

Wtorki 16:50  
Grupa I3  
Kierunek Informatyka  
Wydział Informatyki  
Politechnika Poznańska

Algorytmy i struktury danych

Sprawozdanie z zadania w zespołach nr. 1  
prowadząca: dr hab. inż. Małgorzata Sterna, prof PP

# Algorytmy sortowania

autorzy:

Piotr Więtczak nr indeksu 132339

Tomasz Chudziak nr indeksu 136691

11 kwietnia 2018

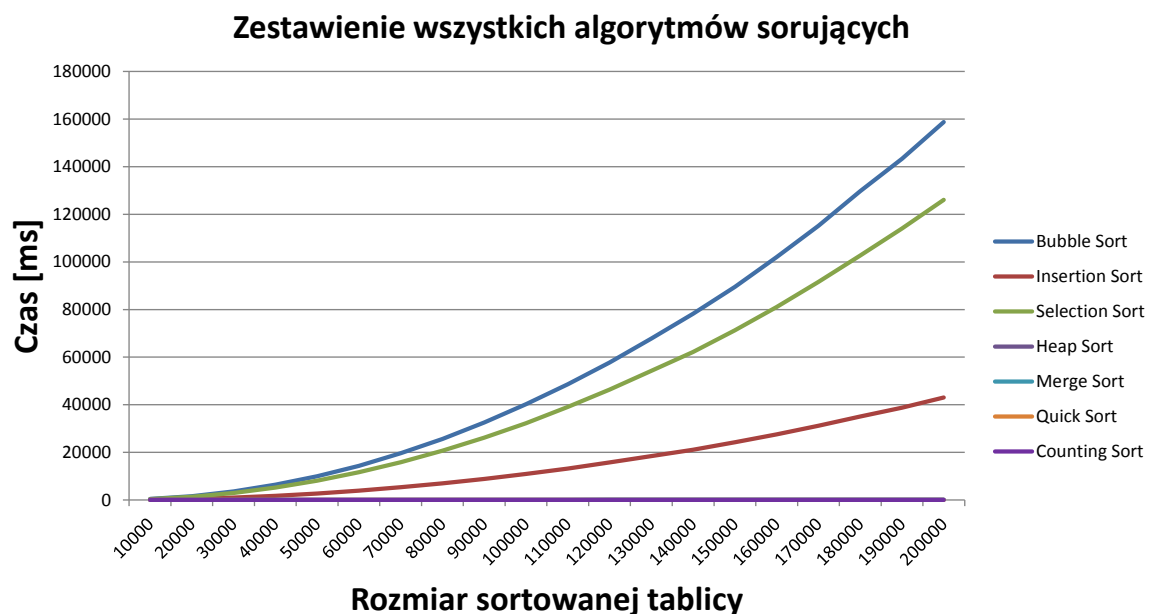
# 1 Implementacja algorytmów sortujących

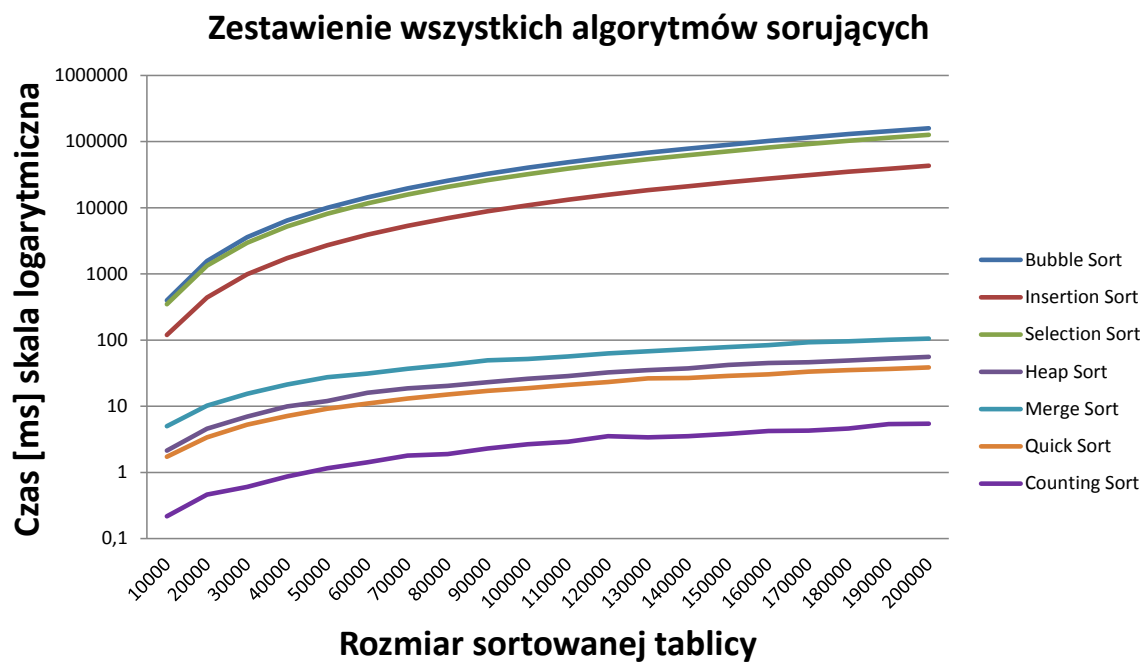
Do implementacji metod sortowania posłużyliśmy się językiem `C++`, każda metoda została napisana w odrębnej funkcji, która za parametry przyjmuje kolejno: wskaźnik na tablicę, rozmiar sortowanej tablicy oraz jako ostatni wartość opcjonalną "reverse" typu bool, która odpowiada za to czy tablica będzie posortowana rosnąco czy malejąco. Do mierzenia czasu poszczególnych metod użyliśmy klasy `std::chrono::high_resolution_clock` z biblioteki `chrono`. Program użyty do obliczania czasów sortowań, wraz z plikami nagłówkowymi zawierającymi metody sortujące, jest dostępny w formie repozytorium git pod adresem [goo.gl/snMdzD](https://github.com/snMdzD).

## 2 Badana zależność czasu obliczeń $t[s]$ od liczby sortowanych elementów $n$ .

### 2.1 Podział metod sortowania

W celu zachowania przejrzystości otrzymanych danych podzieliliśmy metody na dwie grupy, "wolne" (Insertion Sort, Selection Sort, Bubble Sort) i "szybkie" (Counting Sort, Quick Sort, Merge Sort, Heap Sort). Różnice w zależności czasu obliczeń  $t[s]$  od liczby sortowanych elementów  $n$  dla algorytmów "wolnych" i "szybkich" przedstawiają poniższe wykresy.





