# Analiza porównawcza algorytmów: Radix, Quick oraz Bucket Sort

## 1 Wprowadzenie

W niniejszym sprawozdaniu przedstawiono implementację i analizę wydajności algorytmów sortowania:

1. `RADIX_SORT` (sortowanie pozycyjne, baza $d = 10$ oraz $d = 1024$),
2. `QUICK_SORT` (klasyczne sortowanie szybkie),
3. `QUICK_SORT_MOD` (modyfikacja z podziałem na 3 części / dual pivot),
4. `BUCKET_SORT_MOD` (sortowanie kubełkowe dla dowolnych danych).

Dla tablic o różnych rozmiarach zmierzono czas działania algorytmów. Dodatkowo zbadano wpływ wyboru podstawy $d$ na wydajność algorytmu Radix Sort.

## 2 Metodyka pomiarów

Pomiary wykonano w języku C++ (standard C++23) z użyciem `std::chrono::high_resolution_clock` i zapisem wyników do plików CSV. Metodyka zapewniała sprawiedliwe porównanie:

**Rozmiary i powtórzenia.** Dla każdego $n \in \{100, 200, 500, \ldots, 100000\}$ wykonano 10 niezależnych prób. W każdej próbie generowano nowy zestaw danych losowych, a następnie każdy algorytm otrzymywał **kopię** tego samego zestawu danych. Czas w wynikach jest średnią arytmetyczną z 10 prób.

**Losowanie danych.** Elementy tablicy losowano z użyciem `std::mt19937` (Mersenne Twister) z rozkładem jednorodnym w przedziale $[-10^6, 10^6]$.

**Pomiar czasu.** Czas mierzono w mikrosekundach ($\mu s$). Dla algorytmu `BUCKET_SORT` uwzględniono czas konwersji danych do typu `float`.

## 3 Najciekawsze fragmenty kodu z wyjaśnieniami

### RADIX_SORT (z obsługą liczb ujemnych)

```
1     static void counting_sort(vector<int>&A, int ile_narny, int pozycja,bool znak)
2     {
3       int n = A.size();
4       vector<int> C (ile_narny,0);
5       vector<int> B (n);
6
7       //zliczanie wystapien
8       for (int x:A)
9       C[(abs(x)/(int)pow(ile_narny,pozycja))%ile_narny]++;
10
11      for (int j = 1; j <= ile_narny-1; ++j)
12      C[j] += C[j - 1]; //tablica ostatnich pojawien
13
```

```
14
15        for (int i = n - 1; i >= 0; --i) {
16          int wartosc_elementu = A[i];
17          int cyfra=(abs(wartosc_elementu)/(int)pow(ile_narny,pozycja))%ile_narny;
18          // C[wartosc_elementu] zawiera pozycje ostatniego wystapienia tej wartosci
19          int ostateczny_indeks = C[cyfra] - 1;
20
21          B[ostateczny_indeks] = wartosc_elementu;
22
23          C[cyfra]--;
24        }
25        A=B;
26      }
27
28      void radix(vector<int> &A, int ilosc_cyfr, int ile_narny) //w finalnej wersji
            tylko modyfikacja do testow
29      {
30        int n=A.size();
31        vector<int> A_positive;
32        vector<int> A_negative;
33
34        for (int elem:A)
35        {
36          if (elem>=0) A_positive.push_back(elem);
37          else A_negative.push_back(elem);
38        }
39
40
41        for (int i=0; i<ilosc_cyfr;i++)
42        {
43          if (!A_negative.empty()) counting_sort(A_negative, ile_narny, i, 0);
44          if (!A_positive.empty()) counting_sort(A_positive, ile_narny, i, 1);
45        }
46
47
48        int neg_size=A_negative.size();
49        int pos_size = A_positive.size();
50        for (int i=0; i<neg_size/2; ++i)
51        {
52          int buffer=A_negative[i];
53          A_negative[i]=A_negative[neg_size-i-1];
54          A_negative[neg_size-i-1]=buffer;
55        }
56
57        //scalenie dodatnich i ujemnych do A
58        for(int i = 0; i < neg_size; i++) A[i] = A_negative[i];
59        for(int i = 0; i < pos_size; i++) A[neg_size + i] = A_positive[i];
60      }
```

