Analiza Algorytmów Sortowania

Ksawery Józefowski
Politechnika Wrocławska

December 4, 2024

Contents

1 Wstep

W ramach Listy 2 zaimplementowano i przeanalizowano 4 algorytmy sortowania: *Insertion Sort*, *Bucket Sort*, *Quick Sort* oraz *Radix Sort*. Każdy z algorytmów został zmodyfikowany: *Quick Sort* dzieli tablice na 3 cześci, *Bucket Sort* działa nie tylko na przedziale [0-1], *Radix Sort* sortuje również liczby ujemne, a *Insertion Sort* działa na listach. Celem projektu jest porównanie *Bucket Sorta* z *Quick Sortem* jak i również porównanie *Radix Sorta* dla różnych podstaw.

2 Fragmenty Kodów

Poniżej przedstawiono najciekawsze fragmenty kodu dla wybranych algorytmów.

2.1 Radix Sort

Algorytm *Radix Sort* to nieporównawczy algorytm sortujacy, który działa poprzez sortowanie liczb na podstawie ich cyfr. Wykorzystuje on stabilny algorytm sortujacy, taki jak *Counting Sort*, do sortowania elementów na podstawie poszczególnych cyfr. Proces sortowania w *Radix Sort* jest iteracyjny i przebiega od najmniej znaczacej cyfry do najbardziej znaczacej.

- Algorytm najpierw identyfikuje najwieksza liczbe w tablicy, co pozwala określić liczbe iteracji wymaganych do posortowania na podstawie najbardziej znaczacej cyfry.
- Nastepnie wykonuje *Counting Sort* dla każdej cyfry, zaczynajac od najmniej znaczacej.
- W każdej iteracji elementy sa sortowane w oparciu o jedna cyfre, przy zachowaniu stabilności algorytmu, co pozwala zachować poprawna kolejność elementów miedzy iteracjami.

Dzieki tej metodzie, *Radix Sort* jest w stanie sortować liczby całkowite w czasie $O(b \cdot (n + d))$, gdzie:

- n to liczba elementów w tablicy,
- b to liczba cyfr w najwiekszej liczbie,
- d to podstawa systemu liczbowego (np. 10 dla systemu dziesietnego).

2.2 Radix Sort Negative

Algorytm *Radix Sort Negative* jest modyfikacja klasycznego algorytmu *Radix Sort*, która została zaprojektowana z myśla o sortowaniu liczb zarówno dodatnich, jak i ujemnych. Zmiana polega na rozszerzeniu zakresu zliczania, aby mogły być obsługiwane liczby ujemne.

- Zamiast tylko zliczać cyfry liczb dodatnich, algorytm rozszerza zakres zliczania, aby uwzględnić zarówno liczby dodatnie, jak i ujemne.
- Proces sortowania dla liczb ujemnych wymaga przesuniecia cyfr, aby odpowiednio obsłużyć te liczby.
- Algorytm wykonuje iteracyjne sortowanie przez cyfry, podobnie jak w przypadku klasycznego *Radix Sort*, ale z uwzglednieniem liczb ujemnych.

Dzieki tej modyfikacji, *Radix Sort Negative* jest w stanie poprawnie sortować liczby całkowite, niezależnie od ich znaku, w czasie podobnym do *Radix Sort*, tj. $O(d \cdot (n + b))$. Jest to efektywny sposób sortowania liczb zarówno dodatnich, jak i ujemnych, przy zachowaniu stabilności algorytmu.

