Lista 2 sprawozdanie

Matylda Mordal

Zadanie 1

```
int PARTITION(int A[], int p, int r) {
    int x = A[r];
    int i = p - 1;

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

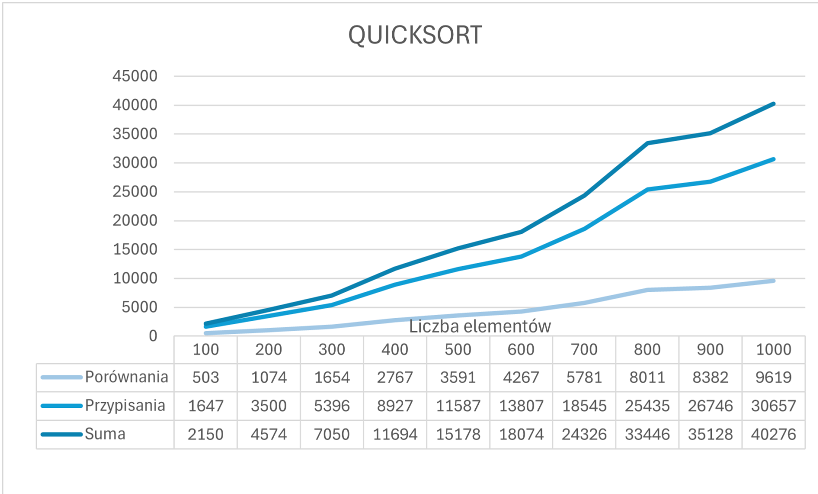
Funkcja PARTITION potrzebna jest do algorytmu QUICK_SORT, dzieli ona tablice wokół elementu zwanego pivotem.

- Ustawienie pivota: Pivotem jest ostatni element podtablicy, czyli x = A[r].
- Inicjalizacja wskaźnika i: Wskaźnik i początkowo wskazuje na "granice" elementów mniejszych lub równych pivotowi. Jest ustawiany na p - 1, czyli na pozycję przed początkiem podtablicy.
- Iteracja po elementach podtablicy: Pętla for przechodzi przez wszystkie elementy od p do r-1: Jeśli bieżący element A[j] jest mniejszy lub równy pivotowi, zwiększa się wskaźnik i i następuje zamiana elementu A[j] z A[i]. To przesuwa element A[j] do strefy "mniejszych lub równych pivotowi".
- Umieszczenie pivota na właściwej pozycji: Po zakończeniu pętli pivot (czyli A[r]) zostaje zamieniony z elementem A[i+1]. Dzięki temu pivot trafia na swoją ostateczną, posortowaną pozycję.
- Zwracanie indeksu: Funkcja zwraca indeks i + 1, czyli pozycję, na której znajduje się pivot.

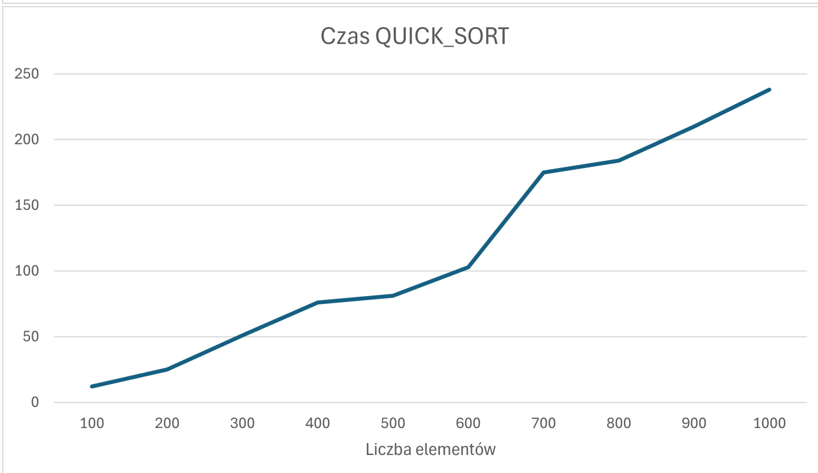
```
void QUICK_SORT(double A[], int p, int r) {
    if (p < r) {
        int q = PARTITION(A, p, r);
        QUICK_SORT(A, p, q - 1);
        QUICK_SORT(A, q + 1, r);
    }
}
```

Algorytm QUICK_SORT to jeden z najszybszych i najczęściej używanych algorytmów sortowania. Działa w oparciu o strategię dziel i zwyciężaj .

- Warunek zakończenia rekurencji: Jeśli $p >= r$ (indeks początkowy jest większy lub równy indeksowi końcowemu), oznacza to, że podtablica ma jeden element lub jest pusta. W takim przypadku jest już posortowana, i algorytm kończy działanie dla tego zakresu.
- Podział tablicy: Wywoływana jest funkcja PARTITION, któram zwraca indeks q, pod którym znajduje się pivot w swojej posortowanej pozycji.
- Rekurencja: Funkcja QUICK_SORT jest wywoływana osobno dla dwóch części: Dla lewej podtablicy: A[p..q-1]. Dla prawej podtablicy: A[q+1..r].



	100	200	300	400	500	600	700	800	900	1000
Porównania	503	1074	1654	2767	3591	4267	5781	8011	8382	9619
Przypisania	1647	3500	5396	8927	11587	13807	18545	25435	26746	30657
Suma	2150	4574	7050	11694	15178	18074	24326	33446	35128	40276



Porównania i przypisania: Liczba porównań i przypisań rośnie mniej więcej liniowo lub kwadratowo wraz ze wzrostem liczby elementów. Jest to zgodne z teoretyczną złożonością QuickSorta:

- w najlepszym przypadku $O(n \log n)$,
- w najgorszym przypadku $O(n^2)$.

Teoretyczna zgodność: Wyniki potwierdzają teoretyczne założenia QuickSorta, czyli średnią złożoność obliczeniową $O(n \log n)$.