**Zasada działania i elastyczna baza.** Algorytm wykorzystuje stabilne sortowanie przez zliczanie (`counting_sort`) jako procedurę pomocniczą, wywoływaną dla każdej pozycji cyfry. Kluczową cechą implementacji jest parametryzacja podstawy systemu liczbowego (`ile_narny`). Pozwala to na sortowanie nie tylko w systemie dziesiętnym, ale również np. w systemie o podstawie 256 (co odpowiada sortowaniu bajtowemu), znacząco wpływając na liczbę iteracji pętli głównej.

**Obsługa liczb ujemnych.** Standardowy Radix Sort nie obsługuje liczb ujemnych. W zaprezentowanym rozwiązaniu zastosowano podział tablicy na dwie części: `A_positive` i `A_negative`. Obie części są sortowane niezależnie na podstawie wartości bezwzględnych (`abs(x)`). Ponieważ dla liczb ujemnych większa wartość bezwzględna

oznacza mniejszą liczbę rzeczywistą, posortowana tablica liczb ujemnych jest w odwrotnej kolejności. Dlatego po zakończeniu sortowania wektor `A_negative` jest odwracany, a następnie scalany z wektorem liczb dodatnich.

## QUICK_SORT (Klasyczny)

```
    static int split(vector<int>& A, int start, int end)
    {
      int pivot=A[end];
      int i=start-1;
      for (int j=start; j<=end-1; j++)
      {
        if(A[j]<pivot)
        {
          i++;
          swap(A,i,j);
        }
      }
      swap(A,i+1,end);
      return i+1;
    }
    void quicksort(vector<int>& A, int start, int end)
    {
      if (start<end)
      {
        int new_pivot=split(A, start, end);
        quicksort(A, start, new_pivot - 1);
        quicksort(A, new_pivot + 1, end);
      }
    }
```

**Mechanika działania.** Standardowa implementacja wykorzystuje schemat partycjonowania Lomuto. Jako element odniesienia (pivot) wybierany jest ostatni element zakresu (`A[end]`). Indeks `i` śledzi granicę elementów mniejszych od pivota. Pętla przechodzi przez tablicę, zamieniając elementy mniejsze od pivota na początek zakresu. Na końcu pivot jest wstawiany na swoje docelowe miejsce (`i+1`), dzieląc tablicę na dwie części, dla których rekurencyjnie wywoływana jest funkcja `quicksort`.

## QUICK_SORT_MOD (Dual Pivot)

```
    static pair<int,int> split_into_3(vector<int>& A, int start, int end)
    {
      // ewentualna zamiana poczatkowa
      if(A[start]>A[end])
      swap(A,start,end);

      int l_pivot=A[start];
      int r_pivot=A[end];
      int R=end;
      int L=start;
      int k=start+1;


      while (k<R)
      {
        if (A[k]<l_pivot)
        {
          L++;
          swap(A,L,k);
          k++;
```

```
        }
        else if (A[k]>r_pivot)
        {
          R--;
          swap(A,R,k);
        }
        else k++;
      }
      swap(A,start,L);
      swap(A,end,R);
      return {L,R};
    }
    void mod_quicksort(vector<int>& A, int start, int end)
    {
      if(start<end)
      {
        pair<int,int> new_pivots=split_into_3(A, start, end);
        mod_quicksort(A,start,new_pivots.first-1);
        mod_quicksort(A,new_pivots.first+1, new_pivots.second-1);
        mod_quicksort(A,new_pivots.second+1,end);
      }
    }
```

**Opis modyfikacji (Dual Pivot).** Zastosowano wariant algorytmu QuickSort z dwoma pivotami, co pozwala podzielić tablicę na trzy części w jednym przebiegu partycjonowania. 1. Wybierane są dwa pivoty: lewy (`l_pivot` z `A[start]`) i prawy (`r_pivot` z `A[end]`), przy zapewnieniu warunku $l\_pivot \leq r\_pivot$. 2. Funkcja `split_into_3` używa trzech indeksów $(L, R, k)$ do klasyfikacji elementów:

- Elementy mniejsze od `l_pivot` trafiają na lewą stronę (zwiększanie $L$).
- Elementy większe od `r_pivot` trafiają na prawą stronę (zmniejszanie $R$).
- Elementy pomiędzy pivotami pozostają w środku.

