Sorting: Quicksort and Merge sort

Zaki Tahmeed, H.M.Mutammil Galib and Faiyaz Rashid Nabil

Bangladesh University of Engineering and Technology

July 24, 2021

Table of Contents

Sorting

Sorting is any process of arranging items systematically and in a sequence ordered by some criterion.The most common uses of sorting are:

Sorting

Sorting is any process of arranging items systematically and in a sequence ordered by some criterion.The most common uses of sorting are:

- making lookup or search efficient.

Sorting

Sorting is any process of arranging items systematically and in a sequence ordered by some criterion.The most common uses of sorting are:

- making lookup or search efficient.
- making merging of sequences efficient.

Sorting

Sorting is any process of arranging items systematically and in a sequence ordered by some criterion.The most common uses of sorting are:

- making lookup or search efficient.
- making merging of sequences efficient.
- enable processing of data in a defined order.

Table of Contents

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort
- Selection sort

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort
- Selection sort
- Heapsort

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort
- Selection sort
- Heapsort
- Merge Sort

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort
- Selection sort
- Heapsort
- Merge Sort
- Quicksort

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms that require input data to be in sorted lists.Some popular sorting algorithms are:

- Insertion Sort
- Selection sort
- Heapsort
- Merge Sort
- Quicksort
- Bubble sort

Table of Contents

Divide and Conquer

Divide and conquer is an algorithm design paradigm.The concept of Divide and Conquer involves three steps:

Divide and Conquer

Divide and conquer is an algorithm design paradigm.The concept of Divide and Conquer involves three steps:

- Divide the problem into multiple small problems.

Divide and Conquer

Divide and conquer is an algorithm design paradigm.The concept of Divide and Conquer involves three steps:

- Divide the problem into multiple small problems.
- Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.

Divide and Conquer

Divide and conquer is an algorithm design paradigm.The concept of Divide and Conquer involves three steps:

- Divide the problem into multiple small problems.
- Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
- Combine the solutions of the subproblems to find the solution of the actual problem.

Divide and Conquer

Divide and conquer is an algorithm design paradigm.The concept of Divide and Conquer involves three steps:

- Divide the problem into multiple small problems.
- Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
- Combine the solutions of the subproblems to find the solution of the actual problem.

Examples

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFT).

Table of Contents

Quicksort

Introduction

Quicksort

Introduction

- Quicksort is a divide-and-conquer algorithm.

Quicksort

Introduction

- Quicksort is a divide-and-conquer algorithm.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

Quicksort

Introduction

- Quicksort is a divide-and-conquer algorithm.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- It was Developed by British computer scientist Tony Hoare in 1959 and published in 1961.

Quicksort

Introduction

- Quicksort is a divide-and-conquer algorithm.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- It was Developed by British computer scientist Tony Hoare in 1959 and published in 1961.
- It is an in-place sorting algorithm, requiring small additional amounts of memory to perform the sorting.

Quicksort

Introduction

- Quicksort is a divide-and-conquer algorithm.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- It was Developed by British computer scientist Tony Hoare in 1959 and published in 1961.
- It is an in-place sorting algorithm, requiring small additional amounts of memory to perform the sorting.
- When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.

Quicksort

Procedure

Quicksort

Procedure

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.

Quicksort

Procedure

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays.

Quicksort

Procedure

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays.
- In the first sub-array, all elements are less than or equal to the pivot value.

Quicksort

Procedure

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays.
- In the first sub-array, all elements are less than or equal to the pivot value.
- In the second sub-array, all elements are greater than or equal to the pivot value.

Quicksort

Procedure

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays.
- In the first sub-array, all elements are less than or equal to the pivot value.
- In the second sub-array, all elements are greater than or equal to the pivot value.
- The sub-arrays are then sorted recursively.

