CSE331 - Assignment #1

Jaemin Kim (20211061)
UNIST
South Korea
jm611@unist.ac.kr

1 PROBLEM STATEMENT

We aim to analyze 6 conventional comparison-based sorting algorithms and 6 contemporary sorting algorithms with respect to the following basic properties: execution time, memory consumption, stability, online capability, cache-friendliness, etc. The 6 conventional comparison-based sorting algorithms are **Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort** and the 6 contemporary sorting algorithms are **Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, and Intro Sort**.

To evaluate sorting algorithms, we use datasets of various sizes n which is (1K-1M) across 5 different dataset types:

- **Random**: a sequence of integers from 0 to n - 1 randomly shuffled without repetition
- **Dense random**: a randomly generated sequence of n integers in the range 0 to 99, allowing many duplicates
- **Ascending**: a sequence of integers from 0 to n - 1 sorted in the ascending order
- **Descending**: a sequence of integers from n - 1 to 0 sorted in the descending order
- **Partially sorted**: a sequence including a sorted subsequence of length n/2, with the remaining elements randomly arranged

Beyond the basic descriptions of algorithms, the discussion on conventional comparison sorting algorithms will include modern adaptations and some modifications aimed at performance improvements or practical limitations. Moreover, the analysis of contemporary sorting algorithms will focus on their key attributes and the conditions under which its performance is maximized or minimized.

The implementations of the sorting algorithms are available on GitHub.[1]

2 BASIC SORTING ALGORITHMS

This section explains how the conventional comparison-based sorting algorithms work and the theoretical analysis on them. We implements them based on the pseudo code from our text book, Introduction to Algorithm [1], with slight modifications. Table 1 shows the overall comparison of 6 basic sorting algorithms.

2.1 Merge Sort

Merge sort is an efficient sorting algorithm which is based on Divide-and-Conquer strategy. It recursively divides the array to be sorted into two parts until each part is no longer divisible. Then, it recursively merges two sorted, or length-1, sub-arrays to make one sorted sub-array and finally it generates a sorted array. The array is divided in *logn* levels, and merging operations over all n elements are performed at each level. For this reason, it has $\mathbf{O(nlogn)}$ **time complexity** regardless of the input. This algorithm requires $\mathbf{O(n)}$ **additional space** in our implementation because temporary arrays are used during the merging process in order to store intermediate results. Merge sort can maintain the relative order of elements when merging, thus it is a **stable sorting algorithm**. However, it is **not an online algorithm** since it requires access to the entire array before actual sorting begins. Additionally, Merge sort is generally considered having **moderate cache-friendliness** because it accesses elements in an array sequentially during merging while its recursive division leads to scattered memory accesses.

2.2 Heap Sort

Heap sort is a sorting algorithm leveraging the properties of heap. First, it builds max heap or min heap in linear time with multiple Heapify calls. Then, it swap the root node of the heap with the end of the array, reduce the size of the heap, and then maintain the heap property through Heapify again. Building heap takes O(n) time, and Heapify, which takes O(log n), is executed in O(n) times. In total, Heap sort has $\mathbf{O(nlogn)}$ **time complexity**. Since the array is maintained in a heap structure, then Heap sort can be performed **in-place**, which means $\mathbf{O(1)}$ **additional space** required. However, Heap sort is an **unstable sorting algorithm** which means it may not maintain the relative order of equal elements. Moreover, it is **not an online algorithm** because the heapify operation assumes that all elements are already available. Finally, Heap sort can be considered **cache-unfriendly**, considering the accesses to child or parent of a node are often not sequential but jumps.

