**INSERTION SORT**

Consider an example: arr[]: {12, 11, 13, 5, 6}

1. First Pass: Initially, the first two elements of the array are compared in insertion sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 12 | 11 | 13 | 5 | 6 |

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 13 | 5 | 6 |

1. Second Pass: Now, move to the next two elements and compare them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 13 | 5 | 6 |

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

1. Third Pass: Now, two elements are present in the sorted sub-array which are 11 and 12

Moving forward to the next two elements which are 13 and 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 13 | 5 | 6 |

Both 5 and 13 are not present at their correct place so swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 5 | 13 | 6 |

After swapping, elements 12 and 5 are not sorted, thus swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 5 | 12 | 13 | 6 |

Here, again 11 and 5 are not sorted, hence swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | 13 | 6 |

Here, 5 is at its correct position

1. Fourth Pass: Now, the elements which are present in the sorted sub-array are 5, 11 and 12

Moving to the next two elements 13 and 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | 13 | 6 |

Clearly, they are not sorted, thus perform swap between both

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | 6 | 13 |

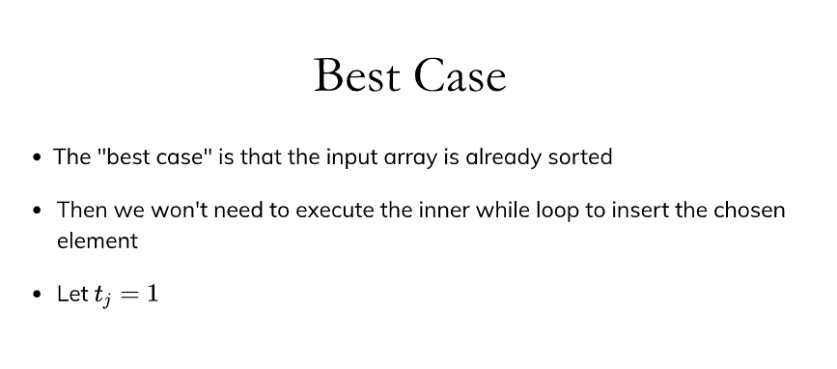
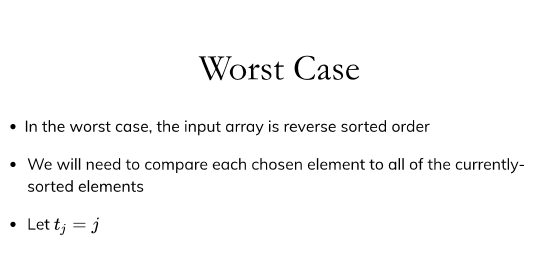
Now, 6 is smaller than 12, hence, swap again

