# Timsort and Introsort

Advanced Data Structures

Unai Perez Mendizabal

## 1 Introduction

Timsort is a hybrid sorting algorithm, named after its creator Tim Peters. As hybrid sorting algorithms do, it combines two other algorithms, in this case merge sort and insertion sort.

Introsort, or introspective sort, is yet another hybrid sorting algorithm with optimal worst-case and average time complexities. It is a combination of insertion sort, quicksort and heapsort.

## 2 Algorithms

In this section, all the implemented algorithms, needed to reach the goal of this homework, will have their basic functioning explained, and overall details regarding performance will be discussed.

### 2.1 Insertion sort

Insertion sort is one of the simplest of the known sorting algorithms. It is usually compared to sorting a deck of cards. Cards, one by one and starting from the second one in the list, are compared to the cards prior to them, until a card with a smaller number is found. Then, the all the passed cards are moved a step forward to fit the initial card in its place. Figure 1 represents this process.

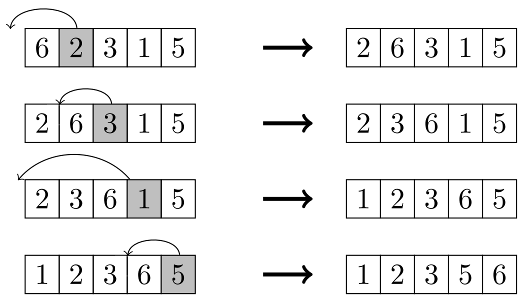

Figure 1: Example of insertion sort

## 2.2 Merge sort

Merge sort works by splitting the element list in half recursively until a one element slice remains; then, all slices, at the different levels of recursion, are merged (as implied by the name). Therefore, the most important operation of this algorithm is the merge of two slices (i.e., specific fragments of a bigger list), which consists of joining two sorted lists of elements into one sorted list. This operation is represented in figure 2.

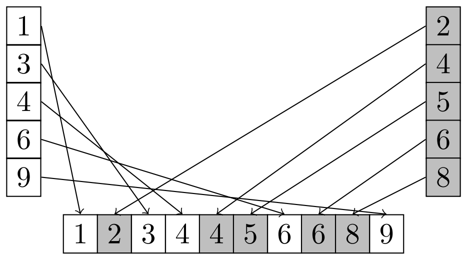

Figure 2: Example of how the merge operation of mergesort is performed

## 2.3 Quicksort

Quicksort works by choosing a reference element from the list, called pivot, then placing all elements smaller than the pivot to its left, and those larger to its right. This procedure is called partitioning. The pivot, at this point, is on its correct position, and only the sublists to its left and right need to be ordered.

Figure 3 shows an example of the partitioning procedure. Any element can be the pivot, but in this case, the last element is given the role. The dark grey element is the one that is compared with the pivot. If it is smaller than the pivot, the dark grey element and the one signed at by the arrow are swapped. Otherwise, no change is made.

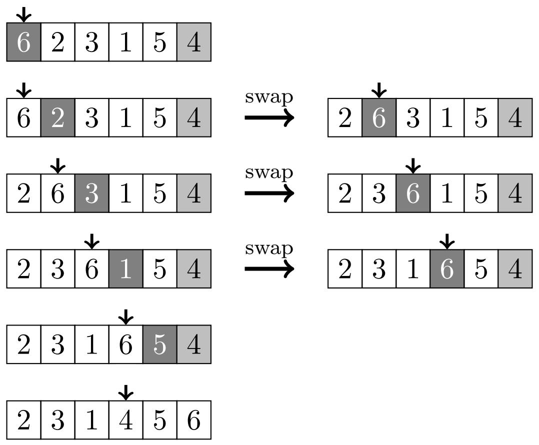

Figure 3: Example of quicksort partitioning

## 2.4 Heapsort

Heapsort works by creating a heap representation of the list of elements, then sorting the elements in the heap using hierarchical operations. When building the heap representation, the first element of the list is used as the root. Then, for element with index $i$, the element with index $2i + 1$ is considered its left child, while the element with index $2i + 2$ is considered the right child. This is shown in figure 4.