## Wnioski do podziału metod sortowania

Jak widać na wykresie przedstawiającym zależność czasu obliczeń od liczby sortowanych elementów, linie metodą "szybkim" zlewają się ze sobą i leżą przy samej osi OX, wykres pokazuje także jak znacząca jest różnica szybkości wykonywania sortowań między grupami. Dopiero przedstawienie danych na wykresie ze skalą logarytmiczną pozwala rozróżnić metody "szybkie".

## 2.2 Metody "wolne"

### 2.2.1 Opis algorytmów "wolnych"

#### Insertion Sort (sortowanie przez wstawianie)

Polega na wstawieniu kolejnego elementu z nieposortowanej części tablicy odpowiednie miejsce w posortowanej części tablicy.

Zalety:

- wydajny dla zbiorów wstępnie posortowanych
- działa w miejscu
- stabilny

Wady:

- wolniejszy od metod "szybkich"
- mało wydajne dla dużej liczby elementów do posortowania

Inne cechy:

- zachowanie naturalne

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Insert Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Tablica 1: Tablica złożoności obliczeniowej dla metody Insert Sort

### **Selection Sort (sortowanie przez wybieranie)**

Polega na wyszukaniu w nieposortowanej części tablicy elementu który powinien się znaleźć w pożądanym miejscu i zamianie miejscami z tym który obecnie się tam znajduje.

Zalety:

- działa w miejscu
- stabilny

Wady:

- wolniejszy od metod "szybkich"
- mało wydajne dla dużej liczby elementów do posortowania

Inne cechy:

- zachowanie naturalne

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Tablica 2: Tablica złożoności obliczeniowej dla metody Selection Sort

### Bubble Sort (sortowanie bąbelkowe)

Polega na porównaniu dwóch kolejnych elementów i zamianie ich kolejności, jeśli nie pasuje ona do porządku sortowania tablicy. Sortowanie kończy się kiedy podczas przejścia nie dokonano żadnej zmiany.

Zalety:

- działa w miejscu
- stabilny

Wady:

- wolniejszy od metod "szybkich"
- mało wydajne dla dużej liczby elementów do posortowania

Inne cechy:

- zachowanie naturalne

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

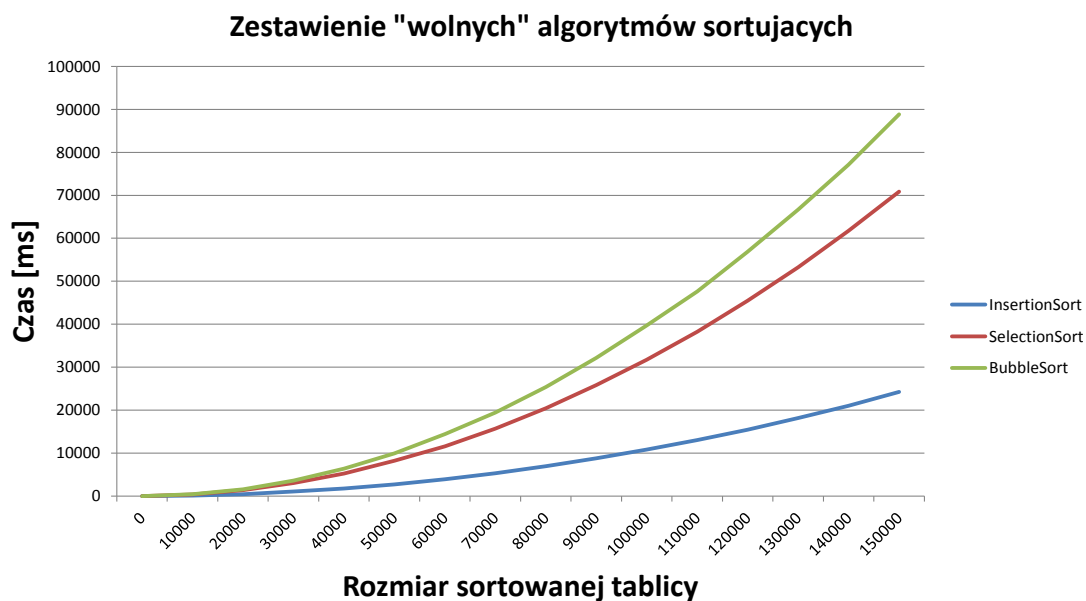
Tablica 3: Tablica złożoności obliczeniowej dla metody Bubble Sort

Tabela ilustrująca zależność czasu sortowania od liczby elementów dla metod "wolnych", zakres liczb  $[1, n]$ .

Liczba elem.	Insertion Sort	Selection Sort	Bubble Sort
10000	116,367	343,188	399,540
20000	443,897	1353,490	1553,660
30000	1031,430	3012,000	3594,100
40000	1729,670	5245,040	6400,490
50000	2736,710	8192,250	9928,660
60000	3893,300	11588,500	14420,700
70000	5305,000	15701,200	19430,200
80000	6920,760	20416,900	25364,500
90000	8782,010	25847,500	32131,700
100000	10820,400	31732,100	39683,900
110000	13004,100	38229,900	47605,000
120000	15467,000	45476,800	56842,200
130000	18181,800	53262,100	66675,100
140000	21019,300	61749,000	77105,200
150000	24263,400	70873,200	88830,400

Tablica 4: Wyniki badań zależności czasu od ilości elementów dla metod "wolnych"

Wykres ilustrujący zależność czasu sortowania od liczby elementów dla metod "wolnych", zakres liczb  $[1, n]$ .



### 2.2.2 Wnioski do metod "wolnych"

Metody "wolne", są zwykle o wiele mniej efektywne od metod "szybkich", a wraz z wzrostem liczby elementów do posortowania dysproporcja się pogłębia. Przeprowadzone badanie wskazuje, że najwolniejsze jest sortowanie bąbelkowe, a najszybsze sortowanie przez proste wstawianie.

## 2.3 Metody "szybkie"

### 2.3.1 Opis algorytmów "szybkich"

#### Quick Sort (sortowanie szybkie)

Działanie algorytmu zaczyna się od wybrania elementu podziału, w badanej przez nas implementacji był to element środkowy, następnie tablica zostaje podzielona na dwie części, do początkowej przenoszone są wszystkie elementy mniejsze od elementu podziału, nie mniejsze trafiają natomiast do końcowej części. Następnie algorytm sortuje osobno w ten sam sposób części początkową i końcową. Warunkiem końcowym rekursji będzie uzyskanie w wyniku podziału tylko jednego elementu, jednoelementowa tablica nie wymaga sortowania.

