Sorting Techniques

Internal and External Sorting

Sorting algorithms that use main memory exclusively during the sort are called internal sorting algorithms. This kind of algorithm assumes high-speed random access to all memory. Some of the common algorithms that use this sorting feature are: Bubble Sort, Insertion Sort., and Quick Sort.

Sorting algorithms that use external memory, during the sorting come under this category. They are comparatively slower than internal sorting algorithms. For example merge sort algorithm. It sorts chunks that each fit in RAM, then merges the sorted chunks together.

Aspect	Internal Sorting	External Sorting
Data Size	Typically works well with data that fits in memory	Designed for sorting large data sets that exceed memory
Memory Usage	Uses the main memory (RAM) for sorting	Utilizes both main memory (RAM) and secondary storage
Performance	Generally faster due to direct memory access	Slower due to the involvement of disk I/O operations
Data Movement	Involves moving data within the main memory	Requires data transfer between main memory and disk
Sorting Algorithms	Various algorithms like Bubble, Selection, Insertion, QuickSort, etc.	Merge Sort
Buffering	May use buffers, but primarily within memory	Utilizes buffers to manage data transfer between memory
Use Cases	Sorting smaller data sets or in-memory structures	Sorting large databases, files, or data too big for memory

In-place and Out-of-place sorting

In-Place means your sorting algorithm does not use any extra memory other than the array to sort, while out-of-place means that your sorting algorithm **uses extra memory**. In-place means **that the input and output occupy the same memory storage space**.

In-place: bubble sort, insertion sort, selection sort etc.

Out-of-place sort: Merge Sort

Aspect	**In-Place Sorting**	**Out-of-Place Sorting**
Definition	Sorts the data within the same memory space	Sorts the data by creating a separate copy
Memory Usage	Uses the same memory space for sorting	Requires additional memory for the copy
Data Movement	Rearranges elements within the same array	Creates a new array for the sorted elements
Performance	Can be more efficient due to fewer memory operations	May require more memory operations for copying
Sorting Algorithms	Bubble, Selection, Insertion, Quicksort	MergeSort

Adaptive and Non-Adaptive Sorting

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some elements already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Adaptive sorting algorithms: Insertion Sort, Quick Sort, Radix Sort

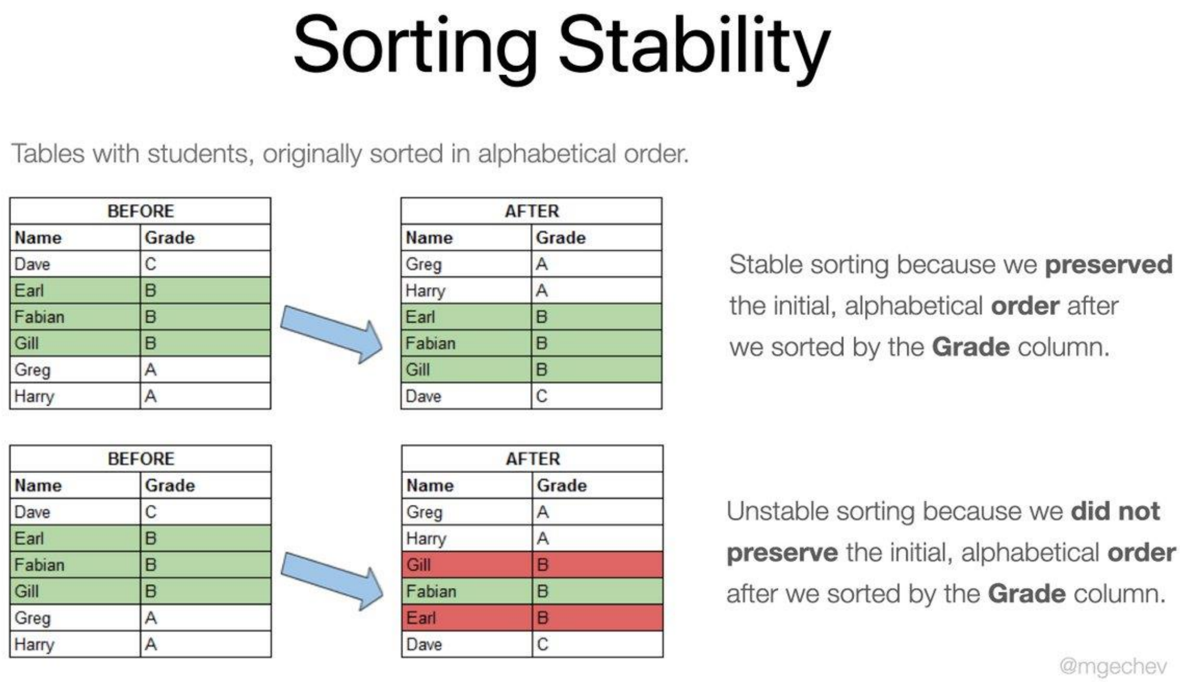
Example: Insertion Sort is an adaptive sorting algorithm. It starts with a partially sorted subarray and inserts each element into its correct position within the sorted portion of the array, adapting its behavior to the input order.

