CS 2383 Data Structures and Algorithms
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

Spring 2023

| | | | |
|---|---|---|---|
| **Instructor:** | Syed Eqbal Alam | **Time:** | T Th 1:00pm – 2:20pm |
| **Email:** | syed.eqbal@unb.ca | **Place:** | GWC 127, Gillin Hallway |

# Sorting Algorithms

## 1 Introduction

- Sorting algorithms are used to arrange comparable data types in increasing or decreasing order.

  For example, the Name of your friends in your contact list. It should be sorted in alphabetic order; otherwise, it will be difficult to access the names (assuming no search facility is given).

- Suppose you are given an array of characters $\{A, B, D, M, E, S\}$ to sort. The sorted array will be $\{A, B, D, E, M, S\}$.

- Suppose you are given an array of integers $\{11, 2, 33, 54, 108, 17\}$ to sort in the increasing (ascending) order. The sorted array will be $\{2, 11, 17, 33, 54, 108\}$.

  Similarly, if you are asked to sort in decreasing (descending) order, the sorted array will be $\{108, 54, 33, 17, 11, 2\}$.

There are many sorting algorithms, such as:

(i) Bubble sort

(ii) Selection sort

(iii) Insertion sort

(iv) Merge sort

(v) Quick sort

(vi) Heap sort

Each of these algorithms has advantages and disadvantages. For example, some will have better time complexity, and some will be easier to implement and suitable for a small data set.

## 2 Bubble sort

- Suppose that we are sorting an integer array in the increasing order. Bubble sort works as follows:
  - ◇ If there is only one element in the array, then nothing to do; return the array.

If there are two or more elements (say 4 elements) in the array to be sorted, then the following steps will be followed.

(Step 1) The first two elements of the array are compared. If the 1st element is greater than the 2nd element, then swap the elements. If no, then go to Step 2.

(Step 2) Compare 2nd and 3rd elements of the array. If the 2nd element is greater than the 3rd element, then swap the elements. If no, then go to Step 3.

(Step 3) Compare 3rd and 4th elements of the array. If the 3rd element is greater than the 4th element, then swap the elements. Now the biggest element is moved to the last of the array.

Now repeat Step 1, and compare 1st element and 2nd element of the array. If the 1st element is greater than the 2nd element, then swap the elements. If no then go to Step 2 and compare 2nd and 3rd elements. If the 2nd element is greater than the 3rd element, then swap the elements. Now the second bigger element is at its correct place. After this we again start Step 1 and compare 1st and 2nd elements of the array, and swap if 2nd element is greater than the 1st element. The array is sorted now.

**Algorithm 1:** Bubble sort.

```
Input: Array of length $n$, $arrayA$;
Initialization: $i = 0, j = 0$;
Output: $arrayA$;
for $i = 0; i < n; i++$ do
    for $j = 0; j < n - i - 1; j++$ do
        if $arrayA[j] > arrayA[j+1]$ then
            //Swap $arrayA[j]$ and $arrayA[j+1]$
            int $temp = arrayA[j]$;
            arrayA[j] = arrayA[j+1];
            arrayA[j+1] = temp;
        end
    end
end
```

The Bubble sort algorithm is quite old, and it is easier to implement. However, it is not suitable for very large datasets.

■ The Bubble sort has time complexity $O(n^2)$ for a given input size $n$.

## Bubble sort:

Sort the following array, A:

| | | | | |
|---|---|---|---|---|
| 12 | 1 | 9 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |

**Iteration 1:** Start from element at index 0 of the array.

Step 1: Compare A[0] and A[1].

| | | | | |
|---|---|---|---|---|
| 12 | 1 | 9 | 7 | 2 |

As 12>1, Swap 12 and 1, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 12 | 9 | 7 | 2 |

Step 2: Compare A[1] and A[2].

| | | | | |
|---|---|---|---|---|
| 1 | 12 | 9 | 7 | 2 |

As 12>9, Swap 12 and 9, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 12 | 7 | 2 |

Step 3: Compare A[2] and A[3].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 12 | 7 | 2 |

As 12>7, Swap 12 and 7, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 12 | 2 |

Step 4: Compare A[3] and A[4].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 12 | 2 |

As 12>2, Swap 12 and 2, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 2 | 12 |

The greatest element (12) is at its correct place (A[4]).

**Iteration 2:** Again, start from element at index 0 of the array.

Step 1: Compare A[0] and A[1].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 2 | 12 |

As 1<9, No Swap, move to the next step

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 2 | 12 |

Step 2: Compare A[1] and A[2].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 7 | 2 | 12 |

As 9>7, Swap 9 and 7, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 9 | 2 | 12 |

Step 3: Compare A[2] and A[3].

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 9 | 2 | 12 |

As 9>2, Swap 9 and 2, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 2 | 9 | 12 |

The second greatest element (9) is at its correct place (A[3]).

**Iteration 3:** Again, start from element at index 0 of the array.

Step 1: Compare A[0] and A[1].

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 2 | 9 | 12 |

As 1<7, No Swap

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 2 | 9 | 12 |

Step 2: Compare A[1] and A[2].

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 2 | 9 | 12 |

As 7>2, Swap 7 and 2, updated array will be

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 7 | 9 | 12 |

Now, 7 is at its correct place (A[2]).

**Iteration 4:** Again, start from element at index 0 of the array.

Step 1: Compare A[0] and A[1].

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 7 | 9 | 12 |

As 1<2, No Swap

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 7 | 9 | 12 |

The array is sorted now.

Notice that in the first iteration there are 4 steps (array length-1); in the second iteration there are 3 steps, in the third iteration there are 2 steps, and in the fourth iteration there is 1 step and at each iteration one element is sorted and placed at correct location.

