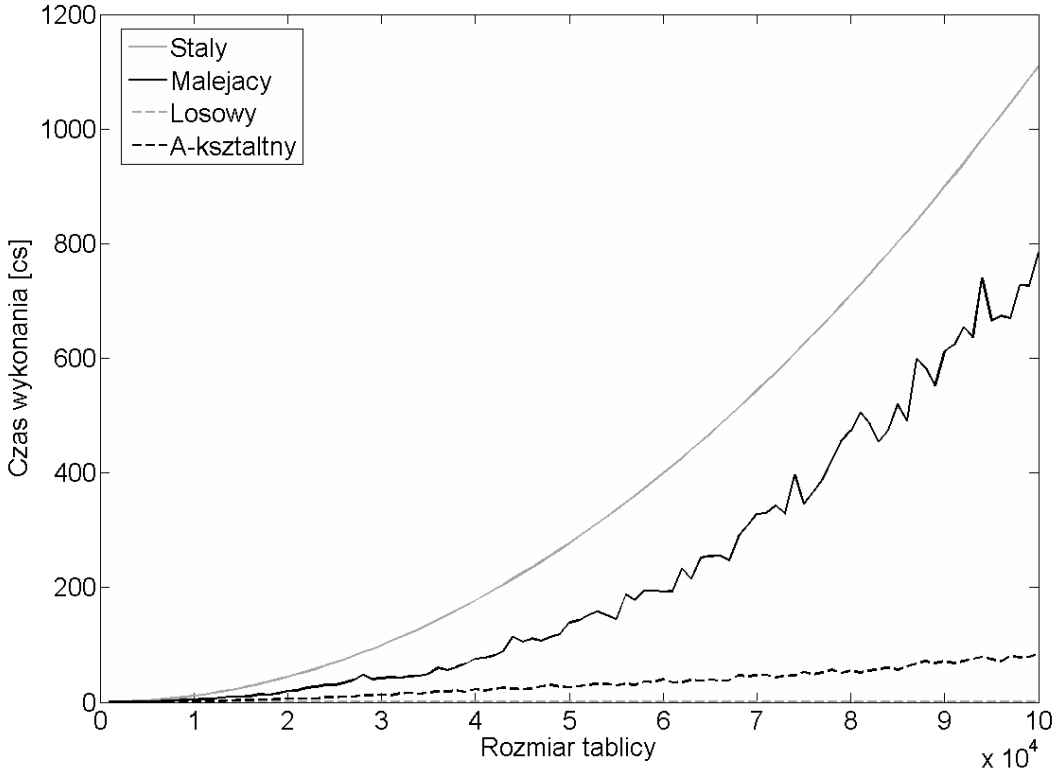
4.3 Element rozdzielający: środkowy



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$0,14 \pm 0,01$
Malejące	$0,22 \pm 0,02$
Losowe	$1,40 \pm 0,07$
A-kształtne	$9,46 \pm 0,26$

4.4 Element rozdzielający: mediana z trzech losowych



Odchylenia wyników w próbie przedstawiają się:

Dane	Odchylenie względne [%]
Stałe	$0,12 \pm 0,008$
Malejące	$29,8 \pm 0,9$
Losowe	$2,03 \pm 0,17$
A-kształtne	$31,1 \pm 0,5$

5 Podsumowanie

Algorytm Selection Sort charakteryzował się złożonością $O(n^2)$, był to algorytm stabilny (niewielkie odchylenia nawet dla losowych danych), rodzaj danych w niewielki sposób wpływał na jego wydajność. Najszybciej program poradził sobie z ciągiem już posortowanym, co wiąże się z tym, iż nie trzeba dokonywać żadnych zamian.

Nieco inaczej zachowywał się Insertion Sort. Ma on tę samą złożoność dla danych otrzymanych w sposób losowy lub malejących. Istotnie inaczej zachowuje się dla danych już posortowanych, tj. rosnących lub stałych. Ponieważ w tym przypadku miejscem dogodnym dla danego elementu jest już to, na którym stoi, całe posortowanie ma w tym przypadku złożoność $O(n)$ i to z tak niewielkimi stałymi, że na wykresie wydają się

być funkcją nieomal stałą. Warto również zwrócić uwagę na znaczne odchylenia w tym najszybszym przypadku. Pewien czas zastanawialiśmy się nad przyczyną takiego zachowania, ponieważ sam algorytm i rodzaj danych są tu zdeterminowane. Okazuje się, że dla tego typu danych algorytm jest tak szybki, że również (na ogół) bardzo niewielkie fluktuacje związane z wykorzystaniem procesora przez kluczowe procesy systemowe mają tu zauważalny udział.

