Przemysław Pawoniak

Indeks - 402697

Grupa - 25

Implementacja sortowania **Quicksort**, z użyciem funkcji **Partition**, z 5 rodzajami wybierania **pivot**-a:

- Ostatni element
- Losowy element
- Mediana z x wartości, z taką samą odległością pomiędzy każdym elementem
  - 3 elementy
  - 5 elementów
  - 7 elementów
- Oraz 3 sposobami losowania liczb w tablicy:
  - Losowo, ale większe niż **size** (**size +1**-**2*size**)
  - Losowo, w małym zakresie (**0-99**)
  - Nie malejąco, czyli posortowane (**1,2,3…**)
- Oraz 3 sposobami radzenia sobie z równością sortowanego elementu i pivota:
  - Zawsze zamieniaj
  - Czasami zamieniaj
  - Nie zamieniaj
- Czas był mierzony za różnicy 2 zmiennych o wartości **system_clock::now()**, odczytanych przed i po posortowaniem tablicy.

Implementacja:

```
enum PopulateType
{
    RandomHigherThanSize,
    RandomRange,
    NotDecreasing,
};

enum PivotType
{
    LastElement,
    RandomElement,
    Median3,
    Median5,
    Median7,
};

enum PartitionType{
    AlwaysSwap,
    SometimesSwap,
    NeverSwap,
};
```

Za pomocą tego, że zostały zastosowane **enum**-y jesteśmy w stanie wygodnie wybrać działanie programu.

PopulateType pozawala wybrać zakres losowanych liczb (większe niż rozmiar tablicy (**size+1-2*size**), losowo w małym zakresie (**0-99**), czy niemalejąco (w tym przypadku po prostu rosnące (**1,2,3…**),

PivotType – który z elementów tablicy ma być **pivot**-em (**ostatni**, **losowy**, **mediana** z (**3**,**5**,**7**) elementów tablicy o równych odległościach między sobą).

PartitionType – daje wybór, taki że jeżeli liczba którą aktualnie sortujemy jest równa **pivot**-owi, czy ją zamieniamy miejscami **zawsze**, **czasami**, czy **w ogóle**.

QuickSort

```
void Quick_Sort(int* array, int size, int leftIndex, int rightIndex, PivotType type, PartitionType partType)
{
    if (leftIndex >= rightIndex)
    {
        return;
    }

    int pivot = Partition(array, size, leftIndex, rightIndex, type, partType);
    Quick_Sort(array, size, leftIndex, pivot - 1, type, partType);
    Quick_Sort(array, size, pivot + 1, rightIndex, type, partType);
}
```

Główna metoda programu, polega na wybraniu **pivot**-a (na 5 różnych sposobów), a następnie rekurencyjnie uruchamianie tej funkcji dzieląc za każdym razem tablice na 2 części, które osobno

sortujemy.

```
int Partition(int* arr, int size, int left, int right, PivotType type, PartitionType partType)
{
    int pivot;
    int index = rand()%(right - left) + left;

    switch(type)
    {
        case LastElement:
            pivot = arr[right];
            break;
        case RandomElement:
            if(index != right)
            {
                swap(arr, index, right);
            }
            pivot = arr[right];
            break;
        case Median3:
            pivot = PivotMedian3(arr, size, left, right);
            break;
        case Median5:
            pivot = PivotMedian5(arr, size, left, right);
            break;
        case Median7:
            pivot = PivotMedian7(arr, size, left, right);
            break;
        default:
            break;
    }

    int j = left-1;

    for(int i = left; i < right; i++)
    {
        if(arr[i] == pivot)
        {
            switch(partType)
            {
                case AlwaysSwap:
                    swap(arr, i, right);
                    break;
                case SometimesSwap:
                    if(rand()%2==0)
                    {
                        swap(arr, i, right);
                    }
                    break;
                case NeverSwap:
                    break;
                default:
                    break;
            }
        }
        else if(arr[i] < pivot)
        {
            j++;
            swap(arr, j, i);
        }
    }
    swap(arr, j + 1, right);
    return j + 1;
}
```

Partition, funkcja w której wybieramy **pivot** tak jak potrzebujemy za pomocą przekazanego do **QuickSort**

PivotType, rozszerzona o wybór co robimy jeśli losowany element jest równy pivotowi.

```
int PivotMedian3(int* arr, int size, int left, int right)
{
    if(size<3)
    {
        throw invalid_argument("Array size too small");
    }

    int middleIndex = (left + right)/2;

    int tmp[3] =
    {
        arr[left],
        arr[middleIndex],
        arr[right]
    };

    sort(begin(tmp), end(tmp));

    arr[left] = tmp[0];
    arr[middleIndex] = tmp[1];
    arr[right] = tmp[2];

    swap(arr, middleIndex, right);

    return arr[right];
}
```

Funkcja **PivotMedian3** bierze jako argumenty tablice, jej rozmiar, oraz indexy początku i kończą szukania medianę z **3**-elementów. Jako że mediana to środkowy element ciągu uporządkowanego to tworzona jest tablica **tmp** 3 elementowa, następnie jest ona sortowana. Później wpisywane są uporządkowane wartości z powrotem do głównej tablicy, z takim wyjątkiem, że mediana jest zamieniana z ostatnim elementem. Na koniec zwracamy ten ostatni element, który jest szukaną medianą.

```
int PivotMedian5(int* arr, int size, int left, int right)
{
    if(size<5)
    {
        throw invalid_argument("Array size too small");
    }

    int middleIndex = (left + right)/2;

    int tmp[5] =
    {
        arr[left],
        arr[(left + middleIndex)/2],
        arr[middleIndex],
        arr[(middleIndex + right)/2],
        arr[right]
    };

    sort(begin(tmp), end(tmp));

    arr[left] = tmp[0];
    arr[(left + middleIndex)/2] = tmp[1];
    arr[middleIndex] = tmp[2];
    arr[(middleIndex + right)/2] = tmp[3];
    arr[right] = tmp[4];

    swap(arr, middleIndex, right);

    return arr[right];
}
```