Try to solve these questions:

**Question 2.1.** *Sort arrayA = {11, 11, 22, 3, 6, 9, 1} through the Bubble sort algorithm. Write all the iterations and steps (compare and swap) in each iteration.*

**Question 2.2.** *Sort arrayB = {22, 11, 11, 9, 6, 3, 1} through Bubble sort algorithm. Check how many iterations and steps (compare and swap) are required to sort the array.*

**Question 2.3.** *Sort arrayC = {1, 3, 6, 9, 11, 11, 22} through the Bubble sort algorithm. Check how many iterations and steps (compare and swap) are required to sort the array? Additionally, compare the number of operations required to sort the arrays in Question 2.1, Question 2.2, and Question 2.3.*

**Question 2.4.** *Sort arrayA = {A, E, M, V, B, D, K} through Bubble sort algorithm. Write all the iterations and steps (compare and swap) in each iteration.*

**Question 2.5.** *Implement Bubble sort algorithm to sort arrayA = {ABC, AEC, MAB, BVV, AAB, D, FED, NB,CA}. Print all the iterations and steps (compare and swap) in each iteration.*

## 3 Insertion sort

■ An element is put in the correct place compared to the elements before it.

Steps to follow for Insertion sort:

(i) The first element is sorted trivially.

(ii) If there are two or more elements in the array, then compare the second element $A[1]$ with the first element $A[0]$. Assign $A[1]$ to *Key*. If $A[0]$ is greater than the Key, then move $A[0]$ to the index 1 of the array and put the Key (second element) at the index 0 (at the place of $A[0]$). If there are two elements in the array, then the array is sorted now. If there are more elements in the array, then move to the next element in the array and compare it with its previous elements, as described in the following steps.

(iii) Compare the 3rd element $A[2]$ with the 2nd element $A[1]$ and then with the first element $A[0]$. Now, Key $= A[2]$. If $A[1]$ is greater than $A[2]$, then move $A[1]$ to the place of $A[2]$. Now, compare the Key with the 1st element of the array $A[0]$, if $A[0]$ is greater than the Key, then move $A[0]$ to index 1 and place the key at index 0 (at the place of $A[0]$).

(iv) Repeat this process until you reach the last element of the array. Finally, you will get a sorted array.

## Insertion sort:

### Sort the following array, A:

| | | | | |
|---|---|---|---|---|
| 12 | 1 | 9 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |

**Iteration 1:**

If just one element in the array, then return the array, nothing to sort. If there are two or more than two elements in the array then follow the below steps.

Step 1: Take second element of the array as Key; thus, in the example, Key= A[1] =1, now compare A[1] with A[0], if A[0] is greater than A[1], then move A[0] to A[1], and place Key at A[0].

| | | | | |
|---|---|---|---|---|
| 12 | 1 | 9 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |

Since 12>1, move 12 to A[1] then place Key=1 to A[0]. We get the following array

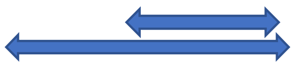
| | | | | |
|---|---|---|---|---|
| 1 | 12 | 9 | 7 | 2 |

**Iteration 2:**

Step 1: As there are more elements to sort, increase the index to 2, now Key =A[2]=9.

Now, compare Key with elements before it that is compare Key=9 with A[1] and A[0].

| | | | | |
|---|---|---|---|---|
| 1 | 12 | 9 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |



Step 2: Compare Key with A[1], since 12>9, move 12 to A[2].

| | | | | |
|---|---|---|---|---|
| 1 | | 12 | 7 | 2 |

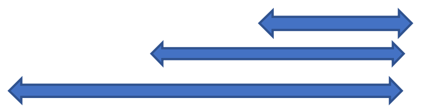
Now, compare Key with A[0]. Since 1<9, place Key=9 to A[1]. We get the following array

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 12 | 7 | 2 |

**Iteration 3:**

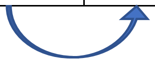
Step 1: More elements to sort, increase the index to 3 now, assign A[3] to Key, that is Key = A[3] = 7. Now, compare Key with all the elements before it that is A[2], A[1], and A[0].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | 12 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |



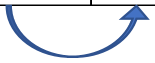
Step 2: Compare Key=7 with A[2], since 12>7, move 12 to A[3].

| | | | | |
|---|---|---|---|---|
| 1 | 9 | | 12 | 2 |



Step 3: Compare Key=7 with A[1], since 9>7, move 9 to A[2].

| | | | | |
|---|---|---|---|---|
| 1 | | 9 | 12 | 2 |



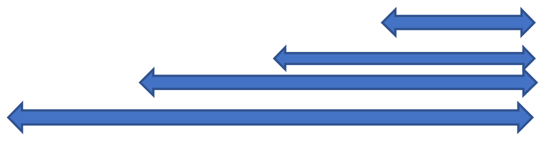
Step 4: Compare Key=7 with A[0], since 1<7, place Key at A[1]. We get the following array

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 9 | 12 | 2 |

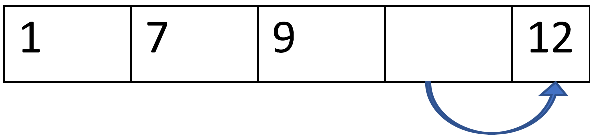
**Iteration 4:**

Step 1: One more element to sort, increase the index to 4, assign A[4] to Key, that is Key = A[4]= 2. Now, compare Key with all the elements before it that is A[3], A[2], A[1], and A[0].

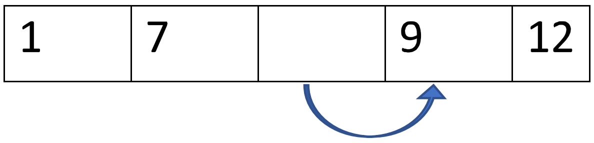
| | | | | |
|---|---|---|---|---|
| 1 | 7 | 9 | 12 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |



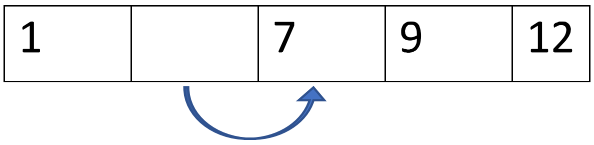
Step 2: Compare Key=2 with A[3], since 12>2, move 12 to A[4].



