SPRAWOZDANIE

PROJEKT 1

PROJEKTOWANIE ALGORYTMÓW I METOD SZTUCZNEJ INTELIGENCJI

AUTOR: EMILIA SZYMAŃSKA

PROWADZĄCY: DR INŻ. KRZYSZTOF HALAWA

GRUPA: WTOREK 13:15 – 15:00

1. Wstęp

Celem projektu jest implementacja wybranych algorytmów sortowania, następnie ich przetestowanie i zbadanie ich efektywności. Wybrane przeze mnie algorytmy sortowania to sortowanie przez scalanie, sortowanie szybkie oraz sortowanie introspektywne. Do przeprowadzenia testów efektywności wygenerowałam tablice zawierające dane typu całkowitoliczbowego w zakresie od -1 000 000 do 1 000 000. Tablic jest 100 każdego rodzaju z każdego rozmiaru. Rozmiary są następujące: 10 000, 50 000, 100 000, 500 000 i 1 000 000. Z rodzajów wyróżniamy: całkowicie losowe wartości, posortowane w 25%, 50%, 75%, 99%, 99,7% oraz posortowane w odwrotnej kolejności.

2. Omówienie algorytmów

2.1. Sortowanie przez scalanie

Ten typ sortowania opiera się na zasadzie ,,dziel i zwyciężaj" oraz fakcie, że pojedynczy element sam w sobie jest już posortowany, dlatego rekurencyjnie dzielimy nasz ciąg na połowy do momentu, aż pozostanie jeden element. Wówczas otrzymane podciągi łączymy w posortowany ciąg.
Złożoność algorytmu w przypadku zarówno średnim, jak i pesymistycznym wynosi O(n · log n). Wynika to z tego, że głębokość wywołań funkcji dla tablicy o n elementach wynosi w przybliżeniu $\log_2 n$, a złożoność samego scalania jest liniowa.

2.2. Sortowanie szybkie

Algorytm szybkiego sortowania zaczyna się od wyboru tzw. piwota (może to być dowolny element, ja w swoich algorytmach brałam środkowy element). Następnie dzielimy ciąg na dwa podciągi, przy czym w lewym podciągu są elementy nie większe niż piwot, a w prawym większe. Piwot ustawiamy na miejscu między ciągami, jest on już na swojej ostatecznej pozycji. Algorytm wywołujemy rekurencyjnie zarówno dla lewego, jak i prawego ciągu do momentu, aż długość podciągu jest równa jeden. Po tym procesie mamy już posortowany ciąg.
Złożoność operacji przenoszenia elementów jest liniowa, a o złożoności samego algorytmu decyduje to, ile będzie rekurencyjnych wywołań funkcji sortującej. W średnim przypadku złożoność algorytmu wynosi O($n \cdot log\ n$). W pesymistycznym przypadku, gdy za piwota będziemy pechowo obierać wartość najmniejszą lub największą, przy każdym kolejnym wywołaniu liczba posortowania jest tylko o 1 mniejsza od poprzedniej, więc funkcję będziemy wywoływać n razy, a samo sortowanie będzie przebiegać ze złożonością O(n^2).

2.3. Sortowanie introspektywne

Sortowanie introspektywne łączy w sobie algorytmy sortowania przez kopcowanie, sortowania szybkiego oraz sortowania przez wstawianie. Celem takiej hybrydy jest wyeliminowanie dużej złożoności obliczeniowej sortowania szybkiego w najgorszym przypadku. Na podstawie maksymalnej dozwolonej głębokości wywołań rekurencyjnych odwołujemy się do odpowiedniego algorytmu. Mimo że złożoność np. sortowania przez wstawianie jest rzędu O(n^2), to dla małych ciągów działa ono relatywnie szybko, przy czym rekurencja sortowania szybkiego jest stosunkowo czasochłonna i pochłania miejsce na stosie.

