Sorting Algorithms

Project 1
Algorithm Design and Analysis

Danil Krassotov 282656

Introduction

The purpose of this report is to implement selected sorting algorithms and analyze their efficiency. As part of this work, Timsort, Dual-pivot Quicksort, and heap sort algorithms were implemented. For each algorithm, the sorting time was studied for arrays of different sizes, and graphs were generated showing their complexities.

1 Algorithm Description

1.1 Timsort

Timsort is a hybrid sorting algorithm that combines **insertion sort** and **merge sort**. The algorithm shown in Listing 1 is a simplified version of Timsort, containing basic elements such as dividing into runs, insertion sort, and merging runs using merge sort method. However, it does not include the full run stack logic and advanced optimizations. Timsort is effective when sorting large datasets with already partial order, but in situations where data is completely random, it may not have a significant advantage over other algorithms.
Computational complexity:

- Best case: *O(n)* – with already sorted data.
- Average case: *O(n log(n))*
- Worst case: *O(n log(n))*
- Space complexity in worst case: *O(n)*

```cpp
void timsort(vector<int>& arr) {
    int n = arr.size();
    int mr = minrun(n);

    for (int i = 0; i < n; i += mr) {
        int end = min(i + mr - 1, n - 1);
        insertion_sort(arr, i, end);
    }

    for (int size = mr; size < n; size *= 2) {
        for (int start = 0; start < n; start += 2 * size) {
            int mid = start + size - 1;
            int end = min(start + 2 * size - 1, n - 1);

            if (mid < end) {
                merge(arr, start, mid, end);
            }
        }
    }
}
```

Listing 1: Timsort - C++ implementation

Visually, the algorithm can be represented as follows:

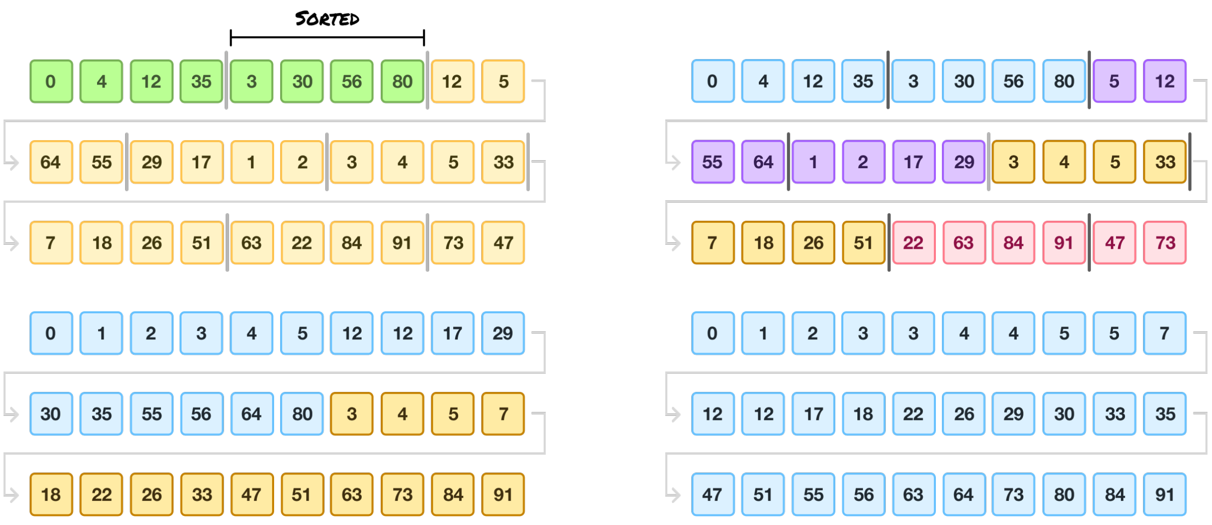


Figure 1: Timsort

1.2 Heap Sort

This algorithm is based on the data structure called a heap. The algorithm builds a heap from the input array, then removes elements from the heap root and places them at the end of the array.
The algorithm presented in Listing 2 contains two versions of sorting:

