Sprawozdanie - Lista 2

Jakub Zdancewicz

Spis treści

1 Wstęp

W poprzednim sprawozdaniu omówiliśmy trzy kluczowe algorytmy sortowania oparte na porównaniach. W tym sprawozdaniu skoncentrujemy się na kolejnym, prawdopodobnie najpopularniejszym w tej kategorii – *Quick Sort* – oraz przedstawimy jego modyfikację, która polega na dzieleniu tablicy z wykorzystaniem dwóch pivotów.

Dla algorytmów sortowania opartych na porównaniach dolne ograniczenie złożoności czasowej wynosi $\Omega(n \log n)$. Istnieją jednak algorytmy, które przy spełnieniu odpowiednich założeń potrafią sortować dane bez porównań, osiągając złożoność lepszą niż $\mathcal{O}(n \log n)$. W tym sprawozdaniu opiszemy dwa takie algorytmy:

- *Counting Sort*
- *Radix Sort*

Dodatkowo omówimy implementację algorytmu *Bucket Sort*, który, łącząc techniki używane w sortowaniu liczb bez porównywania oraz z wykorzystaniem porównań, przy pewnych założeniach, osiąga średnią złożoność czasową rzędu $\mathcal{O}(n)$. Omówimy też implementację struktury list, wykorzystywanej do realizacji algorytmu *Bucket Sort*, oraz zaimplementujemy algorytm *Insertion Sort* działający na listach. Celem tego sprawozdania jest analiza wspomnianych algorytmów oraz porównanie ich efektywności.

2 Opis zaimplementowanych algorytmów

Każdy z algorytmów został przetestowany na 10 losowo wygenerowanych tablicach dla każdej z wybranych długości. Elementy tablic to liczby wymierne lub całkowite z zakresu $[-10^6, 10^6]$. Średnia liczba operacji przypisań i porównań dla m tablic o danej długości n jest wyliczana według wzoru:

$$L_n = \sum_{i=1}^{m} \left\lceil \frac{porownania_i}{m} \right\rceil + \left\lceil \frac{przypisania_i}{m} \right\rceil$$

gdzie:

L_n - średnia liczba operacji dla tablicy o długości n,

porownania_i - liczba porównań wykonanych dla i-tej tablicy,

przypisania_i - liczba przypisań wykonanych dla i-tej tablicy.

Analogicznie obliczamy średni czas wykonania algorytmu dla m tablic o danej długości n.

2.1 Quick Sort

Quick Sort to algorytm sortowania, którego pesymistyczna złożoność wynosi $\mathcal{O}(n^2)$. Taka złożoność występuje jednak w szczególnych, wyjątkowo złośliwych przypadkach. Popularność tego algorytmu wynika z jego średniej złożoności, która wynosi $\mathcal{O}(n \log n)$, co sprawia, że jest on niezwykle efektywny w praktyce. *Quick Sort* jest przykładem algorytmu niestabilnego. Kluczowym elementem implementacji *Quick Sort* jest funkcja **PARTITION**:

```
int partition(double A[], int p, int k)
{
  double x = A[k];
  int i = p - 1;
  for (int j = p; j < k; ++j)
  {
    if (A[j] <= x)
    {
      i += 1;
      swap(A[i], A[j]);
    }
  }
  swap(A[i + 1], A[k]);
  return i + 1;
}
```

Kod 1: Implementacja `Partition`

Zadaniem funkcji **PARTITION** jest podzielenie tablicy A na dwie części względem wybranego elementu, zwanego pivotem, poprzez odpowiednie zamiany elementów. Lewą część tablicy stanowią elementy mniejsze lub równe pivotowi, natomiast prawą część tworzą elementy większe. Po zakończeniu działania funkcji pivot zostaje umieszczony na swojej docelowej pozycji, która jest również jego docelowym miejscem w posortowanej tablicy. Funkcja zwraca indeks tego miejsca. W powyższej implementacji pivotem jest ostatni element tablicy $A[k]$.

Teraz możemy przedstawić implementację algorytmu *Quick Sort*:

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

Kod 2: Implementacja *Quick Sort*

Algorytm ten dzieli tablicę na dwie części: $A[p] \dots A[s-1] \leqslant A[s] < A[s+1] \dots A[k]$, używając funkcji **PARTITION**, gdzie element $A[s]$ jest umieszczony na swoim docelowym miejscu. Następnie algorytm wywołuje się rekurencyjnie na lewej i prawej części tablicy, aż do momentu, gdy każda część będzie zawierała tylko jeden element.