W przypadku zarówno średnim, jak i najgorszym algorytm sortowania introspektywnego ma złożoność O(n · log n). Wytłumaczenie jest zawarte w opisie złożoności sortowania szybkiego z tą tylko różnicą, że poprawiamy przypadek o złożoności kwadratowej poprzez odwołanie do sortowania przez kopcowanie (który ma również złożoność O(n · log n)), gdy głębokość rekurencji jest zbyt duża.

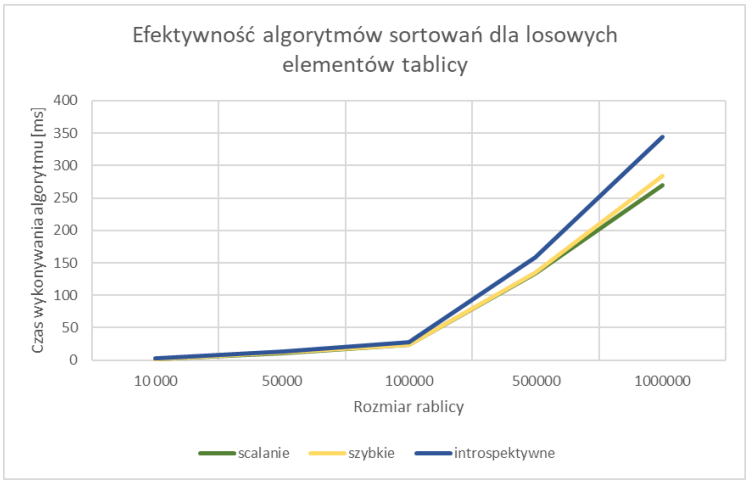
3. Wyniki eksperymentu

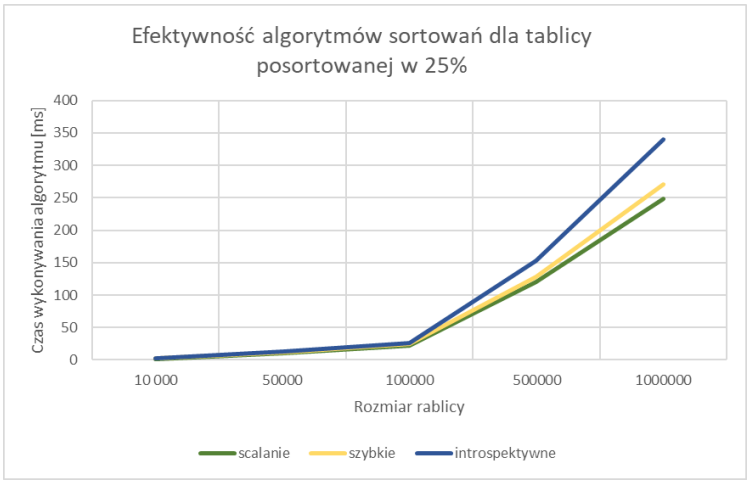
Wyniki wywołań odpowiednich funkcji sortujących dla danych tablic opisanych w punkcie pierwszym prezentuję w postaci tabel oraz wykresów. Tabele zawierają uśrednione wyniki czasu działania algorytmów w milisekundach dla konkretnych przypadków (kolumny odpowiadają rozmiarom tablic, a wiersze ich rodzajom). Wykresy natomiast przedstawiają porównania pomiędzy trzema algorytmami przy tym samym rodzaju tablic, a także dla każdego typu sortowania porównania między czasem dla różnych rodzajów tablic.