Zalety:

- działa w miejscu

Wady:

- niestabilny
- wrażliwy na dane wejściowe

Inne cechy:

- zachowanie nienaturalne
- korzysta z metody "dziel i rządź"
- algorytm rekurencyjny

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Quick Sort	$O(n \log_2(n))$	$O(n \log_2(n))$	$O(n^2)$

Tablica 5: Tablica złożoności obliczeniowej dla metody Quick Sort

### Merge Sort (sortowanie przez scalanie)

Działanie metody polega na podzieleniu sortowanej tablicy na dwie równe części (jeżeli to możliwe), następnie algorytm zostaje użyty ponownie osobno dla obu części podziału, zagłębianie rekursji trwa do momentu aż do podzielenia pozostanie tylko jeden element. Na koniec powracając z rekurencji metoda łączy posortowane podciągi w jeden ciąg posortowany.

Zalety:

- algorytm asymptotycznie optymalny
- stabilny
- niewrażliwy na dane wejściowe

Wady:

- nie działa w miejscu

Inne cechy:

- zachowanie nienaturalne
- korzysta z metody "dziel i rządź"
- algorytm rekurencyjny

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Merge Sort	$O(n \log_2(n))$	$O(n \log_2(n))$	$O(n \log_2(n))$

Tablica 6: Tablica złożoności obliczeniowej dla metody Merge Sort

### Heap Sort (sortowanie przez kopcowanie)

Działanie metody wykorzystuje strukturę binarnego kopca zupełnego. Po utworzeniu z tablicy do posortowania kopca następuje sortowanie, polega ono na zamianę miejscami korzenia kopca z ostatnim elementem i odbudowie kopca bez udziału elementów wcześniej przeniesionych na koniec tablicy. Operację powtarza się do momentu wyczerpania elementów kopca. W ten sposób w części która nie bierze ponownego udziału w kopcowaniu powstaje



posortowana tablica.

Zalety:

- działa w miejscu
- mało wrażliwy na dane wejściowe

Wady:

- niestabilny

Inne cechy:

- zachowanie nienaturalne
- korzysta ze stogów

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Heap Sort	$O(n \log_2(n))$	$O(n \log_2(n))$	$O(n \log_2(n))$

Tablica 7: Tablica złożoności obliczeniowej dla metody Heap Sort

### Counting Sort (sortowanie przez zliczanie)

Polega na sprawdzeniu ile razy w tablicy wystąpił element z sortowanego zakresu, następnie algorytm sprawdza ile wystąpień elementów mniejszych od danego występuje w sortowanej tablicy, na koniec wypisuje posortowany ciąg do tablicy wynikowej. Metoda zakłada, że elementy tablicy należą do liczb całkowitych nieujemnych.

Zalety:

- bardzo szybki algorytm sortowania dla danych z małego zakresu
- stabilny

Wady:

- nie działa w miejscu
- ograniczony ze względu na zakres sortowanych liczb
- mało wydajny dla danych z dużego przedziału

**Tabela przedstawiająca złożoność obliczeniową dla przypadków optymistycznego, średniego i pesymistycznego**

	złożoność obliczeniowa dla przypadku optymistycznego	złożoność obliczeniowa dla przypadku średniego	złożoność obliczeniowa dla przypadku pesymistycznego
Counting Sort	$O(n)$	$O(n)$	$O(n)$

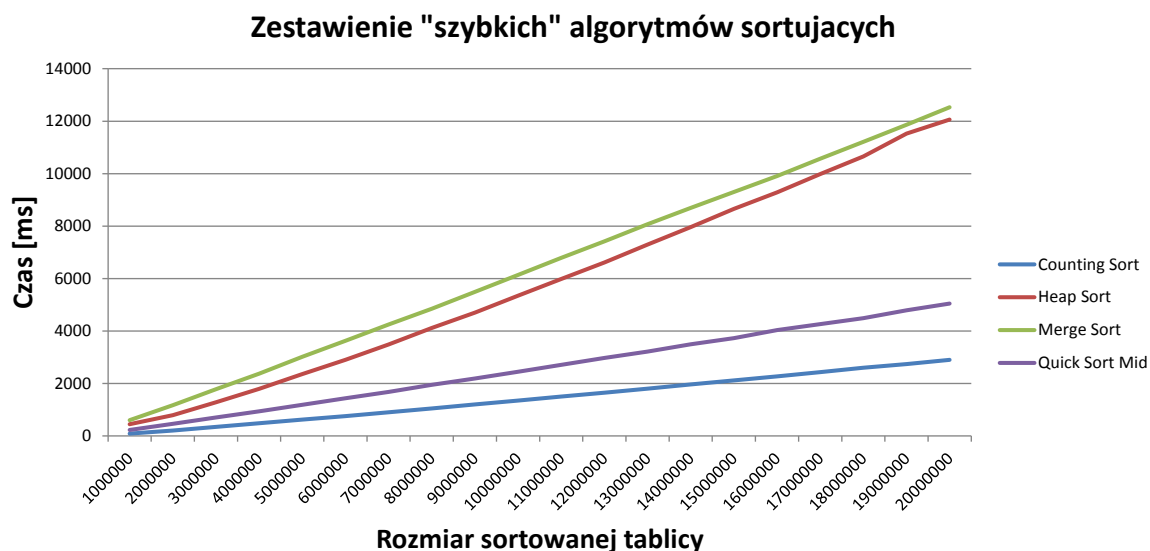
Tablica 8: Tablica złożoności obliczeniowej dla metody Counting Sort

