Chapter 10 | Sorting and Searching

Java AP

Sorting

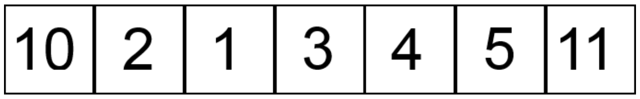
In this section we will take a look at five algorithms that can be used to sort data. What does it mean to sort data? Sorting data means that we put all the elements of an array or list in some sort of order. There are two general orderings for sorted data, ascending (smallest to largest) and descending (largest to smallest). The examples in this chapter will only sort arrays of data in an ascending manner.
Sorting algorithms are usually divided into two categories:

1. **Incremental**: Looping over the data
2. **Divide-and-Conquer**: Divide the data into chunks and work on those chunks. Generally, Divide-and-Conquer algorithms are the fastest but are more complicated and usually implemented in a recursive manner.

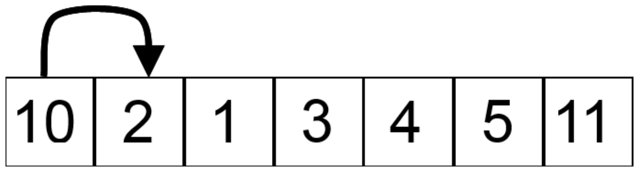
Bubble Sort

Bubble sort is one of the simplest yet least efficient sorting algorithms. The way it works is by going through the array of data, comparing each pair of adjacent elements and swapping them if they are in the wrong order. Passing through the array is repeated until no swaps are made, which indicates and guarantees that the array is sorted.

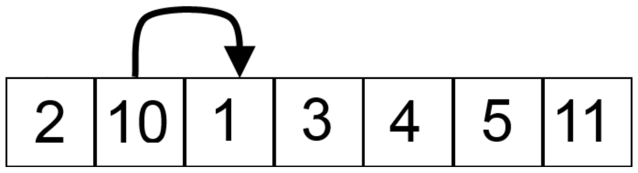
<u>Step by step example:</u>
Originally, we have an integer filled array. In this case, the array is not sorted so we will apply the bubble sort algorithm on it in order to end up with the required sorted array.



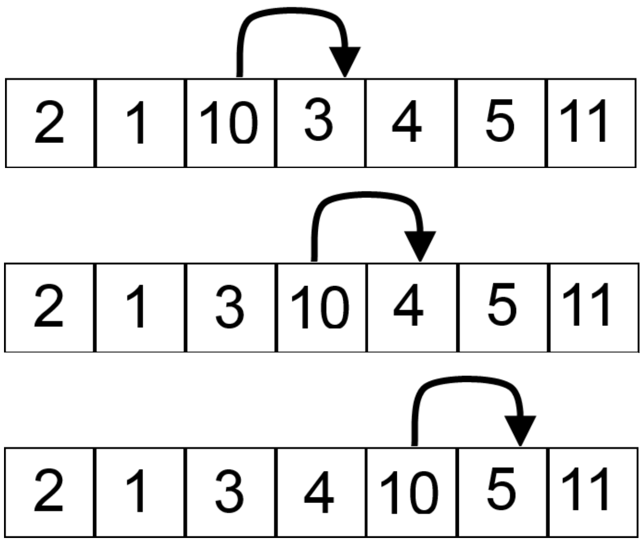
The first step is to compare the value at index 0 to the value at index 1. If the value at index 0 is greater than the value at index 1 then we need to swap the two elements, otherwise we leave them as is and continue through the array. In this case, 10 is greater than 2 so the values need to be swapped.



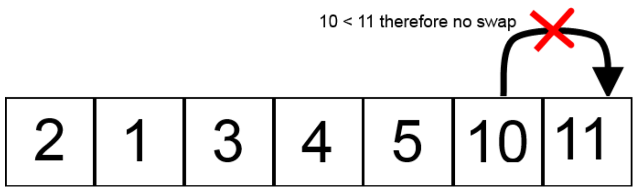
Next we compare index 1 and index 2. Again, the lower index value is greater than the higher index value, 10 greater than 1, so we swap the values.