Non-adaptive sorting algorithms: Bubble Sort, Selection Sort, Merge Sort, Heap Sort

Example: Merge Sort is a non-adaptive sorting algorithm. It divides the input array into smaller subarrays, sorts them independently, and then merges them together using a fixed merge strategy, regardless of the input order.

Stable and Unstable Sorting

Sorting algorithm is stable if two elements with equal values appear in the same order in output as it was in the input. The stability of a sorting algorithm can be checked with how it treats equal elements. Stable algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equals elements similar to one another. For example – Insertion Sort, Bubble Sort, Merge sort, and Radix Sort.



Iterative and Recursive

Sorting algorithms are either recursive (for example – quick sort) or non-recursive (for example – selection sort, and insertion sort), and there are some algorithms which use both (for example – 2-way merge sort).

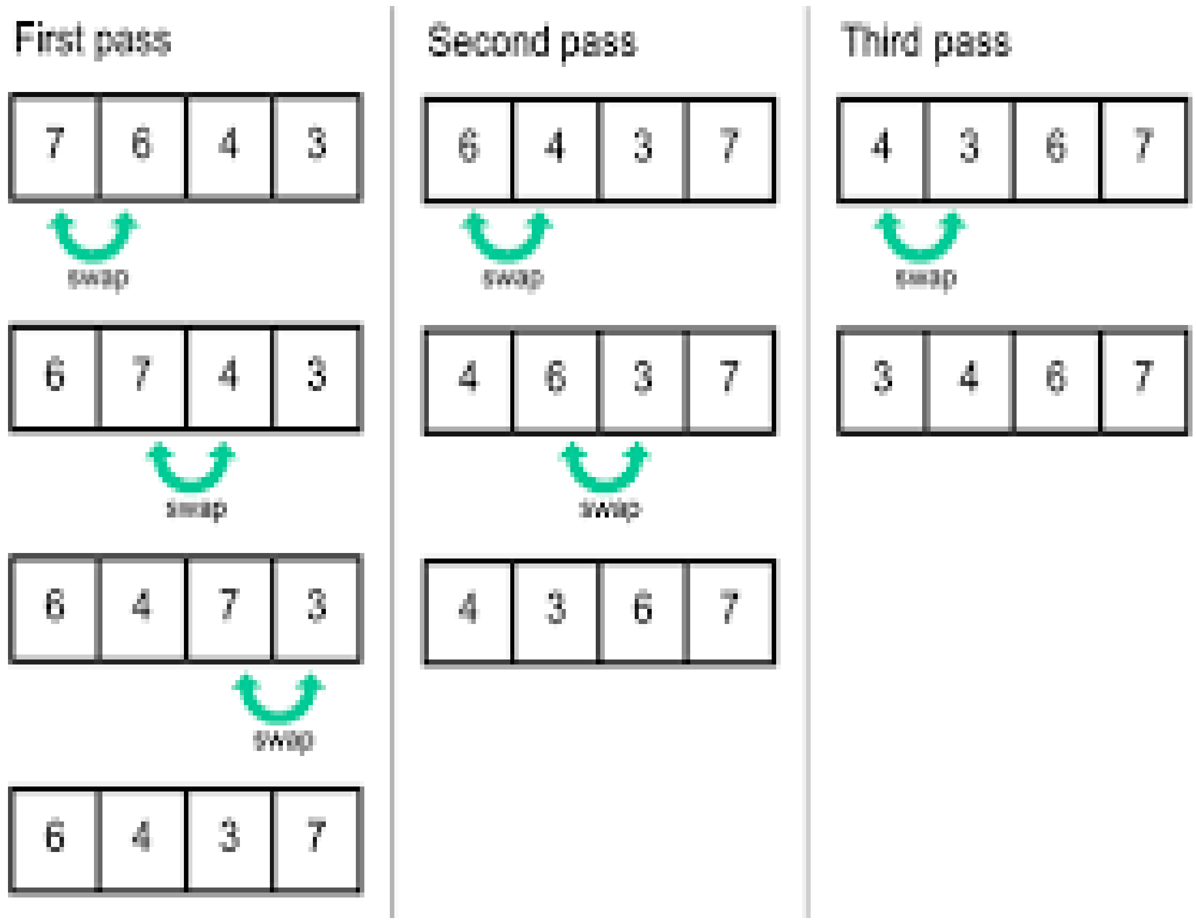
Sorting Algorithm	Time Complexity	Space Complexity	Internal Sorting	External Sorting	In-Place Sorting	Out-of-Place Sorting	Stable Sorting	Unstable Sorting	Adaptive Sorting	Non-Adaptive Sorting
Bubble Sort	O(n^2)	O(1)	Yes	No	Yes	No	Yes	No	No	Yes
Insertion Sort	O(n^2)	O(1)	Yes	No	Yes	No	Yes	No	Yes	Yes
Selection Sort	O(n^2)	O(1)	Yes	No	Yes	No	No	Yes	No	Yes
Merge Sort	O(n log n)	O(n)	Yes	No	No	Yes	Yes	No	No	Yes
Quick Sort	O(n^2)	O(log n)	Yes	No	Yes	No	No	Yes	No	Yes
Heap Sort	O(n log n)	O(1)	Yes	No	Yes	No	No	Yes	No	Yes
Radix Sort	O(d * (n + k))	O(n + k)	Yes	No	Yes	No	Yes	No	Yes	Yes
Counting Sort	O(n + k)	O(n + k)	Yes	No	No	Yes	Yes	No	No	Yes

Note:

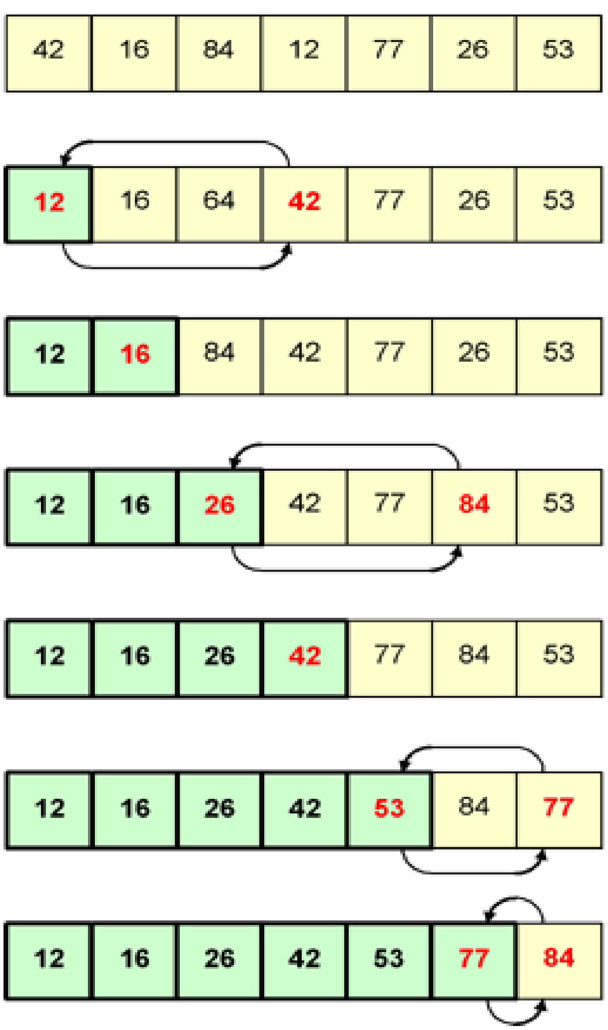
- Time complexity is represented using big O notation.
- "n" refers to the number of elements to be sorted.
- "k" refers to the range of possible key values in the input.
- "d" refers to the number of digits in the maximum key value.
- Space complexity refers to the additional space required by the algorithm, excluding the input space.
- Internal Sorting refers to sorting algorithms that operate entirely within the main memory.
- External Sorting refers to sorting algorithms designed for handling large datasets that cannot fit entirely in the main memory.
- In-Place Sorting refers to sorting algorithms that do not require extra space proportional to the input size.
- Out-of-Place Sorting refers to sorting algorithms that require additional space proportional to the input size.

- Stable Sorting refers to sorting algorithms that maintain the relative order of elements with equal keys.
- Unstable Sorting refers to sorting algorithms that do not guarantee the relative order of elements with equal keys.
- Adaptive Sorting refers to sorting algorithms that are efficient for partially sorted or nearly sorted input.
- Non-Adaptive Sorting refers to sorting algorithms whose performance is not affected by the initial order of the input.

Bubble Sort:



Selection Sort:



The array, before the selection sort operation begins.

The smallest number (**12**) is swapped into the first element in the structure.

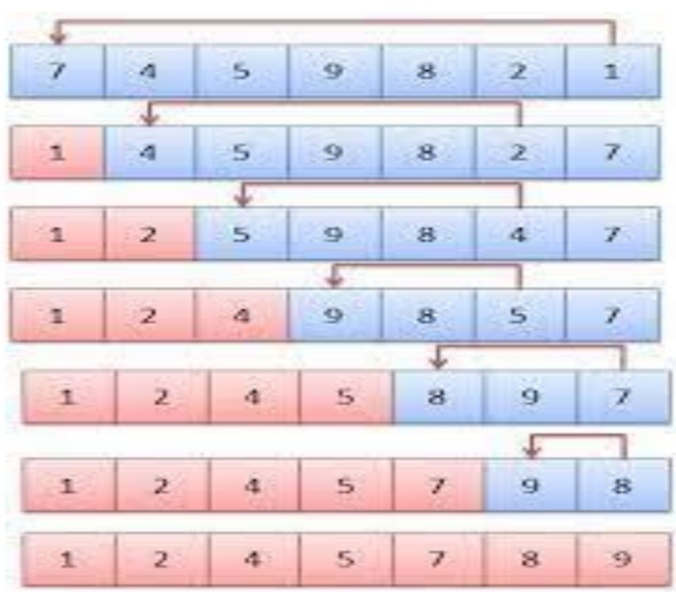
In the data that remains, **16** is the smallest; and it does not need to be moved.

26 is the next smallest number, and it is swapped into the third position.

