Sprawozdanie AiSD Lista 2

Michał Kasjaniuk, 287392

November 2025

1 Wprowadzenie

Celem niniejszego sprawozdania jest implementacja, analiza oraz empiryczne porównanie wydajności trzech algorytmów sortowania: Quick Sort, Radix Sort oraz Bucket Sort.

2 Opis algorytmów

W tej sekcji zajmijmy się przedstawieniem algorytmów wraz z ich modyfikacjami.

2.1 Quick Sort

Quick Sort to jeden z najpopularniejszych i najefektywniejszych algorytmów sortowania ogólnego przeznaczenia. Opiera się na strategii "dziel i rządź". W przeciętnym przypadku złożonośc obliczeniowa wynosi $O(N \log N)$. Natomiast w najgorszym przypadku osiąga $O(N^2)$.

```
int PARTITION(vector<int>& A, int poczatek, int koniec) {
    int pivot = A[koniec];
    int i = poczatek - 1;

    for (int j = poczatek; j < koniec; j++) {
        if (A[j] <= pivot) {
            i++;
            swap(A[i], A[j]);
        }
    }
    swap(A[i + 1], A[koniec]);
    return i + 1;
}
```

Niniejszy kod prezentuje implementację funkcji **PARTITION** (partycjonującej), która jest podstawą algorytmu QuickSort. Jej zadaniem jest podział fragmentu tablicy względem elementu osiowego (piwota) na dwie podtablice: elementów mniejszych lub równych oraz elementów większych. Funkcja ta zwraca indeks, pod którym piwot zostaje umieszczony na swojej ostatecznej pozycji, co stanowi punkt podziału dla kolejnych wywołań rekurencyjnych algorytmu QuickSort.

Wychodząc poza klasyczny wariant algorytmu, zaimplementowano jego modyfikację Dual-Pivot Quick Sort. Głównym celem tego usprawnienia jest zwiększenie wydajności poprzez lepsze wykorzystanie pamięci podręcznej procesora oraz redukcję głębokości stosu rekursji. W przeciwieństwie do wersji standardowej, modyfikacja ta opiera się na jednoczesnym użyciu dwóch elementów osiowych, co pozwala na podzielenie tablicy na trzy, a nie na dwie podtablice w każdym kroku partycjonowania.

```
void PARTITION_DUAL(vector<int>& A, int p, int k, int& lp, int& rp) {
    if (A[p] > A[k]) {
        swap(A[p], A[k]);
    }

    int pivot1 = A[p];
    int pivot2 = A[k];

    int i = p + 1;
    int gt = k - 1;
```

```
    int j = p + 1;

    while (j <= gt) {
        if (A[j] < pivot1) {
            swap(A[j], A[i]);
            i++;
            j++;
        }
        else if (A[j] > pivot2) {
            swap(A[j], A[gt]);
            gt--;
        }
        else {
            j++;
        }
    }

    i--;
    gt++;
    swap(A[p], A[i]);
    swap(A[k], A[gt]);

    lp = i;
    rp = gt;
}
```

2.2 Radix Sort

Radix Sort to algorytm sortowania nieoparty na bezpośrednim porównywaniu elementów. Jego działanie polega na wielokrotnym sortowaniu elementów względem kolejnych pozycji (cyfr) w systemie o określonej podstawie d. Złożoność obliczeniowa to $O(k \cdot (N + d))$, gdzie N to liczba elementów, d to podstawa systemu (np. 10), a k to liczba cyfr w najdłuższej liczbie. W praktyce, przy optymalnym doborze podstawy d (np. 256), algorytm działa w czasie liniowym $O(N)$. Algorytm sortuje liczby pozycja po pozycji (od cyfry najmniej znaczącej), używając stabilnego algorytmu sortującego (zazwyczaj Counting Sort).

1. **Rozłożenie:** W każdej pętli elementy są rozrzucane do kubełków (zgodnie z aktualnie sortowaną cyfrą).
2. **Stabilność:** Kubełki są łączone, co musi odbyć się stabilnie, aby zachować porządek ustalony przez poprzednie, mniej znaczące cyfry.
3. **Iteracja:** Proces jest powtarzany dla każdej kolejnej pozycji (dziesiątek, setek, itd.).

```
void RADIX_SORT(vector<int>& A, int d, int k) {
    long long exp = 1;
    for (int i = 1; i <= k; i++) {
        COUNTING_SORT(A, d, exp);
        exp *= d;
    }
}
```