Quicksort

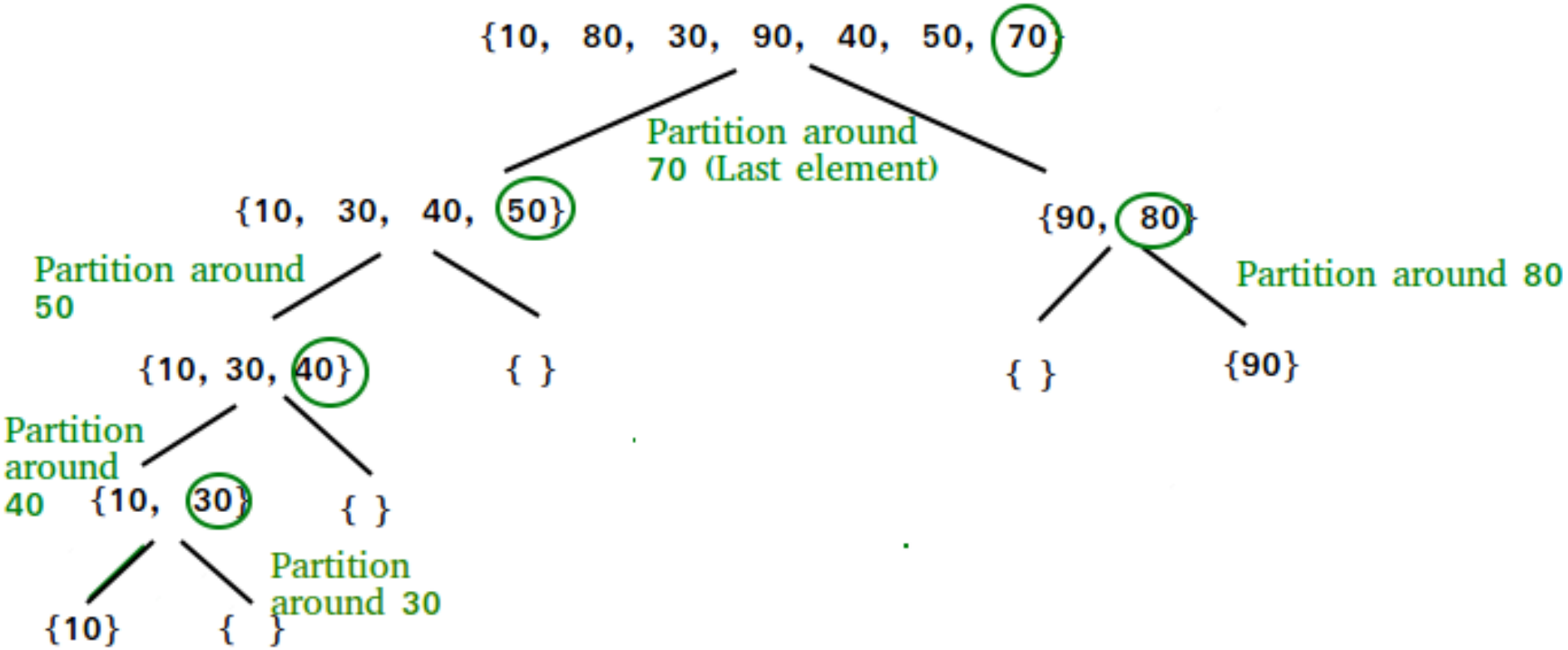

Figure: Quicksort Algorithm

Time Complexity

Best-case Analysis

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size.

Time Complexity

Best-case Analysis

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size.

Consequently, we can make only $log_2 n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $log_2 n$. But no two calls at the same level of the call tree process the same part of the original list.

Time Complexity

Best-case Analysis

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size.

Consequently, we can make only $log_2 n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $log_2 n$. But no two calls at the same level of the call tree process the same part of the original list.

Thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(nlogn)$ time.

Time Complexity

Worst-case Analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n - 1$.This may occur if the pivot happens to be the smallest or largest element in the list.

Time Complexity

Worst-case Analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n-1$.This may occur if the pivot happens to be the smallest or largest element in the list.
If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n-1$ nested calls before we reach a list of size 1.

Time Complexity

Worst-case Analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n-1$.This may occur if the pivot happens to be the smallest or largest element in the list.
If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n-1$ nested calls before we reach a list of size 1.
This means that the call tree is a linear chain of $n-1$ nested calls. The *ith* call does $O(n-i)$ work to do the partition, and $\sum_{i=0}^{n}(n-i) = O(n^2)$, so in that case quicksort takes $O(n^2)$ time.

Time Complexity

Worst-case Analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n-1$.This may occur if the pivot happens to be the smallest or largest element in the list.
If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n-1$ nested calls before we reach a list of size 1.
This means that the call tree is a linear chain of $n-1$ nested calls. The ith call does $O(n-i)$ work to do the partition, and $\sum_{i=0}^{n}(n-i) = O(n^2)$, so in that case quicksort takes $O(n^2)$ time.

Average-case Analysis

To sort an array of n distinct elements, quicksort takes $O(nlogn)$ time in expectation, averaged over all $n!$ permutations of n elements with equal probability.

Table of Contents

Merge sort

Introduction

Merge sort

Introduction

- Merge sort is an efficient, general-purpose, and comparison-based sorting algorithm.

Merge sort

Introduction

- Merge sort is an efficient, general-purpose, and comparison-based sorting algorithm.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.

Merge sort

Introduction

- Merge sort is an efficient, general-purpose, and comparison-based sorting algorithm.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.
- A detailed description and analysis of bottom-up merge sort appeared in a report by Goldstine and von Neumann as early as 1948.

Merge sort

Procedure

Merge sort

Procedure

- The elements are split into two sub-arrays ($n/2$) again and again until only one element is left.

Merge sort

Procedure

- The elements are split into two sub-arrays ($n/2$) again and again until only one element is left.
- Merge sort uses additional storage for sorting the auxiliary array.

Merge sort

Procedure

- The elements are split into two sub-arrays ($n/2$) again and again until only one element is left.
- Merge sort uses additional storage for sorting the auxiliary array.
- Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.

Merge sort

Procedure

- The elements are split into two sub-arrays ($n/2$) again and again until only one element is left.
- Merge sort uses additional storage for sorting the auxiliary array.
- Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
- At last, the all sub arrays are merged to make it 'n' element size of the array.

