Łukasz Przywarty 171018
Mariusz Kacała 171058

Data utworzenia: **24.03.2010r.**
Prowadzący: **prof. dr hab. inż. Adam Janiak** oraz **dr inż. Tomiasz Krysiak**

Zadanie projektowe 1:
Struktury danych i złożoność obliczeniowa

Temat: „Badanie efektywności algorytmów sortowania w zależności od liczby sortowanych elementów”

I. Szczegółowy opis zadania projektowego

Należy zaimplementować oraz przeprowadzić pomiary czasu działania algorytmów sortowania. Należy zaimplementować następujące algorytmy:

- sortowanie bąbelkowe,
- sortowanie Shella,
- sortowanie przez scalanie,
- sortowanie przez kopcowanie,
- QuickSort,
- sortowanie kubełkowe,
- sortowanie introspektywne (introspektywne na ocenę celującą).

Należy zmierzyć czasy działania powyższych algorytmów w zależności od liczby n sortowanych elementów. Pomiarów należy dokonywać wielokrotnie dla ustalonego n i wyznaczyć wartości średnie. Sortowanie Shella zbadać dla różnych wartości parametrów odległości (parametry h-sortingu). Sortowanie QuickSort zbadać dla różnych metod doboru elementu podziału i różnego typu danych wejściowych (elementy posortowane, odwrotnie posortowane itp.)

II. Wstęp

Sortowanie jest jednym z najczęściej rozwiązywanych problemów w programowaniu. Wiąże się to z faktem, iż dużo szybciej można znaleźć informacje w zbiorach uporządkowanych. Problem sortowania możemy przedstawić następująco:

Należy uporządkować dane w n-elementowej tablicy ***A[0 ... n-1]****, która zawiera nieuporządkowany ciąg wartości* $a_0, a_1, a_2, \ldots, a_{n-1}$ *w taki sposób aby każdy element był mniejszy od poprzedniego (lub większy):* $a_0 \leq a_1 \leq a_2 \leq \ldots \leq a_{n-1}$ *. Porządkując dane możemy jedynie porównywać elementy i przepisywać je bądź zamieniać miejscami.*

III. Implementacja algorytmów sortowania

1. Sortowanie bąbelkowe

Algorytm sortowania bąbelkowego charakteryzuje się olbrzymią prostotą i z tego powodu jest najpopularniejszym algorytmem sortowania.

1.1 Opis algorytmu: Tablica jest sukcesywnie przeglądana od dołu do góry. Analizowane są zawsze dwa sąsiadujące ze sobą elementy, jeśli w dołu jest element mniejszy zostaje on zamieniony miejscami z elementem większym. W każdej fazie sortowania strefa pracy algorytmu redukuje się o 1.

1.2 Wyniki pomiarów:

Dla każdej wartości n (ilości elementów tablicy) wykonano 50 pomiarów czasu działania algorytmu. Wyniki zawarte w poniższej tabeli są średnią z tych pomiarów.

Przy sortowaniu rozpatrzyliśmy następujące przypadki:

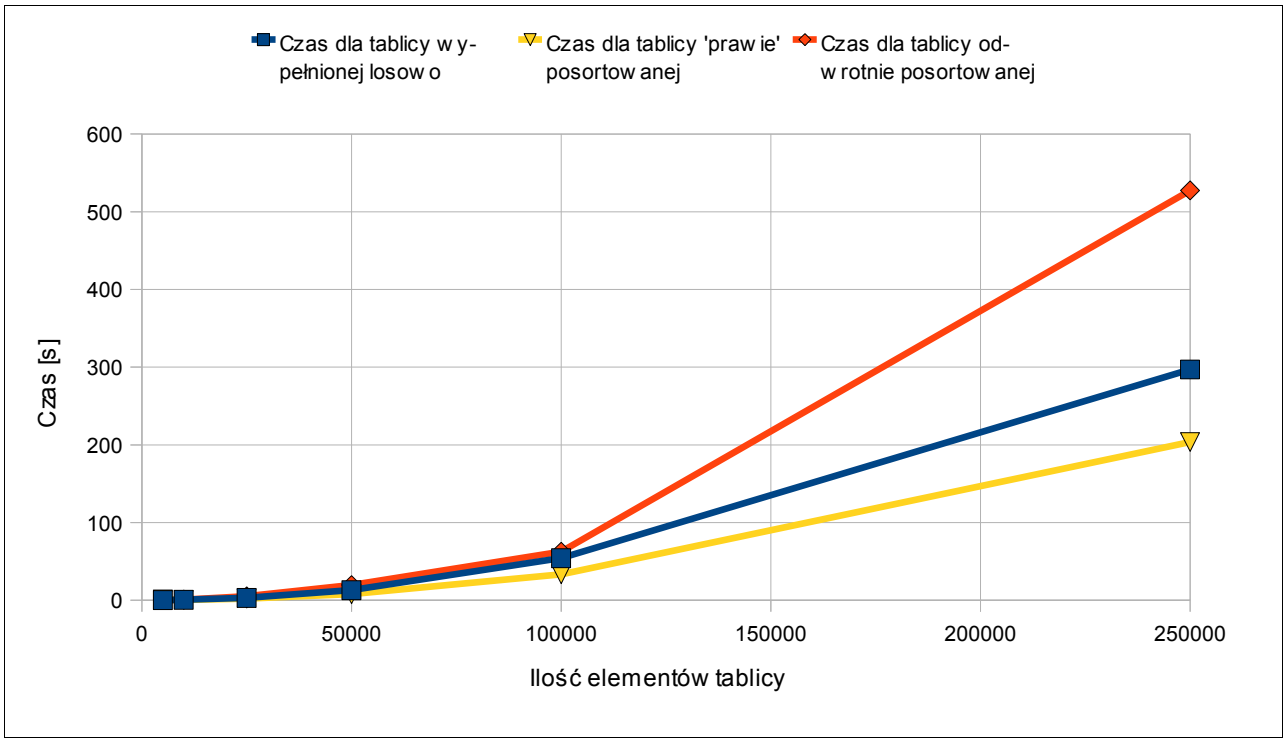
- tablica jest wypełniona losowymi liczbami z przedziału 0-30000,
- tablica jest odwrotnie posortowana,
- tablica jest 'prawie' posortowana.