Step 3: Compare Key=2 with A[2], since 9>2, move 9 to A[3].



Step 4: Compare Key = 2 with A[1], since 7>2, move 7 to A[2].



Step 4: Compare Key = 2 with A[0], since 1<2, place Key at A[1].

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 7 | 9 | 12 |

We checked all the indices 0 to 4 of the array, the array is sorted now.

**Algorithm 2:** Insertion sort.

```
Input: Array of length $n$, $arrayA$;
Initialization: $i = 1$;
Output: Sorted array $arrayA$;
for $i = 1; i < n; i++$ do
    $Key = arrayA[i]$;
    $j = i - 1$;
    while $j >= 0$ && $arrayA[j] > Key$ do
        //Move the $j$'th index element, $arrayA[j]$ to $(j + 1)$'th index of $arrayA$.
        $arrayA[j + 1] = arrayA[j]$;
        $j = j - 1$;
    end
    $arrayA[j + 1] = Key$;
end
```

- The Insertion sort has time complexity $O(n^2)$ for a given input size $n$.
- Using the insertion sort, a sorted array such as $A = \{1, 2, 3, 4, 5\}$ for increasing order will have time complexity $O(n)$.
- A reverse sorted array such as $A = \{5, 4, 3, 2, 1\}$ for increasing order will have time complexity $O(n^2)$. This is the upper bound of time complexity for the Insertion sort.

## 4 Merge sort

- It is a **Divide and Conquer**-based algorithm.
- In the divide and conquer-based algorithms, a problem is **split or divided** into smaller problems or sub-problems. Then the solutions to sub-problems are combined to form the solution to the bigger problem.

The following steps are followed to perform merge sort.

(i) If there is just one element in the array, then return the array, nothing to sort.

(ii) If there are two or more than two elements in the array, then find out the total number of elements in the array, $n$. Then calculate the mid value of the array length, $mid = n/2$.

(iii) Create two arrays, $LeftArray$ and $RightArray$. Copy the elements from index 0 to $mid-1$ of $arrayA$ in the $LeftArray$ and the elements from index $mid$ to $n-1$ of $arrayA$ in the $RightArray$.

(iv) Now pass the $LeftArray$ to the MergeSort method, calculate the mid value of the $LeftArray$ length and create two arrays, $LeftArray$ and $RightArray$. Copy the elements from index 0 to $mid - 1$ of $arrayA$ in the $LeftArray$ and the elements from index $mid$ to $n - 1$ of $arrayA$ in the $RightArray$. As this is a recursive function call, this process repeats until all the arrays have one-one elements.

(v) Now, the comparison of elements and merging process will start from the last level, moving upward (Left to Right).

(vi) Compare each element of the $LeftArray$ and $Rightarray$ and place elements at their correct locations in the merged array.

- Merge sort has time complexity $O(n \log n)$ for a given input size $n$.

## Merge sort:

### Sort the following array, A, using merge sort:

If there is just one element in the array, then return the array, nothing to sort. If there are two or more than two elements in the array, then follow the below steps.

| 12 | 1 | 9 | 7 | 2 |
|---|---|---|---|---|
| A[0] | A[1] | A[2] | A[3] | A[4] |

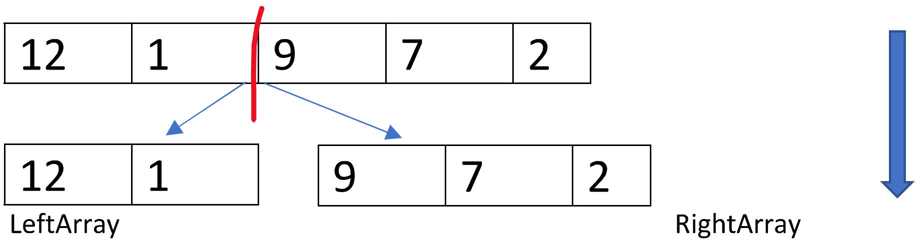
Array length, n = 5.

**Iteration 1:**

Calculate the total number of elements in the array (array length), n. Then calculate the mid value of the array length, mid = n/2. In the example, n=5; thus, mid=5/2=2.

| 12 | 1 | 9 | 7 | 2 |
|---|---|---|---|---|
| A[0] | A[1] | A[2] | A[3] | A[4] |

Now, split array A into two subarrays, LeftArray and RightArray. Copy the elements from index *0 to mid-1* of array A in the LeftArray and the elements from index *mid to n-1* of array A in the RightArray.

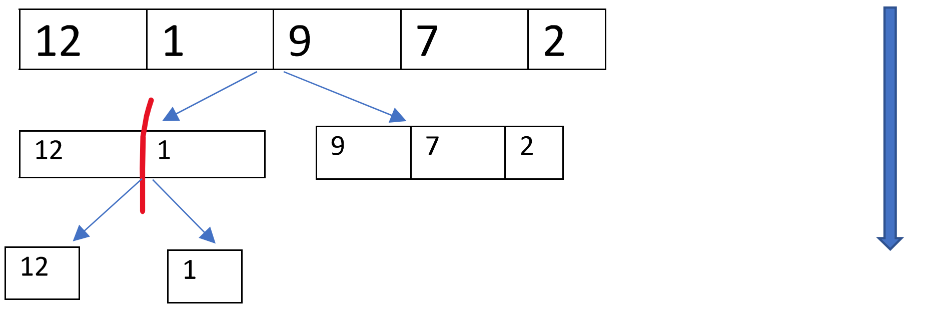


**Iteration 2:** Now, LeftArray {12, 1} will be passed to the Mergesort method.

Length of the LeftArray {12, 1}=2.