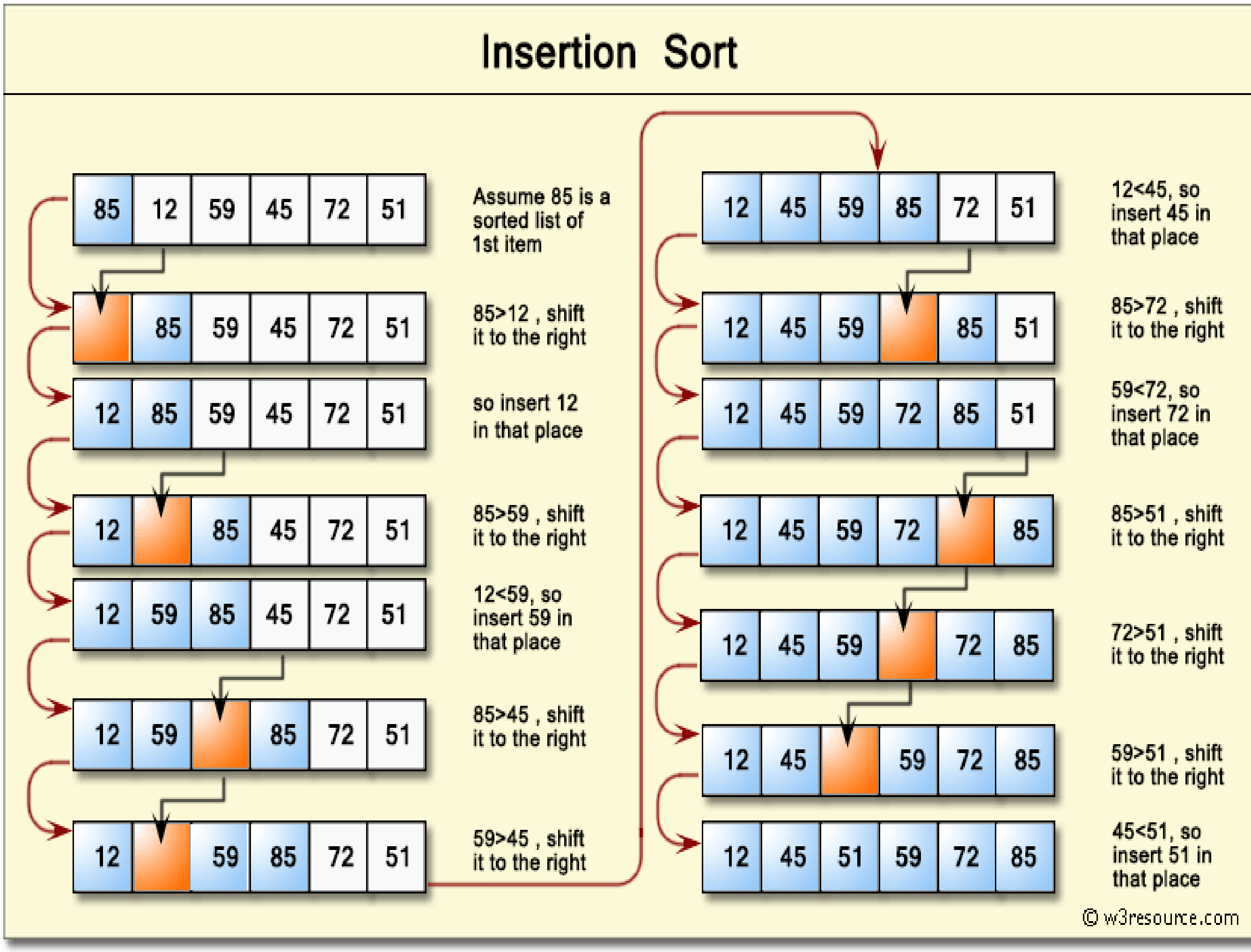
42 is the next smallest number; it is already in the correct position.

53 is the smallest number in the data that remains; and it is swapped to the appropriate position.

Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*



Insertion Sort:



Divide and Conquer Technique

The divide and conquer technique is a problem-solving approach that involves breaking down a complex problem into smaller, more manageable subproblems, solving them independently, and then combining the solutions to solve the original problem. It follows a recursive algorithmic paradigm.

The general steps in the divide and conquer technique are:

1. Divide: Break the problem into smaller subproblems that are similar to the original problem but of reduced size. This step is typically performed recursively until the subproblems become simple enough to be solved directly.
2. Conquer: Solve the subproblems independently. If the subproblems are small enough, they can be solved using a straightforward algorithm or a base case.