```
void PARTITION2(double A[], int p, int r, int &x, int &y) {
    if (A[p] > A[r]) {
        std::swap(A[p], A[r]);
    }
    int pivot1 = A[p];
```

```
    int pivot2 = A[r];

    int i = p + 1;
    int a = p + 1;
    int b = r - 1;

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

    a--;
    b++;
    swap(A[p], A[a]);
    swap(A[r], A[b]);

    x = a;
    y = b;
}
```

Funkcja PARTITION2 implementuje algorytm podziału tablicy z dwoma pivotami w kontekście sortowania.

- Porównanie i zamiana pivotów: Na początku funkcja sprawdza, czy pierwszy element tablicy (A[p]) jest większy od ostatniego (A[r]). Jeśli tak, zamienia je miejscami.
- Przypisanie pivotów: pivot1 = A[p]: Mniejszy pivot. pivot2 = A[r]: Większy pivot.
- Przypisanie zmiennych: i = p + 1: Indeks aktualnego elementu. a = p + 1: Indeks granicy elementów mniejszych od pivot1. b = r - 1: Indeks granicy elementów większych od pivot2.
- Podział tablicy: Pętla przechodzi przez każdy element pomiędzy $p + 1$ a $r - 1$:
  - Jeśli $A[i] < pivot1$: Zamienia miejscami A[i] i A[a]. a jest zwiększany.
  - Jeśli $A[i] > pivot2$: Zamienia miejscami A[i] i A[b]. b jest zwiększany. i jest zmniejszany, aby ponownie sprawdzić przestawiony element.

  - W przeciwnym wypadku element pozostaje w strefie elementów pomiędzy pivotami.

  Po każdej iteracji i jest zwiększane.

- Ostateczne umiejscowienie pivotów: Zamiana mniejszego pivotu z elementem na granicy elementów mniejszych (A[a]). Zamiana większego pivotu z elementem na granicy elementów większych (A[b]). Zwracanie pozycji pivotów: x = pozycja mniejszego pivotu. y = pozycja większego pivotu.

```
void QUICK_SORT2(double A[], int p, int r) {
    if (p < r) {
        int x, y;
        PARTITION2(A, p, r, x, y);

        QUICK_SORT2(A, p, x - 1);
        QUICK_SORT2(A, x + 1, y - 1);
        QUICK_SORT2(A, y + 1, r);
    }
```

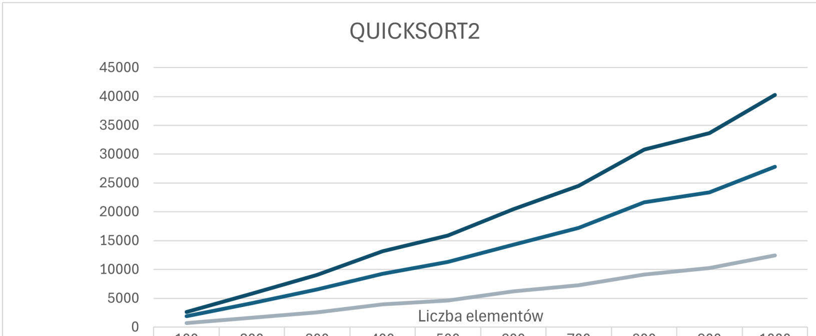
Działanie funkcji QUICK_SORT2:

- Warunek zakończenia rekurencji: Jeśli $p >= r$ (indeks początkowy jest większy lub równy indeksowi końcowemu), oznacza to, że podtablica ma jeden element lub jest pusta. W takim przypadku jest już posortowana, i algorytm kończy działanie dla tego zakresu.

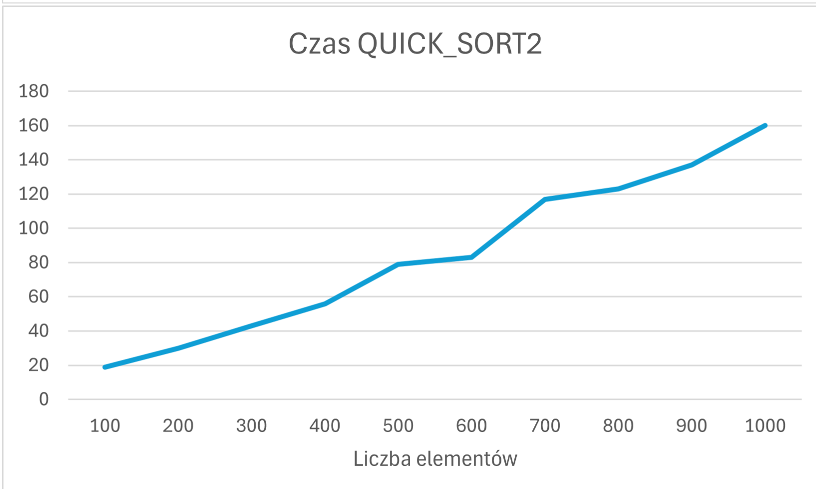
- Podział tablicy: Wywoływana jest funkcja PARTITION2, która zwraca x i y, w których znajdują się odpowiednio miejsca dla pivotów (pivot1 i pivot2).

- Rekurencja: Funkcja następnie wywołuje rekurencyjnie sortowanie dla trzech części:

  - Pierwsza część: sortuje przedział od p do $x-1$, który zawiera elementy mniejsze od pivot1.
  - Druga część: sortuje przedział od $x+1$ do $y-1$, który zawiera elementy pomiędzy pivot1 a pivot2.
  - Trzecia część: sortuje przedział od $y+1$ do r, który zawiera elementy większe od pivot2.



	100	200	300	400	500	600	700	800	900	1000
Porównania	710	1656	2577	3931	4623	6179	7289	9143	10242	12456
Przypisania	1915	4171	6503	9248	11281	14267	17183	21624	23396	27808
Suma	2625	5827	9080	13179	15904	20446	24472	30767	33638	40264



Porównania i przypisania zwiększają się niemal liniowo w miarę wzrostu liczby elementów.
Wraz ze wzrostem liczby elementów, liczba operacji (porównań i przypisań) rośnie szybciej niż liniowo, co sugeruje złożoność czasową zbliżoną do $O(n \log n)$, ale z pewnym kosztem dodatkowym wynikającym z obsługi dwóch pivotów.
Wzrost **czasu** jest bliski liniowej zależności od liczby elementów, co sugeruje, że implementacja jest dobrze zoptymalizowana.
Klasyczny QuickSort zazwyczaj wymaga około $O(n \log n)$ porównań i operacji, a QuickSort2 wydaje się osiągać podobny wynik, ale z większą liczbą przypisań, co wynika z bardziej złożonego procesu dzielenia tablicy.

Zadanie 2