Mid value of the length of the LeftArray {12, 1}, mid = 2/2 = 1.

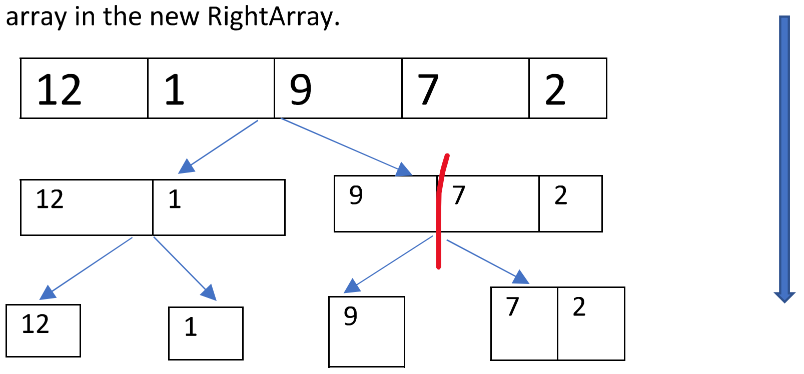
Now, split the array {12, 1} into two subarrays. New LeftArray will contain index *0 to mid-1* (one element, 12) of the array {12, 1} and the elements from index *mid to n-1* (one element, 1) of the array in the new RightArray. Thus, LeftArray ={12} and RightArray ={1}.



**Iteration 3:** Now, array {9, 7, 2} will be passed to the Mergesort method. Length of the array {9, 7, 2} = 3.

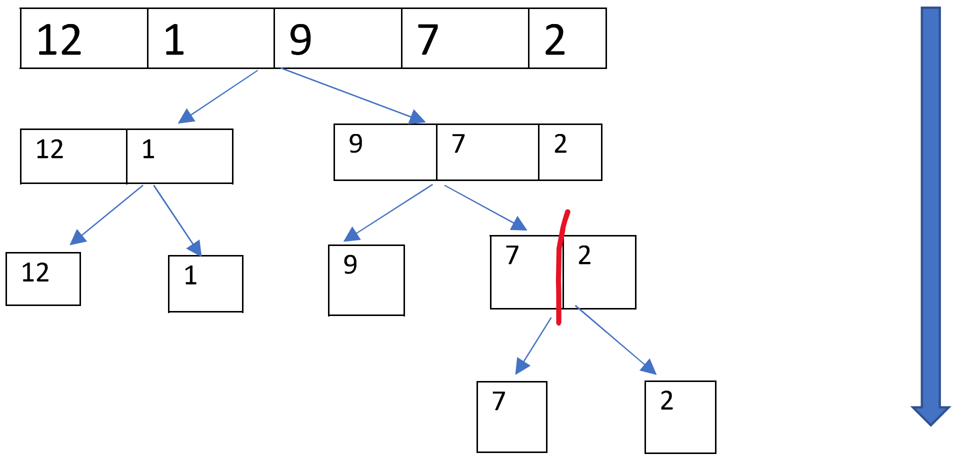
Mid value of the array length of {9, 7, 2} =3/2=1.

Now, split array {9, 7, 2} into two subarrays. Copy the elements from index *0 to mid-1 (one element)* of array {9, 7, 2} in the new LeftArray and copy the elements from index *mid to n-1 (two elements)* of the array in the new RightArray.



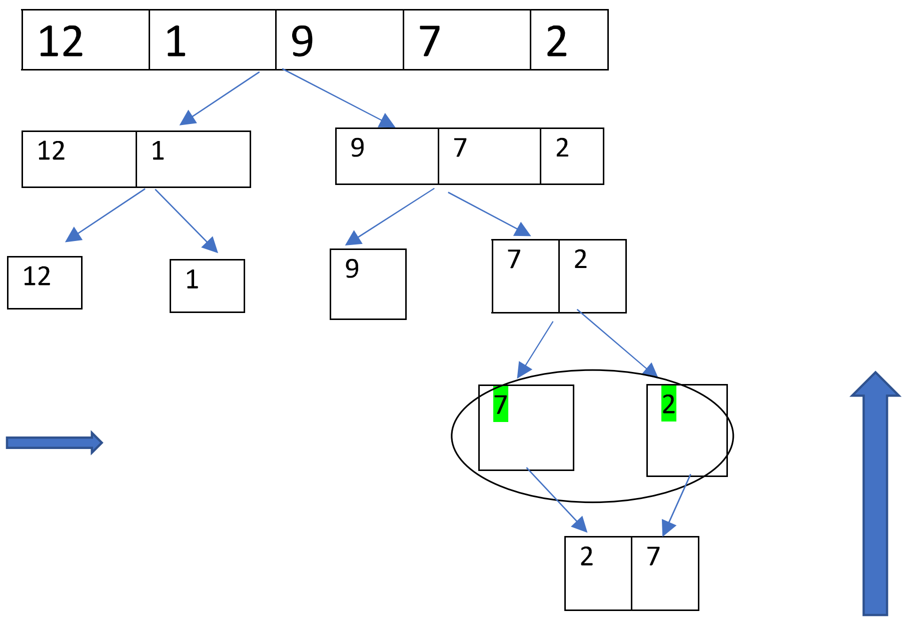
The LeftArray with element 9 has one element, nothing to do.

**Iteration 4:** Move to the RightArray {7, 2}, the array {7,2} will be passed to the Mergesort method now. The mid value of the array length of {7, 2} is 2/2=1. Now, split the array into two subarrays, LeftArray and RightArray. The new LeftArray is {7} and the new RightArray is {2}.

