Algorytmy i struktury danych sprawozdanie z laboratorium nr 2

Aleksandra Cichecka
numer albumu: 287362

Listopad 2025

Spis treści

1 Wstęp

W dokumencie zawarta jest analiza kilku wybranych algorytmów sortowania. Celem pracy było zaimplementowanie algorytmów sortujących, a także ich wersji ze wskazanymi modyfikacjami. Dodatkowo została stworzona własna lista dla której zaimplementowano również działający insertion sort. Algorytmy na których się skupimy to:

- QUICK SORT z modyfikacją polegającą na dzieleniu tablicy przy pomocy dwóch elementów, na trzy części
- RADIX SORT, który sortuje liczby naturalne względem dowolnej podstawy d, z modyfikacją która poprawnie sortuje liczby ujemne
- BUCKET SORT z modyfikacją, która sortuje dowolne dane wejściowe, a nie tylko (0,1] jak w przypadku klasycznej wersji
  - W tym algorytmie dodatkowo został użyty algorytm insertion sort, który działa na listach

Przeprowadzone zostały także testy, które dzięki którymi można porównać efektywność algorytmów.

2 Quick Sort

2.1 Opis działania algorytmu

Quick Sort jest algorytmem sortowania, który działa na zasadzie "dziel i zwyciężaj". Z tablicy wybiera się element rozdzielający, po czym tablica jest dzielona na dwa fragmenty. Do pierwszego fragmentu przenoszone są wszystkie elementy nie większe od rozdzielającego, a do drugiego fragmentu wszystkie większe. Potem sortuje się osobno początkową i końcową część tablicy. Rekursja kończy się gdy kolejny fragment uzyskany z podziału zawiera pojedynczy element, jako że jednoelementowa tablica nie wymaga sortowania.

Zmodyfikowana wersja polega na dzieleniu tablicy na trzy części używając dwóch punktów odniesienia. Są nimi pierwszy oraz ostatni element tablicy. Jeżeli pierwszy jest większy od drugiego to są zamieniane. Tablica podzielona jest na elementy mniejsze od pierwszego punktu, elementy wieksze od drugiego punktu oraz elementy pomiedzy dwoma punktami.

2.2 Fragment kodu

```
1   if(t[dol]>t[gora])
2 {
3     swap(t[dol], t[gora]);
4     przypisania+=3;
5 }
6 long liczba1=t[dol]; //mniejszy punkt odniesienia
```

```
7 long liczba2=t[gora]; //wiekszy punkt odniesienia
```

Inicjalizacja punktów odniesienia w zmodyfikowanej wersji, zapewnia że zawsze mamy uporządkowaną parę punktów.

```
1    while(i<=prawy)
2     {
3         if(t[i]<liczba1)
4         {
5             swap(t[i], t[lewy]);
6             lewy++;
7         } else
8         {
9             if(t[i]>liczba2)
10            {
11                while(t[prawy]>liczba2 && i<prawy) prawy--;
12            }
13                swap (t[i], t[prawy]);
14
15                if(t[i]<liczba1)
16                {
17                    swap(t[i], t[lewy]);
18                    lewy++;
19                }
20            }
21    }
```

Jednoczesne obsługiwanie trzech przedziałów wartości

3 Radix Sort

3.1 Opis działania algorytmu

Radix Sort to algorytm sortowania pozycyjnego, który sortuje elementy względem konkretnej cyfry od najmniej znaczących do najbardziej znaczących pozycji. Algorytm wykorzystuje sortowanie przez zliczanie jako procedurę pomocniczą do sortowania względem poszczególnych cyfr.

Kluczowym parametrem jest podstawa liczbowa (d), która określa system liczbowy, w którym operujemy. Wybór podstawy wpływa na liczbę iteracji i na wydajność. Mniejsza podstawa oznacza więcej cyfr do posortowania, ale mniejsze zbiory do zliczania. W przypadku wersji zmodyfikowanej nasz zakres wynosi 2*d podczas gdy w klasycznej wersji jest to poprostu d.