Już pierwszy rzut oka na wykres związany z sortowaniem Shella podpowiada nam, że złożoność tego algorytmu jest dużo niższa niż $O(n^2)$. Zaiste, analiza przeprowadzona przez D. Knutha, o której wspominamy na początku sprawozdania, stwierdza o złożoności na poziomie $O(n^{\frac{3}{2}})$. Minusem sortowania Shella jest spowolnienie dla przypadku ciągu już posortowanego, co wiąże się z wielokrotną analizą tego samego elementu. Algorytm należy do dosyć stabilnych.

Podobnie niedużą złożonościa charakteryzuje się Heapsort – $O(n \log n)$. Niewielkie odchylenia i kształt wykresu potwierdzają jego stabilność.

Czas na analizę wykresów zawierających wszystkie cztery algorytmy. Pierwszy z nich pokazuje ogromną przewagę sortowania Shella i przez kopcowanie nad pozostałymi metodami. Jest to związane z różnicą złożoności, o których wspominamy już wcześniej.

Dla wykresu z danami rosnącymi interesujące są głównie fluktuacje przy sortowaniu przez wstawianie. Przyczyna takowych podana została już wcześniej. Analogiczny komentarz stosuje się do ciągu stałego.

Dane malejące były najtrudniejszym przypadkiem dla algorytmu Insertion sort, który osiągnął swój pesymistyczny czas działania.

Z kolei dla danych A-kształtnych otrzymano wyniki podobne do danych losowych za wyjątkiem przewagi sortowanie Shella nad Heapsort (oraz niewielkiej przewagi sortowania przez wstawianie nad Selection sort).

Pozostała nam analiza algorytmów sortowania szybkiego. Pierwszy, tj. najprostszy z nich wybierał jako element rozdzielający skrajnie prawy. Ta metoda sprawdza się w średnim, czyli losowym przypadku, daje dobre wyniki również dla ciągu A-kształtnego (jest on również w pewnym stopniu losowy). Trudnymi przypadkami dla tego algorytmu są ciągi już posortowane, dla których procedura rekurencyjna jest wywoływana dla ciągu o długości mniejszej o jeden. Ponadto operacja przebiega najwolniej dla ciągu malejącego z uwagi na konieczność wykonania $n - 1$ zamian dla n elementów.

Drugi algorytm jako element rozdzielający wybiera zawsze środkowy. Dzięki temu poprawia się efektywność sortowania ciągu malejącego, ponieważ rekurencyjnie wywoływane jest sortowaniu dwóch ciągów o długości równej w przybliżeniu połowie ciągu pierwotnego.

Elementem rozdzielającym trzeciego alogrytmu jest mediana z trzech losowo wybranych. Ponadto dokonuje on ich posortowania. Poprawia to efektywność sortowania ciągu A-kształtnego, wpływa również na czas sortowania ciągu malejącego. Nie można nie zauważyć znacznych fluktuacji dla tego przypadku. Wiążą się ono z losowym charakterem samego algorytmu. W celu ich wyeliminowania należałoby powtórzyć algorytm znaczną liczbę razy, co z kolei zajęłoby zbyt dużo czasu.

Oczywiście żaden z nowych algorytmów sortowania nie był w stanie poprawnić efek-

tywności dla ciągu stałego, ponieważ niezależnie od sposobu wyboru element rozdzielający ma tę samą wartość.

