Report Task A

1 Introduction

This report aims to conduct a comprehensive analysis and comparison of the performance of three prominent sorting algorithms, namely Insertion sort, Quick sort, and Merge sort. By identifying the breakpoints of each algorithm, two hybrid sorting algorithms will be created to combine the optimal features of each algorithm and achieve superior sorting efficiency. The report will discuss the approach employed to implement the hybrid sorting algorithms, and the results obtained from testing them will be presented, along with the analyzed data. Through an extensive analysis of the performance of each algorithm, the strengths and limitations of each approach will be identified, thus providing a roadmap for the development of the hybrid algorithms.

2 Background

Sorting has been a subject of research in computer science since the early days of computing. Early computers were limited in their processing power and memory, and sorting large datasets was a challenging task. As computers became more powerful, new sorting algorithms were developed that could handle larger datasets more efficiently. Over the years, a wide variety of sorting algorithms have been proposed, ranging from simple algorithms like Bubble sort and Insertion sort to more complex ones like Quick sort and Merge sort. These algorithms differ in terms of their approach to sorting, the time and memory complexity, and the suitability for different types of data. Sorting algorithms also have practical applications beyond computer science. For example, sorting algorithms are used in everyday life for tasks such as sorting mail or organizing a list of items. Sorting algorithms also have applications in fields like biology, physics, and economics, where data must be sorted and analyzed to identify patterns and relationships. In recent years, research has focused on developing new sorting algorithms that can handle increasingly large datasets efficiently. Parallel sorting algorithms that use multiple processors to sort data simultaneously have been developed, as well as algorithms that can sort data in external storage devices like hard drives. In summary, sorting is a fundamental operation in computer science and beyond, and the development of efficient sorting algorithms has been a subject of research for decades. A deep understanding of sorting algorithms and their trade-offs is crucial for selecting the appropriate

algorithm for a given task and for developing new algorithms that can handle large datasets efficiently.

2.1 Place of insertion

Insertion sort is a simple sorting algorithm that iterates through an array and places each element in its correct position relative to the other elements. The algorithm works by sorting the array in place, with the sorted portion of the array growing with each iteration. The algorithm is named "insertion sort" because each element is inserted into its correct position in the sorted portion of the array.

The insertion sort algorithm works as follows:

1. Iterate through the unsorted portion of the array from the second element to the last element.
2. For each element, compare it with the elements in the sorted portion of the array from right to left.
3. If the element is smaller than the current element, shift the current element to the right.
4. Repeat step 3 until the correct position for the element is found.
5. Insert the element into its correct position in the sorted portion of the array.
6. Repeat steps 1-5 until the entire array is sorted.

The time complexity of the insertion sort algorithm is O(n^2) in the worst case and O(n) in the best case, where n is the number of elements in the array. The worst-case time complexity occurs when the array is sorted in reverse order, and the best-case time complexity occurs when the array is already sorted.

Pros:

1. *Insertion sort is simple and easy to implement.*
2. *Insertion sort is efficient for small arrays or nearly sorted arrays.*
3. *Insertion sort is an adaptive algorithm, meaning that its performance adapts to the input data.*
4. *Insertion sort is stable, meaning that it preserves the relative order of equal elements.*

Cons:

1. *Insertion sort is not efficient for large arrays due to its quadratic time complexity.*

2. *Insertion sort is not efficient for unsorted or randomly ordered data.*
3. *Insertion sort is an in-place algorithm but requires shifting elements to create space for the new element.*
4. *Insertion sort does not parallelize well, limiting its scalability for large datasets.*

2.2 Quick sort

Quick sort is a widely used sorting algorithm that uses a divide-and-conquer strategy to sort an array of elements. The algorithm selects a pivot element from the array, partitions the array into two sub-arrays such that all elements in one sub-array are less than or equal to the pivot element, and all elements in the other sub-array are greater than the pivot element. This process is then recursively applied to the two sub-arrays until the entire array is sorted.

The quick sort algorithm works as follows:

1. Select a pivot element from the array.
2. Partition the array into two sub-arrays: one sub-array containing elements less than or equal to the pivot element, and the other sub-array containing elements greater than the pivot element.
3. Recursively apply steps 1 and 2 to the sub-arrays until each sub-array contains only one element, which means the array is sorted.
4. Combine the sub-arrays to obtain the final sorted array.