Sortowanie przez zliczanie działa w trzech krokach:

1. Zlicza wystąpienia konkretnych cyfr
2. Przekształca zliczenia w pozycje końcowe
3. Wstawia elementy w prawidłowe miejsce

W klasycznej wersji Radix Sort działa jedynie dla liczb nieujemnych. Znajduje wartość maksymalną, a następnie sortuje elementy cyfra po cyfrze za pomocą sortowania przez zliczanie, zaczynając od najmniej znaczącej aż do najbardziej znaczącej.

W zmodyfikowanej wersji, która działa również dla liczb ujemnych dodatkowo do wartości maksymalnej znajduje te minimalną. Nowym krokiem jest także przesunięcie wartości w górę tak aby najmniejsza liczba była równa 0. Następnie sortuje cyfra po cyfrze z wykrywaniem ostatniej cyfry. Na koniec przywraca oryginalne wartości poprzez przesunięcie wartości w dół.

3.2 Fragment kodu

```
1     //iteracja od konca
2     for(int i=n-1; i>=0; i--)
3     {
4         int cyfra = (t[i]/e)%d;
5         b[c[cyfra]-1]=t[i]; // element na wlasciwej pozycji
6         c[cyfra]--; //zmniejsza licznik dla danej cyfry
7     }
```

Iteracja od końca gwarantuje stabilność sortowania. Elementy o tej samej wartości zachowują względną kolejność. Jest to kluczowy moment w sortowaniu przez zliczanie.

```
1     //jesli sa ujemne to dodatkowo przesuwa
2     if(mins<0)
3     {
4         int przesuniecie = -mins;
5         for(int i=0; i<n; i++)
6         {
7             t[i]+=przesuniecie;
8         }
9         maks+=przesuniecie;
10    }
```

Kluczowe przesunięcie w zmodyfikowanej wersji, aby działały liczby ujemne.

4 Insertion Sort na listach

4.1 Opis działania algorytmu

Program tworzy listę jednokierunkową z danych wejściowych, sortuje ją algorytmem Insertion Sort, a następnie wyświetla posortowaną listę. Robi to poprzez stworzenie nowej posortowanej listy. Wybiera jeden element z nieposortowanej i wstawia go w prawidłowe miejsce do posortowanej tablicy.

4.2 Opis implementacji listy jednokierunkowej

Zaimplementowana została lista jednokierunkowa. Składa się ona z węzłów (Node).

```
 class Node
{
public:
    int data; // wartosc elementu
    Node* next; //wskanzik na nastepny elelment

    Node(int value)
    {
        data=value; //przypisanie wartosci do elementu
        next=nullptr; //ze nie ma nastpenego
    }

};
```

Klasa Node jest pojedynczym elementem listy. Przechowuje ona wartość oraz wskaźnik na następny element w polu next. Konstruktor tworzy nowy węzeł w pamięci, inicjalizuje pole data podaną wartością oraz ustawia next na nullptr.

```
class Lista
{
public:
    Node* head; //wskaznik na poczatek listy
    int size; //rozmiar listy

    Lista()
    {
        head=nullptr;
        size=0;
    } //ze na poczatku pusta i rozmiar 0
};
```

implementacja klasy Lista

Klasa posiada 3 funkcje. Append która odpowiada za dodawania na koniec, Insert, która wstawia element w dowolne miejsce, a także Display, która wyświetla listę.

```
     void append(int value)
    {
        Node* newNode = new Node(value); //tworzy nowy element

        if(head==nullptr) //ze jesli pusta to nowy jest poczatkiem
        {
            head=newNode;
        } else //inaczej przechodzi do ostatniego elementu
        {
            Node* temp=head;
            while(temp->next) //szuka gdzie next jest nullptr puste
            {
                temp=temp->next; //idzie do kolejnego
            }
            temp -> next = newNode; //i dopisuje nowy na koneic
        }
```

```
    }
```