2.3 Bubble Sort

Bubble sort is a simple sorting algorithm that iteratively compares two elements and swaps them if they are in the wrong order. The worst case and average case time complexity is $\mathbf{O(n^2)}$, which is caused by the number of all possible combinations of two elements. Originally, it repeats this simple process until all possible combinations of two elements have been tried. However, we add some modification for early exiting if swapping does not occur during an element is compared with all elements. This modification leads the best case time complexity to be $\mathbf{O(n)}$ while the original one is $\mathbf{O(n^2)}$. It is obvious that Bubble sort is an **in-place** algorithm which requires $\mathbf{O(1)}$ **additional space**. Moreover, the two elements having the same value will not swap, which leads Bubble sort to be **stable sorting algorithm**. Bubble sort is not **an online algorithm**. Besides, it is **cache-unfriendly** because it performs a lot of redundant swaps.

2.4 Insertion Sort

Insertion sort is a sorting algorithm that gradually expands the sorted sub-array by inserting an element. Its **time complexity is** $\mathbf{O(n^2}$ in the average and worst cases due to nested comparisons

[1] https://github.com/ginjae/sorting-algorithms-cpp

Sort Algorithm	Time Complexity (best / avg / worst)	Space	In-Place	Stable	Online	Cache-Friendly
Merge Sort	$O(n \log n)$	$O(n)$	No	Yes	No	Moderate
Heap Sort	$O(n \log n)$	$O(1)$	Yes	No	No	Unfriendly
Bubble Sort (early exit)	$O(n)$ / $O(n^2)$ / $O(n^2)$	$O(1)$	Yes	Yes	Yes	Unfriendly
Insertion Sort	$O(n)$ / $O(n^2)$ / $O(n^2)$	$O(1)$	Yes	Yes	Yes	Friendly
Selection Sort	$O(n^2)$	$O(1)$	Yes	No	No	Unfriendly
Quick Sort	$O(n \log n)$ / $O(n \log n)$ / $O(n^2)$	$O(\log n)$	Yes	No	No	Friendly

Table 1: Comparison of Basic Sorting Algorithms

and shift, but it has **O(n) time complexity** in the best case when the array is already sorted. The implementation of Insertion sort can be **in-place**, and moreover, it is **stable** and **online algorithm** due to its nature that sorts the array gradually from the beginning of the array. More importantly, the algorithm is **cache-friendly** because the operations works mostly on contiguous elements in a small sub-array.

2.5 Selection Sort

Selection sort is a simple sorting algorithm that selects a smallest element in unsorted part of the array, then swaps with the current element. Selection sort takes $\mathbf{O(n^2)}$ **time** in the average, best, worst cases, while swap operations are performed up to n - 1 times. It uses swap operations with the smallest element; therefore, it is an **in-place** algorithm. Selection sort is **unstable** and **cache-unfriendly** algorithm because it often compares and swaps with two distant elements in the whole array. If a new minimum value is entered, the sorting must be restarted, which means Selection sort is **not an online algorithm**.

2.6 Quick Sort

Quick sort is a fast sorting algorithm which is based on Divide-and-Conquer strategy. It selects an element as a pivot and partitions the array into two parts that one part has smaller values than the pivot and another part has larger values than the pivot. The process is recursively performed until no more partitions are possible. The worst case is when the pivot is always minimum or maximum value, so we add some modification to Quick sort to avoid the worst case in reversely sorted arrays by randomly selecting pivots. For this reason, our Quick sort takes $\mathbf{O(nlogn)}$ **time**, while Quick sort using the first or last element in a partition as the pivot takes $O(n^2)$ time in the worst case. One interesting thing about Quick sort is that the algorithm is **in-place**, but its recursive call stacks require $\mathbf{O(nlogn)}$ **space**. The implementation of Quick sort is an **unstable** and **online algorithm** because of the characteristics of partitioning. The reason why Quick sort is quick is its **cache-friendliness**. In particular, it works well in cache as the partitions become smaller.

3 ADVANCED SORTING ALGORITHMS

This section explains how the advanced sorting algorithms work and the theoretical analysis on them. We also focus on their key properties and the conditions under which its performance is maximized or minimized. Table 2 shows the overall comparison of 6 advanced sorting algorithms.

