POLITECHNIKA WROCŁAWSKA Wydział Informatyki i Telekomunikacji	
Algorytmy i złożoność obliczeniowa	
Rok akademicki:	2024/2025
Autor projektu:	Seweryn Pastuch
Nr indeksu:	280897
Temat projektu:	Badanie efektywności algorytmów sortowania

Oświadczenie: Przekazując to sprawozdanie do oceny prowadzącemu zajęcia autor oświadcza, że zostało ono przygotowane samodzielnie, bez udziału osób trzecich oraz że żadna jego część nie jest plagiatem.

1. Wstęp

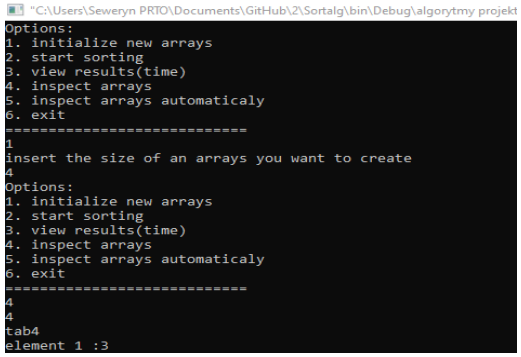
Projekt „Badanie efektywności algorytmów sortowania" ma na celu porównanie wydajności pięciu wariantów algorytmów sortowania poprzez pomiar czasu ich działania na losowych zbiorach danych. W celu realizacji projektu stworzono program w języku C++, który umożliwia automatyczne wykonanie pomiarów bez konieczności wprowadzania dodatkowych ustawień przez użytkownika.

2. Streszczenie projektu

Program wielokrotnie przetwarza losowo wygenerowane dane za pomocą generatora liczb losowych Mersenne Twister (mt19937), którego ziarno pochodzi z czasu systemowego. Dla każdej próby sortowana jest tablica liczb całkowitych, a czas operacji jest mierzony przy pomocy zegara (std::chrono::high_resolution_clock).
Jako iż okazuje się, że wielokrotnie sortowanie dużych zbiorów liczb nie jest takie szybkie dla komputera, jak wcześniej mogło się wydawać, wykonanie dostatecznie imponujących pomiarów wymaga zwykle kilkunastu godzin pracy komputera.

3. Wstęp teoretyczny

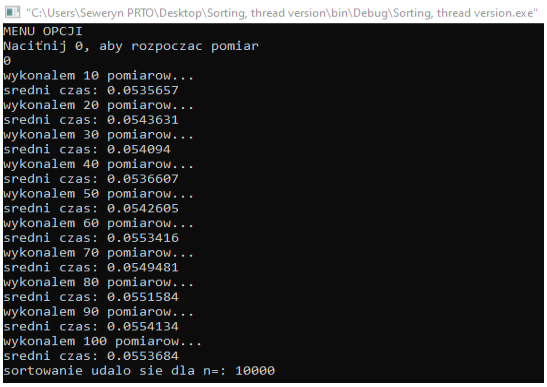
Od samego początku, ze strony technicznej pracę nad projektem zacząć trzeba było od opracowania optymalnych algorytmów sortujących tablice liczb całkowitych, w tym celu pierwsze kilka dni pracy poświęciłem na tworzeniu oprogramowania sortującego oraz wyświetlającego tablice liczb całkowitych. Program nie zostanie tutaj przedstawiony z racji na to, że chce osiągnąć nie dużą nadmiarowość kodu i treści w sprawozdaniu,oraz dlatego, że był on potrzebny tylko na samym początku.
Program wspomagający prace nad kolejnymi algorytmami:



Do generowania danych używałem do tamtej pory funkcji rand(), która losuje maksymalnie z przedziału 0 - 32 767. Była ona jak dotąd jedyną funkcją losującą liczby całkowite którą znałem, a do tego programu testowego była ona wystarczająca. Jak już umiałem efektywnie sprawdzać poprawność sortowań, zacząłem pracę nad 5 algorytmami

sortowania. Gdy już ta część pracy została ukończona po konsultacjach projektowych zabrałem się za tworzenie programu oraz jego właściwej pętli, powtarzającą sekwencję generuj_dane-sortuj-zapisz_pomiar porządaną ,zdefiniowaną w kodzie liczbę razy. Jednocześnie uznałem, że końcowy program nie musi wymagać nie wiadomo jakiego menu opcji, ponieważ spowodowała by ona dość dużą nadmiarowość kodu którego i tak jest dość sporo, bowiem program wspomagający tworzenie algorytmów ma około 250 linijek, a właściwy wyjściowy kod projektu wykonyjący sortowania i pomiary automatycznie ma 220 linii kodu. W projekcie zrezygnowłem również z użycia kontenerów std::vector na rzecz dynamicznie alokowanych tablic typu int.