- **heapsort()** - ascending sorting,
- **heapsort_reverse** - descending sorting.

```cpp
void heapsort(vector<int> &arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        kopiec_max_to_min(arr, i, n);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        kopiec_max_to_min(arr, 0, i);
    }
}

void heapsort_reverse(vector<int> &arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--) {
        kopiec_min_to_max(arr, i, n);
    }
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        kopiec_min_to_max(arr, 0, i);
    }
}
```

Listing 2: Heapsort - C++ implementation

In both cases, data is first transformed into a binary heap, then the largest/smallest element is swapped with the last and removed from the heap. Then the heap structure is restored and the process repeats until the entire array is sorted. The algorithm requires no additional memory and guarantees computational complexity equal to O(n log(n)) regardless of input data. The disadvantage is somewhat lower performance compared to, for example, Timsort.
The function with minimum heap (**heapsort_reverse**) was used to create an array sorted in reverse order when measuring the execution time of the function (**heapsort()**) on different arrays.
Computational complexity:

- Best case: *O(n log(n))*
- Average case: *O(n log(n))*
- Worst case: *O(n log(n))*

- Space complexity: *O(1)* – the algorithm works in-place

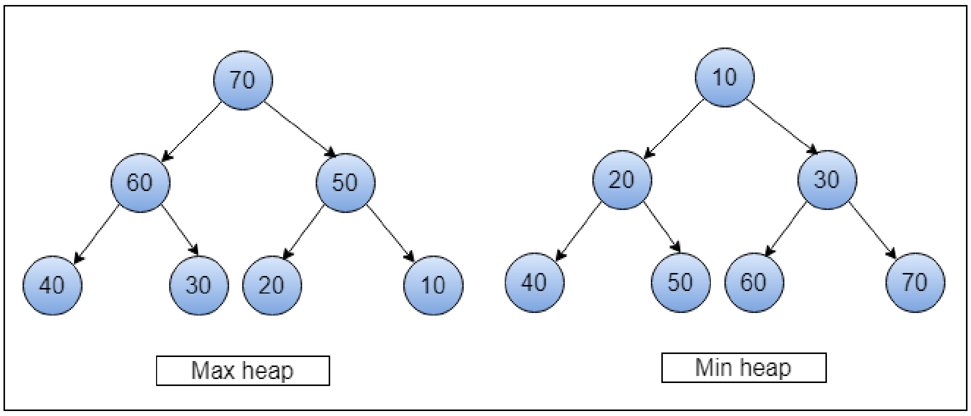


Figure 2: Visualization of maximum and minimum heap

1.3 Dual-Pivot Quicksort

Dual-pivot quicksort is a variant of the quicksort algorithm that uses two pivots instead of one. The array is divided into three parts: elements smaller than the first pivot, elements between the pivots, and elements larger than the second pivot. The algorithm repeats for each part until the array is sorted.
The implementation shown in Listing 3 contains two versions of the algorithm: one sorting in ascending order (qsort), the other in reverse (qsort_reverse). Similar to the previous algorithm, the reverse version was used to create an already ordered array, but in descending order, to observe the algorithm's behavior in the theoretically "worst" case.
Dual-pivot quicksort is simple to implement, fast, and requires additional memory only for recursion. However, it is very sensitive to input data and pivot selection: if data is frequently duplicated or already sorted, this can significantly reduce performance. For large and unfavorable data, the algorithm can cause very deep function calls resulting in stack overflow and program termination.
Computational complexity:

- Best case: *O(n log(n))*
- Average case: *O(n log(n))*
- Worst case: $O(n^2)$ – with poor pivot selection.
- Space complexity: *O(log(n))* – in recursive version, without additional memory allocation.

```
void dualpivot(vector<int> &arr, int left, int right) {
    ...

    while (i <= rk) {
        if (arr[i] < lp) {
            swap(arr[i], arr[lk]);
            lk++;
            i++;
        } else if (arr[i] > rp) {
            swap(arr[i], arr[rk]);
            rk--;
        } else {
            i++;
        }
    }

    ...

    dualpivot(arr, left, lk - 1);
    dualpivot(arr, lk + 1, rk - 1);
    dualpivot(arr, rk + 1, right);
}

void qsort(vector<int> &arr) {
    if (arr.size() > 1) {
        dualpivot(arr, 0, arr.size() - 1);
    }
}
```

Listing 3: Dual-Pivot Quicksort - C++ implementation

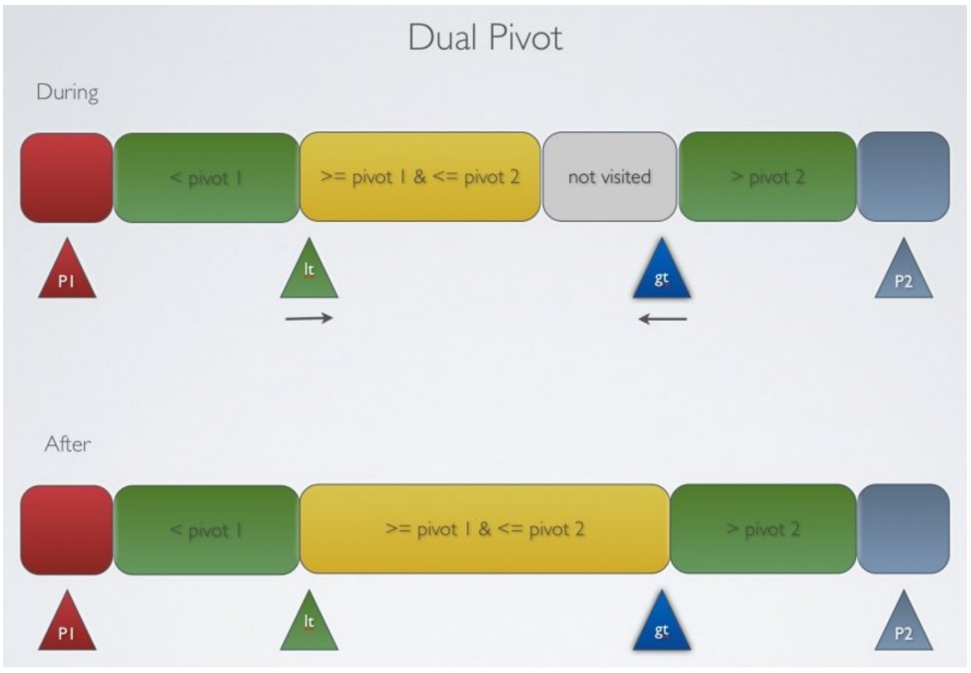
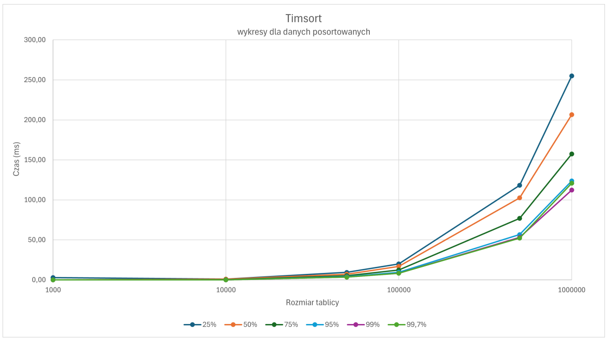


Figure 3: Visualization of *dualpivot()* function

2 Algorithm Analysis

For each algorithm, execution time measurements were performed for sorting arrays of sizes N = 1000, 10000, 50000, 100000, 500000, 1000000. Based on this data, graphs were created to fit theoretical computational complexity models to measurement results.
All implemented algorithms have time complexity *O(nlog(n))*, so the fitted model is based on the function
$f(n) = n * log(n)$,
where n - array size
Graphs:



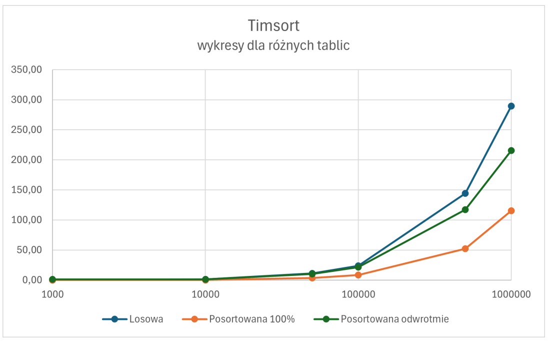


Figure 4: Measurements for Timsort

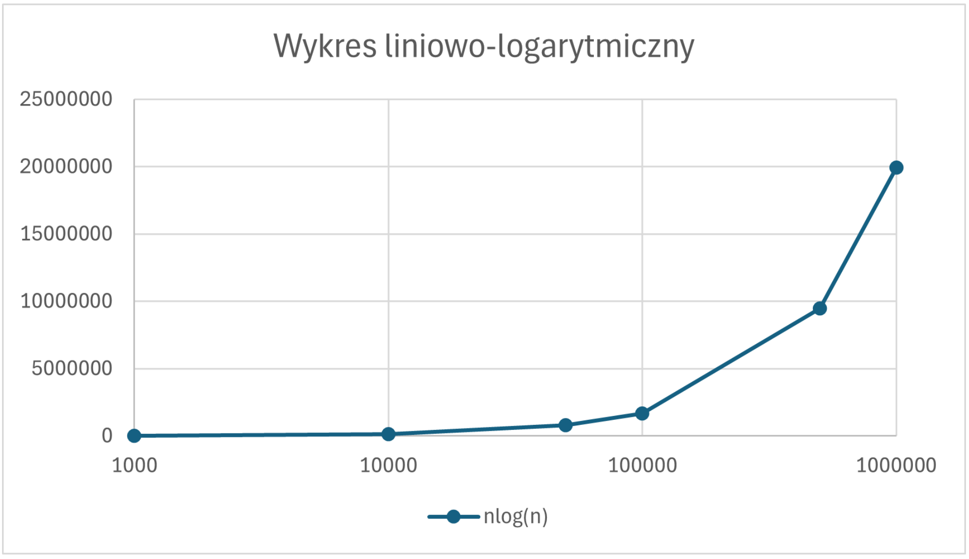


Figure 5: Graph of $f(n) = n * log(n)$

- Time complexity appears to be *O(nlog(n))*, which matches theoretical expectations.
- Time growth is lower with sorted data.

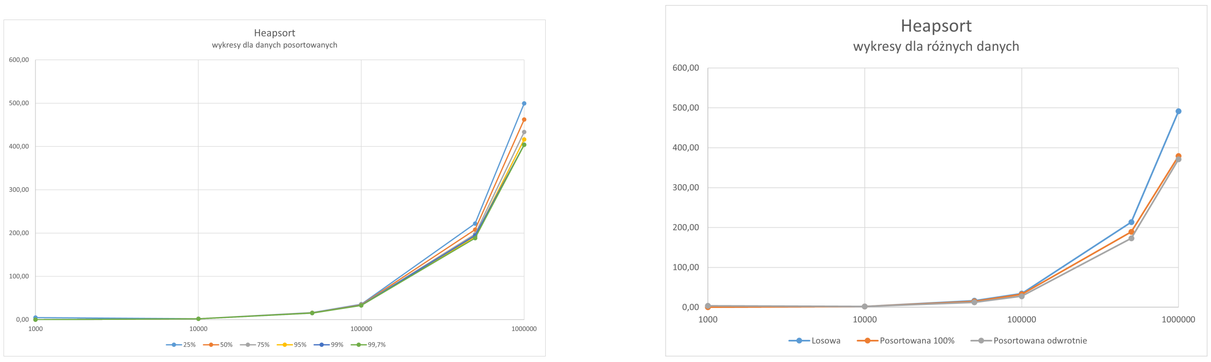


Figure 6: Measurements for Heapsort

- Time complexity appears to be $O(nlog(n))$ and is stable, which matches theoretical expectations.
- It can be seen that it's somewhat slower than Timsort
- Behavior is almost identical for different data types.