3.1 Library Sort

Library sort is a sorting algorithm that utilizes 'gaps' to reduce the number of shift operations to be performed when inserting an element. The algorithm is similar to Insertion sort in terms of inserting one element at a time, but Library sort maintains gaps between elements and find the position to insert by using binary search. It takes $\mathbf{O(nlogn)}$ **time** in the average case because of the use of binary search and gap. However, it takes $\mathbf{O(n^2)}$ **time** in the worst case when the gap is unbalanced, then there is no gap near the position to insert the element. Then, when re-balancing the array, which adds a gap between all elements, it requires **O(n) additional space**. The implementation of Library sort can be a stable algorithm if the binary search always finds the rightmost element among the elements that have the same value. Like Insertion sort, Library sort is also **an online algorithm**. Moreover, it is a **cache-unfriendly** algorithm because it uses binary search that does not utilize spatial locality, and the array becomes a sparse array with a lot of gaps.

Algorithm 1 Library Sort

Require: Array A of length n
Ensure: A is sorted in non-decreasing order
1: $S \leftarrow$ new array of size $c \cdot n$ with elements of A and gaps between elements
2: **for** $i \leftarrow 1$ to $lg(n)$ **do**
3: Rebalance S to make it size 2^i (adding gaps between elements)
4: **for** $j \leftarrow 2^{i-1}$ to $\min(2^i - 1, n)$ **do**
5: $idx \leftarrow Binary\ Search(S, A[j], 1, 2^i)$
6: // Search the position to insert the element A[j]
7:
8: $insert\_with\_shift(S, A[j], 1, 2^i, idx)$
9: // If the position is a gap, just insert.
10: // Otherwise, shift element in the position then insert
11: **end for**
12: **end for**

3.2 Tim Sort

Tim sort is a hybrid sorting algorithm that combines Merge sort and Insertion sort. The core idea of the algorithm is leveraging the already sorted 'run', then using Insertion sort in small sections and using Merge sort in large sections. It takes **O(n) time** in the best case when just checking the sorted runs. and takes $\mathbf{O(nlogn)}$ **time** in the average and worst case because Merge sort guarantees $O(nlogn)$ time. When it merges runs into one run, an **additional O(n) space**

Sort Algorithm	Time Complexity (best / avg / worst)	Space	In-Place	Stable	Online	Cache-Friendly
Library Sort	$O(n \log n)$ / $O(n \log n)$ / $O(n^2)$	$O(n)$	No	Yes	Yes	Unfriendly
Tim Sort	$O(n)$ / $O(n \log n)$ / $O(n \log n)$	$O(n)$	No	Yes	No	Friendly
Cocktail Shaker Sort	$O(n)$ / $O(n^2)$ / $O(n^2)$	$O(1)$	Yes	Yes	No	Unfriendly
Comb Sort	$O(n \log n)$ / $O(n^2)$ / $O(n^2)$	$O(1)$	Yes	No	No	Unfriendly
Tournament Sort	$O(n \log n)$	$O(n)$	No	No	No	Unfriendly
Intro Sort	$O(n \log n)$	$O(\log n)$	Yes	No	No	Moderate

Table 2: Comparison of Advanced Sorting Algorithms

is required. Like Merge sort, it can maintain the relative order of elements that have the same value, which means the algorithm is **stable**. This, like Merge sort, is **not an online algorithm**. However, this algorithm works in the **cache-friendly** manner since it uses Insertion sort which has high cache-friendliness.

