Algorytmy i Struktury Danych
Sprawozdanie 2

Emilia Łagoda
Numer albumu: 287361

Spis treści

0.1 Wstęp

Celem tego ćwiczenia jest porównanie różnic pomiędzy standardową wersją trzech algorytmów sortujących, a ich modyfikacjami, ze szczególną uwagą na zmiany w czasie wykonywania, liczbie wykonanych przypisań oraz porównań danych. Wybrane algorytmy to Quick sort, również znany jako sortowanie szybkie, Radix Sort, czyli sortowanie pozycyjne i Bucket Sort, zwany też sortowaniem kubełkowym.

0.2 Metodyka Badań

Testowane są wcześniej wspomniane zmienne, czyli czas wykonywania, suma przypisań oraz suma porównań. Dla algorytmów Quick Sort oraz Bucket sort wykonane będzie po pięć powtórzeń dwudziestu testów na tablicach o liniowo rosnącej długości, wypełnionych losowo wygenerowanymi wartościami z zakresu $[0, 1000000]$. Dla algorytmu Radix, sprawdzane jest pięć podstaw, $2, 4, 8, 10, 16$, gdzie dla każdej podstawy wykonywane jest po pięć powtórzeń dwudziestu testów na rosnących tablicach, gdzie wartości generowane są o danej podstawie. Zakres rozmiaru tablic to od 5000000 do 100000000 dla Quick Sorta, od 500000 do 10000000 dla Radix sorta oraz od 250000 do 5000000 dla Bucket Sorta.

0.3 Quick Sort

Algorytm Quick Sort, to algorytm sortujący opierający się o szerszą rodzinę algorytmów dziel i zwyciężaj. Tradycyjnie, polega on na wybraniu elementu rozdzielającego (tzw. pivot), po czym podzieleniu względem tego elementu zbioru na dwie części, mniejszą-równą pivota oraz większą. Następnie, rekursywnie sortuje aż do uzyskania jednoelementowych podtablic. Ze względu na wykorzystanie rekursji, jego złożoność to $O(nlogn)$.

```
int partition(int arr[], int p, int k){
    int x = arr[k];
    int i = p - 1;

    for (int j = p; j < k; j++){
        if (arr[j] <= x){
            i++;
            swap(arr[i], arr[j]);
            assignments += 2;
        }
    }

    swap(arr[i+1], arr[k]);
    assignments += 2;
    return i+1;
}

void quick_sort(int arr[], int p, int k){
    if (p < k){
        comparisons++;
```

```
        int s = partition(arr, p, k);

        quick_sort(arr, p, s-1);
        quick_sort(arr, s+1, k);
    }
}
```

Zmodyfikowny quick sort, również znany jako Dual Pivot Quick Sort, działa podobnie do standardowej wersji, jednak wybierane są dwa elementy rozdzielające. Wtedy, zbiór dzielony jest na część od początku do pierwszego pivota, indeksy pomiędzy pivotami oraz od drugiego pivota do końca zbioru. W tej wersji również rekursywnie dzielimy tablicę do otrzymania jednoelementowych podtablic.

```
void three_way_partition(int arr[], int p, int k, int& i, int& j){
    int x = arr[k];
    int low = p;
    int mid = p;
    int high = k;

    while (mid <= high){
        if (arr[mid] < x){
            comparisons++;
            swap(arr[low], arr[mid]);
            assignments += 2;
            low++;
            mid++;
        }
        else if (arr[mid] == x){
            comparisons++;
            mid++;
        }
        else{
            comparisons++;
            swap(arr[mid], arr[high]);
            assignments += 2;
            high--;
        }
    }

    i = low -1;
    j = mid;
}

void quick_sort(int arr[], int p, int k){
    if (p < k){
        comparisons++;
```

```
        int i, j;
        three_way_partition(arr, p, k, i, j);

        quick_sort(arr, p, i);
        quick_sort(arr, j, k);
    }
}
```

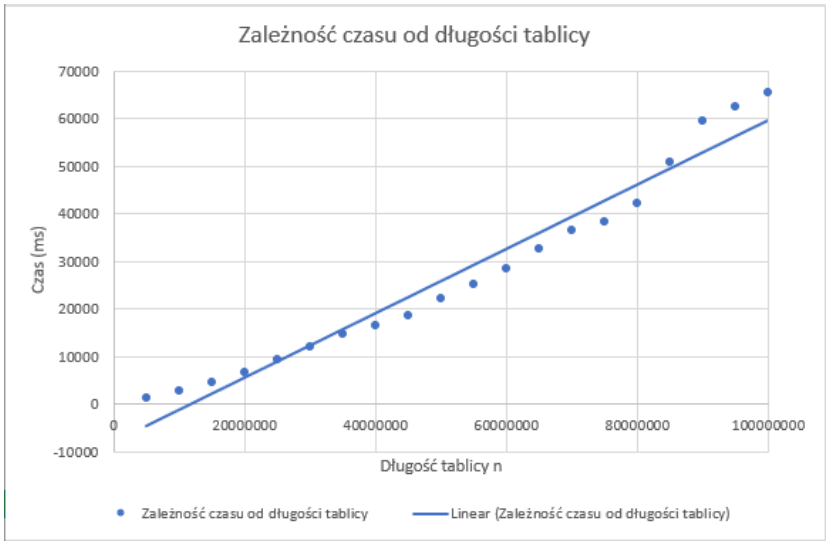
0.3.1 Standardowy Quick Sort

Średnia wyników

Tabela 1: Średnia wyników dla Standardowego Quick Sort

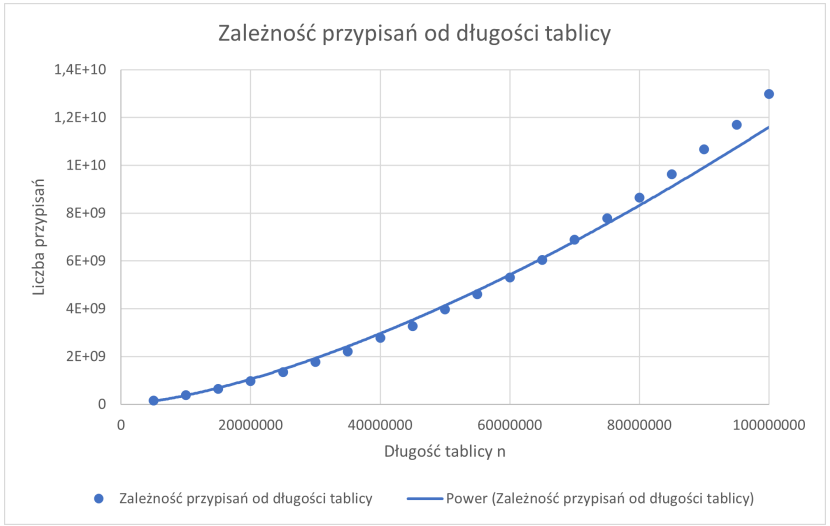
Długość tablicy n	**Czas [ms]**	**Przypisania**	**Porównania**
5000000	1332	163535930.8	4054642.4
10000000	2837.4	392004728.4	9000818.6
15000000	4571.6	648929098.8	14000008.2
20000000	6488	978837110.4	19000001.2
25000000	9375.8	1347901760	24000001
30000000	12023.6	1775085768	29000001
35000000	14569.2	2204949796	34000001
40000000	16373.2	2790602264	39000001
45000000	18655.6	3277330777	44000001
50000000	22105	3966179801	49000001
55000000	25119.6	4613264172	54000001
60000000	28325.4	5303936056	59000001
65000000	32666.6	6038938446	64000001
70000000	36529.2	6895958419	69000001
75000000	38341	7793969168	74000001
80000000	42049.8	8645986908	79000001
85000000	50832.6	9627044241	84000001
90000000	59365.6	10664162558	89000001
95000000	62449.8	11701710200	94000001
100000000	65318.6	12991440146	99000001