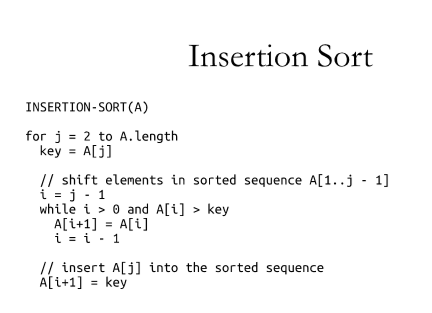
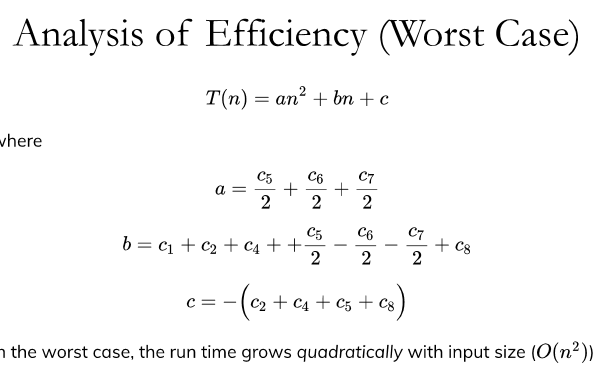
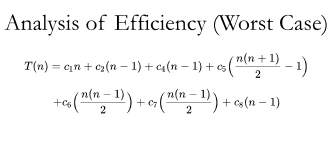
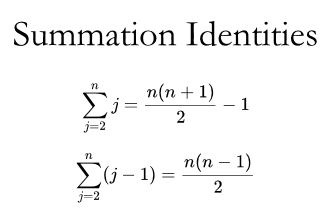
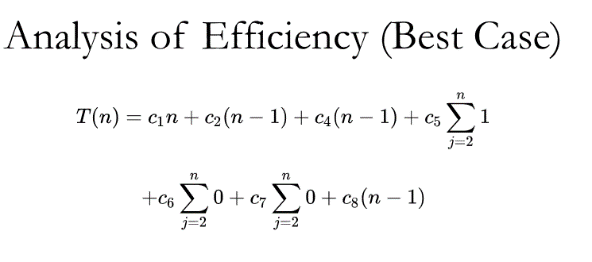
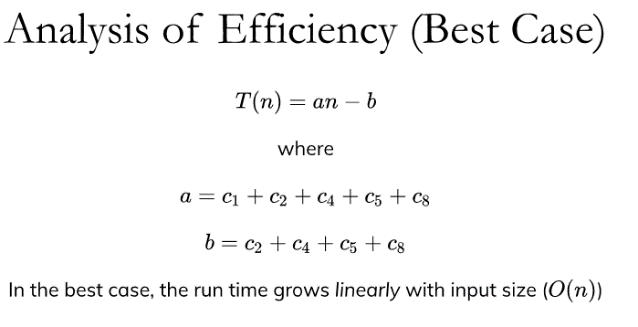
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 6 | 12 | 13 |

Here, also swapping makes 11 and 6 unsorted hence, swap again

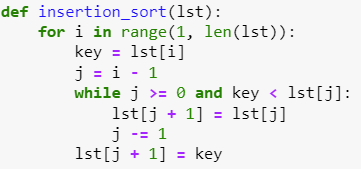
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 6 | 11 | 12 | 13 |

Finally, the array is completely sorted.





* **When to Use**: Insertion Sort is efficient for small data sets or nearly sorted data.
* **Why**: It has a simple implementation and works well when the input data is mostly sorted because it makes minimal comparisons and swaps.

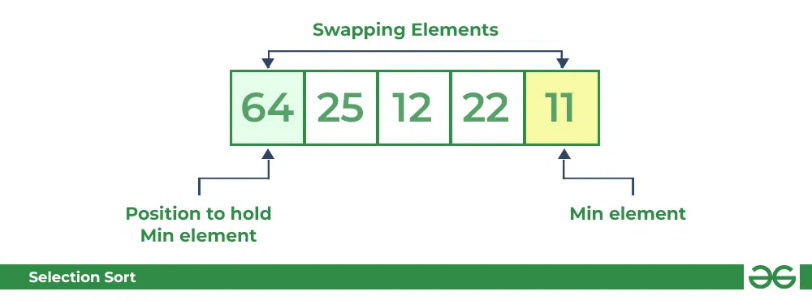


**SELECTION SORT**

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

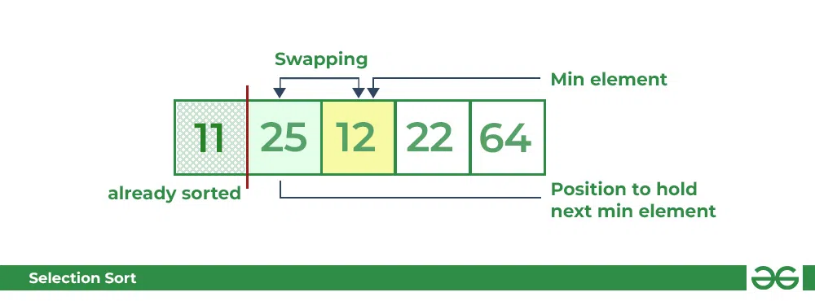
1. For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

