Algorytmy i Struktury Danych
Sprawozdanie 2

Maja Kołakowska

287360

Spis treści

1 Wstęp

W tej pracy zajmiemy się analizą i porównaniem wybranych algorytmów sortowania. Rozpatrzone zostaną: Quick Sort, Bucket Sort, Radix Sort, Dla każdego z wymienionych algorytmów zostaną zaimplementowane ich zmodyfikowane wersje:

- Quick Sort, który dzieli tablice przy pomocy dwóch elementów
- Bucekt Sort, działający na dowolnych danych wejściowych
- Radix Sort, który sortuje również liczby ujemne.

2 Opis metody badawczej

Celem pracy jest zbadanie efektywności algorytmu Radix Sort oraz porównanie efektywności Quick Sorta i Bucket Sorta. Testujemy następujące zmienne: czas, liczbę przypisań oraz liczbę porównań.

Wykorzystujemy podstawową wersję Radix Sorta. Wykonujemy po 5 testów dla różnych podstaw i rozmiarów tablic, a następnie obliczamy średnie wartości naszych wskaźników. Badane podstawy to: 2, 4, 8, 10 i 16, natomiast badane rozmiary tablic to: 1000, 5000, 10000, 25000, 50000, 75000 i 100000.

Przy analizie porównawczej wykorzystujemy obie wersje Quick Sorta oraz zmodyfikowaną wersję Bucket Sorta. Dla każdego z algorytmów wykonujemy 100 testów dla wyżej wymienionych rozmiarów tablic, po czym obliczamy średnie wartości naszych wskaźników.

3 Quick Sort

3.1 Standardowy Quick Sort

Algorytm Quick Sort, to algorytm sortujący opierający się na metodzie "dziel i zwyciężaj". Działa poprzez wybór elementu zwanego pivotem i podzieleniu tablicy na elementy mniejsze oraz elementy większe od pivota. Algorytm następnie rekurencyjnie sortuje obie części sortuje się, aż do uzyskania tablic zawierających jeden element. Jego złożoność to $O(n \log n)$.

```
1     int Partition(int A[], int poczatek, int koniec) {
2     int x = A[koniec];
3     int i = poczatek - 1;
4
5     for (int j = poczatek; j <= koniec - 1; j++) {
```

```
        comparisons++;
        if (A[j] <= x) {
            i++;
            swap(A[i], A[j]);
            assignments+=3;
        }
    }
    swap(A[i + 1], A[koniec]);
    assignments+=3;

    return i + 1;
}

void quickSort(int A[], int p, int k) {
    if (p < k) {
        int s = Partition(A, p, k);
        quickSort(A, p, s-1);
        quickSort(A, s + 1, k);
    }
}
```

3.2 Dual Pivot Quick Sort

Dual Pivot Quick Sort to odmiana klasycznego Quick Sorta, która używa dwa pivoty zamiast jednego. Algorytm dzieli tablicę na trzy części: elementy mniejsze od pierwszego pivota, elementy mieszczące się między pivotami oraz elementy większe od drugiego pivota. Każdą z tych części algorytm sortuje rekurencyjnie. Jego złożoność to również $O(n \log n)$.

```
    void dualPivotPartition(int A[], int left, int right, int &i, int &j) {
    comparisons++;
    if (A[left] > A[right]) {
        swap(A[left], A[right]);
        assignments+=3;
    }

    int pivot1 = A[left];
    int pivot2 = A[right];
    assignments+=2;

    i = left + 1;
    int k = left + 1;
    j = right - 1;
```

```
15
16      while (k <= j) {
17          comparisons++;
18          if (A[k] < pivot1) {
19              swap(A[k], A[i]);
20              assignments+=3;
21              i++;
22              k++;
23          } else {
24              if (A[k] > pivot2) {
25                  while (k < j) {
26                      comparisons++;
27                      if (A[j] > pivot2) {
28                          j--;
29                      } else {
30                          break;
31                      }
32                  }
33                  swap(A[k], A[j]);
34                  assignments+=3;
35                  j--;
36
37                  comparisons++;
38                  if (A[k] < pivot1) {
39                      swap(A[k], A[i]);
40                      assignments+=3;
41                      i++;
42
43                  }
44              }
45                  k++;
46              }
47          }
48
49          i--;
50          j++;
51
52          swap(A[left], A[i]);
53      assignments+=3;
54          swap(A[right], A[j]);
55      assignments+=3;
56      }
57
```