Analiza wykresów



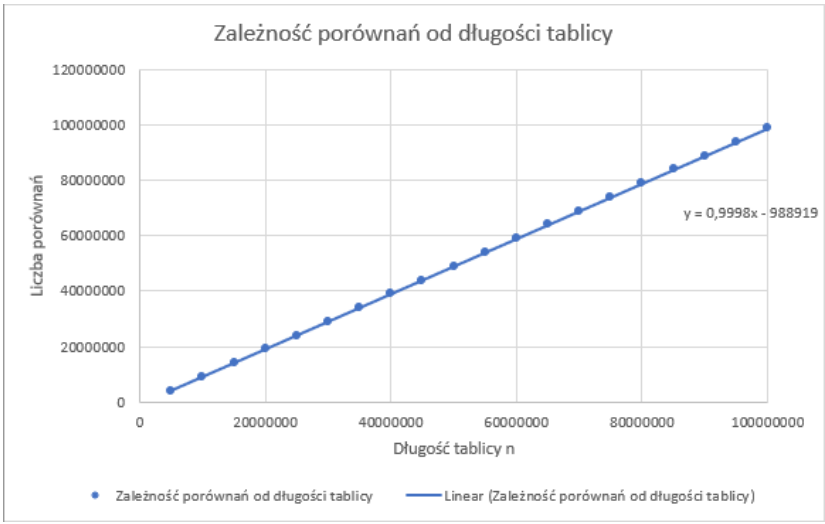
Rysunek 1: Zależność czasu wykonywania od długości tablicy dla algorytmu Quick Sort

Mimo względnie liniowego rozkładu, korelacja pomiędzy punktami a funkcją liniową jest słaba. Występują zauważalne skoki oraz stagnacje, co potwierdza teoretyczne założenia o złożoności $O(nlogn)$.



Rysunek 2: Zależność liczby przypisań od długości tablicy dla algorytmu Quick Sort

W przypadku zależności sumy przypisań od długości tablicy, funkcja przyjmuję formę potęgową. Zależnie od rozmiaru tablicy początkowej, algorytm dzieli zbiór aż nie otrzyma zbiorów jednoelementowych, więc dla liniowo rosnącego rozmiaru tablicy, suma porównań rośnie potęgowo.



Rysunek 3: Zależność liczby porównań od długości tablicy dla algorytmu Quick Sort

Suma porównań rośnie prawie idealnie liniowo. Niezależnie od liczby poziomów rekursji, czy początkowego uporządkowania danych, wartości są porównywane podobną ilość razy.

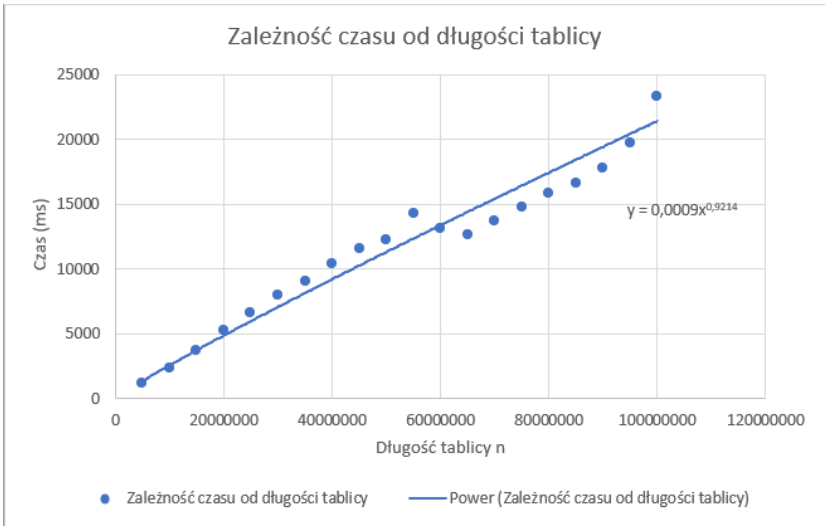
0.3.2 Dual Pivot Quick Sort

Średnia wyników

Tabela 2: Średnia wyników dla Dual Pivot Quick Sort

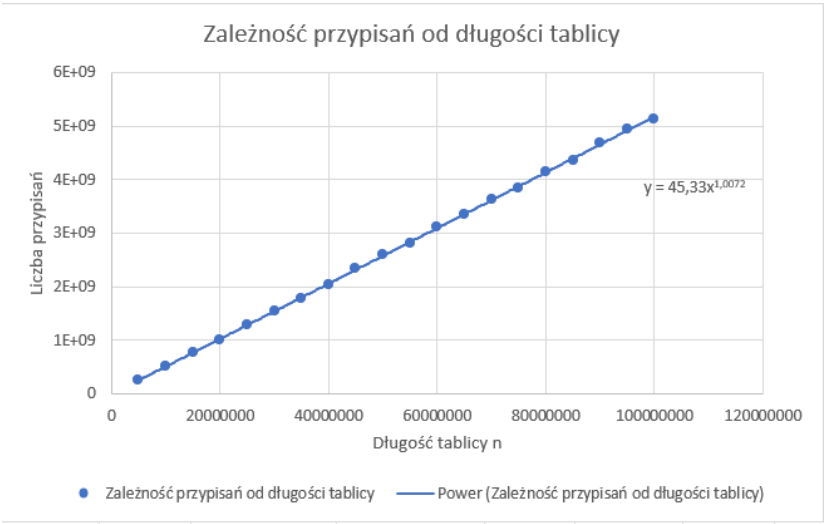
Długość tablicy n	Czas [ms]	Przypisania	Porównania
5000000	1180.8	254120113.2	133005372.4
10000000	2311.4	510152811.6	266075605.4
15000000	3661.6	766803629.2	399401809
20000000	5248.6	1011682700	526841351.8
25000000	6614.6	1280276995	666138499.4
30000000	7955.4	1545476986	803738494.8
35000000	9070.4	1786122008	929061005.8
40000000	10434	2046385515	1064192759
45000000	11612.6	2335924399	1213962202
50000000	12203	2592760656	1347380330
55000000	14241.2	2812997334	1462498669
60000000	13075	3123098723	1622549364
65000000	12674.2	3360064102	1746032053
70000000	13681.8	3628006210	1885003107
75000000	14727	3847922817	1999961411
80000000	15806	4140235070	2151117537
85000000	16601.8	4369515992	2270757998
90000000	17816.8	4689898775	2435949389
95000000	19710	4936189066	2564094535
100000000	23353.4	5143241819	2672620911