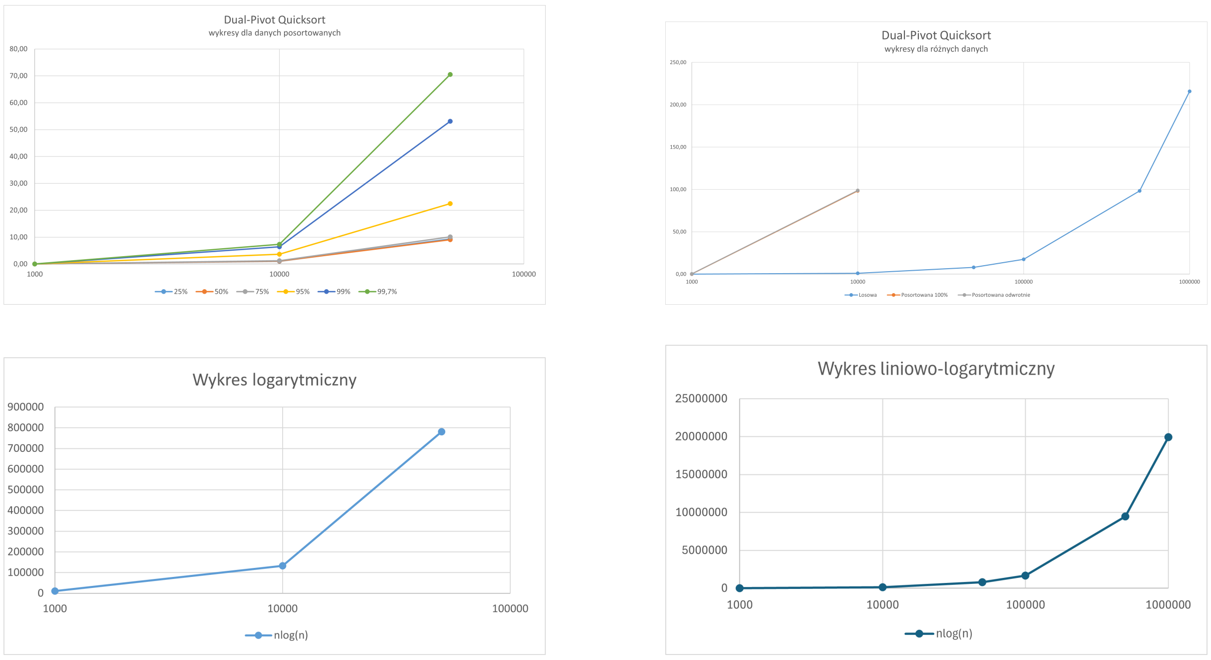


Figure 7: Measurements for Dual-Pivot Quicksort

- For random data it is very fast, time complexity appears to be $O(nlog(n))$
- For sorted data, which is unfavorable for this algorithm, time increases dramatically

3 Conclusions

Based on the analysis of sorting algorithms, theoretical complexity, and conducted research, it can be stated that Timsort has an advantage for partially sorted data - the sorting time was then significantly shorter than with random data. Heap sort has stable execution time regardless of input data. Dual-Pivot Quicksort achieved very good results for random data, however its execution time drastically increased with sorted data, which most likely results from poor optimization - the program overflows the stack by repeatedly calling the function. Despite the same theoretical complexity, algorithms differ significantly in terms of execution time in practice and can be sensitive to specific types of data.