Algorytmy i struktury danych sprawozdanie z laboratorium nr 1

Aleksandra Cichecka
numer albumu: 287362

Październik 2025

Spis treści

1 Wstęp

W dokumencie zawarta jest analiza kilku wybranych algorytmów sortowania. Celem pracy było zaimplementowanie algorytmów sortujących, a także ich wersji z wskazanymi modyfikacjami. Następnie przeprowadzono testy, dzięki którymi można porównać efektywność obu algorytmów.

Algorytmy które zostały przeanalizowane to

- INSERTION SORT oraz INSERTION SORT z modyfikacją polegającą na wstawianiu "na raz"dwóch elementów tablicy
- MERGE SORT oraz MERGE SORT z modyfikacją polegająca na dzieleniu na trzy części zamiast dwóch
- HEAP SORT oraz HEAP SORT z modyfikacją używającą zamiast kopców binarnych kopce ternarne

2 Testy

Testy zostały wykonane mierząc trzy wskaźniki, czas działania algorytmu, ilość porównań oraz ilość przypisań. Dla każdego algorytmu wykonano testy na takich samych danych dla tablic małych (do 5 elementów), średnich (50 elementów), dużych (1000 elementów), inne (kolejno 2, 10, 20, 100, 500 elementów) oraz na różnych losowych danych dla tablic bardzo dużych (10000 elementów i powyżej).

Dla tablic małych przetestowano algorytmy na takich samych danych dla tablicy pustej, z dwoma elementami, pięcioma elementami które są 0, są posortowane, są posortowane odwrotnie, są losowe, posiadają w sobie liczby ujemne i mają powtarzającą się liczbę.

Dla tablic średnich przetestowano algorytmy na takich samych danych dla tablicy o 50 elementach, która ma losowe elementy, jest posortowana, jest posortowana odwrotnie, ma wszystkie równe elementy, posiada liczby ujemne i ma powtarzająca się liczbę.

Dla tablic dużych przetestowano algorytmy na takich samych danych dla tablicy o 1000 elementach, która ma losowe elementy, jest posortowana, jest posortowana odwrotnie, ma wszystkie równe elementy i jest posortowana w 95

Przetestowano także działalność algorytmów na takich samych tablicach o kolejno 2, 10, 20, 100 oraz 500 elementach.

W przypadku tablic bardzo dużych (10000 elementów i powyżej) dla każdego algorytmu każdorazowo losowo generowano elementy w tablicy. Celem było wykonanie 10 testów na każdej wielkości, a następnie pobrania z nich średniej.

Dokładne dane na których testowane były algorytmy (poza tymi 10k+ elementów) znajdują się w dołączonym folderze "DO TESTOW", natomiast zapisane wyniki znajdują się w pliku excel "testy".

3 Insertion sort

Klasyczny algorytm sortowania insertion sort (sortowanie przez wstawianie) działa w sposób podobny do tego jak ludzie ustawiają karty. Kolejne pojedyncze elementy wejściowe są ustawiane na odpowiednie miejsce docelowe.

W przypadku zmodyfikowanego insertion sort, bierzemy na raz dwa elementy, które najpierw porównujemy i sortujemy między sobą, a następnie wstawiamy je w prawidłowe miejsce w tabeli.

3.1 Fragment kodu

```
for(int i=0; i<n-1; i=i+2) {
    if(t[i]>t[i+1]) swap(t[i], t[i+1]);

    int a=t[i], b=t[i+1];
    int j=i-1;

    while(j>=0 && t[j]>b)
    {
        t[j+2]=t[j]; j--;
    }
    t[j+2]=b;

    while(j>=0 && t[j]>a)
    {
        t[j+1]=t[j]; j--;
    }
    t[j+1]=a;
}
```

Powyższy kod przedstawia zmodyfikowane sortowanie przez wstawianie, gdzie sortujemy parami elementów zamiast pojedynczych. Dzięki temu redukujemy liczbę porównań, a także minimalizujemy liczbę przesunięć.

3.2 Analiza testów insertion sort

W testach dla małych, średnich i dużych tablic skupimy się na różnicach w ilości porównań oraz ilości przypisań między algorytmem bez modyfikacji, a tym z modyfikacjami. Czas wykonania algorytmu dla każdego wynosi mniej niż 0 mikrosekund stąd pominiemy tu ten wskaźnik. Dopiero dla tablic z większa ilością elementów niż 10000 czas wykonania algorytmu zaczyna się wydłużać.