Merge sort

39	27	43	3	9	82	10

Figure: Merge sort Algorithm

Merge sort

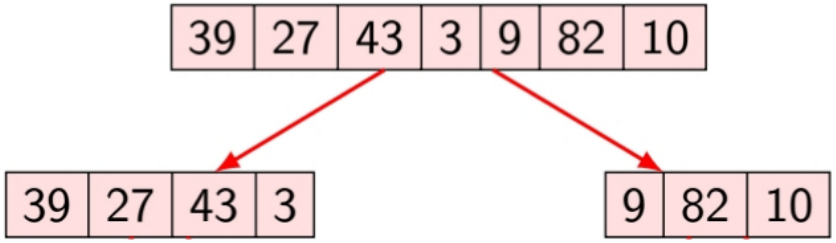

Figure: Merge sort Algorithm

Merge sort

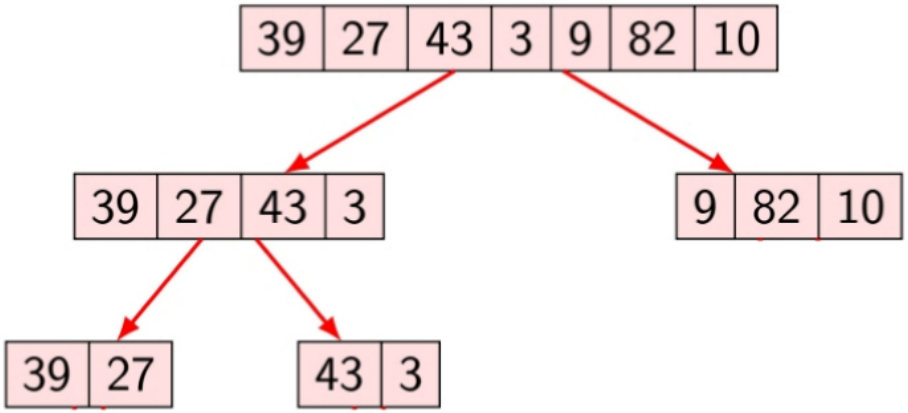

Figure: Merge sort Algorithm

Merge sort

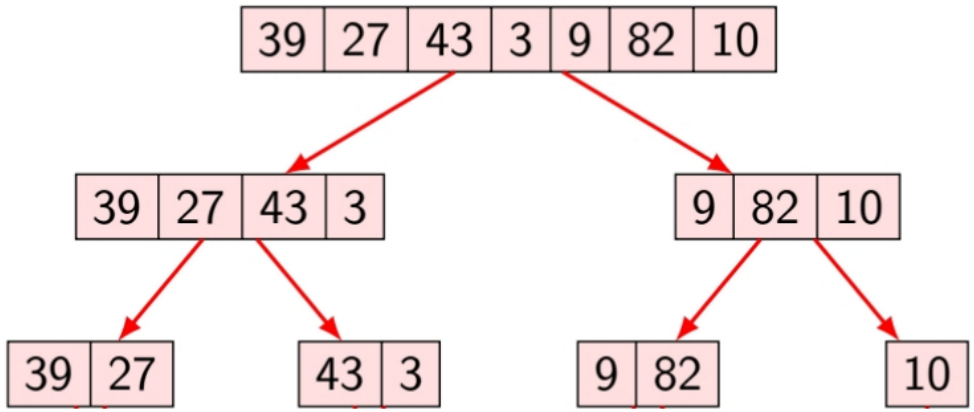

Figure: Merge sort Algorithm

Merge sort

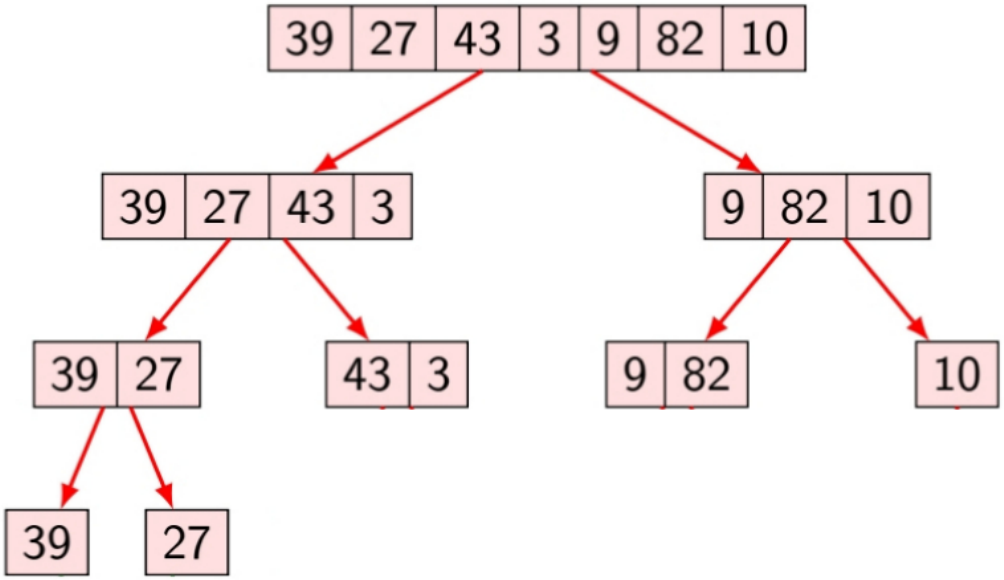

Figure: Merge sort Algorithm

Merge sort

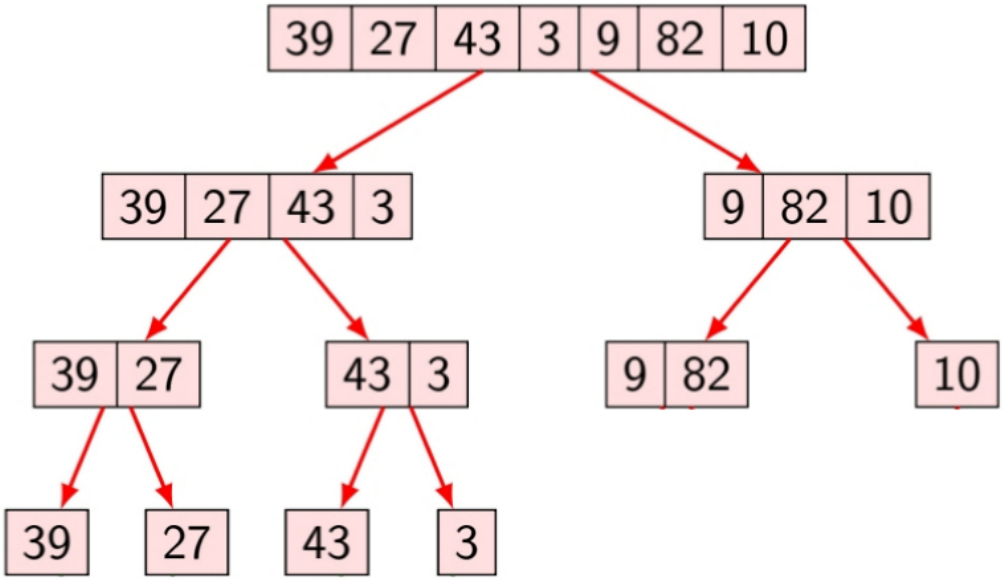

Figure: Merge sort Algorithm

Merge sort

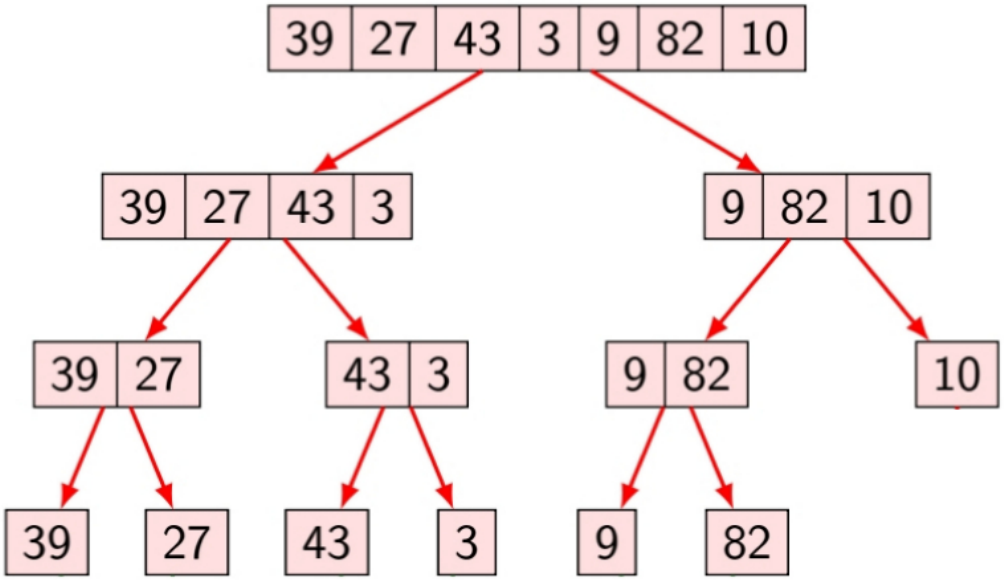

Figure: Merge sort Algorithm

Merge sort

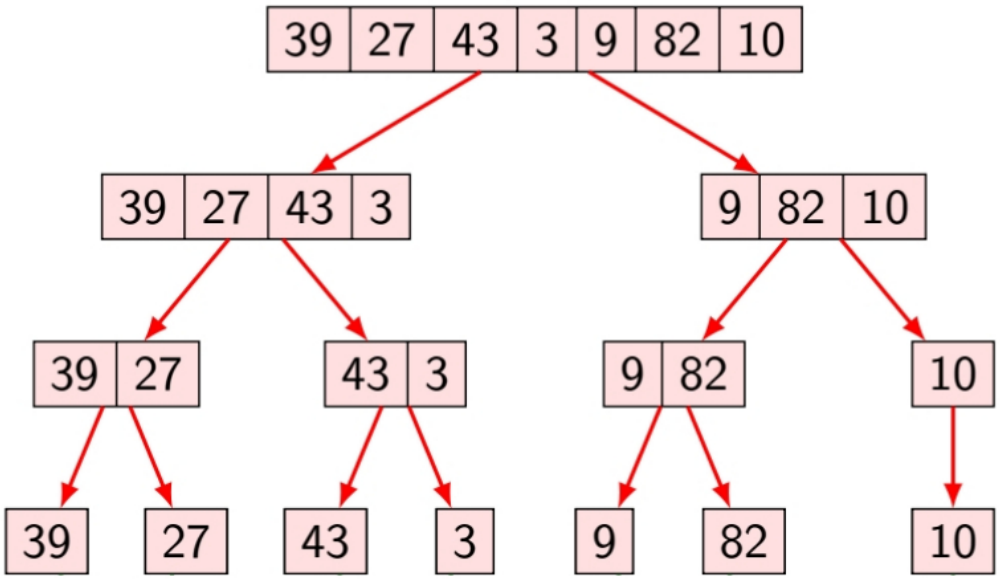

Figure: Merge sort Algorithm

Merge sort

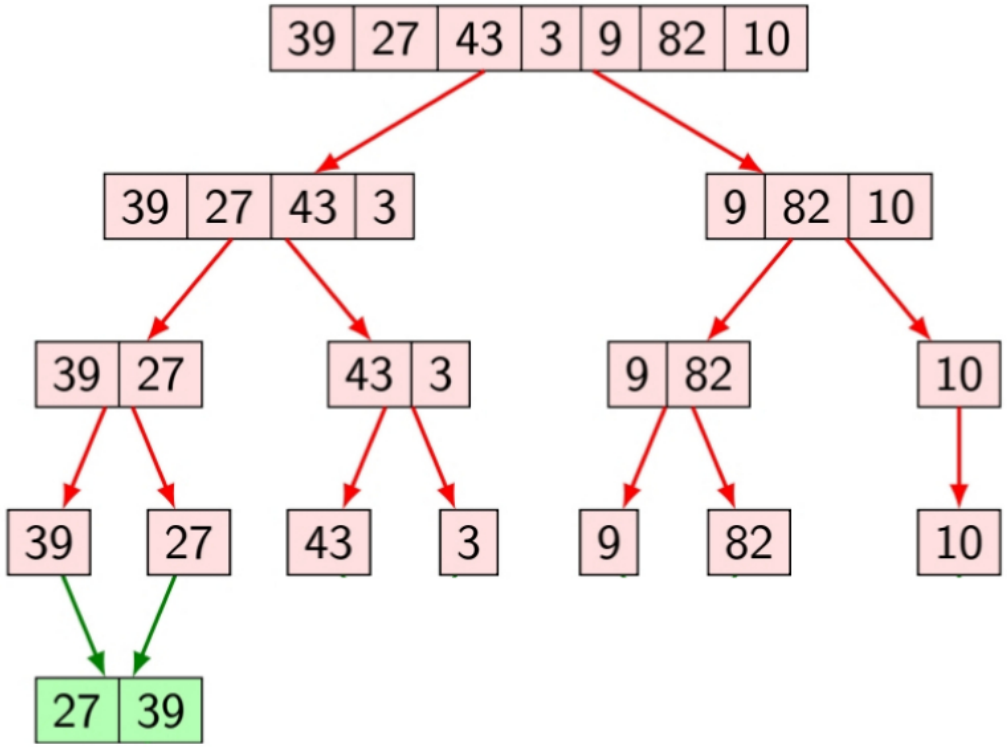

Figure: Merge sort Algorithm

Merge sort