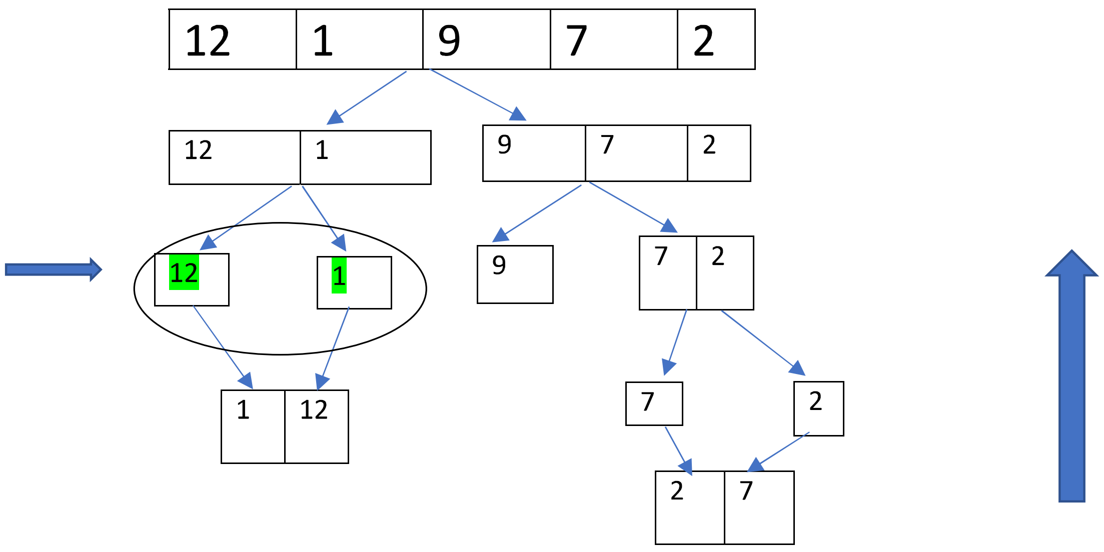


**The arrays are split into one-one elements. The comparison of elements and merging (Conquer) process will start from the last level, moving upward (Left to Right).**

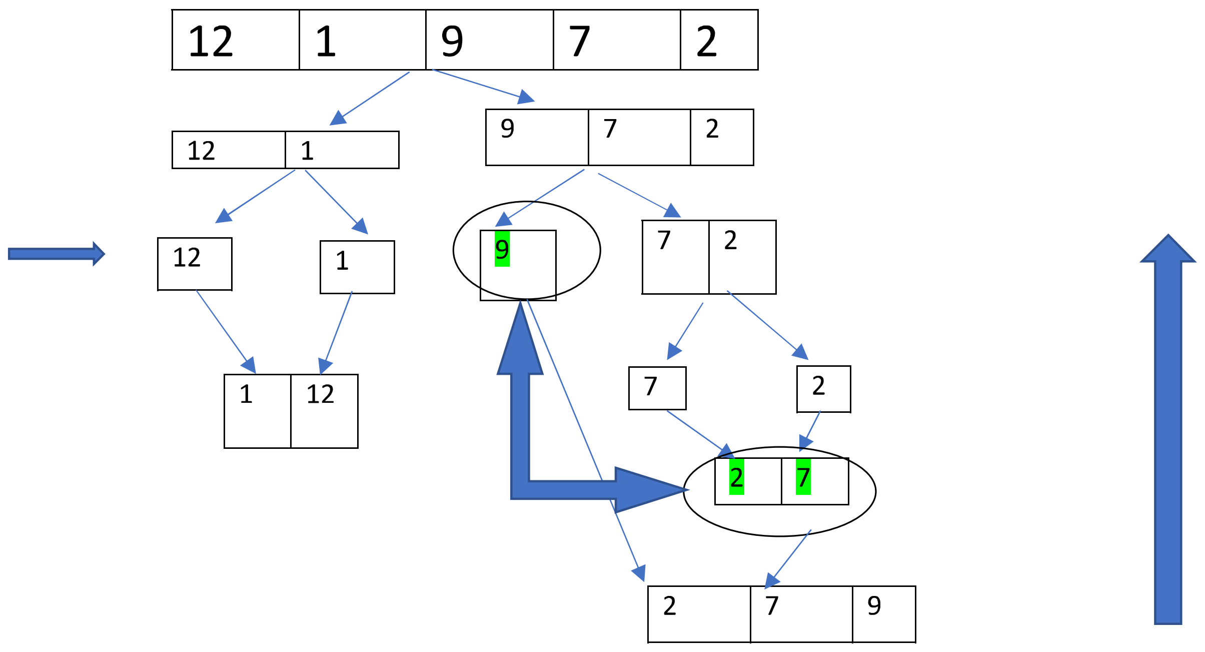
**MergeArrays:** Arrays {7} and {2} are at the lowest level, compare 7 and 2; as 7>2, swap 7 and 2, and merge the arrays. We obtain the array {2,7}.