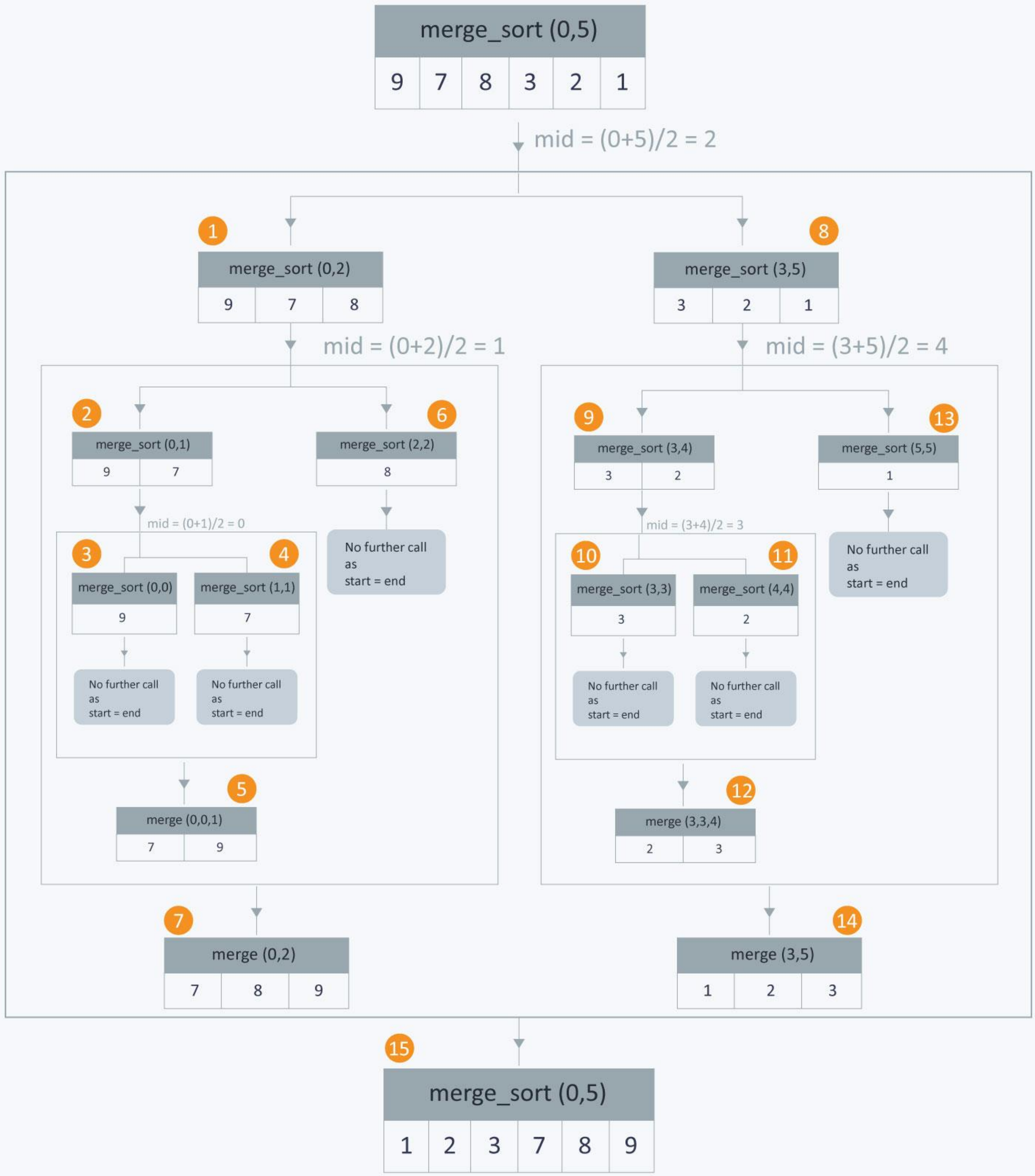
3. Combine: Merge the solutions of the subproblems to obtain the solution for the original problem. This step involves combining the smaller solutions in a way that results in a solution for the larger problem.