6 Kod źródłowy

```
#include "stdafx.h"
#include <time.h>
#include <cstdlib>
#include <iostream>
#include <windows.h>
#include <fstream>
#include <string>

using namespace std;

/* Funkcja generuje dane wejściowe do sortowania o rozmiarze size
oraz danych:
0 - stałych
1 - rosnących
2 - malejących
3 - losowych
4 - A-kształtnych */
int *BuildArray(int size, int dataOrganization)
{
    int *built = new int[size];
    switch (dataOrganization)
    {
    case 0:
        {
            int constant = rand()%size;
            for(int i = 0; i < size; i++)
            {
                built[i] = constant;
            }
            return built;
        }
    case 1:
        {
            built[0] = rand()%(size/10);
            for(int i = 1; i < size; i++)
            {
                built[i] = built[i-1] + rand()%10;
            }
```

```
        return built;
    }
case 2:
    {
        built[size-1] = size + rand()%(size);
        for(int i = size - 2; i >= 0; i--)
        {
            built[i] = built[i+1] + rand()%10;
        }
        return built;
    }
case 3:
    {
        for(int i = 0; i < size; i++)
        {
            built[i] = rand()%(size*2);
        }
        return built;
    }
case 4:
    {
        built[1] = rand()%(size/10);
        for(int i = 3; i < size; i+=2)
        {
            built[i] = built[i-2] + rand()%10;
        }
        if (size%2 == 0)
        {
            built[size-2] = size + rand()%(size);
            for(int i = size - 4; i >= 0; i-=2)
            {
                built[i] = built[i+2] + rand()%10;;
            }
        }
        else
        {
            built[size-1] = size + rand()%(size);
            for(int i = size - 3; i >= 0; i-=2)
            {
                built[i] = built[i+2] + rand()%10;;
            }
        }
```

```
            return built;
        }
    }
    return 0;
}

/*Funkcja usuwa z pamięci tablicę*/
void KillArray(int* sorted) {
    delete[] sorted;
}



/* Funkcja drukuje tablicę o rozmiarze size */
void PrintArray(int *printed, int size)
{
    for(int i = 0; i < size; i++)
    {
        cout << printed[i] << " ";
    }
    cout << "\n\n";
}


/* Funkcja zamienia liczby całkowite w 2 miejscach pamięci */
void Swap(int &a, int &b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}

/* Funkcja sortuje przez wstawianie */
void InsertionSort(int *&sorted, int size)
{
    int i,j,buffer;
    for(i = 1; i < size; i++)
    {
        buffer = sorted[i];
        for(j = i - 1; j >= 0 && buffer < sorted[j]; j--)
        {
            sorted[j+1] = sorted[j];
```

```
        }
        sorted[j+1] = buffer;
    }
}
/* Funkcja sortuje met. Shella */
void ShellSort(int *&sorted, int size)
{
    int buffer,j,leap = 1;
    while (leap < size)
    {
        leap = 3 * leap + 1;
    }
    while (leap > 0)
    {
        for (int i = leap; i < size; i++)
        {
            buffer = sorted[i];
        for(j = i - leap; j >= 0 && buffer < sorted[j]; j-=leap)
        {
            sorted[j+leap] = sorted[j];
        }
        sorted[j+leap] = buffer;
        }
        leap /= 3;
    }
}
/* Funkcja sortuje przez wybór */
void SelectionSort(int *&sorted, int size)
{
    int j,min;
    for(int i = 0; i < size; i++)
    {
        min = i;
        for(j = i + 1; j < size; j++)
        {
            if (sorted[j] < sorted[min])
            {
                min = j;
            }
        }
        Swap(sorted[min], sorted[i]);
    }
}
```

```
/* Funkcja sortuje przez kopcowanie */
void HeapSort(int *&sorted, int size)
{
    int child,j;
    //tworzenie kopca
    for(int i = size/2-1; i >= 0; i--)
    {
        j = i;
        while (j <= size/2-1)
        {
            child = j * 2;
            if (child < size && sorted[child+1]>sorted[child])
            {
                child++;
            }
            if (sorted[j] >= sorted[child])
            {
                break;
            }
            Swap(sorted[j], sorted[child]);
            j = child;
        }
    }
    // "wyciąganie" korzenia
    for(int i = size - 1; i >= 1; i--)
    {
        Swap(sorted[0],sorted[i]);


        j = 0;
        while (j <= i/2-1)
        {
            child = j * 2;
            if (child < size && sorted[child+1]>sorted[child])
            {
                child++;
            }
            if (sorted[j] >= sorted[child])
            {
                break;