Wersja *Radix Sort Negative* może być użyteczna w przypadkach, gdy musimy posortować zbiory danych zawierajace zarówno liczby dodatnie, jak i ujemne, bez konieczności oddzielnego przetwarzania obu typów liczb.

```
void radixSort_negative(int arr[], int n, int base, unsigned
    long long& comparisons, unsigned long long& assignments) {
    const int size = base * 2 - 1;
    auto output = new int[n];

    for (int pos = 1; ; pos *= base) {
        int counter[size]{};

        bool done = true;
        for (int i = 0; i < n; i++) {
            int d = arr[i] / pos;
            assignments++;
            ++counter[d % base + size / 2];
            done &= (d == 0);
        }
        if (done)
            break;

        for (int i = 1; i < size; i++)
            counter[i] += counter[i - 1];

        for (int i = n; i-- > 0; ) {
            output[--counter[arr[i] / pos % base + size / 2]]
                = arr[i];
            assignments++;
        }

        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
            assignments++;
        }
    }

    delete[] output;
}
```

Listing 1: Radix z obsługa ujemnych liczb

2.3 Bucket Sort

Algorytm *Bucket Sort* jest algorytmem sortujacym opartym na rozdzieleniu elementów na różne "wiadra" (ang. *buckets*), a nastepnie posortowaniu ich wewnetrznie, zazwyczaj za pomoca algorytmu *Insertion Sort*. Każde wiadro zawiera elementy, które sa "bliskie" siebie w kontekście ich wartości.

- Na poczatku algorytm znajduje najmniejsza i najwieksza wartość w zbiorze, co pozwala na określenie zakresu wartości.
- Z zakresu tego obliczany jest rozmiar jednego wiadra, który jest równy różnicy miedzy najwieksza a najmniejsza wartościa podzielona przez liczbe elementów.
- Nastepnie każdy element jest przypisany do odpowiedniego wiadra na podstawie jego wartości.
- Każde wiadro jest nastepnie sortowane indywidualnie, a wynikowe posortowane elementy sa łaczone w jeden posortowany zbiór.

Algorytm *Bucket Sort* może działać w czasie $O(n + k)$, gdzie n to liczba elementów, a k to liczba wiader, jednakże jego wydajność zależy od jakości rozdzielenia elementów na wiadra oraz od zastosowanego algorytmu wewnetrznego do sortowania w wiadrach.

2.4 Bucket Sort Mod

Bucket Sort Mod jest zmodyfikowana wersja klasycznego algorytmu *Bucket Sort*, która obsługuje przypadki, w których elementy nie mieszcza sie w standardowym zakresie 0-1.

- Algorytm zaczyna od obliczenia minimalnej i maksymalnej wartości w zbiorze danych, jak w klasycznym *Bucket Sort*.
- Na podstawie tej informacji obliczany jest zakres wartości, a nastepnie liczba wiader, do których beda przypisane elementy. W tym przypadku zakres może być znacznie szerszy.
- Kluczowa modyfikacja w tej wersji algorytmu jest możliwość pracy z liczbami spoza przedziału 0-1, co oznacza, że elementy sa odpowiednio mapowane na wiadra w oparciu o ich wartość unormalizowane do zakresu 0-1.

- Każdy element jest przypisany do odpowiedniego wiadra, a nastepnie wiadra sa sortowane, np. za pomoca *Insertion Sort*. Po posortowaniu, elementy sa scalane w jedno posortowane zestawienie.

Modyfikacja ta pozwala na sortowanie liczb z dowolnego zakresu, nie ograniczajac sie do przedziału 0-1. Złożoność czasowa algorytmu jest podobna do klasycznej wersji i wynosi $O(n + k)$, gdzie n to liczba elementów, a k to liczba wiader. Poprawiona wersja jest bardziej elastyczna i może być używana do sortowania liczb o dowolnym zakresie wartości.

```
void bucketSort_Mod(double arr[], int n) {
    if (n <= 0) return;

    double minVal = arr[0], maxVal = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] < minVal) minVal = arr[i];
        if (arr[i] > maxVal) maxVal = arr[i];
    }

    int numBuckets = n;
    double bucketRange = (maxVal - minVal);

    Node** buckets = new Node*[numBuckets]();

    for (int i = 0; i < n; i++) {
        int bucketIndex = (arr[i] - minVal) / (maxVal -
            minVal) * n;
        if (bucketIndex == numBuckets) bucketIndex--;
        insert(buckets[bucketIndex], arr[i]);
    }

    for (int i = 0; i < numBuckets; i++) {
        insertionSort(buckets[i]);
    }

    int index = 0;
    for (int i = 0; i < numBuckets; i++) {
        Node* current = buckets[i];
        while (current != nullptr) {
            arr[index++] = current->data;
            Node* temp = current;
            current = current->next;
            delete temp;
        }
    }
    delete[] buckets;
}
```

