**Sorting**

**Bubble Sort**

* Not very efficient
* Time complexity – O(n^2)
* Space complexity – O(1)

**Steps in bubble sort:**

1. Compare the first two elements of the array.
2. Switch places if the value of the second element is less than the value of the first element
3. If the value of the second element is greater than the first, leave it alone.
4. Now compare the value of the second element to the third.
5. Repeat steps 2 – 4 until you have reached the end of the array.
6. Once you have reached the end of the array, repeat steps 1 – 5.

**Selection Sort**

* Time Complexity – O(n^2)
* Space Complexity – O(1)
* Looks for the smallest item in the array then places it in the first index. Then it starts from the second index and scans the array again until the smallest element is found and is placed in the second index. This process continues until the whole array is sorted.

**Steps in selection sort:**

1. Compare the first two elements of the array to determine the smallest element.
2. The smallest element gets marked with a red.
3. Now compare the next (third) element to the previous marked smallest element.
4. If it is smaller, then mark red that element as the smallest.
5. Repeat steps 3 and 4 until you have checked all elements in the array
6. Once you’ve traversed the array, move the smallest element to the beginning of the array.
7. Repeat steps 3 to 6. Each time the array is traversed move the smallest element to the next index of the array until the array is sorted.

**Insertion Sort**

* Useful when the list is almost or already sorted.
* Performs well with small data sets.
* Best case = O(n) – linear time when the list is almost sorted

**Steps in insertion sort:**

1. Start with the first element
2. Compare the next element to the first
3. If the next element is smaller than the first, switch them
4. Now let’s look at the third element
5. Compare it to the previous element
6. If that element is smaller than the previous element, move the smaller element to the correct place in the sorted list on the left.
7. Repeat this until you have traversed the list

**Merge Sort**

* Time Complexity – O(n log n)
* Space Complexity – O(n)
* Divide and Conquer technique using Recursion
* If there are duplicates, merge sort will keep the original order of the items in the array and not delete the duplicate. The duplicate that came first will be placed first before the duplicate that came after.

**Steps in merge sort:**

1. Take a list and divide it in half.
2. Then divide each halves into half.
3. Continue to divide until there is only 1 item in a set.
4. Now compare the first element to the second element and combine them into a set that is sorted.
5. Move to the third and fourth elements and do the same thing as step 4.
6. Repeat steps 4 and 5 until the list is traversed and you have an ordered pair of sets.
7. Now compare each pair of set to the other next to it to create a sorted list for each half
8. Now compare both halves and move each element in their correct spots.

**Quick Sort**

Usually the best time complexity on average

**Steps in quick sort:**

1. Choose a pivot. Let say ‘4’
2. Place everything that is less than the pivot to the left and everything greater than pivot to the right.
3. 3 is less than 4 so it stays where it is at.
4. 7 is greater than 4 so it gets moved to the right of 4.
5. We move 4 over to the left by one position and the value that 4 took over is now placed where 7 use to be.
6. Now 5 is greater than 4 so we move 5 to the right of 4 and move 4 to the left by one index and place 9 in the place where 5 use to be.
7. Repeat the steps until you have traversed the list.
8. Now we have two lists, one to the left of 4 and one to the right of 4.
9. Repeat steps 1 – 8 until each subset are sorted.
10. Assemble the list back together to get the sorted list.

A picture containing graphical user interface

Description automatically generated