Funkcja append. Tworzy nowy węzeł z podaną wartością i jeśli lista jest pusta, to nowy element staje się pierwszym elementem listy. Jeżeli nie jest pusta to przechodzimy przez wszystkie elementy aż do ostatniego i dopisujemy nowy element na końcu.

```
    void insert(Node* node, int value)
    {
        Node* newNode = new Node(value); //tworzy nowy elment

        if(!node) //jesli pusty to na poczatek
        {
            newNode->next=head; //nowy elmenyt wkakuje na stary poczatek
            head=newNode; //nowy elment jest nowym poczatkiem
        } else //wstaiwanie za podanym elmeentem
        {
            newNode -> next = node -> next; //nowy na to co wskazywal stary
            node -> next = newNode; //stary elemnt wkazuje na nowy
        }
    }
```

Funkcja insert. Wstawia nowy element za wskazanym węzłem. Jeżeli przekazany wskaźnik jest pusty to nowy węzeł wstawiany jest na początek listy. Jego next wskazuje dotychczasowy head, a head zostaje ustawiony na nowy element. W innym przypadku nowy element wstawiany jest za węzłem.

```
     void display()
    {
        Node* temp =head;
        while(temp)
        {
            cout << temp->data << " ";
            temp=temp->next;
        }
        cout << endl;
    }
```

Funkcja display. Przechodzi po liście i wypisuje wszystkie wartości elementów aż do wskaźnika nullptr.

Ostatnim elementem listy jest destruktor, który zwalnia pamięć

```
    ~Lista()
    {
         while(head)
      {
            Node* temp=head;
            head=head->next;
            delete temp;
       }
    }
```

Destruktor

4.3 Fragmenty kodu Insertion Sort

```
if (!lista.head || !lista.head->next)
    {
        return;
    }
```

Jeżeli tablica ma 0 lub 1 element to znaczy że jest już posortowana

```
while (posortowanacurrent && posortowanacurrent->data < current->
    data)
        {
            porownania++;
            prev=posortowanacurrent; //zapamietuje obecny element
            posortowanacurrent=posortowanacurrent->next;
        }
```

Szukanie miejsca do wstawienia elementu. Przechodzi przez posortowaną część listy szukając pierwszego elementu który jest większy bądź równy od aktualnego.

5 Bucket sort

5.1 Opis działania algorytmu

Idea działania algorytmu sortowania kubełkowego jest taka, że zadany przedział liczb dzieli na k podprzedziałów o równej długości. Następnie przypisuje liczby z sortowanej tablicy do odpowiednich kubełków. Sortuje liczby w niepustych kubełkach. W zaimplementowanej wersji używamy do tego wyżej opisanego Insertion Sort działającego na listach. Potem wypisuje po kolei ich zawartość. W standardowej wersji przyjmuje się że sortowane liczby należa do przedziału od 0 do 1.

Zmodyfikowana wersja różni się od klasycznej tym, że obsługuje dowolne liczby całkowite. Robi to poprzez przesunięcie wszystkich elementów, tak aby najmniejsza wartość równała się zero. Następnie skaluje je do przedziału [0,1), gdzie działa już jak klasyczny algorytm.

5.2 Fragment kodu

```
double skala = (double)(t[i] - mins) / (maks - mins + 1);
int ktorykubelek = int(n * skala);
```

Mechanim skalowania wartości, przekształca dowolne liczby całkowite na format [0,1). Przesuwa wszystkie wartości tak aby najmniejsza stała się 0.

6 Opis testów

6.1 dla Radix Sort

Testujemy klasyczny algorytm sortujący Radix Sort. Testujemy działanie programu dla różnych podstaw d. Testowane podstawy to: 2, 10, 16, 100, 256, 1000. Wykonujemy po 10 testów dla każdej wielkości od 8 elementów, rosnących wykładniczo x2 aż do 4194304 elementów. Program losuje liczby od 1 do 1000000, przypisuje je do tablicy, a następnie sortuje. Z 10 testów dla jednej wielkości, dla jednej podstawy, wyciągamy medianę, którą jest później analizowana. Analizowane wskaźniki to czas wykonania algorytmu, liczba porównań oraz liczba przypisań.