Listing 2: Bucket sort z obsługa liczb spoza zakresu 0-1

2.5 QuickSort

Algorytm *QuickSort* jest algorytmem sortujacym działajacym na zasadzie "dziel i zwyciężaj" (ang. *divide and conquer*). Polega na wybieraniu elementu z tablicy jako tzw. "pivot" (punkt odniesienia), a nastepnie przekształca tablice tak, aby wszystkie elementy mniejsze od pivota znajdowały sie po jednej stronie, a wieksze po drugiej. Nastepnie algorytm rekurencyjnie sortuje obie cześci.

- Na poczatku wybierany jest element pivot (zwykle ostatni element tablicy).
- Nastepnie wykonywana jest funkcja *partition*, która reorganizuje elementy w taki sposób, że elementy mniejsze od pivota znajduja sie po lewej stronie, a elementy wieksze po prawej.
- Po dokonaniu podziału algorytm rekurencyjnie sortuje dwie cześci tablicy, które powstały po podziale.
- Proces powtarza sie, aż tablica zostanie w pełni posortowana.

Złożoność czasowa algorytmu *QuickSort* w przypadku średnim to $O(n \log n)$, gdzie n to liczba elementów. W najgorszym przypadku (np. gdy tablica jest już posortowana lub odwrotnie posortowana) złożoność wynosi $O(n^2)$. Jednak dzieki losowemu doborowi pivota lub jego dobremu wyborowi (np. mediana trzech elementów), algorytm zazwyczaj działa efektywnie.

2.6 Dual-Pivot QuickSort

Dual-Pivot QuickSort jest zmodyfikowana wersja algorytmu *QuickSort*, która używa dwóch pivotów zamiast jednego. Ta zmiana ma na celu poprawienie efektywności algorytmu w niektórych przypadkach.

- W tej wersji wybierane sa dwa pivoty: jeden na poczatku tablicy, a drugi na jej końcu.
- Funkcja *dualPivotPartition* organizuje tablice w taki sposób, że elementy mniejsze niż pierwszy pivot znajduja sie po jego lewej stronie, elementy wieksze niż drugi pivot po jego prawej, a miedzy nimi znajduja sie elementy pomiedzy tymi dwoma pivotami.
- Nastepnie algorytm rekurencyjnie sortuje trzy cześci tablicy: przed pierwszym pivotem, pomiedzy pivotami, oraz po drugim pivotie.

Złożoność czasowa algorytmu *Dual-Pivot QuickSort* w przypadku średnim to również $O(n \log n)$.

```
void dualPivotPartition(double arr[], int low, int high, int&
    lp, int& rp) {
    if (arr[low] > arr[high]) {
        swap(arr[low], arr[high]);
    }

    int pivot1 = arr[low];
    int pivot2 = arr[high];

    int i = low + 1, lt = low + 1, gt = high - 1;

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

    lt--;
    gt++;

    swap(arr[low], arr[lt]);
    swap(arr[high], arr[gt]);

    lp = lt;
    rp = gt;
}

void dualPivotQuickSort(double arr[], int low, int high) {
    if (low < high) {
        int lp, rp;
        dualPivotPartition(arr, low, high, lp, rp);

        dualPivotQuickSort(arr, low, lp - 1);
        dualPivotQuickSort(arr, lp + 1, rp - 1);
        dualPivotQuickSort(arr, rp + 1, high);
    }
}
```

Listing 3: Quick Sort z 2 pivotami/ dzieleniem na 3 cześci

2.7 Insertion Sort dla list

Algorytm *Insertion Sort* jest algorytmem sortujacym, który działa na zasadzie iteracyjnego wstawiania elementów do już posortowanej cześci danych. W przypadku listy, proces ten polega na wstawianiu elementów do posortowanej listy w odpowiednim miejscu, tak aby lista pozostała uporzadkowana.