Analiza wykresów



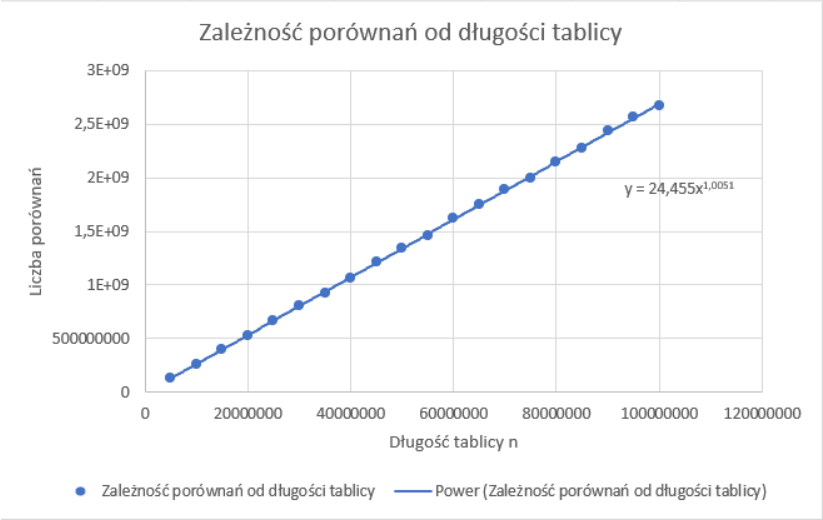
Rysunek 4: Zależność czasu wykonywania od długości tablicy dla Dual Pivot Quick Sort

Podobnie do tradycyjnego algorytmu Quick Sort, mimo zależności liniowej, widoczne są skoki i stagnacje, co ponownie sugeruje założoną złożoność $O(nlogn)$. Warto jednak zwrócić uwagę na zakres wartości na osi y, ponieważ dla algorytmu Dual Pivot Quick sort, wartość największa jest prawie trzykrotnie mniejsza. Ta poprawa wynika z wyższej liczby podzbiorów, na które dzielona jest początkwa tablica, co zmiejsza liczbę poziomów rekursji.



Rysunek 5: Zależność liczby przypisań od długości tablicy dla Dual Pivot Quick Sort

W porównaniu do standardowej wersji algorytmu, suma przypisań prezentuje zależność liniową. Ta poprawa jest skutkiem zmiejszonej liczby poziomów rekursji.



Rysunek 6: Zależność liczby porównań od długości tablicy dla Dual Pivot Quick Sort

Dual Pivot Quick sort również wykazuej zależność liniową pomiędzy rozmiarem tablicy a sumą porównań. Można jednak zauważyć wzrost w sumie, mimo zachowania zależności.

0.3.3 Wnioski

W obu przypadkach złożoność czasowa to $O(nlogn)$, co jest zgodne z założeniem teoretycznym.

0.4 Radix Sort

Radix sort to algorytm sortujący polegający na porównywaniu i porządkowaniu liczb względem znaczących pozycji. Zaczynając od najmniej znaczącej cyfry, iteruje przez zbiór uporządkowując wartości rosnąco względem i-tej pozycji, kończąc na rzędzie cyfr najbardziej znaczących dla wartości największej. Ten algorytm ma złożoność $O(d(n+k))$, gdzie d to długość klucza (równoważne liczba iteracji przez zbiór), n to liczba elementów zbioru, a k to liczba możliwych cyfr na danej pozycji (np. 2 dla systemu binarnego).

```
void counting_sort(int arr[], int n, int digitplace, int d){
    int* sorted_arr = new int[n];
    int* count = new int[d](); //inicjalizacja zerami

    for (int i = 0; i < n; i++){
        int digit = (arr[i]/digitplace)%d;
        count[digit]++;
        assignments++;
    }

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

```
    for (int i = n-1; i >= 0; i--){
        int digit = (arr[i] / digitplace) % d;
        sorted_arr[count[digit] - 1] = arr[i];
        count[digit]--;
        assignments += 2;
    }

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

    delete[] sorted_arr;
    delete[] count;

}


void radix_sort(int arr[], int n, int d){

    if (n <= 1) return; //base case
    comparisons++;

    int max = getMax(arr, n);

    for (int digitplace = 1; max/digitplace > 0; digitplace *= d){
        counting_sort(arr, n, digitplace, d);
    }

}
```

Modyfikacja w tym przypadku polega jedynie na rozszerzeniu zakresu liczb wchodzących o liczby ujemne. Aby poprawnie porządkować liczby ujemne, brana jest wartość bezwzględna największej liczby z zbiorze, a następnie, zbiór jest dzielony na dwie podtablice, liczby ujemne oraz nieujemne. Te podtablice sortowane są osobno algorytmem standardowego Radix Sorta, gdzie dla liczb ujemnych porządkowane są ich wartości bezwzględne. Posortowane tablice są scalane poprzez odwrócenie tablicy wartości ujemnych i przywracaniu znaku i łączenie wyników.

```
//counting_sort pozostaje bez zmian
void radix_sort(int arr[], int n, int d){
    if (n <= 1) return; //d case
    comparisons++;

    int negative_count = 0;
    int positive_count = 0; // dla u Ćatwienia 0 jest w positive count

    for (int i = 0; i < n; i++) {
```

```
        if (arr[i] < 0) {
            comparisons++;
            negative_count++;
        } else {
            positive_count++;
        }
    }

    int *negative = new int[negative_count];
    int *positive = new int[positive_count];

    int negative_index = 0;
    int positive_index = 0;

    for (int i = 0; i < n; i++){
        if (arr[i] < 0){
            comparisons++;
            negative[negative_index++] = -arr[i];
            assignments++;
        }
        else {
            positive[positive_index++] = arr[i];
            assignments++;
        }
    }


    if (positive_count > 0) {
        comparisons++;
        int maxpositive = getMaxAbs(positive, positive_count);
        for (int digitplace = 1; maxpositive / digitplace > 0; digitplace *= d
            counting_sort(positive, positive_count, digitplace, d);
        }
    }

    // Sortowanie liczb ujemnych (wartoŻci bezwzglŹdnych)
    if (negative_count > 0) {
        comparisons++;
        int maxNegative = getMaxAbs(negative, negative_count);
        for (int digitplace = 1; maxNegative / digitplace > 0; digitplace *= d
            counting_sort(negative, negative_count, digitplace, d);
        }

        // Odwracanie kolejnoŻci liczb ujemnych i przywracanie znaku
        for (int i = 0; i < negative_count / 2; i++) {
            int temp = negative[i];
            negative[i] = negative[negative_count - 1 - i];
```