The divide and conquer technique is often used for solving problems that exhibit overlapping subproblems and can be divided into independent parts. Examples of algorithms that use this technique include merge sort, quicksort, binary search, and the fast Fourier transform (FFT).

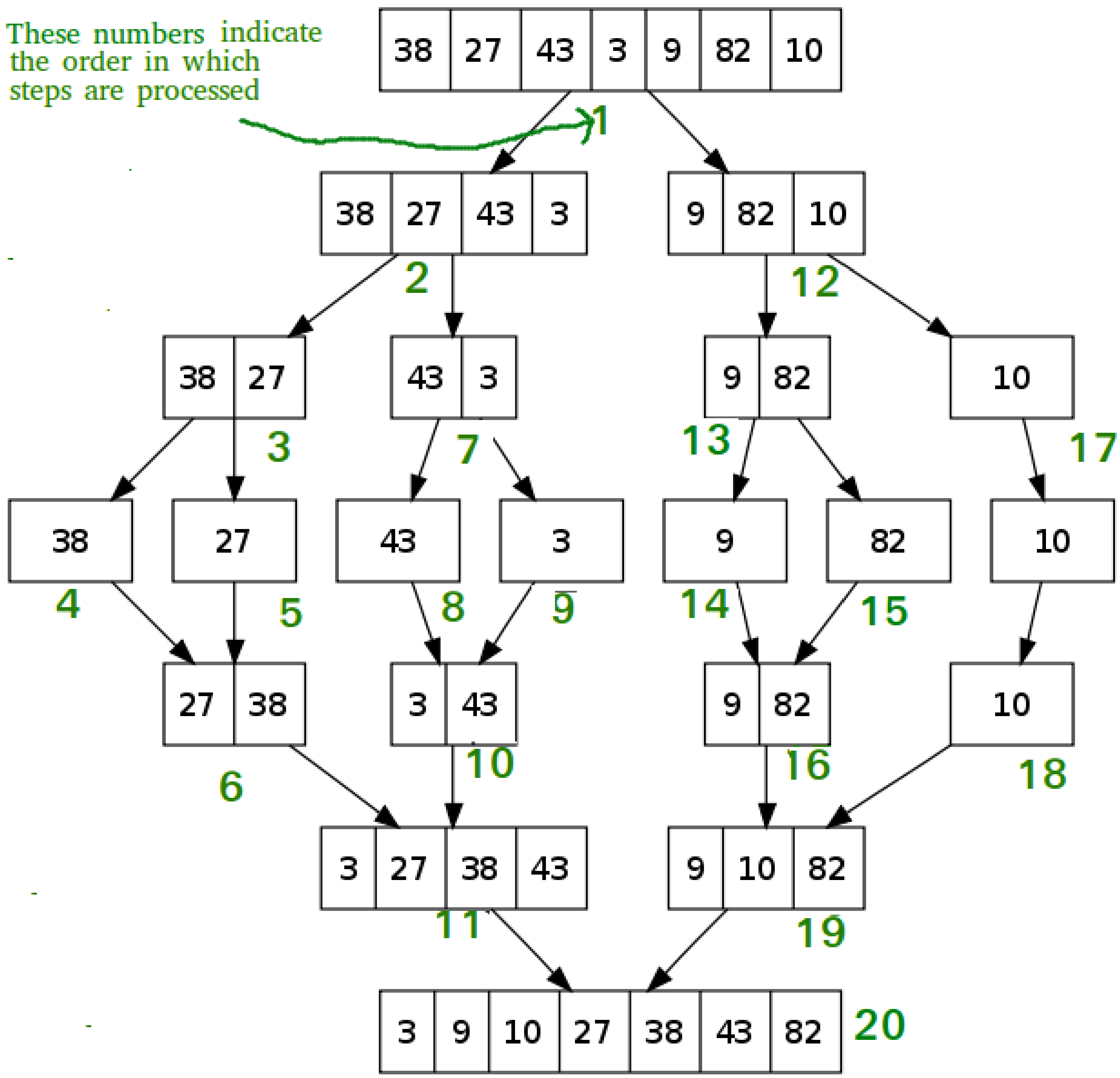
By dividing the problem into smaller, manageable parts and leveraging the concept of recursion, the divide and conquer technique can often lead to efficient and elegant solutions for a wide range of problems.