- Na poczatku lista jest traktowana jako nieposortowana, a elementy beda wstawiane w odpowiednie miejsce w posortowanej cześci listy.
- Dla każdego elementu z nieposortowanej cześci listy, algorytm porównuje go z elementami posortowanej cześci i wstawia go w odpowiednie miejsce.
- Proces ten jest iteracyjny: elementy sa wstawiane w taki sposób, że lista jest stopniowo posortowana.

2.7.1 Opis implementacji

2.7.2 Struktura wezła

Algorytm działa na liście, gdzie każdy wezeł zawiera dane oraz wskaźnik na nastepny wezeł. Struktura wezła jest zdefiniowana jako:

```
1   struct Node {
2       double data;
3       Node* next;
4   };
```

2.7.3 Funkcja insert

Funkcja dodaje nowy wezeł na poczatek listy.

```
1   void insert(Node*& head, double value) {
2       Node* newNode = new Node{value, head};
3       head = newNode;
4   }
```

2.7.4 Funkcja insertionSort

Funkcja wykonuje sortowanie listy przy użyciu algorytmu *Insertion Sort*. Działa to w ten sposób, że dla każdego elementu z nieposortowanej cześci listy, element ten jest wstawiany w odpowiednie miejsce w posortowanej cześci listy.

```
1   void insertionSort(Node*& head) {
2       if (!head) return;
3
4       Node* sorted = nullptr;
5
6       while (head) {
7           Node* current = head;
8           head = head->next;
9
10          if (!sorted || sorted->data >= current->data) {
11              current->next = sorted;
12              sorted = current;
13          } else {
14              Node* temp = sorted;
15              while (temp->next && temp->next->data <
                    current->data) {
16                  temp = temp->next;
17              }
18              current->next = temp->next;
19              temp->next = current;
20          }
21      }
22      head = sorted;
23  }
```

2.7.5 Opis działania algorytmu

Algorytm działa w sposób podobny do klasycznego *Insertion Sort*, ale zamiast tablicy, operuje na liście. Działa to w nastepujacy sposób:

- Na poczatku lista jest traktowana jako nieposortowana, a posortowana cześć jest pusta.
- Iteracyjnie, dla każdego elementu z nieposortowanej cześci listy, algorytm porównuje go z elementami posortowanej cześci i wstawia go na odpowiednia pozycje.
- Proces kończy sie, gdy wszystkie elementy zostały przeniesione do posortowanej cześci listy.

2.7.6 Złożoność czasowa

Złożoność czasowa algorytmu *Insertion Sort* w przypadku listy wynosi $O(n^2)$ w najgorszym przypadku, gdzie n to liczba elementów w liście. W przypadku najlepszym, gdy lista jest już posortowana, złożoność wynosi $O(n)$.

3 Analiza i Wyniki

Porównujac Bucket i Quick Sort zliczono liczbe operacji i czas potrzebny do posortowania tablic o rozmiarach 10, 1000, 10000, 100000.

3.1 Tabela Wyników

Rozmiar danych	Algorytm	Porównania	Przypisania	Czas(s)
10	Quick Sort	9	51	$0.12\mu s$
10	Dual Pivot Quick Sort	13	77	$0.11\mu s$
10	Bucket Sort Mod	42	93	$3\mu s$
1000	Quick Sort	4237	15379	$89\mu s$
1000	Dual Pivot Quick Sort	3924	15670	$81\mu s$
1000	Bucket Sort Mod	4457	8897	$101\mu s$
10000	Quick Sort	75542	253282	$1249\mu s$
10000	Dual Pivot Quick Sort	64332	230516	$1135\mu s$
10000	Bucket Sort Mod	45087	88105	$1312\mu s$
100000	Quick Sort	962694	3185786	$15985\mu s$
100000	Dual Pivot Quick Sort	812595	2816001	$13475\mu s$
100000	Bucket Sort Mod	431233	831259	$16678\mu s$

Table 1: Porównanie liczby operacji dla różnych rozmiarów danych

3.2 Wykresy Wyników

Poniżej przedstawiono wykresy liczby porównań, przypisań i czasów dla Bucket i Quick Sorta w zależności od rozmiaru danych.