```
            negative[negative_count - 1 - i] = temp;
            assignments += 2;
        }

        for (int i = 0; i < negative_count; i++) {
            negative[i] = -negative[i];
            assignments++;
        }
    }

    //    Łączenie    wynik w
    int index = 0;
    for (int i = 0; i < negative_count; i++) {
        arr[index++] = negative[i];
        assignments++;
    }
    for (int i = 0; i < positive_count; i++) {
        arr[index++] = positive[i];
        assignments++;
    }

    delete[] negative;
    delete[] positive;
}
```

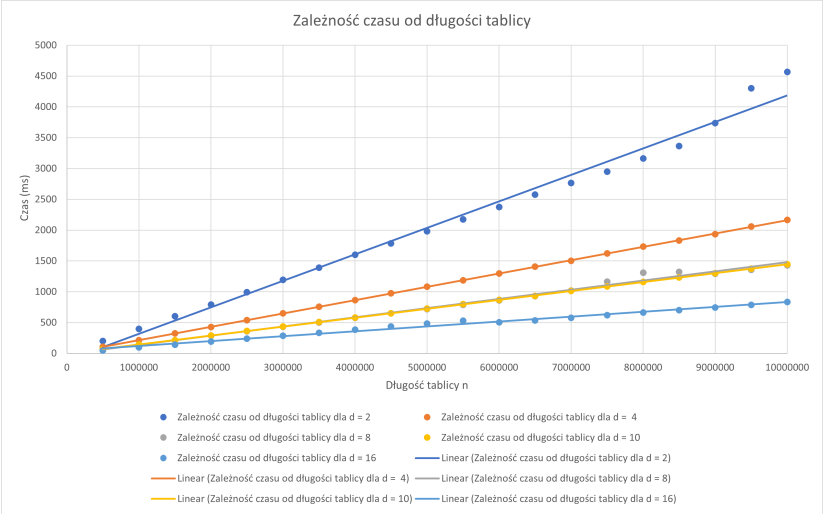
0.4.1 Standardowy Radix Sort

Średnia wyników

Tabela 3: Średnia wyników dla Standardowego Radix Sort dla podstawy $d = 2$

Długość tablicy n	Czas [ms]	Przypisania	Porównania
500000	202.4	36998008099	2639.6
1000000	398.4	37074008118	2651.6
1500000	603	37188008137	2664.4
2000000	790	37340008156	2680.6
2500000	991.8	37530008175	2693.8
3000000	1192	37758008194	2706.4
3500000	1390.8	38024008213	2715.8
4000000	1601	38328008232	2730
4500000	1784.8	38670008251	2742.4
5000000	1982.2	39050008270	2756.2
5500000	2175.6	39468008289	2770.2
6000000	2376.8	39924008308	2783.4
6500000	2575	40418008327	2798.4
7000000	2766	40950008346	2814
7500000	2950.2	41520008365	2826.6
8000000	3162.8	42128008384	2840.2
8500000	3364.6	42774008403	2854.2
9000000	3735.4	43458008422	2868
9500000	4301.8	44180008441	2881.6
10000000	4568.6	44940008460	2897.4

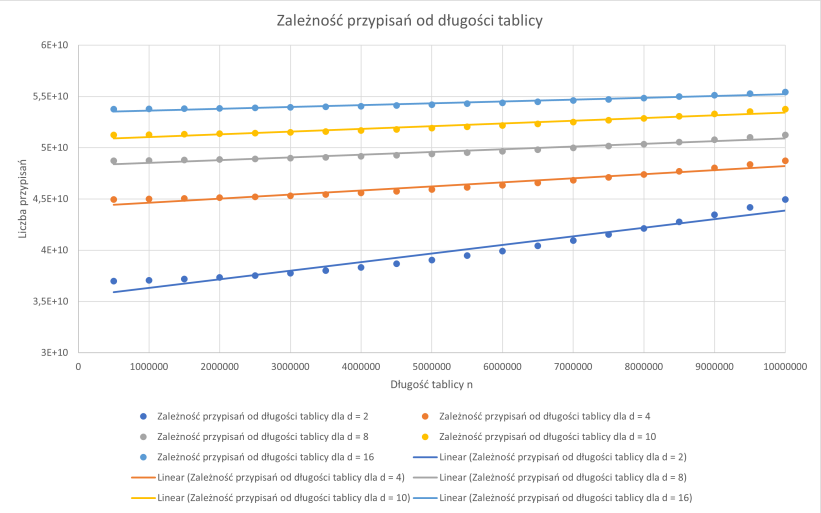
Analiza wykresów



Rysunek 7: Zależność czasu wykonywania od długości tablicy dla algorytmu Radix Sort

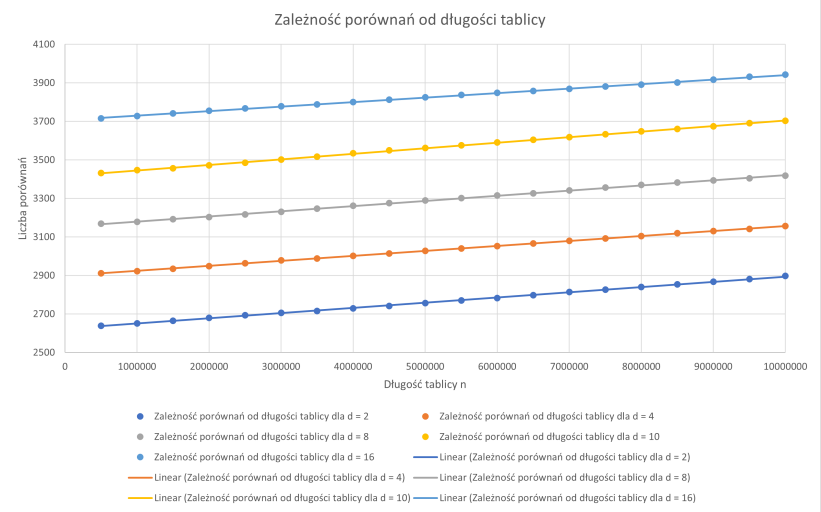
Dla każdej z pięciu podstaw, zależność czasu od rozmiaru tablicy początkowej jest liniowa, różnią się jedynie współczynnikiem. Jest to zgodne z założeniem teoretycznym złożoności

$O(d(n+k))$.



Rysunek 8: Zależność liczby przypisań od długości tablicy dla algorytmu Radix Sort

Zależność sumy przypisań od rozmiaru tablicy również zachowuje się w przewidywany sposób. Można zauważyć też wzrost wartości wraz z wzrostem podstawy d.



Rysunek 9: Zależność liczby porównań od długości tablicy dla algorytmu Radix Sort

Suma porównań podobnie do sumy przypisań jest liniowy z zauważalnym przesunięciem w górę wraz z rosnącą podstawą liczb. W przypadku porównań dla każdej podstawy, suma rośnie linowo z takim samym współczynnikiem.

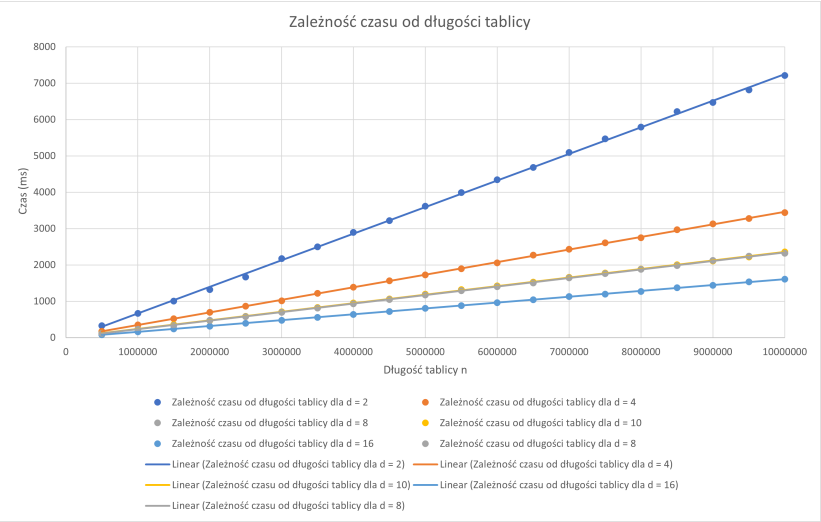
0.4.2 Zmodyfikowany Radix Sort

Średnia wyników

Tabela 4: Średnia wyników dla Zmodyfikowanego Radix Sort dla podstawy $d = 2$

