Algorytmy i Struktury Danych
Sprawozdanie 1

Emilia Łagoda
Numer albumu: 287361

Spis treści

0.1 Wstęp

Celem tego ćwiczenia jest porównanie wydajności algorytmów sortowania tablic w oparciu o czas wykonywania, liczbę przypisań wartości do tablic, oraz liczbę porównań. Czas wykonywania to mierzony w milisekundach (ms) czas od rozpoczęcia do zakończenia wykonywania funkcji sortującej. Liczba przypisań wartości, to suma wykonanych przez funkcję przypisań wartości do tablicy podczas jednej instancji funkcji. Podobnie do przypisań, liczba porównań, to suma wykonanych porównań pomiędzy wartościami w trakcie jednego przebiegu algorytmu sortującego.

Zmienne te, zależne są od długości tablicy n. Dla każdego algorytmu, wykonywane jest pięć prób testów na dwudziestu tablicach o liniowo rosnącej długości n, zapełnionymi losowymi liczbami całkowitymi z zakresu od 1 do 1000000.

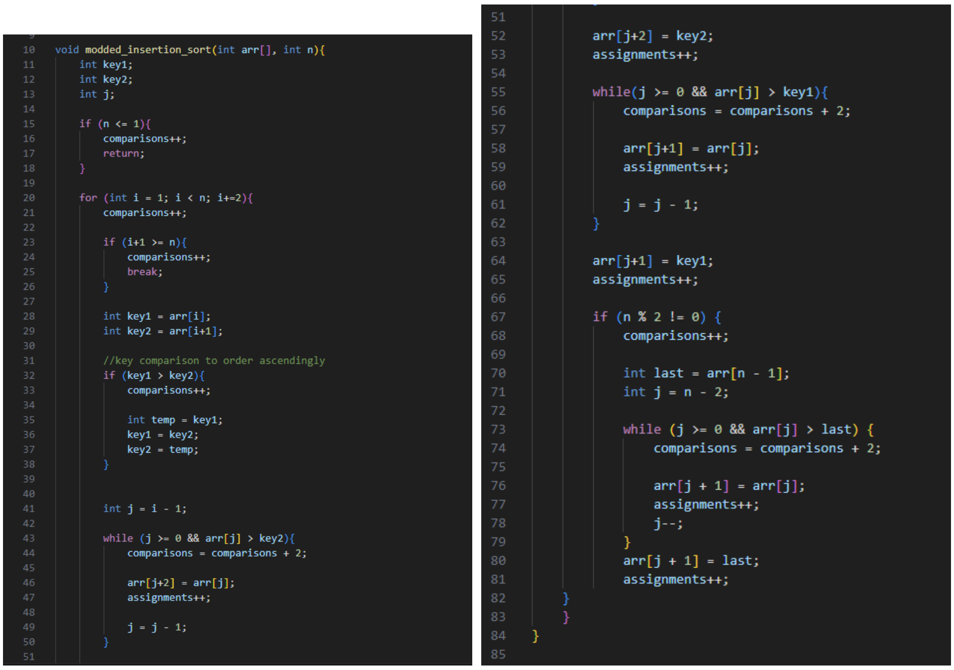
0.2 Insertion Sort

Insertion sort, czy też sortowanie przez wstawianie, to algorytm sortujący dzielący tablicę na dwie części, posortowaną i nieposortowaną, względem klucza. Klucz ten, z każdą iteracją algorytmu porusza się względem indeksu tablicy, sortując kolejne elementy. Samo sortowanie polega na porównywaniu klucza z elementami posortowanej części tablicy i wstawieniu go na odpowiednie miejsce.

Zmodyfikowany insertion sort od standardowego różni się tym, że w zmodyfikowanym algorytmie, wyznaczane są dwa klucze, które po uprzednim porównaniu i ustawieniu w odpowiedniej kolejności, wstawiane są na swoje odpowiednie miejsca.

```
void insertion_sort(int arr[], long int n){
    int key;
    int j;

    for (int i = 1; i < n; i++){
        comparisons++;

        int key = arr[i];
        int j = i - 1;

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

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

            j = j - 1;
        }
        arr[j+1] = key;
        assignments++;
    }
}
```

Rysunek 1: Kod algorytmu Insertion Sort



```c
void modded_insertion_sort(int arr[], int n){
    int key1;
    int key2;
    int j;

    if (n <= 1){
        comparisons++;
        return;
    }

    for (int i = 1; i < n; i+=2){
        comparisons++;

        if (i+1 >= n){
            comparisons++;
            break;
        }

        int key1 = arr[i];
        int key2 = arr[i+1];

        //key comparison to order ascendingly
        if (key1 > key2){
            comparisons++;

            int temp = key1;
            key1 = key2;
            key2 = temp;
        }


        int j = i - 1;

        while (j >= 0 && arr[j] > key2){
            comparisons = comparisons + 2;

            arr[j+2] = arr[j];
            assignments++;

            j = j - 1;
        }

        arr[j+2] = key2;
        assignments++;

        while(j >= 0 && arr[j] > key1){
            comparisons = comparisons + 2;

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

            j = j - 1;
        }

        arr[j+1] = key1;
        assignments++;

        if (n % 2 != 0) {
            comparisons++;

            int last = arr[n - 1];
            int j = n - 2;

            while (j >= 0 && arr[j] > last) {
                comparisons = comparisons + 2;

                arr[j + 1] = arr[j];
                assignments++;
                j--;
            }
            arr[j + 1] = last;
            assignments++;
        }
    }
}
```