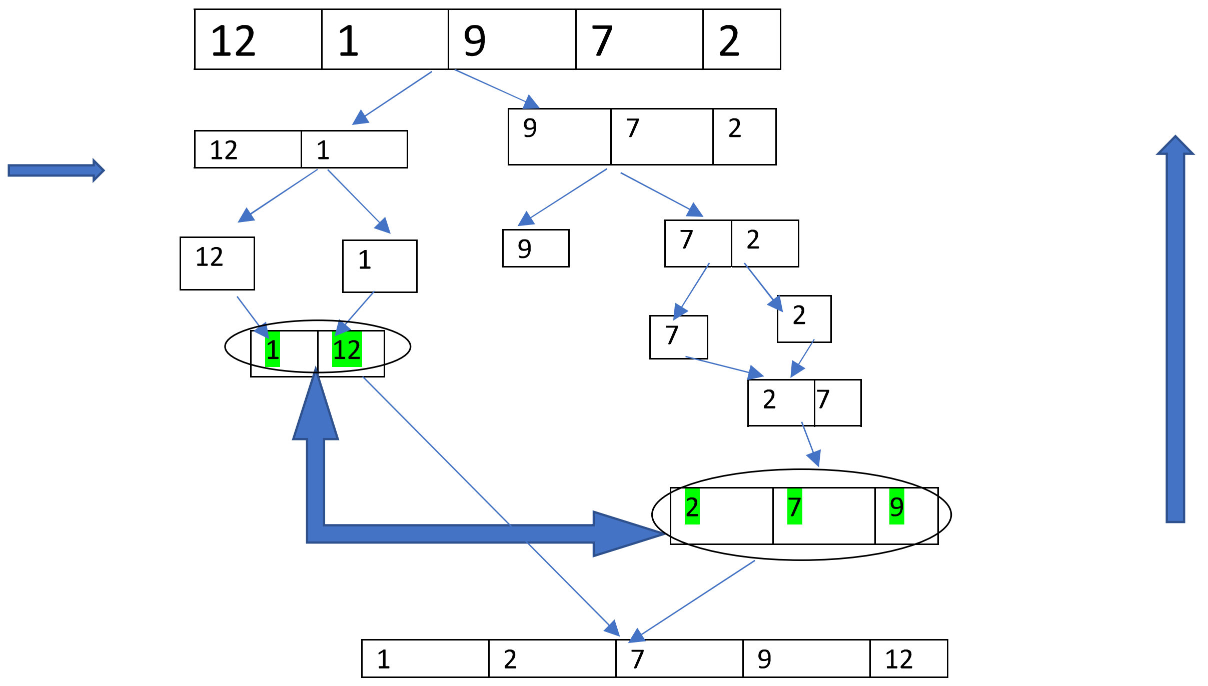
Now move one level up; compare the left-side arrays {12} and {1}, sort them and merge them. Compare 12 and 1, as 12>1, swap 12 and 1, and merge the arrays to obtain {1, 12}. Then move to the right-side arrays, follow the same steps.



Moving to the right-side, compare 9 with the array {2,7}, as 9>2, place 2 at index 0 of the merged array, then compare 9 with 7, as 9>7, place 7 at index 1 and then place 9 at index 2 of the merged array.



Now move a level up; there are two arrays {1,12} and {2,7,9} to compare and merge. Compare each element of the array {1, 12} with {2, 7, 9}, and place elements at the correct locations. Compare 1 and 2, as 1<2, place 1 at index 0 of the merged array. Since 1 is smaller than the first element of the array {2,7,9}, so we move to the next element of the LeftArray {1, 12} and compare it with all elements of {2,7,9}. Thus, compare 12 with 2; as 12>2, place 2 at index 1 of the array. Then compare 12 with 7, as 12>7, place 7 at index 2. Next, compare 12 with 9; as 12>9, place 9 at index 3. No more elements to compare. Finally, place 12 at index 4 of the merged array.



We obtained the sorted array.

The algorithm for merge sort is presented in Algorithm 3 .

**Algorithm 3:** Mergesort Algorithm.

```
Input: Array of length $n$, $arrayA$;
Initialization: $i = 1$;
Output: Sorted array $arrayA$;
MergeArrays(int [] LeftArray, int [] RightArray, int [] arrayA)):
    $n = arrayA.length$;
    $n1 = LeftArray.length$;
    $n2 = RightArray.length$;
    $i = 0$;
    $j = 0$;
    $t = 0$;
    while $i < n1$ && $j < n2$ do
        if $LeftArray[i] <= RightArray[j]$ then
            $arrayA[t] = LeftArray[i]$;
            $i = i + 1$;
        end
        else
            $arrayA[t] = RightArray[j]$;
            $j = j + 1$;
        end
        $t = t + 1$;
    end
    while $i < n1$ do
        $arrayA[t] = LeftArray[i]$;
        $i = i + 1$;
        $t = t + 1$;
    end
    while $j < n2$ do
        $arrayA[t] = RightArray[j]$;
        $j = j + 1$;
        $t = t + 1$;
    end
    return $arrayA$;
MergeSort(int [] arrayA)):
    if $n$=1 then
        return $arrayA$;
    end
    $mid = \frac{n}{2}$ ; // calculate the mid index of $arrayA$, it returns the floor value $\lfloor \frac{n}{2} \rfloor$.
    Create two arrays:
    $LeftArray$ of length $mid$ and $RightArray$ of length $n - mid$;
    Copy elements 0 to $mid - 1$ of $arrayA$ to $LeftArray$.
    Copy elements $mid$ to $n - 1$ of $arrayA$ to $RightArray$.
    MergeSort(LeftArray);
    MergeSort(RightArray);
    MergeArrays(LeftArray, RightArray, arrayA);
```

## 5 Quick sort

- Quick sort is also a **Divide and Conquer**-based algorithm.

- As noted previously, in the divide and conquer-based algorithms, a problem is **split or divided** into smaller problems or sub-problems. Then the solutions to sub-problems are combined to form the solution to the bigger problem.

- In the quick sort, an element of the sequence or array is chosen as a *Pivot*, then smaller elements than the Pivot is moved to the left-side of the Pivot and greater elements are moved to the right-side. The step is repeated until one-one elements remain in each array (visualize it as leaf nodes of a tree). Then merging process starts, left-side elements are copied first then the Pivot and then right-side elements are copied. Merging starts at lowest level elements, and moving upward (left, Pivot, right).

- Pivot elements can be:
  - The first element of the array.
  - The last element of the array.
  - The mid element of the array.
  - or a randomly chosen element of the array, following certain probability distribution.

- Quick sort has time complexity $O(n^2)$ for a given input size $n$.

- Quick sort has lower bound time complexity $\Omega(n \log n)$ for a given input size $n$.

- Quick sort has average bound time complexity $\Theta(n \log n)$ for a given input size $n$.

