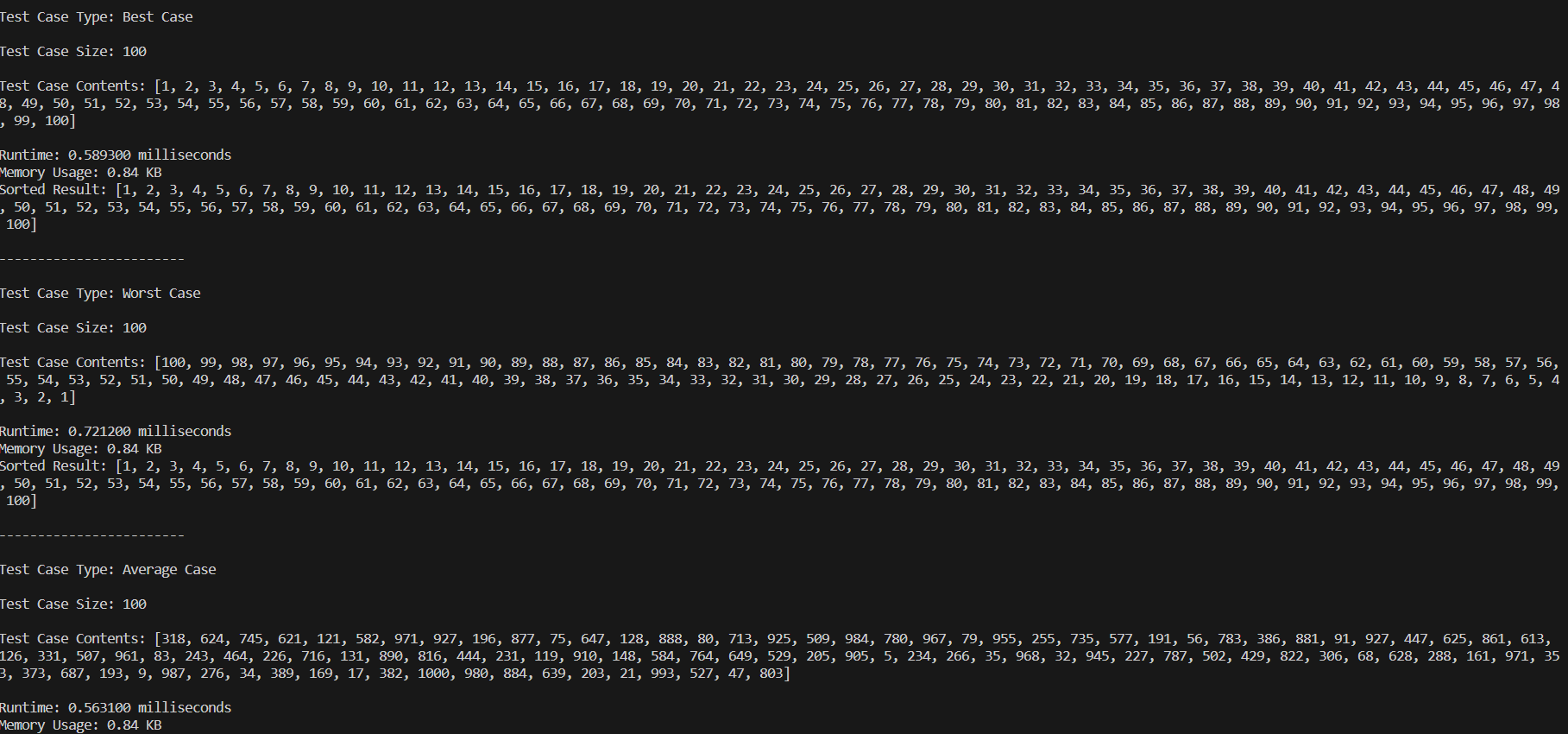
2)