            }
            Swap(sorted[j], sorted[child]);
```

```
            j = child;
        }
    }
}

/* Wlasciwa funkcja QuickSort, względem elementu skrajnie prawego */
void QS(int *&sorted, int sta, int sto) {
    if(sta<sto) {
        int i=sta;
        for(int k=sta;k<sto;k++) {
            if(sorted[k]<sorted[sto]) {
                Swap(sorted[k], sorted[i++]);
            }
        }
        Swap(sorted[i], sorted[sto]);
        QS(sorted, sta, i-1);
        QS(sorted, i+1, sto);
    }
}

/* Wlasciwa funkcja QuickSort, względem elementu środkowego */
void QS2(int *&sorted, int sta, int sto) {
    if(sta<sto) {
        int i=sta;
        int k;
        int l=sorted[sto/2];
        for(k=sta;k<=sto;k++) {
            if(sorted[k]<l) {
                Swap(sorted[k], sorted[i++]);
            }
        }
        QS(sorted, sta, i-1);
        QS(sorted, i+1, sto);
    }
}

/* Wlasciwa funkcja QuickSort, względem elementu
będącego medianą z trzech losowo wybranych*/
void QS3(int *&sorted, int sta, int sto) {
    if(sta<sto) {
        int i=sta;
        int k,l,m;
        k=rand()%(sto+1);
```

```
        l=rand()%(sto+1);
        //cout << "|" << l <<"|";
        m=rand()%(sto+1);
        if(sorted[k]>sorted[l]) Swap(sorted[k], sorted[l]);
        if(sorted[l]>sorted[m]) Swap(sorted[l], sorted[m]);
        if(sorted[k]>sorted[l]) Swap(sorted[k], sorted[l]);
        l=sorted[l];
        for(k=sta;k<=sto;k++) {
            if(sorted[k]<l) {
                Swap(sorted[k], sorted[i++]);
            }
        }
        QS(sorted, sta, i-1);
        QS(sorted, i+1, sto);
    }
}

/*Funkcja umozliwiająca wskazywanie na QS tym samym wskaźnikiem co na
pozostałe funkcje sortujące*/
void QuickSort(int *&sorted, int size)
{
    QS(sorted, 0, size-1);
}
/*Funkcja umozliwiająca wskazywanie na QS2 tym samym wskaźnikiem co na
pozostałe funkcje sortujące*/
void QuickSort2(int *&sorted, int size)
{
    QS2(sorted, 0, size-1);
}
/*Funkcja umozliwiająca wskazywanie na QS3 tym samym wskaźnikiem co na
pozostałe funkcje sortujące*/
void QuickSort3(int *&sorted, int size)
{
    QS3(sorted, 0, size-1);
}


/*Funkcja mierzy i zwraca czas wykonywania funkcji wzkazywanej
przez wskaźnik wsk*/
long double MeasureRunningTime(void (*wsk)(int*&, int), int *&sorted, int size)
{

    LARGE_INTEGER sta,sto,zeg;
```

```
    LARGE_INTEGER *start = &sta, *zegar = &zeg, *stop = &sto;
    LONGLONG timeStart,timeStop,result,uSecond;
    QueryPerformanceFrequency(zegar);
    uSecond = (*zegar).QuadPart;

    QueryPerformanceCounter(start);
    wsk(sorted, size);
    QueryPerformanceCounter(stop);

    timeStart = (*start).QuadPart;
    timeStop = (*stop).QuadPart;
    result = timeStop - timeStart;
    long double floatResult = (long double)result;
    long double floatUSecond = (long double)uSecond;
    floatUSecond /= 100;
    floatResult /= floatUSecond;
    cout << "to: " << floatResult << "cs\n";
    return floatResult;
}

/*Funkcja wywołuje całą obsługę zdarzeń związanych z pomiarem, w tym: zapis
do pliku, wyświetlanie danych kontrolnych w konsoli, pomiar czasu przy pomocy
funkcji MeasureRunningTime, tworzenie danych testowych i ich usuwanie z
pamięci powykonaniu testu*/

void Call(int przejscia, int min, int max, int step, int metoda,
    void (*wsk)(int*&, int), string nazwa)
{
    ofstream plik;
    cout << "Teraz plik " << nazwa << "\n";
    plik.open(nazwa.c_str());
    for(int n=min; n<= max; n+=step) {
        for(int m=1; m<=przejscia; m++) {
            cout << "*";
            int *sorted = BuildArray(n,metoda);
            //PrintArray(sorted, n);
            plik << MeasureRunningTime(wsk, sorted, n) << " ";
            //_sleep(25);
            //PrintArray(sorted, n);
            KillArray(sorted);
        }
        cout<<"byl to plik " << nazwa << ", rozmiar tablicy " << n << "\n";
    }
```

```
    //_sleep(25);
    plik.close();
    cout<<"\n";

}

void DoLos(int przejscia, int min, int max, int step)
{
    Call(przejscia, min, max, step, 2, &QuickSort3, "wynix_qs32.txt");
}

void DoNLos(int przejscia, int min, int max, int step)
{
    Call(przejscia, min, max, step, 0, &InsertionSort, "wynix_is0.txt");
    Call(przejscia, min, max, step, 1, &InsertionSort, "wynix_is1.txt");
}

int _tmain(int argc, _TCHAR* argv[])
{
    getchar();
    //DoNLos(40, 1000, 100000, 2000);
    DoLos(20, 1000, 100000, 1000);
    return 0;
}
```