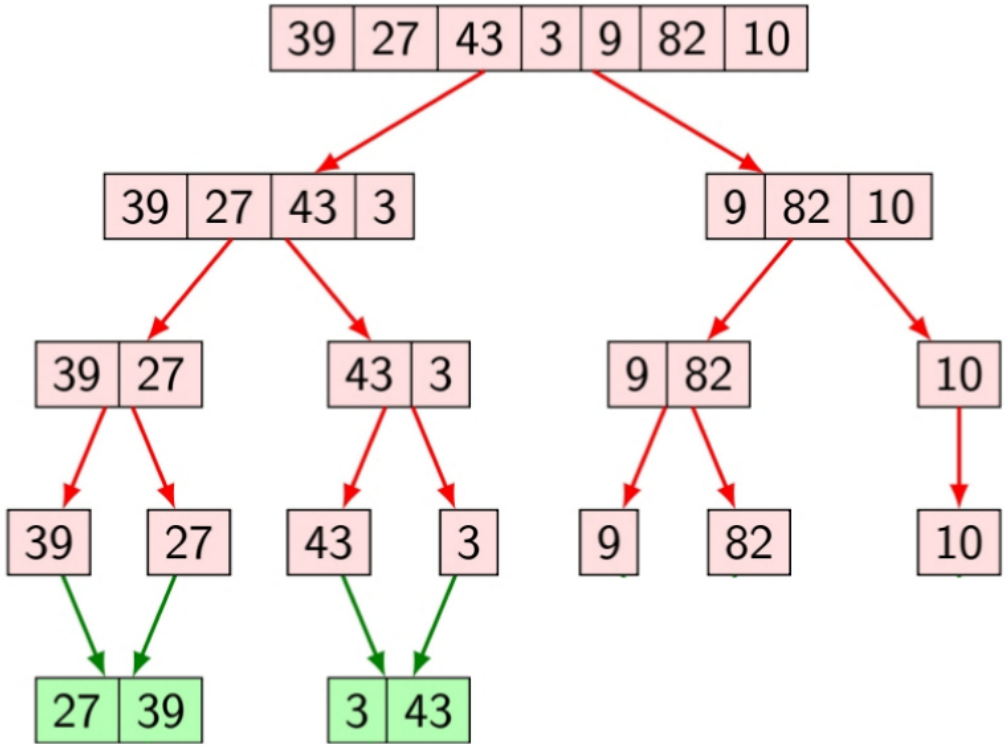

Figure: Merge sort Algorithm

Merge sort

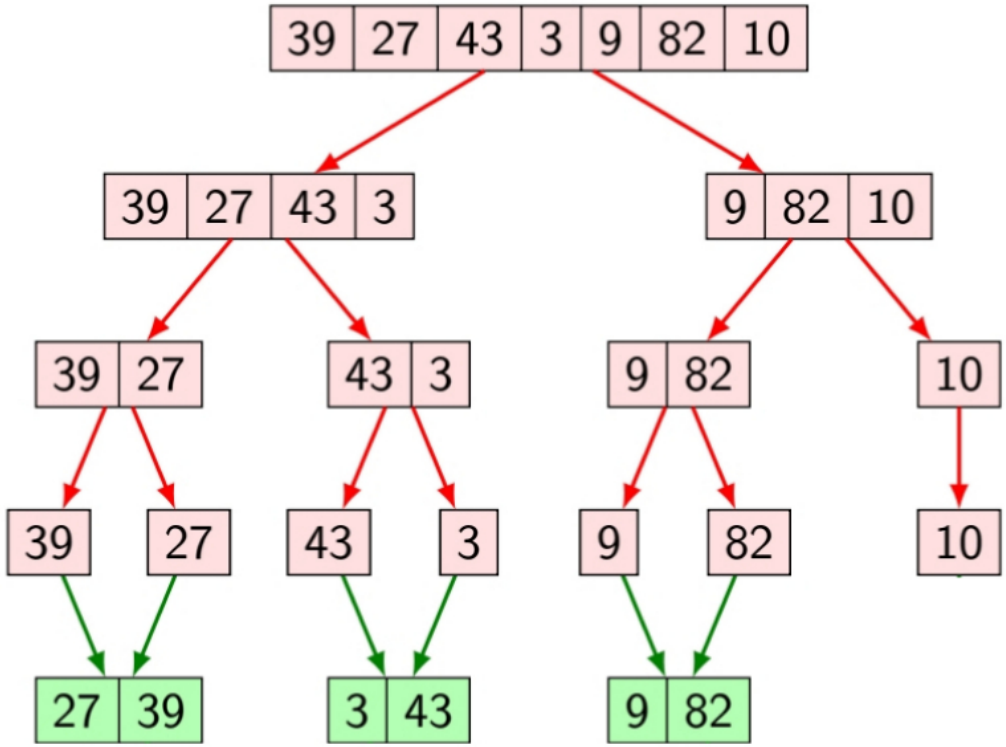

Figure: Merge sort Algorithm

Merge sort

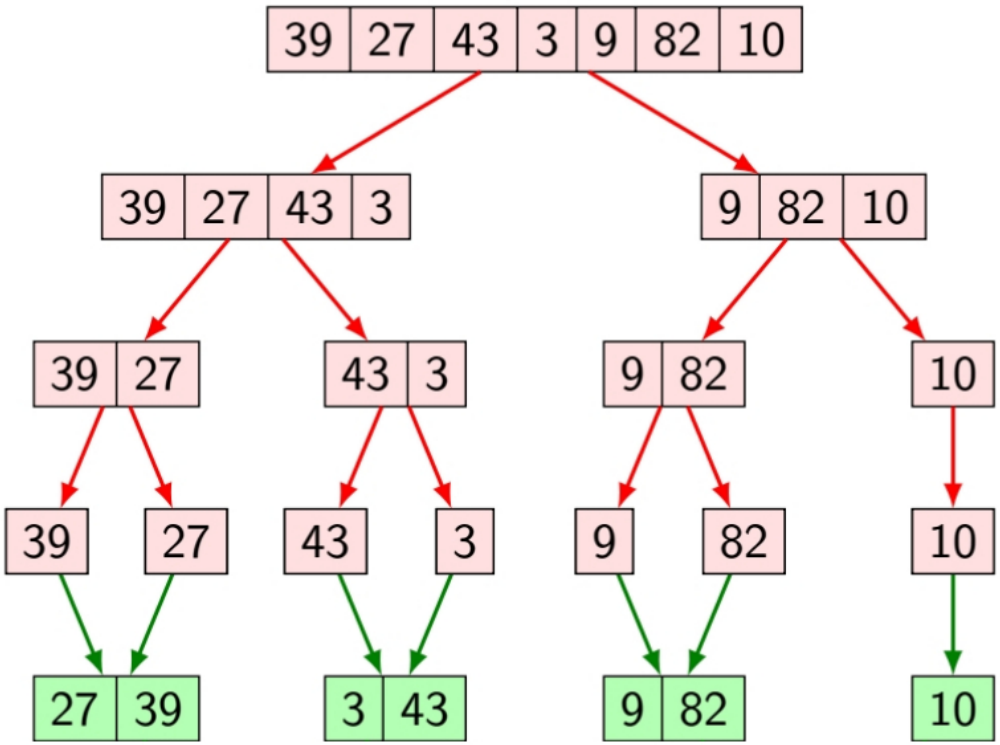

Figure: Merge sort Algorithm

Merge sort

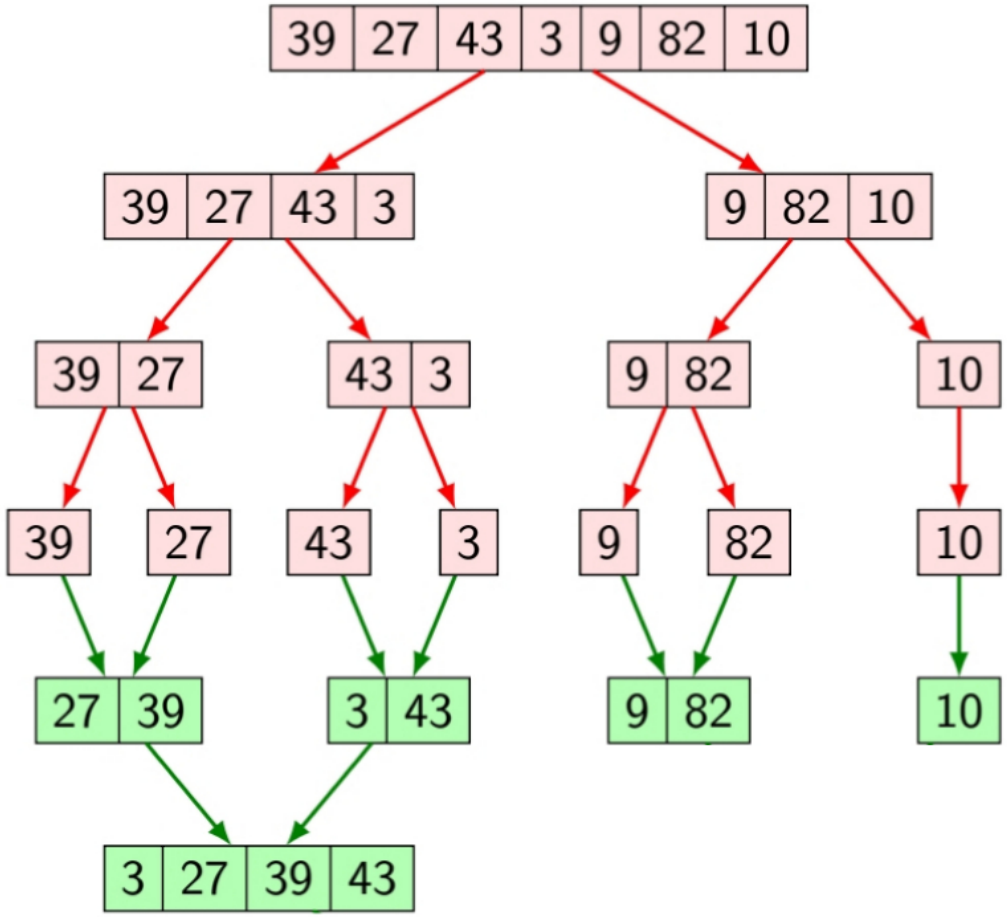

Figure: Merge sort Algorithm

Merge sort

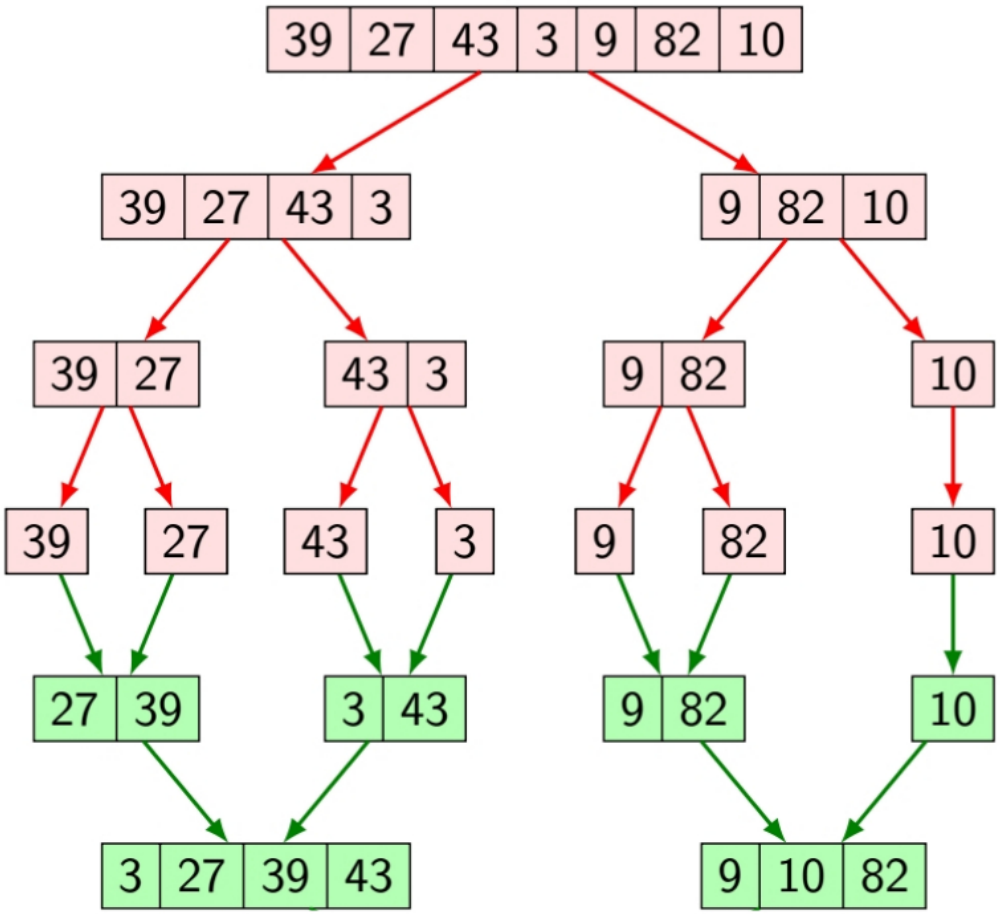

Figure: Merge sort Algorithm

Merge sort

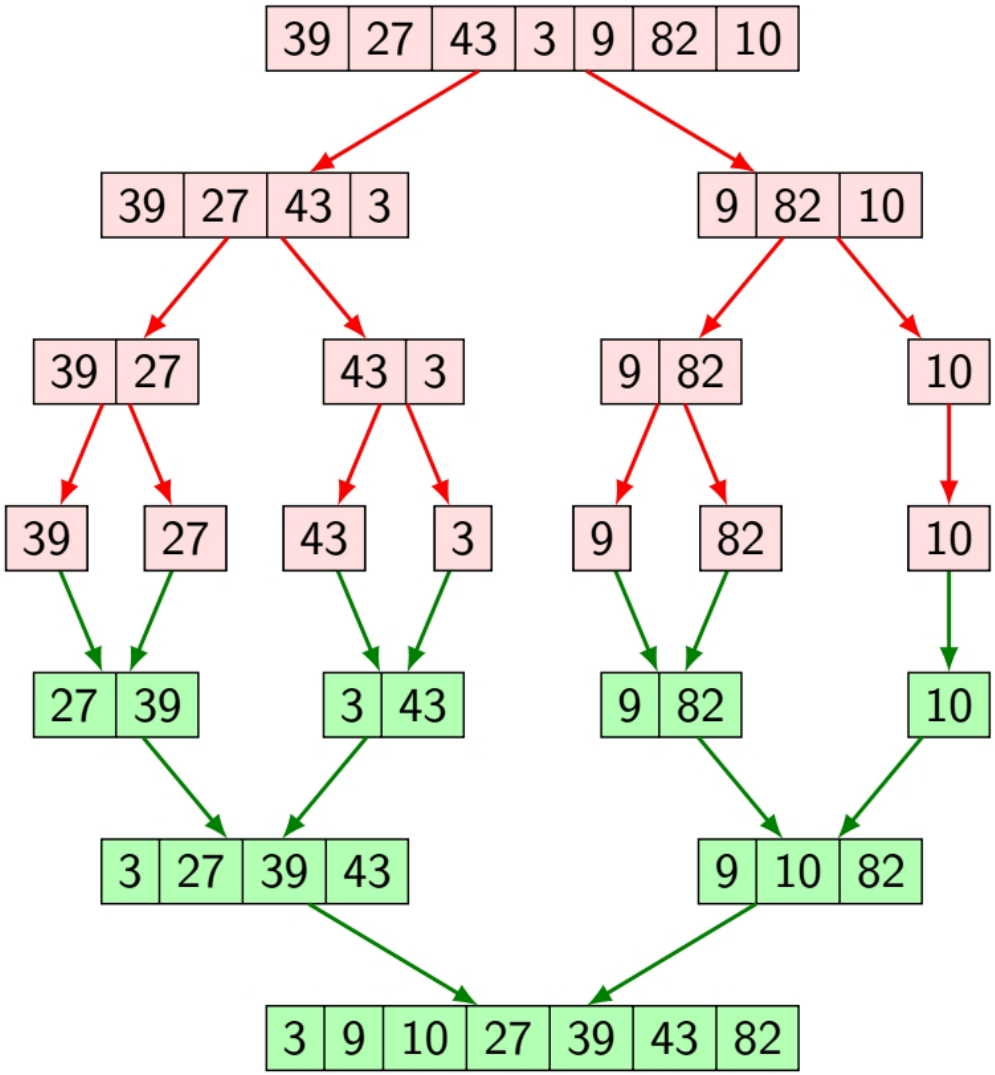

Figure: Merge sort Algorithm

Time Complexity

Time complexity of Merge Sort is $O(nlogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Time Complexity

Time complexity of Merge Sort is $O(nlogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Whenever we divide a number into