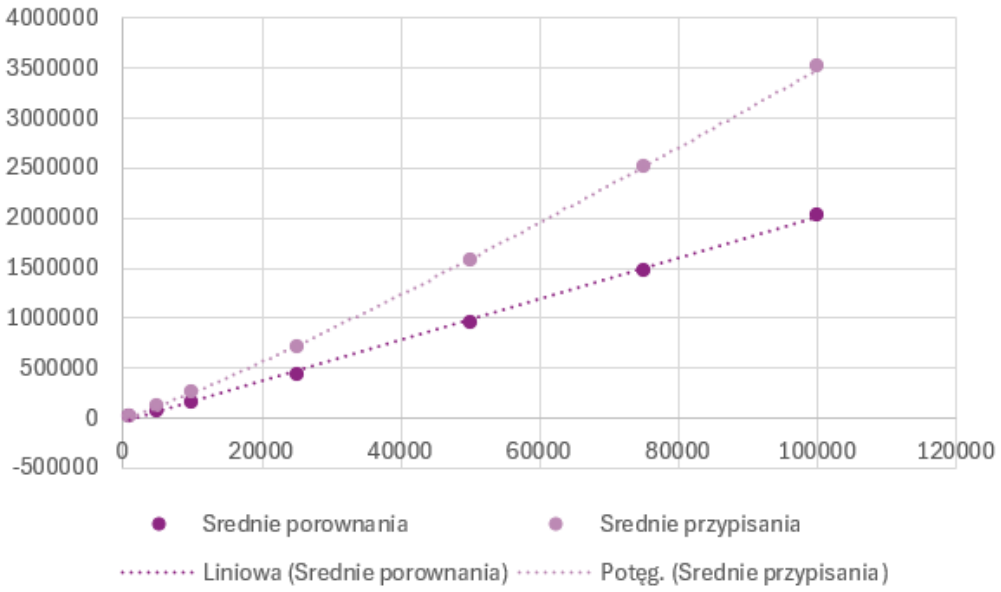
```
58
59 void dualPivotQuickSort(int A[], int left, int right) {
60     comparisons++;
61     if (left < right) {
62         int i, j;
63         dualPivotPartition(A, left, right, i, j);
64
65         dualPivotQuickSort(A, left, i - 1);
66         dualPivotQuickSort(A, i + 1, j - 1);
67         dualPivotQuickSort(A, j + 1, right);
68     }
69 }
```

3.3 Pomiary Standardowej Wersji

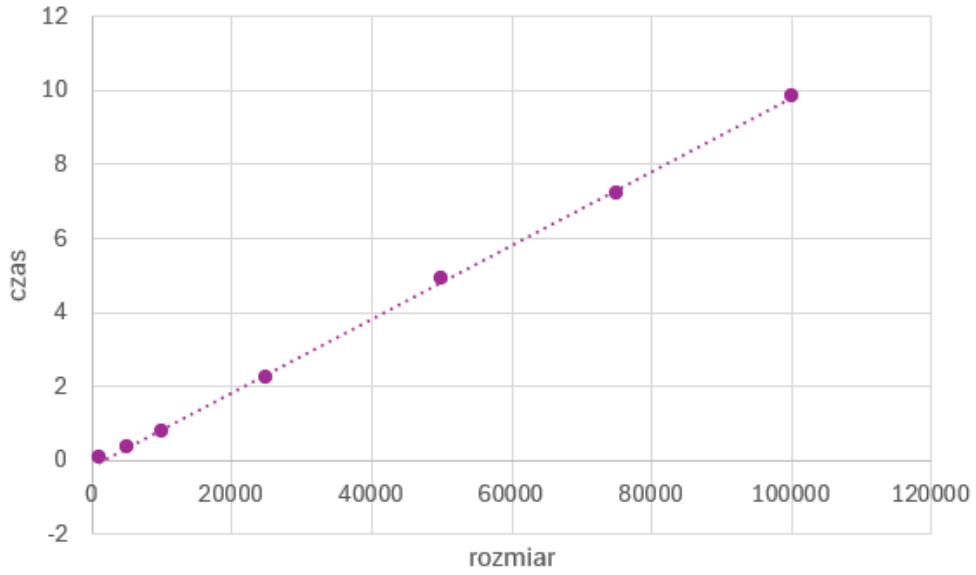
Dla każdego z siedmiu podanych uprzednio rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, porównań i przypisań. Wyniki tych pomiarów prezentuje Tabela 1.

Rozmiar tablicy	**Porównania**	**Przypisania**	**Czas (ms)**
1000	10850	18814	0.07
5000	70875	119714	0.38
10000	154460	256428	0.78
25000	436920	719927	2.24
50000	948473	1582107	4.93
75000	1483426	2511401	7.22
100000	2037230	3512333	9.85

Tabela 1: Wyniki testów algorytmu Quick Sort



Rysunek 1: Porównania i Przypisania Quick Sort



Rysunek 2: Czas Quick Sort

Funkcję czasu aproksymowano funkcją liniową, ponieważ jest ona zbliżona do funkcji liniowo-logarytmicznej dla małych rozmiarów danych. Wyniki w

tabeli są zbliżone do funkcji liniowej, dlatego możemy uznać, że dla niewielkich danych złożoność jest lepsza, niż zakładaliśmy.