PivotMedian5 działa tak samo jak **PivotMedian3**, ale bierze **5** elementów oddalonych od siebie o mniej więcej równą odległość. Reszta funkcji przebiega tak samo jw.

```
int PivotMedian7(int* arr, int size, int left, int right)
{
    if(size<7)
    {
        throw invalid_argument("Array size too small");
    }

    int middleIndex = (left + right)/2;

    int tmp[7] =
    {
        arr[left],
        arr[left + (middleIndex - left)/3],
        arr[left + 2 * (middleIndex - left)/3],
        arr[middleIndex],
        arr[middleIndex + (right - middleIndex)/3],
        arr[middleIndex + 2 *(right - middleIndex)/3],
        arr[right]
    };

    sort(begin(tmp), end(tmp));

    arr[left] = tmp[0];
    arr[left + (middleIndex - left)/3] = tmp[1];
    arr[left + 2 *(middleIndex - left)/3] = tmp[2];
    arr[middleIndex] = tmp[3];
    arr[middleIndex + (right - middleIndex)/3] = tmp[4];
    arr[middleIndex + 2 * (right - middleIndex)/3] = tmp[5];
    arr[right] = tmp[6];

    swap(arr, middleIndex, right);

    return arr[right];
}
```

PivotMedian7 działa tak samo jak **PivotMedian3**, oraz **PivotMedian5**, ale bierze **7** elementów oddalonych od siebie o mniej więcej równą odległość. Reszta funkcji przebiega tak samo jak reszta.

```
void swap(int* array, int left, int right)
{
    int tmp = array[right];
    array[right] = array[left];
    array[left] = tmp;
}
```

```
int* Populate(int size, PopulateType type)
{
    int* array = new int[size];

    switch (type)
    {
        case RandomHigherThanSize:
            for (int i = 0; i < size; i++)
            {
                array[i] = rand() % size + 1 + size;
            }
            break;
        case RandomRange:
            for (int i = 0; i < size; i++)
            {
                array[i] = rand() % 100;
            }
            break;
        case NotDecreasing:
            for (int i = 0; i < size; i++)
            {
                array[i] = i;
            }
            break;
        default:
            break;
    }
    return array;
}
```

Populate służy do zapełnienia tablicy odpowiednio wylosowanymi liczbami, zależnymi od **PopulateType**.

```
void RunForTime(PivotType type, PopulateType popType, PartitionType partType)
{
    int* array;

    time_point<system_clock> start,end;

    duration<double> dur= end - start;
    duration<double> durMini= end - start;

    cout << "TYP UZUPELNIENIA::: " << popType + 1 << endl;
    cout<<"i elementow---czas"<<endl;
    for (int i = 10 * 1000; i <= 100 * 1000 ; i += 10 * 1000)
    {
        for (int j = 0; j < times; j++)
        {
            array = Populate(i, popType);

            WriteOutArray(array,i);

            start = system_clock::now();

            Quick_Sort(array, i, 0, i - 1, type, partType);
            end = system_clock::now();

            WriteOutArray(array,i);

            delete[] array;

            durMini = end-start;
            dur+= end - start;
        }
        cout<<i<<"---"<<duration_cast<milliseconds>(durMini).count()<<endl;
    }

    cout << "CZAS LACZNY::: " << duration_cast<milliseconds>(dur).count()/times << endl;
    cout<<endl;
}
```

RunForTime jest funkcją, która w zależności od **PivotType**, **PopulateType**, oraz **PartitionType** działa inaczej. Dla zakresu od 10 * 1000 do 100 * 1000, rosnących co 10 * 1000 uruchamia 5 generowanie, oraz sortowanie tablicy, wypisuje do konsoli czas sortowania każdego z kroków, oraz czas średni sortowania, tuż przed zwiększeniem zakresu o 10 * 1000.

```
void RunAllPopulateTypes(PivotType type, PartitionType partTime)
{
    cout << "TYP PARTITION::: " << partTime + 1 << endl;
    RunForTime(type, RandomHigherThanSize, partTime);
    RunForTime(type, RandomRange, partTime);
    RunForTime(type, NotDecreasing, partTime);
    cout<<endl;
}
```

RunAllPopulateTypes uruchamia funkcję wyżej dla wszystkich rodzajów losowania elementów tablicy.

```
void Run( PivotType type)
{
    cout << "TYP PIVOTA::: " << type + 1 << endl;
    RunAllPopulateTypes(type, AlwaysSwap);
    RunAllPopulateTypes(type, SometimesSwap);
    RunAllPopulateTypes(type, NeverSwap);
    cout<<endl;
}
```

Run uruchamia funkcję wyżej dla wszystkich rodzajów decydowania odnośnie części dodatkowej funkcji **Partition**.

```
void RunAll()
{
    Run(LastElement);
    Run(RandomElement);
    Run(Median3);
    Run(Median5);
    Run(Median7);
}
```

RunAll uruchamia funkcję wyżej dla wszystkich rodzajów wybierania **pivot**-a.

```
int main()
{
    srand(time(NULL));

    cout<<"TYPY PIVOTA:"<<endl;
    cout<<"1 - ostatni"<<endl;
    cout<<"2 - losowy"<<endl;
    cout<<"3 - mediana3"<<endl;
    cout<<"4 - mediana5"<<endl;
    cout<<"5 - mediana7"<<endl;
    cout<<endl;
    cout<<"TYPY PARTITION:"<<endl;
    cout<<"1 - zawsze zamieniaj"<<endl;
    cout<<"2 - czasami zamieniaj"<<endl;
    cout<<"3 - nie zamienaj"<<endl;
    cout<<endl;
    cout<<"TYPY UZUPELNIENIA TABELI:"<<endl;
    cout<<"1 - losowe wieksze niz rozmiar"<<endl;
    cout<<"2 - losowe zakres 0-99"<<endl;
    cout<<"3 - uporzadkowane"<<endl;
    cout<<endl;

    RunAll();

    return 0;
}
```