**Tabela ilustrująca zależności czasu sortowania od liczby elementów dla metod "szybkich", zakres liczb  $[1, n]$ .**

Liczba elem.	Counting Sort	Heap Sort	Merge Sort	Quick Sort
1000000	85,986	441,528	599,972	223,697
2000000	204,386	790,249	1168,860	459,433
3000000	344,066	1281,850	1781,070	698,220
4000000	480,980	1793,700	2373,110	938,052
5000000	617,621	2358,770	3014,620	1185,580
6000000	756,102	2902,170	3626,700	1433,960
7000000	901,962	3486,930	4246,060	1675,530
8000000	1043,010	4115,850	4843,670	1947,040
9000000	1200,790	4700,670	5490,970	2185,940
10000000	1347,630	5347,840	6139,220	2446,770
11000000	1503,040	5980,280	6791,130	2704,960
12000000	1649,680	6614,990	7416,890	2969,830
13000000	1803,170	7291,140	8067,890	3210,140
14000000	1961,380	7964,010	8694,710	3489,290
15000000	2114,890	8658,490	9302,720	3724,190
16000000	2269,110	9285,660	9908,000	4032,190
17000000	2432,230	9984,300	10570,900	4258,680
18000000	2597,700	10655,100	11213,500	4489,620
19000000	2740,300	11530,300	11864,900	4786,570
20000000	2896,790	12065,700	12529,900	5046,560

Tablica 9: Wyniki badań zależności czasu od ilości elementów dla metod "szybkich"

Wykres ilustrujący zależności czasu sortowania od liczby elementów dla metod "szybkich", zakres liczb  $[1, n]$ .



### 2.3.2 Wnioski do metod "szybkich"

Metody "szybkie" są zwykle bardziej efektywne od metod "wolnych", nie ma to jednak zawsze miejsca o czym przekonamy się dalszej części sprawozdania. Badanie wskazuje, że najszybciej sortuje się metodą Counting Sort, a najwolniej Merge Sort.

## 2.4 Wnioski do podziału metod sortowania

Dzieląc metody sortowania można brać pod wiele rzeczy takich jak: czas wykonywania, dodatkowe zużycie pamięci, czy wrażliwość na dane wejściowe i wiele innych. Najczęściej wyróżnia się jednak podział ze względu na właśnie te trzy czynniki, na metody "szybkie" (Counting Sort, Merge Sort, Quick Sort, Heap Sort) i "wolne" (Insertion Sort, Selection Sort, Bubble Sort), ale także na pracujące w miejscu (Insertion Sort, Selection Sort, Bubble Sort, Heap Sort, Quick Sort) i wymagające dodatkowej pamięci (Merge Sort, Counting Sort), oraz wrażliwe na dane wejściowe (Quick Sort), mało wrażliwe na dane wejściowe (Insertion Sort, Bubble Sort, Selection Sort, Counting Sort, Heap Sort) i niewrażliwe na dane wejściowe (Merge Sort). Więcej o czynnikach wspomagających wybór metody sortowania powiemy we wnioskach do sekcji 3.

### 3 Badanie zależności czasu $t$ od liczby sortowanych elementów $n$ , przy rozkładach losowych i rosnących, dla metod Quick Sort z podziałem wg: skrajnego i środkowego elementu, oraz dla metody Insert Sort

W dalszej części sprawozdania algorytm Quick Sort z podziałem według środkowego elementu będziemy nazywać Quick Sort Mid, a z podziałem według skrajnego elementu Quick Sort Right.

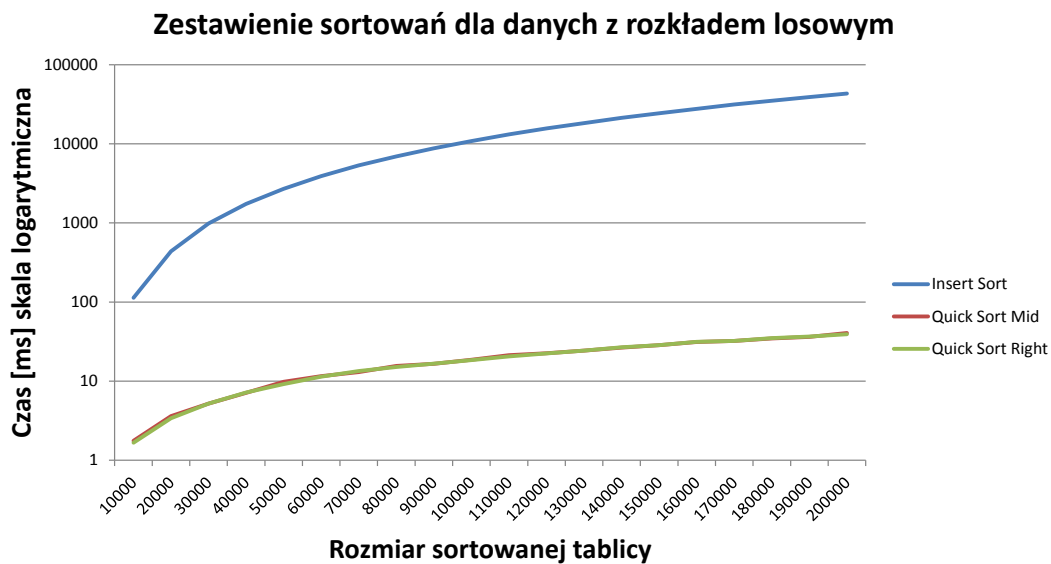
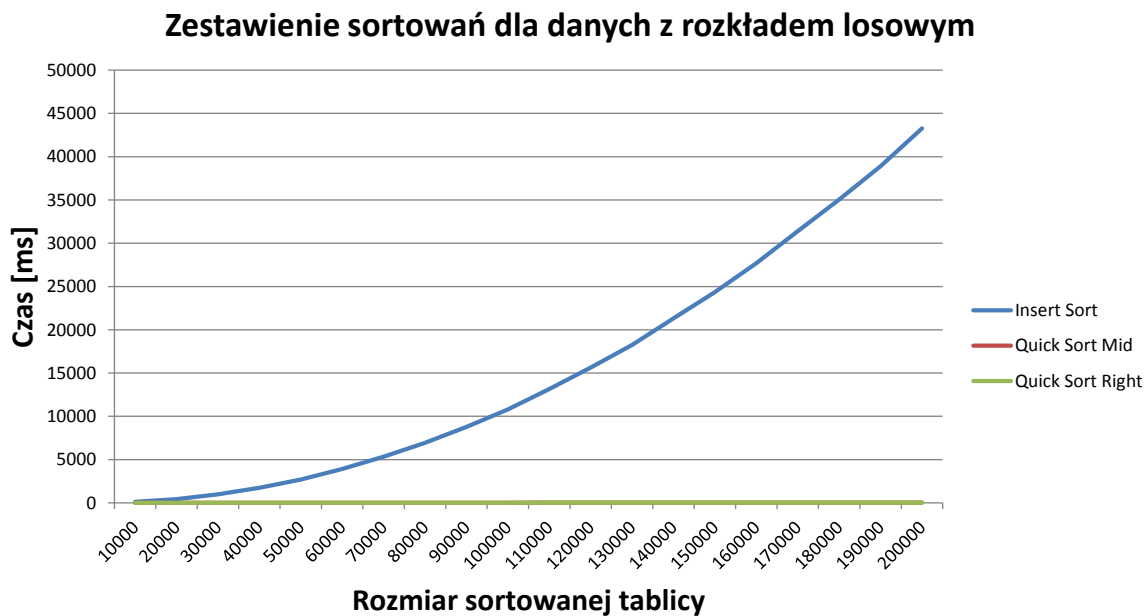
Tabela ilustrująca zależności czasu sortowania od liczby elementów dla metod Quick Sort Mid, Quick Sort Right, Insertion Sort, dla rozkładów losowego i rosnącego.