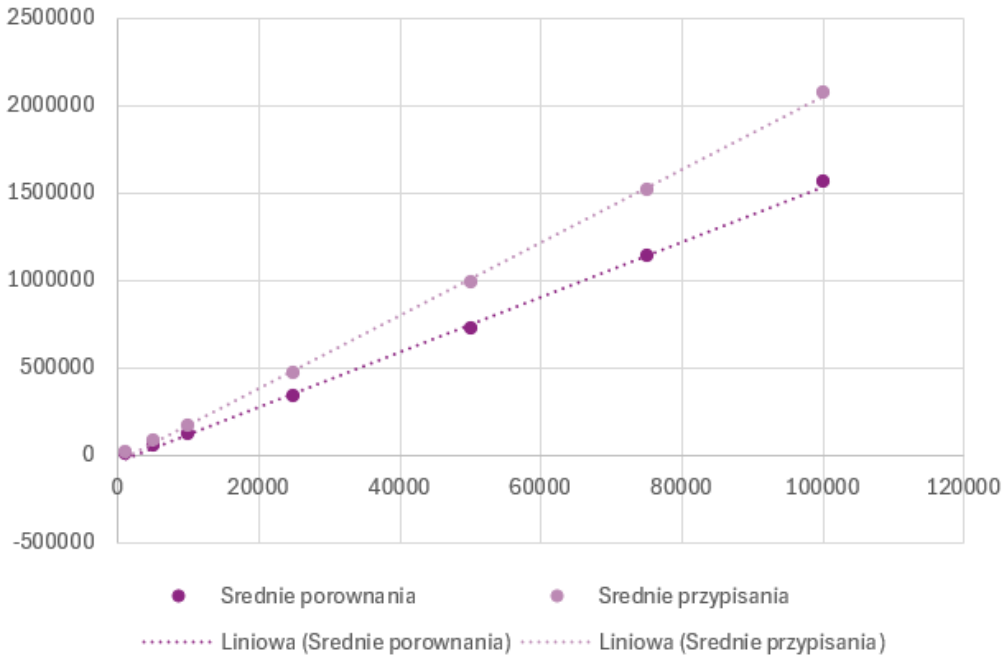
Algorytm wykonuje zdecydowanie więcej przypisań niż porównań. Przypisania przypominają funkcję potęgową, natomiast porównania - liniową. Przypisania rosną w ten sposób, ze względu na rekurencję.

3.4 Pomiary Zmodyfikowanej Wersji

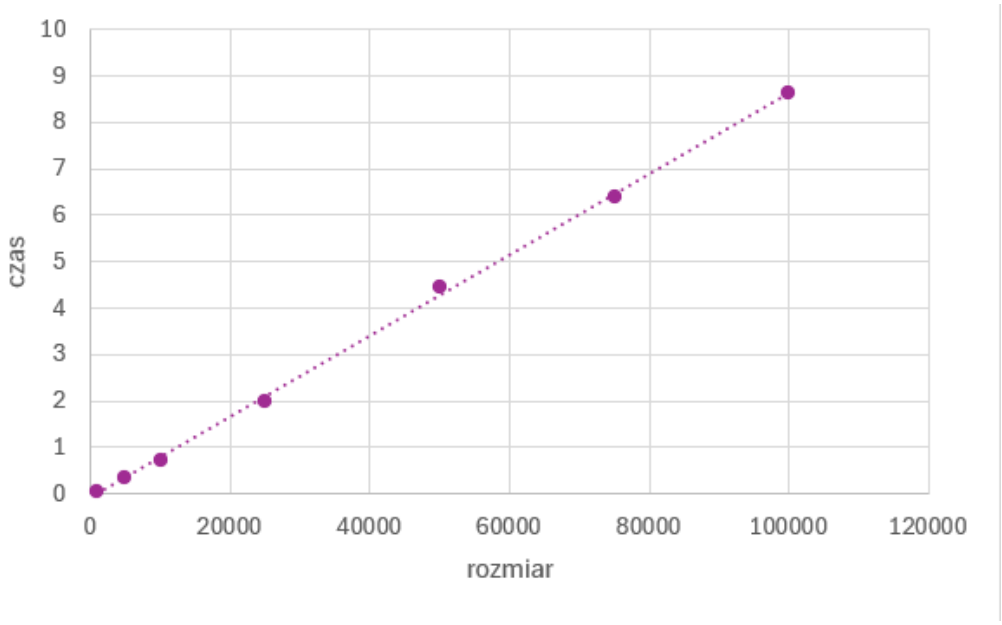
Dla każdego z siedmiu podanych uprzednio rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, porównań i przypisań. Wyniki tych pomiarów prezentuje Tabela 2.

Rozmiar tablicy	**Porównania**	**Przypisania**	**Czas (ms)**
1000	9068	13360	0,06
5000	56566	80172	0,35
10000	122900	172547	0,73
25000	339054	471846	1,99
50000	725693	988949	4,45
75000	1138581	1519117	6,39
100000	1564638	2077849	8,62

Tabela 2: Wyniki testów algorytmu Dual Pivot Quick Sort



Rysunek 3: Porównania i Przypisania Dual Pivot Quick Sort



Rysunek 4: czas Dual Pivot Quick Sort

Funkcję czasu aproksymowano funkcją liniową, ponieważ jest ona zbliżona

do funkcji liniowo-logarytmicznej dla małych rozmiarów danych. Wyniki w tabeli są zbliżone do funkcji liniowej, dlatego możemy uznać, że dla niewielkich danych złożoność jest lepsza, niż zakładaliśmy.

Algorytm wykonuje więcej przypisań niż porównań jednak różnica nie jest aż tak widoczna jak w przypadku standardowej implementacj. Obie funkcje zostały aproksymowane funkcją liniową, dla przypisań, zmiana ta wynika z zmniejszonej ilości rekurencji.