i) Improvement of Quick sort implementation

<https://www.techiedelight.com/hybrid-quicksort/>

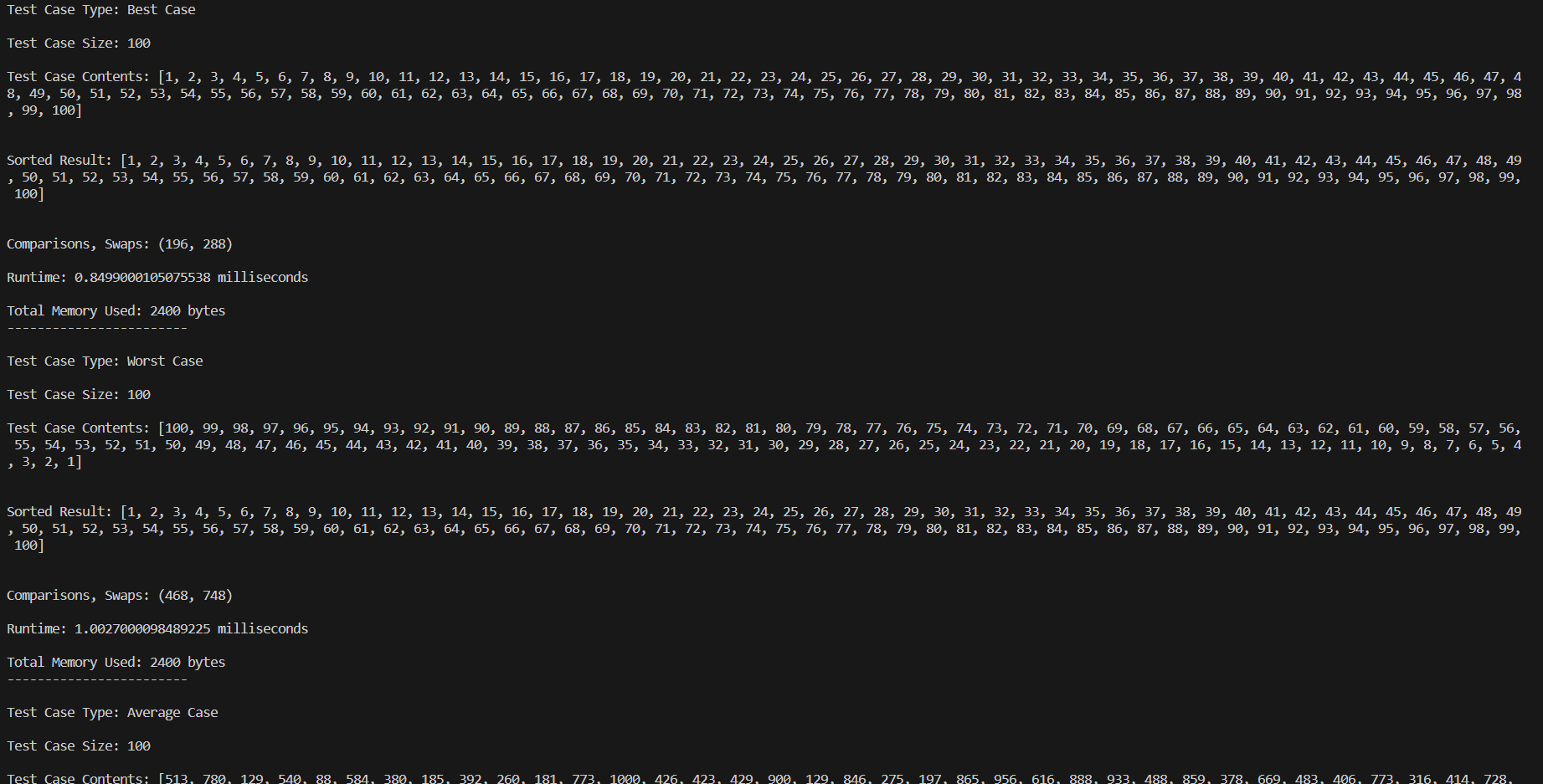


When the total number of elements is below some threshold (perhaps ten elements), switch to a non-recursive sorting algorithm such as insertion sort that performs fewer swaps, comparisons, or other operations on such small arrays.

Instead of “many small sorts” optimization, we can stop when the total number of elements is less than some threshold k. Later, when the whole array has been processed, each element will be at most k positions away from its final sorted position. Now, if we perform insertion sort on it, it will take O(k.n) time to finish the sort, which is linear as k is a constant.