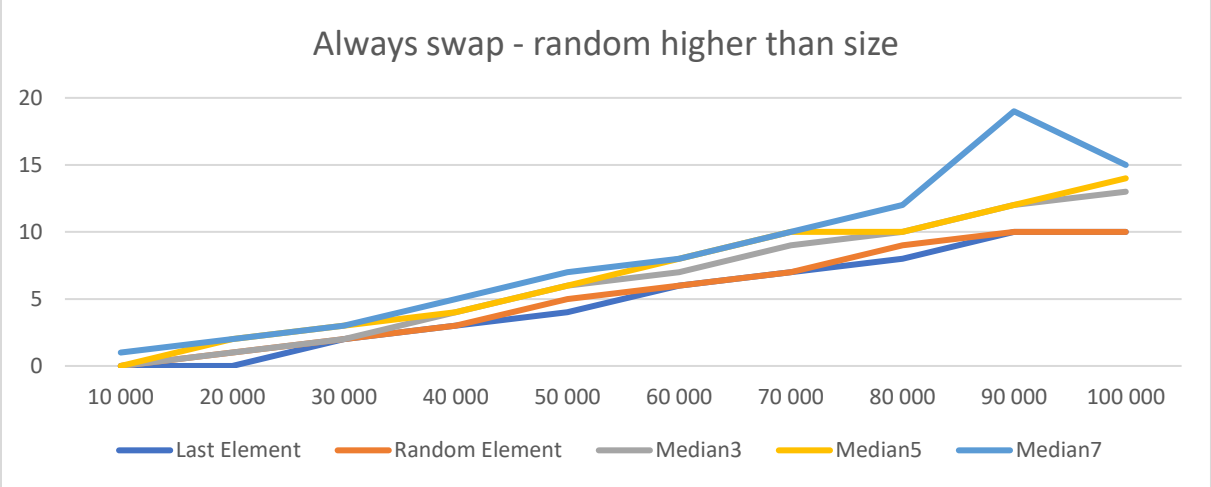
Main jest funkcją, która musi być w projekcie konsolowym, w niej uruchamia się całe działanie programu. Następuje tu użycie ziarna, aby uzyskać lepsze liczby pseudolosowe, z każdym uruchomieniem inne.

WNIOSKI:

Wykresy pokazują zależność między ilością elementów tablicy, oraz czasu, który jest potrzebny do posortowania w odpowiednich ustawieniach. Każdy czas jest średnią z 5 sortowań, w celu zmniejszenia błędu.

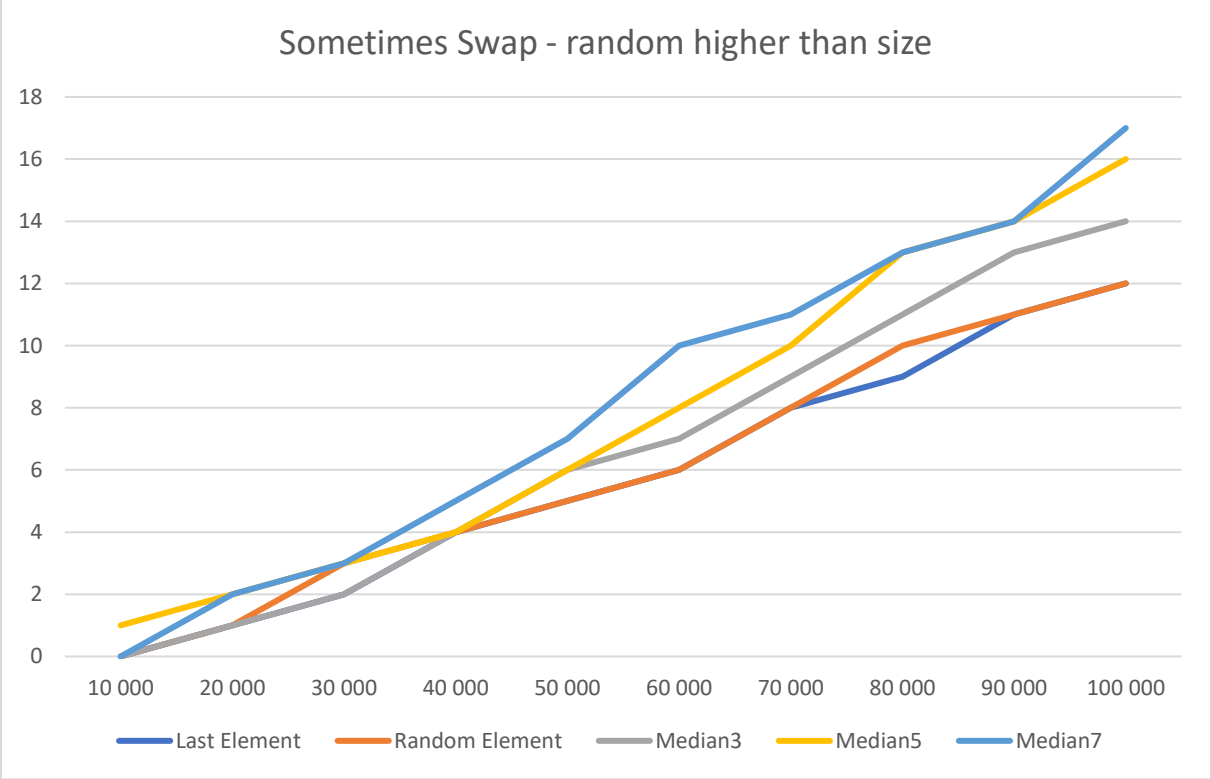
Dla losowania wartości tablicy rodzaju **większy niż rozmiar:**