6.2 dla Quick Sort i Bucket Sort

Testujemy zarówno klasyczną wersje jak i zmodyfikowaną Quick Sort, a następnie porównujemy ją do zmodyfikowanej wersji Bucket Sort. Podobnie jak dla Radix Sort wykonujemy po 10 testów dla każdej wielkości od 8 elementów rosnących wykładniczo x2 aż do 4194304 elementów na losowych danych. Wyciągamy medianę, a następnie porównujemy wyniki, na takich samych wskaźnikach co w Radix Sort.

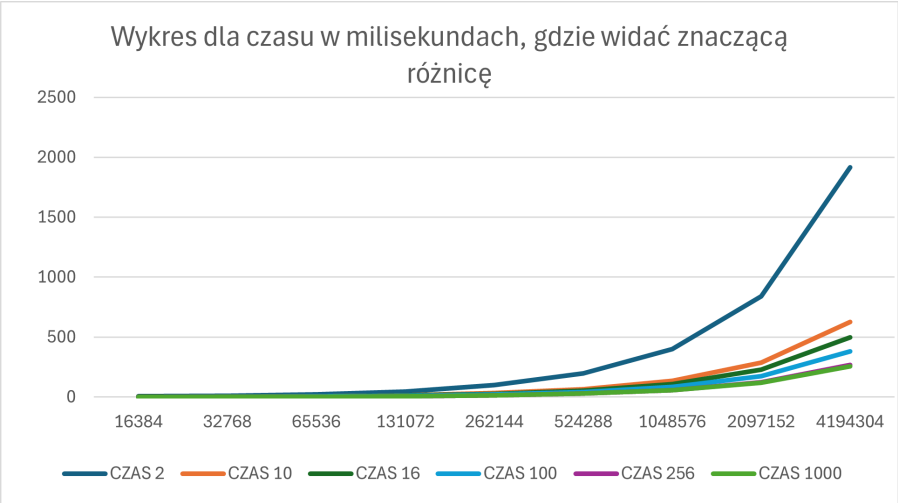
7 Testy i ich analiza

7.1 Radix Sort dla rożnych podstaw d

7.1.1 Czas

Do 4096 elementów tablicy, radix sort wykonuje się bardzo szybko niezależnie od podstawy d. Wszędzie wynosi blisko 0 milisekund.Przy większych ilościach elementów, czas przy podstawie d=2 zaczyna się powoli zwiększać. Znaczącą różnice widać przy elementach 16k+. Można zauważyć, że najwolniej działa algorytm o podstawie 2, a najszybciej ten o podstawie 1000

	16384	**32768**	**65536**	**131072**	**262144**	**524288**	**1048576**	**2097152**	**4194304**
CZAS 2	5,5	11	22,5	45,5	98	195	401	838	1917
CZAS 10	1	3,5	7	15	33	65	135	287,5	627,5
CZAS 16	1	3	6	12	26	52,5	107,5	229	499
CZAS 100	0	2	4,5	10	21	40	85	172,5	379
CZAS 256	0	1	3	6,5	14	28	58	122	268
CZAS 1000	0	0,5	3	6	14	28	55,5	118	256