Algorithm 2 Tim Sort

```
Input: Array A of length n
Output: Sorted array A
minrun ← calculateMinRun(n)
i ← 0
left ← 0
size ← minrun
while i < n do
    tmp ← min(i + run − 1, n − 1)
    Insertion Sort(A, i, tmp)
    i ← i + run
end while
while size < n do
    while left < n do
        mid ← left + size − 1
        right = min(left + 2 ∗ size − 1, n − 1)
        if mid < right then
            merge(A, left, mid, right)
        end if
        left ← left + 2 ∗ size
    end while
    left ← 0
    size ← size ∗ 2
end while
```

This pseudo code of Tim sort is from Wikipedia.[2]

3.3 Cocktail Shaker Sort

Cocktail Shaker sort is a sorting algorithm that is a modified version of Bubble sort algorithm. It alternates between both directions, which improves sorting efficiency while Bubble sort performs in one direction. While it can reduce the number of iterations compared to Bubble sort, most of its properties follow Bubble sort. Therefore, it takes **O(n) time** in the best case and $\mathbf{O(n^2)}$ **time** in the average and worst cases. Like Bubble sort, it is **in-place** and **stable**, but **not an online algorithm**. Moreover, it is **cache-unfriendly** algorithm because swap operations are cache-unfriendly and backward traversing of the array can also be cache-unfriendly

[2]https://en.wikipedia.org/wiki/Timsort

Algorithm 3 Cocktail Shaker Sort

```
Require: Array A of length n
Ensure: A is sorted in non-decreasing order
begin ← 1
end ← n
while begin <= end do
    new_begin ← end − 1
    new_end ← begin
    for i = begin to end − 1 do
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1]
            new_end = i
        end if
    end for
    end ← new_end − 1
    for i = end − 1 downto begin do
        if A[i] > A[i + 1] then
            swap A[i] and A[i + 1]
            new_begin = i
        end if
    end for
    begin ← new_begin + 1
end while
```

3.4 Comb Sort

Comb sort is a sorting algorithm that improves Bubble sort by solving the problem that small values at the end of the array take a long time to be sorted. Unlike Bubble sort, it repeatedly compares two elements that are separated by 'gap' which is decreasing by 'shrinking factor'. It takes $\mathbf{O(n^2)}$ **time** in the worst case due to its base of Bubble sort and in the average case because the algorithm behaves like Bubble sort when gap is smaller, though it reduces the number of comparisons early by using larger gaps. However, Comb Sort often runs faster in practice than Bubble sort because large gaps help fix out-of-place elements earlier. Comb sort takes **O(nlogn)** time in the best case when the input is already sorted because it does not compare one element to all other elements at a iteration so that it cannot exit early. Moreover, it is obvious that Comb sort is an **in-place** algorithm and **not an online algorithm**. Due to swapping two elements that are far apart, it is **unstable** and **cache-unfriendly**.

This pseudo code of Comb sort is based on An Efficient Variation of Bubble Sort [2].

Algorithm 4 Comb Sort

Require: Array A of length n
Ensure: A is sorted in non-decreasing order

```
gap ← n/2
shrink ← 4/3
for ℓ = 1 to t do
    inc ← gap
    gap ← max(⌊gap/shrink⌋, 1)
    for i = 1 to n − inc do
        if A[i] > A[i + inc] then
            swap A[i] ↔ A[i + inc]
        end if
    end for
end for
Bubble Sort(A)
```

3.5 Tournament Sort

Tournament sort is a sorting algorithm that leverages Tournament tree so that it extracts maximum or minimum value repeatedly. Tournament tree is a complete binary tree having the array values as leaves. In our implementation of Tournament tree, the parent is the smaller of the two children. It extracts the root node and re-construct the tree. The tree height is bounded by log n; therefore, re-constructing n times takes O(log n), which means the algorithm takes **O(nlogn) times**. Since the Tournament tree should be constructed with size O(n)—larger than or equal to n—, then it takes **O(n) additional space**, which means it is **not an in-place** algorithm. Moreover, the nature of constructing the tree makes the algorithm **unstable**, **not an online algorithm**, and **cache-unfriendly**. The advantage of this algorithm includes that re-constructing the tree modifies only one path from the root to a leaf, and is maximized if the cost of comparison is immensely high.

