**SORTING ALGORITHMS**

1. **Bubble Sort:**

**Algorithm:**

1. Start from the first element of the list.
2. Compare the current element with the next element.
3. If the current element is greater than the next element (for ascending order), swap them.
4. Move to the next pair and repeat the process until the end of the list.
5. After each full pass through the list, the largest element is in its correct position, so reduce the range of comparisons.
6. Repeat the process for the rest of the list until no more swaps are needed.

Time Complexity: O (n²)

Space Complexity: O (1)

Stable: Yes

It is because the sizeof() operator returns the size of a type in **bytes**.

You learned from the [Data Types chapter](https://www.w3schools.com/cpp/cpp_data_types.asp) that an int type is usually 4 bytes, so from the example above, *4 x 5 (4 bytes x 5 elements) =****20 bytes***.

**To find out how many elements an array has**, you have to divide the size of the array by the size of the first element in the array:

Example

int myNumbers[5] = {10, 20, 30, 40, 50};  
int getArrayLength = **sizeof(myNumbers) / sizeof(myNumbers[0])**;  
cout << getArrayLength;

1. **Selection Sort:**

**Algorithm:**

1. Start with the first element and assume it's the minimum.
2. Compare it with all other elements in the unsorted portion to find the smallest element.
3. Swap the smallest element with the first element of the unsorted portion.
4. Move the boundary between the sorted and unsorted portions one step to the right.
5. Repeat until the list is sorted.

Time Complexity: O (n²)

Space Complexity: O (1)

Stable: No

1. **Insertion Sort:**

**Algorithm:**

1. Start from the second element (assume the first element is already sorted).
2. Compare the current element with elements in the sorted section (to its left).
3. Shift elements that are greater than the current element one position to the right to make space.
4. Insert the current element in its correct position in the sorted section.
5. Repeat for all elements until the array is sorted.

Time Complexity: O(n²)

Space Complexity: O(1)

Stable: Yes

1. **Merge Sort:**

**Merge Sort** is a divide-and-conquer algorithm that splits the array into two halves, recursively sorts each half, and then merges the two sorted halves to produce a sorted array. It's one of the most efficient sorting algorithms with a time complexity of O(n log n).

**Key Steps in Merge Sort:**

1. **Divide**: The array is divided into two halves until each subarray contains a single element.
2. **Conquer**: Each subarray is sorted recursively.
3. **Combine**: The sorted subarrays are merged together to form a sorted array.

**Time Complexity:**

* Best, Average, and Worst Case: O(n log n)
* Space Complexity: O(n) (due to the use of temporary arrays during merging).

**Merge Sort Algorithm:**

1. Recursively split the array in half until each subarray has one element.
2. Merge the subarrays back together in sorted order.

Time Complexity: O(n log n)

Space Complexity: O(n)

Stable: Yes