Rysunek 2: Kod zmodyfikowanego algorytmu Insertion Sort

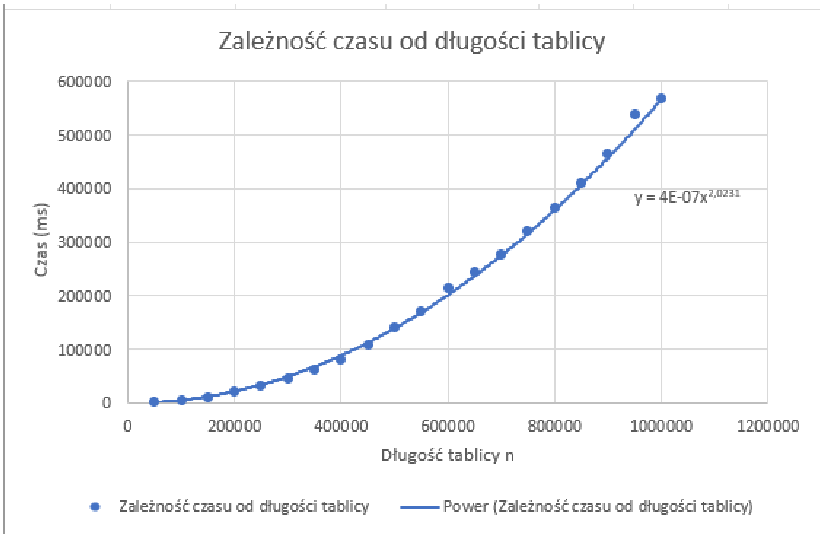
0.2.1 Standardowy Insertion Sort

Średnia wyników

Tabela 1: Średnia wyników dla Standardowego Insertion Sort

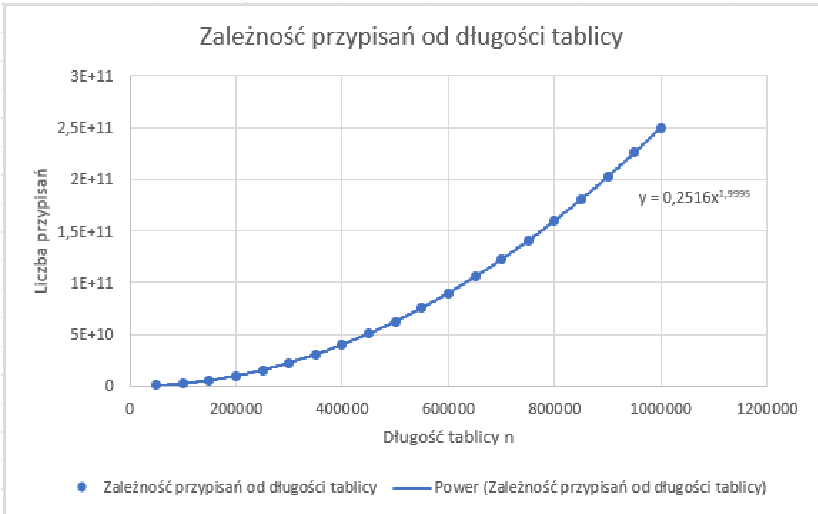
Długość tablicy n	Czas [ms]	Przypisania	Porównania
50000	1460.6	625693848.8	1125881143.0
100000	5630.6	2502690992.0	4504572689.0
150000	11915.6	5627484790.0	10129578487.0
200000	20464.6	10005296302.0	18009641225.0
250000	31738.4	15637177431.0	28139762278.0
300000	45818.0	22489216307.0	40485684037.0
350000	62686.8	30616884016.0	55108119160.0
400000	82132.6	40006877509.0	72013494126.0
450000	109452.0	50572291622.0	91033489015.0
500000	142461.2	62475289474.0	112450000000.0
550000	172510.4	75607962989.0	136091000000.0
600000	214247.0	90018829624.0	162029000000.0
650000	245980.6	105617000000.0	190095000000.0
700000	277063.8	122515000000.0	220529000000.0
750000	319725.2	140648000000.0	253177000000.0
800000	363222.0	160029000000.0	288060000000.0
850000	411431.4	180517000000.0	324949000000.0
900000	464463.2	202415000000.0	364358000000.0
950000	539519.0	225542000000.0	405989000000.0
1000000	568657.8	249961000000.0	449884000000.0

Analiza wykresów



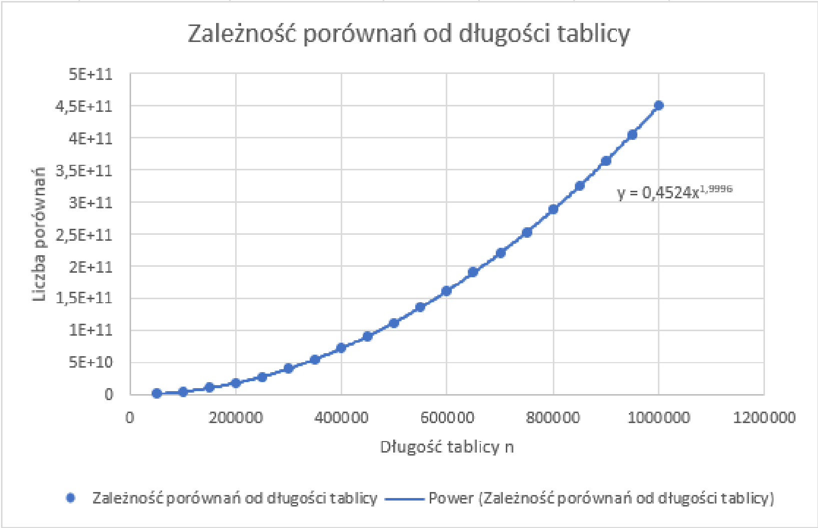
Rysunek 3: Zależność czasu wykonywania od długości tablicy dla algorytmu Insertion Sort

Wykazuje charakterystyczny wzrost kwadratowy $O(n^2)$, gdzie krzywa przypomina parabolę. Równanie aproksymujące to $T(n) \approx k \cdot n^2$, co potwierdza teoretyczną złożoność czasową algorytmu.



Rysunek 4: Zależność liczby przypisań od długości tablicy dla algorytmu Insertion Sort

Krzywa rośnie zgodnie z zależnością $O(n^2)$, co jest typowe dla algorytmów o złożoności kwadratowej. Wzrost jest symetryczny względem osi czasu.



Rysunek 5: Zależność liczby porównań od długości tablicy dla algorytmu Insertion Merge Sort

Demonstruje najszybszy wzrost spośród wszystkich mierzonych parametrów, również o złożoności $O(n^2)$, ale ze stałą proporcjonalności około 1.8 razy większą niż dla przypisań