PartitionType = **AlwaysSwap**



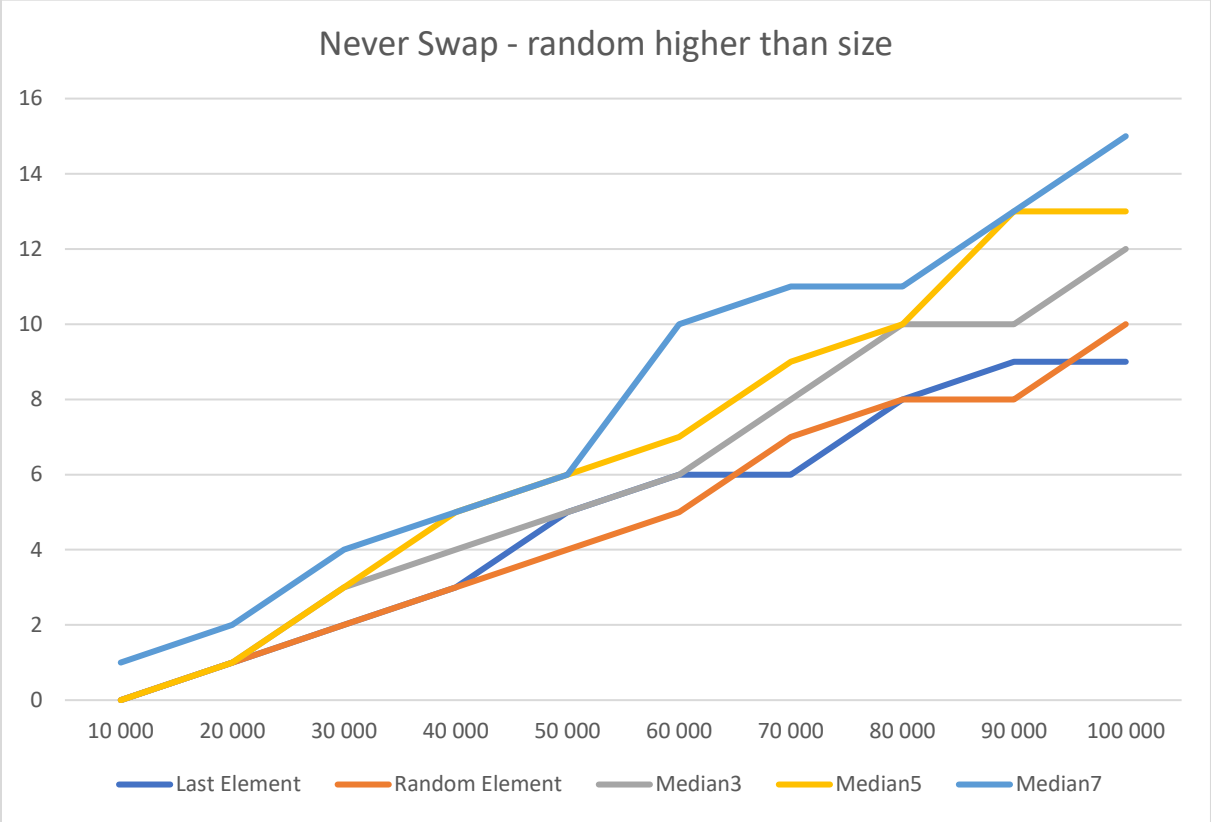
Mediana7 jest najwolniejszym sposobem wybierania **pivot**-a, jednakże czas jest tak mały, że jest to nieważne dla takich rozmiarów tablicy.

PartitionType = **SometimesSwap**



Czasy znowu są bardzo do siebie zbliżone, jednakże można zauważyć drobną zmianę w rośnięciu czasu za pomocą sortowania **Median5**.

PartitionType = **NeverSwap**



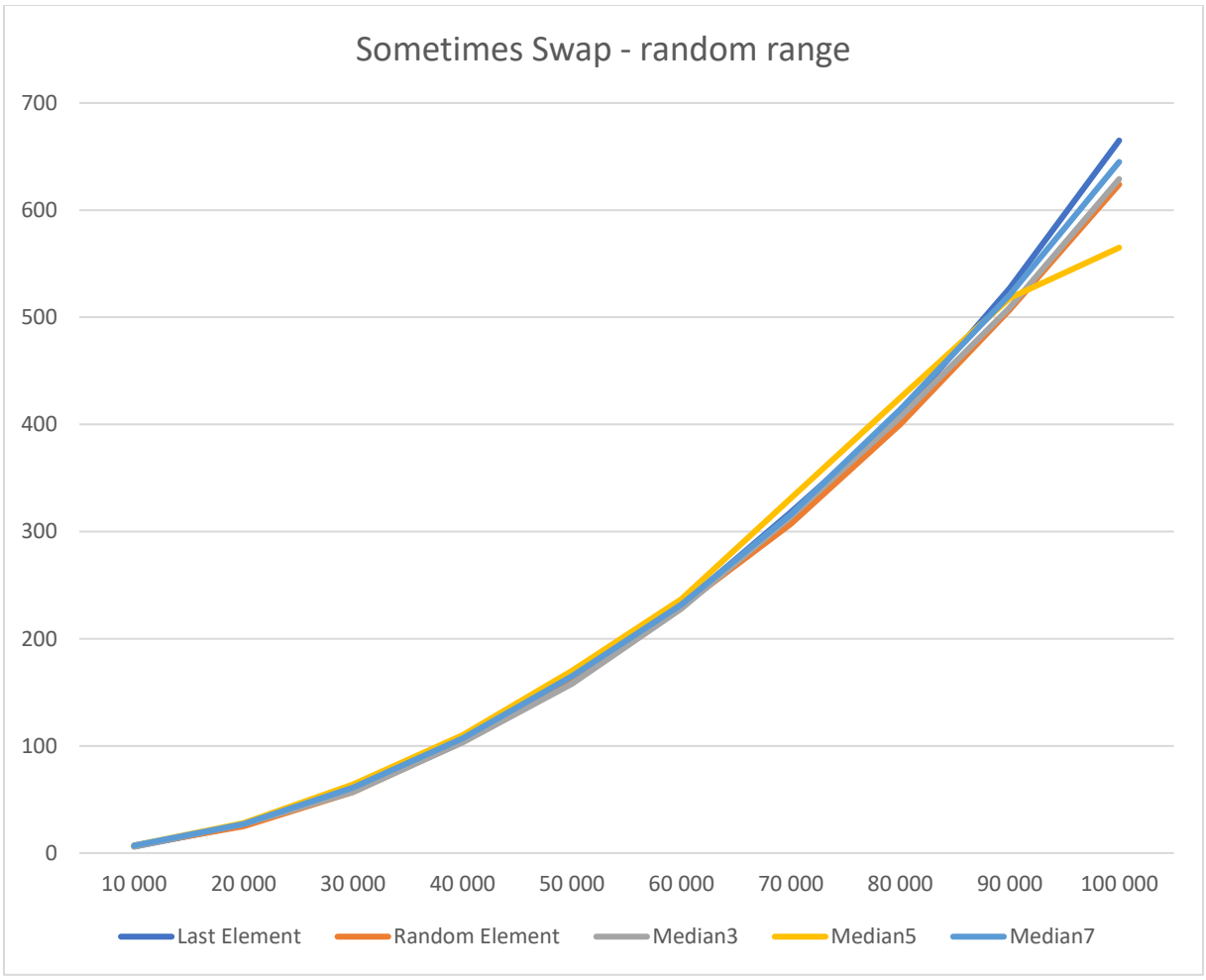
Widać na skali, że czasy są o 20% mniejsze niż w poprzednich przypadkach, ale rozkład pozostaje praktycznie bez zmian.