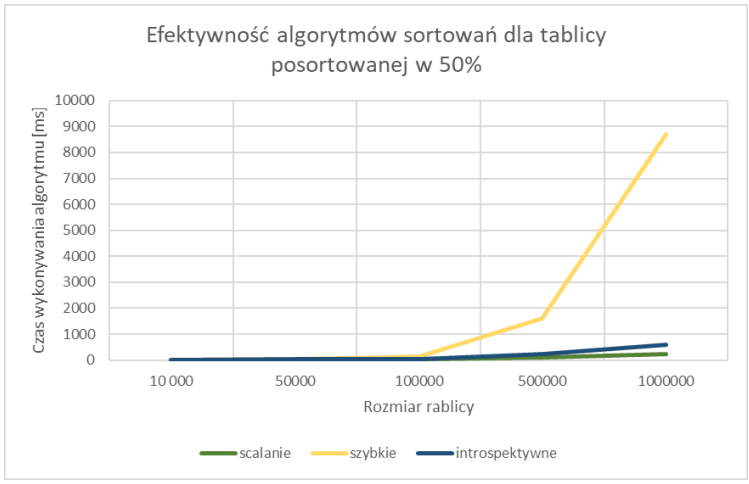
SORTOWANIE PRZEZ SCALANIE					
	10 000	50 000	100 000	500 000	1 000 000
losowe	2,05951	10,70601	23,6635	133,461	269,163
posortowane w 25%	1,80266	10,70646	22,02012	120,6672	248,8995
posortowane w 50%	1,69173	9,37658	19,77012	112,1359	218,976
posortowane w 75%	1,48249	7,98895	17,10424	85,83457	194,753
posortowane w 95%	1,38001	7,11977	15,52725	80,74274	172,442
posortowane w 99%	1,2972	7,16269	15,00529	81,91946	182,6955
posortowane w 99,7%	1,28509	7,01359	14,57783	86,50872	190,399
posortowane odwrotnie	1,26889	6,92338	15,02062	93,6741	168,157

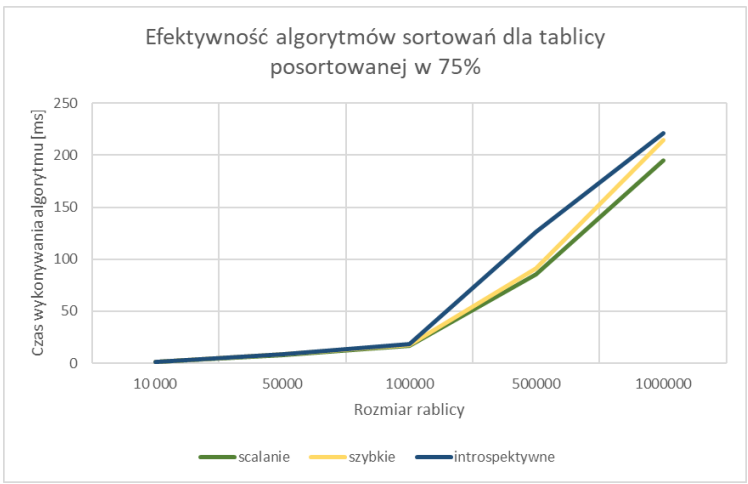
SORTOWANIE SZYBKIE					
	10 000	50 000	100 000	500 000	1 000 000
losowe	2,09515	11,73256	23,66977	134,8697	284,1075
posortowane w 25%	2,06533	11,39859	24,5766	128,3536	271,1315
posortowane w 50%	5,98456	51,24172	133,079	1604,773	8685,364
posortowane w 75%	1,68266	8,45045	17,68324	94,40125	214,33
posortowane w 95%	1,29849	6,95815	14,56473	84,91125	173,521
posortowane w 99%	1,21886	7,38302	14,47149	82,58602	170,8435
posortowane w 99,7%	1,20469	6,57976	13,85467	80,51018	177,2745
posortowane odwrotnie	1,23659	6,40115	13,35954	78,16982	158,3725