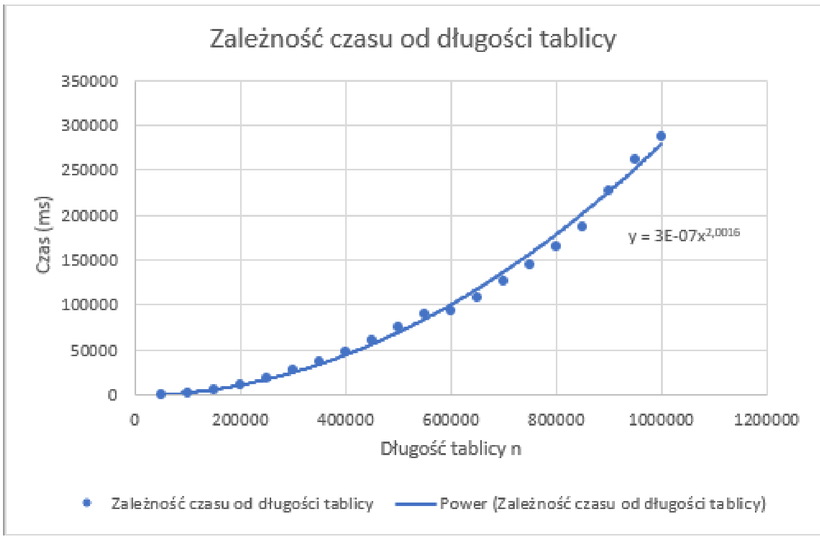
0.2.2 Zmodyfikowany Insertion Sort

Średnia wyników

Tabela 2: Średnia wyników dla Zmodyfikowanego Insertion Sort

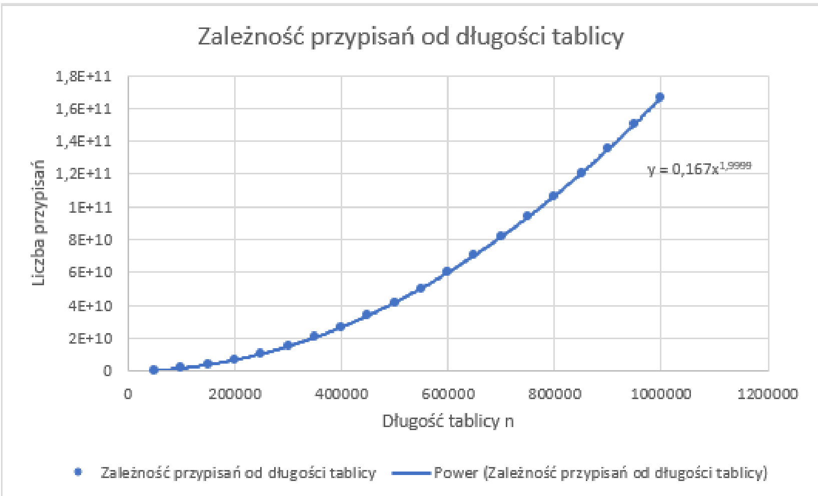
Długość tablicy n	Czas [ms]	Przypisania	Porównania
50000	664.4	417475188.4	834887891.4
100000	2608.2	1665545035.0	3330965135.0
150000	5848.8	3745523723.0	7490859923.0
200000	11496.8	6665918032.0	13331585944.0
250000	18454.6	10420222419.0	20840132289.0
300000	27284.2	15006567635.0	30012760208.0
350000	37149.0	20417025369.0	40833613136.0
400000	48297.2	26670695600.0	53340891200.0
450000	60939.6	33745742875.0	67490923298.0
500000	75550.0	41662543135.0	83324461561.0
550000	89716.4	50384054634.0	100767000000.0
600000	92711.2	59979882045.0	119959000000.0
650000	108813.0	70416165108.0	140832000000.0
700000	126774.0	81665472799.0	163330000000.0
750000	144939.0	93750686555.0	187500000000.0
800000	165385.2	106694000000.0	213386000000.0
850000	187129.6	120460000000.0	240919000000.0
900000	226793.8	135006000000.0	270012000000.0
950000	261210.4	150505000000.0	301009000000.0
1000000	287836.0	166618000000.0	333234000000.0

Analiza wykresów



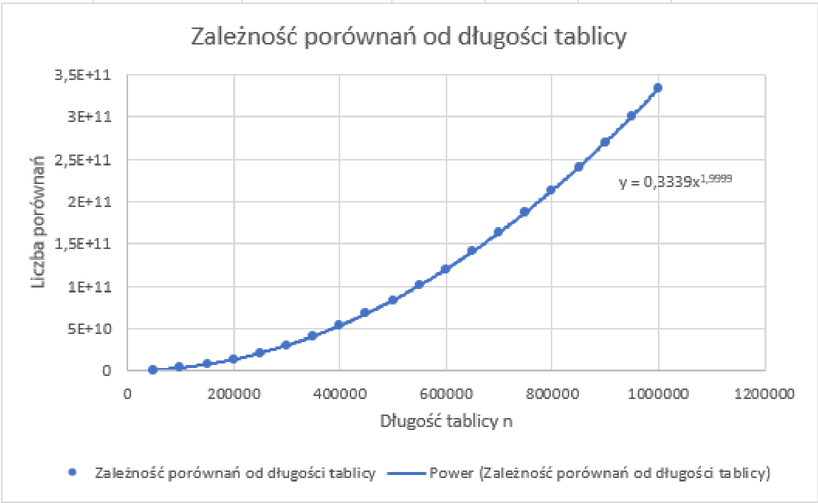
Rysunek 6: Zależność czasu wykonywania od długości tablicy dla zmodyfikowanego Insertion Sort

Zachowuje złożoność kwadratową $O(n^2)$, jednak ze zmniejszoną stałą proporcjonalności. Nachylenie krzywej jest mniejsze, co wskazuje na praktyczną poprawę wydajności przy zachowaniu tej samej złożoności teoretycznej.



Rysunek 7: Zależność liczby przypisań od długości tablicy dla zmodyfikowanego Insertion Sort

Zmodyfikowany algorytm utrzymuje złożoność $O(n^2)$, ale z redukcją liczby przypisań.



Rysunek 8: Zależność liczby porównań od długości tablicy dla zmodyfikowanego Insertion Merge Sort

Podobnie do zmian w liczbie przypisań, suma porównań również zachowuje złożoność $O(n^2)$, a równocześnie doświadcza redukcji w liczbie porównań.

0.2.3 Wnioski

Zmodyfikowany Insertion Sort wykazuje znaczną poprawę wydajności czasowej, redukując czas wykonania na całym zakresie danych. Oba algorytmy zachowują złożoność kwadratową $O(n^2)$, jednak modyfikacja skutecznie redukuje stałą proporcjonalności.

Wersja zmodyfikowana osiąga redukcję liczby przypisań, co bezpośrednio przekłada się na poprawę wydajności.

Podobnie jak w przypadku przypisań, zmodyfikowany algorytm redukuje liczbę porównań.

0.3 Merge Sort

Megre sort, znany również jako algorytm ”dziel i zwyciężaj”, polega na dzieleniu większego porblemu na mniejsze segmenty, rozwiązywaniu ich, a następnie scalaniu gotowych podsekcji. W przypadku sortowania tablic liczb całkowitych, merge sort polega na rekursywnym dzieleniu tablicy na k jednoelementowych podtablic (tutaj została uzyta wersja, która dzieli tablicę na dwie części), które są uważane za posortowane, a następnie scalaniu ich w ostateczną posortowaną tablicę.