2.1.1 Modyfikacja Quick Sort

Rozważmy modyfikację algorytmu *Quick Sort* polegającą na dzieleniu tablicy przy pomocy dwóch elementów. Kluczowa różnica zachodzi w implementacji funkcji **PARTITION**:

```
indexPair partition(double A[], int p, int k)
{
  if (A[p] > A[k])
  {
    swap(A[p], A[k]);
  }
  double x = A[p];
  double y = A[k];
  int i = p;
  int j = k;
  int s = p + 1;
  while (s < j)
  {
    if (A[s] <= x)
    {
      ++i;
      swap(A[i], A[s]);
      ++s;
    }
    else if (A[s] >= y)
    {
      --j;
      swap(A[j], A[s]);
    }
    else
    {
      ++s;
    }
  }
  swap(A[i], A[p]);
  swap(A[j], A[k]);
  indexPair pair;
  pair.first = i;
  pair.second = j;
  return pair;
}
```

Kod 3: Implementacja Modyfikacji `Partition`

W tej modyfikacji funkcja **PARTITION** realizuje podział tablicy na trzy części przy pomocy dwóch pivotów $A[p]$ oraz $A[k]$, przy czym na początku wykonujemy zamianę, aby zagwarantować, że $A[p] \leqslant A[k]$. Lewą część, którą tworzymy przez przesuwanie wskaźnika i, stanowią elementy mniejsze bądź równe $A[p]$, część środkową tworzą elementy, których wartości mieszczą się między $A[p]$ a $A[k]$, natomiast prawa część, tworzona przez przesuwanie wskaźnika j, stanowią elementy większe od $A[k]$. Na końcu funkcja wstawia oba pivoty na odpowiednie miejsca, które są ich docelowymi pozycjami w posortowanej tablicy, a następnie zwraca indeksy tych miejsc używając struktury **indexPair**.

Dodajemy również dodatkowe rekurencyjne wywołanie w implementacji algorytmu *Quick Sort*:

```
void quicksort(double A[], int p, int k)
{
  if (p < k)
  {
    indexPair s = partition(A, p, k);
    quicksort(A, p, s.first - 1);
    quicksort(A, s.first + 1, s.second - 1);
    quicksort(A, s.second + 1, k);
  }
}
```

Kod 4: Implementacja Modyfikacji `Quick Sort`

Zatem tym razem dzielimy tablicę na trzy części: $A[p] \dots A[s.first - 1] \leqslant A[s.first] \leqslant A[s.first + 1] \dots A[s.second - 1] \leqslant A[s.second] \leqslant A[s.second + 1] \dots A[k]$, gdzie $A[s.first]$ i $A[s.second]$ są wstawione na odpowiednie miejsca. Następnie algorytm wywołuje się rekurencyjnie na każdej części tablicy, aż do momentu, gdy każda z nich będzie zawierała tylko jeden element.

Zaproponowane modyfikacje nie prowadzą do zmniejszenia złożoności czasowej algorytmu, jednak znacząco redukują liczbę wykonywanych operacji.

2.1.2 Wyniki dla Quick Sort

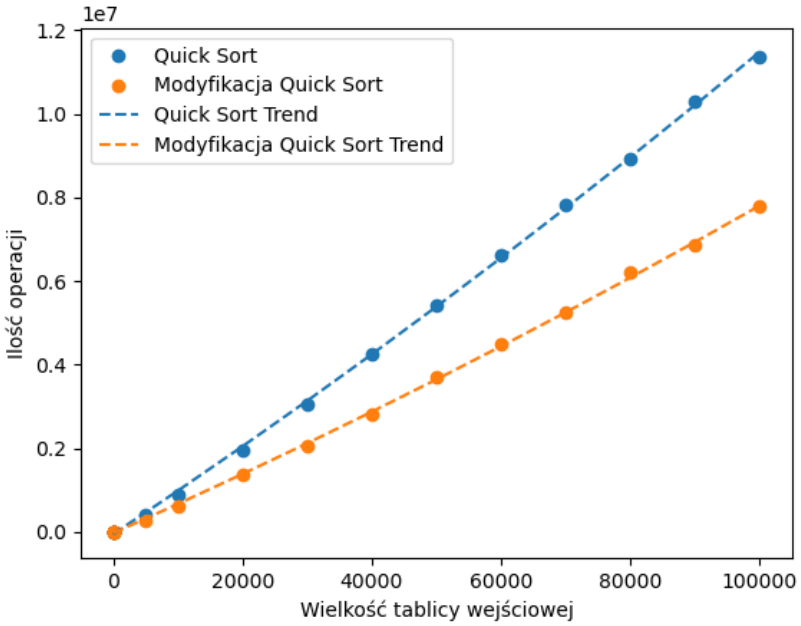
Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Quick Sort* oraz jego modyfikacji na tablicach o różnej wielkości:

Wielkość tablicy	Ilość przypisań	Ilość porównań	Łączna liczba operacji	Czas (ms)
5	31	30	61	0
10	89	81	170	0
100	2,015	1,833	3,848	0
5,005	219,072	189,569	408,641	1
10,000	475,704	412,936	888,640	2
50,005	2,925,003	2,494,433	5,419,436	10
80,000	4,796,315	4,133,818	8,930,133	16
100,000	6,101,569	5,273,746	11,375,315	20

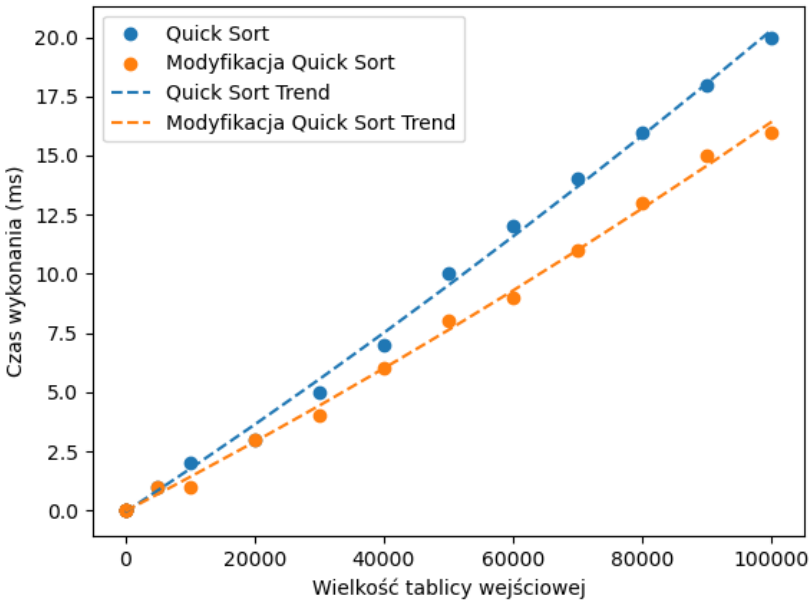
Tabela 1: Liczba operacji dla algorytmu `Quick Sort` przy różnych rozmiarach tablicy

Wielkość tablicy	Ilość przypisań	Ilość porównań	Łączna liczba operacji	Czas (ms)
5	31	25	56	0
10	88	68	156	0
100	1,597	1,269	2,866	0
5,005	155,489	123,292	278,781	1
10,000	348,442	276,157	624,599	1
50,005	2,061,746	1,637,566	3,699,312	8
80,000	3,466,685	2,740,471	6,207,156	13
100,000	4,327,709	3,456,949	7,784,658	16

Tabela 2: Liczba operacji dla modyfikacji algorytmu `Quick Sort` przy różnych rozmiarach tablicy



Wykres 1: Ilość operacji w zależności od rozmiarów tablicy wejściowej



Wykres 2: Czas wykonania algorytmu w zależności od rozmiarów tablicy wejściowej

2.2 Radix Sort

Radix Sort to algorytm sortowania, który należy do grupy algorytmów nieopierających się na porównaniach. Jego działanie polega na porządkowaniu liczb na podstawie kolejnych cyfr w danym systemie liczbowym, zaczynając od najmniej, a kończąc na najbardziej znaczącej pozycji. Z tego względu algorytm ten jest stosowany przy założeniu, że sortujemy tablice liczb naturalnych. Złożoność czasowa algorytmu wynosi $\Theta(k(n+d))$, gdzie:

- k to maksymalna liczba pozycji (cyfr) w liczbach,
- n to liczba elementów w tablicy,
- d to liczba różnych cyfr w systemie liczbowym (np. w systemie dziesiętnym $d = 10$).