Pros:

1. *Uses a divide-and-conquer strategy to sort an array of elements.*
2. *Selects a pivot element from the array and partitions it into two sub-arrays.*
3. *Has a high average case performance of O(n*log n)*
4. *Is an in-place sorting algorithm that does not require additional memory space.*

Cons:

1. *Worst-case time complexity of O(n^2) when the pivot element selected is either the smallest or largest element in the array.*
2. *Inefficient performance on already sorted or nearly sorted arrays.*
3. *Not a stable sorting algorithm, meaning it does not preserve the relative order of elements with equal keys.*

4. *Selection of the pivot element can significantly impact the efficiency of the algorithm.*

2.3 Merge sort

Merge sort is a popular sorting algorithm that uses the divide-and-conquer approach to sort an array of elements. The algorithm works by recursively dividing the array into two halves until each half has only one element. It then merges the two halves together in sorted order to produce the final sorted array.

Here are the steps of the merge sort algorithm:

1. Divide the array into two halves, if the array has more than one element.
2. Recursively apply merge sort to each half.
3. Merge the two sorted halves into a single sorted array.

Pros:

1. *Consistent and predictable **O(n log n)** time complexity, regardless of input data*
2. *Stable sorting algorithm that maintains the relative order of equal elements*
3. *Good choice for sorting large amounts of data.*

Cons:

1. *Requires additional memory to store the two sub-arrays during the merge step.*
2. *High space complexity of O(n) due to the need to create additional arrays during the sorting process.*

3 Method and implementation choice

Quick sort

The given code implements the QuickSort algorithm to sort a list of numbers. QuickSort is a divide-and-conquer algorithm that partitions the list into two sub-lists based on a chosen pivot element. The left sub-list contains elements smaller than the pivot, while the right sub-list contains elements larger than the pivot. This process is repeated recursively until the sub-lists contain only one or zero elements. The partition function is used to select a pivot element and partition the list around that pivot. It takes the lower and upper bounds of the sublist to partition and the list itself as input and returns the index of the pivot element after partitioning. The partition function works by selecting the last element of the sublist as the pivot and

then comparing each element in the sublist to the pivot. If an element is smaller than or equal to the pivot, it is swapped with the element at the current index, and the index is incremented. After all elements have been compared to the pivot, the pivot is swapped with the element at the current index, and the index is returned.
The qsort function is used to sort the list recursively using the partition function. It takes the lower and upper bounds of the sublist to sort and the list itself as input and sorts the sublist in place. It does this by calling the partition function to select a pivot element and partition the list around it. It then recursively calls itself on the left and right sub-lists created by the partition until the sub-lists contain only one or zero elements. The quicksort function is used to sort a given list of numbers in ascending order using the quicksort algorithm. It takes the list itself as input and sorts the list in place by calling the qsort function with the lower and upper bounds of the list.
The given code chooses the last element of the sublist as the pivot element for partitioning. This is a simple and commonly used method for selecting pivot elements, but it can lead to inefficient performance in some cases. An alternative method is to choose a random element as the pivot, which can reduce the chance of worst-case behavior. The implementation of quicksort in the given code uses a recursive approach to sort the list. This approach has a space complexity of O(log n) due to the call stack used to store recursive function calls. An alternative approach is to use an iterative implementation that uses a stack or queue to keep track of the sub-lists that need to be sorted. This approach has a space complexity of O(log n) as well, but it can reduce the overhead of function calls and potentially improve performance.

Insertion sort

The insertion sort algorithm was implemented using a simple iterative approach, whereby the list was iterated through, and each element was inserted into its correct position in a growing sorted sequence. The algorithm began with the second element in the list and compared it to the elements before it, swapping any that were out of order. This process was repeated for each subsequent element until the entire list was sorted. The decision to use this approach was informed by the fact that insertion sort is generally considered to be an efficient algorithm for small datasets and partially sorted datasets. Furthermore, the simplicity of its implementation made it easy to understand and implement. While insertion sort has a worst-case time

complexity of O(n^2), which can be inefficient for large datasets, it can be faster than more complex sorting algorithms such as quicksort or mergesort for small datasets or partially sorted datasets. In summary, the choice to use the insertion sort algorithm was based on its simplicity and efficiency for certain scenarios, particularly small datasets and partially sorted datasets.

Merge sort

For the implementation of the mergesort algorithm, a recursive divide-and-conquer approach was chosen. The 'mergesort' function first checks if the length of the input list is greater than one. If so, it finds the middle index of the list using integer division and divides the list into two sub-lists, 'left' and 'right'. It then recursively calls the 'mergesort' function on each sub-list until the base case is reached, which is when the sub-lists have only one element. To merge the sub-lists, the 'merge' function is called, which compares the elements at the current indices of each sub-list and appends the smaller one to a new list called 'result'. If one of the sub-lists is fully consumed, the remaining elements from the other sub-list are appended to 'result'. The resulting sorted list is then returned. Finally, after the 'merge' function returns the sorted list, the 'mergesort' function assigns the sorted list back to the input list using slice assignment, allowing the function to sort the original list rather than creating a new one. The time complexity of the mergesort algorithm is O(n log n) in the worst case, where 'n' is the number of elements in the input list. This makes it a more efficient sorting algorithm for larger datasets compared to algorithms with a worst-case time complexity of O(n^2), such as insertion sort. However, mergesort requires additional memory to store the sub-lists, which may be a disadvantage for very large datasets or constrained memory environments.