Algorithm 5 Tournament Sort

Require: Array A of length n
Ensure: A is sorted in non-decreasing order

```
Build a Tournament Tree T
Set array elements as leaves of T
Internal node initialization with the smaller value of two children
for i = 1 to n do
    min ← winner at the root of T
    Append min to the sorted output
    Replace min in the tree (going downward) with ∞
    Reconstruct the Tournament tree T
end for
```

3.6 Intro Sort

Intro sort is a hybrid sorting algorithm that combines Quick sort, Heap sort, and Insertion sort. Initially, Quick sort is performed, but Heap sort is performed when the recursion depth exceeds a certain threshold. At the end of the whole process, it uses Insertion sort to sort short unsorted sequences. Since Quick sort switches to Heap sort when the pivot selection is the worst choice that increases the recursion depth, then the whole algorithm takes **O(nlogn) time**. Quick sort, Heap sort, and Insertion sort are all in-place algorithms, then Intro sort is also **in-place** algorithm, while Quick sort of it takes **O(logn)** space as recursive call stack. It is obvious that Intro sort is **unstable** and **not an online algorithm** when considering the three sorting algorithms used. In addition, it has **moderate cache-friendliness** because Quick sort and Insertion sort are cache-friendly, but Heap sort is cache-unfriendly.

Algorithm 6 Intro Sort

Require: Array A,
f: the first position of A,
b: the first position beyond the end of A
Ensure: A is sorted in non-decreasing order

```
Intro Sort Loop(A, f, b, 2 * ⌊lg(b − f)⌋)
Insertion Sort(A, f, b)
```

Algorithm 7 Intro Sort Loop

Require: $A, f, b, depth\_limit$
Ensure: A is sorted in non-decreasing order

```
while b − f > size_threshold do
    if depth_limit = 0 then
        Heap Sort(A, f, b)
        return
    end if
    depth_limit ← depth_limit − 1
    p ← Partition(A, f, b, Median_of_3(A[f], A[f + (b − f)/2], A[b − 1]))
    b ← p
end while
```

These pseudo codes of Intro sort are based on Introspective Sorting and Selection Algorithms [3].

4 EXPERIMENTAL RESULTS AND ANALYSIS

We experiment the 12 sorting algorithms on 5 types of datasets whose input sizes are varied from (1K-1M)—such as 1K, 10K, 100K, 200K, 400K, 600K, 800K, and 1M—. We will analyze the average running time, peak memory usage, and stability for each dataset.

4.1 Running time

The experiment was performed to evaluate the average running time of each sorting algorithms. We run the algorithms 10 times on each dataset except in cases where it takes huge time. Specifically, we repeat 3 times on datasets having a size of 1M if the sorting algorithm is Bubble sort, Insertion sort, Selection sort, or Cocktail Shaker sort. These algorithms have $O(n^2)$ time complexity in the average case; therefore, a single iteration of the one of these algorithm with an 1M-element dataset takes around 30 minutes in our environment, though it varies slightly depending on which algorithm is used. Figure 1, 2, 3, 4, 5 shows the complete view of the average running time of each algorithm in each dataset. Note that both the x-axis and the y-axis are on a logarithmic scale in

each graph. Therefore, the slope of the line corresponds to the exponent in the time complexity, indicating the asymptotic growth rate of each algorithm. A steeper slope means it has a higher time complexity.