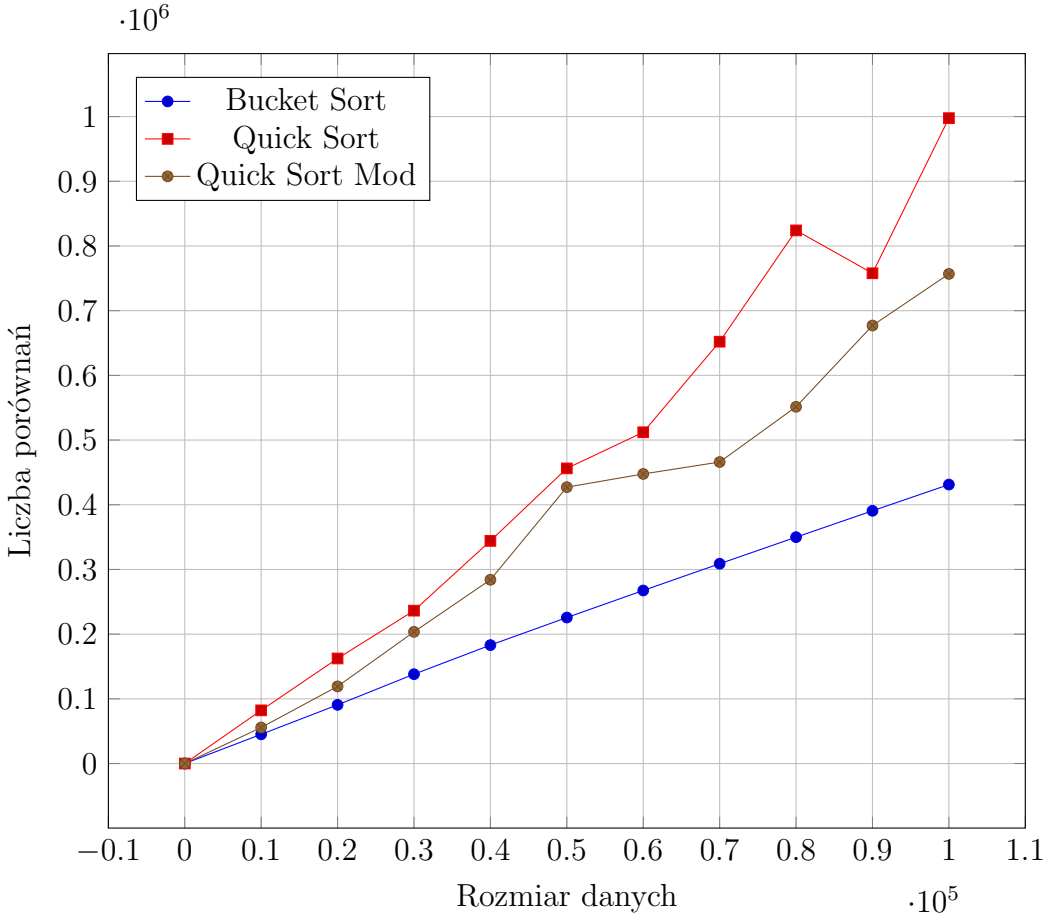

Figure 1: Liczba porównań w zależności od rozmiaru danych

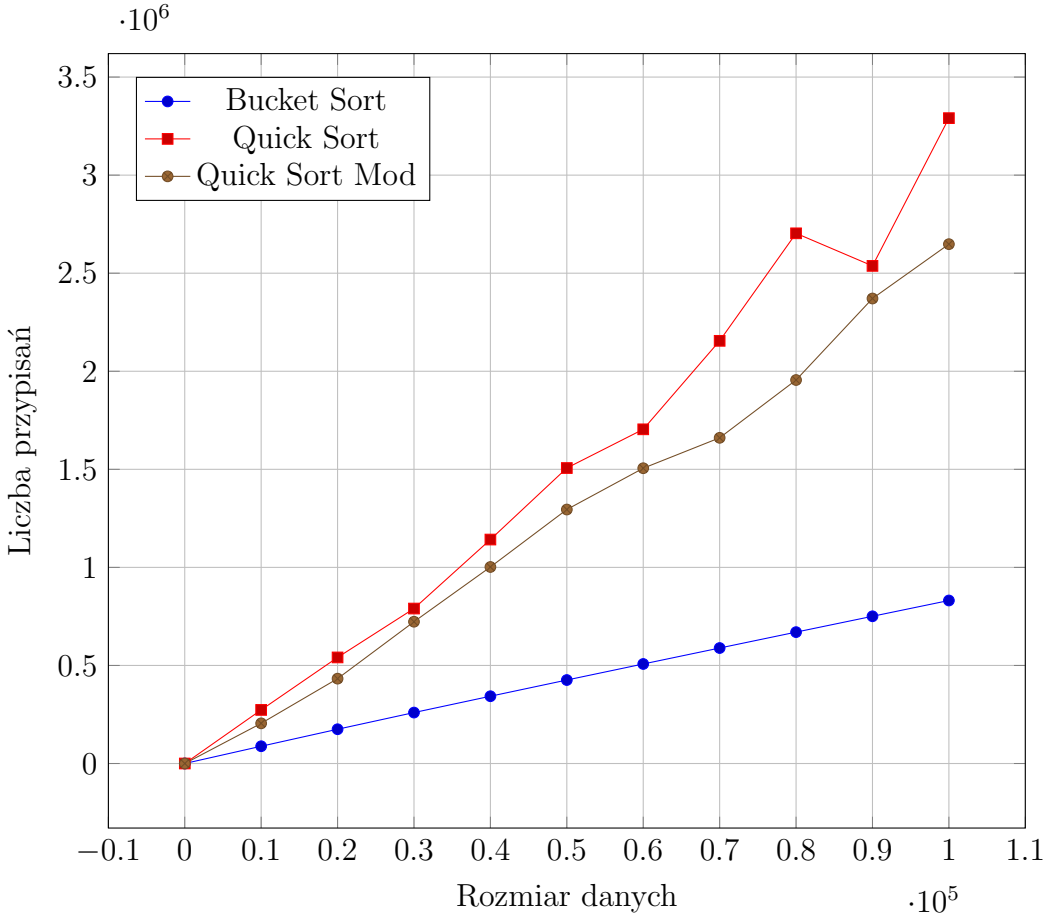

Figure 2: Liczba przypisań w zależności od rozmiaru danych

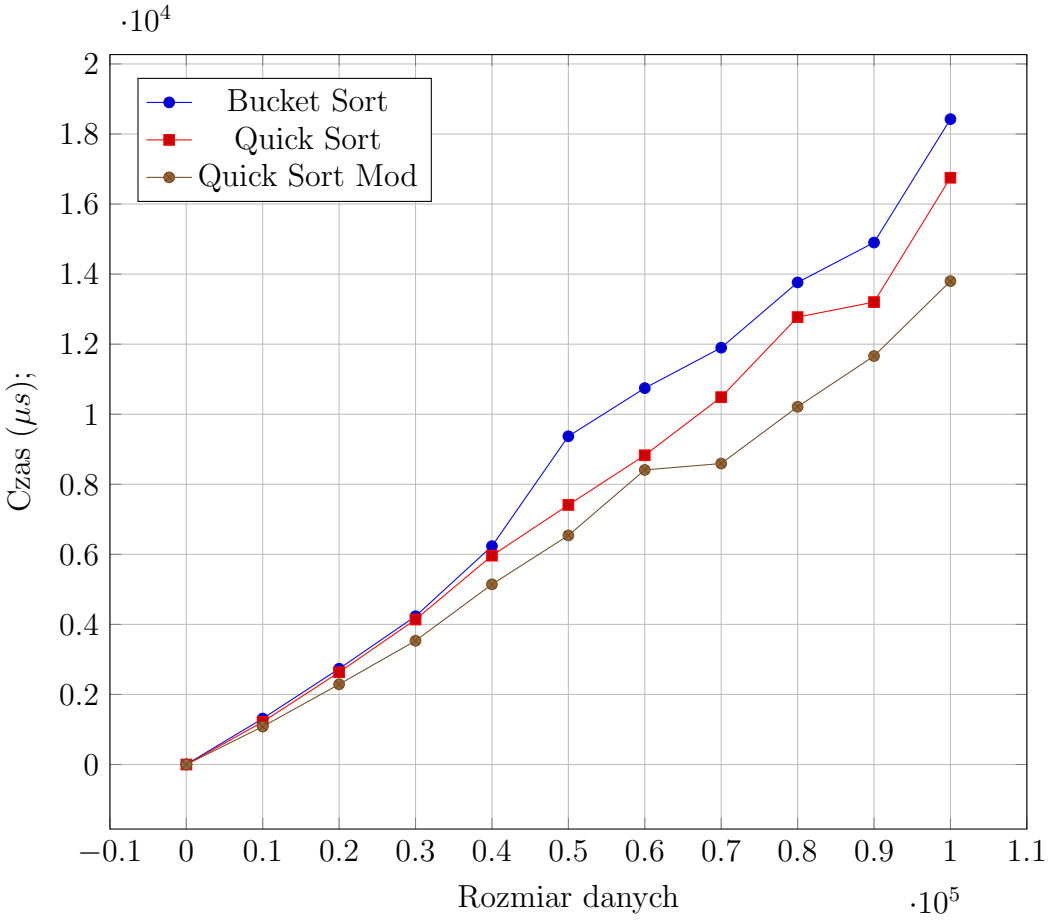

Figure 3: Czas wykonania w zależności od rozmiaru danych

3.3 Tabela Wyników Radix Sort

Rozmiar danych	Podstawa	Algorytm	Porównania	Przypisania	Czas(s)
10	10	Radix Sort	0	147	$2\mu s$