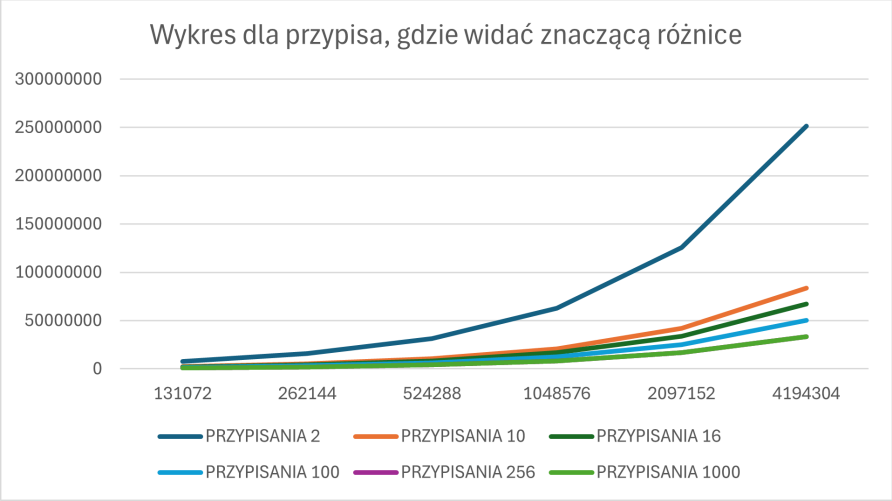


7.1.2 Porównania

Niezależnie od podstawy wszędzie jest taka sama liczba porównań. Zwiększa się ona jedynie wraz z zwiększeniem ilości elementów w tablicy.

7.1.3 Przypisania

Dla elementów od 8 do 128 najmniej przypisań ma algorytm o podstawie 10. Następnie do samego końca najmniej ma ich ten o podstawie 256, ale przy 4k elementach oraz więcej bliski jemu jest ten o podstawie 1000. A najwięcej zdecydowanie ma algorytm o podstawie 2. Poniżej wykres dla przypisań gdzie widać znaczącą różnicę.



7.1.4 Wnioski

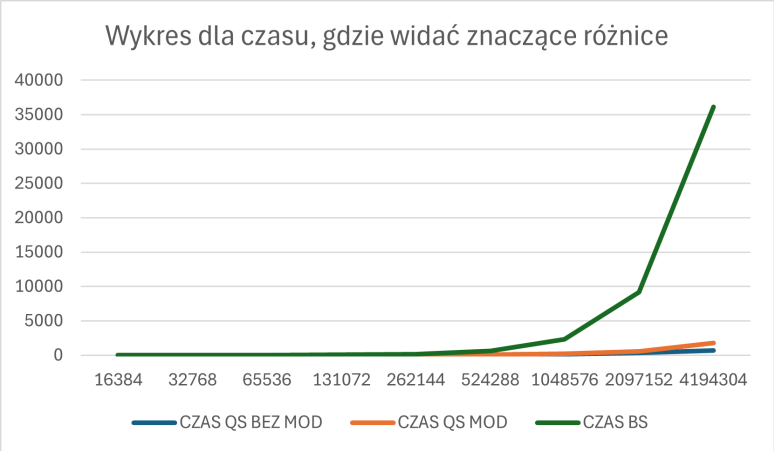
Dla małych tablic (do 4096 elementów) wszystkie warianty Radix Sort sa bardzo szybkie. Przy większych rozmiarach widać, że ta o podstawie 2 jest najwolniejsza, a o podstawie 1000 najszybsza. Z uwagi na to że liczba porównań jest identyczna dla wszystkich podstaw możemy wysnuć wniosek, że różnica w czasie wykonywania pochodzą z innych operacji. Przypisań najwięcej ma podstawa 2, najmniej natomiast 256 oraz 1000. Najbardziej optymalną wersją jest ta o podstawie 256 bądź 1000, ponieważ różnice między nimi są niewielkie, a oferują najlepszy czas wykonania przy najmniejszej ilości przypisań. Zdecydowanie warto unikać podstawy 2.