Takie same przypadki będą analizowane dla każdego algorytmu sortowania.

Konfiguracja sprzętowa komputera, na którym uruchamiane były algorytmy:

- procesor: Intel Core 2 Duo: 2,4 Ghz
- pamięć: 3 GB pamięci RAM
- system operacyjny: Windows XP

Ilość elementów tablicy		**5000**	**10000**	**25000**	**50000**	**100000**	**250000**
Czas dla tablicy wypełnionej losowo	**[s]**	0,165	0,609	2,948	13,031	54,142	297,021
Czas dla tablicy odwrotnie posortowanej	**[s]**	0,179	0,844	5,091	19,516	62,48	527,451
Czas dla tablicy 'prawie' posortowanej	**[s]**	0,784	0,337	2,071	7,770	33,328	203,587



1.3 Analiza wyników:

Jak widać na wykresie im więcej elementów tym więcej czasu jest potrzebne na przeprowadzenie sortowania. Sortowanie metodą bąbelkową jest dość efektywne dla tablic 'prawie' posortowanych.

Zalety:

- bardzo łatwy do zaimplementowania algorytm.

Wady

- zdarzają się 'puste przebiegi' – nie dokonuje się żadna wymiana ze względu na to, że elementy są już posortowane,
- algorytm jest wrażliwy na konfiguracje danych np. przypadek gdy najmniejsza wartość jest na samym końcu – zwiększa to ilość niezbędnych wymian,
- algorytm czasochłonny, niewydajny dla większej ilości danych.

1.4 Złożoność:

O(n^2) – złożoność kwadratowa

2. Sortowanie Shella

Sortowanie Shella jest rozszerzeniem sortowania przez wstawianie. Proces sortowania jest znacznie przyspieszony dzięki możliwości wymiany odległych od siebie elementów.

2.1 Opis algorytmu:

Sortowany zbiór dzielimy na podzbiory , których elementy są od siebie odległe o pewien odstęp **h**. Każdy z tych podzbiorów sortujemy algorytmem sortowania przez wstawianie. Zmiejszamy odstęp h (ilość podzbiorów zmniejsza się) i kolejny raz stosujemy sortowanie przez wstawianie.

Procedurę powtarzamy do momentu gdy h wyniesie 1.

2.2 Implementacja:

Algorytm można zaimplementować w zależności od parametru h. Shell proponował h początkowe równe połowie ilości elementów zbioru

- **$h = n/2$.**

Propozycja Shella jest jednak jedną z najgorszych ze względu na to, iż w kolejnych podzbiorach występują wielokrotnie te same elementy. Inne rozwiązanie znalazł Donald Knuth (sekwencja skoków 1, 3, 7, 15, 31...):

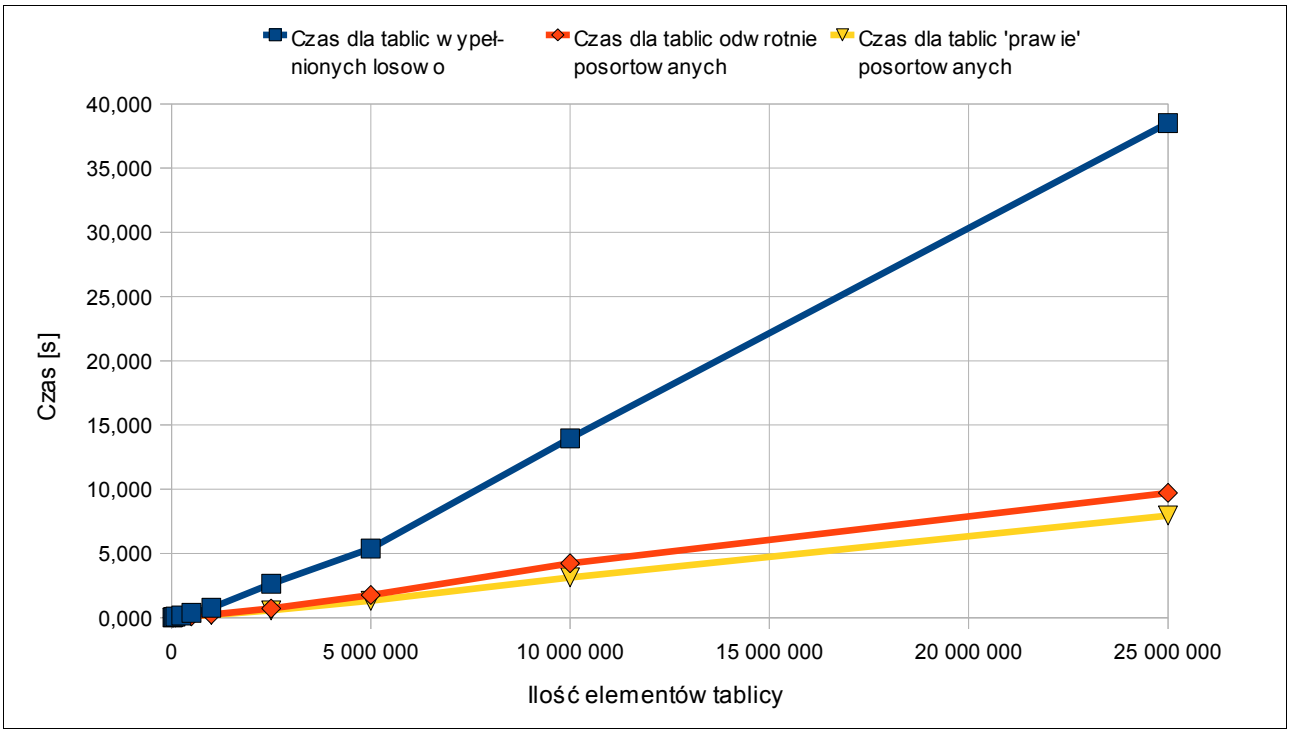
- **$h_0 = 1$**
- **$h_i = 3h_{i-1} + 1$ dla $i \geq 1$**