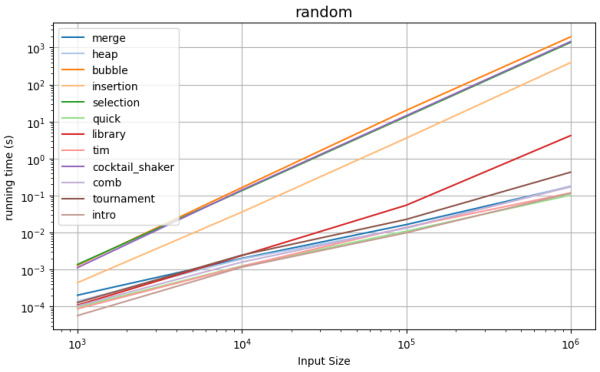

Figure 1: Average running time for Random dataset

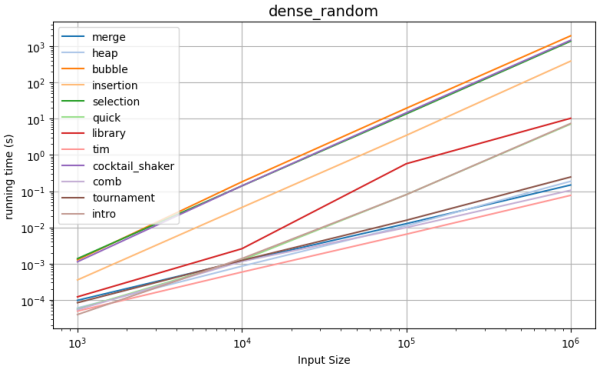

Figure 2: Average running time for Dense random dataset

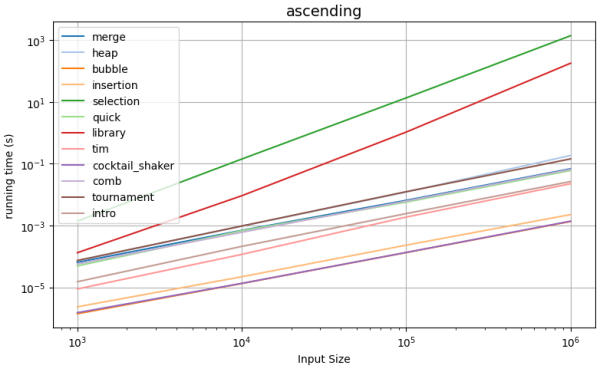

Figure 3: Average running time for Ascending dataset

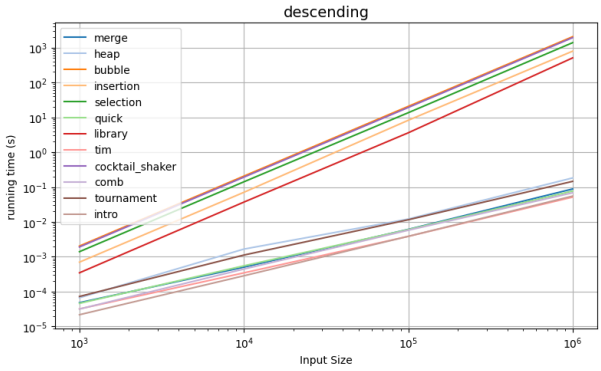

Figure 4: Average running time for Descending dataset

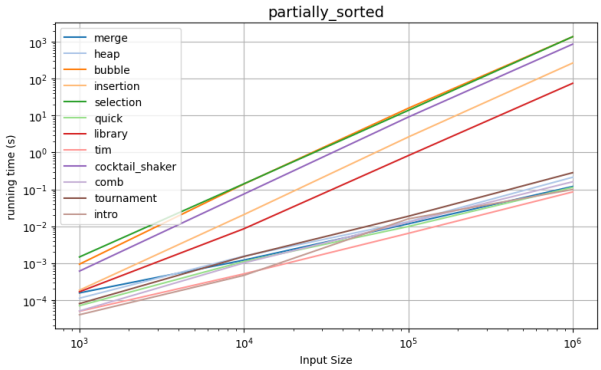

Figure 5: Average running time for Partially sorted dataset

4.2 Running time analysis

Figure 1, 2, 3, 4, 5 shows that Tim sort achieves excellent performance in overall, which appears to be due to its time complexity with cache-friendliness. Moreover, since real-world datasets often contain a large number of duplicate values, the results on Dense random dataset suggest that using Tim sort is a reasonable choice in many real-world cases. . More interestingly, sorting algorithms that are based on Bubble sort with early exit achieves faster speed than Tim sort when sorting an already sorted array. Figure 4 shows that Intro sort (and Tim sort, obviously) tends to be the best choice if a reversely sorted array should be considered. Regarding Library Sort, sorting an already sorted array can actually be a worst-case scenario for it as Figure 3 shows. To analyze, the performance of library sort becomes excellent when elements are distributed evenly between gaps in the array so that gaps are distributed evenly. Moreover, when sorting Descending dataset, Bubble sort, Insertion sort, Selection sort, and Cocktail Shaker sort records poor performances because that dataset is the worst case of these sorting algorithms. As mentioned earlier, a modification added to Bubble sort by allowing early exit leads it to perform efficiently in the best case when the array is already sorted (Ascending dataset). Figure 3 shows that Bubble sort is at the bottom of the graph. Additionally, the pivot selection strategy in Quick sort is randomized to avoid the worst case

Algorithm	Memory Usage (B)	Memory Usage (MB)	Space complexity
Merge Sort	4,112,384	3.92	O(n)
Heap Sort	0	0	O(1), in-place
Bubble Sort	0	0	O(1), in-place
Insertion Sort	0	0	O(1), in-place