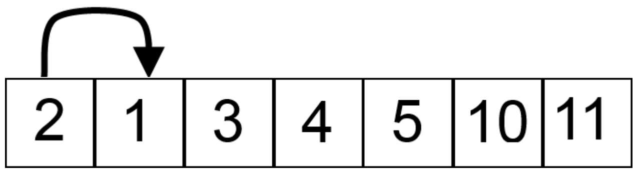
The same behavior continues until 10 is next to 11.



When 10 is compared to 11, the algorithm obviously shouldn't swap the values since they are in order. If swapping actually occurs in that situation, the algorithm will never end.



We have now reached the end of the array, however as we made swaps it is still possible that the data is not sorted. As such, we go back to the beginning of the array and start another pass.



Starting at index 0 and 1 again, the value 2 found at index 0 has to be swapped with the value 1 found at index 1. The pass continues till the end without any additional swaps.

Although it is obvious for us that the array is sorted, because at least one swap has been made in the last pass, the algorithm cannot stop and will run another pass which will end with no swaps made.

Now that a full pass occurred without any swaps made, the algorithm knows that the array is definitely sorted.

Algorithm	`public static void bubbleSort(int Array[])`
	`{`
	`    boolean swap = false;`
	`    do`
	`    {`
	`        swap = false;`
	`        for(int i = 0; i < (Array.length - 1); ++i)`
	`        {`
	`            if(Array[i] > Array[i + 1])`
	`            {`
	`                int temp = Array[i];`
	`                Array[i] = Array[i+1];`
	`                Array[i+1] = temp;`
	`                swap = true;`
	`            }`
	`        }`
	`    }`
	`    while(swap);`
	`}`

Best Case	Average Case	Worst Case
O(n)	**$O(n^2)$**	**$O(n^2)$**

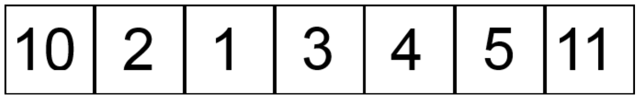
Note: ***The best case for bubble sort occurs when the data is already sorted as we only run though the array once with no swaps and then stops. The worst case occurs when the data is sorted in the reverse order.***

Selection Sort

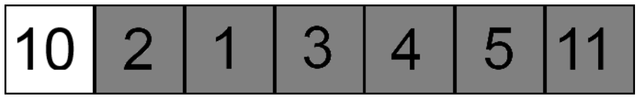
Selection sort is another simple but not that efficient sorting algorithm. Though, it almost always outperforms bubble sort.

Instead of bubbling the large values up, the selection sort is based on searching the values to the right of an index for the minimum value smaller than the index's value. If a smaller value was found, the index's value and the smallest value are swapped. We keep repeating the same steps for each index from 0 to (array length - 1).

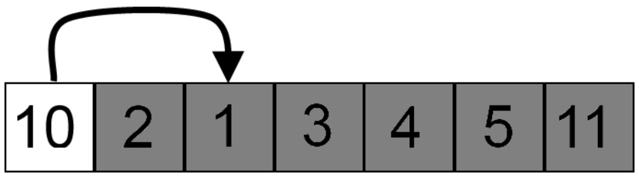
Step by step example:



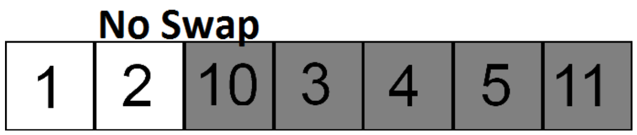
Starting at index 0, we loop through all the values from index 1 to 6 (which is array length - 1) in the search of a value smaller than 10.



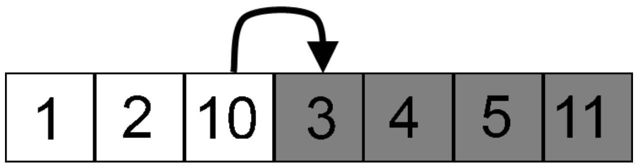
In this case, the value 1 (found at index 2) is the lowest value smaller than 10 so we swap the two values.