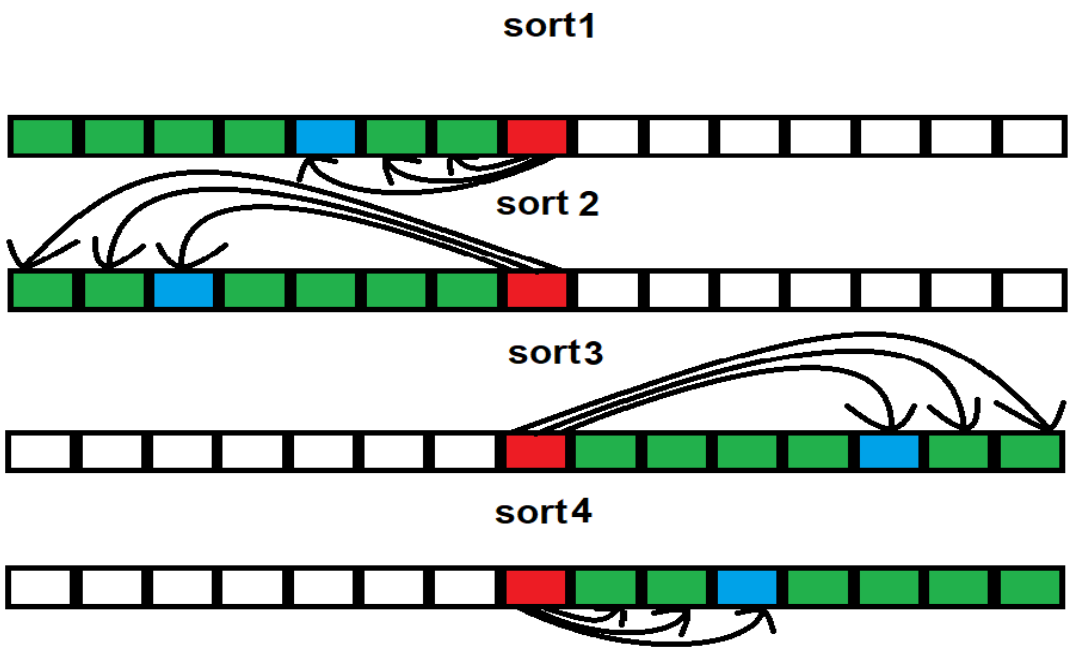
Program wykonujący zadanie opisane w strzeszczeniu:



4. Opis działania algorytmów

Program implementuje pięć wariantów algorytmu sortowania poprzez wstawianie liniowe. Algorytm ten działa podobnie do sposobu, w jaki ludzie zwykle układają karty w ręku - jeden element na raz jest pobierany z nieposortowanej części i wstawiany w odpowiednie miejsce w posortowanej już części kart. Różnica między takim zobrazowaniem a implementacją w języku programowania jest taka, że w implementacji w kodzie musimy z reguły mieć mechanizm przesuwania tych kart aby zrobić lukę na wstawienie naszego elementu któremu poprzednio znaleźliśmy miejsce.

- **Sort1**: Klasyczny insertion sort.
- **Sort2**: Wstawianie elementu na odpowiednią pozycję poprzez liniowe wyszukiwanie miejsca od początku.
- **Sort3**: Odmiana insertion sortu z odwrotnym kierunkiem przeglądania - od końca.
- **Sort4**: Odmiana sortowania podobna do Sort1, ale iterująca w odwrotną stronę tablicy.



Wyżej obraz kierunków wyszukiwania w posortowanym ciągu powyższych algorytmów

- **Sort5**: Wersja insertion sortu, w której miejsce wstawienia elementu wyszukiwane jest za pomocą wyszukiwania binarnego. Jest to modyfikacja klasycznego sortowania przez wstawianie, w której liniowe przeszukiwanie posortowanej części zastępujemy wyszukiwaniem binarnym. Dzięki temu zmniejszamy liczbę porównań potrzebnych do znalezienia właściwej pozycji wstawienia co prowadzi do dużo lepszych wyników sortowania przez wstawianie binarne.

5. Plan eksperymentu

Eksperyment 1:

Założenia dotyczące danych:

- Dane generowane są losowo w przedziale liczb całkowitych od 1 do 1 000 000.
- Rozmiary tablic: od 10 000 do 610 000 elementów, ze skokiem co 100 000 elementów.

Sposób pomiaru:

- Czas sortowania mierzony jest dla każdego algorytmu osobno, w mikrosekundach (przeliczonych na sekundy).

Zapisywanie wyników:

- Podczas pracy programu sumy czasów sortowań dla rozmiarów tablic są zapisywane w zmiennej tablicowej 2D double results[5][190]
- Średni czas wykonania dla 100 prób dla każdego rozmiaru tablicy(suma z komórki tablicy podzielona/100) jest zapisywany do pliku tekstowego w postaci .txt.