4 Results and Analysis

4.1 Sorteringsalgoritmer

Table 1 Description you need to change. For the files of size 10

Algorithm	**Running time** $(s \mid ms \mid \mu s)$

	1	2	3	4	5	Average value
Insertionsort	**0.0000070 s**	**0.0000053 s**	**0.0000050 s**	**0.0000049 s**	**0.0000050s**	**0.0000054 s**
Quicksort	**0.0000211 s**	**0.0000146 s**	**0.0000138 s**	**0.0000140 s**	**0.0000139 s**	**0.0000155 s**
Mergesort	**0.0000231 s**	**0.0000157 s**	**0.0000148 s**	**0.0000149 s**	**0.0000151 s**	**0.0000167 s**

Table 2 Description you need to change. For the files of size 100

Algorithm	**Running time $(s \mid ms \mid \mu s)$**					
	1	**2**	**3**	**4**	**5**	**Average value**
Insertionsort	**0.0001690 s**	**0.0001693 s**	**0.0001612 s**	**0.0001620 s**	**0.0001648 s**	**0.0001653 s**
Quicksort	**0.0001149 s**	**0.0001032 s**	**0.0001002 s**	**0.0000976 s**	**0.0000970 s**	**0.0001026 s**
Mergesort	**0.0001401 s**	**0.0001294 s**	**0.0001304 s**	**0.0001283 s**	**0.0001331 s**	**0.0001323 s**

Table 3 Description you need to change. For the files of size 1000

Algorithm	**Running time $(s \mid ms \mid \mu s)$**					
	1	**2**	**3**	**4**	**5**	**Average value**
Insertionsort	**0.0171299 s**	**0.0204198 s**	**0.0298614 s**	**0.0293128 s**	**0.0297834 s**	**0.0253015 s**
Quicksort	**0.0022838 s**	**0.0021754 s**	**0.0022481 s**	**0.0021807 s**	**0.0021482 s**	**0.0022072 s**
Mergesort	**0.0018747 s**	**0.0016116 s**	**0.0015842 s**	**0.0016108 s**	**0.0015127 s**	**0.0016388 s**

Table 4 Description you need to change. For the files of size 10 000

Algorithm	**Running time $(s \mid ms \mid \mu s)$**

	1	2	3	4	5	Average value
Insertionsort	**2.83477 40 s**	**2.83583 54 s**	**2.644458 1 s**	**2.78687 52 s**	**2.71953 59 s**	**2.7642957 s**
Quicksort	**0.01728 09 s**	**0.02744 46 s**	**0.015347 8 s**	**0.02572 14 s**	**0.02865 65 s**	**0.0228902 s**
Mergesort	**0.03410 70 s**	**0.03870 21 s**	**0.036522 3 s**	**0.03582 90 s**	**0.03502 62 s**	**0.0360373 s**

Table 5 Description you need to change. For the files of size 100 000

Algorithm	**Running time** $(s \mid ms \mid \mu s)$					
	1	**2**	**3**	**4**	**5**	**Average value**
Insertionsort	**275.703 7932 s**	**272.222 0515 s**	**272.7638 486 s**	**274.264 6918 s**	**275.626 8510 s**	**274.1162472 s**
Quicksort	**0.24415 62 s**	**0.27476 32 s**	**0.323407 5 s**	**0.30759 13 s**	**0.31930 92 s**	**0.2938455 s**
Mergesort	**0.34778 62 s**	**0.42169 59 s**	**0.426118 1 s**	**0.39271 57 s**	**0.42719 34 s**	**0.4031019 s**

Table 6 Table description for displaying each c-constant and the mean value of the c-constants, i.e. c^{-}

Algorithm	**c constants**					
	$C10$	$C100$	$C1000$	$C10000$	$C100000$	c^{-}
Insertionsort	**5.4 x10^{-8}**	**1.653x10^{-8}**	**2.53015 x 10^{-8}**	**2.764295 7 x 10^{-8}**	**2.741162 472 x 10^{-8}**	**3.01748 x 10^{-8}**
Quicksort	**1.55 x 10^{-6}**	**5.13 x 10^{-7}**	**7.357 x 10^{-7}**	**5.72255 x 10^{-7}**	**5.8769 x 10^{-7}**	**7.9321 x 10^{-7}**
Mergesort	**1.67 x 10^{-6}**	**6.615 x 10^{-7}**	**5.46266 7x 10^{-7}**	**9.0093 x 10^{-7}**	**8.06203 8 x 10^{-7}**	**9.63121 x 10^{-7}**