half in every step, it can be represented using a logarithmic function ($logn$) and the number of steps can be represented by $logn + 1$(at most).

Time Complexity

Time complexity of Merge Sort is $O(nlogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Whenever we divide a number into half in every step, it can be represented using a logarithmic function ($logn$) and the number of steps can be represented by $logn + 1$(at most).
Also, we perform a single step operation to find out the middle of any subarray which requires $O(1)$ time.

Time Complexity

Time complexity of Merge Sort is $O(nlogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Whenever we divide a number into half in every step, it can be represented using a logarithmic function ($logn$) and the number of steps can be represented by $logn + 1$(at most).
Also, we perform a single step operation to find out the middle of any subarray which requires $O(1)$ time.
And to merge the subarrays, made by dividing the original array of n elements, a running time of $O(n)$ will be required.

Time Complexity

Time complexity of Merge Sort is $O(nlogn)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Whenever we divide a number into half in every step, it can be represented using a logarithmic function ($logn$) and the number of steps can be represented by $logn + 1$(at most).
Also, we perform a single step operation to find out the middle of any subarray which requires $O(1)$ time.
And to merge the subarrays, made by dividing the original array of n elements, a running time of $O(n)$ will be required.
Hence the total time for merge Sort function will become $n(logn + 1)$, which gives us a time complexity of $O(nlogn)$.

Table of Contents

Comparison

Basis for Comparison	Quicksort	Merge sort

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays
Sorting method	Internal	External

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays
Sorting method	Internal	External
Preferred for	Arrays	Linked Lists

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays
Sorting method	Internal	External
Preferred for	Arrays	Linked Lists
Speed of execution	It works faster on small data set	It has a consistent speed on any size of data

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays
Sorting method	Internal	External
Preferred for	Arrays	Linked Lists
Speed of execution	It works faster on small data set	It has a consistent speed on any size of data
Additional storage space requirement	Less	More

Comparison

Basis for Comparison	Quicksort	Merge sort
Worst case complexity	$O(n^2)$	$O(nlogn)$
Efficiency	Inefficient for larger arrays	More efficient for larger arrays
Sorting method	Internal	External
Preferred for	Arrays	Linked Lists
Speed of execution	It works faster on small data set	It has a consistent speed on any size of data
Additional storage space requirement	Less	More
Partition of elements	An array can be divided into any ratio	An array will be divided into two sub arrays