Własne funkcje pomocnicze:

- void randomize(int b) – generowanie danych losowych
- void ini(int b) – inicjalizowanie pamięci
- void cpy(int z) – kopiowanie tablic
- void deletemem() – zwalnianie pamięci po każdej próbie.
- void test2(bool kol=0) – generowanie danych posortowanych

Eksperyment 2:

Drugi eksperyment dot. pomiaru efektywności algorytmów sortowania będzie polegał na posortowaniu tablicy posortowanej oraz tablicy odwrotnie posortowanej 100 k elementów badanymi algorytmami. Taki test może nam dać szersze pojęcie o specyfice naszych algorytmów

Zapamiętanie danych:

Dane z pomiarów sumowane są w tablicy dwuwymiarowej[tab.1], z której następnie wyciągana jest średnia czasu sortowania oraz zapisywana w pliku tekstowym [tab.2].

Tab.1-format zapisu wyników w pamięci programu tabela 5x(liczba_zmierzonych_rozmiarów_tablic):

time(sort1)	time(sort2)	time(sort3)	time(sort4)	time(sort5)	array_size
0,0379679	0,0549638	0,0565902	0,0399049	0,0275337	10000
5,37046	7,37197	7,43851	5,54859	3,84069	110000
19,6387	26,668	26,772	20,0315	13,8196	210000
42,79	58,157	58,4228	43,6539	30,1794	310000

Tab.2-format zapisu danych wynikowych w wynikowym pliku tekstowym:

```
0.0379679;0.0549638;0.0565902;0.0399049;0.0275337;
5.37046;7.37197;7.43851;5.54859;3.84069;
19.6387;26.668;26.772;20.0315;13.8196;
42.79;58.157;58.4228;43.6539;30.1794;
74.5457;101.516;101.861;76.1279;52.4445;
```

6.0 Kod programu-Uwagi

Biorąc pod uwagę fakt, że przez cały semestr aktywnie uczestniczyłem w konsultacjach projektowych, tłumaczenie logiki programu pozostawię ogólnie w punktach odnoszących się do zasad zapisu danych, inicjowania tablic itd. oraz w komentarzach samego kodu. Kod komentowałem od początku w języku angielskim z racji na fakt, iż moje projekty publikuję na githubie, nie w nadziei, że ktoś z nich kiedyś będzie korzystał, lecz raczej traktuję to jako moje portfolio do rozmów rekrutacyjnych.

Uwagą odnośnie kodu jest jeszcze dodanie przeze mnie większej ilośći instrukcji std::cout z racji na to, że teraz program jest gotowy do uruchomienia po skompilowaniu poniższego kodu. Używanie długich instrukcji „cout<<...<<...<<...<<...<<...<<...<<..<<..;” po skopiowaniu kodu i wklejeniu go do edytora tekstu porowadzi do rozdzielenia tej intrukcji losowo na parę linijek co prowadzi do błędów kompilacji.