7.2 Quick Sort i Bucket Sort

7.2.1 Czas

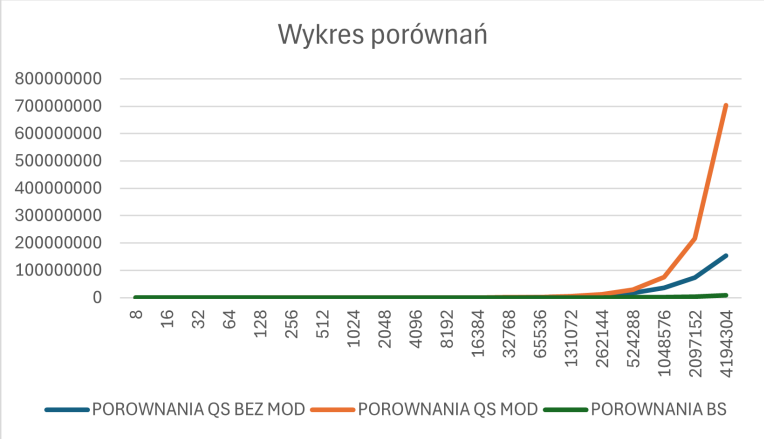
Dla tablic małych i średnich, aż do 8k elementów, sortowanie wykonuje się w tempie bardzo szybkim. Nie została naliczona ani milisekunda. Dopiero przy 16k elementowych tablicach czas zaczyna się pojawiać. Quick Sort klasyczny jak i zmodyfikowany ma bardzo bliski czas działania aż do 52k elementów. Dla większych elementów zdecydowanie widać przewagę zmodyfikowanego sortowania. Przegrywa on jedynie w 4mln elementach tablicy dla której ma ponad 2,5 razy dłuższy czas niż standardowy. Czas dla Bucket Sort jest najgorszy i w każdym przypadku przegrywa z sortowaniem szybkim.

	32768	**65536**	**131072**	**262144**	**524288**	**1048576**	**2097152**	**4194304**
CZAS QS BEZ MOD	4	9	19	39	79	162,5	335,5	688,5
CZAS QS MOD	4	9	18,5	40	90	217	576	1751,5
CZAS BS	12	18	53	164	614	2342,5	9163	36105,5



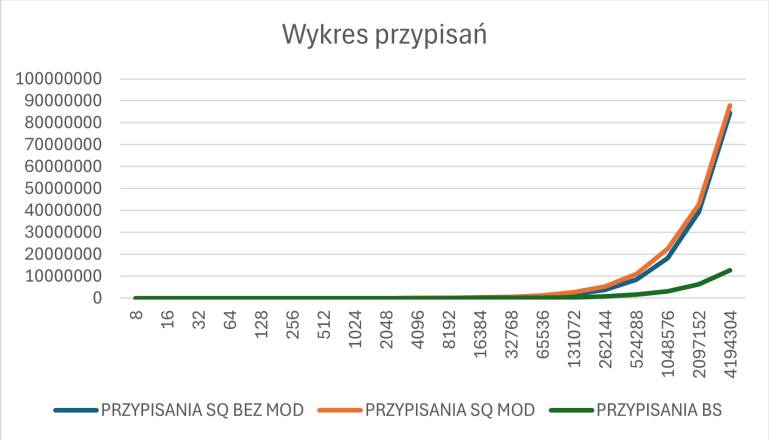
7.2.2 Porównania

W przypadku porównań widać zdecydowaną przewagę Bucket Sorta nad sortowaniami szybkimi. Najwięcej porównań wykonuje zmodyfikowana wersja Quick Sort, a nieco mniej od niej klasyczna wersja.



7.2.3 Przypisania

Tak samo jak porównania, Bucket Sort wykonuje również najmniejsza liczbę przypisań. Jednak tutaj w przypadku Quick Sortów różnice nie są aż tak zauważalne.



7.2.4 Wnioski

Pomimo najmniejszej liczby porównań i liczby przypisań, Bucket Sort ma najdłuższy czas wykonania. Jest to spowodowane użyciem list. Program musi

skakać po całej pamięci za każdym razem gdy chce się dostać do następnego elementu. Dlatego w danych implementacjach najwydajniejszy jest algorytm standardowy Quick Sort. Wykonuje się najszybciej, przy stosunkowo małej liczbie porównań oraz liczbie przypisań. Przy powtarzaniu eksperymentu należałoby poprawić implementacje listy i wtedy jest szansa na inne wyniki.

8 Podsumowanie

Pokazano zaimplementowane programy, a także przeanalizowano testy, dzięki którymi można porównać działanie algorytmów. Wynika z nich, że dla Radix Sort najbardziej opłaca się wykorzystać wersję o podstawie 256 bądź 1000, a pomiędzy Bucket Sort, a Quick Sort najlepiej wybrać standardową implementacje sortowania szybkiego. Dodatkowo została rozpatrzona implementacja własnej listy, na której przeprowadzono Insertion Sort, którego później użyto w Bucket Sort.