4.2 Identification of breakpoints

Mergesort

We can set the time complexities of Insertionsort and Mergesort equal to each other and solve for the value of n at which they intersect:

c1 * n^2 = c2 * n * log(n**)**

We can substitute in the given values of c1 and c2:

$3.01748 \times 10^{-8} * n^2 = 9.63121 \times 10^{-7} * n * \log(n)$

We get n is 56

Quicksort

We can set the time complexities of Insertionsort and quicksort equal to each other and solve for the value of n at which they intersect:

c1 * n^2 = c2 * n * log(n**)**

We can substitute in the given values of c1 and c2:

$3.01748 \times 10^{-8} * n^2 = 7.9321 \times 10^{-7} * n * \log(n)$

We get n is 43

4.3 Hybrid sorting algorithms

Table 7 Table description

File size	Algorithm	Running time $(s \,/\, ms \,/\, \mu s)$					
		1	2	3	4	5	Average value
10	HybridQ	0.0000034 s	0.0000029 s	0.0000028 s	0.0000028 s	0.0000026 s	0.0000029 s
	HybridM	0.0000062 s	0.0000036 s	0.0000032 s	0.0000030 s	0.0000030 s	0.0000038 s
100	HybridQ	0.0000774 s	0.0000716 s	0.0000687 s	0.0000687 s	0.0000675 s	0.0000708 s
	HybridM	0.0001180 s	0.0001146 s	0.0001137 s	0.0001144 s	0.0001134 s	0.0001148 s
1000	HybridQ	0.0010013 s	0.0009910 s	0.0020226 s	0.0023322 s	0.0020520 s	0.0016798 s

	HybridM	**0.0012 862 s**	**0.0012 015 s**	**0.00112 73 s**	**0.0016 838 s**	**0.0020 904 s**	**0.0014778 s**
10000	**HybridQ**	**0.0211 238 s**	**0.0248 741 s**	**0.02506 16 s**	**0.0258 122 s**	**0.0264 662 s**	**0.0246676 s**
	HybridM	**0.0305 747 s**	**0.0322 628 s**	**0.01767 78 s**	**0.0299 838 s**	**0.0288 875 s**	**0.0278773 s**
100000	**HybridQ**	**0.2546 458 s**	**0.2884 070 s**	**0.26296 01 s**	**0.2730 246 s**	**0.2560 411 s**	**0.2670157 s**
	HybridM	**0.3821 412 s**	**0.3931 929 s**	**0.38336 19 s**	**0.3859 029 s**	**0.3670 688 s**	**0.3823335 s**

5 Conclusions

This hybrid approach of using insertionsort for small input sizes and either quicksort or mergesort for larger input sizes proved to be effective in improving the overall efficiency of sorting algorithms. The specific break points chosen for the hybrid implementations of mergesort and quicksort were based on empirical testing and analysis.

To identify the breakpoints for n where insertionsort starts to become slower than quicksort and mergesort, we need to mathematically find the point of intersection between the time complexity functions of insertionsort and quicksort/mergesort. The intersection point represents the input size at which the time complexity of insertionsort becomes equal to that of quicksort or mergesort. We can solve the equation of the time complexity functions for the intersection point using algebraic methods, and then verify the result empirically by measuring the actual running times of the algorithms for different input sizes around the intersection point.

While the hybrid approach of using insertionsort for small input sizes and either quicksort or mergesort for larger input sizes can improve the overall efficiency of sorting algorithms, there are situations where it may not perform well.

One possible reason is that the chosen breakpoints may not be optimal for the particular input data distribution. For example, if the input data is mostly sorted or

partially sorted, insertionsort may be more efficient than quicksort or mergesort for larger input sizes, which means that the breakpoint should be set to a larger value than the usual threshold. On the other hand, if the input data is highly unstructured, the threshold for the breakpoint may need to be set lower to ensure better performance.

Another reason is that the overhead of switching between different algorithms can affect the overall efficiency of the hybrid approach. When the input size is close to the threshold value, the overhead of switching to a different algorithm can outweigh the benefits of using that algorithm for the particular input size. Therefore, careful consideration should be given to the choice of threshold values to minimize the overhead of switching.

Overall, the hybrid approach of using insertionsort and either quicksort or mergesort can be an effective strategy for improving the efficiency of sorting algorithms, but the choice of breakpoints and the overhead of switching between algorithms should be carefully considered to ensure optimal performance.