W naszej implementacji *Radix Sort* korzystamy z algorytmu *Counting Sort* do sortowania elementów na podstawie każdej cyfry. Dzięki stabilności *Counting Sort*, algorytm *Radix Sort* jest również stabilny. Poniżej przedstawiamy jego implementację oraz opis:

```
void countingSort(int A[], int n, int d, int base)
{
  int C[d];
  int B[n];
  for (int j = 0; j < d; ++j)
  {
    C[j] = 0;
  }
  for (int i = 0; i < n; ++i)
  {
    ++C[(A[i] / base) % d];
  }
  for (int j = 1; j < d; ++j)
  {
    C[j] += C[j - 1];
  }
  for (int i = n - 1; i > -1; --i)
  {
    int digit = (A[i] / base) % d;
    B[C[digit] - 1] = A[i];
    --C[digit];
  }
  for (int i = 0; i < n; ++i)
  {
    A[i] = B[i];
  }
}
```

Kod 5: Implementacja `Counting Sort`

W ogólnej formie algorytm `Counting Sort` działa poprzez zliczanie wystąpień poszczególnych liczb w tablicy C, a następnie obliczenie sum prefiksowych, które reprezentują ostatni indeks, na którym może znaleźć się dana liczba. Następnie, iterując od tyłu po tablicy A, możemy stabilnie posortować elementy, przypisując je do tablicy B w odpowiednie miejsca wskazane przez tablicę C.

W powyższej implementacji algorytm został dostosowany do sortowania kolejnych cyfr w d-arnej reprezentacji liczb, dzięki zastosowaniu parametrów d oraz `base`. Dowolną liczbę x w d-arnej reprezentacji można zapisać jako sumę:

$$x = \sum_{p=0}^{k} a_p d^p,$$

gdzie $a_p \in \{0, \ldots, d-1\}$.

Wtedy poprzez dzielenie całkowite oraz operację modulo:

$$\frac{x}{d^p} \mod d,$$

możemy wyłuskać p-tą cyfrę w d-arnej reprezentacji liczby.

Teraz możemy przedstawić kod implementacji algorytmu `Radix Sort`:

```
void radixSort(int A[], int n, int d)
{
  if (n == 0)
  {
    return;
  }
  int max = maximumNumber(A, n);
  for (int base = 1; max / base > 0; base *= d)
  {
    countingSort(A, n, d, base);
  }
}
```

Kod 6: Implementacja `Radix Sort`

Algorytm ten najpierw znajduje największą liczbę max w tablicy A, a następnie iteruje przez wszystkie potęgi d, które występują w zapisie max. W każdej iteracji sortuje liczby w tablicy A według kolejnych cyfr w d-arnym zapisie liczbowym, korzystając z algorytmu `Counting Sort`.

2.2.1 Rozszerzenie algorytmu Radix Sort

Powyższy algorytm można łatwo rozszerzyć na tablice liczb całkowitych. Poniżej przedstawiamy szczegóły implementacji rozszerzonej wersji `Radix Sort`:

```
void negativeRadixSort(int A[], int n, int d)
{
  int positive[n];
  int negative[n];
  int p = 0;
  int neg = 0;
  for (int i = 0; i < n; ++i) {
    if (A[i] >= 0) {
      positive[p] = A[i];
      ++p;
    }
    else {
      negative[neg] = -A[i];
      ++neg;
    }
  }
  radixSort(positive, p, d);
  radixSort(negative, neg, d);
  int index = 0;
  for (int i = neg - 1; i > -1; --i) {
    A[index] = -negative[i];
    ++index;
  }
  for (int i = 0; i < p; ++i) {
    A[index] = positive[i];
    ++index;
  }
}
```

Kod 7: Implementacja rozszerzonej wersji `Radix Sort`

Rozszerzenie algorytmu `Radix Sort` polega na rozdzieleniu tablicy A na dwie części:

- tablicę wartości nieujemnych *positive*, zawierającą wszystkie liczby $\geqslant 0$ z tablicy A,
- tablicę modułów wartości ujemnych *negative*, zawierającą wartości $|x|$, gdzie $x < 0$.

Następnie wykonywane są następujące kroki:

1. Tablica *positive* jest sortowana przy użyciu standardowego algorytmu `Radix Sort`.
2. Tablica *negative* jest sortowana również algorytmem `Radix Sort`.
3. Tablica posortowana jest tworzona poprzez połączenie odwróconej tablicy *negative* (z powrotem jako liczby ujemne) z tablicą *positive*.