Trójkątny merge sort dzieli tablicę na trzy części zamiast standardowych dwóch, po czym bez zmian rekursywnie tworzy podtablice aż nie otrzyma k tablic jednoelementowych.

```cpp
//first subarray is L[left..mid]
//second subarray is R[mid+1..right]
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Use dynamic allocation instead of stack arrays
    int* L = new int[n1];
    int* R = new int[n2];

    for (int i = 0; i < n1; i++){
        comparisons++;
        L[i] = arr[left + i];
        assignments++;
    }
    for (int j = 0; j < n2; j++){
        comparisons++;
        R[j] = arr[mid + 1 + j];
        assignments++;
    }

    int i = 0;
    int j = 0;
    int k = left;

    //placing the smallest element
    while (i < n1 && j < n2) {
        comparisons = comparisons + 2;

        if (L[i] <= R[j]) {
            comparisons++;
            arr[k] = L[i];
            assignments++;
            i++;
        } else {
            arr[k] = R[j];
            assignments++;
            j++;
        }
        k++;
    }
```

```cpp
    //copy the remaining elements of L[], if any
    while (i < n1) {
        comparisons++;
        arr[k] = L[i];
        assignments++;
        i++;
        k++;
    }

    //copy the remaining elements of R[], if any
    while (j < n2) {
        comparisons++;
        arr[k] = R[j];
        assignments++;
        j++;
        k++;
    }

    // Clean up dynamically allocated memory
    delete[] L;
    delete[] R;
}

//subarray to be sorted is in the index range [left..right]
void merge_sort(int arr[], int left, int right) {
    if (left < right) {
        comparisons++;

        //calculate the midpoint
        int mid = left + (right - left) / 2;

        //sort first and second halves
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);

        //merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

Rysunek 9: Kod algorytmu Merge Sort

```cpp
void three_way_merge(int arr[], int left, int mid1, int mid2, int right){

    int n1 = mid1 - left + 1;
    int n2 = mid2 - mid1;
    int n3 = right - mid2;

    //temporary subarray (vector because of the dynamic size) for each third
    vector <int> L(n1), M(n2), R(n3);

    for (int i=0; i<n1; i++){
        comparisons++;

        L[i] = arr[left + i];
        assignments++;
    }

    for (int i=0; i<n2; i++){
        comparisons++;

        M[i] = arr[mid1 + 1 + i];
        assignments++;
    }

    for (int i=0; i<n3; i++){
        comparisons++;

        R[i] = arr[mid2 + 1 + i];
        assignments++;
    }

    int i = 0, j = 0, k = 0, index = left;
```

```cpp
    //finding and placing the smallest from the three subarrays
    while (i < n1 || j < n2 || k < n3){
        comparisons = comparisons + 3;

        int smallest = INT_MAX, smallest_index = -1;

        //assigning indexes 0, 1, 2 to subarrays L, M, R respectively
        if (i < n1 && L[i] < smallest){
            comparisons = comparisons + 2;

            smallest = L[i];
            smallest_index = 0;
        }
        if (j < n2 && M[j] < smallest){
            comparisons = comparisons + 2;

            smallest = M[j];
            smallest_index = 1;
        }
        if (k < n3 && R[k] < smallest){
            comparisons = comparisons + 2;

            smallest = R[k];
            smallest_index = 2;
        }
```

```cpp

        //placing it according to index
        if (smallest_index == 0){
            comparisons++;

            arr[index++] = L[i++];
            assignments++;
        }
        else if (smallest_index == 1){
            comparisons++;

            arr[index++] = M[j++];
            assignments++;
        }
        else{
            arr[index++] = R[k++];
            assignments++;
        }
    }


}

void three_way_merge_sort(int arr[], int left, int right){
    //single element case
    if (right - left < 1) {
        comparisons++;
        return;
    }

    //midpoint on the 1/3 of array length
    int mid1 = left + (right - left)/3;
    //same but with 2*1/3
    int mid2 = left + 2*(right-left)/3;

    three_way_merge_sort(arr, left, mid1);
    three_way_merge_sort(arr, mid1 + 1, mid2);
    three_way_merge_sort(arr, mid2+1, right);