6.1 Kod programu

```
#include<iostream>
#include<random>
#include<ctime>
#include<chrono>
#include<thread>
#include<fstream>
using namespace std;
int a; /*globally initialized array size variable, used in sorting algorithms*/
int czas=time(0); /*system time variable used as a seed for the random number generator, to prevent some special cases, it is incremented each time when the new arrays are about to be generated*/
int* tab1; /* pointers to arrays that will be initialized in void ini() func*/
int* tab2;
int* tab3;
int* tab4;
int* tab5;
int* tab0; /* an array, we will never sort, it is the base array that is later filled with data with randomization process in void randomize() func*/
double results[5][190]; /*fixed size array for results of time each algorithm had to inflict in order to sort the array*/
void ini(int b){/*allocates memory dynamically, asigns pointers to memory*/
tab0=new int[b];
tab1=new int[b];
tab2=new int[b];
tab3=new int[b];
tab4=new int[b];
tab5=new int[b];
}
void cpy(int z){ /*function that copies tab0 array values to the arrays we will be sorting in the main
```

```
function of the program, takes the current array size as the only argument*/
memcpy(tab1, tab0,(4*z));/* the arguments of the memcpy are as shown in this example:
memcpy(where to copy,from where take the data,number of bytes)*/
memcpy(tab2, tab0,(4*z));/* on my computer the int size is 4 bytes so we basically move
4*array_size bytes from the source to the destination.*/
memcpy(tab3, tab0,(4*z));
memcpy(tab4, tab0,(4*z));
memcpy(tab5, tab0,(4*z));
}
void randomize(int b){ /* function that randomizes five tabs of int with values from range 1 to 1
000 000. the function takes a basic argument of current array size*/
ini(b); /*initializing the memory with the size of current array size*/
int i=0;
mt19937 gen(czas); // setting the seed for the random number generator*/
uniform_int_distribution<int> dist(1, 1000000); /*we set the random number generator settings
so in the end we will be getting int numbers from range 1-1 000 000*/
while(i<b){
tab0[i]= dist(gen); /*filling every tab0 array index memory with a randomly generated values*/
i++;
}
cpy(b); /* in the end we copy tab0 to every other array*/
}
void test2(bool kol=0){ /* function that initializes the memory with a fixed size array of 100k
elements.*/
a=100000;/*later it fills the tab0 values with ascending or descending ints depending on the bool
argument given from user*/
ini(a); /* the function is needed to perform an second experiment described in the paperwork i
made*/
if(kol==0){
for(int i=0; i<a; i++){/* sorted array*/
tab0[i]= i;
}
}
else{
int m=99999;
for(int i=0; i<a; i++){/*descending sorted array*/
tab0[i]= m;
m--;
}
}
cpy(a);
}
void deletemem(){/* function that deletes alocated arrays after the sorting algorithms complete*/
delete[] tab0;
delete[] tab1;
delete[] tab2;
delete[] tab3;
delete[] tab4;
delete[] tab5;
}
void sort1(int p){/*basic insert sorting alg described in the paper*/
int temp;
```

```
auto start=std::chrono::high_resolution_clock::now();/* here we use a high resolution clock from chrono libray to measure the start time of the algorithm*/
for(int i=1;i<a;i++){
temp=tab1[i];
int j=i;
while((j>0)&&(tab1[j-1]>temp))
{
tab1[j]=tab1[j-1];
j--;
}
tab1[j]=temp;
}
auto end=std::chrono::high_resolution_clock::now(); /* we set another mark to measure the end time of the sorting algorithm*/
std::chrono::duration<double> duration=end - start; /* here we get the time that the algorithm had to take in order to sort the array, we get it by substracting auto start - auto end*/
results[0][p]+=duration.count(); /* we sum the time result in a specific result index, for the sort1 algorithm: results[fixed 0 collumn][row depending on how many array sizes measurments have been made before]*/
}
void sort2(int p){/*variation 2 of insertion sort*/
int temp;
auto start=std::chrono::high_resolution_clock::now();
int j;
for(int i=1;i<a;i++){
j=0;
while((j<i)&&(tab2[j]<tab2[i])){
j++;
}
temp=tab2[i];
for(int ii=i;ii>j;ii--){
tab2[ii]=tab2[ii-1];
}
tab2[j]=temp;
}
auto end=std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration=end - start;
results[1][p]+=duration.count();
}
void sort3(int p) {/*variation 3 of insertion sort("mirror" of sort2)*/
int temp;
int j;
auto start = std::chrono::high_resolution_clock::now();
for (int i = a - 2; i >= 0; i--) {
j = a - 1;
while((j>i) && (tab3[j]>tab3[i])){
j--;
}
temp = tab3[i];
for (int ii = i; ii < j; ii++) {
tab3[ii] = tab3[ii + 1];
}
```

```
tab3[j] = temp;
}
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration = end - start;
results[2][p]+=duration.count();
}
void sort4(int p){ /*variation 4 of insert sorting algorithm mirror of sort1*/
int temp;
auto start=std::chrono::high_resolution_clock::now();
for(int i=a-1;i>=0;i--){
temp=tab4[i];
int j=i;
while((j<(a-1))&&(tab4[j+1]<temp)){
tab4[j]=tab4[j+1];
j++;
}
tab4[j]=temp;
}
auto end=std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration=end - start;
results[3][p]=results[3][p]+duration.count();
}
void sort5(int p){/*Binary insertion sorting alg*/
int temp;
int left;
int right;
int shot;
auto start=std::chrono::high_resolution_clock::now();
for(int i=1; i<a; i++){
temp=tab5[i];
left=0;
right=i;
while(left<right){
shot=(left+right)/2;
if(tab5[shot] < temp)
left = shot + 1;
else
right = shot;
}
shot = left;
for (int ii=i; ii>shot; ii--){
tab5[ii]=tab5[ii - 1];
}
tab5[shot]=temp;
}
auto end=std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration=end - start;
results[4][p]+=duration.count();
}
int main() {
int choice;
cout<<"MENU OPCJI"<<endl;
```

```
cout<<"Nacisnij 2, aby rololoczac pomiar automatyczny";
cout<<endl;
cout<<"Nacicnij 0, aby rozpoczac pomiar posortowanej";
cout<<endl;
cout<<"Nacisnij 1, aby rozpoczac pomiar odwrotnie posortowanej";
cout<<endl;
cin>>choice; /* after we chose the option number the algorithm goes as follows:*/
if (choice==2){/*here we perform very time consuming automatic measurments that perform
experiment 1 task*/
int c=0;/* c value is the number of how many array sizes measurments have been made
before*/
ofstream plik("test.txt", std::ios::out | std::ios::trunc );/* we open an output text file to save the
data later in the sorting measurment process*/
for(int n=10000;n<110000;n=n+100000){/* this is the main measure loop that varying from
the author wish iterates on the exact array sizes that are about to be generated, initially it has only
one array size - 10 000*/
a=n;/* we have to set a globall array size value to current n array size for the algorithms to
perform sorting*/
for(int i=1;i<=100;i++){/* we repeat the procedure generate data=>sort arras=>delete data
100 times in order to compute average time later*/
czas++;
randomize(a);
thread worker1(sort1,c);/* we use threads to speed up the whole process five times on the
worst case scenario, the c argument given for the sorting*/
thread worker2(sort2,c);/*algorithms is needed in order to sum the sorting time in the
right row of the result table*/
thread worker3(sort3,c);
thread worker4(sort4,c);
thread worker5(sort5,c);
worker1.join();/* the fastest algorithm complete theirs task to this point where the
program stops and waits for the slowest algorithms to join to continue the program on one thread*/
worker2.join();
worker3.join();
worker4.join();
worker5.join();
if(i%10==0){/*here we set some onscreen markings for the user to show him how much
time on average is needed for sorting 10 times the current n size arrays*/
cout<<"wykonalem "<<i<<" pomiarow..."<<endl;// with the slowest algortihm*/
cout<<"sredni czas: "<<(results[2][c]/i)<<endl;
}
deletemem();/*sorted arrays deletion*/
}
cout<<"sortowanie udalo sie dla n=: "<<n<<endl;/* another marking, telling us that we
performed 100 sortings on current array size and that no we gonna increase the arr size by 100 000
elements*/
for(int ii=0;ii<5;ii++) /* at the end of sorting specyfic array size 100 times we compute the
average sorting time for each algorithm by following loop:*/
results[ii][c]/=100;
c++;
}
for(int x=0;x<2;x++){ /* when we finish sorting for all array sizes we wanted to, we output the
results array to the output file by following loops:*/
```

```
for(int z=0;z<5;z++){
plik<<results[z][x]<<";";
}
plik<<endl;
}
plik.close();
}
else{
test2(choice);/* here we perform data generation for the asc sorted/desc sorterd sorting
experiment 2*/
sort1(0);/*sorting process*/
sort2(0);
sort3(0);
sort4(0);
sort5(0);
cout<<"wyniki dla tablic posortowanych/nieposortowanych"<<endl;/* this time we output
the results onscreen because there are always only five numbers as a result*/
cout<<results[0][0]<<" "<<results[1][0]<<" "<<results[2][0];
cout<<" "<<results[3][0];
cout<<" "<<results[4][0]<<endl;/* row indicies in the results table are now fixed as we perform
only one
array size sorting*/
deletemem();
}
return 0;
}
```