Dzięki temu podejściu algorytm może efektywnie i stabilnie sortować zarówno liczby dodatnie, jak i ujemne.

2.2.2 Wyniki dla Radix Sort

Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Radix Sort* względem różnych podstaw d:

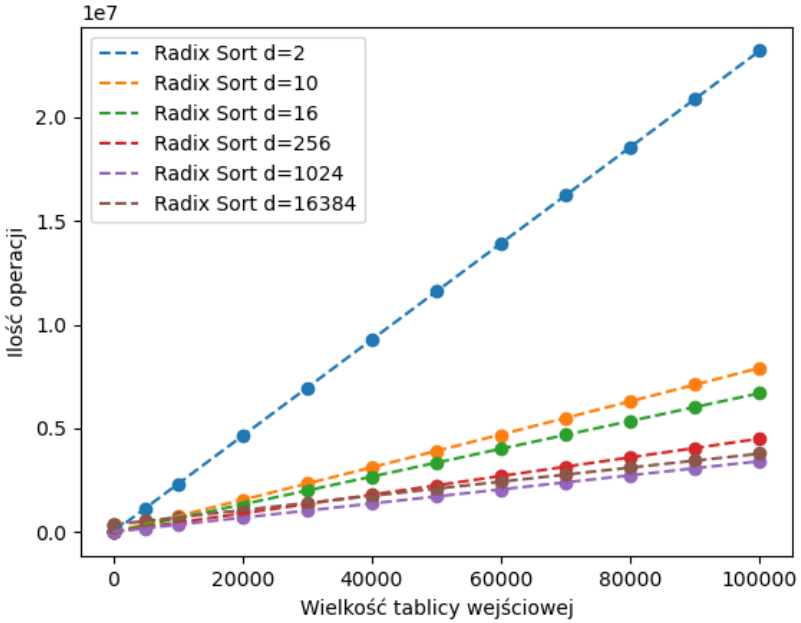
Wielkość tablicy	$d = 2$	$d = 10$	$d = 16$	$d = 256$	$d = 1024$	$d = 16384$
5	2,018	1,247	1,414	9,064	23,580	373,788
10	3,255	1,653	1,762	9,753	24,982	393,622
100	24,147	8,678	7,797	13,808	28,046	396,688
5,005	1,162,114	391,277	336,441	234,542	194,825	563,465
10,000	2,320,954	780,888	671,106	459,318	364,654	733,297
50,005	11,602,117	3,928,768	3,351,442	2,259,543	1,724,830	2,093,470
80,000	18,560,960	6,328,616	5,361,108	3,609,319	2,744,659	3,113,298
100,000	23,200,960	7,911,114	6,701,111	4,509,323	3,424,662	3,793,299

Tabela 3: Liczba operacji dla algorytmu `Radix Sort` przy podstawie d w zależności od rozmiarów tablicy wejściowej

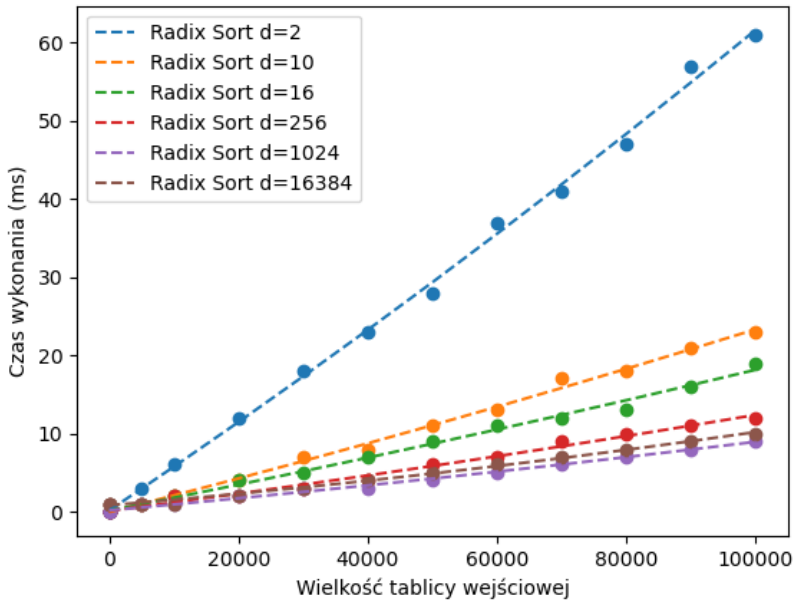
Wielkość tablicy	$d = 2$	$d = 10$	$d = 16$	$d = 256$	$d = 1024$	$d = 16384$
5	0	0	0	0	0	1
10	0	0	0	0	0	1
100	0	0	0	0	0	1
5,005	3	1	1	1	1	1
10,000	6	2	2	2	1	1
50,005	38	11	9	6	4	5
80,000	47	18	13	10	7	8
100,000	61	23	19	12	9	10