    three_way_merge(arr, left, mid1, mid2, right);
}
```

Rysunek 10: Kod algorytmu Trójkątny Merge Sort

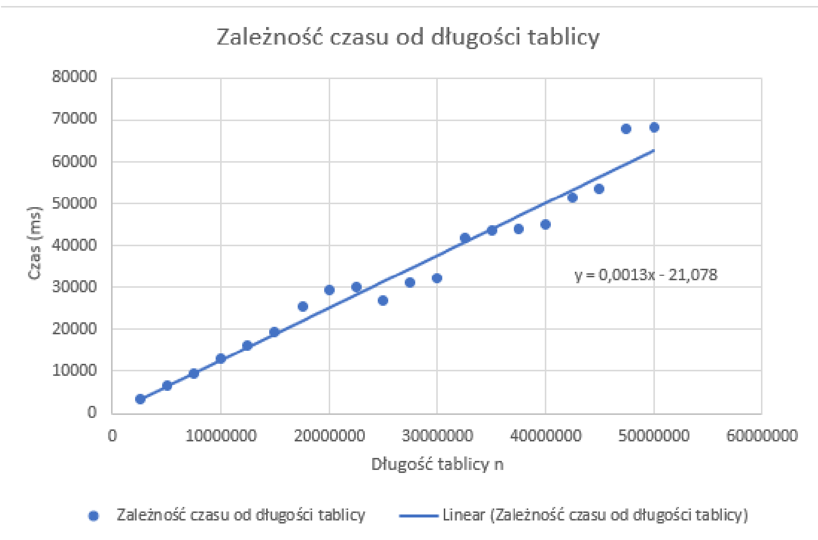
0.3.1 Standardowy Megre Sort

Średnia wyników

Tabela 3: Średnia wyników dla Standardowego Merge Sort

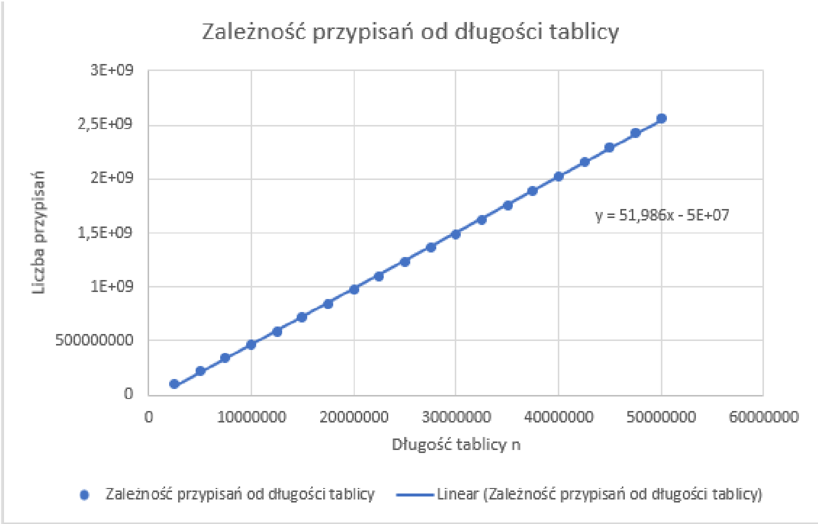
Długość tablicy n	Czas [ms]	Przypisania	Porównania
2500000	3287.4	106611392.0	184426528.2
5000000	6534.8	223222784.0	386353583.0
7500000	9384.8	343222784.0	594213019.4
10000000	12902.8	466445568.0	807703324.6
12500000	16185.6	591445568.0	1023507236.0
15000000	19508.0	716445568.0	1240920773.0
17500000	25411.2	842891136.0	1460144182.0
20000000	29478.6	972891136.0	1685405923.0
22500000	30061.8	1102891136.0	1910512658.0
25000000	26782.2	1232891136.0	2134518065.0
27500000	31033.8	1362891136.0	2361190759.0
30000000	32323.2	1492891136.0	2586845789.0
32500000	41853.8	1622891136.0	2811885973.0
35000000	43517.6	1755782272.0	3042790579.0
37500000	44097.4	1890782272.0	3276797165.0
40000000	45050.2	2025782272.0	3510822057.0
42500000	51407.6	2160782272.0	3744016342.0
45000000	53537.8	2295782272.0	3978528095.0
47500000	67685.4	2430782272.0	4212023660.0
50000000	67988.4	2565782272.0	4444033062.0

Analiza wykresów



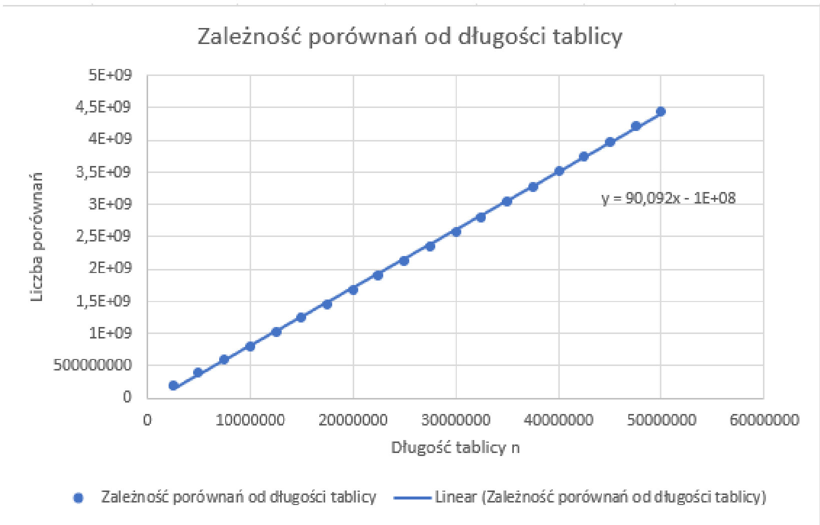
Rysunek 11: Zależność czasu wykonywania od długości tablicy dla algorytmu Merge Sort

Mimo liniewego trendu, krzywa wykazuje charakterystyczne ”schodkowe” zachowanie, gdzie okresy liniowego wzrostu zamieniają się z nagłymi skokami. Wynika to z rekursywności algorytmu.



Rysunek 12: Zależność liczby przypisań od długości tablicy dla algorytmu Merge Sort

Wykres rośnie w sposób liniowy $O(n)$. Każdy element jest kopiowany stałą liczbę razy podczas procesu scalania. Stały stosunek przypisań do długości tablicy wynika z faktu, że w każdym poziomie rekurencji wszystkie elementy są przepisywane.



Rysunek 13: Zależność liczby porównań od długości tablicy dla algorytmu Merge Sort

Wykres liczby porównań również rośnie w sposób liniowy $O(n)$. Liczba porównań jest zawsze większa niż liczba przypisań, co odzwierciedla dodatkową pracę wykonywaną podczas scalania posortowanych podtablic.

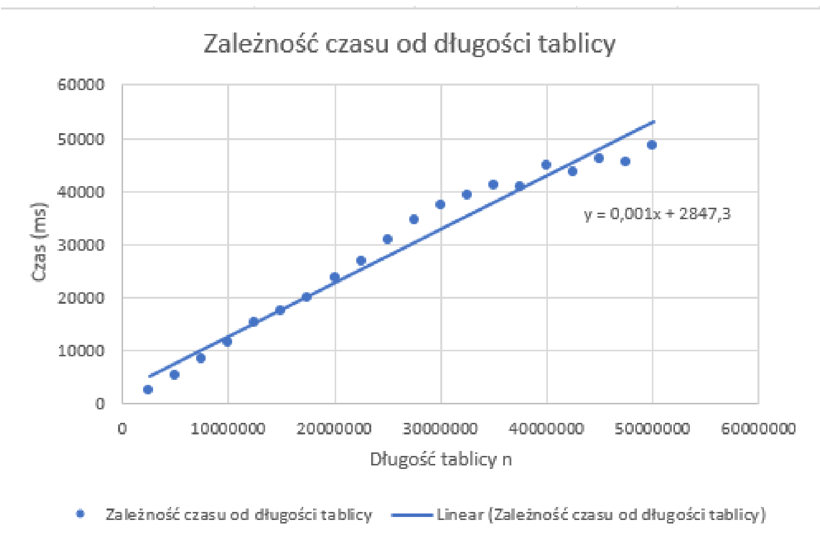
0.3.2 Trójkątny Merge Sort

Średnia wyników

Tabela 4: Średnia wyników dla Trójkątnego Merge Sort

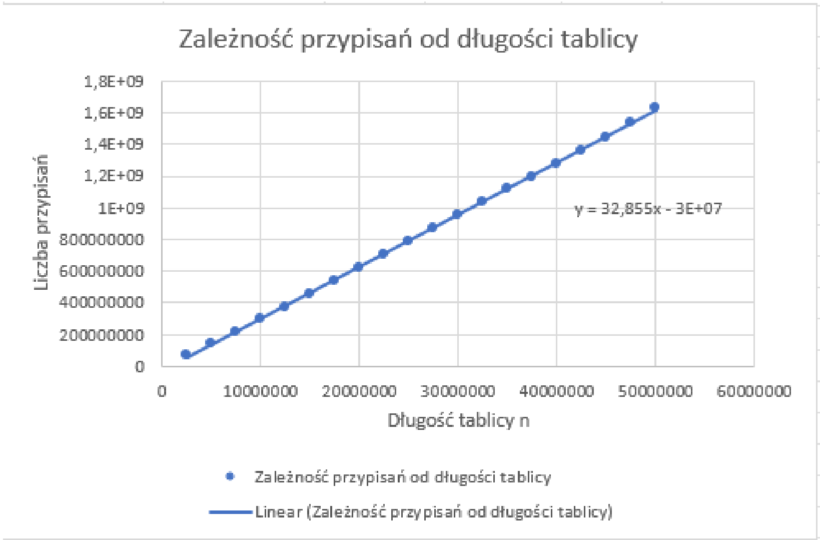
Długość tablicy n	Czas [ms]	Przypisania	Porównania
2500000	2688.2	68622708.0	283285950.6
5000000	5232.2	140868124.0	582380459.0
7500000	8624.8	220868124.0	910252886.6
10000000	11572.8	300000000.0	1236715436.0
12500000	15236.0	375000000.0	1546613817.0
15000000	17473.8	452604372.0	1867186884.0
17500000	19970.0	537604372.0	2214873928.0
20000000	23657.8	622604372.0	2562455967.0
22500000	26878.8	707604372.0	2910767839.0