```
void COUNTINGSORT(int A[], int n, int exp, int d) {
    int B[n];
    int C[d] = {0};

    for (int i = 0; i < n; i++) {
        int j = (A[i] / exp) % d;
        C[j]++;
    }

    for (int i = 1; i < d; i++) {
        C[i] += C[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        int j = (A[i] / exp) % d;
        B[C[j] - 1] = A[i];
        C[j]--;
    }

    for (int i = 0; i < n; i++) {
        A[i] = B[i];
    }
}
```

Działanie funkcji COUNTINGSORT:

- Tworzenie pomocniczych tablic: B[n]: Tablica wynikowa, która będzie zawierać posortowane liczby. C[d]: Tablica zliczająca wystąpienia każdej wartości na danej pozycji

- Zliczanie wystąpień dla danej cyfry:

  Dla każdej liczby w tablicy A[], wyciągana jest cyfra na pozycji zależnej od exp . Cyfra ta jest obliczana przez $(A[i] \div \text{exp}) \mod d$, gdzie i to indeks w tablicy. Tablica C[] na końcu zawiera liczbę wystąpień każdej wartości w danym miejscu.

- Przekształcanie tablicy C[] na tablicę sum: Zaczynając od indeksu 1, do każdego elementu w tablicy C[] dodawana jest wartość poprzedniego elementu. Dzięki temu C[i] zawiera informację o liczbie elementów mniejszych lub równych i na danej pozycji. Ta operacja przekształca tablicę z liczby wystąpień na liczbę pozycji, na których powinny znaleźć się odpowiednie elementy w tablicy wynikowej B[].

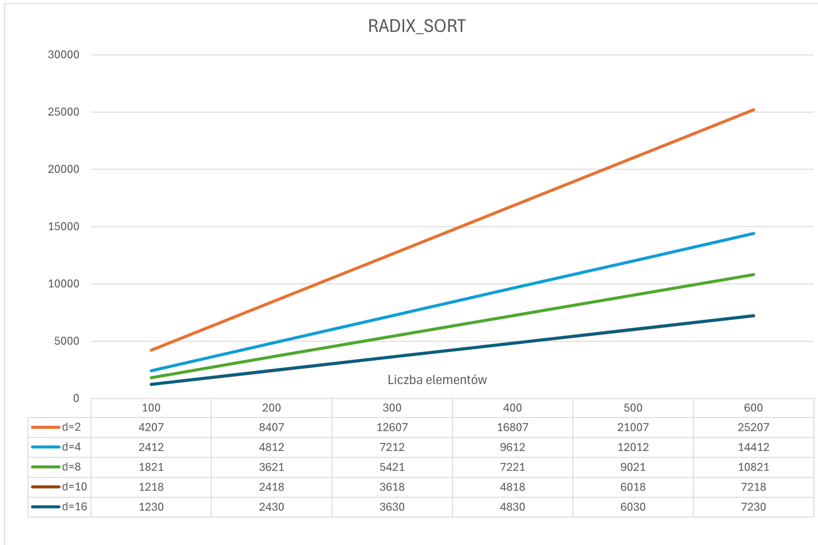
- Tworzenie tablicy wynikowej B[]: Rozpoczynamy iterację przez tablicę A[] od końca. Jest to ważne, aby nie zmieniać kolejności elementów, które mają tę samą cyfrę na danej pozycji. Dla każdego elementu w A[], wyliczamy jego cyfrę na danej pozycji, a następnie umieszczamy ten element w odpowiednim miejscu w tablicy B[] zgodnie z liczbą wystąpień w tablicy C[]. Po umieszczeniu elementu, zmniejszamy wartość w C[], aby odpowiednio wskazać kolejne dostępne miejsce.

- Na końcu, po posortowaniu liczb na danej pozycji, zawartość tablicy B[] jest kopiowana do tablicy A[], aby tablica A[] zawierała teraz posortowane liczby.

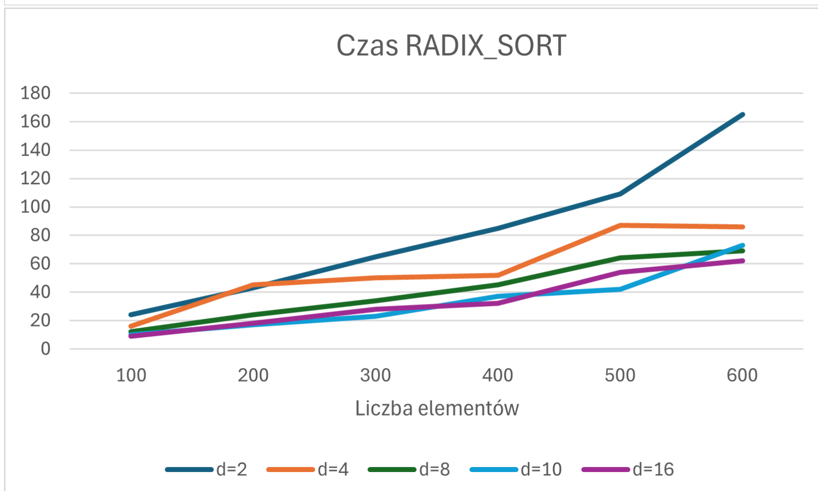
```
void RADIX_SORT(int A[], int n, int d, int k) {
    for (int i = 0, exp = 1; i < k; i++, exp *= d) {
        COUNTINGSORT(A, n, exp, d);
    }
}
```

Działanie funkcji RADIX_SORT:

- Zmienna exp reprezentuje wykładnik, który określa, na jaką pozycję wartości będziemy sortować w tej iteracji. Rozpoczynamy od exp = 1 , a w każdej kolejnej iteracji exp będzie mnożone przez d, aby przechodzić na kolejne pozycje wartości.
- Funkcja wykonuje pętlę for o k iteracjach. k to liczba wartości, więc w każdej iteracji sortujemy liczby według jednej pozycji. W każdej iteracji zmienia się wykładnik exp, który wskazuje, którą wartość w liczbach będziemy analizować.
- W każdej iteracji, dla danej pozycji wartości (określonej przez exp), wywoływana jest funkcja COUNTINGSORT, która sortuje liczby w tablicy A[] na podstawie tej konkretnej wartości.
- W każdej iteracji pętli exp jest mnożone przez d, co powoduje, że w kolejnej iteracji funkcja COUNTINGSORT będzie sortować liczby według następnej wartości.
- Proces jest powtarzany k razy. Po wykonaniu tych wszystkich iteracji tablica A[] jest posortowana.



	100	200	300	400	500	600
d=2	4207	8407	12607	16807	21007	25207
d=4	2412	4812	7212	9612	12012	14412
d=8	1821	3621	5421	7221	9021	10821
d=10	1218	2418	3618	4818	6018	7218
d=16	1230	2430	3630	4830	6030	7230



Liczba przypisań i porównań:

- Liczba przypisań rośnie praktycznie liniowo wraz ze wzrostem liczby elementów n.
- Zwiększenie podstawy d (systemu liczbowego) zmniejsza liczbę przypisań, ponieważ mniej iteracji jest potrzebnych do przetworzenia elementów.
- Dla $d = 16$, liczba przypisań jest praktycznie najmniejsza, podczas gdy dla $d = 2$ — największa.

Czas wykonania:

- Czas sortowania generalnie zmniejsza się wraz ze wzrostem podstawy d.

- Dla większych d, liczba iteracji w głównej pętli algorytmu jest mniejsza, co prowadzi do oszczędności czasu, mimo że pojedynczy krok może mieć większy narzut obliczeniowy.

Efektywność dla różnych podstaw:

- Podstawy $d = 8$, $d = 10$, i $d = 16$ wydają się najbardziej efektywne zarówno pod względem liczby przypisań, jak i czasu wykonania.
- Dla $d = 2$, algorytm wykonuje znacząco więcej operacji, co czyni go najmniej efektywnym.

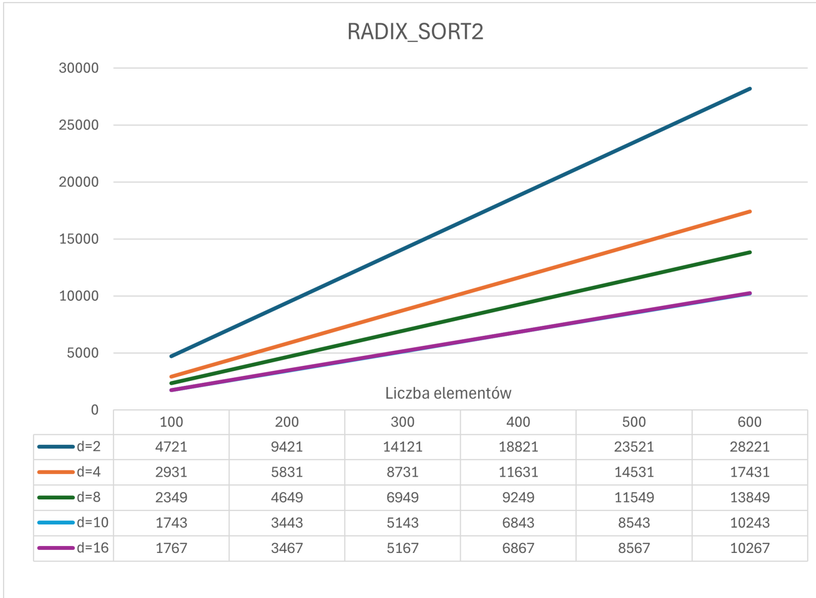
```
void RADIX_SORT2(int A[], int n, int d, int k) {
    int liczba_dodatnich = 0, liczba_ujemnych = 0;