2.3 Wyniki pomiarów:

Pomiary przeprowadziliśmy odrębnie dla sortowania z parametrem h proponowanym przez Shella i Knutha według założeń z punktu 1.3 niniejszej pracy.

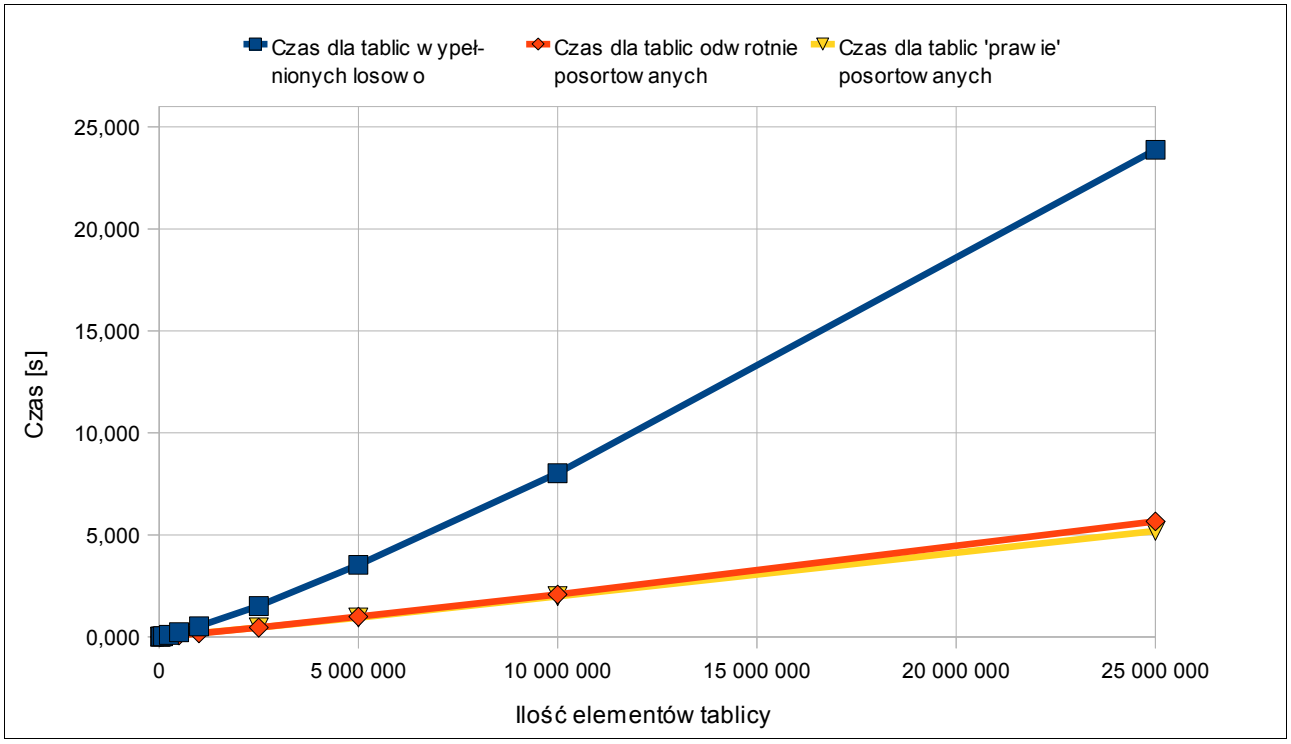
Wyniki dla h Shella:

Ilość elementów tablicy [tys]		**25**	**50**	**100**	**250**	**500**	**1000**	**2500**	**5000**	**10000**	**25000**
Czas dla tablicy wypełnionej losowo	**[s]**	0,016	0,040	0,790	0,174	0,367	0,767	2,642	5,382	13,966	38,512
Czas dla tablicy odwrotnie posortowanej	**[s]**	0,014	0,017	0,026	0,056	0,112	0,25	0,717	1,767	4,234	9,724
Czas dla tablicy 'prawie' posortowanej	**[s]**	0,010	0,020	0,030	0,060	0,110	0,210	0,570	1,310	3,140	7,960



oraz h Knutha:

Ilość elementów tablicy [tys]		25	50	100	250	500	1000	2500	5000	10000	25000
Czas dla tablicy wypełnionej losowo	[s]	0,007	0,016	0,036	0,102	0,232	0,521	1,519	3,535	8,029	23,873
Czas dla tablicy odwrotnie posortowanej	[s]	0,003	0,007	0,014	0,042	0,078	0,18	0,454	0,984	2,079	5,664
Czas dla tablicy 'prawie' posortowanej	[s]	0,004	0,007	0,015	0,042	0,088	0,185	0,462	0,947	1,995	5,199



2.4 Analiza wyników:

Ze względu na lepszą implementację algorytm z h zaproponowanym przez Knutha jest o wiele szybszy. Algorytm sortowania Shella jest nieefektywny dla tablic wypełnionych wartościami losowymi.