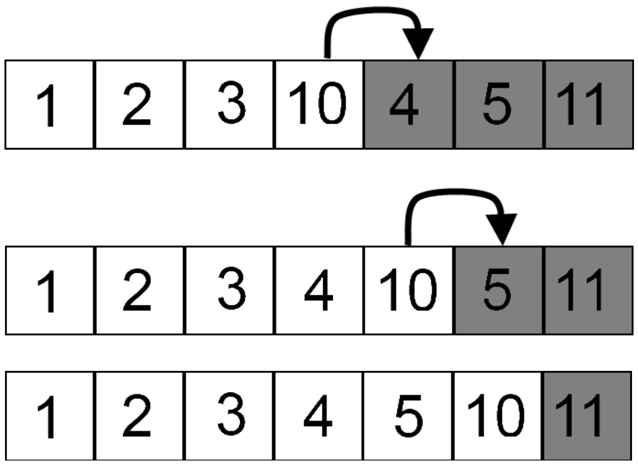


When swapped, the value 1 is guaranteed to be in its final position, that is why we can move our index to the value adjacent to it and redo the same step.
In this pass, we loop through all the values from index 2 to 6 but will not find any value smaller than the index's value 2. In this case, no swaps are made which indicates that the value 2 is in its final position. The index is moved to the adjacent value.



This process continues until the array is sorted.





The array is guaranteed to be sorted once the index reaches the end of the array.

Pseudo-Code

```
• loop through all the elements starting with i = 0
    ○ Starting j at i, loop through the array and find the index of
      the minimum value in the array
    ○ Swap the value at index i with the value at the minimum's
      index
```

Best Case	Average Case	Worst Case
$O(n^2)$	**$O(n^2)$**	**$O(n^2)$**

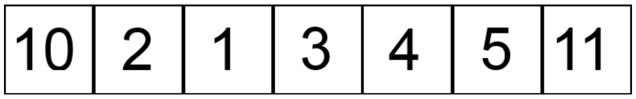
Insertion Sort

The last incremental sorting algorithm that we will cover is Insertion Sort. Insertion sort is very simple to implement, very stable and quite efficient for small, nearly sorted, data sets. The larger the data set the less efficient it becomes.

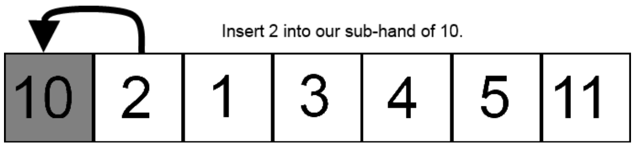
You can think of this algorithm like sorting a hand of cards. Starting at card 2, you compare it with all the cards to its left (which at first is only card one) and insert it in its proper sorted location. Next you move to card 3 and compare it to the first 2 cards and figure out where to insert it in order for it to be in its proper sorted position. If you keep going till the last card, you will end up with a sorted hand.

<u>Step by step example:</u>

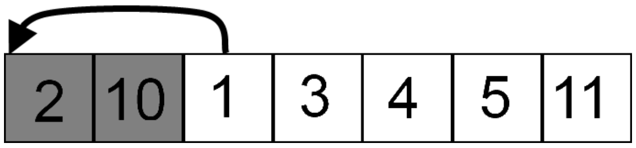
Our array originally contains the following unsorted values.



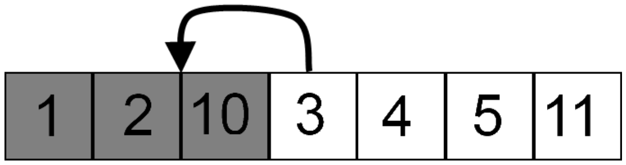
We first start at index 1, which in this case contains the value 2. Our sub-hand will be all the indices below 1, which at first is the index 0 that contains the value 10. Now, it is time to insert 2 into that sub-hand, which is done by moving 10 one index to the right and replacing it by 2 (of course, that happened because 10 is greater than 2).