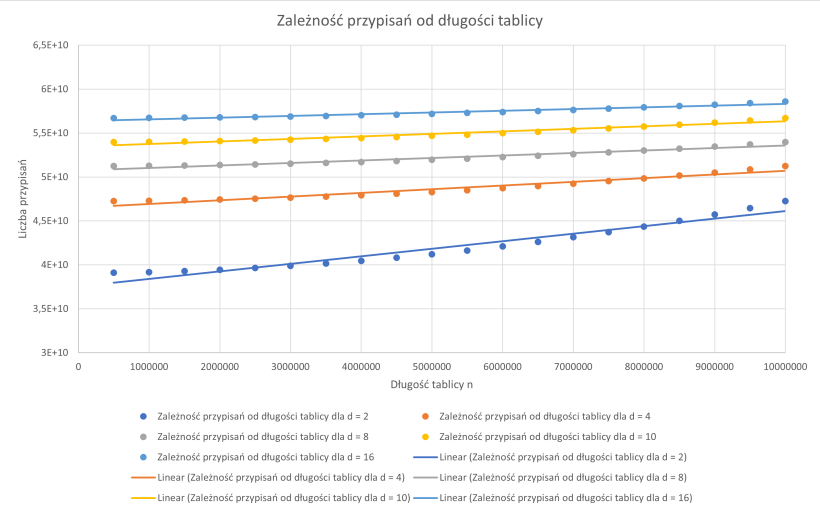
Długość tablicy n	Czas [ms]	Przypisania	Porównania
500000	332	39099008099	2849.6
1000000	668.6	39177008118	2865.2
1500000	1005.6	39294008137	2880.2
2000000	1322	39450008156	2895.4
2500000	1668.2	39645008175	2908.4
3000000	2173.8	39879008194	2922.4
3500000	2500	40152008213	2937.4
4000000	2896.8	40464008232	2951.4
4500000	3220.8	40815008251	2965.2
5000000	3612.8	41205008270	2978.4
5500000	3992.6	41634008289	2991.8
6000000	4347.6	42102008308	3006.6
6500000	4683.6	42609008327	3020.6
7000000	5098	43155008346	3033.4
7500000	5468.2	43740008365	3048.2
8000000	5795.4	44364008384	3068
8500000	6222.4	45027008403	3085
9000000	6472.6	45729008422	3098.2
9500000	6812.2	46470008441	3112.2
10000000	7212.8	47250008460	3126.6

Analiza wykresów



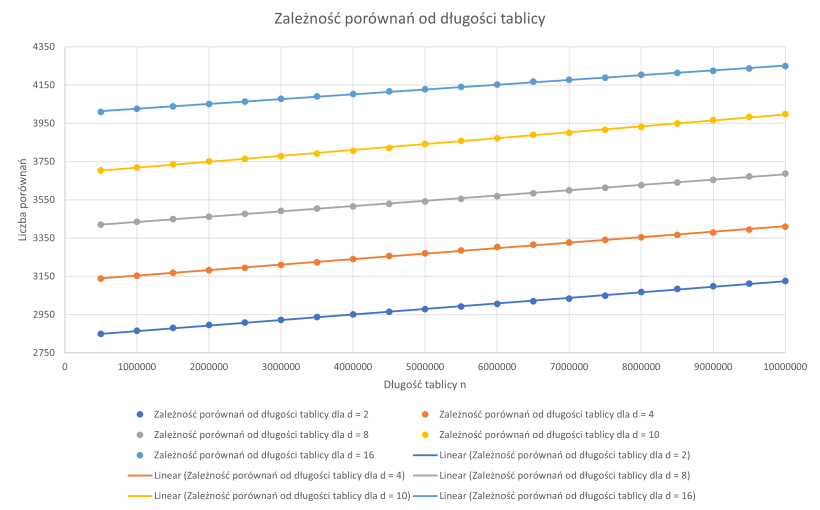
Rysunek 10: Zależność czasu wykonywania od długości tablicy dla zmodyfikowanego algorytmu Radix Sort

Zależność czasu od rozmiaru tablicy pozostaje liniowa, tak samo jak w przypadku standardowej wersji algorytmu, więc złożoność $O(d(n+k))$ jest zachowana. Można zauważyć wzrost wartości największej, co jest bezpośrednim skutkiem modyfikacji, czyli rozszerzenia zbioru wartości, które uporządkowuje program. Zwiększana jest wartość d, ponieważ przy rozdzieleniu zbioru na wartości ujemne i nieujemne, algorytm potrzebuje wyższej liczby iteracji.



Rysunek 11: Zależność liczby przypisań od długości tablicy dla zmodyfikowanego algorytmu Radix Sort

Suma przypisań dla rozmiaru n pozostaje liniowa, ale warto zwrócić uwagę na wzrost wartości największej, co można połączyć z etapem dzielenia zbioru na dwie podtablice.



Rysunek 12: Zależność liczby porównań od długości tablicy dla zmodyfikowanego algorytmu Radix Sort

Suma porównań nie różni się niczym od standardowej wersji algorytmu. Zachowuje zależność liniową, tym samym nie doświadczając znaczącej zmiany w łącznej sumie porównań.

0.4.3 Wnioski

Mimo rozszerzenia domeny algorytmu, nie występują znaczące zmiany w zależności czasu, sumy przypisań ani porównań od rozmiaru tablicy początkowej.

0.5 Linked List Insertion Sort

Insertion Sort na liście jednokierunkowej zachowuje tę samą zasadę co wersja korzystająca z tablic. Buduje posortowany zbiór przez wstawianie elementów w odpowiednie miejsca względem klucza. Różnią się strukturą porządkowania wartości. W przypadku tablicy, są one zamieniane względem indeksów, w liście manipulujemy wskaźnikami, a wstawianie realizujemy przez modyfikację wskaźników next.