10	10	Radix Sort Neg	0	160	$8\mu s$
10	100	Radix Sort	0	359	$1\mu s$
10	100	Radix Sort Neg	0	100	$4\mu s$
1000	10	Radix Sort	0	10057	$83\mu s$
1000	10	Radix Sort Neg	0	16000	$111\mu s$
1000	100	Radix Sort	0	6309	$49\mu s$
1000	100	Radix Sort Neg	0	10000	$67\mu s$
10000	10	Radix Sort	0	100057	$858\mu s$
10000	10	Radix Sort Neg	0	160000	$922\mu s$
10000	100	Radix Sort	0	60309	$496\mu s$
10000	100	Radix Sort Neg	0	100000	$733\mu s$
100000	10	Radix Sort	0	1000056	$8517\mu s$
100000	10	Radix Sort Neg	0	1600000	$9156\mu s$
100000	100	Radix Sort	0	600308	$4917\mu s$
100000	100	Radix Sort Neg	0	1000000	$5701\mu s$

Table 2: Porównanie liczby operacji dla różnych rozmiarów danych i baz

3.4 Wykresy Wyników Radix Sort

Poniżej przedstawiono wykres przedstawiajacy jak wyglada czas działania Radix Sort i jego modyfikacji dla tabeli rozmiaru 10000 i różnych baz.