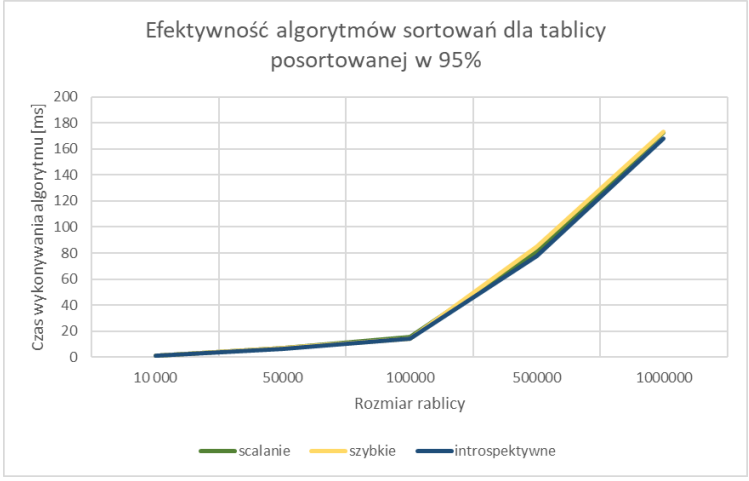
SORTOWANIE INTROSPEKTYWNE					
	10 000	50 000	100 000	500 000	1 000 000
losowe	2,35499	12,81142	27,44169	158,4499	344,061
posortowane w 25%	2,30593	12,96232	26,52724	152,6931	339,497
posortowane w 50%	3,5167	19,7171	42,24975	240,4804	597,2095
posortowane w 75%	1,57726	8,66633	18,05971	126,1032	221,5815
posortowane w 95%	1,24377	6,45076	13,99794	77,82894	167,7855
posortowane w 99%	1,13818	6,19934	13,1455	74,36076	162,333
posortowane w 99,7%	1,07332	6,15669	12,9962	73,84062	161,5285
posortowane odwrotnie	1,05894	6,19478	13,04618	73,03127	159,5995

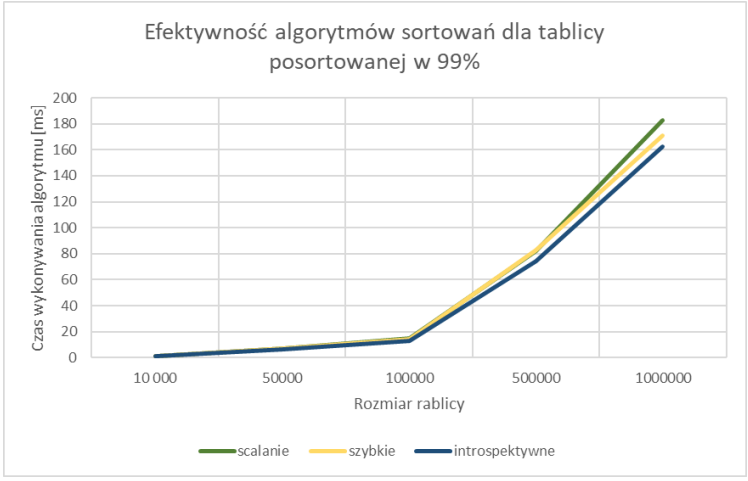

Efektywność algorytmów sortowań dla losowych elementów tablicy
Czas wykonywania algorytmu [ms]
0
50
100
150
200
250
300
350
400
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne

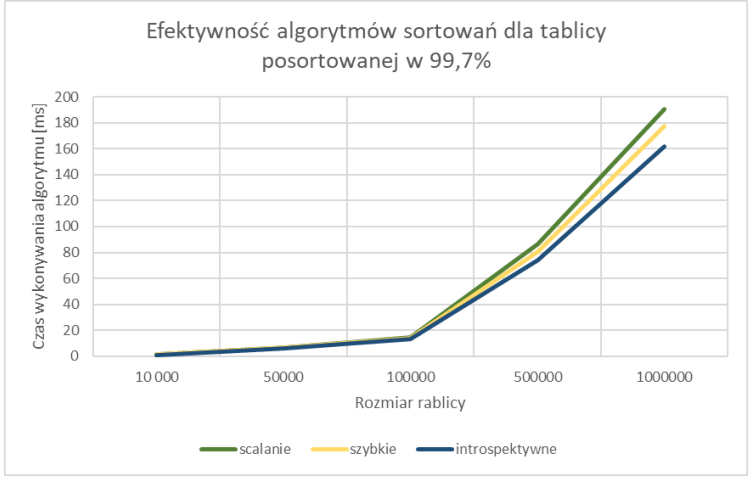

Efektywność algorytmów sortowań dla tablicy posortowanej w 25%
Czas wykonywania algorytmu [ms]
0
50
100
150
200
250
300
350
400
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne

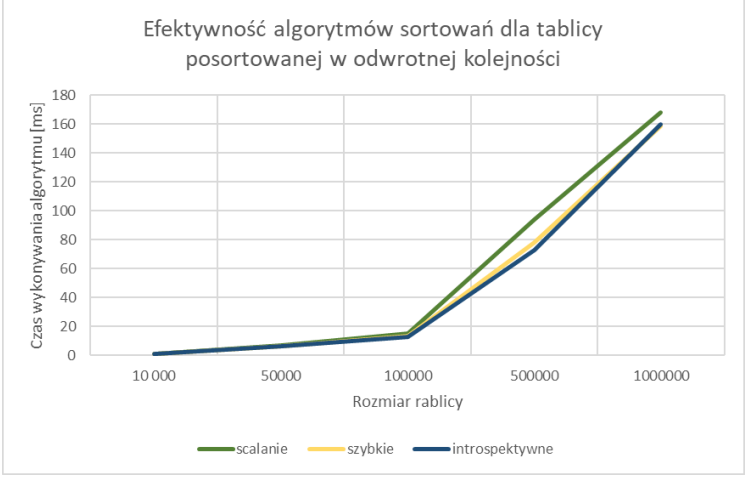

Efektywność algorytmów sortowań dla tablicy posortowanej w 50%
Czas wykonywania algorytmu [ms]
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne

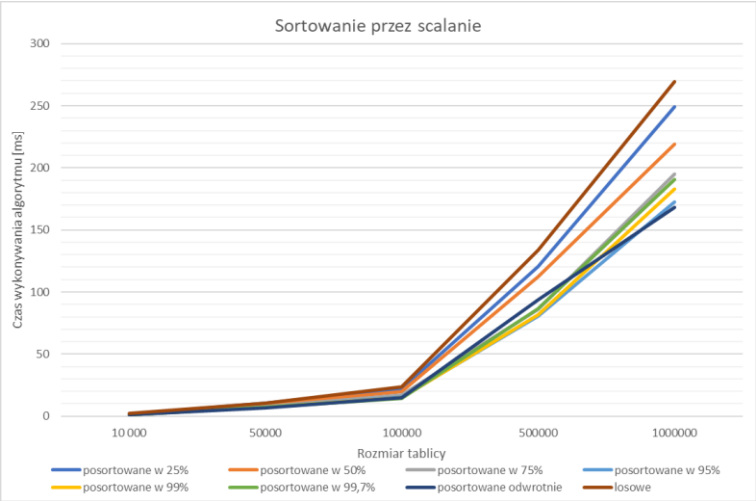

Efektywność algorytmów sortowań dla tablicy posortowanej w 75%
Czas wykonywania algorytmu [ms]
0
50
100
150
200
250
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne

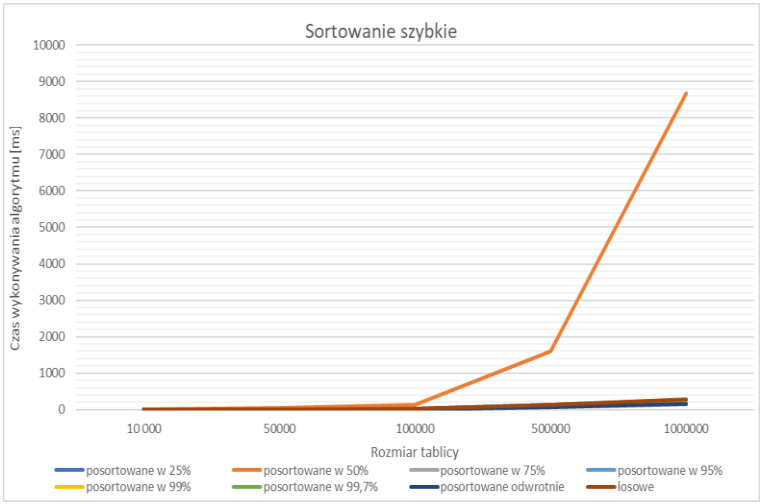

Efektywność algorytmów sortowań dla tablicy posortowanej w 95%
Czas wykonywania algorytmu [ms]
0
20
40
60
80
100
120
140
160
180
200
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne

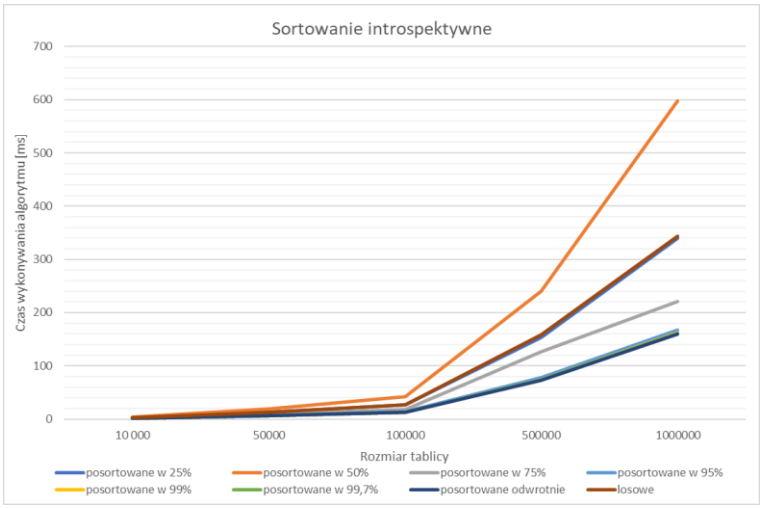

Efektywność algorytmów sortowań dla tablicy posortowanej w 99%
Czas wykonywania algorytmu [ms]
0
20
40
60
80
100
120
140
160
180
200
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne


Efektywność algorytmów sortowań dla tablicy posortowanej w 99,7%
Czas wykonywania algorytmu [ms]
0
20
40
60
80
100
120
140
160
180
200
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne


Efektywność algorytmów sortowań dla tablicy posortowanej w odwrotnej kolejności
Czas wykonywania algorytmu [ms]
0
20
40
60
80
100
120
140
160
180
10 000
50000
100000
500000
1000000
Rozmiar rablicy
scalanie
szybkie
introspektywne







4. Wnioski

Zgodnie z oczekiwaniami – jako że trzy rozpatrywane algorytmy mają podobną złożoność obliczeniową – przy porównaniach między samymi algorytmami nie ma dużej różnicy poza przypadkiem tablic posortowanych w 50%. W tym bowiem przypadku wyszła na jaw kwadratowa złożoność obliczeniowa sortowania szybkiego – jako że lewa część tablicy jest już posortowana, to pośrodku znajduje się największy element z lewej części tablicy, a ja ten element obieram za piwota i dalej wykonuję procedurę sortowania. Zazwyczaj przy wzroście procentowego posortowania tablicy wejściowej czas wykonywania algorytmu maleje, gdyż musimy wykonać mniej zamian elementów, skoro już są na dobrych pozycjach. W przypadku sortowania introspektywnego przy tablicy posortowanej w 50% mamy znaczący wzrost, jednak nie jest on złożonością kwadratową jak w przypadku sortowania szybkiego – wynika to z faktu, że sortowanie szybkie jest częściowo wykorzystywane przez algorytm, jednak od pewnej wartości głębokości wywołań rekurencji porzucamy procedurę szybką, a wybieramy kopcowanie.