```
    class Node {
        public:
        int val;
        struct Node* next;
        Node(int x) {
                val = x;
                next = NULL;
        }
};


Node* sortedInsert(Node* newnode, Node* sorted) {

    if (sorted == NULL ||
        sorted->val >= newnode->val) {
        newnode->next = sorted;
        sorted = newnode;
        assignments += 2;
        comparisons++;
    }
    else {
        Node* curr = sorted;
        assignments++;

        while (curr->next != NULL &&
               curr->next->val < newnode->val) {
            curr = curr->next;
            assignments++;
            comparisons += 2;
        }
        comparisons += 2;

        newnode->next = curr->next;
        curr->next = newnode;
        assignments += 2;
```

```
    }

    return sorted;
}

Node* insertionSort(Node* head) {

    Node* sorted = NULL;
    Node* curr = head;
    assignments += 2;

    while (curr != NULL) {
        comparisons++;

        Node* next = curr->next;
        assignments++;

        sorted = sortedInsert(curr, sorted);
        assignments++;

        curr = next;
        assignments++;
    }
    comparisons++;

    return sorted;
}

void deleteList(Node* head) {
    Node* current = head;
    while (current != NULL) {
        Node* next = current->next;
        delete current;
        current = next;
    }
}
```

0.6 Bucket Sort

Bucket sort, to algorytm sortujący, który opiera się o podzielenie zakresu wartości, standardowo odcinka $[0, 1)$, na kubły, czyli podtablice, do których przydzielane są wartości z danego mniejszego zakresu. Następnie, zawartość każdego kubła jest porządkowana, w tym przypadku Insert Sortem, aby na końcu scalić kubły w posortowaną tablicę. Jego złożoność jest silnie zależna od algorytmu wykorzystanego do porządkowania elementów kubła. W tym przypadku jest to Insertion Sort, więc złożoność całego programu to $O(n^2)$.

```
    void insertion_sort(vector<float>& bucket){
```

```
    int key;
    int j;

    for (int i = 1; i < bucket.size(); i++){
        comparisons++;

        float key = bucket[i];
        int j = i - 1;

        while (j >= 0 && bucket[j] > key){
            comparisons = comparisons + 2;

            bucket[j+1] = bucket[j];
            assignments++;

            j = j - 1;
        }
        bucket[j+1] = key;
        assignments++;
    }
}

void bucket_sort(vector<float>& arr, int n){
    vector<vector<float>> buckets(n);

    for (int i = 0; i < n; i++){
        int bi = n * arr[i];
        buckets[bi].push_back(arr[i]);
        assignments++;
    }

    for (int i = 0; i < n; i++){
        insertion_sort(buckets[i]);
    }

    int index = 0;

    for (int i = 0; i < n; i++){
        for (int j = 0; j < buckets[i].size(); j++){
            arr[index++] = buckets[i][j];
            assignments++;
        }
    }
}
```

Modyfikacja tego algorytmu polega, podobnie jak w przypadku Radix Sorta, na rozszerzenie zakresu danych wchodzących, czyli aby porządkował dowolne wartości nieujemne rzeczywiste.

Zmiana polega na dostosowaniu zakresu wartości, na postawie którego tworzone są kubły. Aby ustalić zakres, znajdujemy różnicę wartości największej i najmniejszej, a następnie normalizujemy do odcinka $[0, 1)$. Po normalizacji podobnie jak w standardowej wersji stosowany jest insert sort, a wyniki scalane.

```
    //insertion_sort pozostaje bez zmian
    void bucket_sort(vector<float>& arr, int n){
    vector<vector<float>> buckets(n);

    float max = getMax(arr, n);
    float min = getMin(arr, n);
    float range = max - min;

    for (int i = 0; i < n; i++){
        //normalizacja do odcinka [0, 1)
        float normal = (arr[i] - min)/range;
        int bi = normal * (n - 2);
        buckets[bi].push_back(arr[i]);
        assignments++;
    }

    for (int i = 0; i < n; i++){
        if (!buckets[i].empty()){
            comparisons++;
            insertion_sort(buckets[i]);
        }
    }

    int index = 0;