3. Funkcja zwraca parę indeksów nowych pozycji pivotów, a rekurencja wywoływana jest trzykrotnie dla każdego z powstałych podzbiorów. Zwiększa to efektywność dla dużych zbiorów danych poprzez zmniejszenie głębokości drzewa rekurencji.

## BUCKET_SORT (Klasyczny i struktury pomocnicze)

```
    struct Node {
      float val;
      Node* next;

      Node(float v) : val(v), next(nullptr) {}
    };

    struct LinkedList {
      Node* head;

      LinkedList() : head(nullptr) {}

      void sortedInsert(float newVal)
      {
        Node* newNode = new Node(newVal);

        if (head == nullptr || head->val >= newVal) {
          newNode->next = head;
          head = newNode;
```

```
        }
        else {
          Node* current = head;
          while (current->next != nullptr && current->next->val < newVal) {
            current = current->next;
          }
          newNode->next = current->next;
          current->next = newNode;
        }
      }

      ~LinkedList() {
        Node* current = head;
        while (current != nullptr) {
          Node* next = current->next;
          delete current;
          current = next;
        }//czyszczenie pamieci
      }
    };

    void bucketsort(vector<float>& A) {
      if (A.empty()) return;
      int n = A.size();

      vector<LinkedList> buckets(n);

      for (float val : A) {
        int bucketIndex = static_cast<int>(std::ceil(static_cast<float>(n) * val)) -
            1;

        if (bucketIndex < 0) bucketIndex = 0;
        if (bucketIndex >= n) bucketIndex = n - 1;

        buckets[bucketIndex].sortedInsert(val);
      }

      int index = 0;
      for (int i = 0; i < n; i++) {
        Node* current = buckets[i].head;

        while (current != nullptr) {
          A[index++] = current->val;

          current = current->next;
        }
      }
    }
```

**Struktury danych i mechanika.** Algorytm opiera się na tablicy list jednokierunkowych (`buckets`). Zaimplementowano własną strukturę listy (`LinkedList`) z węzłami (`Node`). Kluczowym elementem jest metoda `sortedInsert`, która realizuje sortowanie przez wstawianie (Insertion Sort) bezpośrednio w momencie dodawania elementu do kubełka. Dzięki temu każdy kubełek jest zawsze posortowany. Wersja klasyczna zakłada, że dane wejściowe znajdują się w przedziale $(0, 1]$. Indeks kubełka wyznaczany jest wzorem $j = \lceil n \cdot A[i] \rceil - 1$. Po dystrybucji danych następuje ich konkatenacja z powrotem do tablicy głównej.

## BUCKET_SORT_MOD (Dowolne dane)

```
1     void mod_bucketsort(std::vector<float>& A) {
2       if (A.empty()) return;
3       int n = A.size();
4
5       // zakres z uzyciem biblioteki algorithm
6       auto minMax = std::minmax_element(A.begin(), A.end());
7       float minVal = *minMax.first;
8       float maxVal = *minMax.second;
9
10      if (minVal == maxVal) return;
11
12      double range = maxVal - minVal;
13      std::vector<LinkedList> buckets(n);
14
15      for (float val : A) {
16
17        double normalized_val = (val - minVal) / range;
18
19        int bucketIndex = static_cast<int>(std::floor(static_cast<double>(n) *
              normalized_val));
20
21        if (bucketIndex >= n)
22        bucketIndex = n - 1;
23
24        buckets[bucketIndex].sortedInsert(val);
25      }
26      int index = 0;
27      for (int i = 0; i < n; i++) {
28        Node* current = buckets[i].head;
29        while (current != nullptr) {
30          A[index++] = current->val;
31          current = current->next;
32        }
33      }
34    }
```