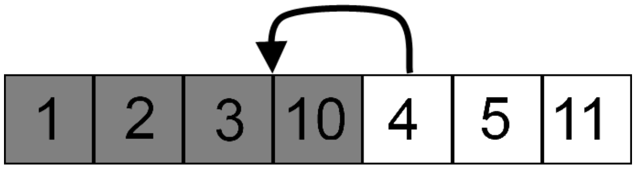


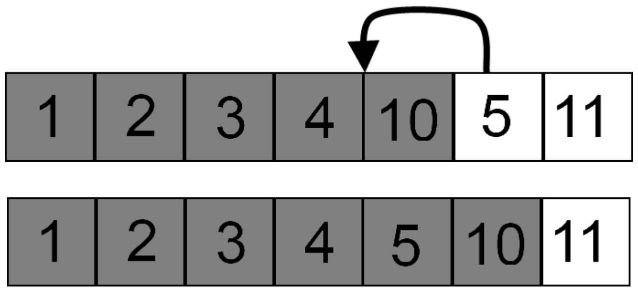
As we can see in the image below, the correct position for the value 2 is to the left of 10. Now we do the same thing with our index being 2 and our sub-hand being all the indices below 2. That leads to placing the value 1 before 2.



This process continues for the rest of the array.







At this point the array is sorted. As you may notice, every time we need to insert a value in its correct position we need to move all the elements between its insertion location and itself by one position to the right.

Best Case	Average Case	Worst Case
O(n)	**O(n^2)**	**O(n^2)**

Quick Sort (Recursive)

The Quicksort algorithm is a widely used algorithm that follows the divide and conquer technique.

Divide and conquer algorithms are formally described in three steps that are common to all divide and conquer algorithms.

1. **Division**: the algorithm divides the list of data into smaller sub-lists.
2. **Conquering**: The subdivisions are sorted recursively by applying the algorithm again to them
3. **Combining**: The subdivision are merged into a single sorted list

The Quicksort is considered to be a general purpose algorithm since it works well in a variety of situations. In addition, Quicksort consumes less resources than other algorithms.

Finally, the algorithm has a short inner loop. It is recursive and has different order of growth depending on the data. The worst case of Quicksort is quadratic O(n^2), while its average case is O(n log(n)).

First, we divide the element into two roughly equal groups. Second, we sort the two groups by a recursive call. Third, we combine the two sorted groups into a single sorted list.

A value C called the pivot is used to compare the elements in one partition. Then the partition is divided into two sub-partition where the left partition contains all the elements less then C and the right partition contains all the elements greater or equal to C. Using recursive calls, we apply the same steps for each new partition until the array is sorted.

The above explanation is correct but definitely not enough for you to implement the algorithm yet. So let us change it to pseudo-code.

Pseudo-Code

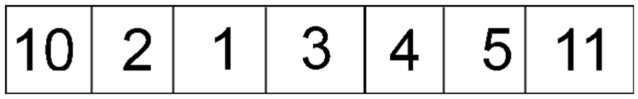
```
void quickSort(int Array[], int left, int right)
{
      o  if(left < right)
            ▪  Rearrange elements around the pivot. Values smaller than the
               pivot go to its left and values greater than the pivot go to
               its right.
            ▪  Recursive call of the quickSort algorithm on the sub-array
               containing all the elements smaller than the pivot value:
               quickSort(Array, ...)
            ▪  Recursive call of the quickSort algorithm on the sub-array
               containing all the elements greater than the pivot value:
               quickSort(Array, ...)
}
```

The only part in the above pseudo-code that should still be ambiguous is rearranging the elements around the pivot. Here are the steps that should be followed in order to rearrange the elements:

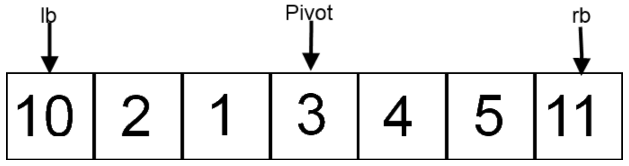
1. Choose the pivot value that will be the element to be placed in its final position
2. While the left index is less than the right index
   i. Scan from the left end of the list until an element greater than or equal to the pivot is found
   ii. Scan from the right end of the list until an element less than or equal to the pivot is found
   iii. Swap the out of place elements
   iv. Go back and check the loop condition

Step by step example:

Let us start with the following unsorted array.