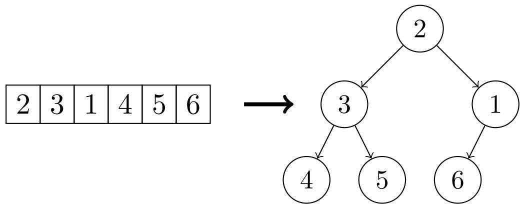

Figure 4: Representation of the element list as a heap

Heapsort uses the *heapify* operation, which recursively builds a max-heap from the existing list of elements. A max-heap is a heap in which, for any given node that is not the root, the key of the parent is greater than the node's key. Figure 5 shows heapsort's first heapification, which results in a max-heap. Starting from the middle element of the list, and going back one by one, elements are chosen as reference to run heapify. If any of the reference element's children has a larger key, they are swapped. If both children have greater keys, the right child is preferred. If a swap happens, as explained earlier, heapify is run again with the same element as reference in its new position.

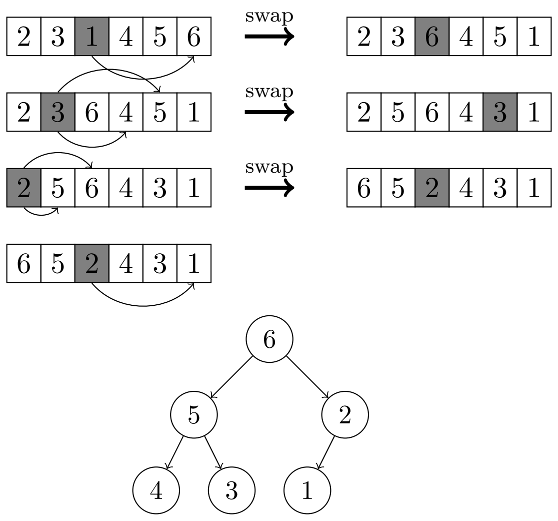

Figure 5: Example of the use of the heapify function on heapsort to create a max-heap

Finally, after the max-heap generation, the actual sorting is done by choosing a reference element, starting from the last one and all the way until the first one. This reference element is replaced with the first element, and then the subset of elements from the first to the original index of the reference element is heapified.

## 2.5 Timsort

Timsort divides the element list in *runs* or sublists of predefined sizes. First, these runs are locally ordered using insertion sort. Afterwards, adjacent run couples are merged using the merge operation from the merge sort. Then, the size of the run is doubled, the merging process repeated, and this procedure is repeated until the size of the run is greater than the size of the element list.

### 2.6 Introsort

Introsort is a recursive method that uses a depth limit to avoid the overhead of running deep into recursion. This depth limit is calculated using the length of the element list, with the formula $2 \cdot \log_2 |L|$, $|L|$ being the size of the list. The algorithm takes into account three conditions. If the passed element list is smaller than 16, which is a hard-coded value, insertion sort is used. If the depth limit has been reached, the passed list is sorted with heapsort.

If any of these conditions are the case, a pivot must be chosen, between the first element, the last, and the one in the middle. The pivot must be the one with the intermediate value. For example, if the first element is the larger than both other elements, and the last element is the smallest, the middle element must be the pivot. This pivot must be used to partition the list using quicksort's partition method as explained earlier. Then, these partitions are passed to a recursive call of introsort, with the depth limit decremented by one.

## 3 Implementation

These algorithms have been implemented using Python 3.6. All the implementations have been done within a class named `Sorter`. The `main` method of the provided script will serve as an example on how to use the Python script. It can also be directly used by means of its flags. Running `python3 main.py -h` will show all available options.

## 4 Complexity analysis and experiment design

Table 1 shows the different computational complexities of the different sorting algorithms used for the development of this homework. This table makes the relation between the complexities of the hybrid algorithms, and the algorithms they use, more obvious.

On the one hand, see timsort's best case scenario. Timsort, as explained earlier, sorts the runs using the insertion sort algorithm. If a run is already in order, the insertion sort would run in $O(n)$ time, which corresponds to its best case scenario. Then, the whole list is merged, run by run, and doubling the run size for each merge loop. But, if the size of the list is smaller than the

| | worst case | best case | average |
|---|---|---|---|
| insertion sort | $O(n^2)$ | $O(n)$ | $O(n^2)$ |
| merge sort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| timsort | $O(n \log n)$ | $O(n)$ | $O(n \log n)$ |
| quicksort | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ |
| heapsort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| introsort | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |

Table 1: Computational complexity of the different sorting algorithms

defined run size, the merge loop will never be run in the first place, therefore the algorithm will run in $O(n)$ time. This is practically useless, as the run sizes tend to be small, and to be able to run timsort on its best case scenario, the list should be small or the run size large enough. However, this would also mean just running a regular insertion sort, so there is no point in using timsort in these cases.