Liczba. elem.	Insert Sort roz. losowy	Quick Sort R roz. losowy	Quick Sort M roz. losowy	Insert Sort roz.rosnący	Quick Sort R roz. rosnący	Quick Sort M roz. rosnący
10000	113,203	1,669	1,764	0,032	155,830	0,838
20000	437,488	3,391	3,608	0,065	509,500	1,721
30000	984,901	5,201	5,210	0,096	969,508	2,611
40000	1735,750	7,167	7,098	0,127	1590,150	3,524
50000	2702,440	9,206	9,803	0,156	2498,850	4,637
60000	3912,190	11,418	11,567	0,195	3675,900	5,469
70000	5331,380	13,415	12,994	0,217	4884,010	6,156
80000	6932,690	15,126	15,562	0,265	6331,660	7,157
90000	8767,070	16,623	16,593	0,313	7946,000	8,442
100000	10802,700	18,438	18,636	0,331	9934,800	9,628
110000	13141,500	20,523	21,309	0,351	11940,000	9,661
120000	15639,800	22,441	22,495	0,386	13993,600	10,823
130000	18263,400	24,328	24,335	0,409	16322,700	11,251
140000	21317,500	26,808	26,532	0,425	18797,000	11,890
150000	24363,900	28,516	28,513	0,459	21520,300	13,256
160000	27679,000	31,367	31,224	0,505	24480,600	14,159
170000	31397,900	32,184	32,278	0,535	27719,300	15,593
180000	35062,800	35,045	34,674	0,547	31037,200	16,456
190000	38904,700	36,609	36,295	0,586	34542,100	19,043
200000	43261,100	39,166	40,578	0,701	38319,800	19,765

Tablica 10: Wyniki badań zależności czasu od liczby elementów dla metod Quick Sort M (z podziałem według środkowego elementu), Quick Sort R (z podziałem według skrajnego elementu), Insertion Sort, dla rozkładów losowego i rosnącego.

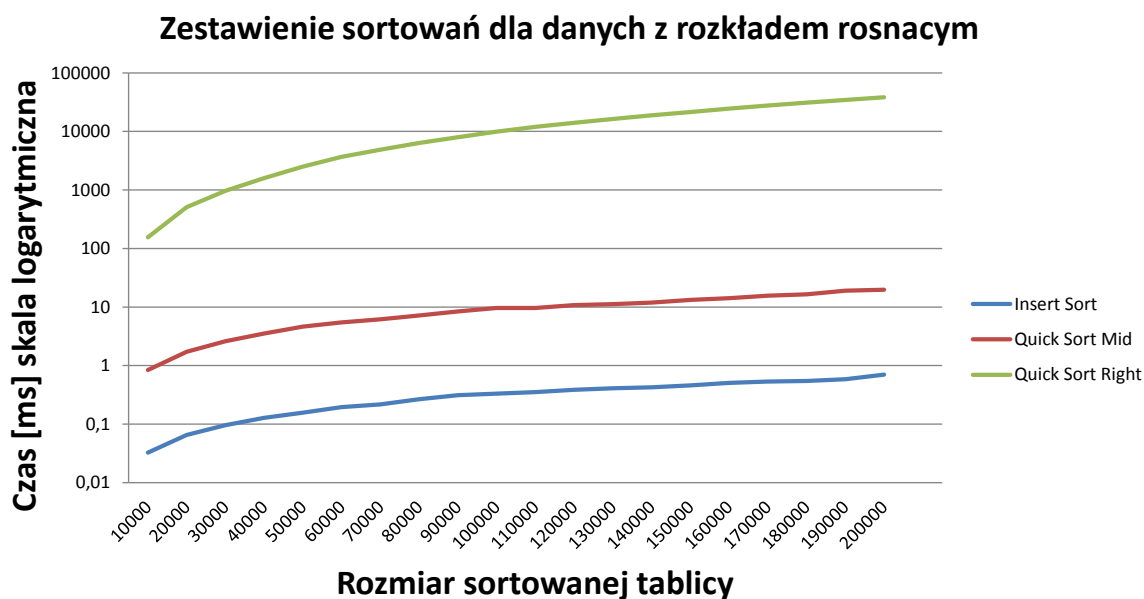
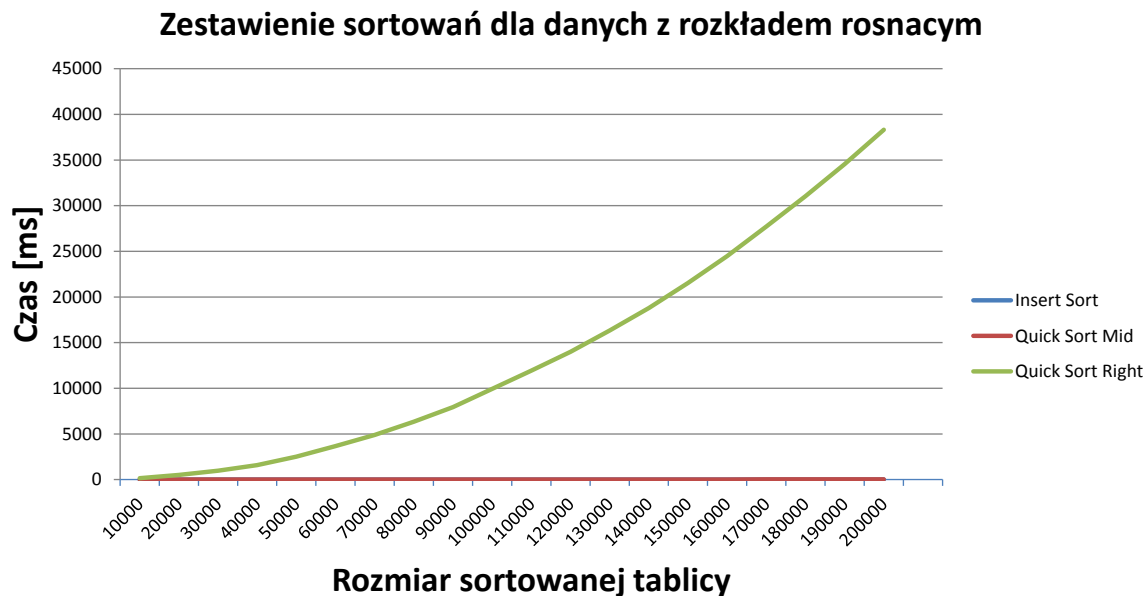
### 3.1 Rozkład losowy