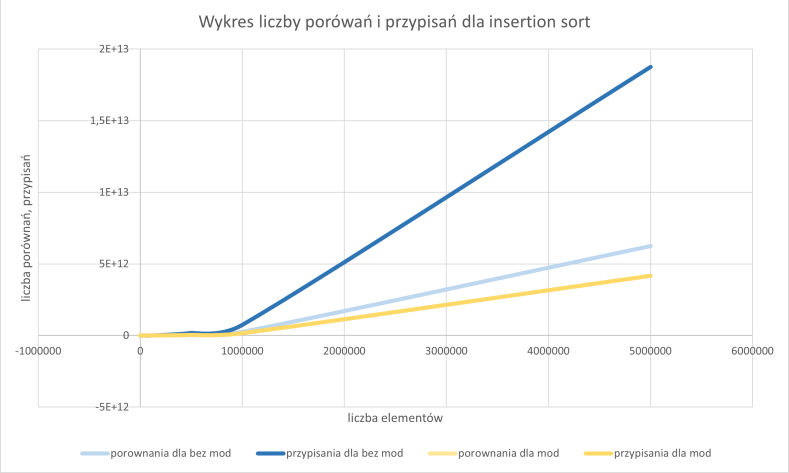
3.2.1 Testy dla tablic małych, średnich i dużych

Dla pustej tablicy oba algorytmy wykonują się poprawnie i zwracają wszędzie zera. Insertion sort bez modyfikacji zawsze wygrywa (tzn. ma mniejsza ilość przypisań i porównań) z zmodyfikowanym jeżeli tablica jest posortowana, prawie posortowana bądź ma elementy równe. W małych tablicach wygrywa również

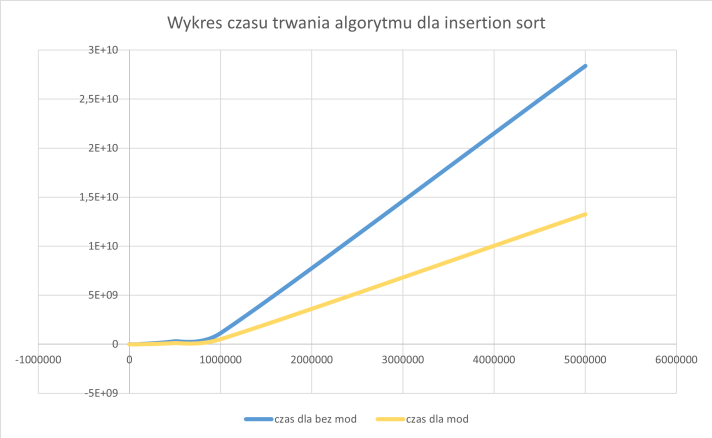
jeżeli elementy są ujemne. Dla średnich lepszy jest już zmodyfikowany, a także w przypadku tablic losowych czy posortowanych odwrotnie.

3.2.2 Testy dla losowych tablic o różnych elementach

W tym przypadku już weźmiemy pod uwagę czas wykonywania algorytmu. Poniżej znajdują się wykresy pokazujące zależności między czasem wykonywania zmodyfikowanego sortowania, a niezmodyfikowanego. To samo dla ilości porównań i przypisań.



Algorytm bez modyfikacji wykonuje zdecydowanie więcej powtórzeń i przypisań niż ten zmodyfikowany.



Algorytm z modyfikacjami wykonuje się znacznie szybciej niż ten bez modyfikacji.

3.2.3 Wnioski

Dla większości przypadków, kiedy mamy do czynienia z danymi losowymi, wielkimi zbiorami czy najgorszymi przypadkami wersja zmodyfikowana jest wyraźnie lepsza. Jeżeli nie wiemy nic o danych lepiej wybrać wersję zmodyfikowaną. Natomiast gdy dane są już w większości posortowane to znacznie lepsza będzie klasyczna wersja algorytmu.

4 Merge sort

Klasyczny algorytm sortowania merge sort (sortowanie przez scalanie) stosuje metodę "dziel i zwyciężaj". Użyta jest w nim struktura rekurencyjna. Wyróżniamy w nim trzy podstawowe kroki. Pierwszym jest podział elementów na dwie równe części. Następnie stosujemy sortowanie przez scalanie dla każdej z nich oddzielnie. Na koniec łączymy posortowane podciągi w jeden posortowany ciąg.