Zalety:

- mimo złożoności rzędu n^2 algorytm jest o wiele szybszy od sortowania bąbelkowego
- wydajny dla zbiorów 'prawie' posortowanych oraz tablic o małej liczebności

2.5 Złożoność

Podobnie jak w przypadku sortowania bąbelkowego **O(n^2)**. Algorytm Shella jest jednak szybszy.

3. Sortowanie przez scalanie

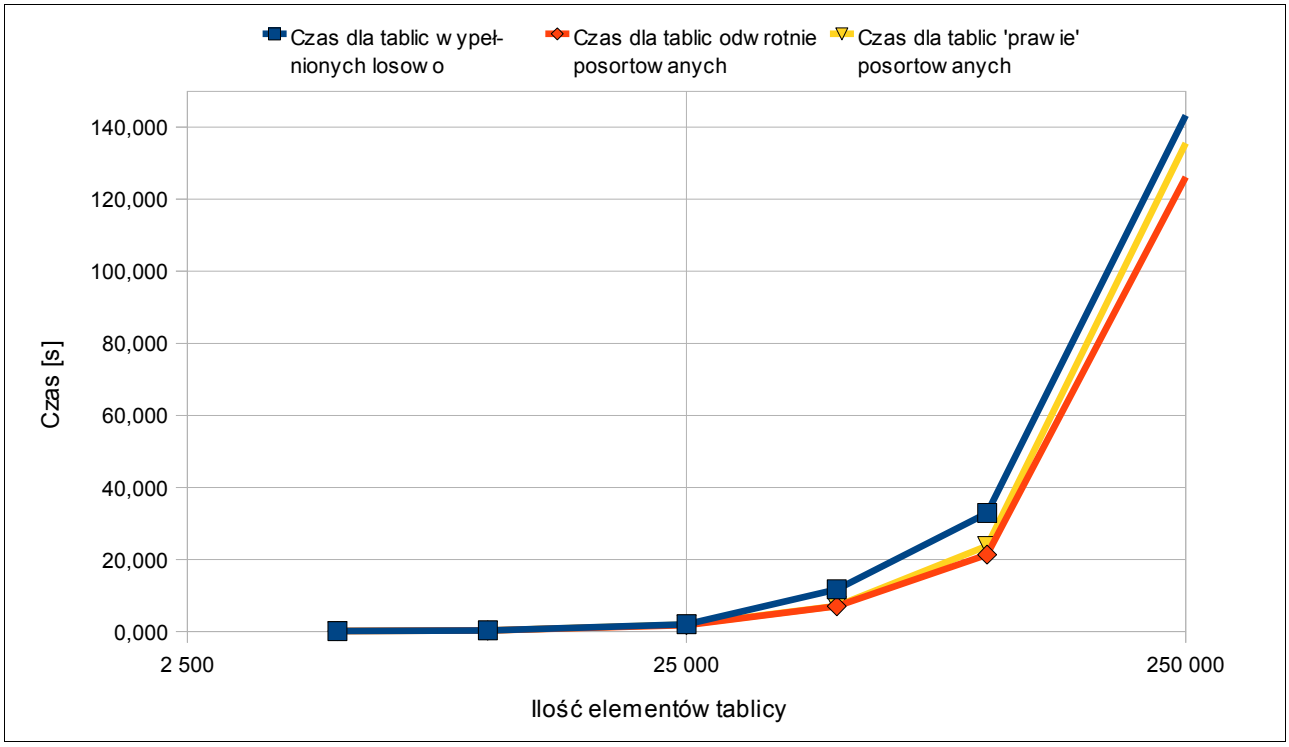
Sortowanie przez scalanie to (podobnie jak Quicksort) metoda rekurencyjna z gatunku „dziel i zwyciężaj”.

3.1 Opis algorytmu:

Dzielimy zbiór o wielkości n na dwa podzbiory, które maja n/2 elementów. Następnie sortujemy przez scalanie każdy podzbiór i łączymy posortowane podzbiory w jeden zbiór.

3.2 Wyniki pomiarów:

Ilość elementów tablicy		5000	10000	25000	50000	100000	250000
Czas dla tablicy wypełnionej losowo	[s]	0,094	0,283	2,006	11,680	32,891	143,250
Czas dla tablicy odwrotnie posortowanej	[s]	0,088	0,280	1,779	6,996	21,32	126,172
Czas dla tablicy 'prawie' posortowanej	[s]	0,090	0,266	1,919	7,091	23,703	135,612



3.3 Analiza wyników:

Jak widać na wykresie algorytm sortowania przez scalanie najdłużej działa kiedy ma posortować losowe dane w tablicy. Najlepszą efektywność algorytm wykazuje sortując dane odwrotnie posortowane i 'prawie' posortowane, przy założeniu, że elementów jest niewiele.