    for (int i = 0; i < n; i++) {
        if (A[i] >= 0) {
            liczba_dodatnich++;
        } else {
            liczba_ujemnych++;
        }
    }

    int dodatnie[liczba_dodatnich];
    int ujemne[liczba_ujemnych];

    int dod = 0, uje = 0;
    for (int i = 0; i < n; i++) {
        if (A[i] >= 0) {
            dodatnie[dod++] = A[i];
        } else {
            ujemne[uje++] = -A[i];
        }

        if (liczba_dodatnich > 0) {
            RADIX_SORT(dodatnie, liczba_dodatnich, d, k);
        }

        if (liczba_ujemnych > 0) {
            RADIX_SORT(ujemne, liczba_ujemnych, d, k);
        }

        int x = 0;

        for (int i = liczba_ujemnych - 1; i >= 0; i--) {
            A[x++] = -ujemne[i];
```

```
    }

    for (int i = 0; i < liczba_dodatnich; i++) {
        A[x++] = dodatnie[i];
    }
}
```

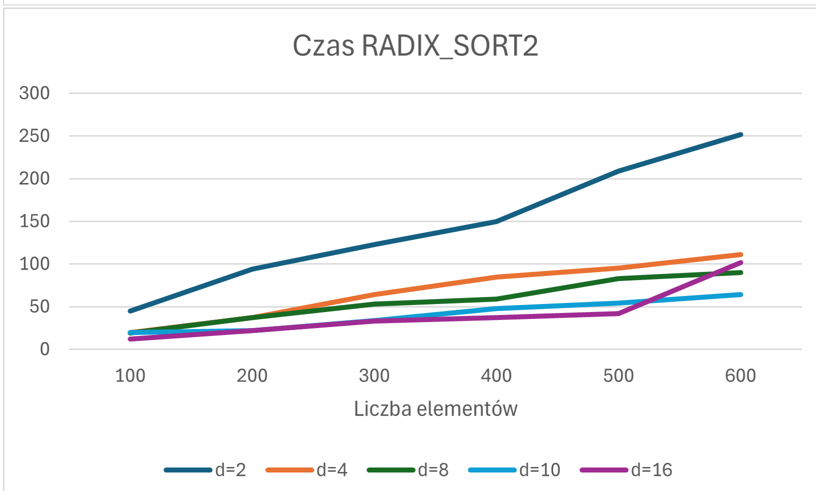
Działanie funkcji RADIX_SORT2:

- Zliczanie liczb dodatnich i ujemnych: Funkcja zaczyna od przejścia przez wszystkie elementy tablicy A[], licząc liczbę liczb dodatnich i liczb ujemnych.
- Inicjalizacja tablic pomocniczych: Tworzone są dwie tablice: dodatnie[], ujemne[]. Inicjalizowane są również zmienne dod i uje.
- Przepisanie liczb do odpowiednich tablic: Funkcja przechodzi przez tablicę A[] i, w zależności od znaku liczby, umieszcza ją w odpowiedniej tablicy, ale przed dodaniem każda liczba ujemna jest zmieniana na liczbę dodatnią.
- Sortowanie obu tablic: Jeśli istnieją liczby dodatnie, wywoływana jest funkcja RADIX_SORT. Jeśli istnieją liczby ujemne , wywoływana jest funkcja RADIX_SORT.
- Kopiowanie liczb z tablicy ujemnych i dodatnich do oryginalnej tablicy:

  Liczby ujemne są kopiowane do tablicy A[] w odwrotnej kolejności, dlatego że liczby ujemne powinny być w porządku malejącym po dodaniu minusa. Następnie liczby dodatnie są kopiowane do tablicy A[] w kolejności rosnącej.



	100	200	300	400	500	600
d=2	4721	9421	14121	18821	23521	28221
d=4	2931	5831	8731	11631	14531	17431
d=8	2349	4649	6949	9249	11549	13849
d=10	1743	3443	5143	6843	8543	10243
d=16	1767	3467	5167	6867	8567	10267



Liczba przypisań i porównań:

- Liczba przypisań rośnie prawie liniowo wraz ze wzrostem liczby elementów n.
- Większa podstawa d zmniejsza liczbę przypisań, ponieważ liczba iteracji algorytmu spada. Najmniej przypisań występuje dla $d = 16$ i $d = 10$, a najwięcej dla $d = 2$.

Czas wykonania:

- Czas działania maleje wraz ze wzrostem podstawy d, ponieważ większe d oznacza mniej iteracji w algorytmie.
- Dla $d = 16$ i $d = 10$, algorytm osiąga najlepsze wyniki czasowe, podczas gdy dla $d = 2$ działa najwolniej.

Efektywność dla różnych podstaw:

- Podstawy $d = 8$, $d = 10$, i $d = 16$ zapewniają dobrą równowagę między liczbą przypisań a czasem wykonania.
- Obsługa ujemnych liczb jest skuteczna dzięki rozdzieleniu ich od dodatnich i niezależnemu sortowaniu.

Zadanie 3