    for (int i = 0; i < n; i++){
        for (int j = 0; j < buckets[i].size(); j++){
            arr[index++] = buckets[i][j];
            assignments++;
        }
    }
}
```

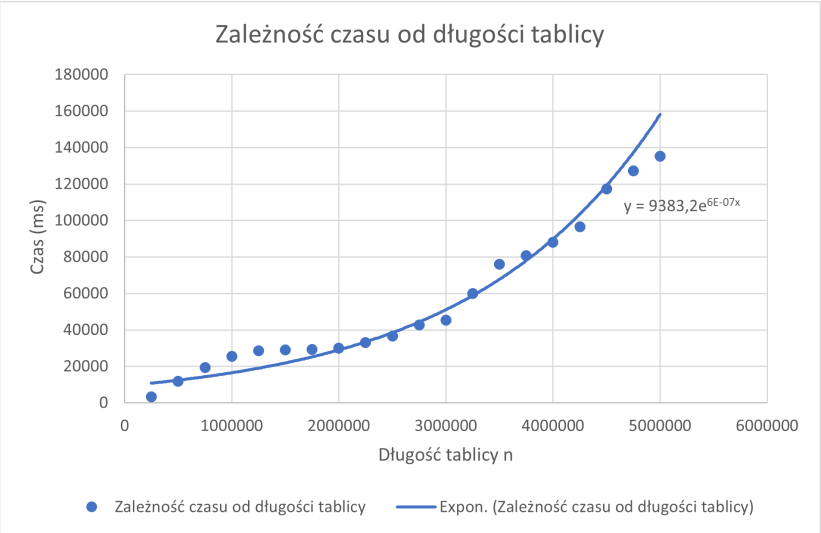
0.6.1 Standardowy Bucket Sort

Średnia wyników

Tabela 5: Średnia wyników dla Standardowego Bucket Sort

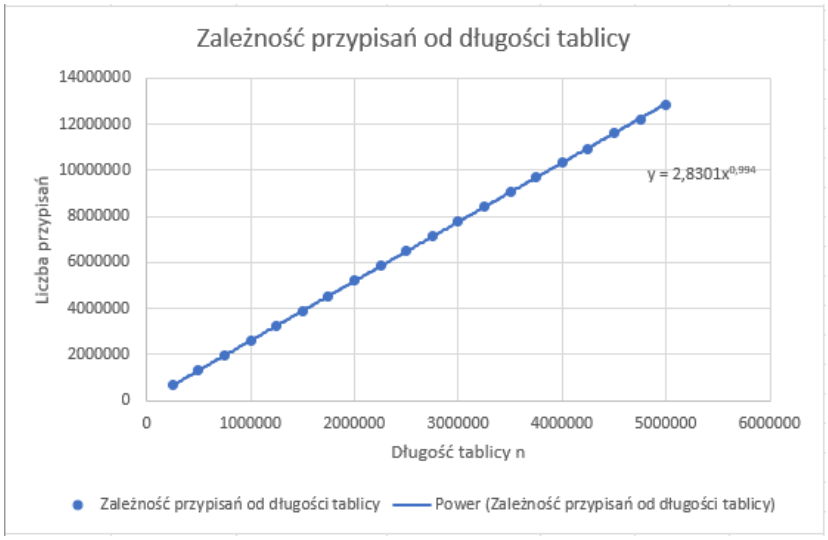
Długość tablicy n	Czas [ms]	Przypisania	Porównania
250000	3343.8	654021.6	216036
500000	11874.4	1306167.4	428475
750000	19289.6	1957939.4	639897
1000000	25524.4	2608334.6	848792.4
1250000	28589.8	3257093.4	1054241.4
1500000	29041	3905373.4	1258608.4
1750000	29208.6	4551520.8	1458938.8
2000000	30065.8	5197604.2	1658744
2250000	33146	5842233	1855716.4
2500000	36576	6485594	2050231
2750000	42786.4	7125446.4	2238508
3000000	45416.8	7769901.6	2433601.6
3250000	59851.4	8406980	2616693
3500000	76072	9047081.4	2803893.4
3750000	80709.6	9688710.8	2993035
4000000	88169.2	10321080	3167036.8
4250000	96496.4	10949550.4	3334106.8
4500000	117290.6	11595651.4	3528877
4750000	127238.4	12232170	3707850
5000000	135264.4	12865013	3880375.2

Analiza wykresów



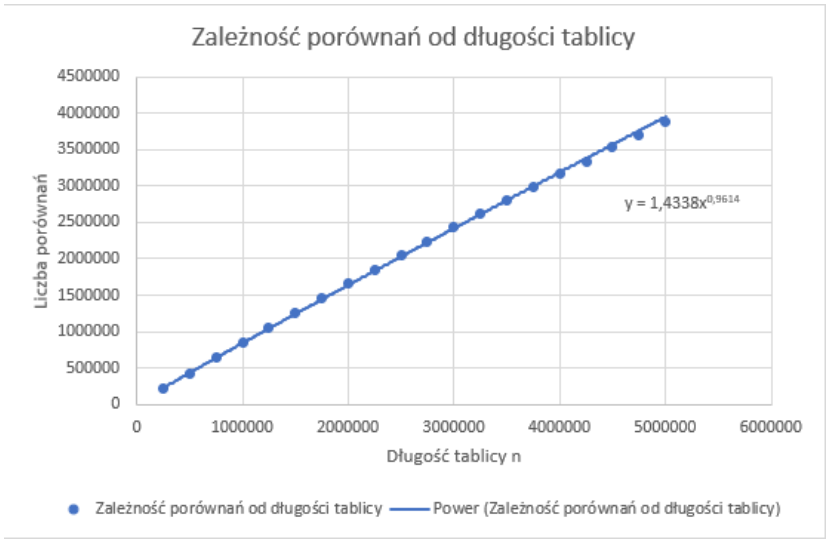
Rysunek 13: Zależność czasu wykonywania od długości tablicy dla algorytmu Bucket Sort

Mimo założenia teoretycznego złożoności $O(n^2)$, wykres przyjmuje formę podobną do funkcji potęgowej, lub algorytmu o złożoności $O(nlogn)$. Jest to możliwy skutek zbyt niskiego zakresu danych wejściowych, a dokładnie za niskich wartości n, czyli rozmiarów tablicy,



Rysunek 14: Zależność liczby przypisań od długości tablicy dla algorytmu Bucket Sort

Zależność sumy przypisań od długości tablicy początkowej jest liniowa, co można przypisać dzieleniu zbiory na mniejsze zakresy, które są względnie prostsze do uporządkowania.



Rysunek 15: Zależność liczby porównań od długości tablicy dla algorytmu Bucket Sort

Podobnie do sumy przypisań, suma porównań również demonstruje zachowanie liniowe.

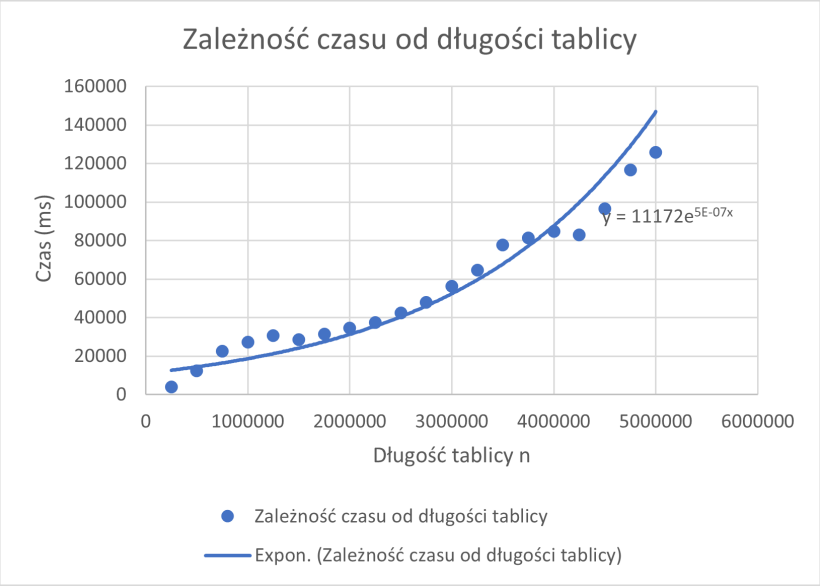
0.6.2 Zmodyfikowany Bucket Sort

Średnia wyników

Tabela 6: Średnia wyników dla Zmodyfikowanego Bucket Sort

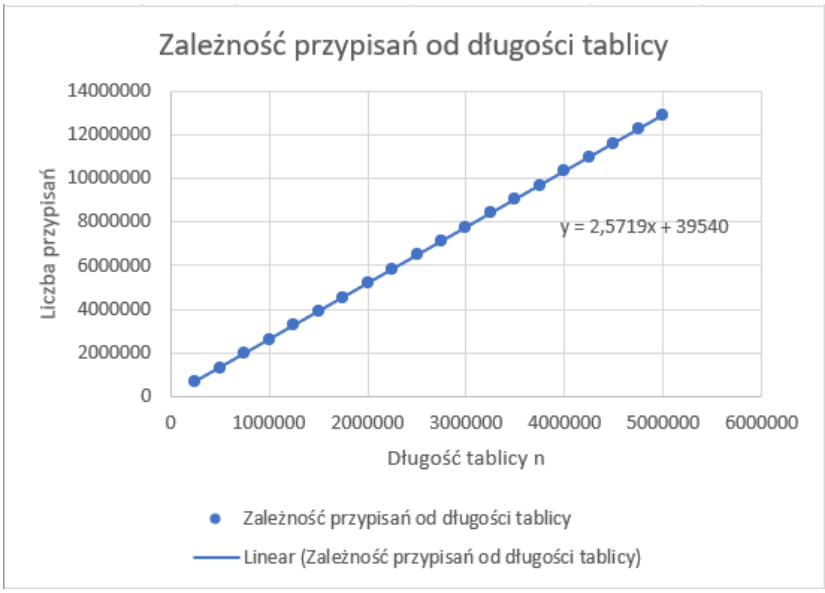
Długość tablicy n	Czas [ms]	Przypisania	Porównania
250000	3986.6	654034.6	373986.4
500000	12433.2	1306402.4	744835.8
750000	22515.4	1957231.6	1112618.6
1000000	27340	2607380.2	1478608.6
1250000	30720.8	3256741	1842726.8
1500000	28605	3904534.8	2203725.2
1750000	31448.4	4550662.8	2561545
2000000	34627.2	5197725	2919342.6
2250000	37580	5842907.2	3274322.8
2500000	42309.2	6486278.8	3625474
2750000	47910.2	7126274.2	3972000.6
3000000	56363.4	7771223.8	4322689.4
3250000	64745.4	8409642.2	4664855.4
3500000	77703.8	9050507.4	5009000.2
3750000	81433.8	9691687.8	5352374.4
4000000	84816	10326841.8	5686785
4250000	82856	10956319	6013285.2
4500000	96519	11602460.4	6359261.6
4750000	116754.6	12240925.8	6694921.4
5000000	125870	12876031.4	7024906.4

Analiza wykresów



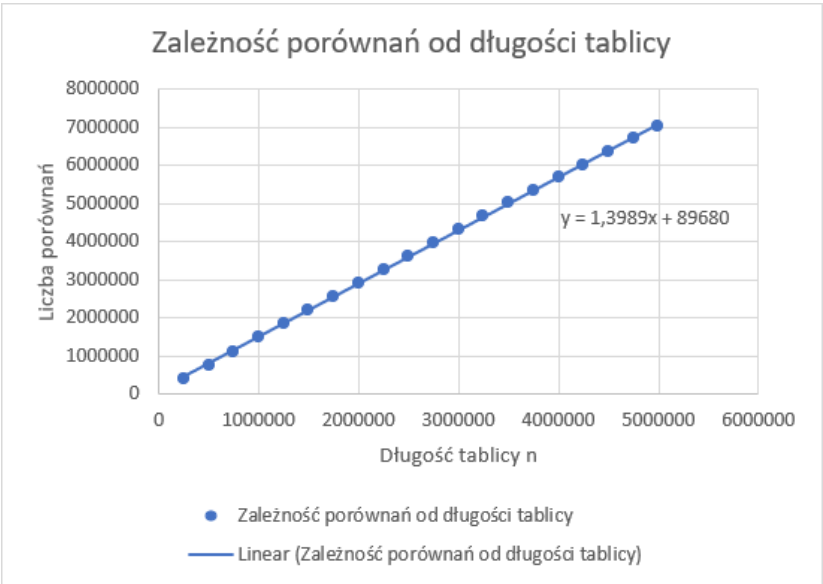
Rysunek 16: Zależność czasu wykonywania od długości tablicy dla zmodyfikowanego Bucket Sort