Zalety:

- rekurencyjny algorytm klasy 'dziel i zwyciężaj'
- szybko sortuje tablice o małej ilości elementów (do kilkudziesięciu tysięcy)

Wady:

- słabo radzi sobie z większymi ilościami danych

3.4 Złożoność

Algorytm sortowania przez scalanie charakteryzuje się złożonością **O(n log n)**

4. Sortowanie przez kopcowanie

Algorytm sortowania przez kopcowanie używa struktury zwanej kopcem, który łatwo można zaimplemen-

tować w postaci tablicy:

- wierzchołek kopca wstawiamy do A[0]
- dla każdego węzła powstają zależności: lewe dziecko w A[2i+1] oraz prawe dziecko w A[2i+2]

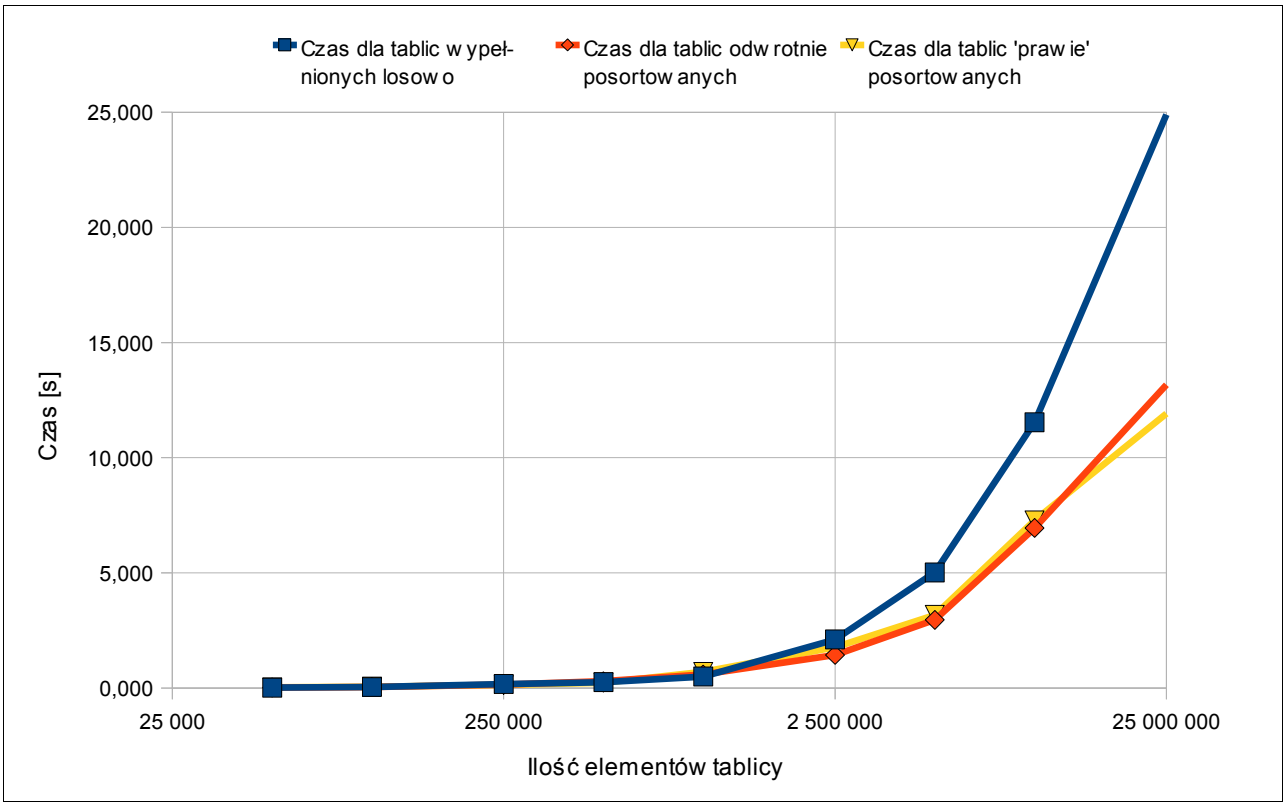
4.1 Opis algorytmu:

Tworzymy strukturę kopca i układamy elementy w tablicy (wierzchołek będzie elementem pierwszym, kolejne elementy to dzieci, następne dzieci dzieci itd., aż dotrzemy na sam dół kopca). Usuwamy wierzchołek z kopca poprzez zamianę z ostatnim liściem z drzewa. Dla pozostałej części kopca przywracamy własność kopca, usuwamy kolejny wierzchołek itd. Procedura przywracania własności kopca przedstawia się następująco:

- Jeśli wierzchołek jest większy od dzieci – kończymy
- Zamieniamy wierzchołek z większym dzieckiem i przywracamy własność kopca w części, w której zastosowaliśmy zmiany.

4.2 Wyniki pomiarów:

Ilość elementów tablicy [tys]		**50**	**100**	**250**	**500**	**1000**	**2500**	**5000**	**10000**	**25000**
Czas dla tablicy wypełnionej losowo	**[s]**	0,018	0,043	0,173	0,256	0,496	2,109	5,022	11,542	24,900
Czas dla tablicy odwrotnie posortowanej	**[s]**	0,025	0,047	0,151	0,294	0,58	1,434	2,960	6,950	13,160
Czas dla tablicy 'prawie' posortowanej	**[s]**	0,027	0,057	0,133	0,234	0,703	1,776	3,172	7,278	11,910



4.3 Analiza wyników:

Łatwo zauważyć, że algorytm sortowania przez kopcowanie dobrze radzi sobie z elementami posortowanymi odwrotnie oraz elementami 'prawie' posortowanymi, ma jednak problem z tablicami wypełnionymi elementami losowymi. Dla dużej ilości elementów sortowanie przez kopcowanie jest dość szybkie, jednak mniej efektywne niż Quicksort czy sorotowanie kubełkowe.

Zalety:

- korzystanie ze struktury kopca gwarantuje, że oprócz stałego czasu dostępu do elementu maksymalnego (lub minimalnego) uzyskamy logarytmiczny czas wstawiania i usuwania elementów
- dość wydajnie sortuje tablice elementów odwrotnie posortowanych.