4 Bucket Sort

4.1 Implementacja Listy i Insertion Sort

Insertion Sort na liście jednokierunkowej działa według tej samej idei co wersja tablicowa – elementy są kolejno wstawiane w odpowiednie miejsce w już posortowanej części zbioru. Różnica polega jednak na sposobie reorganizacji danych: zamiast zamieniać wartości w tablicy, algorytm operuje na wskaźnikach, wpinając aktualny węzeł w odpowiednie miejsce listy.

```
1  struct Node {
2      double data;
3      Node* next;
4
5      Node(double val) : data(val), next(nullptr) {}
6  };
7
8  class LinkedList {
9  private:
10     Node* head;
11
12 public:
13     LinkedList() : head(nullptr) {}
14
15     ~LinkedList() {
16         Node* current = head;
17         while (current) {
18             Node* next = current->next;
19             delete current;
20             current = next;
21         }
22     }
23
24     void insert(double val) {
```

```
25          Node* newNode = new Node(val);
26          newNode->next = head;
27          head = newNode;
28      }
29
30      void insertionSort() {
31          if (!head || !head->next) return;
32
33          Node* sorted = nullptr;
34          Node* current = head;
35
36          while (current) {
37              Node* next = current->next;
38
39              if (!sorted || sorted->data >= current->data) {
40                  current->next = sorted;
41                  sorted = current;
42              } else {
43                  Node* temp = sorted;
44                  while (temp->next && temp->next->data < current->data) {
45                      temp = temp->next;
46                  }
47                  current->next = temp->next;
48                  temp->next = current;
49              }
50
51              current = next;
52          }
53
54          head = sorted;
55      }
56  };
```

4.2 Standardowy Bucket Sort

Bucket sort to algorytm sortowania oparty na rozdzielaniu danych na kilka „kubełków", z których każdy przechowuje wartości z określonego zakresu, standardowo zakresy są z odcinka [0,1). Po umieszczeniu elementów w odpowiednich kubełkach każdy z nich jest sortowany osobno za pomocą insertion sorta, a następnie kubełki są łączone w jedną posortowaną listę. Złożoność programu zależy od algorytmu wybranego do porządkowania zawartości kubełków. W przypadku insertion sorta jest to $O(n^2)$. W przypadku

idealnie rozłożonych danych złożoność to $O(n)$.

```
void bucketSort(double A[], int n) {
    LinkedList* B = new LinkedList[n];

    for (int i = 0; i < n; i++) {
        int bucket_index = (n * A[i]) - 1;
        bucket_index = max(0, min(n - 1, bucket_index));
        B[bucket_index].insert(A[i]);
    }

    for (int j = 0; j < n; j++) {
        if (!B[j].empty()) {
            B[j].insertionSort();
        }
    }

    int index = 0;
    for (int j = 0; j < n; j++) {
        B[j].copyToArray(A, index);
    }

    delete[] B;
}
```

4.3 Zmodyfikowany Bucket Sort

Zmodyfikowana wersja Bucket Sort, w odróznieniu do standardowej, porządkuje liczby na dowolnym zakresie, nie tylko z przedzialu [0,1). Realizuje to poprzez wyznaczenie minimum i maksimum, a następnie przeskalowanie wartości tak, aby poprawnie przydzielić je do kubełków. Złożonośc algorytmu jest podobna do standardowej wersji; $O(n^2)$, lub w przypadku idealnie rozłożonych danych $O(n)$.

```
void bucketSort(int A[], int n, int& comparisons, int& assignments) {
    if (n <= 0) return;

    int min_val = A[0];
    int max_val = A[0];
    for (int i = 1; i < n; i++) {
        comparisons += 2;
        if (A[i] < min_val) min_val = A[i];
        if (A[i] > max_val) max_val = A[i];
    }
```

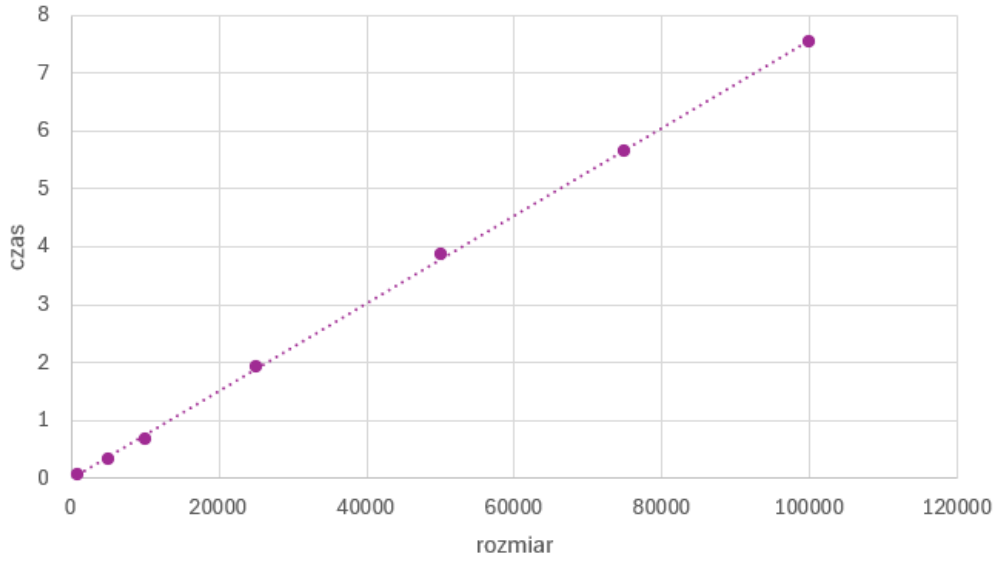
```
11
12      LinkedList* B = new LinkedList[n];
13
14      for (int i = 0; i < n; i++) {
15          int bucket_index = (int)((long long)(A[i] - min_val) * n / (max_val - min_val +
16          B[bucket_index].insert(A[i]);
17          assignments += 2;
18      }
19
20      if (min_val != max_val) {
21          for (int j = 0; j < n; j++) {
22              if (!B[j].empty()) {
23                  B[j].insertionSort(comparisons, assignments);
24              }
25          }
26      }
27
28      int index = 0;
29      for (int j = 0; j < n; j++) {
30          B[j].copyToArray(A, index);
31      }
32
33      delete[] B;
34  }
```