Tak samo jak w przypadku standardowego Bucket Sorta, złożoność przypomina bardziej funkcję potęgową, lub złożoność $O(nlogn)$, co może być skutkiem niewystarczających rozmiarów tablic. Mimo błędu pomiarowego, wykazuje on szybsze działanie od standardowej wersji, mimo większego zakresu wchodzących wartości.



Rysunek 17: Zależność liczby przypisań od długości tablicy dla zmodyfikowanego Bucket Sort

Suma przypisań pozostaje liniowo zależna od rozmiaru tablicy. Ponadto, nie ma znaczącej różnicy wartości pomiędzy obiema wersjami algorytmu.



Rysunek 18: Zależność liczby porównań od długości tablicy dla zmodyfikowanego Bucket Sort

Tak samo jak dla sumy przypisań, zależność sumy porównań od długości n jest liniowa. Różnią się jednak wartościami sum. Dla zmodyfikowanej wersji jest ona prawie dwukrotna dla najwyższych n.

0.6.3 Wnioski

Obie wersje algorytmu demonstrują podobne złożoności, mimo niezgodności z założeniami teoretycznymi, co ponownie wskazuje na zbyt mały zakres testowanych tablic. Występuje zauważalna różnica w czasie wykonywania, gdzie zmodyfikowany Bucket Sort jest szybszy.

Oba algorytmy nie mają znaczących różnic w zależności sumy przypisań od rozmiaru tablicy.

Zależność sumy porównań od n jest liniowa dla wersji standardowej i zmodyfikowanej, jednak występuje prawie dwukrotny wzrost dla algorytmu zmodyfikowanego.

0.7 Quick Sort a Bucket Sort - Porównanie

Różnice w algorytmach Quick i Bucket Sort wynikają z rozbieżności w ich budowie. Algorytm Quick Sort opiera się o rekursję w celu uporządkowania elementów, a Bucket Sort, po uprzedniej kategoryzacji elementów na mniejsze zakresy, zapożycza metodę sortowania innego algorytmu, w tym przypadku Insertion Sort. Różnice te odzwierciedlają ich złożoności, gdzie dla Quick Sorta jest to $O(nlogn)$, a złożoność Bucket Sorta dominuje złożoność algorytmu, który zapożycza, czyli $O(n^2)$.

Mimo trudności przy pomiarach dla algorytmu Bucket Sort, można zauważyć że Bucket Sort jest znacznie wolniejszy. Jest to widoczne po równaniach best-fit-line zależności czasu od długości tablicy, gdzie Quick Sort przyjmuje założone $O(nlogn)$, a Bucket Sort funkcję potęgową.

W przypadku Quick Sorta (zarówno wersji standardowej, jak i Dual Pivot) suma przypisań rośnie w sposób zbliżony do liniowego, co jest związane z podziałem tablicy i mniejszą liczbą operacji zamiany elementów. W Bucket Sorcie suma przypisań również wykazuje zależność liniową od n, jednak wartości są znacząco niższe niż w Quick Sorcie, co wynika z mniejszej liczby operacji swap oraz prostszej struktury operacji na kubłach. Modyfikacja Bucket Sorta nie wpłynęła istotnie na liczbę przypisań.

Dla Quick Sorta suma porównań rośnie liniowo wraz ze wzrostem rozmiaru tablicy początkowej, co jest zgodne z oczekiwaniami, każdy element jest porównywany średnio $logn$ razy. W Bucket Sorcie suma porównań również rośnie liniowo, jednak w wersji zmodyfikowanej można zauważyć dwukrotny wzrost w stosunku do wersji standardowej. Jest to efekt dodatkowych operacji związanych z szerszym zakresem wartości oraz etapu normalizacji do odcinka $[0, 1)$ i scalania do tablicy posortowanej.