Tabela 4: Czas wykonywania(ms) dla algorytmu `Radix Sort` przy podstawie d w zależności od rozmiarów tablicy wejściowej

Z tabel wynika, że zwiększanie wartości d przynosi wymierne korzyści w przypadku dużych tablic, jednak dla małych tablic powoduje wydłużenie czasu działania algorytmu.



Wykres 3: Ilość operacji w zależności od rozmiarów tablicy wejściowej i podstawy d



Wykres 4: Czas działania algorytmu (ms) w zależności od rozmiarów tablicy wejściowej i bazy d

2.3 Bucket Sort

Bucket Sort to algorytm sortowania, który opiera się na podziale danych na mniejsze grupy, zwane *kubełkami* (ang. *buckets*), a następnie posortowaniu każdego kubełka osobno. W zależności od algorytmu użytego do sortowania zawartości kubełków, *Bucket Sort* ma pesymistyczną złożoność czasową $\mathcal{O}(n^2)$ lub $\mathcal{O}(n \log n)$. Popularność tego algorytmu wynika z faktu, że przy założeniu, iż sortujemy liczby z przedziału $(0, 1]$, rozłożone jednostajnie na tym przedziale, średnia złożoność czasowa wynosi $\Theta(n)$. Kluczową rolę w implementacji algorytmu *Bucket Sort* pełni struktura listy, której najważniejszą część przedstawiamy poniżej:

```
struct Node {
  Node *prev = nullptr;
  double key;
  Node *next = nullptr;
};
struct List {
  Node *head = nullptr;
};
```

Kod 8: Implementacja `Listy`

Struktura ta składa się z węzłów, które są połączone za pomocą wskaźników wskazujących na poprzedni oraz następny element listy. Często implementuje się razem z listą metody **INSERT**, **DELETE** oraz **SEARCH**. Pierwsza z nich jest wykorzystywana w implementacji algorytmu *Bucket Sort*, dlatego jej implementacja została przedstawiona poniżej:

```
void insert(Node *x)
{
  x->next = this->head;
  x->prev = nullptr;
  if (this->head == nullptr)
  {
    this->head = x;
  }
  else
  {
    this->head->prev = x;
    this->head = x;
  }
}
```

Kod 9: Implementacja `Insert`

Metoda ta wstawia nowy węzeł na początek listy oraz aktualizuje wskaźnik na **HEAD**. Bardzo ważnym elementem algorytmu *Bucket Sort* jest również możliwość sortowania listy. Można to przeprowadzić na dwa sposoby: sortując całą listę po wstawieniu wszystkich elementów lub sortując listę na bieżąco, podczas dodawania każdego elementu. W tej implementacji algorytmu *Bucket Sort* zastosowaliśmy algorytm *Insertion Sort*. Zdecydowaliśmy się na sortowanie na bieżąco, dlatego poniżej przedstawiamy zmieniony kod metody **INSERT**. Dla zainteresowanych, alternatywna wersja sortowania całej listy została zaimplementowana w pliku *list.cpp*.

```
void insert(Node *x)
{
  if (this->head == nullptr)
  {
    this->head = x;
    return;
  }
  Node *curr = this->head;
  if (curr->key >= x->key)
  {
    x->next = this->head;
    this->head->prev = x;
    x->prev = nullptr;
    this->head = x;
    return;
  }
  while (curr->next != nullptr && curr->next->key < x->key)
  {
    curr = curr->next;
  }
  if (curr->next == nullptr)
  {
    curr->next = x;
    x->prev = curr;
    x->next = nullptr;
  }
  else
  {
    x->next = curr->next;
    x->prev = curr;
    curr->next->prev = x;
    curr->next = x;
  }
}
```

Kod 10: Implementacja `Insert` z sortowaniem

Nowa wersja metody **INSERT** na początku oddzielnie rozpatruje przypadki, gdy lista jest pusta lub gdy nowy węzeł musi zostać wstawiony na początek listy. Jeśli te przypadki nie zachodzą, metoda szuka miejsca, w które nowy węzeł powinien zostać wstawiony. Następnie wstawia nowy węzeł w odpowiednie miejsce, w zależności od tego, czy jest to koniec, czy środek listy, wykonując odpowiednie operacje i aktualizując wskaźniki.