```
struct Wezel {
    double wartosc;
    Wezel* prev;
    Wezel* next;

    Wezel(double war) : wartosc(war), prev(nullptr), next(nullptr) {}
};

struct Lista {
    Wezel* head;

    Lista() : head(nullptr) {}
};
```

Struktura Wezel reprezentuje pojedynczy element listy. Każdy węzeł zawiera trzy składowe:

- wartosc (typ double): Przechowuje wartość przechowywaną w danym węźle listy.
- prev (typ Wezel*): Wskaźnik do poprzedniego węzła w liście.
- next (typ Wezel*): Wskaźnik do następnego węzła w liście.
- Konstruktor Wezel(double war): Jest to konstruktor, który przyjmuje wartość typu double i ustawia ją jako wartość węzła (wartosc). Ponadto, wskaźniki prev i next są inicjalizowane jako nullptr, ponieważ na początku węzeł nie jest połączony z żadnym innym węzłem.

Struktura Lista reprezentuje całą listę. Zawiera ona tylko jedną składową:

- head (typ Wezel*): Jest to wskaźnik do pierwszego węzła w liście. Jeśli lista jest pusta, head jest ustawiony na nullptr.
- Konstruktor Lista(): Jest to konstruktor, który inicjalizuje wskaźnik head jako nullptr. Początkowo lista jest pusta, ponieważ nie ma żadnych węzłów.

```
void LIST_INSERT(Lista& L, Wezel* x) {
    x->next = L.head;
    x->prev = nullptr;

    if (L.head != nullptr) {
        L.head->prev = x;
    }
    L.head = x;
}
```

```cpp
void LIST_DELETE(Lista& L, Wezel* x) {
    if (x->prev != nullptr) {
        x->prev->next = x->next;
    } else {
        L.head = x->next;
    }

    if (x->next != nullptr) {
        x->next->prev = x->prev;
    }

    delete x;
}

Wezel* LIST_SEARCH(Lista& L, double k) {
    Wezel* x = L.head;

    while (x != nullptr && x->wartosc != k) {
        x = x->next;
    }

    return x;
}

void PRINT_LIST(Lista& L) {
    Wezel* x = L.head;
    while (x != nullptr) {
        cout << x->wartosc << " ";
        x = x->next;
    }
    cout << endl;
}
```

1. LIST_INSERT(Lista& L, Wezel x)

Ta funkcja wstawia nowy węzeł `x` na początek listy `L`.

- Ustawia `x->next` na dotychczasową głowę listy (`L.head`), a `x->prev` na `nullptr`.
- Jeśli lista nie jest pusta (`L.head != nullptr`), aktualizuje wskaźnik `L.head->prev` na `x`.
- Ustawia `L.head` na `x`, który staje się nowym pierwszym węzłem.

2. LIST_DELETE(Lista& L, Wezel x)

Ta funkcja usuwa węzeł `x` z listy `L`.

- Jeśli `x` nie jest pierwszym węzłem (`x->prev != nullptr`), aktualizuje wskaźnik `x->prev->next` na `x->next`.
- Jeśli `x` jest pierwszym węzłem (`x->prev == nullptr`), aktualizuje `L.head` na `x->next`.
- Jeśli `x->next != nullptr`, aktualizuje wskaźnik `x->next->prev` na `x->prev`.
- Na końcu usuwa węzeł `x` za pomocą `delete`.

3. LIST_SEARCH(Lista& L, double k)

Funkcja ta przeszukuje listę `L` w poszukiwaniu węzła o wartości `k`.

- Zaczyna od `L.head` i przechodzi przez kolejne węzły, porównując ich wartość z `k`.
- Jeśli znajdzie węzeł o wartości `k`, zwraca wskaźnik do niego.
- Jeśli nie znajdzie takiego węzła, zwraca `nullptr`.

4. PRINT_LIST(Lista& L)

Funkcja ta wypisuje wszystkie wartości w liście `L` w kolejności od głowy listy do ostatniego węzła.

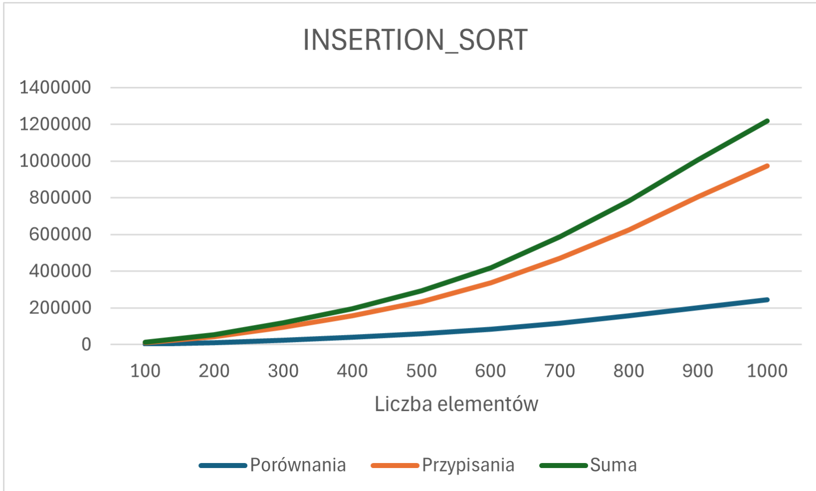
- Zaczyna od `L.head` i przechodzi po wszystkich węzłach, wypisując ich wartości.
- Po wypisaniu wszystkich wartości, wypisuje znak nowej linii.

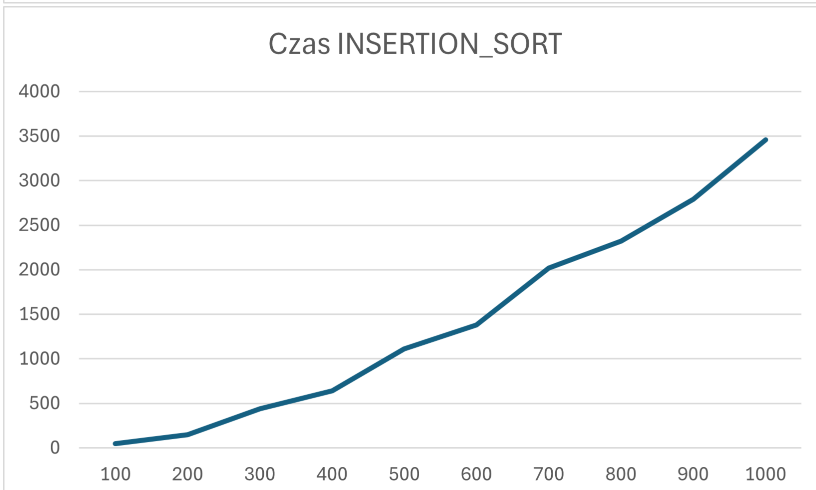
5. INSERTION_SORT(Lista& L)