Merge Sort:

Merge Sort



merge_sort (0,5)
9 7 8 3 2 1
mid = (0+5)/2 = 2
1
merge_sort (0,2)
9 7 8
mid = (0+2)/2 = 1
2
merge_sort (0,1)
9 7
mid = (0+1)/2 = 0
3
merge_sort (0,0)
9
No further call as start = end
4
merge_sort (1,1)
7
No further call as start = end
5
merge (0,0,1)
7 9
6
merge_sort (2,2)
8
No further call as start = end
7
merge (0,2)
7 8 9
8
merge_sort (3,5)
3 2 1
mid = (3+5)/2 = 4
9
merge_sort (3,4)
3 2
mid = (3+4)/2 = 3
10
merge_sort (3,3)
3
No further call as start = end
11
merge_sort (4,4)
2
No further call as start = end
12
merge (3,3,4)
2 3
13
merge_sort (5,5)
1
No further call as start = end
14
merge (3,5)
1 2 3
15
merge_sort (0,5)
1 2 3 7 8 9



Time Complexity of Merge sort:
The list of size **N** is divided into a max of **logN** parts, and the merging of all sublists into a single list takes **O(N)** time, the worst case run time of this algorithm is **O(NlogN)**.

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:
1. Repeatedly increase the pointer 'up' until a[up] >= pivot.
2. Repeatedly decrease the pointer 'down' until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

Time Complexity

The average and best-case time complexity of the Quicksort algorithm is **O(n log n)**, where "n" represents the number of elements in the input array.

Quicksort achieves this time complexity through its divide-and-conquer strategy. It works by selecting a pivot element from the array, partitioning the array into two subarrays (elements less than the pivot and elements greater than the pivot), and recursively applying the same process to the subarrays.

In the average and best case scenarios, Quicksort generally partitions the array into approximately equal-sized subarrays during each recursive call. This balanced partitioning leads to a logarithmic height of the recursion tree, resulting in an overall time complexity of O(n log n).

In the worst-case scenario, the time complexity of Quicksort can indeed become **O(n^2)**, where "n" represents the number of elements in the input array. This occurs when the chosen pivot element consistently partitions the array into highly imbalanced subarrays.

The worst-case scenario typically arises in Quicksort when the input array is already sorted (in ascending or descending order) or contains many duplicate elements. If the pivot chosen is either the smallest or largest element in each partition, the resulting subarrays will have sizes that differ greatly.

In such cases, the recursion tree of Quicksort becomes skewed, resembling a linear chain, where each recursive call processes only one less element than the previous call. As a result, the algorithm requires a large number of comparisons and swaps, leading to a time complexity of O(n^2).

To mitigate the worst-case scenario, several techniques can be employed, such as:

1. Randomized Pivot Selection: Randomly selecting the pivot element can help reduce the likelihood of encountering a worst-case scenario. By choosing a random pivot, the probability of consistently selecting the smallest or largest element decreases, promoting more balanced partitioning.
2. Median-of-Three Pivot Selection: Instead of selecting a pivot randomly, this technique selects the pivot as the median of the first, middle, and last elements of the array. This approach helps improve partitioning even in the presence of sorted or partially sorted input.

By using these techniques, the worst-case time complexity of Quicksort can be avoided, ensuring that the algorithm performs efficiently with an average and best-case time complexity of O(n log n).

Example of Quicksort:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.
This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.
Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	