25000000	30742.8	792604372.0	3260198111.0
27500000	34523.8	877604372.0	3611109316.0
30000000	37348.8	960000000.0	3950145853.0
32500000	39261.6	1040000000.0	4279191183.0
35000000	41191.8	1120000000.0	4608768960.0
37500000	40922.8	1200000000.0	4939828209.0
40000000	44788.6	1280000000.0	5270957241.0
42500000	43788.6	1360000000.0	5605174303.0
45000000	46220.2	1447813116.0	5961596713.0
47500000	45589.8	1537813116.0	6328690131.0
50000000	48676.6	1627813116.0	6694156277.0

Analiza wykresów



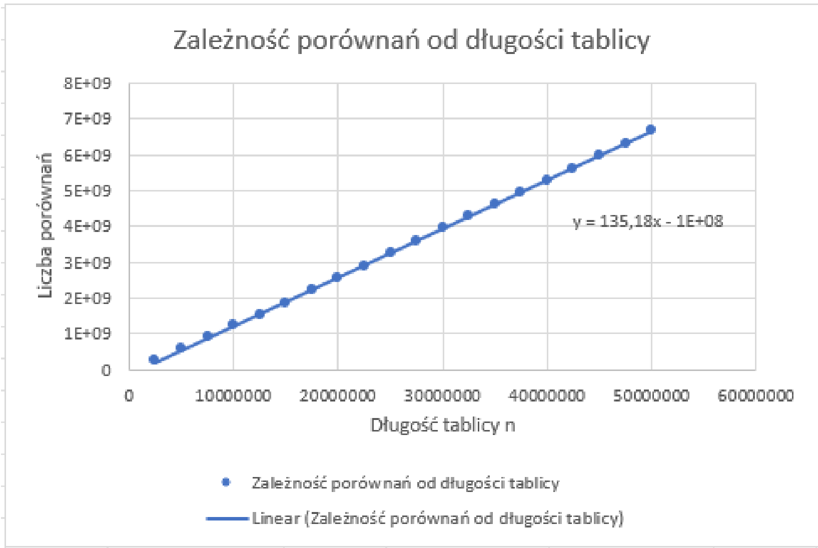
Rysunek 14: Zależność czasu wykonywania od długości tablicy dla algorytmu Trójkątny Merge Sort

Zachowuje złożoność $O(n)$ z zauważalnie mniejszą stałą. Krzywa jest gładsza niż w przypadku standardowego Merge Sorta, co sugeruje bardziej równomierny rozkład operacji. Podział na trzy części zmniejsza głębokość algorytmu, co skutkuje mniejszą liczbą rekursywnych kroków.



Rysunek 15: Zależność liczby przypisań od długości tablicy dla algorytmu Trójkątny Merge Sort

Wykazuje znaczną redukcję liczby operacji. Pomimo zachowania złożoności $O(n)$, mniejsza liczba poziomów rekursji bezpośrednio zmniejsza całkowitą liczbę przypisań.



Rysunek 16: Zależność liczby porównań od długości tablicy dla algorytmu Trójkątny Merge Sort

Zachowuje złożoność $O(n)$ z większą liczbą operacji niż w standardowym Merge Sorcie. Wynika to z faktu, że scalanie trzech podtablic zamiast dwóch wymaga więcej porównań, co pozwala na redukcję głębokości rekurencji i mniejszą liczbę przypisań.

0.3.3 Wnioski

Trójkątny Merge Sort pokazuje lepszą wydajność czasową, osiągając poprawę dla dużych zbiorów danych. Mimo zachowania złożoności $O(n)$, występuje zauważalna redukcja współczynnika przy x.

Można też zaobserwować redukcję liczby przypisań dla trójkątnej wersji Merge Sorta. Redukcja ta jest bezpośrednim skutkiem zmiejszonej liczby poziomów rekurencji algorytmu.

Pomimo ogólnej poprawy wydajności, trójkątny Merge Sort wykonuje więcej porównań. Wynika to ze złożoności procesu scalania trzech zamiast dwóch podtablic.

0.4 Heap Sort

Heap sort, czyli sortowanie przez kopcowanie, to dwufazowy algorytm sortujący. Pierwsza faza polega na tworzeniu kopca binarnego, czyli tablicowej struktury danych imitującej drzewo binarne. W tym celu, elementy tablicy są przestawiane aby umożliwić stworzenie kopca, gdzie najwyższa wartość jest korzeniem drzewa. Druga faza polega na umieszczaniu najwyższej wartości na początku tablicy aż kopiec nie będzie pusty.

Ternarny heap sort różni się od binarnego konstrukcją kopca. Kopiec trójkowy charakteryzuje się tym, że każdy węzeł ma maksymalnie tróje dzieci. Następnie, jak przy kopcach binarnych, korzeń jest umieszczany na początku tablicy.