7. Warunki badawcze

Tab.3-sprzęt na którym wykonywany był pomiar:

Nazwa systemu operacyjnego	Microsoft Windows 10 Pro
Wersja	10.0.19045 Kompilacja 19045
Typ systemu	x64-based PC
Procesor	11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz, 2712 MHz, Rdzenie: 6, Procesory logiczne: 12
Zainstalowana pamięć fizyczna (RAM)	16,0 GB

8. Postawienie hipotez badawczych

Eksperyment 1:

Hipoteza H_0: Klasyczne algorytmy insertion sort (Sort1-Sort4) są mniej efektywne niż wariant wykorzystujący wyszukiwanie binarne (Sort5) dla losowych danych.

Hipoteza alternatywna H_1: Klasyczne algorytmy insertion sort (Sort1-Sort4) są bardziej efektywne niż wariant wykorzystujący wyszukiwanie binarne (Sort5)dla losowych danych.

Eksperyment 2:

Hipoteza H_0: sort1 oraz sort4 będą najszybsze w przypadku sortowania tablicy posortowanej oraz najwolniejsze w przypadku sortowania tablicy odwrotnie posortowanej.

Hipoteza alternatywna H_1: sort1 oraz sort4 będą najwolniejsze w przypadku sortowania tablicy posortowanej oraz najwolniejsze w przypadku sortowania tablicy odwrotnie posortowanej.

