Sorting is an important operation in computer science, as so many of our applications make more sense and appear more organized when things are sorted. For example, your email inbox is sorted vertically by time and date so you know when they arrived. Another example is lists of student names, which are usually sorted in alphabetical order.

What is the most intuitive way to sort an array of integers? A straightforward approach could be to look for the minimum integer and then bring it to the front of the list – keep doing this until the array is sorted in increasing order.

Here's how it looks:

```
[2, 8, 5, 3, 9, 4, 1] // Find min item, which is the 1

[1, 2, 8, 5, 3, 9, 4] // Bring it to the front of the list

[1, 2, 8, 5, 3, 9, 4] // Find min item, which is the 2

[1, 2, 8, 5, 3, 9, 4] // It's already at its place at the front

[1, 2, 8, 5, 3, 9, 4] // Find min item, which is the 3

[1, 2, 3, 8, 5, 9, 4] // Bring it to the front of the line

...

[1, 2, 3, 4, 5, 8, 9] // Eventually, list is sorted
```

However, arrays don't easily do this operation – simply picking up the min item and inserting it at its place at the front. Instead, 'moving' an item means overwriting one index with another, which would look like swapping two items in the list.

See example:

```
[2, 8, 5, 3, 9, 4, 1] // Find min item, which is the 1

[1, 8, 5, 3, 9, 4, 2] // Assign 0th index to be min and assign
final index to be what was overwritten, the integer 2
```

So swapping is how integers can be quickly 'moved' in an array.

Another point in the sorting procedure above is that after we bring a min item to the front of the line, we don't need to include it in the search for the next minimum. Once we brought the 1 to the front, we forgot about. When we brought 2 to the front, we also then left it there for the rest of the sorting.

Hence, we can think of the items already moved to the front of the array as being in a sorted partition, which is not included in the sorting anymore.

We can think of sorting an array as 'building a sorted partition' within the array, which keeps growing every time we add an item to the front. The rest of the array we can call the 'unsorted partition'.

In fact, it's like we're appending the min item from the unsorted partition to the sorted partition:

```
[2, 8, 5, 3, 9, 4, 1]  // Begin with unsorted array

                       // Move min item 1 to the front

[1] [2, 8, 5, 3, 9, 4] // Sorted and unsorted partitions

                       // Min item 2 already at front

[1, 2] [8, 5, 3, 9, 4] // Sorted and unsorted partitions

                      // Move min item 3 to the front

[1, 2, 3] [8, 5, 9, 4] // Sorted and unsorted partitions
```

Selection Sort

When we put these two concepts together – **swapping** and **partitioning**, we get Selection Sort.

This algorithm repeatedly does this process:

1. Find the minimum element in the unsorted partition
2. Swap the min element with the first item in the unsorted position ('bring it to the front')

3. Include min element in sorted partition and exclude it from unsorted partition

[2, 8, 5, 3, 9, 4, 1] // Begin with unsorted partition

// Find min item, which is the 1

[1, 8, 5, 3, 9, 4, 2] // Swap it with first item of unsorted

[*1*, 8, 5, 3, 9, 4, 2] // Sorted partition italicized

[*1*, 8, 5, 3, 9, 4, 2] // Operate only on unsorted partition

[*1*, 8, 5, 3, 9, 4, 2] // Find min item, which is the 2

[*1*, 2, 5, 3, 9, 4, 8] // Swap with first item of unsorted

[*1*, *2*, 5, 3, 9, 4, 8] // Operate only on unsorted partition

[*1*, *2*, 5, 3, 9, 4, 8] // Find min item, which is the 3

[*1*, *2*, *3*, 5, 9, 4, 8] // Swap with first item of unsorted

[*1*, *2*, *3*, 5, 9, 4, 8] // Operate only on unsorted partition

[*1*, *2*, *3*, 5, 9, 4, 8] // Find min item, which is the 4

[*1*, *2*, *3*, 4, 9, 5, 8] // Swap with first item of unsorted

...