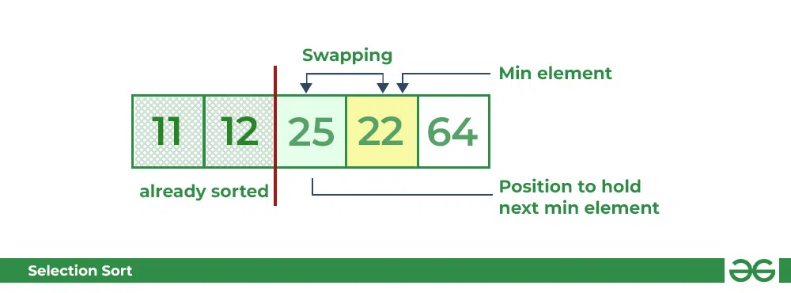


1. Second Pass: For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

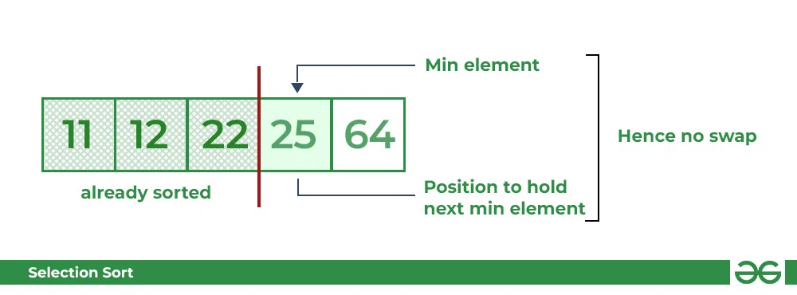
After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



1. Third Pass: Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array. While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



1. Fourth pass: Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array. As 25 is the 4th lowest value hence, it will place at the fourth position.



1. Fifth Pass: At last the largest value present in the array automatically get placed at the last position in the array. The resulted array is the sorted array.

A green rectangular box with numbers

Description automatically generated

A math equation with numbers and symbols

Description automatically generated with medium confidenceA math equation with black text

Description automatically generated

A white paper with writing on it

Description automatically generated

* **When to Use**: Selection Sort is suitable for small data sets and is generally more efficient than Bubble Sort.
* **Why**: It consistently performs the same number of swaps regardless of the initial order of the elements.

