**# Exercises**

**## Exercise 1 - Consider the JavaScript Code**

```js

// Waits until the page is fully loaded before generating bars

document.addEventListener("DOMContentLoaded", () => {

  generateBars();

});

// Function to generate random bars

function generateBars() {

  const barsContainer = document.getElementById("bars");

  barsContainer.innerHTML = ""; // Clears existing bars

  // Generates an array of 10 random heights between 10 and 100 pixels

  let numbers = Array.from(

    { length: 10 },

    () => Math.floor(Math.random() \* 100) + 10

  );

  numbers.forEach((num) => {

    let bar = document.createElement("div");

    bar.classList.add("bar");

    bar.style.height = `${num}px`;

    barsContainer.appendChild(bar);

  });

}

// Function to disable sorting buttons during sorting

function disableButtons() {

  document.querySelector("button[onclick='bubbleSort()']").disabled = true;

  document.querySelector("button[onclick='quickSort()']").disabled = true;

}

// Function to enable sorting buttons

function enableButtons() {

  document.querySelector("button[onclick='bubbleSort()']").disabled = false;

  document.querySelector("button[onclick='quickSort()']").disabled = false;

}

// Function to perform the bubble sort algorithm on the bars

async function bubbleSort() {

  disableButtons();

  // TODO: Implement the code for the Bubble Sort algorithm

  enableButtons();

}

// Function to perform the quick sort algorithm on the bars

async function quickSort(low = 0, high = null) {

  let bars = document.querySelectorAll(".bar");

  if (high === null) high = bars.length - 1;

  if (low < high) {

    let pivotIndex = await partition(bars, low, high);

    await quickSort(low, pivotIndex - 1);

    await quickSort(pivotIndex + 1, high);

  }

  // Enable buttons after sorting is completely done (only when recursion ends)

  if (low === 0 && high === bars.length - 1) {

    enableButtons();

  }

}

// Partition function for Quick Sort

async function partition(bars, low, high) {

  let pivot = parseInt(bars[high].style.height);

  let i = low - 1;

  for (let j = low; j < high; j++) {

    let heightJ = parseInt(bars[j].style.height);

    if (heightJ < pivot) {

      i++;

      await swap(bars[i], bars[j]);

    }

  }

  await swap(bars[i + 1], bars[high]);

  return i + 1;

}

// Function to swap two bars (divs)

function swap(bar1, bar2) {

  return new Promise((resolve) => {

    setTimeout(() => {

      let temp = bar1.style.height;

      bar1.style.height = bar2.style.height;

      bar2.style.height = temp;

      resolve();

    }, 200);

  });

}

```

**### Task:**

Implement the Bubble Sort method.

**### Hint**

These are the correct lines of code put in a random order.

```js

let len = bars.length; // Number of bars

await swap(bars[j], bars[j + 1]); // Calls the swap function

let height1 = parseInt(bars[j].style.height); // Gets the height of the first bar

// Inner loop iterates through unsorted part of the array

for (let j = 0; j < len - 1 - i; j++) {

  // What goes inside here?

}

// If the current bar is taller than the next one, swap them

if (height1 > height2) {

  // What goes inside here?

}

let height2 = parseInt(bars[j + 1].style.height); // Gets the height of the next bar

// Outer loop iterates through the array

for (let i = 0; i < len - 1; i++) {

  // What goes inside here?

}

let bars = document.querySelectorAll(".bar"); // Select all bars

```

**## Exercise 2 - Consider the sudo code for these algorithms**

**### Bubble Sort**

**#### Plaintext**

1. Start at the *\_beginning\_* of the list.

2. Compare the *\_first two elements\_* :

   - If the first is *\_greater\_* than the second, *\_swap them\_*.

   - Otherwise, leave them as they are.

3. Move to the *\_next pair\_* and *\_repeat\_* the process.

4. Continue until the *\_end of the list\_*—this completes one full *\_pass\_*.

5. Repeat the process `N` *\_times\_* (where `N` is the number of elements).

6. With each pass, the *\_largest unsorted element "bubbles up" to its correct position\_*.

7. Continue until no swaps are needed.

**#### Pseudocode**

```

BubbleSort(array)

    n = length of array

    repeat

        swapped = false

        for i = 0 to n-2 do

            if array[i] > array[i + 1] then

                swap array[i] with array[i + 1]

                swapped = true

        until swapped is false

```

swapped is used to track whether any elements were swapped during a single pass through the array. If no elements were swapped, the array is already sorted, and the algorithm stops.

The outer repeat loop keeps going until no swaps are made during a full pass through the array.

The inner for loop compares adjacent elements and swaps them if necessary.

**### Quick Sort**

**#### Plaintext**

1. Pick a "pivot" element from the list (usually the last element, but it could be any element).

2. Partition the list around the pivot:

   Rearrange the list so that all elements smaller than the pivot are on the left.

   All elements greater than the pivot are on the right.

   The pivot is now in its correct position, where it will stay once the list is fully sorted.

3. Recursively apply the same steps to the left and right sublists created by the pivot:

   The left sublist contains elements smaller than the pivot.

   The right sublist contains elements larger than the pivot.

4. Continue until each sublist contains only one element or is empty, at which point the sublists are considered sorted.

5. Merge the sublists—the result is a sorted list.

**#### Pseudocode**

```

QuickSort(array, low, high)

    if low < high then

        pivotIndex = Partition(array, low, high)

        QuickSort(array, low, pivotIndex - 1)  // Sort elements before the pivot

        QuickSort(array, pivotIndex + 1, high) // Sort elements after the pivot

Partition(array, low, high)

    pivot = array[high]  // Choose the pivot element (can be any element)

    i = low - 1          // Index of the smaller element

    for j = low to high - 1 do

        if array[j] <= pivot then

            i = i + 1

            swap array[i] with array[j]

    swap array[i + 1] with array[high]  // Place pivot in the correct position

    return i + 1  // Return the pivot's final index

```

*\_Quick Sort\_*: This function works by choosing a pivot and dividing the array into two parts: elements smaller than the pivot and elements greater than the pivot. The recursion continues on these smaller parts.

*\_Partition\_*: This function partitions the array around the pivot. It rearranges the array so that all elements less than the pivot are on its left, and all elements greater than the pivot are on its right. After partitioning, the pivot is placed in its correct position, and its index is returned.

**### Task:**

Implelment one of these algorithms with the programming language of your choice.