```
void INSERTION_SORT(Lista& L) {
    if (L.head == nullptr || L.head->next == nullptr) {
        return;
    }

    Wezel* x = L.head->next;
    while (x != nullptr) {
        Wezel* Key = x;
        Wezel* prevKey = x->prev;

        while (prevKey != nullptr && prevKey->wartosc > Key->wartosc) {
            swap(prevKey->wartosc, Key->wartosc);
            Key = prevKey;
```

```
            prevKey = prevKey->prev;
        }

        x = x->next;
    }
}
```

- Funkcja rozpoczyna działanie od sprawdzenia, czy lista jest pusta lub zawiera tylko jeden element. W takim przypadku sortowanie nie jest potrzebne i funkcja kończy działanie.
- Następnie, funkcja ustawia wskaźnik `x` na drugi węzeł listy, ponieważ pierwszy węzeł uznawany jest za ”posortowany” w kontekście tego algorytmu.
- Dla każdego węzła na liście, funkcja porównuje jego wartość z wartością poprzednich węzłów. Jeśli wartość bieżącego węzła jest mniejsza niż wartość poprzedniego, element jest przesuwany do odpowiedniej pozycji w ”posortowanej części” listy.
- Wstawianie elementu polega na iteracyjnym porównywaniu i zamienianiu wartości węzłów, aż element znajdzie swoje miejsce w posortowanej części listy.
- Proces ten powtarza się dla wszystkich węzłów w liście, aż cała lista zostanie posortowana.





Funkcja insertion ma rosnący czas wykonania w miarę zwiększania liczby elementów. Czas rośnie nieliniowo, szczególnie dla większej liczby elementów. Wraz ze wzrostem liczby elementów rośnie liczba porównań i przypisań.

Zadanie 4

```
void BUCKET_SORT(double A[], int n) {
    Lista B[n];

    for (int j = 0; j < n; j++) {
        B[j] = Lista();
    }

    for (int i = 0; i < n; i++) {
        int index = static_cast<int>(n * A[i]);
        Wezel* nowy = new Wezel(A[i]);
        LIST_INSERT(B[index], nowy);
    }

    for (int j = 0; j < n; j++) {
        INSERTION_SORT(B[j]);
    }

    int i = 0;
    for (int j = 0; j < n; j++) {
        Wezel* x = B[j].head;
        while (x != nullptr) {
            A[i++] = x->wartosc;
            x = x->next;
        }
    }
}
```

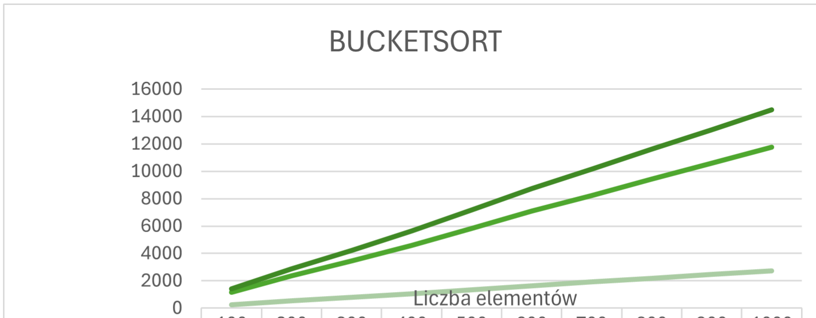
Algorytm **BUCKET_SORT** dzieli dane na kilka "kubełków" (ang. *buckets*), sortuje każdy kubełek osobno (. *Insertion Sort*), a następnie łączy posortowane kubełki w jedną posortowaną tablicę.

1. **Inicjalizacja kubełków:** Algorytm tworzy tablicę B, która zawiera n pustych list, reprezentujących kubełki. Każdy kubełek przechowuje elementy, które zostaną przypisane do niego na podstawie wartości z tablicy wejściowej A. Dla każdego kubełka tworzona jest pusta lista.

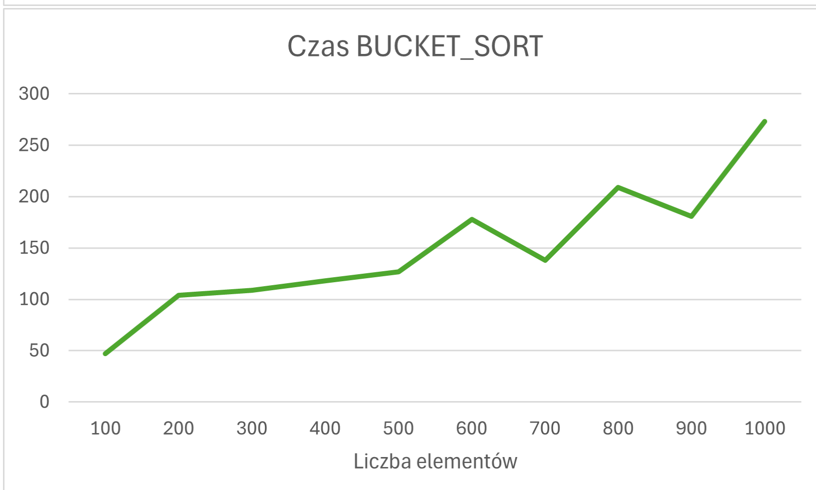
2. **Rozdzielanie elementów do kubełków:** Każdy element z tablicy A jest przypisywany do odpowiedniego kubełka na podstawie swojej wartości. Obliczamy indeks kubełka dla każdego elementu $A[i]$, który zależy od wartości $A[i]$ oraz liczby kubełków n.

3. **Sortowanie wewnętrzne kubełków:** Po przypisaniu wszystkich elementów do kubełków, każdy kubełek jest sortowany indywidualnie. Ponieważ kubełki zawierają elementy z określonego przedziału, sortowanie w każdym kubełku jest szybkie.

4. **Łączenie posortowanych kubełków:** Po posortowaniu każdego kubełka elementy z kubełków są zbierane i kopiowane z powrotem do tablicy A. Proces ten polega na przejściu przez wszystkie kubełki i przeniesieniu posortowanych elementów z listy węzłów do tablicy wynikowej.



	100	200	300	400	500	600	700	800	900	1000
Porównania	263	541	792	1054	1345	1643	1909	2190	2458	2727
Przypisania	1152	2344	3436	4595	5832	7080	8233	9437	10591	11763
Suma	1415	2885	4228	5649	7177	8723	10142	11627	13049	14490



Algorytm Bucket Sort wykazuje wzrost czasu wykonania oraz liczby operacji (porównań i przypisań) wraz z liczbą elementów. Czas rośnie nieliniowo. Dla mniejszych zbiorów jest stosunkowo szybki, ale dla dużych danych jego wydajność spada, co ogranicza jego zastosowanie przy dużych zbiorach.