Wady:

- sortowanie jest niestabilne

4.4 Złożoność

Algorytm sortowania przez kopcowanie ma złożoność równą: **O(n log n)**

5. Quicksort

Popularny algorytm sortowania szybkiego (w skrócie Quicksort), który pozwala osiągnąć znaczny zysk szybkości sortowania.

5.1 Opis algorytmu:

Procedura sortowania dzieli się zasadniczo na dwie częśći:

- część odpowiedzialną za właściwe sortowanie,
- część rozdzielającą elementy tablicy względem wartości pewnej komórki tablicy (służy ona za oś podziału).

Gdy zostanie odczytany element osiowy P tablica jest dzielona na podzbiory. W jednym (lewym) podzbiorze znajdują się elementy mniejsze od wartości znajdującej się w komórce P, natomiast w drugim (prawym) podzbiorze elementy większe od wartości P. Kolejnym etapem jest aplikacja algorytmu Quicksort na lewym i prawym fragmencie tablicy.

5.2 Impementacja:

Impelemtacja algorytmu sortowania Quicksort zależy od wyboru wartości osiowej P. Możliwości jest wiele, jednak wybraliśmy dwie najbardziej popularne:

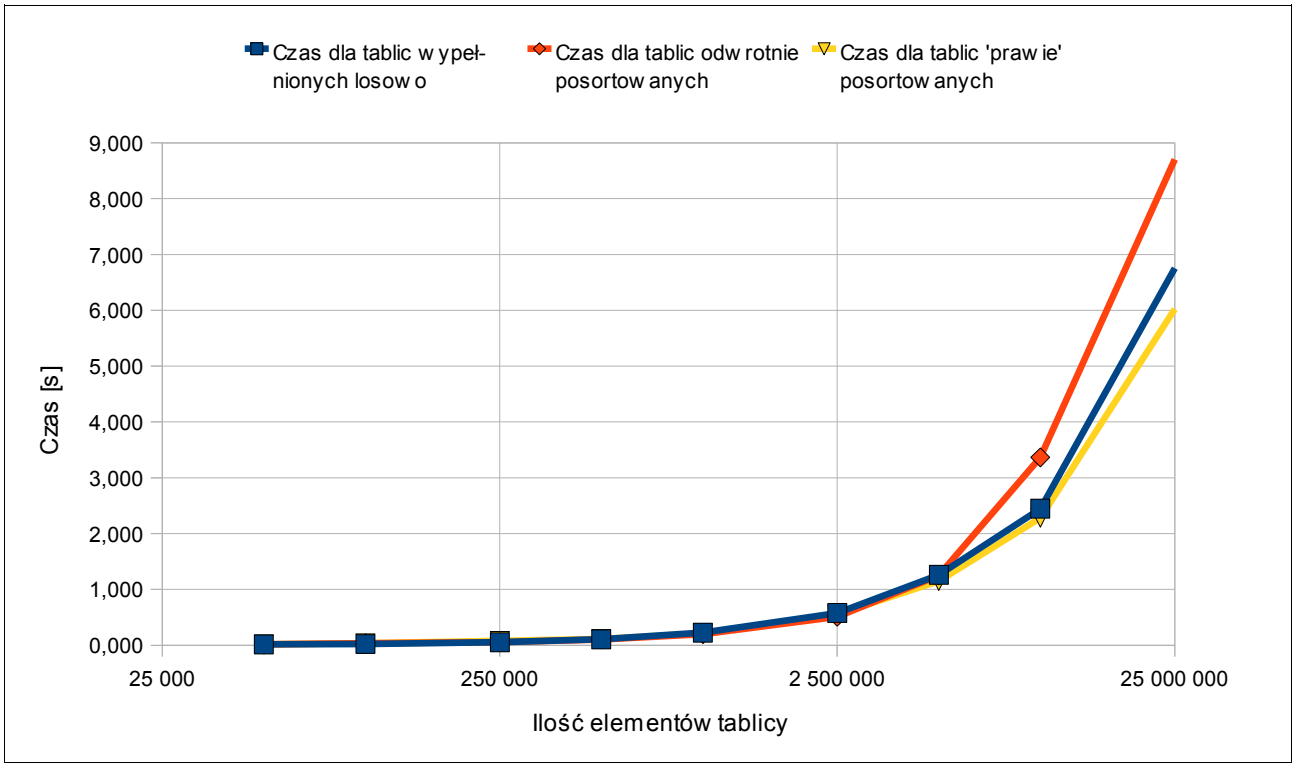
- element P jest pierwszym od lewej elementem tablicy
- element P jest środkowym elementem tablicy