## Quick sort:

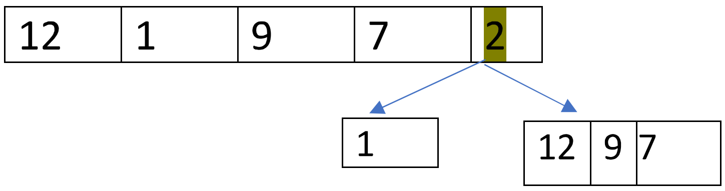
### Sort the following array, A, using Quick sort:

If there is just one element in the array, then return the array, nothing to sort. If there are two or more than two elements in the array, then follow the below steps.

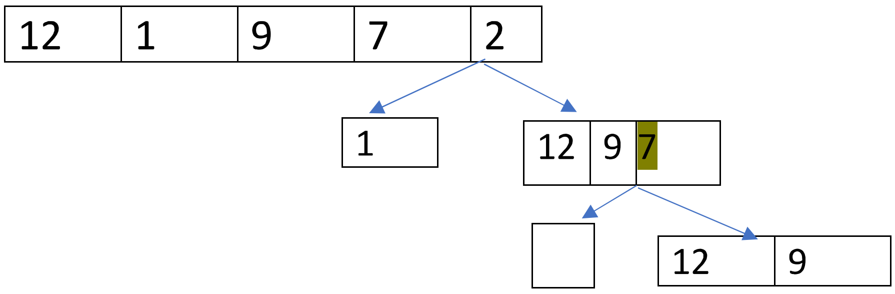
| | | | | |
|---|---|---|---|---|
| 12 | 1 | 9 | 7 | 2 |
| A[0] | A[1] | A[2] | A[3] | A[4] |

Let us choose the last element as the Pivot element of the array. In the example, Pivot =2.

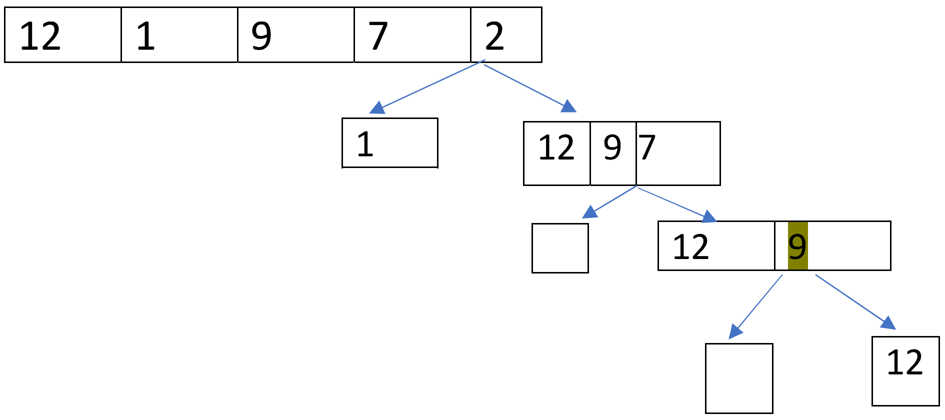
**Iteration 1:** Pivot =2, copy the elements smaller than 2 to the Left array and copy the elements greater than 2 to the Right array. Now, left array will be {1} and Right array will be {12,9,7}.



**Iteration 2:** Array {1} has just one element, now move to Right array {12,9,7}. Choose the last element of the array as Pivot, here Pivot =7, copy the elements smaller than 7 to the Left array (no element) and copy the elements greater than 7 to the Right array.

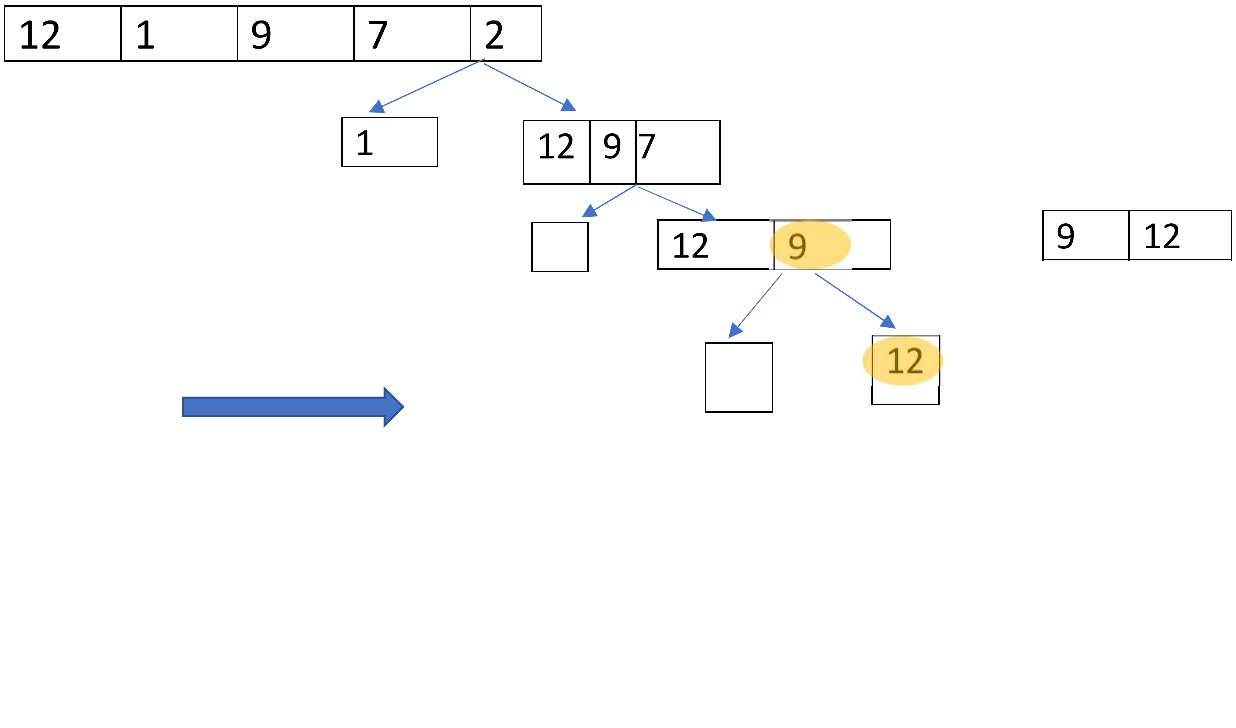


**Iteration 3:** Left array {} has no element, now move to Right array {12, 9}. Choose the last element of the array as Pivot, here Pivot = 9, copy the elements smaller than 9 to the Left array (no element) and copy the elements greater than 9 to the Right array.