```
void BUCKET_SORT2(double A[], int n) {
    double minimalna = A[0];
    double maksymalna = A[0];
```

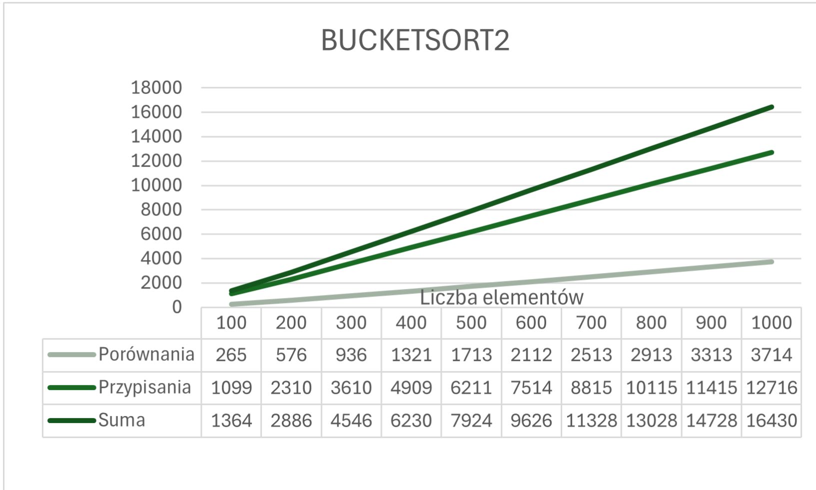
```
    for (int i = 1; i < n; i++) {
        if (A[i] < minimalna) minimalna = A[i];
        if (A[i] > maksymalna) maksymalna = A[i];
    }

    if (maksymalna == minimalna) {
        for (int i = 0; i < n; i++) {
            A[i] = minimalna;
        }
        return;
    }

    Lista B[n];

    for (int j = 0; j < n; j++) {
        B[j] = Lista();
    }

    for (int i = 0; i < n; i++) {
        int index = static_cast<int>(n*(A[i]-minimalna)/(maksymalna-minimalna));
        if (index == n) index--;
        Wezel* nowy = new Wezel(A[i]);
        LIST_INSERT(B[index], nowy);
    }

