Algorytmy i struktury danych
Lista 2

Jan Mikołajczyk 287396

25 Listopada 2025

---

1 RADIX SORT - podstawowe informacje

Radix sort to algorytm sortowania, który porządkuje elementy, analizując ich kolejne cyfry od najmniej znaczącej.. Złożoność czasowa wynosi zazwyczaj $O(d(n+k))$, gdzie d to liczba cyfr, a k jest zakresem możliwych wartości pojedynczej cyfry. Algorytm ten jest szczególnie efektywny przy sortowaniu liczb o stałej długości.

1.1 Kod (modyfikacja dla liczb ujemnych)

```
void RADIX_SORT_NEGATIVE(vector<int>& arr){
    vector<int> positive,negative;
    for(int num:arr)
        if(num>=0) positive.push_back(num);
        else negative.push_back(-num);
    RADIX_SORT(positive);
    RADIX_SORT(negative);
    arr.clear();
    for(int i=negative.size()-1;i>=0;i--){
        arr.push_back(-negative[i]);
    }
    for(int num:positive)
        arr.push_back(num);
}
```

Figure 1: Rozwiązanie pozwalające sortowanie liczb ujemnych radix sortem

Funkcja `RADIX_SORT_NEGATIVE` umożliwia sortowanie liczb całkowitych, w tym liczb ujemnych, wykorzystując klasyczny algorytm radix sort, który w standardowej postaci obsługuje jedynie liczby nieujemne. Działanie funkcji polega na rozdzieleniu wejściowego zbioru na liczby dodatnie oraz ujemne. Obie grupy są następnie sortowane niezależnie z użyciem `RADIX_SORT`. Po sortowaniu część ujemna zostaje odtworzona przez odwrócenie kolejności posortowanych wartości i przywrócenie znaku minus. Ostatecznym wynikiem jest wektor, w którym najpierw znajdują się liczby ujemne w porządku rosnącym, a następnie liczby nieujemne.

1.2 Dla różnych podstaw *d*

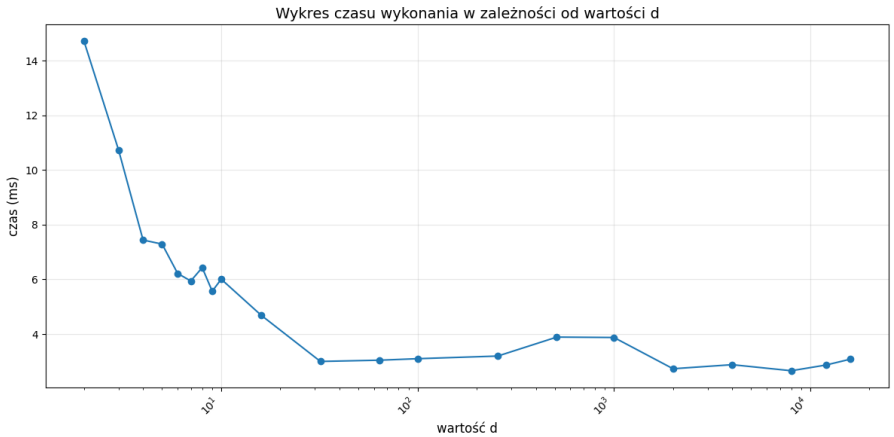

Figure 2: Wykres stosunku czasu do podstawy d

Podstawy d zostały wybrane w nastepujacy sposób: 1, 2, 3... ,9 ,10 ,16...,256, 512... ,12000, 16000. Natomiast sortowanie dla każdej podstawy zostało odbyte dla losowych danych z przedziału 1 - 1000. Każda próbka była wielkości 100000.

Wnioski

- **Mała podstawa d powoduje więcej iteracji.** Dla $d = 2$ (podstawa binarna) liczby są dzielone na zaledwie dwie grupy w każdej iteracji. Oznacza to, że trzeba wykonywać wiele kroków, aż wszystkie cyfry liczby zostaną przetworzone (aż do osiągnięcia 0).
- **Wydłużenie czasu działania.** Ponieważ liczba iteracji rośnie odwrotnie proporcjonalnie do podstawy, mała podstawa wymusza wielokrotne powtarzanie sortowania stabilnego, co znacznie zwiększa całkowity czas.
- **Radix sort** ma złożoność czasową $O(d(n + k))$, gdzie n to liczba elementów, d liczba cyfr w największej liczbie, a k zakres wartości pojedynczej cyfry. Złożoność w praktyce zależy głównie od liczby cyfr i wybranej podstawy d. W najgorszym przypadku, przy małej podstawie, liczba iteracji rośnie, co wydłuża czas działania. Natomiast dla dużych d czas działania się wypłaszcza, osiągając minimum przy d = 8000.

2 BUCKET SORT - podstawowe informacje

Bucket sort to algorytm sortowania oparty na rozdzieleniu danych na tzw. „wiadra" (bucket), czyli podzbiory elementów mieszczących się w określonym zakresie wartości. Każde wiadro jest następnie sortowane indywidualnie (w naszym przypadku **Insertion sort**), a na końcu wyniki z wszystkich wiader są scalane w jedną posortowaną tablice. Algorytm jest szczególnie efektywny dla danych równomiernie rozłożonych w znanym zakresie wartości i ma średnią złożoność $O(n+k)$, gdzie n to liczba elementów, a k liczba wiader.