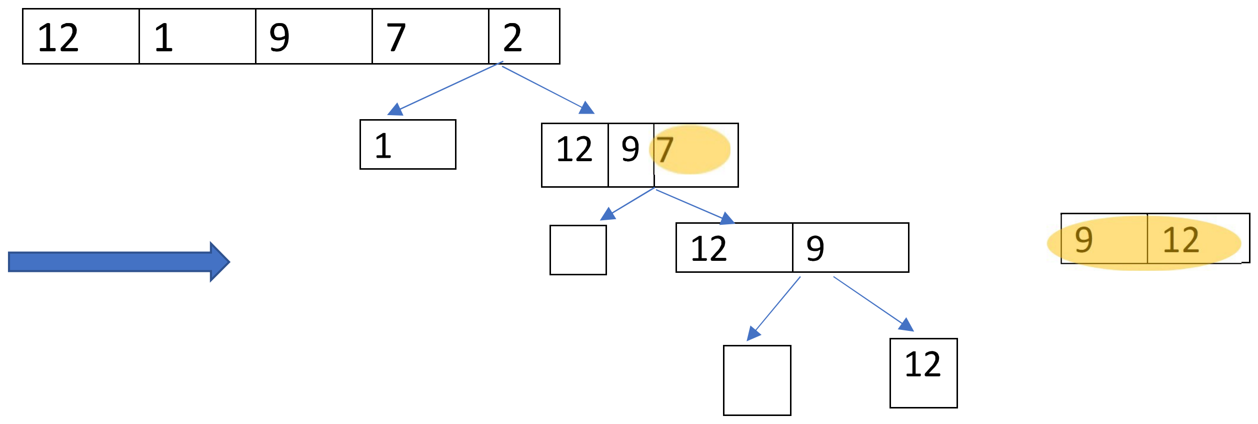


**Now the merging process starts from the lowest level moving upward.**

At the lowest level, Left array {} has no element, check the Pivot 9 and Right array {12}. Merge the arrays: Left array, Pivot, Right array. Thus, the merged array will be {9, 12}.



Moving up, left array {} has no element; now check Pivot=7 and Right array {12, 9}. Merge the arrays: Left array, Pivot, Right array. Thus, the merged array will be {7, 9, 12}.



Moving up, left array {1} has one element, Pivot =2, and Right merged array is {7, 9, 12}. Merge all the arrays: Left array, Pivot, Right array. Thus, the merged array will be {1, 2, 7, 9, 12}.

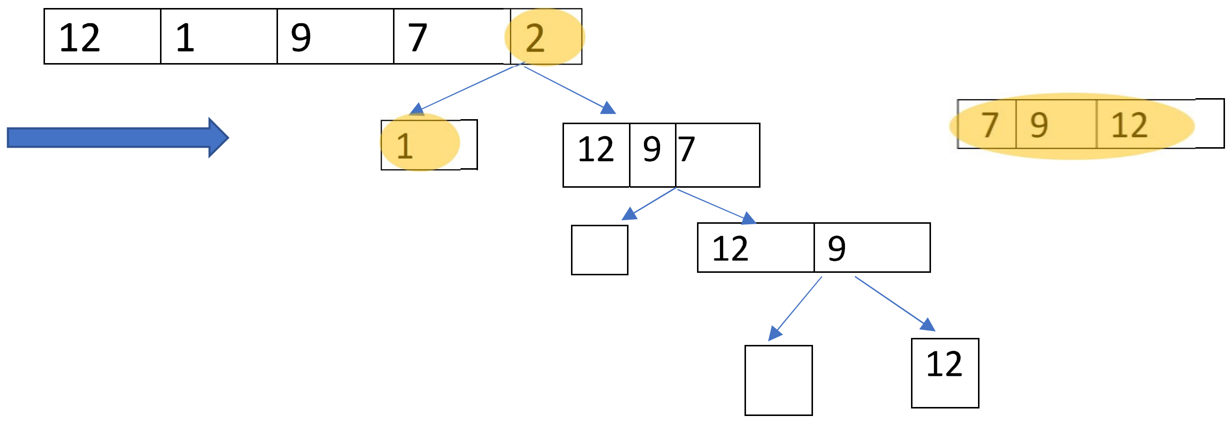


Figure 4: Quick sort.

## 6 Stable sorting algorithms

Stable algorithms retain relative order of elements in the sorted sequence.
Some of the stable sorting algorithms are:

- Insertion sort
- Bubble sort
- Merge sort

Unstable sorting algorithms are:

- Quick sort.

## 7 In-place sorting algorithms

In in-place algorithms, additional space or memory is required; nevertheless, a small extra amount is tolerated for swap and method calls.
Some of the in-place sorting algorithms are:

- Bubble sort
- Insertion sort
- Quick sort.

Not in-place sorting algorithms:

- Merge sort, Merge sort takes of the order of $O(n)$ extra space.

Quick sort is considered as an in-place algorithm; however, it takes a few additional space.
**Reference book:** [1]

## References

[1] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Java.* Wiley, 6th edition, 2014. 20