Poniższy algorytm stanowi modyfikację klasycznego Radix Sort, która pozwala na poprawną obsługę liczb całkowitych, wliczając w to liczby ujemne. Różnica od wersji klasycznej polega na zastosowaniu Metody Przesunięcia: Algorytm najpierw znajduje najmniejszą wartość ujemną (minVal) w tablicy, a następnie dodaje jej wartość bezwzględną do wszystkich elementów. To przesuwa cały zakres liczb tak, że najmniejszy element staje się zerem, a wszystkie liczby są nieujemne. Na koniec, po posortowaniu przez standardowy Radix Sort, algorytm cofa przesunięcie, przywracając oryginalne wartości i ich prawidłową, rosnącą kolejność.

```
void RADIX_SORT_W_NEG(vector<int>& A, int d, int k) {
    if (A.empty()) return;

    int minVal = A[0];
    for (int i = 1; i < A.size(); i++) {
        if (A[i] < minVal) {
```

```
            minVal = A[i];
        }
    }

    for (int i = 0; i < A.size(); i++) {
        A[i] = A[i] - minVal;
    }


    RADIX_SORT(A, d, k);


    for (int i = 0; i < A.size(); i++) {
        A[i] = A[i] + minVal;
    }
}
```

2.3 Bucket Sort

Bucket Sort to algorytm sortowania oparty na zasadzie dystrybucji elementów do mniejszych pojemników, których liczba wynosi k (zwanych kubełkami).

Następnie algorytm przypisuje liczby z sortowanej tablicy do odpowiednich kubełków na podstawie ich wartości. Proces sortowania przebiega w trzech głównych etapach:

1. Elementy są rozdzielane do k kubełków.
2. Zawartość każdego kubełka jest sortowana niezależnie (np. przy użyciu algorytmu Bubble Sort).
3. Następnie zawartości kolejnych kubełków są przepisywane, tworząc posortowaną tablicę wynikową.

W optymistycznych warunkach (równomierny rozkład danych) złożonosc obliczeniowa algorytmu osiąga złożoność liniową $\mathcal{O}(\mathbf{n})$. W przypadku pesymistycznym, gdy wszystkie elementy trafią do jednego kubełka, złożoność tego algorytmu wynosi $\mathcal{O}(\mathbf{n^2})$.

```
void BucketSort(vector<float>& A) {
    int n = A.size();
    vector<vector<float>> b(n);

    for (int i = 0; i < n; i++) {
        int bi = n * A[i];

        if (bi >= n) bi = n - 1;

        b[bi].push_back(A[i]);
    }

    for (int i = 0; i < n; i++) {
        INSERTION_SORT(b[i]);
    }

    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < b[i].size(); j++) {
            A[index++] = b[i][j];
        }
    }
}
```

Poniższy algorytm stanowi uniwersalną modyfikację Bucket Sorta, mającą na celu usunięcie ograniczenia, które narzuca konieczność pracy w zakresie $[0, 1)$. Działanie opiera się na procesie **normalizacji i skalowania** danych:

1. Analiza zakresu: Najpierw oblicza się minimalną i maksymalną wartość w tablicy wejściowej.

2. Skalowanie do indeksu: Następnie każda liczba jest mapowana na indeks kubełka za pomocą wzoru:

$$\text{index} = \left\lfloor \frac{A[i] - \min}{\max - \min} \cdot N \right\rfloor$$

Modyfikacja ta pozwala na poprawne sortowanie dowolnych liczb rzeczywistych (ujemnych, dużych), zachowując przy tym oczekiwaną średnią złożoność liniową $\mathcal{O}(\mathbf{n})$, kosztem minimalnego narzutu obliczeniowego na wstępne znalezienie granic zakresu.

```
void BucketSort_Modfied(vector<float>& A) {
    int n = A.size();
    if (n <= 1) return;

    float minVal = A[0];
    float maxVal = A[0];
    for (float x : A) {
        if (x < minVal) minVal = x;
        if (x > maxVal) maxVal = x;
    }
    float range = maxVal - minVal;
    if (range == 0) return;
    vector<vector<float>> B(n);


    for (int i = 0; i < n; i++) {
        int index = (int)((A[i] - minVal) / range * n);

        if (index >= n) {
            index = n - 1;
        }

        B[index].push_back(A[i]);
    }

    for (int j = 0; j < n; j++) {

        if (!B[j].empty()) {
            INSERTION_SORT(B[j]);
        }
    }

    int k = 0;
    for (int j = 0; j < n; j++) {
        for (float val : B[j]) {
            A[k++] = val;
        }
    }
}
```

3 Porównanie algorytmów

3.1 Porównanie działania Radix Sort dla róznych podstaw d.

Podstawa (d)	**Liczba Przebiegów (k)**	**Czas [ms]**
2	29	426
10	10	148
100	6	89
256	5	76
1024	4	59

Table 1: Porównanie wydajności Radix Sort dla różnych podstaw systemu liczbowego (d). Testy dla $n = 1\,000\,000$ losowych liczb.