```
int left(int i){
    return 2*i + 1;
}

int right(int i){
    return 2*i + 2;
}

void heapify(int arr[], int i){
    int l = left(i);
    int r = right(i);

    int largest;

    if ((l < heap_size) && (arr[l] > arr[i])){
        comparisons = comparisons + 2;

        largest = l;
    }
    else{
        largest = i;
    }

    if ((r < heap_size) && (arr[r] > arr[largest])){
        comparisons = comparisons + 2;

        largest = r;
    }

    if (i != largest){
        comparisons++;

        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        assignments = assignments + 2;

        heapify(arr, largest);
    }
}
```

```
void build_heap(int arr[], int n){
    heap_size = n;
    for (int i = floor(n/2); i >= 0; i--){
        comparisons++;

        heapify(arr, i);
    }
}

void heap_sort(int arr[], int n){
    build_heap(arr, n);

    for (int i = n-1; i >= 1; i--){
        comparisons++;

        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        assignments = assignments + 2;

        heap_size = heap_size - 1;
        heapify(arr, 0);
    }
}
```

Rysunek 17: Kod algorytmu Heap Sort

```c
int left(int i){
    return 3*i + 1;
}

int middle(int i){
    return 3*i + 2;
}

int right(int i){
    return 3*i + 3;
}

void ternary_heapify(int arr[], int i){
    int l = left(i);
    int m = middle(i);
    int r = right(i);

    int largest;

    if ((l < heap_size) && (arr[l] > arr[i])){
        comparisons = comparisons + 2;

        largest = l;
    }
    else{
        largest = i;
    }

    if ((m < heap_size) && (arr[m] > arr[largest])){
        comparisons = comparisons + 2;

        largest = m;
    }

    if ((r < heap_size) && (arr[r] > arr[largest])){
        comparisons = comparisons + 2;

        largest = r;
    }
```

```c

    if (i != largest){
        comparisons++;

        int temp = arr[i];

        arr[i] = arr[largest];
        arr[largest] = temp;
        assignments = assignments + 2;

        ternary_heapify(arr, largest);
    }
}

void ternary_build_heap(int arr[], int n){
    heap_size = n;
    for (int i = floor((n/3)-1); i >= 0; i--){
        comparisons++;
        ternary_heapify(arr, i);
    }
}

void ternary_heap_sort(int arr[], int n){
    ternary_build_heap(arr, n);

    for (int i = n-1; i >= 1; i--){
        comparisons++;