Wykresy ilustrujące zależności czasu sortowania od liczby elementów dla metod Quick Sort Mid, Quick Sort Right, Insertion Sort, dla rozkładu losowego.



### 3.2 Rozkład rosnący

Wykresy ilustrujące zależności czasu sortowania od liczby elementów dla metod Quick Sort Mid, Quick Sort Right, Insertion Sort, dla rozkładu rosnącego.



### 3.3 Wnioski do zależności czasu $t$ od liczby sortowanych elementów $n$ , przy rozkładach losowych i rosnących, dla metod Quick Sort z podziałem wg: skrajnego i środkowego elementu, oraz dla metody Insert Sort

Na złożoność obliczeniową Quick Sort-a wpływa punkt podziału tablicy. Można zauważyć, że ta metoda wykorzystująca skrajny element (w przeprowadzonym badaniu prawy) radzi sobie bez porównania gorzej niż ta,

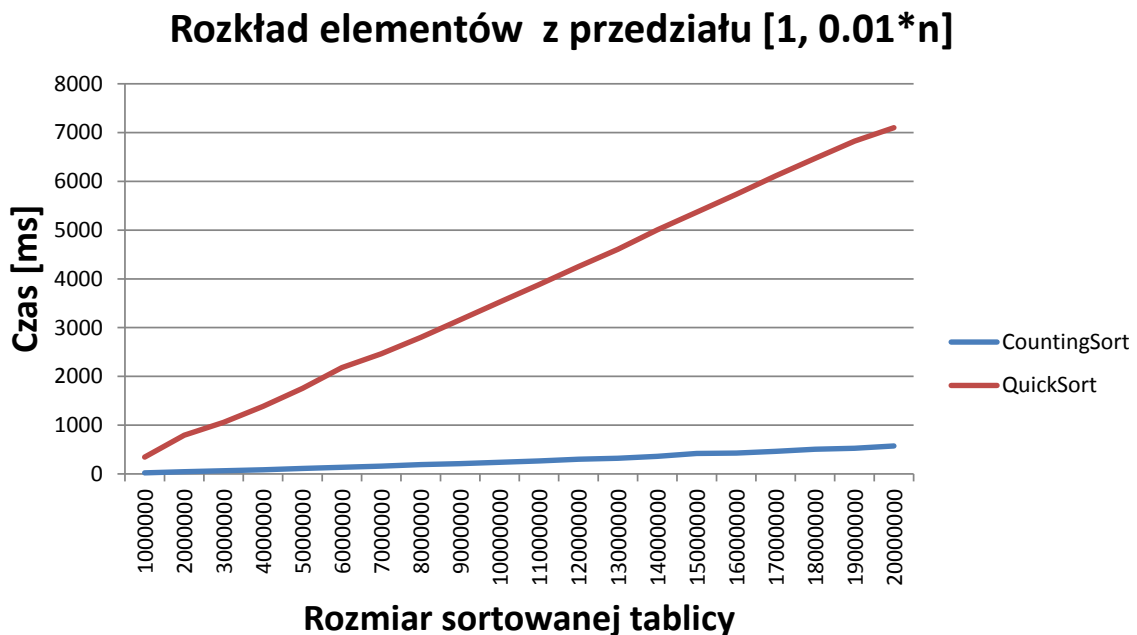
wybierając środkowy dla tablicy o rosnącym rozkładzie danych. Najprawdopodobniej wynika to z tego, że wolniejszy algorytm (dla tego typu danych) zawsze dzieli rekurencyjnie zbiór danych na jednoelementowe podzbiory, przez co liczba nieposortowanych elementów zmniejsza się tylko o jeden, co w konsekwencji prowadzi do wywołania większej liczby rekurencji. Ponadto warto zwrócić uwagę na fakt, że dla rozkładu losowego nie można wybrać jednoznacznie szybszej metody (w 12 na 20 przypadków szybszy był Quick Sort ze skrajnym podziałem tablicy, w pozostałych 8 ze środkowym elementem). Godnym uwagi wydaje się również spostrzeżenie, że Insert Sort pomimo niewspółmiernie gorszych wyników w stosunku do Quick Sort dla danych losowych, pozostawia go daleko w tyle dla danych o rozkładzie rosnącym. Z przeprowadzonego badania można wyciągnąć wnioski, że Quick Sort radzi sobie o niebo lepiej niż Insert Sort dla danych o rozkładzie losowym, jednakże sortowanie przez proste wstawianie nieporównywalnie szybciej przejdzie przez tablicę już uporządkowaną rosnąco.

Nie ma uniwersalnej metody sortowania dla każdego typu danych. Podczas wyboru algorytmu warto zwrócić uwagę na kilka aspektów. Pierwszym z nich jest zakres, z jakiego dane pochodzą, jeżeli jest on wąski, to najlepszym pomysłem będzie użyć Counting Sort-a, jednakże jeżeli pochodzi on z dużego zakresu, to ta metoda jest odradzana. Innym równie ważnym aspektem jest liczba zasobów, jakimi dysponujemy. Jeżeli nie mamy dodatkowej pamięci, przewagę zdobywają algorytmy sortujące w miejscu takie jak Insert Sort, Selection Sort czy Bubble Sort, z kolei, jeżeli ją posiadamy warto sięgnąć po metody bardziej wydajne np. Merge Sort. Kolejnym czynnikiem jest rozkład danych. Jeżeli posiadamy jakieś informacje o nim (np. czy jest rosnący, malejący lub losowy), możemy dobrać odpowiednią metodę, która przyspieszy czas trwania porządkowania informacji np. jeżeli wiemy, że dane są rosnące to rozsądnym wyborem będzie Quick Sort.