Dla losowania wartości tablicy rodzaju **losowy z przedziału:**

PartitionType = **AlwaysSwap**

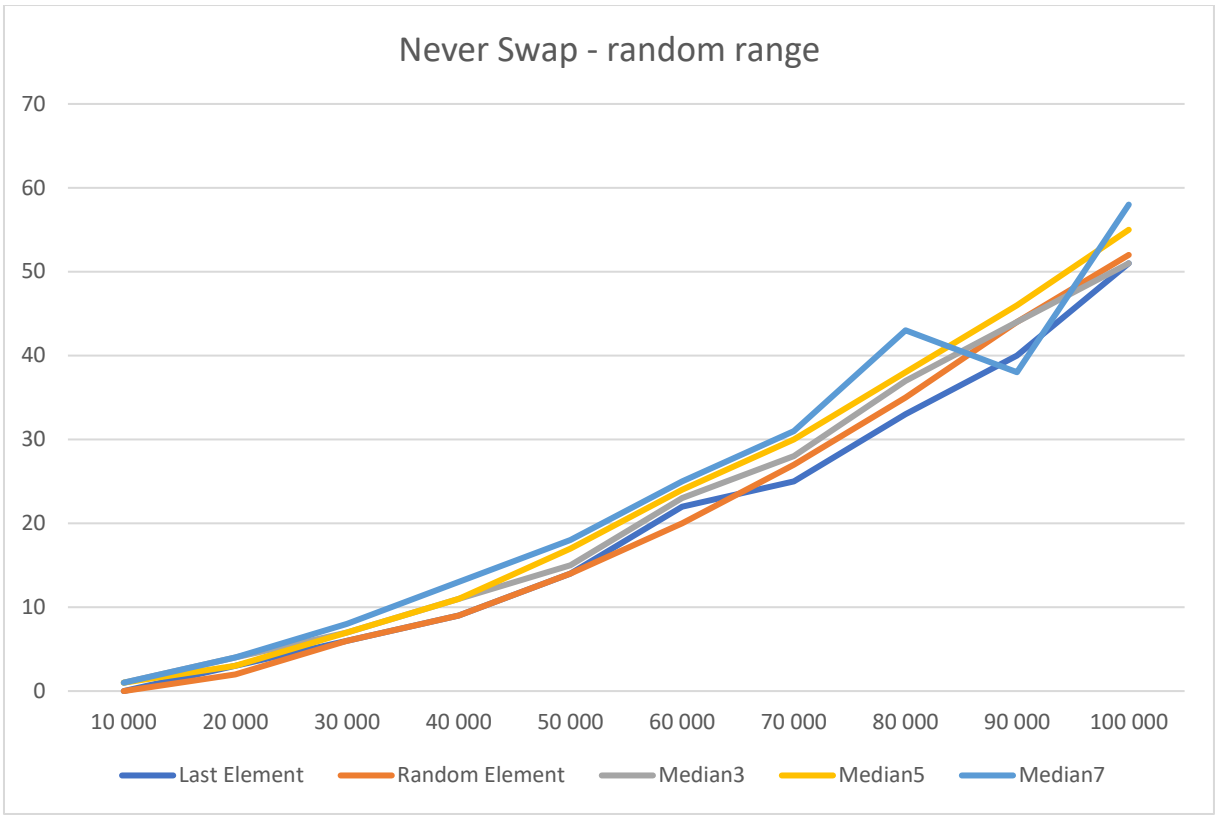
Od razu widać ze skali, że czasy są około 10 razy większe, jednakże złożoność wszystkich funkcji jest jeszcze bliższa sobie nawzajem niż w poprzednich przypadkach.

PartitionType = **SometimesSwap**



Czasy są praktycznie w okolicy sekundy, jednakże nie ma dużego rozrzutu, jak wcześniej.