Wnioski z testów Radix Sort

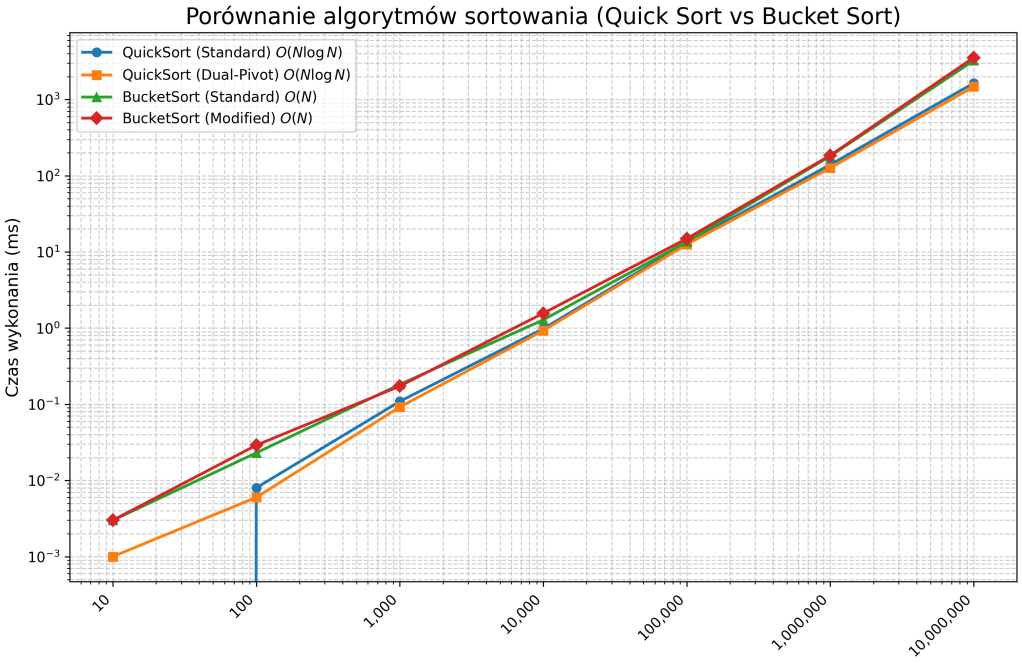
Analiza wydajności Radix Sort dla różnych podstaw d wyraźnie potwierdza, że czas wykonania algorytmu zależy głównie od liczby wymaganych przebiegów pętli (k).

- **Kluczowa rola k:** Najważniejszym czynnikiem wpływającym na czas wykonania jest liczba przebiegów pętli (**k**). Wzrost podstawy d powoduje logarytmiczny spadek k, co jest głównym źródłem zysku czasowego.
- **Ekstremalny koszt ($d = 2$):** Sortowanie binarne ($d = 2$) było najwolniejsze (**426 ms**) z powodu konieczności wykonania aż **29** pełnych cykli sortowania (tyle, ile bitów potrzeba na zapisanie zakresu liczb). Jest to bezpośredni koszt wielokrotnego przepisywania tablicy.
- **Optimum Wydajności ($d \geq 256$):** Najlepsze czasy uzyskano dla dużych podstaw: (**d = 1024** - 59 ms oraz **d = 256** - 76 ms). Osiągnięto w ten sposób najlepszy kompromis: minimalizację k do 4-5 przebiegów przy akceptowalnym koszcie pamięci dla tablicy liczników.
- **Wniosek ogólny:** Dla maksymalizacji wydajności należy zawsze dążyć do minimalizacji liczby przejść k poprzez wybór największej możliwej podstawy d.

3.2 Porównanie algorytmu Quick Sort i Bucket Sort.

Algorytm	**Czas wykonania [ms]**
Quick Sort	139.678
Quick Sort Modfieid (Dual-Pivot)	126.557
Bucket Sort (Standard)	180.994
Bucket Sort Modified (Uniwersalny / Mod.)	184.849

Table 2: Porównanie wydajności sortowania dla $N = 1\,000\,000$.



Wnioski końcowe (Porównanie Quick Sort i Bucket Sort)

Analiza wyników dla $n = 10\,000\,000$ elementów dostarcza następujących kluczowych konkluzji:

1. Anomalia implementacyjna: Algorytm Bucket Sort, mimo teoretycznej złożoności liniowej ($\mathcal{O}(\mathbf{N})$), okazał się w teście ponad dwukrotnie wolniejszy od Quick Sorta ($\mathcal{O}(\mathbf{N} \log \mathbf{N})$).

2. Wariant **Quick Sort Dual-Pivot** był konsekwentnie najszybszy w całym zestawieniu. Jest to dowód na to, że techniki optymalizacji (dwa piwoty, redukcja rekursji) w algorytmach operujących *in-place* dają lepszy rezultat końcowy niż algorytmy o niższej złożoności, ale wysokim koszcie zarządzania pamięcią dynamiczną.