* Randomized Pivot Selection**:** One of the most common improvements to Quick Sort is to use a randomized pivot selection strategy. Instead of always choosing the first or last element as the pivot, you randomly select a pivot from within the subarray. This helps avoid worst-case scenarios and ensures better average-case performance.
* Median Pivot**:** This improvement involves selecting the pivot as the median of three elements (e.g., the first, middle, and last elements of the subarray). This helps mitigate issues with extreme values and contributes to more balanced partitions.
* Hybrid Sorting Algorithms: In practice, Quick Sort is often combined with other sorting algorithms, such as Insertion Sort or Heap Sort, to improve performance for small subarrays.
* Tail Recursion Elimination: The function's return value is directly derived from the recursive call without any further computation or processing. Tail Recursion Elimination works by reusing the current function's stack frame (activation record) for the next function call, rather than creating a new stack frame for each recursive call. This optimization reduces the overhead associated with function calls and stack frame creation, ultimately making the code more memory-efficient and potentially faster.

ii) Improvement of Merge sort implementation



* In-Place Merge Sort**:** It is a variation of the traditional Merge Sort algorithm that sorts an array without using additional memory, except for a small, constant amount of auxiliary memory. In the standard Merge Sort, a separate array is typically used for merging, while in the in-place version, the merging is done directly within the original array. These variants optimize the memory usage aspect.
* Parallel Merge Sort: In modern computing environments, parallelization is crucial for performance. Parallel versions of Merge Sort distribute the sorting work among multiple processors or cores to speed up the process.
* Optimizations for Small Arrays: Merge Sort can be optimized for small subarrays by switching to a more efficient sorting algorithm, such as Insertion Sort, when the size of the subarray falls below a certain level.

Merge Sort provides a guaranteed time complexity but is not adaptive. Tim Sort is adaptive and efficient, making it a practical choice for sorting real-world data, especially when combined with small data sets and nearly sorted data. Both algorithms are stable and have similar space complexities.

Improvement of Quick sort implementation

* Randomized Pivot Selection**:** One of the most common improvements to Quick Sort is to use a randomized pivot selection strategy. Instead of always choosing the first or last element as the pivot, you randomly select a pivot from within the subarray. This helps avoid worst-case scenarios and ensures better average-case performance.
* Median Pivot**:** This improvement involves selecting the pivot as the median of three elements (e.g., the first, middle, and last elements of the subarray). This helps mitigate issues with extreme values and contributes to more balanced partitions.

Placing the median value at the end of the array improves the pivot selection strategy, reduces the likelihood of worst-case scenarios, and simplifies the implementation by making swap operations more convenient during the partition step.

* Hybrid Sorting Algorithms: In practice, Quick Sort is often combined with other sorting algorithms, such as Insertion Sort or Heap Sort, to improve performance for small subarrays.

When the size of the subarray is below a certain threshold, the algorithm switches to Insertion Sort

* Tail Recursion Elimination: The function's return value is directly derived from the recursive call without any further computation or processing. Tail Recursion Elimination works by reusing the current function's stack frame (activation record) for the next function call, rather than creating a new stack frame for each recursive call. This optimization reduces the overhead associated with function calls and stack frame creation, ultimately making the code more memory-efficient and potentially faster.

Time complexity:

* The time complexity for the recursive part is *O*(*n*log*n*) in the best and average cases.
* The **partition** function has a time complexity of *O*(*n*).
* The **insertion\_sort** function has a time complexity of *O*(*k*), where *k* is the sorted subarray size.
* Considering all parts, the overall time complexity is **O(n log n)** in the best and average cases, and **O(n^2)** in the worst case

\*\*The improvements made can lead to a reduction in the number of recursive calls and comparisons, improving the average performance of the algorithm.

Improvement of Merge sort implementation

* In-Place Merge Sort**:** It is a variation of the traditional Merge Sort algorithm that sorts an array without using additional memory, except for a small, constant amount of auxiliary memory. In the standard Merge Sort, a separate array is typically used for merging, while in the in-place version, the merging is done directly within the original array. These variants optimize the memory usage aspect.
* Parallel Merge Sort: In modern computing environments, parallelization is crucial for performance. Parallel versions of Merge Sort distribute the sorting work among multiple processors or cores to speed up the process.
* Optimizations for Small Arrays: Merge Sort can be optimized for small subarrays by switching to a more efficient sorting algorithm, such as Insertion Sort, when the size of the subarray falls below a certain level. The program uses Insertion Sort for small subarrays (size <= threshold). The worst-case time complexity of Insertion Sort is *O*(*n^*2), but for small subarrays, it can be more efficient than the O(*n*log*n*) Merge Sort.

Time complexity:  
The time complexity of the Merge Sort algorithm is O(*n*log*n*) in all cases (best, worst, and average). This is because the array is recursively divided into halves, and then merged back together, with each merge operation taking *O*(*n*) time.