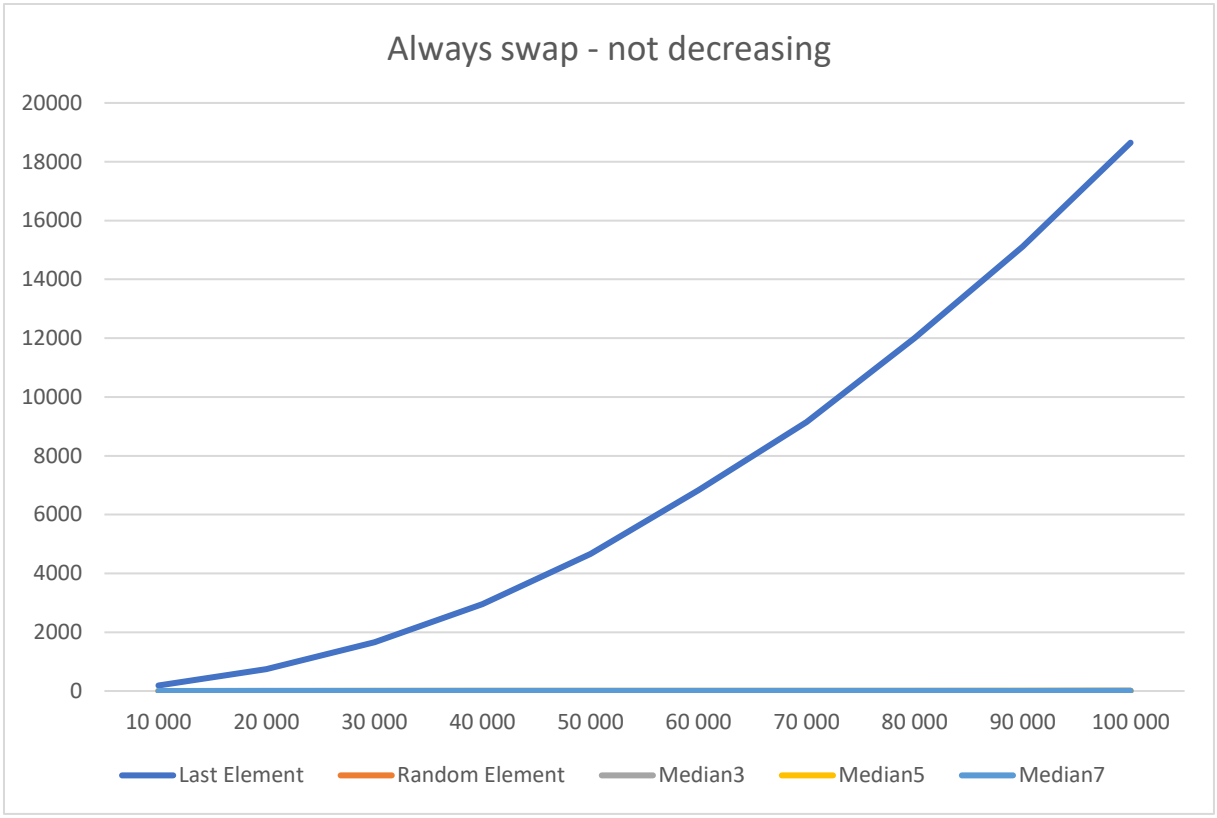
PartitionType = **NeverSwap**

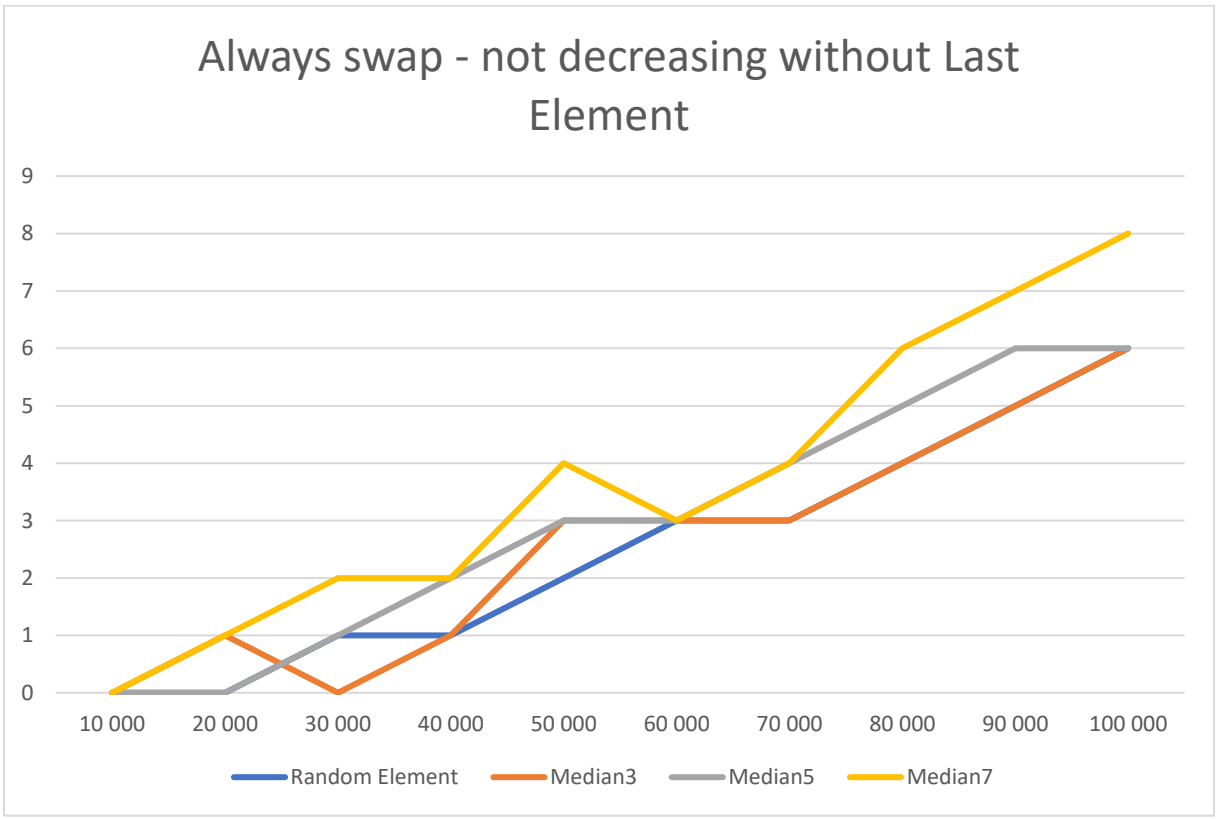


Czas zmalał około 10 krotnie, rozrzut nadal mały.