5. Opis kodu

5.1. Sortowanie przez scalanie (katalog merge_sort)

Na kod składają się pliki merge_sort.cpp, merge_sort.hh oraz main.cpp. W pliku main.cpp wczytuję tablicę, wywołuję funkcję sortowania, a następnie wypisuję tablicę. W pliku .hh mam nagłówek funkcji sortującej, a w pliku merge_sort.cpp jej ciało. To, co wykonuję, to wybranie środkowego indeksu podziałowego (index_middle), a następnie wywoływanie rekurencyjnie funkcji sortowania dla lewego i prawego podciągu. Następnie, dopóki nie skończy mi się tablica,

scalam podciągi przy pomocy dynamicznie zaalokowanej dodatkowej tablicy, a potem przepisuję wartości z tablicy tymczasowej do tablicy ostatecznej, na którą zwracam wskaźnik.

5.2. Sortowanie szybkie (katalog quick_sort)

Na kod składają się pliki quick_sort.cpp (ciało funkcji sortującej), quick_sort.hh (nagłówek funkcji sortującej) oraz main.cpp (gdzie wczytuję tablicę, wywołuję działanie funkcji i wypisuję posortowaną tablicę). W ciele funkcji wybieram za piwota element środkowy, przy pomocy dodatkowo wydzielonej tablicy dzielę tablicę początkową na dwa podciągi (lewy z wartościami nie większymi, a prawy z wartościami większymi niż piwot). Na koniec wywołuję rekurencyjnie funkcję sortującą dla lewego i prawego podciągu, ale nie ruszając już z miejsca piwota, który znajduje się na swojej ostatecznej pozycji w posortowanym ciągu.

5.3. Sortowanie introspektywne (katalog introspective_sort)

W tym katalogu znajdują się pliki heap.cpp (ciała metod struktury kopca oraz funkcja zamiany elementów), heap.hh (struktura kopca z kontruktorem, dekonstruktorem i krótkimi, nieskomplikowanymi metodami), introspective_sort.hh (z nagłówkami funkcji sortujących), introspective_sort.cpp (ciała funkcji potrzebnych do wykonania sortowania) oraz main.cpp (wywołanie sortowania dla wczytanej tablicy i wypisanie jej po posortowaniu). W heap.cpp najważniejsze są funkcje usuwania i dodawania elementu do kopca. Przy usuwaniu zamieniam element na górze kopca z ostatnim, usuwam ostatni (który był wcześniej pierwszym) i odpowiednio umieszczam element z korzenia kopca, by spełniał zależności charakterystyczne dla tej struktury (,,ojciec" może mieć maksymalnie dwóch synów, ale mniejszych pod względem wartości względem niego). Przy dodawaniu wrzucam element na ostatnie miejsce, a następnie przemieszczam go tak, by spełniał założenia kopca. Przy sortowaniu przez kopcowanie korzystam z faktu, że w korzeniu kopca jest największy element, więc wczytaną tablicę układam w kopiec, a następnie zabieram z kopca elementy i w odwrotnej kolejności wpisuję do tablicy. Do sortowania introspektywnego potrzeba także funkcji wziętej z sortowania szybkiego, która dzieli na podciągi lewy i prawy (tak jak to jest opisane w podpunkcie wyżej), a także algorytmu sortowania przez wstawianie (podobnego do sortowania, jakie gracze wykonują podczas gry w karty, ustawiając je na ręce w odpowiedniej kolejności). W funkcji sortującej w zależności od głębokości wywołań rekurencji wykonujemy odpowiednio sortowanie przez kopcowanie lub funkcję wziętą z sortowania szybkiego. Na koniec pomocniczo sortujemy algorytmem sortowania przez wstawianie.

6. Literatura: https://eduinf.waw.pl/inf/alg/003_sort/index.php
   https://pl.wikipedia.org/wiki/Sortowanie
   ,,Wprowadzenie do algorytmów" - T. Cormen, C. Leiserson, R. Rivest, C. Stein