5.3 Wyniki pomiarów:

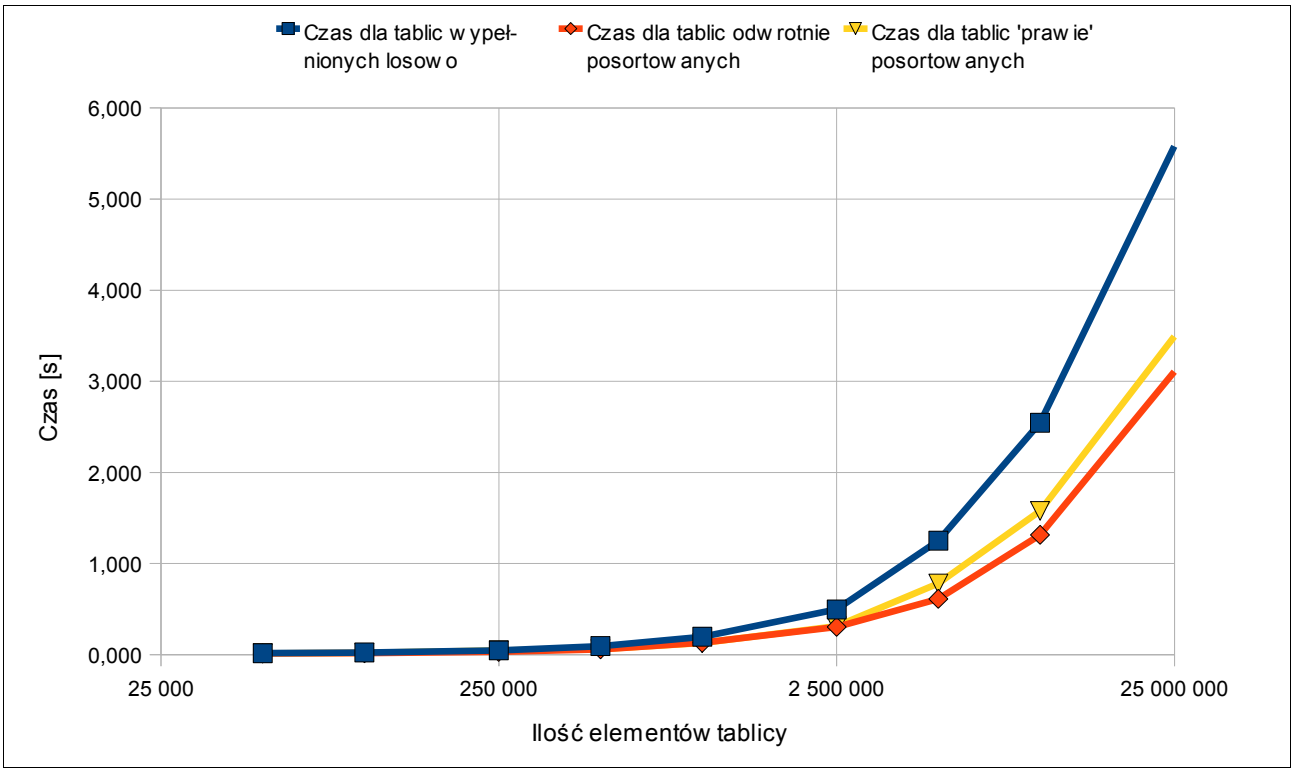
Wyniki dla P, które jest lewym, najbardziej skrajnym elementem tablicy:

Ilość elementów tablicy [tys]		**50**	**100**	**250**	**500**	**1000**	**2500**	**5000**	**10000**	**25000**
Czas dla tablicy wypełnionej losowo	**[s]**	0,015	0,021	0,053	0,105	0,226	0,578	1,261	2,445	6,753

Ilość elementów tablicy [tys]		50	100	250	500	1000	2500	5000	10000	25000
Czas dla tablicy odwrotnie posortowanej	[s]	0,016	0,034	0,054	0,103	0,2	0,508	1,251	3,364	8,706
Czas dla tablicy 'prawie' posortowanej	[s]	0,016	0,034	0,078	0,112	0,210	0,550	1,149	2,283	6,025



oraz P, które jest elementem środkowym tablicy:



Ilość elementów tablicy [tys]		50	100	250	500	1000	2500	5000	10000	25000
Czas dla tablicy wypełnionej losowo	**[s]**	0,016	0,024	0,047	0,095	0,197	0,497	1,251	2,545	5,578
Czas dla tablicy odwrotnie posortowanej	**[s]**	0,014	0,016	0,032	0,059	0,13	0,305	0,613	1,315	3,106
Czas dla tablicy 'prawie' posortowanej	**[s]**	0,015	0,020	0,048	0,060	0,127	0,312	0,781	1,578	3,492

5.4 Analiza wyników:

Quicksort jest jednym z lepszych algorytmów sortowania. Pod względem szybkości przewyższa wszystkie pozostałe algorytmy sortowania, których opisy znajdują się w niniejszej pracy (oprócz kubełkowego). Istotny skok wartości czasu potrzebnego na posortowanie elementów jest zauważalny w przypadku sortowania 2mln elementów. Losowe wypełnienie tablicy wydłuża czas działania algorytmu.

Zalety:

- szybkość działania oraz prostota implementacji,
- wydajnie sortuje duże ilości danych.
- powszechność implementacji (np. qsort w języku C czy PHP).

Wady:

- efektywność działania algorytmu zależy od wybrania wartości osiowej i jest trudna do przewidzenia
- problemy z sortowaniem tablic odwrotnie posortowanych w przypadku gdy P jest skrajnym elementem tablicy,
- głęboka rekurencja w przypadku dużej ilości danych może spowodować niestabilność programu