The point of timsort lays in how it is supposed to palliate the high average cost of insertion sort, whilst taking profit of it to make merge sort faster. So, although the average complexity stays the same, **timsort should work faster than merge sort**. This is one of the points to prove with the experiments.

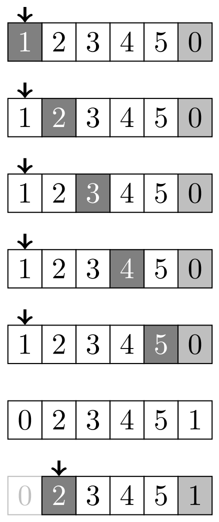

Figure 6: Example of quicksort on its worst case

On the other hand, see quicksort's worst case scenario. This worst case scenario depends on the pivot. In the implementation done for this homework, as mentioned, the last element of the list is used as the pivot. This means that, if the smallest element is both the pivot and the last element, the partition will result in a list of size $n-1$, where $n$ was the size of the original list. If the pivot of the new list is, again, the smallest element, and this keeps recursively happening until the length of the list is 1, basically an insertion sort on its worst case is happening, where all elements were sitting in the opposite of their correct position. This is represented in figure 6. This can happen regardless of how the pivot is selected.

Introsort makes use of the same partition method as quicksort to divide the element list, but also introduces two concepts to avoid the worst case scenario of quicksort, by running insertion sort or heapsort if given conditions are met. This makes the algorithm run in an average time of $O(n \log n)$. However, despite both heapsort and introsort share time complexities, **introsort should run faster than heapsort**, which is used as part of introsort itself, otherwise it would not have a point. Some experimentation will also be performed regarding the optimal size of the run.

Thus, the experiments will run using randomly created arrays of non-repeating numbers, and a different range of sizes, to measure the average times, as well as using specifically crafted arrays of numbers that will cause the mentioned best case and worst case scenarios to happen.

## 5 Results

First off, the graph on figure 7 shows the evolution of execution time of Timsort for different run sizes and element amounts. Many of the run sizes overlap in the graph, offering very similar performance, so it is difficult to tell which run size is optimal. Also, it stands out that all run sizes suffer from sudden performance losses in intervals. The biggest of these leaps could be the one that the 75 run size suffers. According to the gathered data, the difference between the execution times with a list of 39000 and 38000 elements is of almost 4 seconds, when, up to that point, the same difference in the element amounts would be translated into around 0.3 seconds of execution time. However, considering the trend of the graph, it can be inferred that the remaining run sizes will suffer of the same leaps soon afterwards. So, for the comparison with the rest of the algorithms, the 75 run size-based Timsort will be used.