**Opis modyfikacji.** Standardowy Bucket Sort nie obsługuje liczb ujemnych ani wartości spoza zakresu jednostkowego. Zmodyfikowana funkcja `mod_bucketsort` rozwiązuje ten problem poprzez normalizację danych: 1. Znajdowane są wartości minimalna ($min$) i maksymalna ($max$) w tablicy. 2. Każdy element jest skalowany do przedziału $[0, 1)$ za pomocą wzoru:

$$normalized\_val = \frac{val - min}{max - min}$$

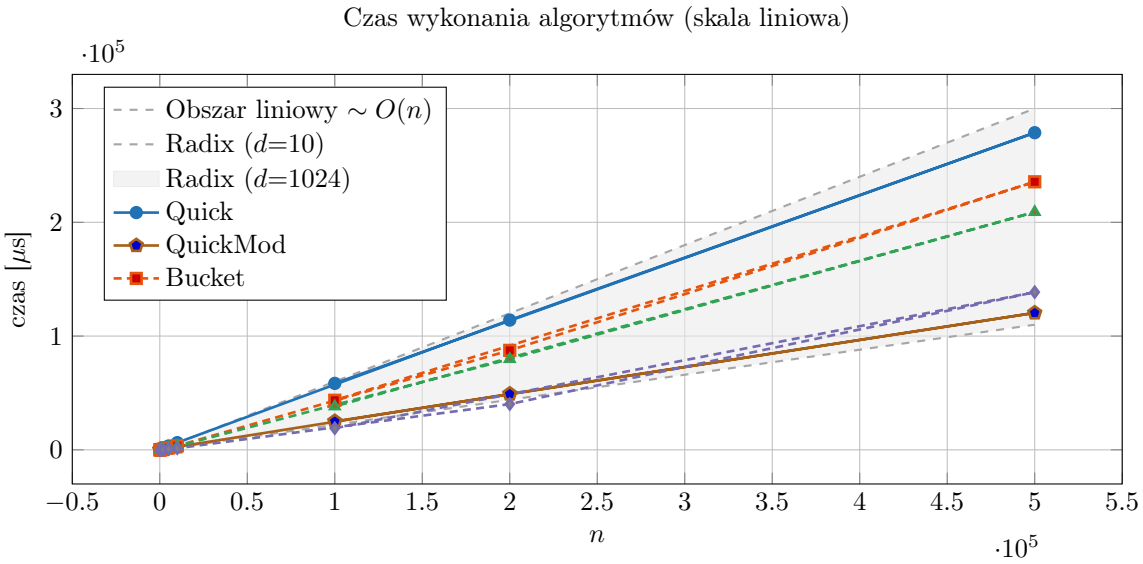
3. Indeks kubełka obliczany jest na podstawie znormalizowanej wartości: $\lfloor n \cdot normalized\_val \rfloor$. Dzięki temu algorytm działa poprawnie dla dowolnego zakresu liczb rzeczywistych, zachowując liniową złożoność przy założeniu jednostajnego rozkładu danych.

# 4 Wyniki pomiarów: Porównanie Algorytmów

## 4.1 Tabela wyników (czas w $\mu s$)

| $n$ | Radix ($d$=10) | Radix ($d$=1024) | Quick | QuickMod | Bucket |
|---|---|---|---|---|---|
| 100 | 93 | 196 | 21 | 20 | 18 |
| 200 | 136 | 169 | 38 | 34 | 23 |
| 500 | 406 | 320 | 141 | 135 | 89 |
| 1000 | 725 | 439 | 287 | 259 | 157 |
| 2000 | 1257 | 696 | 600 | 534 | 267 |
| 5000 | 3392 | 1511 | 1675 | 1638 | 715 |
| 10000 | 6326 | 2681 | 3542 | 3243 | 1415 |
| 200000 | 114065 | 49119 | 87437 | 79775 | 40212 |
| 500000 | 278760 | 120395 | 235594 | 209018 | 138590 |
| 100000 | 58409 | 25018 | 43584 | 38173 | 18988 |