5.5 Złożoność

Algorytm Quicksort charakteryzuje się złożonością **O(n log n)**

6. Sortowanie kubełkowe

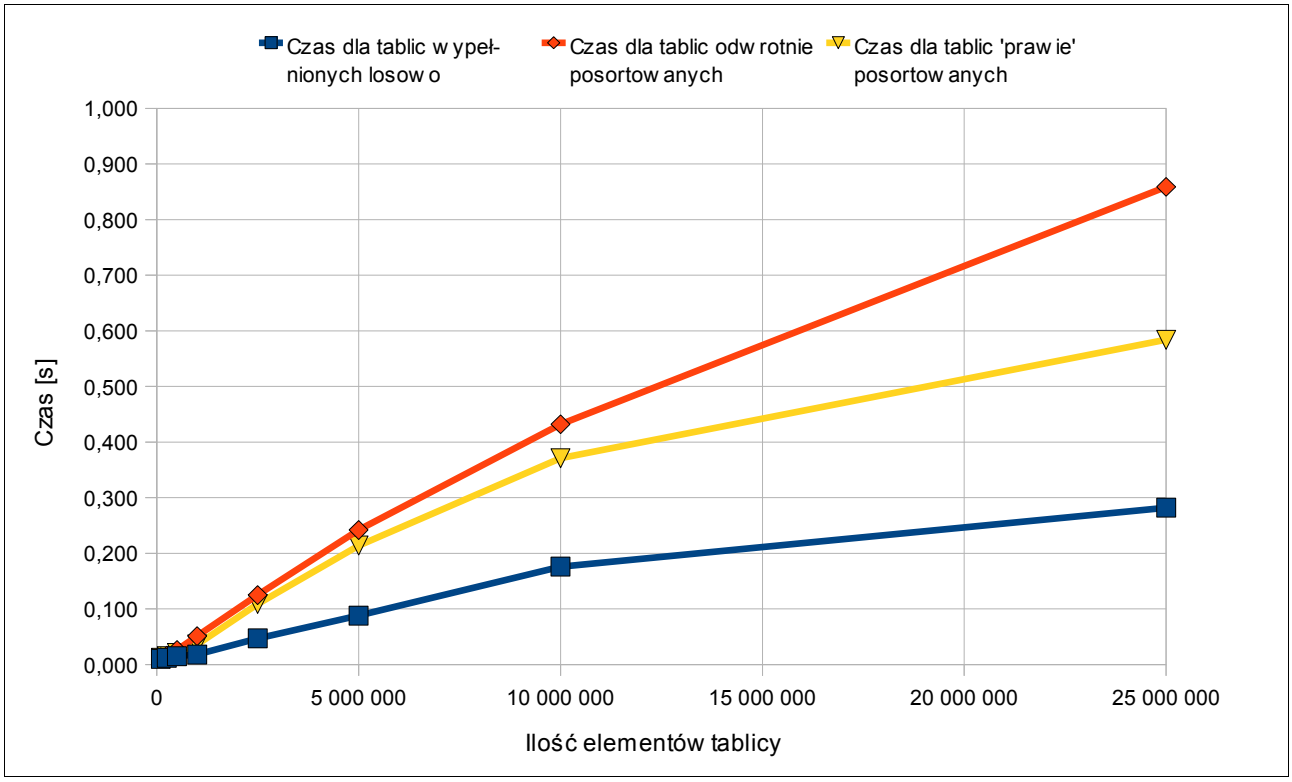
Sortowanie Shella jest rozszerzeniem sortowania przez wstawianie. Proces sortowania jest znacznie przyspieszony dzięki możliwości wymiany odległych od siebie elementów.

6.1 Opis algorytmu:

Na samym początku działania algorytmu dzielimy przedział liczb na n podzbiorów o równej długości. Liczby z sortowanej tablicy przyporządkowujemy do odpowiedniej tablicy. Następnym krokiem jest posortowanie elementów w niepustych kubełkach. Zazwyczaj przyjmuje się, że sortowane liczby należą do przedziału od 0 do 1, w innym przypadku sortowane liczby należy wyskalować. Podstawowym problemem przy sortowaniu metodą kubełkową jest wybór wartości n reprezentującej równe podzbiory. Nie możemy ustalić większej ilości podzbiorów niż wynosi liczba elementów tablicy.

6.2 Wyniki pomiarów:

Ilość elementów tablicy [tys]		100	250	500	1000	2500	5000	10000	25000
Czas dla tablicy wypełnionej losowo	[s]	0,010	0,012	0,015	0,018	0,047	0,088	0,176	0,282
Czas dla tablicy odwrotnie posortowanej	[s]	0,013	0,016	0,026	0,05	0,125	0,242	0,432	0,859
Czas dla tablicy 'prawie' posortowanej	[s]	0,012	0,015	0,021	0,036	0,109	0,214	0,371	0,584



6.3 Analiza wyników:

Algorytm sortowania bąbelkowego okazał się najszybszy z testowanych algorytmów. Jest on zdecydowanie najszybszy jeżeli chodzi o sortowanie dużej ilości elementów, przy małej ilości się nie sprawdza i pochłania duże obszary pamięci. Jeszcze lepszą efektywność można osiągnąć sortując dane, które są równomiernie rozłożone w przedziale.

Zalety:

- algorytm wyjątkowo stabilny,
- algorytm pracuje w liniowym czasie – ułatwia to sortowanie dużych liczb,
- stały czas na sortowanie elementu, czas nie zwiększa się diametralnie wraz ze wzrostem ilości elementów.

Wady:

- duże wymagania pamięciowe,
- nieefektywny dla małej ilości danych.

6.4 Złożoność

W przypadku gdy liczby w przedziale są rozłożone jednostajnie: **O(n)**, w ogólnym przypadku **O(n^2)**

IV. Wnioski

Kazdy algorytm sortowania ma swoje wady i zalety, które zostały przedstawione podczas omawiania poszczególnych algorytmów. Wybór metody sortowania zależy od potrzeb programistów i ogólnych specyfikacji problemu, takich jak ilość danych, czy sposób ich ułożenia w tablicy.

V. Bibliografia

1. Cormen T. H., Leiserson Ch. E., Rivest R.L. *Wprowadzenie do algorytmów.* Wydawnictwo Naukowo-Techniczne, 2001
2. Wróblewski P., Algorytmy, struktury danych i techniki programowania. Helion, 2010
3. http://wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych
4. http://pl.wikipedia.org/wiki/Kategoria:Algorytmy_sortowania