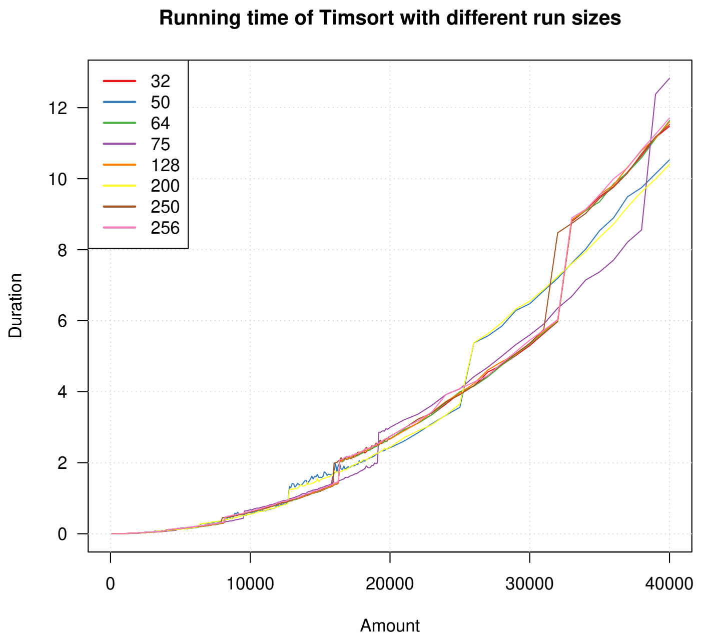

Figure 7: Graph of relation between execution time and run size for Timsort

Figure 8 shows the graph comparing the execution times of the different algorithms, including the mentioned 75 run size-based Timsort. As table 1 predicted, the insertion sort is, by far, the slowest of algorithms. And, surprisingly, Timsort is the overall second slowest algorithm, tying with merge sort at times. The fastest algorithms are, with no doubt, Quicksort and Introsort, the latter begin slightly better. The graph in figure 8 is limited to lists of 60000 elements and this offers a limited view of the performance gains of Introsort. It takes much bigger figures to really be able to notice the difference in performance of these algorithms. Even though no graph was generated to display this information, Quicksort runs in an average of 3.52 seconds with 1 million elements and 52.02 seconds with 10 million elements, whereas Introsort runs in an average of 2.71 and 41.95 seconds with the same respective element amounts. So, definitely Introsort is much faster than any other algorithm, including heapsort.

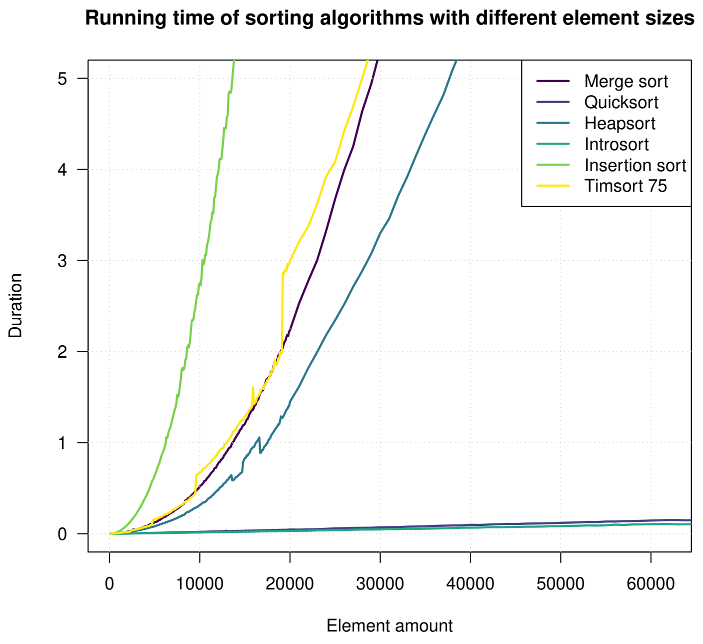

Figure 8: Comparison of the evolution of execution time depending on the element amount

## 6 Conclusions

The graph in figure 7 already proved the clear instability of the Timsort algorithm or, more likely, the implementation, while the graph in figure 8 has shown it can not keep up with other sorting algorithms. After some code reviewing, it was found that these huge performance leaps happen due to a specific reason. When merging the runs in Timsort, the run size is doubled until the list size is reached, which causes the whole element list to merge. If starting with a run size of 75, and after doubling it exactly 9 times, the run size of 38400 is obtained. With an element list of 38000, this is enough to stop iterating. However, with a element list of 39000, another iteration will be run, causing the merge of 39000 elements. This is the reason for the performance leaps shown in figure 7, which are caused every time a new iteration of the merge operation is required. The merge operation should be fast, though, as it is run in linear time, which could mean the implementation of the merge operation is not efficient enough. Nevertheless, merge sort does seem to work properly.

The definitive conclusion of this experimentation is that both Quicksort and Introsort are extremely fast and do not seem to showcase any corner case or worst case scenario, whereas Timsort, as just explained, does show strange, irregular behaviour every now and then. In addition to this, the varying run sizes do not seem to have a noteworthy effect. Using small runs would translate into irregular performance, because the iteration problem discussed earlier would happen more frequently. Using bigger runs to avoid this problem, on the other hand, translates into using the slow insertion sort with bigger lists, which would decrease performance.