#### 4 Badanie zależności czasu obliczeń $t$ od liczby sortowanych elementów $n$ dla metod Counting Sort, Quick Sort, przy rozkładzie losowym, gdy wartości elementów mieszczą się w przedziałach $[1; 0,01n]$ , $[1; 100n]$ .

##### 4.1 Przedział $[1; 0,01n]$

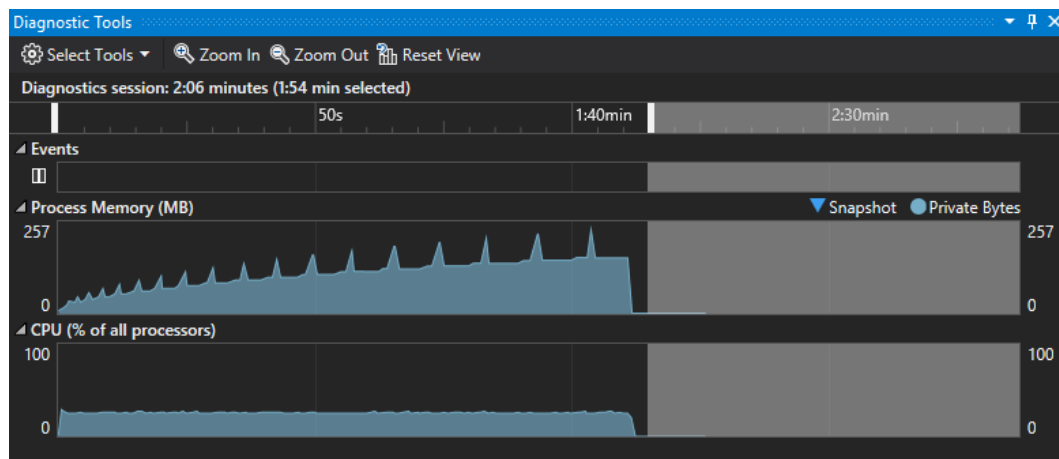
Wykres ilustrujący zależność czasu obliczeń od liczby elementów dla metod Counting Sort i Quick Sort, dla rozkładu losowego w przedziale  $[1; 0,01n]$ .



Badanie odbywało się w sposób następujący: wylosowanie tablicy do posortowania o rozmiarze  $n$  z wartościami z przedziału  $[1; 0,01n]$ , skopiowanie wylosowanej tablicy do tablicy pomocniczej, sortowanie metodą Quick Sort z pomiarem czasu, przywrócenie posortowanej tablicy do stanu przed sortowaniem przy użyciu tablicy pomocniczej, sortowanie metodą Counting Sort z pomiarem czasu, zwiększenie rozmiaru  $n$  i ponowne przeprowadzenie całego procesu, do momentu aż  $n$  osiągnie pożądaną wartość. Dzięki takiemu rozwiązaniu w sekcji Diagnostic Tools programu Visual Studio 2015 można było obserwować zauważalny wzrost zużywanej pamięci podczas metodą sortowania przez zliczanie, w porównaniu do liczby pamięci używanej w trakcie sortowania metodą szybką.



Zrzut ekranu sekcji Diagnostic Tools programu Visual Studio 2015 prezentujący, zużycie pamięci i procesora w trakcie sortowania algorytmami Counting Sort i Quick Sort, dla rozkładu losowego w przedziale  $[1; 0,01n]$ .

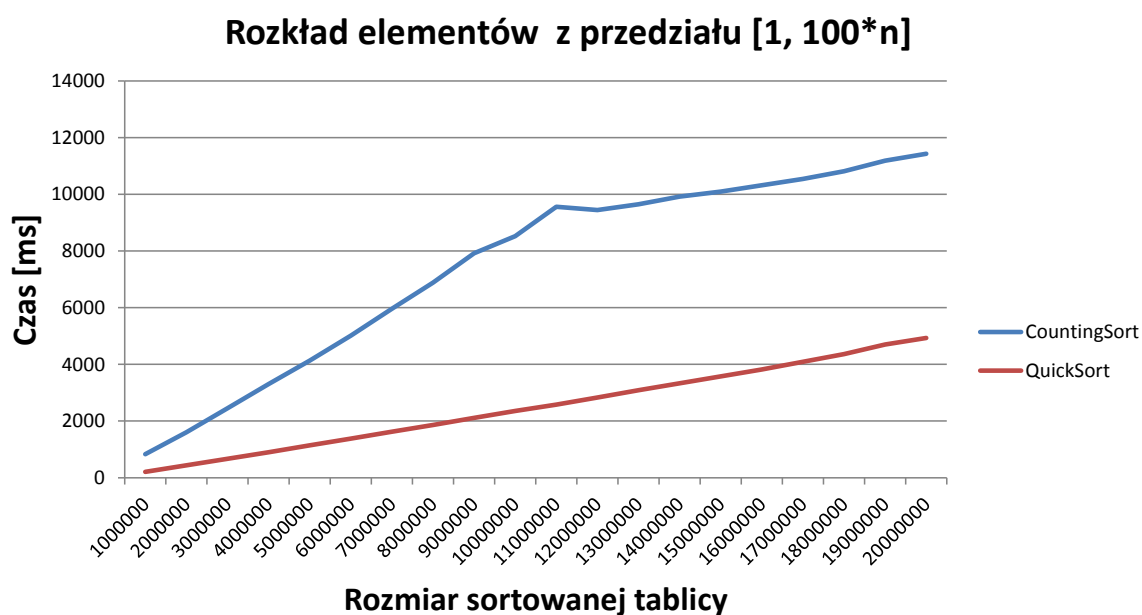


#### 4.1.1 Wnioski do przedziału $[1; 0,01n]$

Po przeanalizowaniu wyników badania można zauważyć że metoda sortowania przez zliczanie działa znacznie szybciej niż Quick Sort w przedziale  $[1; 0,01n]$ , przy trochę większym wykorzystaniu pamięci. Metoda Counting Sort sprawdza się lepiej przy sortowaniu tablic z małym zakresem liczb od metody szybkiej, co czyni ją efektywniejszą, ale tylko w przypadku kiedy mamy wystarczającą liczbą pamięci przeznaczoną dla procesu.