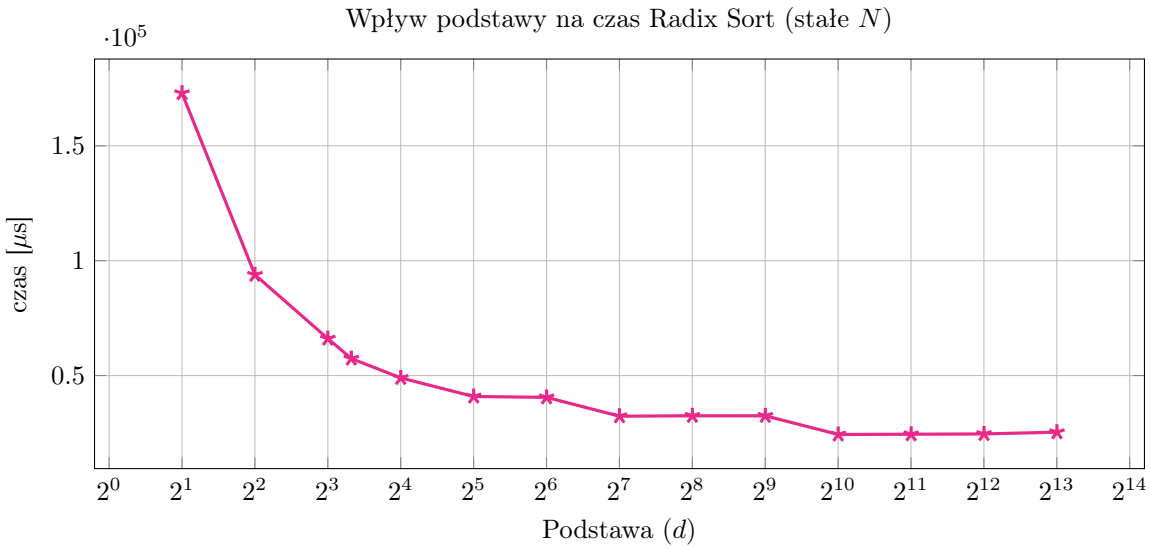
## 4.2 Wykresy wydajności




Rysunek 1: Zależność czasu wykonywania od rozmiaru danych $n$. Szary obszar wskazuje teoretyczny trend liniowy.

# 5 Analiza Radix Sort: Wpływ podstawy $d$

W tej części zbadano, jak wybór podstawy wpływa na szybkość sortowania dla ustalonego, dużego $N$.




Rysunek 2: Czas działania Radix Sort w zależności od podstawy systemu liczbowego.

# 6 Wnioski

- **Radix Sort:** Wariant z podstawą $d = 1024$ jest znacznie szybszy od standardowego $d = 10$ dla dużych $N$, co może wynikać z mniejszej liczby przebiegów pętli (mniejsze $k$) przy efektywnych operacjach bitowych. Wykres wpływu podstawy pokazuje, że optimum wydajności osiągane jest dla $d \in [2^8, 2^{10}]$.
- **Quick Sort vs Mod:** Zaimplementowana modyfikacja z dwoma pivotami (`QuickMod`) osiąga konsekwentnie lepsze czasy niż wersja klasyczna. Podział tablicy na trzy części w jednym przebiegu partycjonowania skutecznie zmniejsza głębokość rekurencji, co przekłada się na widoczny zysk wydajnościowy rosnący wraz z $N$.
- **Bucket Sort:** Algorytm ten, obok zoptymalizowanego Radix Sorta, okazał się najwydajniejszy. Wykres liniowy potwierdza jego złożoność $O(n)$ dla danych o rozkładzie jednostajnym.
- **Porównanie ogólne:** Wyniki potwierdzają przewagę algorytmów o złożoności liniowej ($O(n)$) nad logarytmiczno-liniowymi ($O(n \log n)$) dla dużych zbiorów danych. Jednakże przykład Radix Sorta ($d = 10$) pokazuje, że sama klasa złożoności to nie wszystko — wysokie stałe ukryte w notacji $O$ (np. wolne operacje modulo i duża liczba przebiegów) mogą sprawić, że algorytm liniowy będzie wolniejszy od Quick Sorta. Dopiero odpowiednia optymalizacja (Radix $d = 1024$, Bucket) pozwala w pełni wykorzystać potencjał złożoności liniowej.