W przypadku zmodyfikowanego merge sort, dzielimy elementy na trzy równe części zamiast na dwie. Reszta sortowania przebiega w taki sam sposób.

4.1 Fragmenty kodu

```
void MERGESORT(int t[], int p, int k)
{
    int s=0;
    if(p<k)
    {
        s=(p+k)/2; //srodek
        MERGESORT(t, p, s); //sortuje lewa czesc od p do s
        MERGESORT(t, s+1, k); //sortuje prawa czesc od s+1 do k
        MERGE(t, p, s, k); //laczy posortowane czesci
    }

}
```

Powyższy kod przedstawia główną funkcje klasycznego sortowania merge sort, w którym wyznaczamy środek, a następnie rekurencyjnie stosuje metode "dziel i zwyciężaj".

```
//scala trzy tablice
for(int l=p; l<=k; l++)
{
    porownania+=3;
    if(L[i]<=S[j] && L[i]<=R[g])
    {
        t[l]=L[i];
        i=i+1;
    }else if(S[j]<=L[i] && S[j]<=R[g])
    {
            t[l]=S[j];
            j=j+1;
    }else
    {
        t[l]=R[g];
        g++;
    }
    przypisania+=1;

}
```

Powyższy kod przedstawia trójdzielne scalanie zmodyfikowanego merge sorta. W tym przypadku musimy porównać więcej elementów między sobą.

4.2 Analiza testów merge sort

Podobnie jak w przypadku insertion sort, przy analizie małych, średnich oraz dużych tablic również pominiemy czas wykonywania algorytmu, ponieważ jest on tak samo niski w każdym wypadku. Dopiero dla bardzo dużych tablic weźmiemy go pod uwagę. A teraz będziemy porównywać wyniki patrząc tylko na porównania i przypisania.

4.2.1 Testy dla tablic małych, średnich i dużych

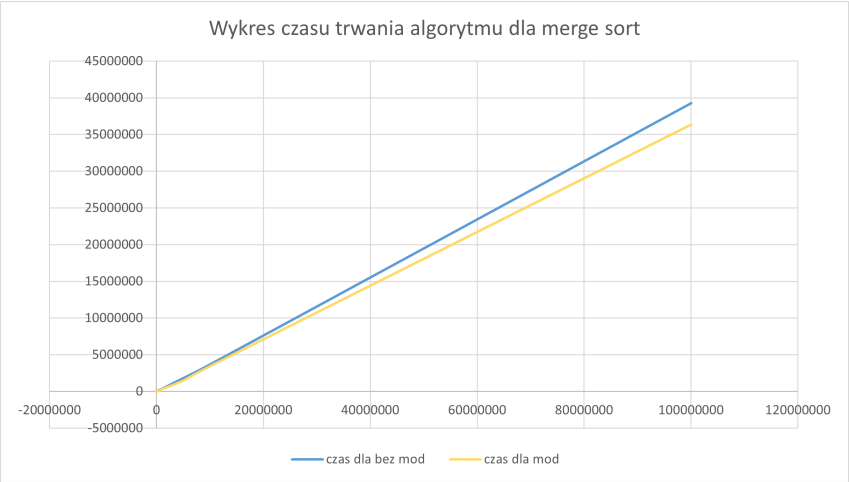
Dla pustej tablicy oba algorytmy wykonują się poprawnie i zwracają wszędzie zera. Niezależnie od kolejności elementów tablicy, liczba porównań oraz liczba przypisań jest zawsze taka sama. Tyczy się to zarówno zmodyfikowanego jak i klasycznego algorytmu. Tablice posortowane, posortowane odwrotnie, losowo, a także z duplikatami dają taką samą ilość.

W każdym przypadku zmodyfikowany merge sort ma wiekszą ilość porównań, ale mniejszą liczbą przypisań. Z uwagi na to, że przypisania często są 'kosztowniejsze' niż porównania, ogólna wydajność jest lepsza w jego przypadku. Zwykły merge sort ma przewagę nad nim tylko w przypadku tablic do 10 elementów.