Selection Sort	0	0	O(1), in-place
Quick Sort	0*	0*	O(log n), in-place
Library Sort	32,264,192	30.77	O(n)
Tim sort	3,858,432	3.68	O(n)
Cocktail Shaker Sort	0	0	O(1), in-place
Comb Sort	0	0	O(1), in-place
Tournament Sort	16,871,424	16.09	O(n)
Intro sort	0*	0*	O(log n), in-place

Table 3: Memory usage comparison of sorting algorithms on a 1M-element array

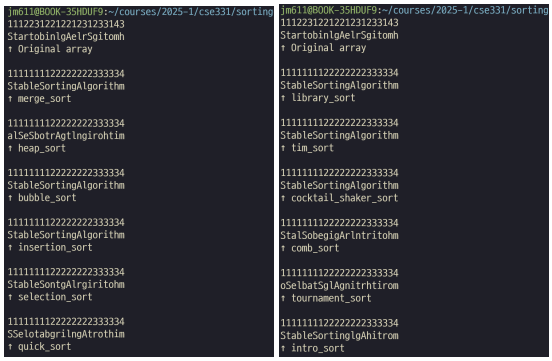

Figure 6: Sorting algorithm stability test

that is reversely sorted arrays (Descending array). Figure 4 shows that Quick sort achieves not terrible performance consequently.

4.3 Memory usage

Table 3 shows memory usages of each sorting algorithm on Random dataset of size 1M. We measure memory usage by recording the peak memory of the current program before and after calling the sorting function, and calculating the difference between them. Due to this measurement method, the reported memory usage for Quick sort and Intro sort appears as zero, which is an inaccurate result. However, this result implies that Quick sort and Intro sort requires not significantly large memory space even if the input size becomes huge—such as 1M—. Except for the two sorting algorithms, we acquire the result that we expect in the theoretical step. Specifically, Merge sort and Tim sort have the similar memory usage and Tournament sort has larger memory usage because of newly constructed Tournament tree. Moreover, it is easily seen that Library sort has the largest memory usage because it constructs O(n) number of gaps in an array.

4.4 Stability

Figure 6 shows whether each of sorting algorithms is stable or not. We perform a test with an array of pairs—(integer, char)—. When the char values of the elements in a sorted array are listed, the sorting algorithm used is stable if the string the char values make is "StableSortingAlgorithm". Then, we acquire the same result as Table 1 and Table 2.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction to Algorithms.* MIT Press.
[2] Wlodzimierz Dobosiewicz. 1980. An Efficient Variation of Bubble Sort. *Inf. Process. Lett.* 11 (1980), 5–6.
[3] David R. Musser. 1997. Introspective sorting and selection algorithms. *Softw. Pract. Exper.* 27, 8 (Aug. 1997), 983–993.