    for (int j = 0; j < n; j++) {
        INSERTION_SORT(B[j]);
    }

    int i = 0;
    for (int j = 0; j < n; j++) {
        Wezel* x = B[j].head;
        while (x != nullptr) {
            A[i++] = x->wartosc;
            x = x->next;
        }
    }
}
```

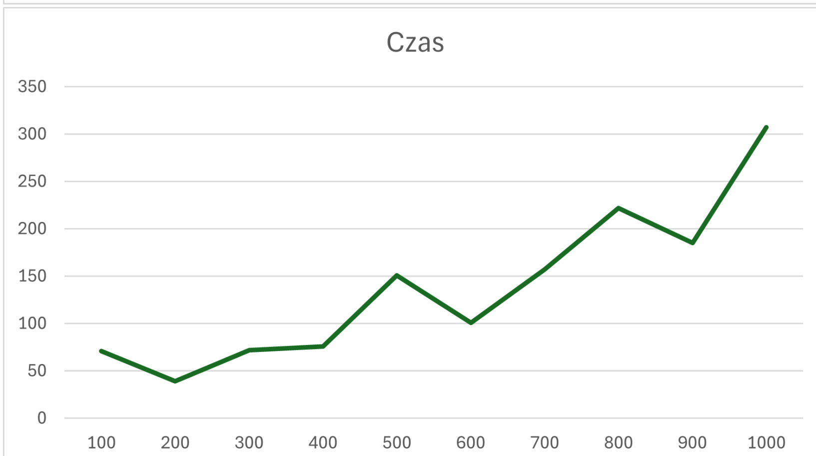
Algorytm **BUCKET_SORT2** działa na zasadzie dzielenia elementów na "kubełki", które są następnie sortowane indywidualnie, a następnie scalane w jedną posortowaną tablicę.

1. **Znalezienie minimalnej i maksymalnej wartości w tablicy A:**

    - Algorytm rozpoczyna od obliczenia minimalnej i maksymalnej wartości w tablicy A. Te wartości będą potrzebne do prawidłowego rozdzielenia elementów na kubełki.

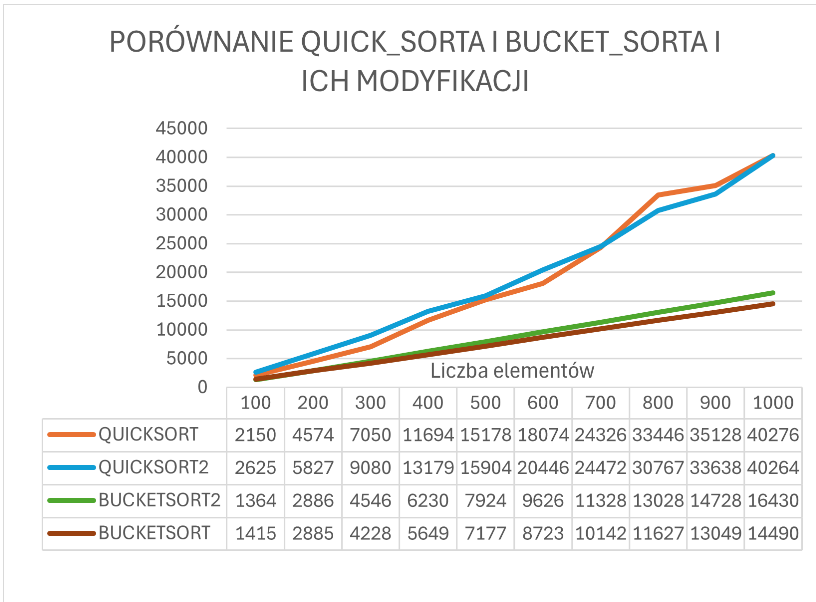
2. **Inicjalizacja kubełków:**
   - Tworzymy tablicę kubełków B, która składa się z n pustych list.
3. **Rozdzielanie elementów do kubełków:**
   - Dla każdego elementu $A[i]$ z tablicy A obliczamy indeks kubełka, do którego należy go przypisać.
   - Indeks kubełka obliczany jest na podstawie wzoru:
   $$\text{index} = \left\lfloor n \cdot \frac{A[i] - \text{minimalna}}{\text{maksymalna} - \text{minimalna}} \right\rfloor$$
   - Indeks kubełka jest liczony jako liczba całkowita z zakresu od 0 do $n-1$, co pozwala na odpowiednie przypisanie elementów do kubełków.
   - Jeśli indeks wynosi n (np. dla wartości równych maksymalnej), zmniejszamy go o 1, aby uniknąć przekroczenia zakresu.
   - Element $A[i]$ jest następnie dodawany do odpowiedniego kubełka $B[\text{index}]$ za pomocą funkcji `LIST_INSERT`.
4. **Sortowanie kubełków:**
   - Po przypisaniu wszystkich elementów do kubełków, każdy kubełek jest sortowany indywidualnie.
   - Sortowanie odbywa się przy użyciu algorytmu sortowania przez wstawianie (Insertion Sort) na każdej z list w kubełkach.
5. **Łączenie posortowanych kubełków:**
   - Po posortowaniu kubełków algorytm łączy posortowane elementy z każdego kubełka z powrotem do tablicy A.



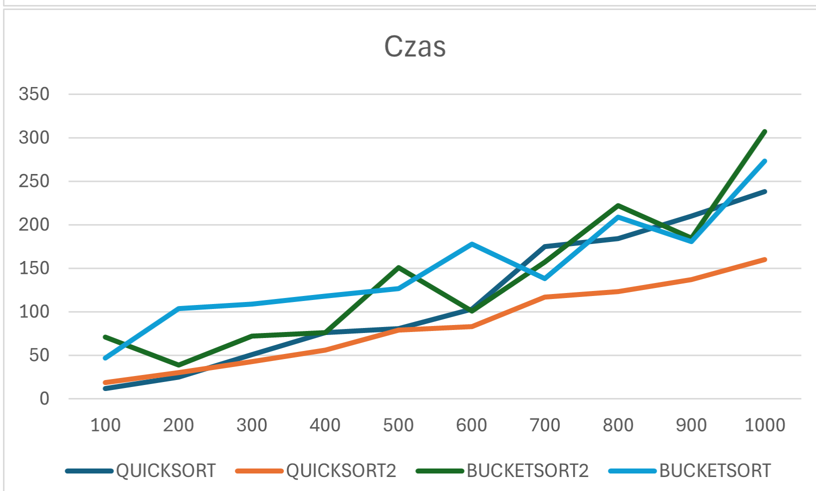


W Bucket_Sort2 czas wykonania rośnie nieliniowo z liczbą elementów. Dla małych zbiorów działa efektywnie, jednak przy większych danych czas wykonania i liczba operacji rosną znacząco, co ogranicza jego wydajność przy dużych zbiorach.

Quick_Sort i Bucket_Sort



	100	200	300	400	500	600	700	800	900	1000
QUICKSORT	2150	4574	7050	11694	15178	18074	24326	33446	35128	40276
QUICKSORT2	2625	5827	9080	13179	15904	20446	24472	30767	33638	40264
BUCKETSORT2	1364	2886	4546	6230	7924	9626	11328	13028	14728	16430
BUCKETSORT	1415	2885	4228	5649	7177	8723	10142	11627	13049	14490



Porównania i przypisania:

- **BUCKETSORT2** jest najbardziej efektywnym algorytmem pod względem liczby przypisań i porównań dla wszystkich rozmiarów tablic.
- **BUCKETSORT** jest nieznacznie gorszy od **BUCKETSORT2**, ale nadal bardziej efektywny niż **QUICKSORT**.

- **QUICKSORT** i **QUICKSORT2** są do siebie bardzo podobne, a ich wydajność jest gorsza w porównaniu do **BUCKETSORT** i **BUCKETSORT2**, szczególnie w przypadku mniejszych dużej ilości elementów.

Czas:

- **QUICKSORT2** wydaje się być najszybszym algorytmem spośród wszystkich, ponieważ jego czasy wykonania są najniższe w porównaniu do pozostałych algorytmów dla wszystkich rozmiarów tablic.
- **BUCKETSORT** i **BUCKETSORT2** mają nieregularne czasy wykonania, które rosną z liczbą elementów. Niemniej jednak, ich czas rośnie szybciej niż czas dla **QUICKSORT2**, szczególnie przy większej liczbie elementów.
- **QUICKSORT** i **QUICKSORT2** mają stosunkowo regularny wzrost czasów wykonania, jednak **QUICKSORT2** wypada lepiej pod względem wydajności niż **QUICKSORT**.