Teraz możemy przedstawić finalną implementację algorytmu *Bucket Sort*. W implementacji uwzględniliśmy dodatkową modyfikację, która umożliwia sortowanie na dowolnym przedziale liczbowym, a nie tylko na przedziale $(0, 1]$. Osiągnęliśmy to dzięki zastosowaniu normalizacji min-max, czyli przekształcenia danych za pomocą wzoru:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

gdzie x to wartość oryginalna, a x' to wartość znormalizowana, która mieści się w przedziale $(0, 1]$.

```
void bucketSort(double A[], int n)
{
  if (n == 0)
  {
    return;
  }
  Pair maxmin = maxminNumber(A, n);
  double min = maxmin.first;
  double max = maxmin.second;
  if (max == min)
  {
    return;
  }
  int range = max - min;
  List *B[n];
  for (int i = 0; i < n + 1; ++i)
  {
    B[i] = new List;
  }
  for (int i = 0; i < n; ++i)
  {
    Node *element = new Node;
    element->key = A[i];
    int index = int((element->key - min) / range * n);
    if (index == n)
    {
      --index;
    }
    B[index]->insert(element);
  }
  int idx = 0;
  for (int i = 0; i < n; ++i)
  {
    Node *curr = B[i]->head;
    while (curr != nullptr)
    {
      A[idx] = curr->key;
      ++idx;
      curr = curr->next;
    }
  }
}
```

Kod 11: Implementacja `Bucket Sort`

W pierwszym kroku algorytmu *Bucket Sort* znajdujemy najmniejszą i największą liczbę w tablicy A. Następnie tworzymy tablicę kubełków B o rozmiarze równym liczbie elementów w tablicy A, a każdemu kubełkowi przypisujemy listę, która będzie pełniła rolę kubełka.

Następnie, iterując po tablicy A, przypisujemy każdy element do odpowiedniego kubełka. Indeks kubełka, do którego trafi dany element, obliczany jest na podstawie znormalizowanej wartości elementu, którą uzyskujemy poprzez obliczenie różnicy między wartością elementu a minimalną wartością w tablicy, a następnie przemnożenie tej różnicy przez n (liczba elementów w tablicy). W przypadku, gdy obliczony indeks odpowiada najwyższemu możliwemu indeksowi

w tablicy kubełków, stosujemy specjalny warunek, aby uniknąć przekroczenia zakresu tablicy. Warto wspomnieć, że wstawiany element jest automatycznie sortowany dzięki zastosowaniu algorytmu wstawiania przedstawionego wyżej.

Na końcu algorytmu, iterując po wszystkich kubełkach w tablicy B, wyciągamy elementy z poszczególnych list i wstawiamy je z powrotem do tablicy A. Po tej operacji tablica A będzie posortowana.

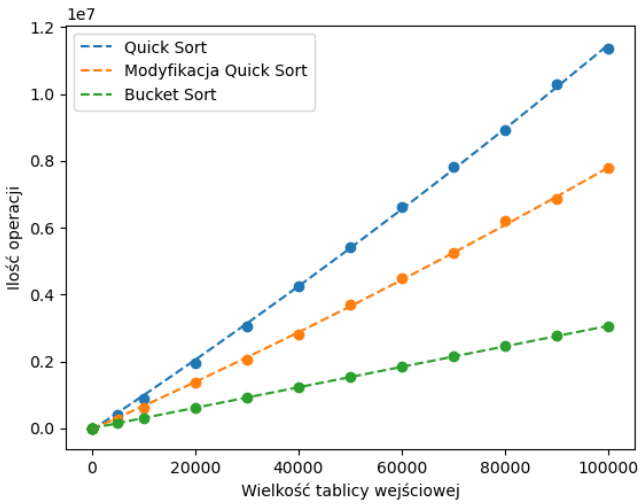
2.3.1 Wyniki dla Bucket Sort

Tabele poniżej przedstawiają wyniki eksperymentów dla algorytmu *Bucket Sort*:

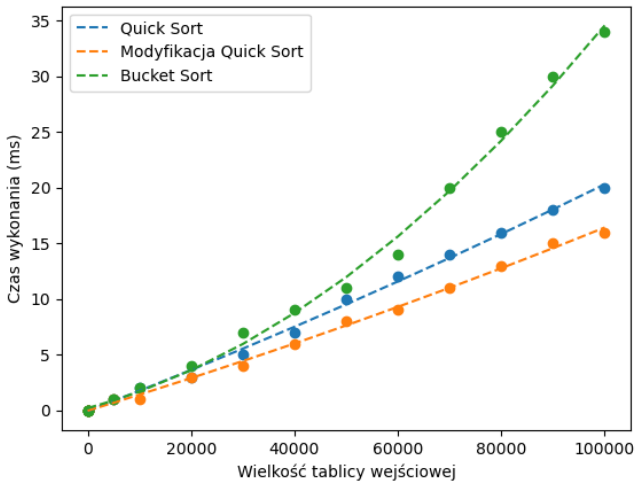
Wielkość tablicy	Ilość przypisań	Ilość porównań	Łączna liczba operacji	Czas (ms)
5	91	79	170	0
10	167	153	320	0
100	1,582	1,518	3,100	0
5,005	78,101	75,400	153,501	1
10,000	156,168	150,793	306,961	2
50,005	780,305	753,587	1,533,892	11
80,000	1,248,346	1,205,677	2,454,023	25
100,000	1,560,435	1,507,203	3,067,638	34

Tabela 5: Liczba operacji dla modyfikacji algorytmu `Bucket Sort` przy różnych rozmiarach tablicy

Poniżej przedstawiamy wykresy, na których porównujemy algorytm *Bucket Sort* z algorytmem *Quick Sort*.



Wykres 5: Ilość operacji w zależności od rozmiarów tablicy wejściowej



Wykres 6: Czas działania algorytmów w zależności od rozmiarów tablicy wejściowej

Co ciekawe, pomimo mniejszej liczby operacji wykonywanych przez algorytm `Bucket Sort`, czas jego działania jest zauważalnie dłuższy. Możliwe, że wykonywanie porównań na strukturach danych jest bardziej obciążające czasowo, a prawdziwą moc tego algorytmu dostrzegalibyśmy dopiero przy ekstremalnie dużych tablicach danych.

3 Wnioski

Z przeprowadzonej analizy wynika, że wybór algorytmu sortującego zależy głównie od charakterystyki danych wejściowych oraz ich rozmiaru. W ogólnym przypadku (bez dodatkowych założeń) algorytm `Quick Sort` jest najczęściej najlepszym wyborem, ze względu na jego elastyczność (może posortować praktycznie każdy rodzaj danych) oraz bardzo dobrą złożoność średnią ($\mathcal{O}(n \log n)$. W sytuacjach, gdy dane wejściowe są liczbami całkowitymi, algorytm `Radix Sort` wykazuje znacznie lepszą wydajność, szczególnie gdy wybór odpowiedniej podstawy d jest dopasowany do specyfiki danych. `Radix Sort` jest algorytmem liniowym w przypadku liczb całkowitych, co czyni go niezwykle wydajnym, szczególnie dla dużych zbiorów, ale jego skuteczność w dużej mierze zależy od właściwego dobrania podstawy.

Z kolei, jeśli dodatkowym założeniem jest jednostajny rozkład liczb w zadanym przedziale, bardzo dobrą efektywność wykazuje algorytm `Bucket Sort`. Jego złożoność czasowa jest zależna od liczby kubełków oraz rozkładu danych w tych kubełkach, dlatego sprawdza się szczególnie dobrze, gdy dane są równomiernie rozmieszczone. Dla danych, które są rozproszone w szerokim przedziale,

`Bucket Sort` może znacznie poprawić wydajność sortowania.

Wynika z tego, że decyzja o wyborze algorytmu sortującego powinna być oparta na analizie charakterystyki danych. W wielu przypadkach, odpowiedni dobór algorytmu pozwala na znaczną optymalizację procesu sortowania, a tym samym na zmniejszenie złożoności obliczeniowej problemu. W szczególności, dobór algorytmu dostosowanego do danych może zredukować czas wykonania operacji sortowania, co jest istotne w przypadku pracy z dużymi zbiorami danych.