Dla losowania wartości tablicy rodzaju **niemalejący:**

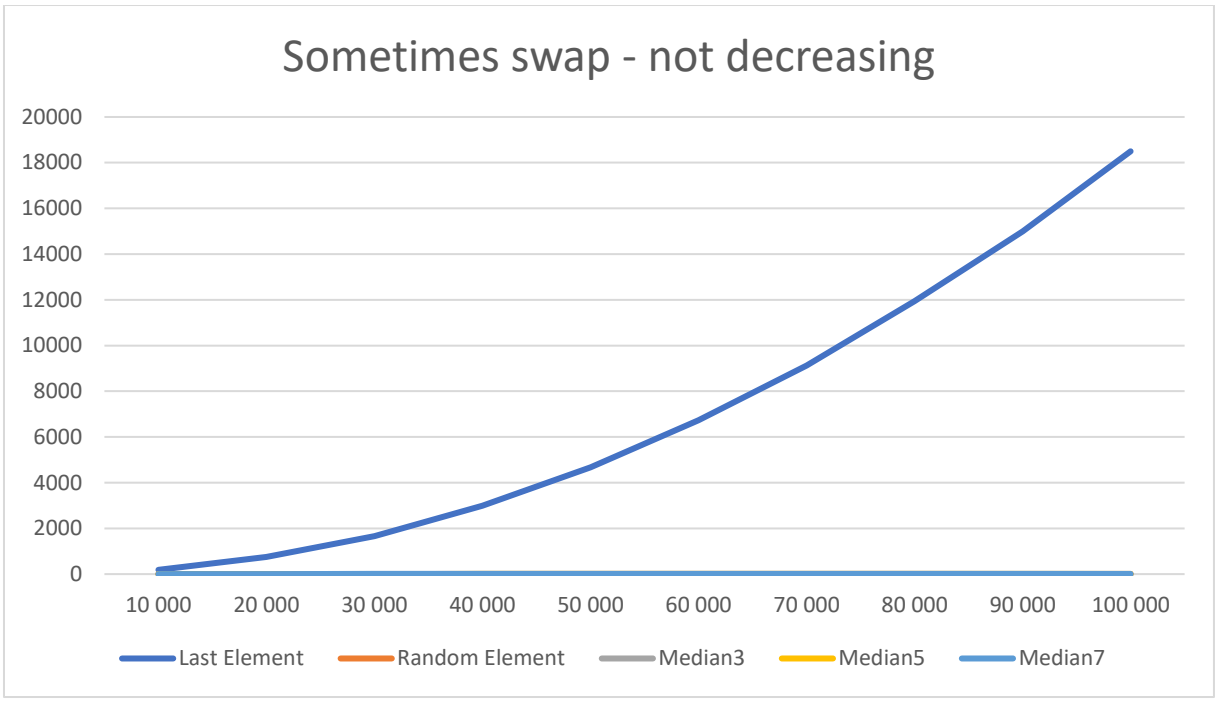
PartitionType = **AlwaysSwap**



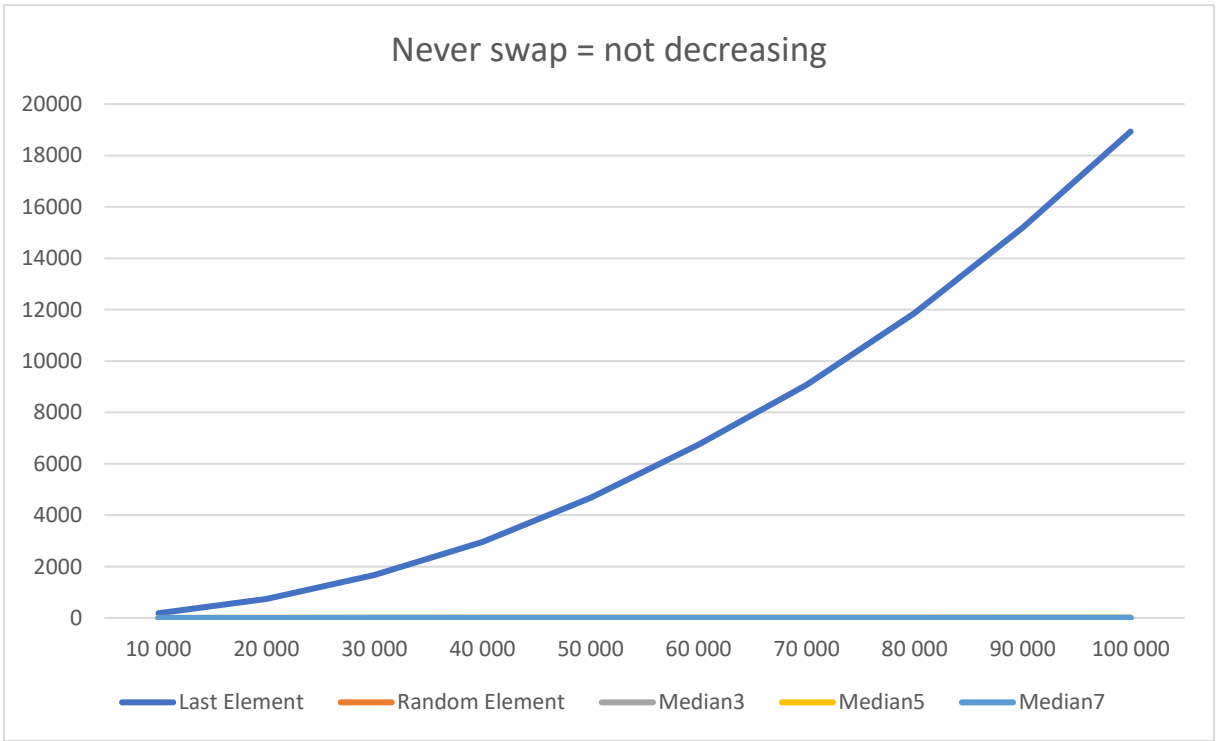


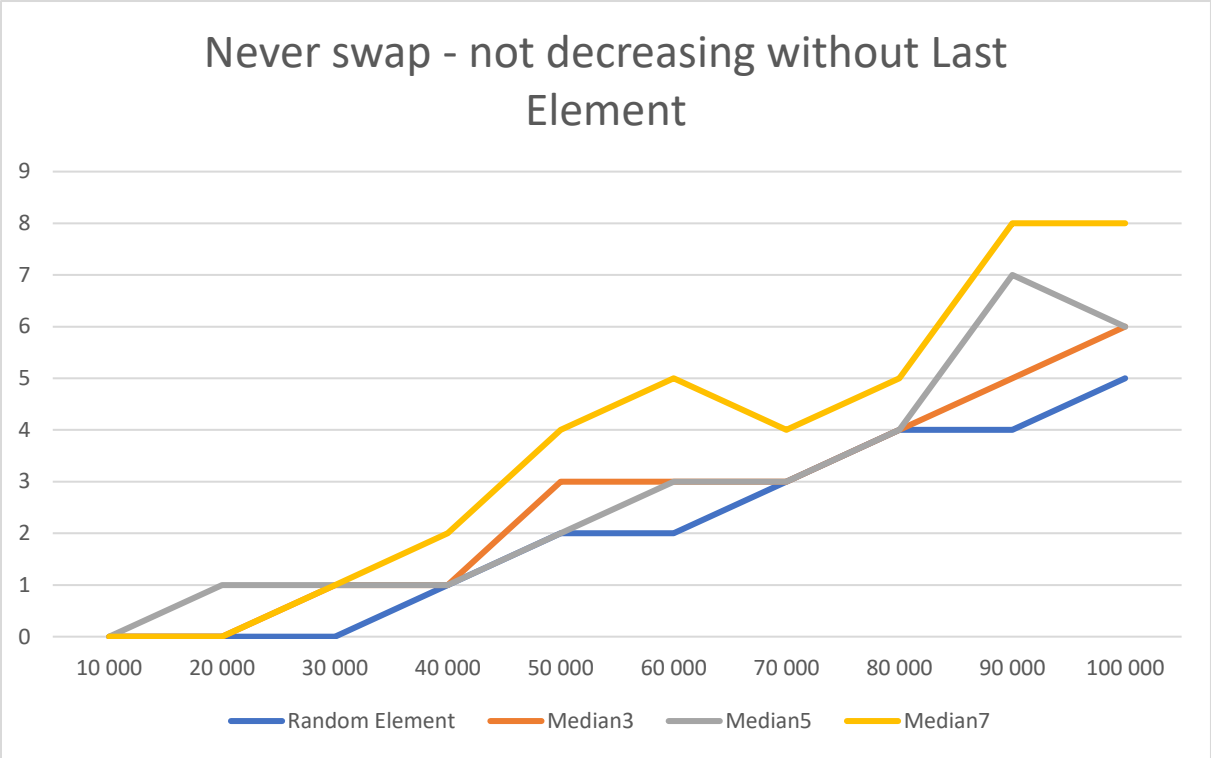
Dla sortowania, gdzie **pivot** jest wybierany jako ostatni element czas jest w granicach 20s na 100 * 1000 elementów, gdzie na resztę czasy są mniejsze niż 10 millisekund.

PartitionType = SometimesSwap



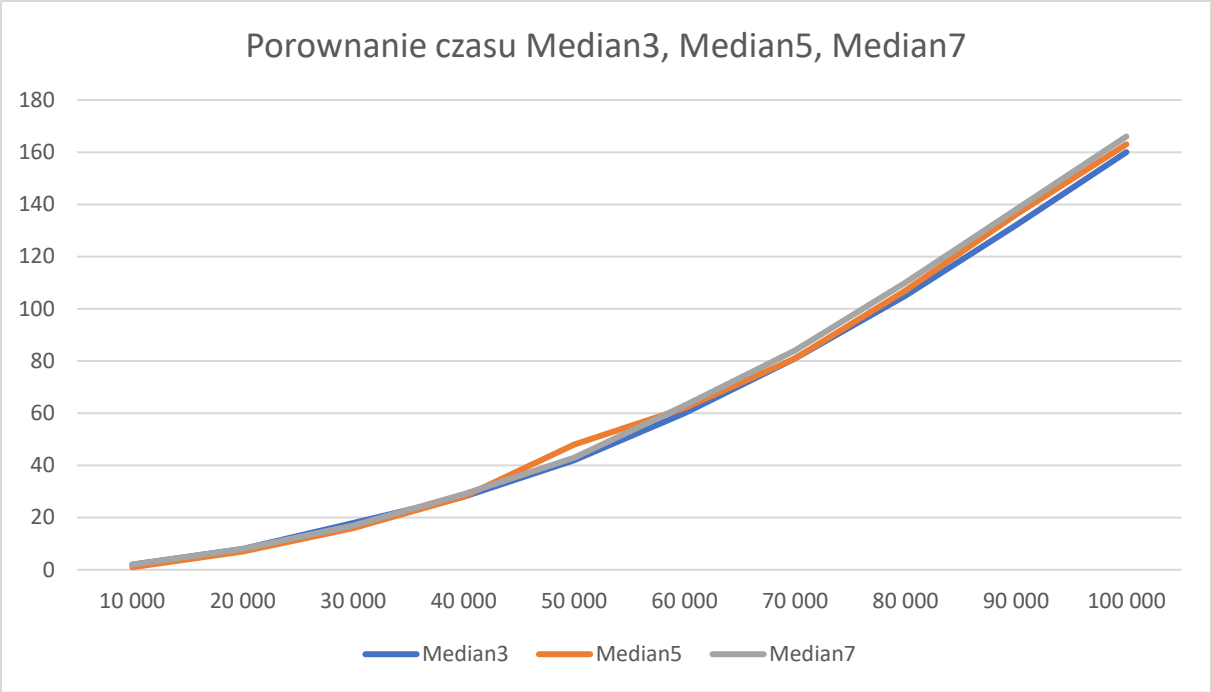
PartitionType = NeverSwap





Jak widać z wykresów, jeżeli wartości już są uporządkowane, używając QuickSort-a musimy przejść przez całą tablicę kilka razy, porównując z sobą wszystkie elementy, żeby uzyskać dokładnie tą samą tablicę z którą się zaczynało.

Możliwe jest odczytanie z wykresów, że najlepszym PartitionType jest NeverSwap, ponieważ nie obciąża to kompilatora, robiąc sortowanie szybsze. Drugim najlepszym jest AlwaysSwap, kompilator nie musi się zastanawiać czy zamienić czy nie, przez co może od razu sam zoptymalizować jak kod ma być wykonany.



Z tego wykresu wynika, że czas sortowania z wyborem pivot-a typem Median3, Median5 i Median7 są praktycznie identyczne.