[*1*, *2*, *3*, *4*, *5*, *8*, *9*] // Unsorted array is now sorted

Q1) Write out the steps of Selection Sort for the array [4, 2, 8, 6], following the three-step process as illustrated above. Show what the array looks like after each step. Mark the sorted and unsorted partitions.

Q2) Write pseudocode or Java code for Selection Sort

Q3) Convert the iterative Selection Sort to a recursive version.

Think about how we keep partitioning in Selection Sort – it's easy to see that the base case is when the partition is empty and the smaller case is when the recursive call is be only on the remaining unsorted partition. All we have to do is implement the swap of unsorted partition's minimum value with the unsorted partition's first element, within the body of the recursive function.

For the next sorting algorithm, Bubble Sort, we'll learn it a different way. We'll first see the pseudocode directly, then practice with example arrays. It will sort arrays in increasing order.

```
def BubbleSort(array){

n = len(array)

//do for as many array elements
for i in [0,…,n-1]:

     // go through whole array
     for j in [0,…,n-1]:

          // put compared elements in increasing order
          if array[j] > array[j+1]:
               swap array[j] and array[j+1]
}
```

Bubble Sort

Let's talk through Bubble Sort – what is the 'bubbling'? The bubbling refers to how larger numbers end up 'bubbling' to the end of the array, where they belong.

Here is an example of an array which requires the most sorting – one in purely decreasing order, the array [8, 6, 4, 2]. We will see how larger items at the front will bubble to the end.

```
BubbleSort([8,6,4,2]){
```

```
n=4 // length of array

i=0)

     j=0) Compare ar[0] to ar[1], or 8 to 6. The 8 is larger, so
     swap them, moving 8 forward. Array is [6,8,4,2]

     j=1) Compare ar[1] to a[2], or 8 to 4. The 8 is larger, so
     swap them, moving 8 forward. Array is [6,4,8,2].

     j=2) Compare ar[2] to ar[3], or 8 to 2. The 8 is larger, so
     swap them, moving 8 to the end. Array is [6,4,2,8]
}
```

Do you see how the 8 moved from the front all the way to the end? It 'bubbled' up to the end by the swaps.

In the next ith iteration, the 6 will bubble towards the end (halting before the 8), then in the next iteration the 4 will bubble towards the end (halting before the 6).

Q4) Write out the rest of the steps for i=[1,2,3], just as was done above.

Let's practice Bubble Sort again, with another example.

Q5) Write out the steps, just as in Q4, for array [7, 5, 1, 3].

Q 6) Convert Bubble Sort to its recursive version and write pseudo code for the recursive version of BubbleSort.

Merge Sort

There is another way of sorting, which isn't as straightforward as Selection Sort or Bubble Sort. Imagine an algorithm which sorts by 'breaking down and reassembling'.

What this means is we keep splitting an original array of size 'n' into halves. We keep splitting until the halves get smaller and smaller, and we're left with 'n' arrays with a single element! We then reassemble them, two at a time, joining size-1 arrays into size-2 arrays in sorted order, then joining size-2 arrays into size-4 arrays in sorted order, etc. We keep doing this until we have a size-n array, which is completely sorted!

Let's walk through the pseudocode with an example.

```
def MergeSort(array):

    n = len(array)

    // base case when array is size-1
    if n == 1:
        return array

    // Else, split current array in half

    array1 = array[0, …, $\frac{n}{2}$]
    array2 = array[$\frac{n}{2}$+1, …]

    // And split each half further into halves, until size-1

    array1 = MergeSort(array1)
    array2 = MergeSort(array2)

    // At this point, we have two sorted array halves which //…need to
    be joined in sorted order
    sorted_and_joined = Merge(array1, array2)

    return sorted_and_joined
```

Note the helper function Merge(array1, array2), which takes two arrays and sorts them simply by creating a new array, then comparing the first element of array1 and array2. Whichever first element is smaller, gets copied into the new array. Repeat this until all elements from array1 and array2 have been copied over, and the new array contains all their elements in sorted order.

Can you write this function?