9. Instrukcja obsługi programu

Po skompilowaniu programu uruchamiamy go, wpisujemy znak z klawiatury '2', aby wykonac automatyczny pomiar tablic 10 000 alementów,

znak '1' aby posortować tablice nieposortowane 100 000 elementow,

znak '0' aby posortować tablice posortowane 100 000 elementow,

a następnie naciskamy enter, aby zatwierdzic wybór opcji

10. Wyniki eksperymentu

Tab.4Średnie wyniki pomiarów dla tablic 10 000 – 610 000:

arr size	sort1	sotrt2	sort3(rev2)	sort4(rev1)	binary insert
10000	0,0379679	0,0549638	0,0565902	0,0399049	0,0275337
110000	5,37046	7,37197	7,43851	5,54859	3,84069
210000	19,6387	26,668	26,772	20,0315	13,8196
310000	42,79	58,157	58,4228	43,6539	30,1794
410000	74,5457	101,516	101,861	76,1279	52,4445
510000	116,927	159,119	159,394	119,064	82,1244
610000	163,092	222,919	223,362	166,614	114,107

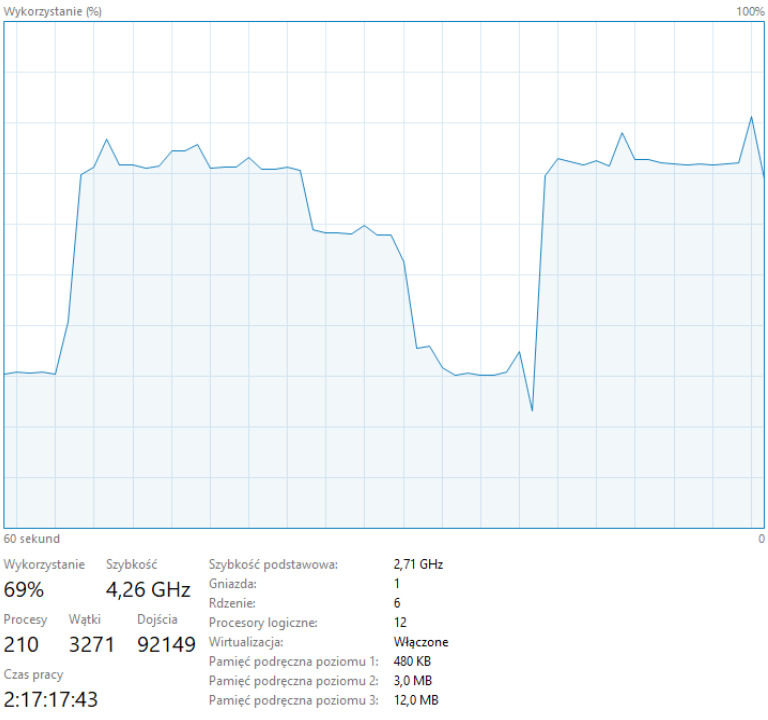
* rev – oznacza „lustrzane odbicie” algorytmu. np. insert3 jest tym samym algorytmem co insert2, jednak wyszukiwanie miejsca do wstawienia zaczyna od innej strony niż insert2

Czas wykonania pomiaru sortowania tablic 10-410k elementów:

```
sortowanie udalo sie dla n=: 310000
wykonalem 20 pomiarow...
wykonalem 40 pomiarow...
wykonalem 60 pomiarow...
wykonalem 80 pomiarow...
wykonalem 100 pomiarow...
sortowanie udalo sie dla n=: 410000

Process returned 0 (0x0)   execution time : 19576.541 s
Press any key to continue.
```

P



Zużycie procesora osiągnięte dzięki zastosowaniu wątków w trakcie sortowań:

Czas wykonania pomiaru sortowania tablic 510-610k elementów:

```
wykonalem 90 pomiarow...
sredni czas: 223.221
wykonalem 100 pomiarow...
sredni czas: 223.362
sortowanie udalo sie dla n=: 610000

Process returned 0 (0x0)   execution time : 38540.712 s
Press any key to continue.
```

Tab.5 Wynik posortowania tablicy posortowanej 100k elementów:

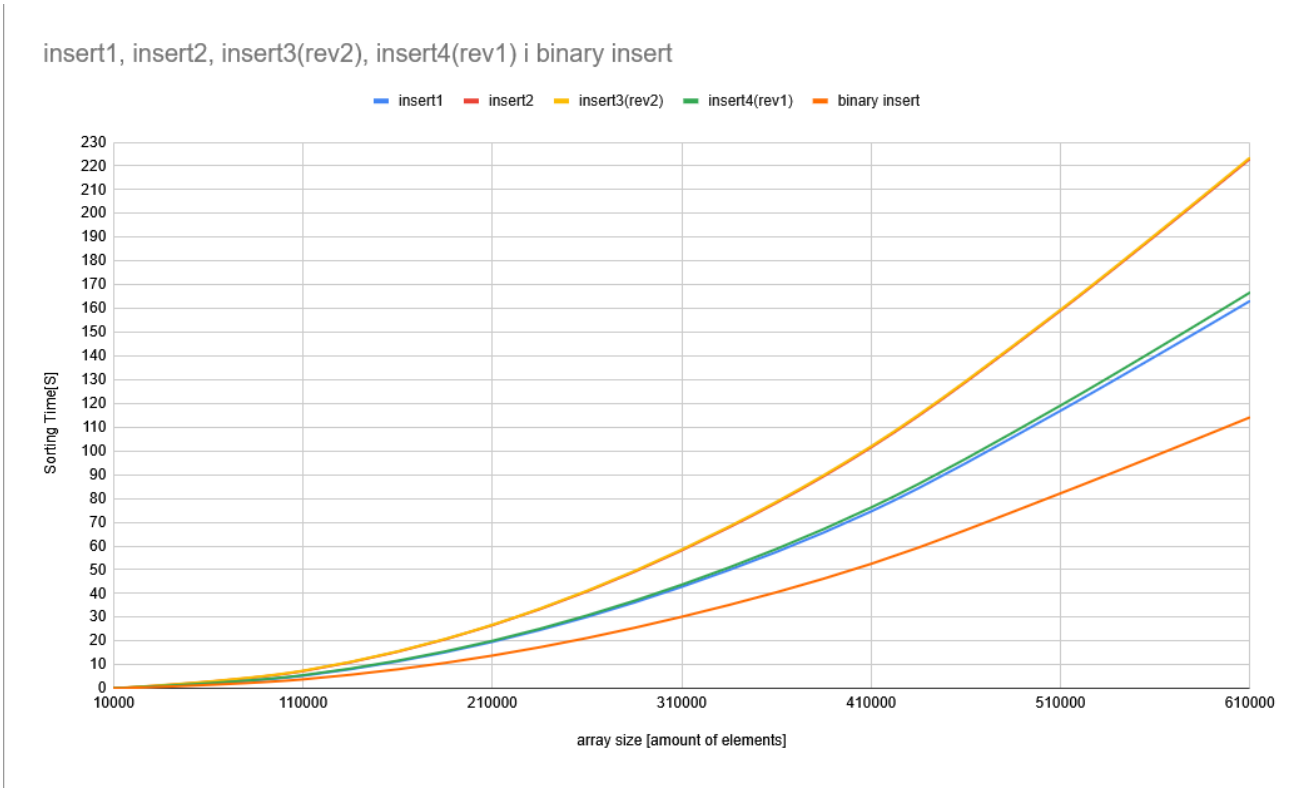
sort1	sort2	sort3(rev2)	sort4(rev1)	binary insert-sort5
0	7.36396	7.10483	0	0.0156215

Tab.6 Wynik posortowania tablicy posortowanej 100k elementów:

sort1	sort2	sort3(rev2)	sort4(rev1)	binary insert-sort5
9.46526	6.55733	6..4951	9.58054	6.53988

** insert 1 oraz 4 sortują tablice w nie mierzalnym czasie przez zegar z biblioteki chrono.

Wykres stworzony na podstawie tabeli Tab4:



11. Dyskusja i interpretacja wyników

Eksperyment 1:

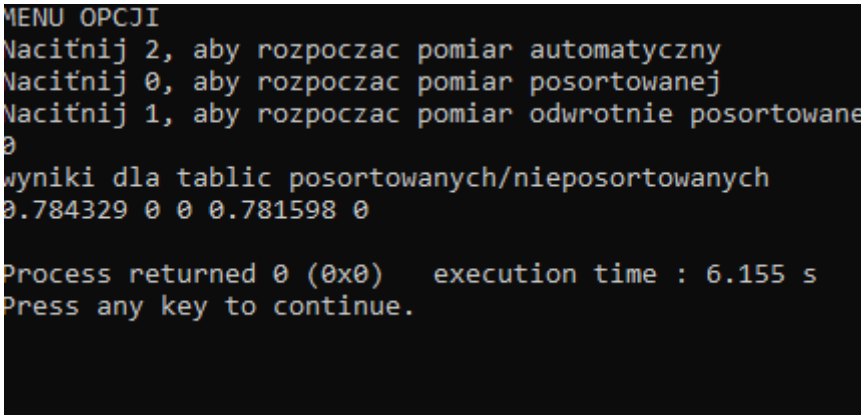
Algorytm Sort5 (binary insert) uzyskał najkrótszy czas wykonania dla każdego rozmiaru tablicy, co potwierdza hipotezę H_1.

Klasyczne warianty insertion sort(sort1 oraz sort4) miały drugą najlepszą wydajność z porównywanych algorytmów. Algorytmy sort2 oraz sort3 prezentowały najgorszą efektywność, co spowodowane jest zagnieżdżoną dodatkową pętlą for w algorytmie sortowania, czyli oddzieloną operacją szukania miejsca do wstawienia liczby w ciągu posortowanym od faktycznego przesuwania elementów ciągu posortowanego w celu zrobienia miejsca dla liczby do wstawienia.