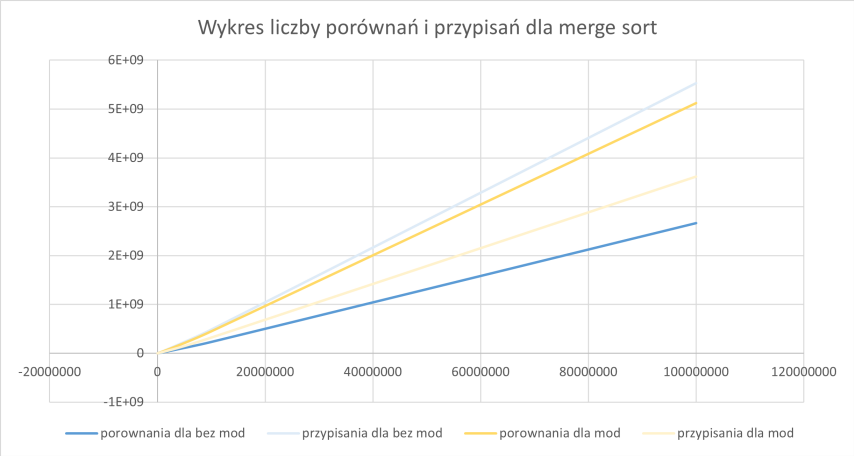
4.2.2 Testy dla losowych tablic o różnych elementach

Tutaj bierzemy już pod uwagę czas wykonywania algorytmu.

Poniżej znajdują się wykresy pokazujące zależności między czasem wykonywania zmodyfikowanego programu, a niezmodyfikowanego. To samo dla ilości porównań oraz ilości przypisań.



Na wykresie powyżej widzimy, że czas wykonania dla algorytmu bez modyfikacji jest dłuższy niż dla tego z modyfikacją. Na początku różnica ta jest niewielka, ale ze zwiększaniem ilości elementów robi się coraz większa.



Na powyższym wykresie widzimy natomiast różnice w ilości przypisań i porównań dla coraz to większych tablic. Zmodyfikowane sortowanie wykonuje zdecydowanie więcej porównań niż zwykłe, ale ma o wiele mniej przypisań niż on.

4.2.3 Wnioski

Jeżeli chodzi o sortowanie róznie ułożonych danych w tablicach o takiej samej wielkości nie ma znaczenia które sortowanie wybierzemy, ponieważ zawsze wykonują one tę samą liczbę powtórzeń oraz tę samą liczbę porównań. Jeżeli natomiast pracujemy na danych o różnych rozmiarach tablic, raczej powinniśmy wybierać zmodyfikowany algorytm, gdyż jest on odrobinę szybszy niż zwykły.

5 Heap sort

Klasyczny algorytm sortowania heap sort (sortowanie przez kopcowanie) podczas wykonywania konstruuje kopiec, który w rzeczywistości jest drzewem binarnym. Działanie algorytmu jest podzielone na dwie fazy. W pierwszej sortowane elementy reorganizowane są w celu utworzenia kopca, w drugiej zaś dokonywane jest właściwe sortowanie.

W Zmodyfikowanej wersji heap sorta korzystamy z kopców ternarnych zamiast binarnych. W niej każdy wierzchołek ma co najwyżej troje dzieci.

5.1 Fragment kodu

```
if (l<heap_size)
{
    if (t[l]>t[i])
    {
        largest=l;
    }
}
if (m<heap_size)
{
    if (t[m]>t[largest])
    {
        largest = m;
    }
}
if (r < heap_size)
{
    if (t[r] > t[largest])
    {
        largest = r;
    }
}
```

Powyżej znajduje się kluczowy fragment zmodyfikowanego heap sorta. W odróżnieniu od klasycznej wersji, gdzie korzystamy z kopca binarnego tutaj używamy kopca trójdzielnego, a wraz z tym sprawdzamy trójkę dzieci zamiast dwóch.

5.2 Analiza testów heap sort

Tak samo jak dla poprzednich przypadków, czas weźmiemy pod uwagę dopiero przy większych tablicach. Na razie skupimy się na różnicach w porównaniach i przypisaniach.