4.4 Pomiary Zmodyfikowanej Wersji

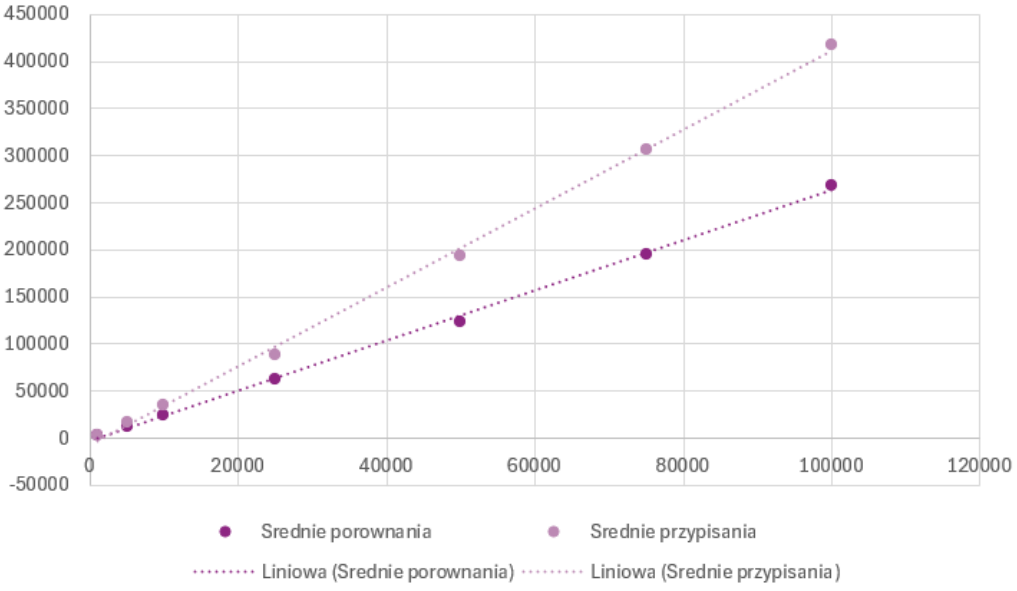
Dla każdego z siedmiu podanych uprzednio rozmiarów tablic wykonano serię stu sortowań z pomiarem czasu, porównań i przypisań. Wyniki tych pomiarów prezentuje Tabela 3.

Rozmiar tablicy	**Porównania**	**Przypisania**	**Czas (ms)**
1000	2461	3526	0,06
5000	12346	17667	0,33
10000	24973	35651	0,66
25000	62041	89121	1,93
50000	124349	193026	3,86
75000	195548	306634	5,66
100000	268782	417031	7,55

Tabela 3: Wyniki testów algorytmu Bucket Sort



Rysunek 5: czas Bucket Sort



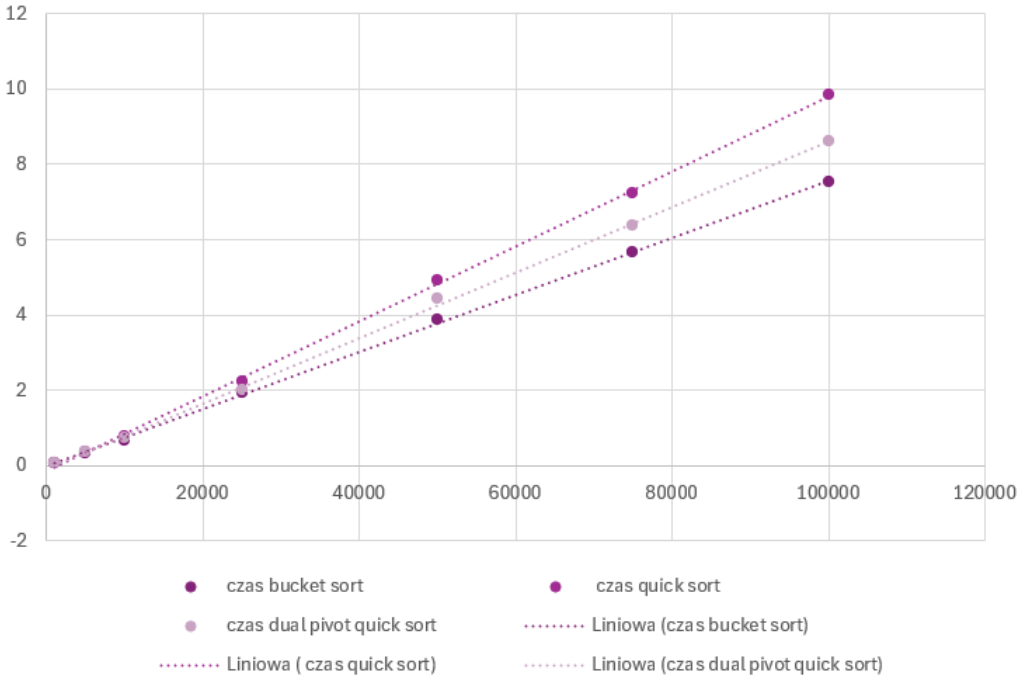
Rysunek 6: Przypisania i porównania Bucket Sort