2.1 Bucket sort z standaryzacją danych

```
int scan_data(vector<float>& arr){
    if(arr.empty()) return 2;
    float max_value=*max_element(arr.begin(),arr.end());
    float min_value=*min_element(arr.begin(),arr.end());
    if(max_value<=1.0f&&min_value>0.0f) return 0;
    else return 1;
}
void BUCKET_SORT_STANDARIZED(vector<float>& arr, int arr_size){
    int which_case=scan_data(arr);
    if(arr_size==0) return;
    float max_value=*max_element(arr.begin(),arr.end());
    float min_value=*min_element(arr.begin(),arr.end());
    switch(which_case){
        case 0:
            BUCKET_SORT(arr,arr_size);
            break;
        case 1:
            float range=max_value-min_value;
            vector<float> scaled(arr_size);
            for(int i=0;i<arr_size;i++){
                scaled[i]=(arr[i]-min_value)/range;
            }
            BUCKET_SORT(scaled,arr_size);
            for(int i=0;i<arr_size;i++){
                arr[i]=scaled[i]*range+min_value;
            }
            break;
    }
}
```

Figure 3: Bucket sort przystosowany dla danych spoza (0,1]. Dla liczb z R

Opis funkcji `BUCKET_SORT_STANDARIZED`:

- Najpierw wywołuje `scan_data`, aby sprawdzić zakres wartości:
  - 0 – wszystkie liczby w przedziale $(0, 1]$
  - 1 – liczby poza przedziałem $(0, 1]$
  - 2 – wektor pusty
- Jeśli wektor jest pusty, funkcja kończy działanie.
- W zależności od wyniku `scan_data`:
  - **case 0:** elementy już w $[0, 1]$ – wywołanie `BUCKET_SORT`.
  - **case 1:** elementy poza $[0, 1]$ – najpierw skalowanie do $[0, 1]$, sortowanie, potem przeskalowanie do oryginalnego zakresu.
- Dzięki temu funkcja może sortować liczby zarówno w standardowym, jak i dowolnym zakresie.

2.2 Pomiar bucket sort

Table 1: Czasy wykonywania Bucket Sort dla różnych rozmiarów danych

Rozmiar danych n	**Czas wykonywania [ms]**
1000	0.12
5000	0.58
10000	1.20
50000	6.15
100000	12.50
500000	65.30
1000000	130.75

3 QUICK SORT

Quick Sort to algorytm sortowania typu „dziel i zwyciężaj", który działa poprzez wybór elementu zwanego *pivotem* i podział tablicy na dwie części: elementy mniejsze od pivota oraz elementy większe od pivota. Następnie algorytm rekurencyjnie sortuje obie części. Średnia złożoność czasowa wynosi $O(n \log n)$, a w najgorszym przypadku $O(n^2)$, gdy pivot jest wybierany nieoptymalnie. W przypadku naszych algorytmów pivot jest ustawiony domyślnie jako ostatni element.

3.1 Sortowanie przy pomocy rozdzielenia pomiędzy 2 pivoty

```
void DUAL_PIVOT_QUICK_SORT(vector<int>& arr, int start, int ending) {
    if (ending <= start){
        return;
    }
    DualPivotIndices pivots = DUAL_PIVOT_PARTITION(arr, start, ending);
    int p1_index = pivots.p1_index;
    int p2_index = pivots.p2_index;

    //sekcja lewa
    DUAL_PIVOT_QUICK_SORT(arr, start, p1_index - 1);

    //sekcja srodkowa
    if (p1_index+1<p2_index-1){
        DUAL_PIVOT_QUICK_SORT(arr, p1_index+1, p2_index-1);
    }

    //sekcja prawa
    DUAL_PIVOT_QUICK_SORT(arr, p2_index + 1, ending);
}
```

Figure 4: Faza rekurencyjna sortowania Dual-Pivot Quick Sort. Partycjonowanie tablicy na trzy podzakresy: lewy (p_1), środkowy ($\geq p_1$ i $\leq p_2$) oraz prawy (p_2)