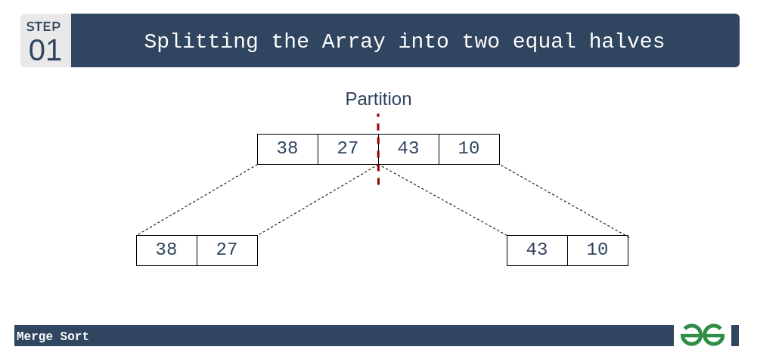
A screenshot of a computer code

Description automatically generated

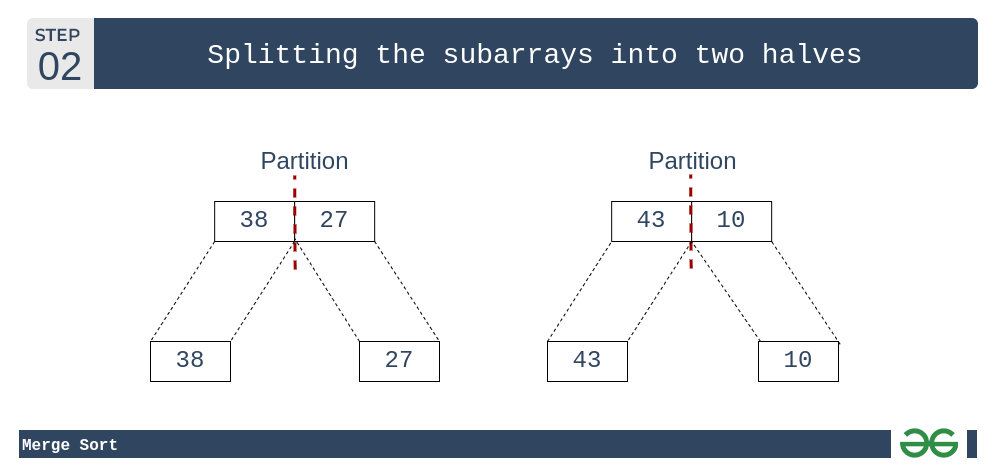
**MERGE SORT**

Lets consider an array arr[] = {38, 27, 43, 10}

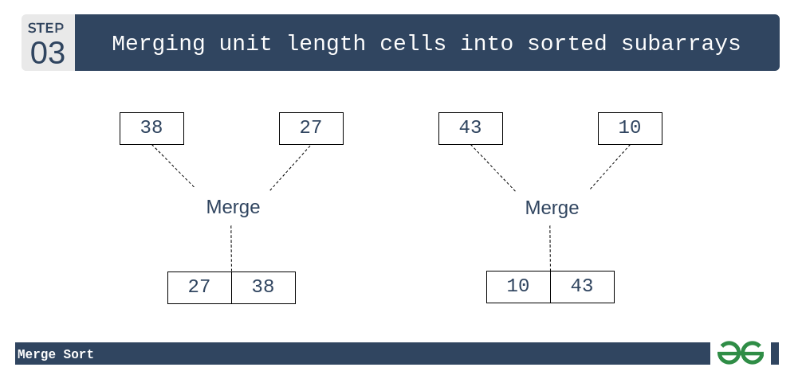
1. Initially divide the array into two equal halves:



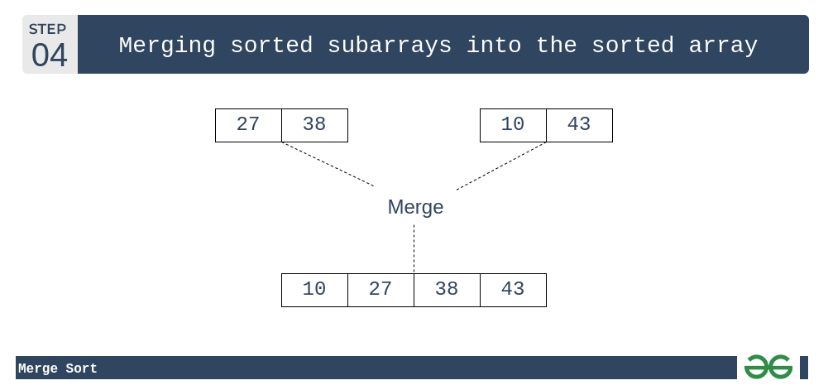
1. These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



1. These sorted subarrays are merged together, and we get bigger sorted subarrays.



1. This merging process is continued until the sorted array is built from the smaller subarrays.



A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generatedA diagram of a mathematical equation

Description automatically generated

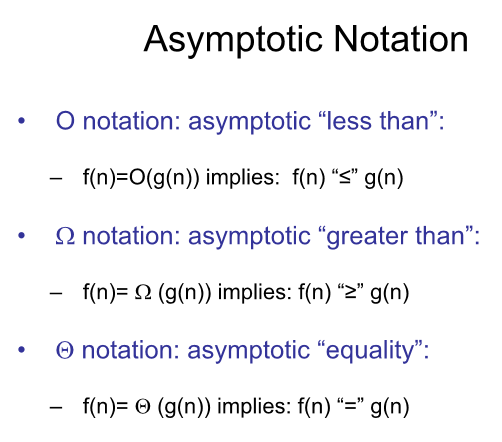
* **When to Use**: Merge Sort is efficient for large data sets and is a good choice for sorting linked lists.
* **Why**: It's a divide-and-conquer algorithm that guarantees a stable sort and has a consistent O(n log n) time complexity.

A screenshot of a computer screen

Description automatically generatedA white background with black text

Description automatically generated

**BIG O Notation**



A math equations on a white background

Description automatically generated

A math equations on a white background

Description automatically generated