5.2.1 Testy dla tablic małych, średnich i dużych

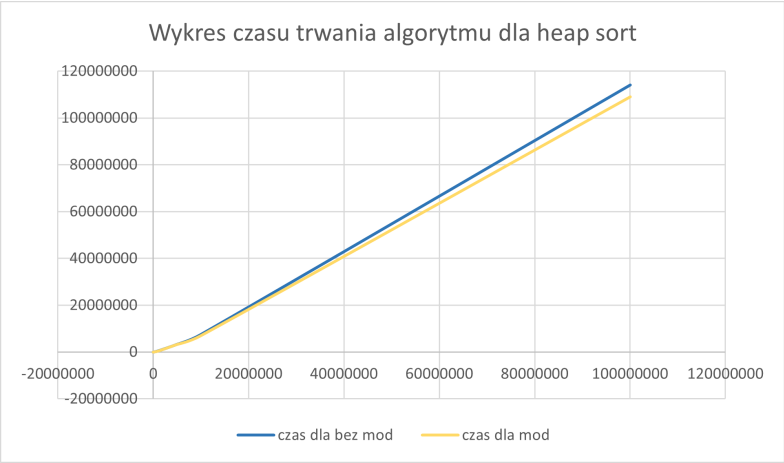
Dla pustych tablic oba algorytmy wykonują się prawidłowo, chociaż ten z modyfikacjami wykonuje 4 porównania. Dla małych tablic lepsze wyniki w tablicach o dwóch elementach, równych elementach, posortowanych, a także ujem-

nych ma heap sort bez modyfikacji. Jednak dla tablic losowych i z duplikatami lepiej działa z modyfikacjami.

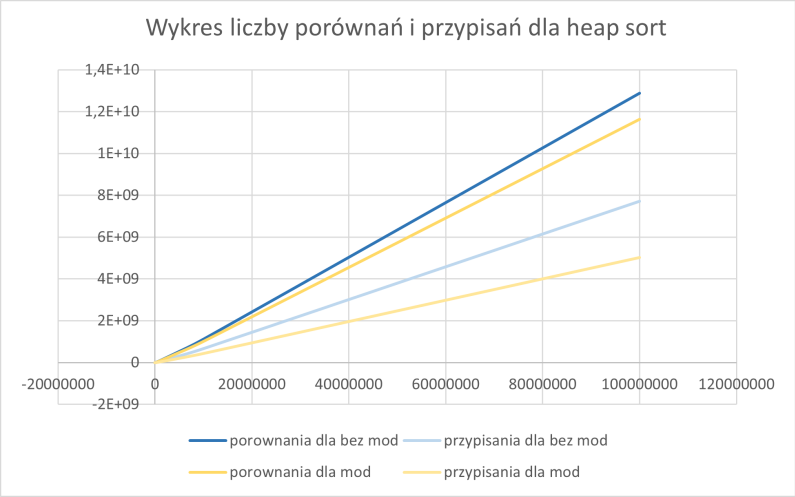
Dla tablic średnich lepszy jest heap sort z modyfikacjami. Przegrywa jedynie w przypadku gdy wszystkie elementy tablicy są równe. Tak samo przebiega sprawa w tablicach dużych.

5.2.2 Testy dla losowych tablic o różnych elementach

Poniżej znajdują się wykresy pokazujące zależności między czasem wykonywania zmodyfikowanego programu, a niezmodyfikowanego. To samo dla ilości porównań oraz ilości przypisań.



Na wykresie powyżej widzimy że czas wykonania obu algorytmów nieznacznie się różni, co ciekawe na korzyść dla algorytmu bez modyfikacji.



Na powyższym wykresie zdecydowanie widać przewagę jaką ma zmodyfikowane sortowanie nad klasycznym. Ma on zarówno mniejszą ilość porównań jak i ilość przypisań.

5.2.3 Wnioski

W większości przypadkach, zwłaszcza gdy mowa jest o tablicach z duża ilością elementów lepiej wybrać zmodyfikowany algorytm. Zwykłe sortowanie jest lepsze jedynie w przypadku gdy wszystkie elementy są sobie równe, ale raczej przy takich danych nie potrzebujemy ich sortować.

5.3 Zakończenie

Na podstawie wyżej przeprowadzonych testów na algorytmach zwykłych, a także ich zmodyfikowanych wariantach wynika, że zawsze lepiej jest korzystać z sortowań z modyfikacjami. Są one szybsze oraz wykonują mniej porównań, a także mniej przypisań.