Funkcję czasu aproksymowano funkcją liniową, ponieważ jest ona zbliżona do funkcji liniowo-logarytmicznej dla małych rozmiarów danyc. Wyniki w tabeli są zbliżone do funkcji liniowej, dlatego możemy uznać, że dla niewiel-

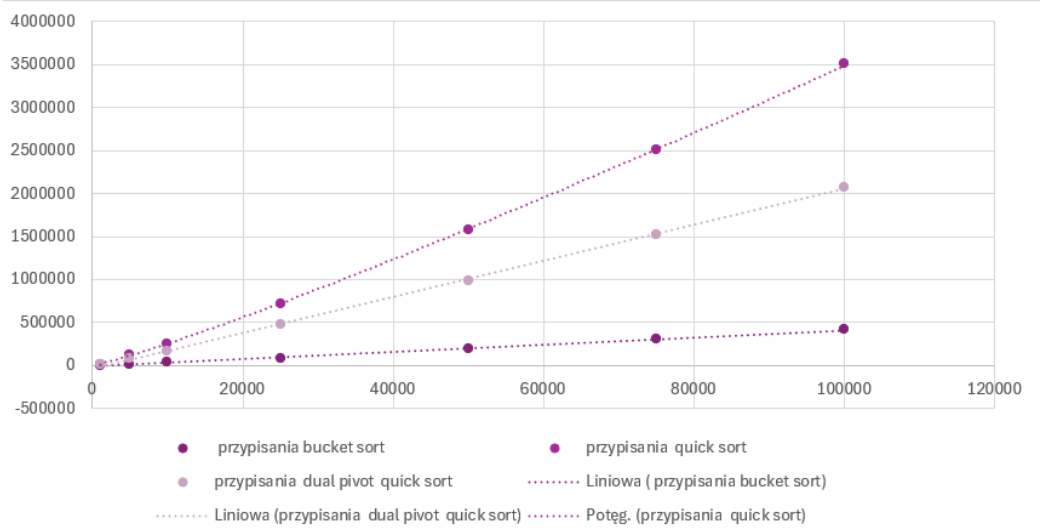
kich danych złożoność jest lepsza, niż zakładaliśmy. Może to też świadczyć o dobrze rozłożonych danych na kubełki.

Algorytm wykonuje więcej przypisań niż porównań jednak różnica ta nie jest ogromna. Obie funkcje zostały aproksymowane funkcjami liniowymi. Wynika to z podziału listy na mniejsze podlisty, co upraszca sortowanie.

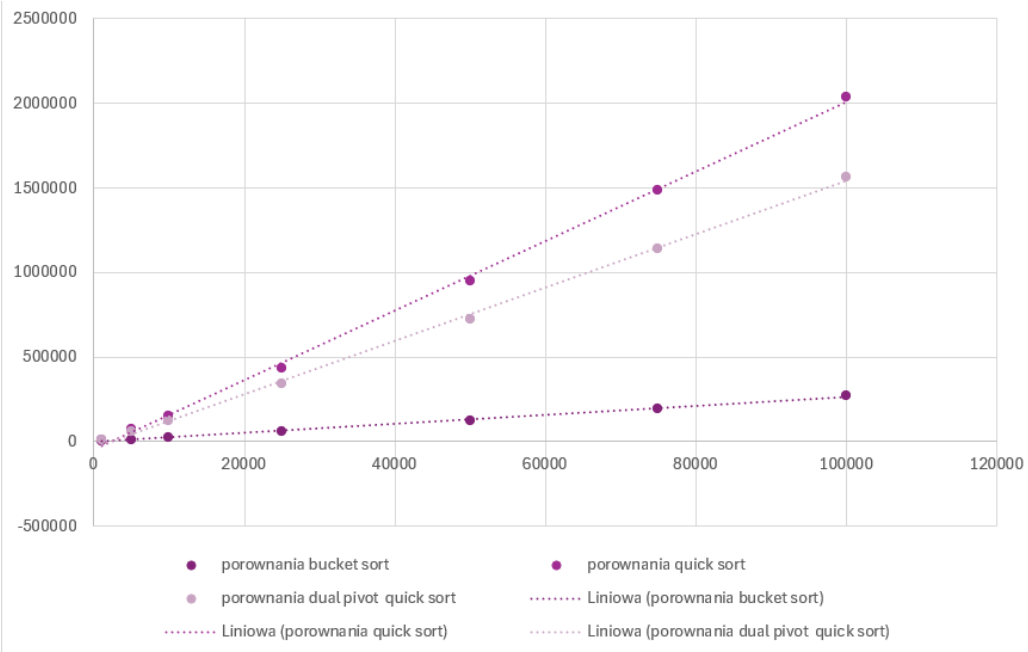
5 Porównanie Quick Sort i Bucket Sort



Rysunek 7: Porównanie czasu



Rysunek 8: Porównanie przypisań



Rysunek 9: Porównanie porównań

Na Rysunkach 7, 8, 9 zostały przedstawione zbiorcze wykresy naszych algorytmów. Wynika z nich, że najefektywniejszym algorytmem jest Bucket Sort. Nieco gorszy Dual Pivot Quick Sort, a najgorzej wypada klasyczny Quick Sort. Różńice między dwoma Quick Sortami wynikają z lepszego podziału