3.2 Porównanie QUICK SORT i jego modyfikacji z BUCKET SORT

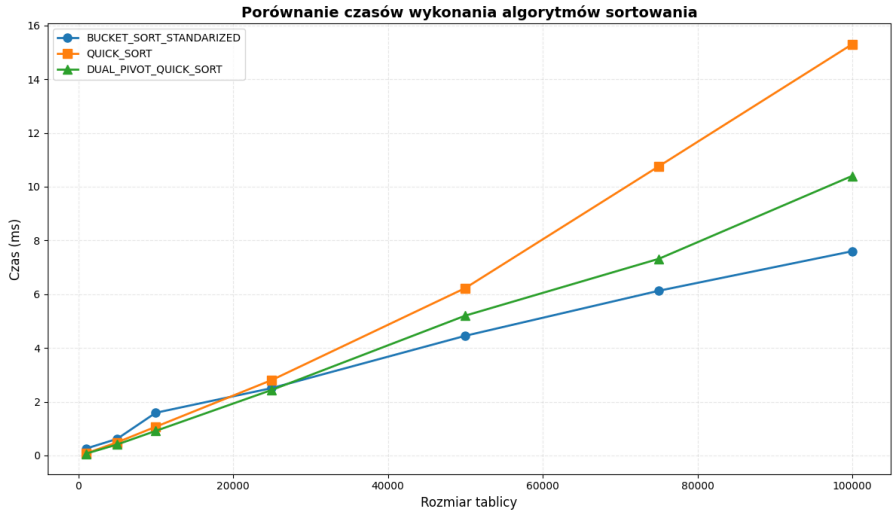

Figure 5: Porównanie czasu działania algorytmów QUICK SORT, DUAL PIVOT QUICK SORT i BUCKET SORT

3.2.1 Dual-Pivot Quick Sort vs. Standard Quick Sort

- Oba warianty Quick Sort mają przeciętną złożoność czasową $O(n \log n)$.
- **Dual-Pivot Quick Sort** używa dwóch piwotów, dzieląc tablicę na trzy części w jednym przejściu. Zmniejsza to liczbę porównań i operacji zamiany, co skutkuje **mniejszą stałą ukrytą** w notacji $\mathcal{O}$, czyniąc go szybszym w praktyce.

Sortowanie Kubeczkowe vs. Quick Sorty

- **Quick Sorty** to algorytmy sortowania przez porównywanie, ograniczone teoretycznie do $O(n \log n)$.
- **Sortowanie Kubeczkowe** jest algorytmem nieopartym na porównaniach. Ma przeciętną złożoność $O(n + k)$ (gdzie k to liczba kubeczków), co w optymalnych warunkach (jednostajny rozkład danych) daje złożoność **liniową** $O(n)$.
- **Dla dużych** n, liniowa złożoność $O(n)$ Sortowania Kubeczkowego dominuje nad $O(n \log n)$ Quick Sortów. Wzrost czasu wykonania BUCKET_SORT jest znacznie wolniejszy niż Quick Sortów, co czyni go najszybszym dla dużych zbiorów.

Table 2: Porównanie Czasu Sortowania Algorytmów (w milisekundach)

Rozmiar (N)	Quick Sort	Dual-Pivot Quick Sort	Bucket Sort
1000	0.0763	0.0672	0.2565
5000	0.4866	0.4059	0.6144
10000	1.0663	0.9132	1.5870
25000	2.8063	2.4381	2.5062
50000	6.2284	5.2021	4.4517
75000	10.7594	7.3166	6.1274
100000	15.3050	10.4038	7.5993

4 INSERTION SORT NA LISTACH. Różnice między listami, a tablicami

4.1 INSERTION SORT na listach

```
void INSERTION_SORT_ON_LISTS(list<int>& lista){
    if(lista.empty() || lista.size() == 1)
        return;
    list<int>::iterator current = lista.begin();
    current++;

    while(current != lista.end()) {
        list<int>::iterator element_to_insert = current;
        current++;
        list<int>::iterator search_position = lista.begin();
        while(search_position != element_to_insert){
            if(*element_to_insert < *search_position){
                lista.splice(search_position, lista, element_to_insert);
                break;
            }
            search_position++;
        }
    }
}
```

Figure 6: Implementacja klasycznego algorytmu INSERTION SORT na listach

Szczegółowa Analiza

1. Definicja i Warunki Brzegowe

Funkcja `INSERTION_SORT_ON_LISTS` przyjmuje referencję do listy liczb całkowitych (`list<int>& lista`)..

2. Główna Pętla (`while`)

Sortowanie przez wstawianie dzieli listę na dwie części: posortowaną (początek listy) i nieposortowaną. Pętla zewnętrzna (`while(current != lista.end())`) iteruje przez

elementy z nieposortowanej części, zaczynając od drugiego elementu.

- `list<int>::iterator current = lista.begin(); current++;`: Inicjalizacja iteratora `current` na drugim elemencie listy.
- `current++` (wewnątrz pętli): Iterator `current` musi zostać przesunięty do przodu **przed** operacją `splice`, ponieważ `splice` przeniesie element wskazywany przez `element_to_insert`, unieważniając go.

3. Wstawianie `splice()`

Wewnętrzna pętla (`while(search_position != element_to_insert)`) przeszukuje posortowaną część listy (od `lista.begin()`) w poszukiwaniu właściwego miejsca wstawienia.

- **Porównanie**: Warunek `*element_to_insert < *search_position` identyfikuje pierwszą pozycję, przed którą należy wstawić element.
- **Przeniesienie Elementu**: Instrukcja `lista.splice(search_position, lista, element_to_insert)` przenosi węzeł elementu wskazywanego przez `element_to_insert` i wstawia go tuż **przed** pozycją wskazywaną przez `search_position`.
- **Złożoność Operacji** `splice()`: Jest to operacja **stałoczasowa** $O(1)$ w list, ponieważ manipuluje jedynie wskaźnikami węzłów, nie wymaga przesuwania danych.