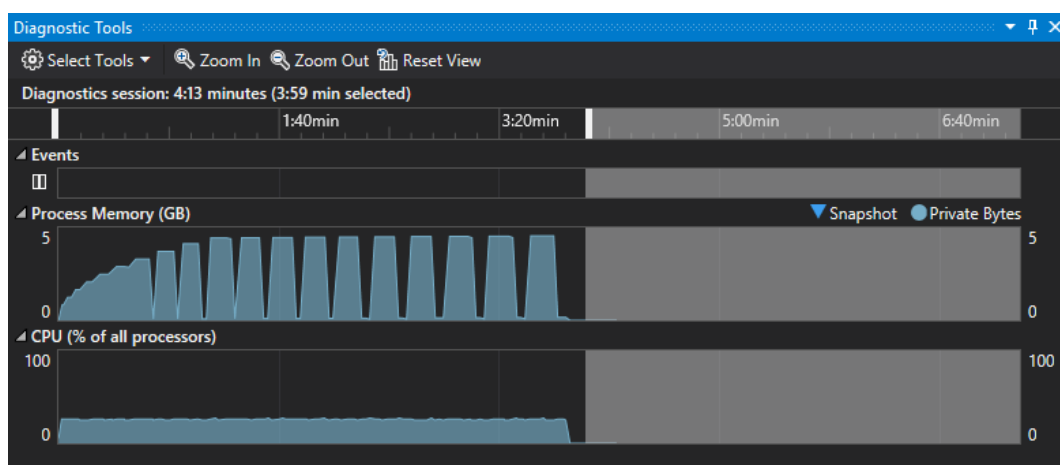
#### 4.2 Przedział $[1; 100n]$

Wykres ilustrujący zależność czasu obliczeń od liczby elementów dla metod Counting Sort i Quick Sort, dla rozkładu losowego w przedziale  $[1; 100n]$ .



Badanie odbywało się w sposób następujący: wylosowanie tablicy do posortowania o rozmiarze  $n$  z wartościami z przedziału  $[1; 100n]$ , skopiowanie wylosowanej tablicy do tablicy pomocniczej, sortowanie metodą Quick Sort z pomiarem czasu, przywrócenie posortowanej tablicy do stanu przed sortowaniem przy użyciu tablicy pomocniczej, sortowanie metodą Counting Sort z pomiarem czasu, zwiększenie rozmiaru  $n$  i ponowne przeprowadzenie całego procesu, do momentu aż  $n$  osiągnie pożądaną wartość. Dzięki takiemu rozwiązaniu w sekcji Diagnostic Tools programu Visual Studio 2015 można było obserwować nagły wzrost zużywanej pamięci podczas metodą przez zliczanie, w porównaniu do liczby pamięci używanej w trakcie sortowania metodą szybką.

**Zrzut ekranu sekcji Diagnostic Tools programu Visual Studio 2015 prezentujący, zużycie pamięci i procesora w trakcie sortowania algorytmami Counting Sort i Quick Sort, dla rozkładu losowego w przedziale  $[1; 100n]$ .**



#### 4.2.1 Wnioski do przedziału $[1; 100n]$

Po przeanalizowaniu wyników badania można zauważyć że metoda Quick Sort działa znacznie szybciej niż Counting Sort w przedziale  $[1; 100n]$ , przy nieporównywalnie większym (w niektórych przypadkach nawet dwudziestokrotnie) wykorzystaniu pamięci. Metoda szybka sprawdza się lepiej przy sortowaniu tablic z dużym zakresem liczb od metody przez liczenie, co czyni ją efektywniejszą.

### 4.3 Wnioski do badania zależności czasu obliczeń $t$ od liczby sortowanych elementów $n$ dla metod Counting Sort, Quick Sort, przy rozkładzie losowym, gdy wartości elementów mieszczą się w przedziałach $[1; 0,01n], [1; 100n]$ .

Po przeprowadzeniu badania jasno widać jak ważne jest dobranie odpowiedniej metody sortowania dla swojego rodzaju danych. Na efektywność użytej metody ma wpływ między innymi przedział elementów. W przypadku kiedy pracujemy na mniejszym przedziale warto wybrać metodę sortowania przez zliczanie, jeżeli posiadamy wystarczającą liczbę pamięci. Przy większym przedziale metoda Quick Sort staje się efektywniejsza, Counting Sort nie tylko działa wolniej, ale też zużywa wielokrotnie więcej pamięci.

# Spis treści

<b>1</b>	<b>Implementacja algorytmów sortujących</b>	<b>1</b>
<b>2</b>	<b>Badana zależność czasu obliczeń <math>t[s]</math> od liczby sortowanych elementów <math>n</math>.</b>	<b>1</b>
2.1	Podział metod sortowania	1
2.2	Metody "wolne"	2
2.2.1	Opis algorytmów "wolnych"	2
2.2.2	Wnioski do metod "wolnych"	6
2.3	Metody "szybkie"	6
2.3.1	Opis algorytmów "szybkich"	6
2.3.2	Wnioski do metod "szybkich"	10
2.4	Wnioski do podziału metod sortowania	10
<b>3</b>	<b>Badanie zależności czasu <math>t</math> od liczby sortowanych elementów <math>n</math>, przy rozkładach losowych i rosnących, dla metod Quick Sort z podziałem wg: skrajnego i środkowego elementu, oraz dla metody Insert Sort</b>	<b>11</b>
3.1	Rozkład losowy	12
3.2	Rozkład rosnący	13
3.3	Wnioski do zależności czasu $t$ od liczby sortowanych elementów $n$ , przy rozkładach losowych i rosnących, dla metod Quick Sort z podziałem wg: skrajnego i środkowego elementu, oraz dla metody Insert Sort	13
<b>4</b>	<b>Badanie zależności czasu obliczeń <math>t</math> od liczby sortowanych elementów <math>n</math> dla metod Counting Sort, Quick Sort, przy rozkładzie losowym, gdy wartości elementów mieszczą się w przedziałach <math>[1; 0,01n]</math>, <math>[1; 100n]</math>.</b>	<b>15</b>
4.1	Przedział $[1; 0,01n]$	15
4.1.1	Wnioski do przedziału $[1; 0,01n]$	16
4.2	Przedział $[1; 100n]$	16
4.2.1	Wnioski do przedziału $[1; 100n]$	17
4.3	Wnioski do badania zależności czasu obliczeń $t$ od liczby sortowanych elementów $n$ dla metod Counting Sort, Quick Sort, przy rozkładzie losowym, gdy wartości elementów mieszczą się w przedziałach $[1; 0,01n], [1; 100n]$ .	17