w przypadku zmodyfikowanej wersji. Bucket Sort wypada lepiej, ze względu na podział danych do sortowania, ponieważ zna zakres danych i dzieli je równomiernie. Jednak musimy zauważyć, że operujemy na małym zakresie danych, może mieć to znaczenie, ponieważ mimo optymalnego podziału na kubełki Bucket Sort jest oparty na Insertion Sort, który zaczyna działać bardzo wolno dla dużych wartości. Powtarzając eksperymet warto sprawdzić tą tezę.

6 Radix Sort

6.1 Standardowy Radix Sort

Radix sort to algorytm sortowania pozycyjnego, który porządkuje liczby na podstawie ich poszczególnych cyfr, zaczynając od najmniej znaczącej pozycji. W każdej iteracji przegląda zbiór i grupuje elementy w kolejności rosnącej względem aktualnie analizowanej cyfry, aż do momentu, gdy zostaną uwzględnione wszystkie cyfry najbardziej znaczące w największej wartości. Jego złożoność czasowa wynosi $O(d(n+k))$, gdzie k to liczba różnych cyfr, a d liczba cyfr w kluczach.

```
1   void counting_sort_by_digit(int A[], int n, int d, int divisor) {
2   int* B = new int[n];
3   int* C = new int[d];
4
5   for (int j = 0; j < d; j++) {
6       C[j] = 0;
7   }
8
9   for (int i = 0; i < n; i++) {
10      int digit = get_digit(A[i], divisor, d);
11      C[digit]++;
12  }
13
14  for (int j = 1; j < d; j++) {
15      C[j] += C[j - 1];
16  }
17
18  for (int i = n - 1; i >= 0; i--) {
19      int digit = get_digit(A[i], divisor, d);
20      B[C[digit] - 1] = A[i];
21      C[digit]--;
22  }
```

```
23
24      for (int i = 0; i < n; i++) {
25          assignments++;
26          A[i] = B[i];
27      }
28
29      delete[] B;
30      delete[] C;
31  }
32
33  void radix_sort(int A[], int n, int d, int k) {
34      for (int i = 0; i < k; i++) {
35          int divisor = 1;
36          for (int j = 0; j < i; j++) {
37              divisor *= d;
38          }
39          counting_sort_by_digit(A, n, d, divisor);
40      }
41  }
```

6.2 Zmodyfikowany Radix Sort

Zmodyfikowana wersja Radix Sort działa na tej samej zasadzie. W odróżnieniu od klasycznej wersji, algorytm ten obsługuje również liczby ujemne, przesuwając cały zbiór o wartość minimalną tak, aby wszystkie wartości stały się nieujemne, a po sortowaniu cofając przesunięcie.

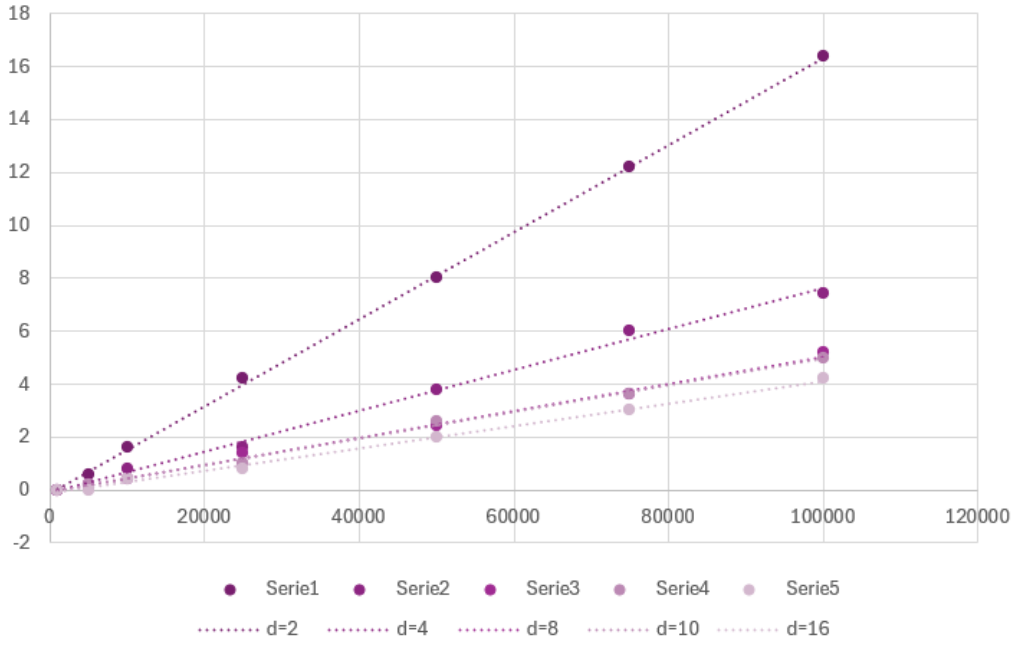
```
1       int shift = 0;
2       if (min_val < 0) {
3           shift = -min_val;
4           for (int i = 0; i < n; i++) {
5               A[i] += shift;
6           }
7       }
8
9       for (int i = 0; i < k; i++) {
10          int divisor = 1;
11          for (int j = 0; j < i; j++) {
12              divisor *= d;
13          }
14          counting_sort_by_digit(A, n, d, divisor);
15      }
16
```

```
17      if (shift > 0) {
18          for (int i = 0; i < n; i++) {
19              A[i] -= shift;
20          }
21      }
22  }
```