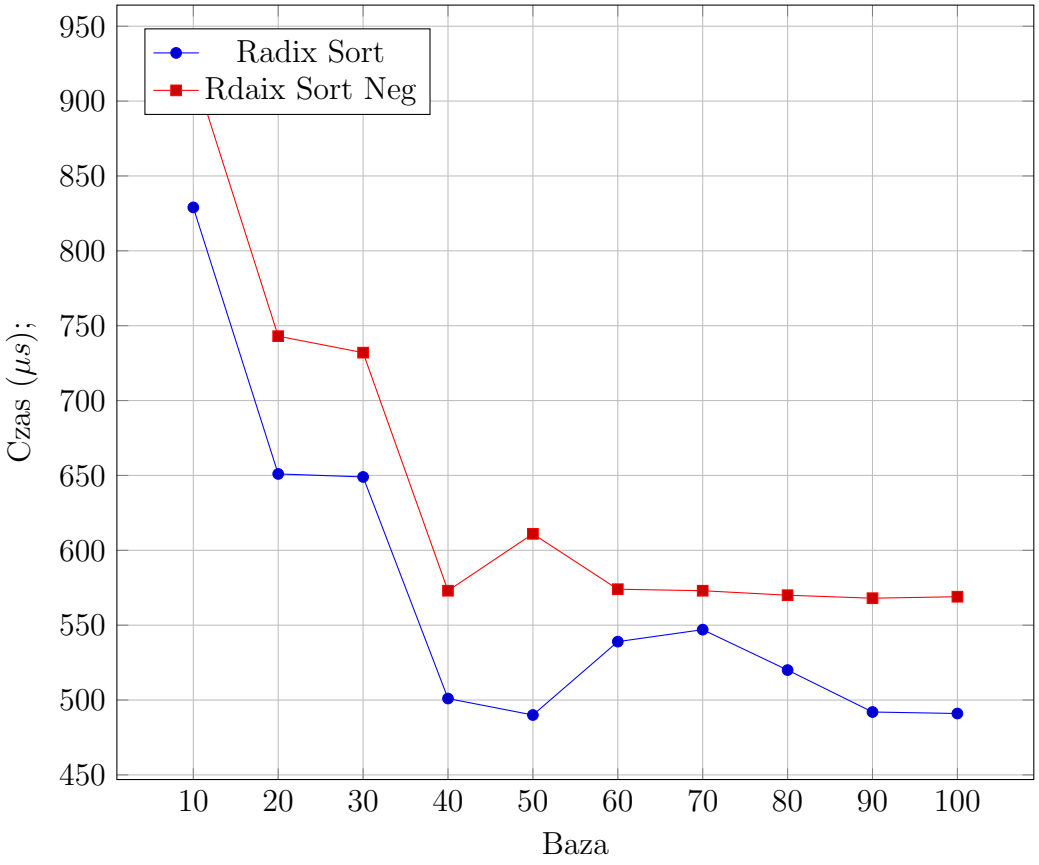

Figure 4: Czas wykonania w zależności od bazy

4 Wnioski

Przeprowadzone badania pozwalaja dostrzec, jak różne algorytmy sortowania zachowuja sie w zależności od danych i parametrów. W przypadku algorytmu Radix Sort, zmiana bazy (liczby elementów, na które dzielimy dane w każdym przebiegu) ma istotny wpływ na liczbe wymaganych iteracji oraz operacji. Wraz ze wzrostem bazy liczba iteracji maleje, ponieważ wiecej bitów liczby jest przetwarzanych jednocześnie. Jednak zwiekszenie bazy zwieksza złożoność operacji na pojedynczym przebiegu, co może wpłynać na ogólna wydajność.

Porównanie Quick Sort z Bucket Sort wykazało, że Quick Sort jest zazwyczaj szybszy w praktyce, szczególnie dla wiekszych zbiorów danych, dzieki swojej efektywności w wykorzystaniu pamieci podrecznej i dobrze zbalansowanym podziałom. Mimo to, Quick Sort wykonuje znacznie wiecej operacji w porównaniu z Bucket Sort, co może czynić ten drugi bardziej efektywnym w specyficznych przypadkach, zwłaszcza przy równomiernie rozłożonych danych.

Podsumowujac, wybór odpowiedniego algorytmu zależy od charakterystyki danych wejściowych oraz wymagań dotyczacych wydajności i złożoności obliczeniowej.