        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        assignments = assignments + 2;

        heap_size = heap_size - 1;
        ternary_heapify(arr, 0);
    }
}
```

Rysunek 18: Kod algorytmu Ternarny Heap Sort

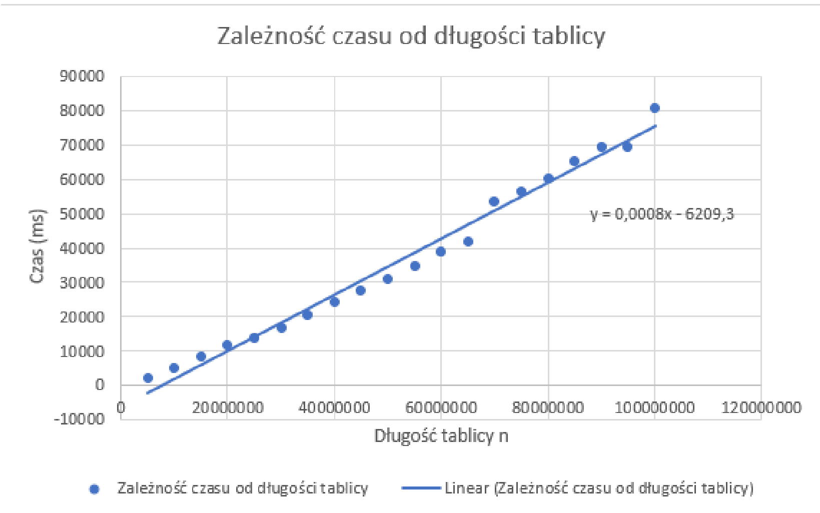
0.4.1 Standardowy Heap Sort

Średnia wyników

Tabela 5: Średnia wyników dla Standardowego Heap Sort

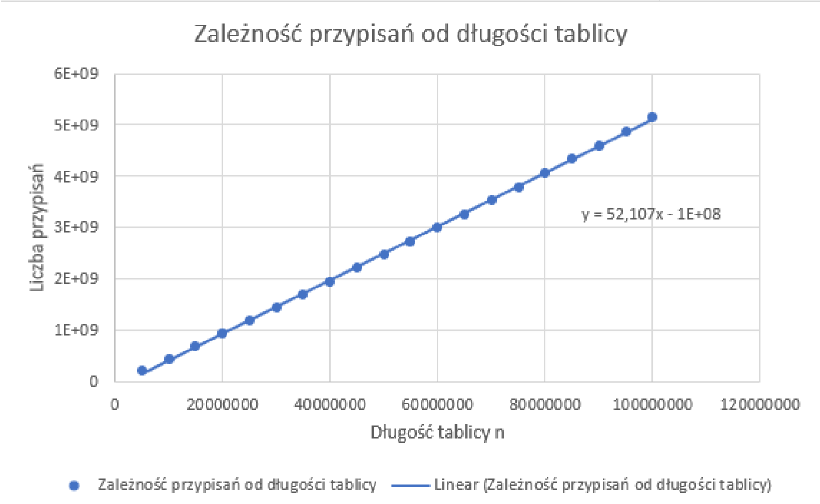
Długość tablicy n	Czas [ms]	Przypisania	Porównania
5000000	2211.0	213834560.4	410275052.0
10000000	5101.0	447666482.0	860544553.6
15000000	8276.8	688962480.4	1326095778.0
20000000	11659.0	935337334.8	1801097325.0
25000000	14061.8	1185791312.0	2284606538.0
30000000	16967.2	1437913583.0	2772175245.0
35000000	20580.0	1692209664.0	3265659549.0
40000000	24181.6	1950676736.0	3762195307.0
45000000	27788.0	2210599457.0	4263052224.0
50000000	30941.6	2471573784.0	4769199443.0
55000000	34779.8	2733372117.0	5274672128.0
60000000	39186.4	2995842662.0	5784393823.0
65000000	42076.6	3258815663.0	6297646624.0
70000000	53595.6	3524420468.0	6811333179.0
75000000	56386.0	3792455391.0	7326819747.0
80000000	60371.8	4061361248.0	7844407728.0
85000000	65521.0	4330977145.0	8365287596.0
90000000	69525.0	4601333374.0	8886124057.0
95000000	69698.8	4872313300.0	9410594341.0
100000000	80820.4	5143272838.0	9938592052.0

Analiza wykresów



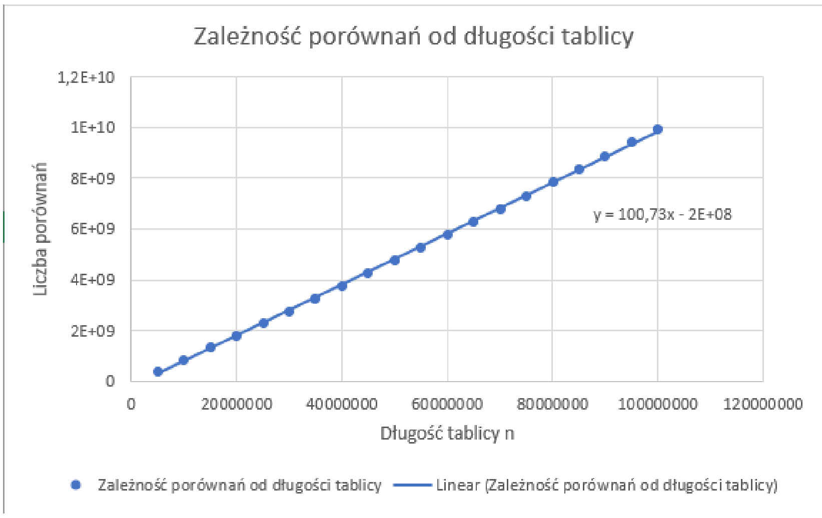
Rysunek 19: Zależność czasu wykonywania od długości tablicy dla algorytmu Heap Sort

Podobnie do algorytmu merge sort, linia wykazuje złożoność liniową $O(n)$. Tu również można zauważyć minimalne zachowanie schodkowe.



Rysunek 20: Zależność liczby przypisań od długości tablicy dla algorytmu Heap Sort

Złożoność $O(n)$ z mniejszą stałą niż dla porównań, co wynika z efektywnej organizacji kopca binarnego



Rysunek 21: Zależność liczby porównań od długości tablicy dla algorytmu Heap Sort

Wykres rośnie zgodnie z $O(n)$, z około dwukrotnie większą liczbą operacji niż przypisań.

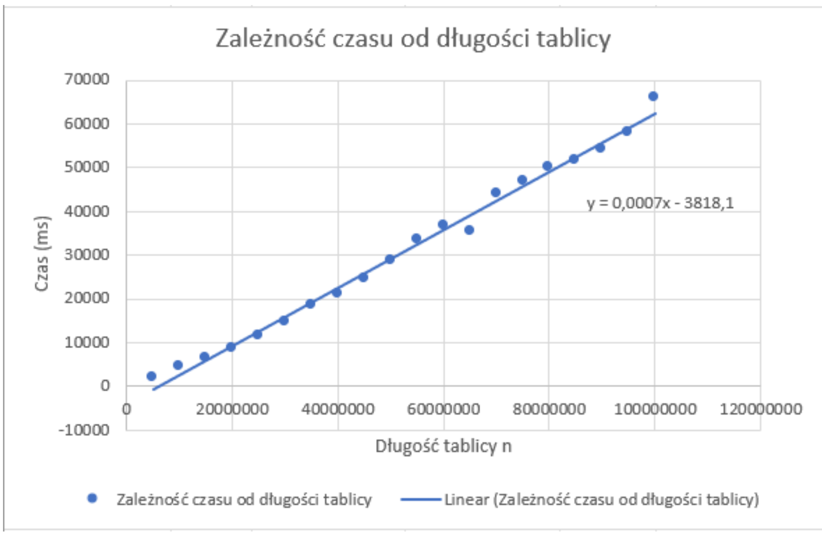
0.4.2 Ternarny Heap Sort

Średnia wyników

Tabela 6: Średnia wyników dla Ternarnego Heap Sort

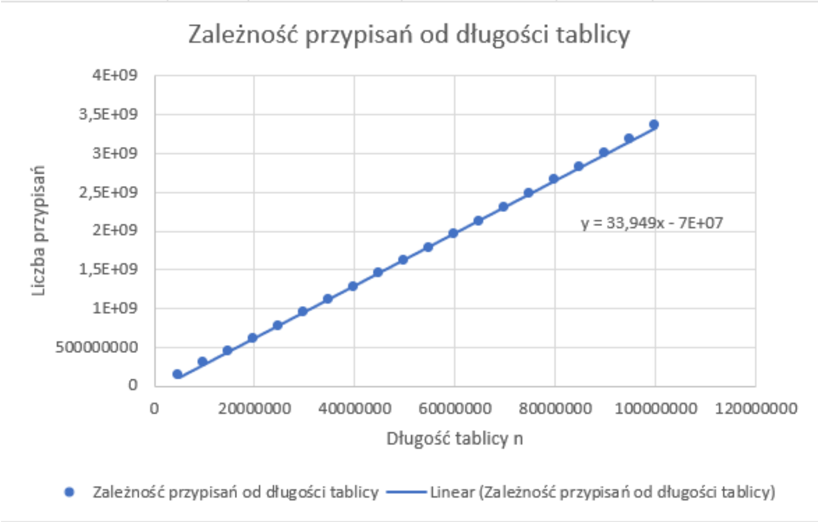
Długość tablicy n	Czas [ms]	Przypisania	Porównania
5000000	2057.6	140466500.0	301969390.8
10000000	4666.0	293351114.8	631445316.8
15000000	6558.8	451397322.4	972287178.4
20000000	8867.2	611206538.4	1319317936.0
25000000	11486.4	774218025.6	1670016530.0
30000000	14624.6	940049453.6	2026133481.0
35000000	18603.2	1107236398.0	2385863241.0
40000000	21102.6	1275363186.0	2747592970.0
45000000	24519.2	1444197728.0	3113601646.0
50000000	28818.8	1613586508.0	3482108847.0
55000000	33611.2	1783420720.0	3849282755.0
60000000	36858.8	1953609431.0	4219491290.0
65000000	35404.8	2124331277.0	4590938244.0
70000000	44148.0	2298061218.0	4962213423.0
75000000	46873.0	2472651957.0	5336884480.0
80000000	49966.8	2647920249.0	5712584208.0
85000000	51761.2	2823777184.0	6089670249.0
90000000	54124.6	3000141188.0	6470141900.0
95000000	57978.4	3176951462.0	6850619846.0
100000000	65987.4	3354152304.0	7231154219.0

Analiza wykresów



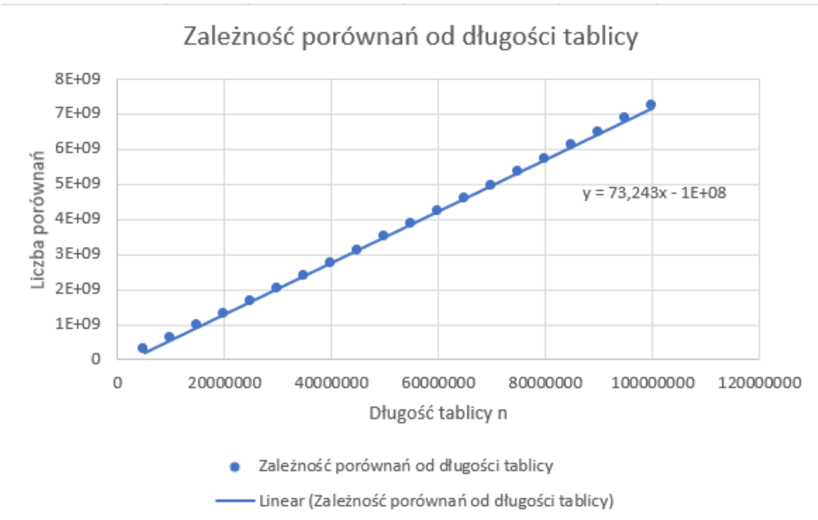
Rysunek 22: Zależność czasu wykonywania od długości tablicy dla algorytmu Ternarny Heap Sort

Zachowuje złożoność $O(n)$ ale z mniejszą stałą. Kopiec trójkowy redukuje wysokość drzewa, co teoretycznie powinno zmniejszyć liczbę iteracji ze względu na większą liczbę operacji w jednej iteracji.



Rysunek 23: Zależność liczby przypisań od długości tablicy dla algorytmu Ternarny Heap Sort

Mimo redukcji w czasie wykonywania, liczba przypisań zachowuje złożoność $O(n)$ oraz występuje redukcja w liczbie przypisań względem binarnego kopca.



Rysunek 24: Zależność liczby porównań od długości tablicy dla algorytmu Ternarny Heap Sort

Pomimo utrzymania złożoności $O(n)$, liczba porównań jest redukowana dzięki efektywniejszej strukturze kopca trójkowego.

0.4.3 Wnioski

Ternarny Heap Sort wykazuje widoczną poprawę wydajności czasowej, co można przypisać do mniejszej liczby iteracji ze względu na trójkową budowę kopca. Algorytm zachowuję złożoność $O(n)$, ale zmiana zauważalna jest w zmniejszonym współczynniku.

Wersja ternarna wykonuje znacznie mniej przypisań, ponieważ korzeń drzewa porównywany jest z trzema elementami, zamiast dwóch, co redukuje finalną sumę przypisań.

Mimo redukcji w liczbie przypisań, zauważalna jest też redukcja w liczbie porównań.