6.3 Pomiary Standardowej Wersji

d \ Rozmiar	1000	5000	10000	25000	50000	75000	100000
2	0,0	0,6	1,6	4,2	8,0	12,2	16,4
4	0,0	0,2	0,8	1,6	3,8	6,0	7,4
8	0,0	0,2	0,4	1,4	2,4	3,6	5,2
10	0,0	0,2	0,4	1,0	2,6	3,6	5,0
16	0,0	0,0	0,4	0,8	2,0	3,0	4,2

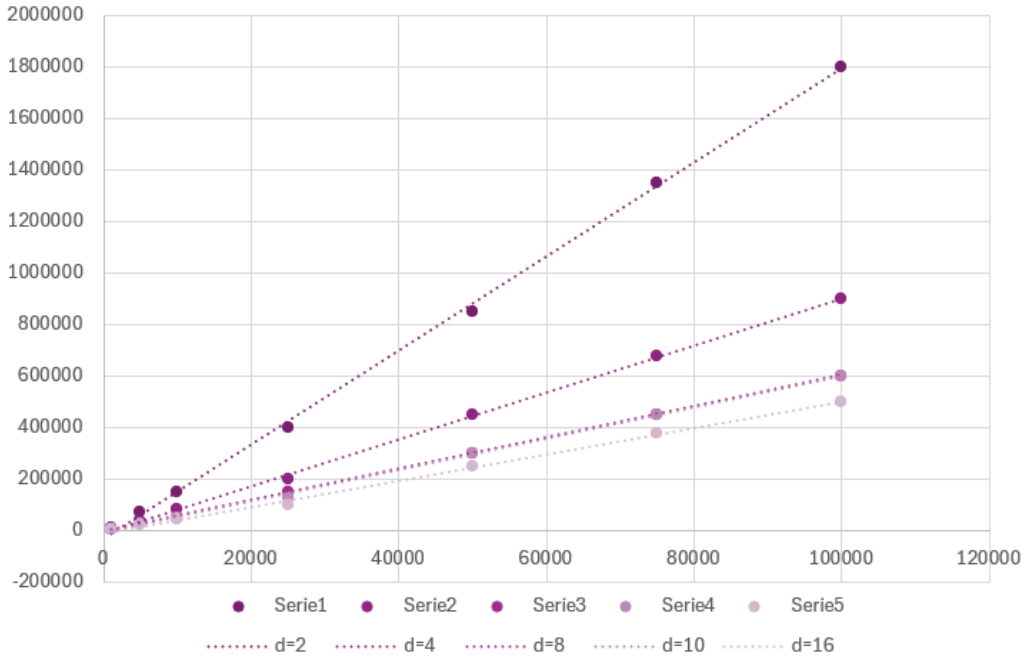
Tabela 4: Czas dla różnych rozmiarów tablic i podstaw d



Rysunek 10: czas Radix Sort

d \ Rozmiar	1000	5000	10000	25000	50000	75000	100000
2	11000	70000	150000	400000	850000	1350000	1800000
4	6000	35000	80000	200000	450000	675000	900000
8	4000	25000	50000	150000	300000	450000	600000
10	4000	25000	50000	125000	300000	450000	600000
16	3000	20000	40000	100000	250000	375000	500000

Tabela 5: Przypisania dla różnych rozmiarów tablic i podstaw d



Rysunek 11: przypisania Radix Sort

Funkcje czasu, dla różnych podstaw d, aproksymowano funkcjami liniowymi. Funkcje różnią się współczynikami, co jest zgodnez teoretycznym założeniem, Widzimy, że im mniejsza podstawa, tym szybciej rośnie funkcja. Wynika to z faktu, że dla mniejszych podtsaw d, algorytm musi wykonać więcej iteracji sortowania po cyfrach, co zwiększa całkowity czas wykonania.

Podobnie funkcje przypisań, dla różńych podstaw d, sproksymowano funkcjami liniowymi, które również różnią się wspólczynnikami. Zochowana jest również ta sama zależność, im mniejsza podstawa, tym szybciej rośnie funkcja, która również wynika z większej ilości iteracji sortowania po cyfrach. Pomijamy liczenie porównań, ponieważ algorytm nie porównuje elementów między sobą, a jedynie przetwarza ich cyfry w kolejnych iteracjach.