Wzrost rozmiaru tablicy powoduje wykładniczy wzrost czasu wykonania dla wszystkich algorytmów.

Eksperyment 2:

Wyniki z sortowania tablicy posortowanej wykazują całkowite zdeklasowanie przez sort1 oraz sort4 (tych klasycznych sortowań przez wstawianie) reszty algorytmów, z racji na to, że sortowanie tablicy posortowanej to idealny, przypadek dla klasycznego, wydajnego sortowania przez wstawianie liniowe, to potwierdza hipotezę H_1 z ekperymentu 2. Posiadając procesor 2,7GHz, który wedle menadżera zadań w trakcie wykonywania sortowania przyspiesza do 4,5GHz sortowanie sort1 oraz sort2 mogą wykonywać się w czasie od 22µs do 88µs, co może być czasem niemierzalnym dla zegara biblioteki chrono.



Jak się okazuje, przy rozmiarze posortowanej tablicy równym 400 000 000 klasyczne wydajne algorytmy sortowania przez wstawianie sort1 oraz sort4 zaczynają wykazywać solidny, mierzalny czas sortowania, który przedstawiam na powyższym zdjęciu. W kodzie programu postanawiam jednak zostawić stały rozmiar tablicy posortowanej/nieposortowanej dla optymalnego przejrzenia realizacji kodu.

Wyniki z sortowania tablicy odwrotnie posortowanej są najgorszym możliwym przypadkiem sortowania dla klasycznego sortowania przez wstawianie liniowe. Dla tak zainicjowanej tablicy wszystkie warianty – poza sort1 i sort4 – w praktyce spędzają prawie cały czas na przesuwaniu elementów, a nie na porównaniach, stąd biorą się zbliżone czasy sort2, sort3 i sort5.

12. Wnioski

Wprowadzenie wyszukiwania binarnego w procesie znajdowania miejsca wstawienia znacznie poprawia wydajność algorytmu wstawiania liniowego. Przy dużych zbiorach danych efektywność algorytmów prostych jak warianty wymienionego wyżej algorytmu drastycznie spada – dla praktycznych zastosowań zalecane by były bardziej zaawansowane algorytmy sortowania nie implementowane w tym projekcie.

Ponad to realizacja projektu przyniosła mi dość sporo doświadczenia w używaniu wątków w języku C++ oraz doświadczenie w zarządzaniu pamięcią dynamiczną oraz obsługą repozytorium git.

13. Podsumowanie

Projekt umożliwił praktyczne porównanie wariantów prostych algorytmów sortowania. Otrzymane wyniki potwierdziły teoretyczne przewidywania dotyczące wydajności algorytmów oraz wskazały na znaczenie optymalizacji takich jak binarne wyszukiwanie miejsca wstawienia elementu.

14. Źródła

- „Binary Search Animated” https://youtu.be/eVuPCG5eIr4?si=8L_MnvfmMW0Z9OcH
- „The FASTEST sorting algorithm: Part 2 - Binary Insertion Sort” https://www.youtube.com/watch?v=6DOhQyqAAvU&t=7s
- „Insertion Sort Visualised” https://www.youtube.com/shorts/nZHNwb_evBg
- „Binary Insertion Sort” https://www.youtube.com/watch?v=-OVB5pOZJug
- „Top 7 Algorithms for Coding Interviews Explained SIMPLY” https://www.youtube.com/watch?v=kp3fCihUXEg&t=777s
- „10 Sorting Algorithms Easily Explained” https://www.youtube.com/watch?v=rbbTd-gkajw
- „Understanding Static in C++” https://www.youtube.com/watch?v=g4Dn2cwSrC4&t=600s
- „C++ Random Number Generator AKA STOP USING Rand()” https://www.youtube.com/watch?v=oW6iuFbwPDg&t=164s
- „Timing in C++” https://www.youtube.com/watch?v=oEx5vGNFrLk&t=261s
- „C++ Chrono Time Libray Tutorial In 12minutes” https://www.youtube.com/watch?v=QYaQStudgnE&t=298s
- „Build your first multithread aplication-Introdution to multithreading in modern C++” https://www.youtube.com/watch?v=xPqnoB2hjjA&t=1030s
- „memcpy() in C/C++” https://www.geeksforgeeks.org/memcpy-in-cc/
- „std::mt19937 Class in C++” https://www.geeksforgeeks.org/stdmt19937-class-in-cpp/
- „Don't use rand(): a guide to random number generators in C++ ” https://codeforces.com/blog/entry/61587?locale=ru
- Notatki z kursu Algorytmy i złożonośc obliczeniowa (pkt 4)