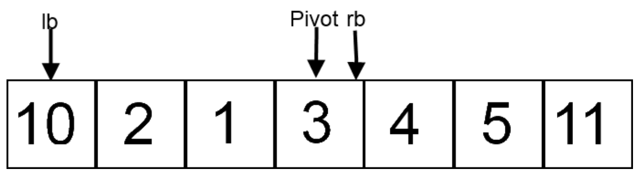


First thing, we need to pick the pivot value. For the sake of simplicity, we will pick the value at the middle of the array as the pivot point. Since the left index is originally 0 and the right index is 6 (our array has 7 elements), the pivot value will be at array index 3 which in this case is also the value 3.
Note: In all the below pictures: "Pivot" is the pivot value's index, "lb" is the left boundary index and "rb" is the right boundary index.



Going from left to right, position "lb" at the first value greater than or equal to the pivot value. Then, going from right to left, position "rb" at the first value less than or equal to the pivot value.

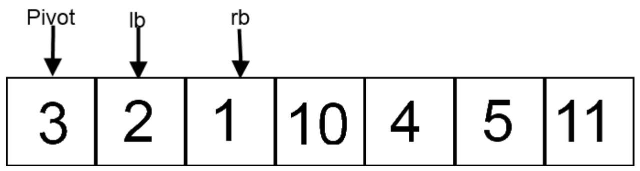
Below, is our current state after moving "lb" and "rb":



First thing to check after positioning "lb" and "rb" is if they crossed each other. If they crossed, meaning "lb" index is greater than "rb" index, no swapping happens and it's time to start a two new recursive calls with two sub arrays.

At this point, since "lb" and "rb" didn't cross yet, we have the first two values that we need to swap. Swap the values at "lb" and "rb", then move "lb" one element to the right and "rb" one element to the left.

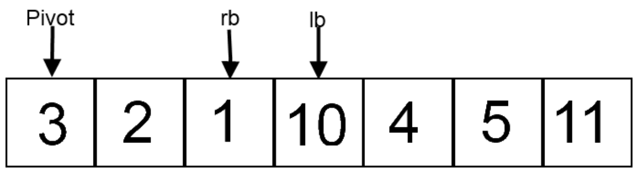
Below is our current status:



Note: **As you can see, the pivot element got moved. That is totally normal, we only care about the pivot's value and not its index in the array.**

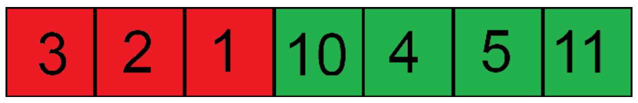
Next, we repeat the same step above; moving "lb" from left to right and stopping it at the first value greater than or equal to the pivot value, then movin "rb" from right to left and stopping it at the first value less than or equal to the pivot value.

The index "lb" will stop at the value 10 and "rb" at 1.



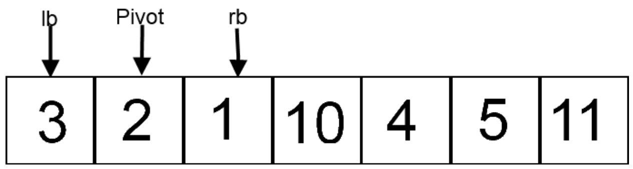
Notice that "lb" and "rb" crossed, the values they index should not be swapped. Instead, we need to recursively call the same algorithm on two sub-arrays. The sub-arrays are still part of the same parent array, just think about as we are going to work on two smaller portions of the same array.

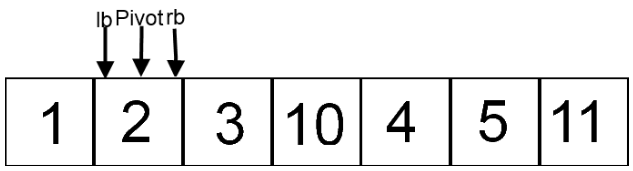
The first recursive call will deal with the elements between the index 0 (which is where "lb" was first positioned at the beginning of this algorithm call) and "rb", shown in red in the below picture. The second recursive call will deal with all elements between "lb" and the end of the array (which is where "rb" was positioned at the beginning of this algorithm call), shown in green in the below picture.



Every recursive call will of course follow the same algorithm but with different starting indices. Below, you can see, in brief, the steps done on the first sub-array.

- Pivot chosen.
- New "lb" and "rb" indices going respectively from left to right and right to left.
- Swapping values when needed.
- Stopping when "lb" and "rb" cross.
- New recursive calls if necessary.





After every recursive call ends, we will get our sorted array.

1	2	3	4	5	10	11

Pseudo-Code

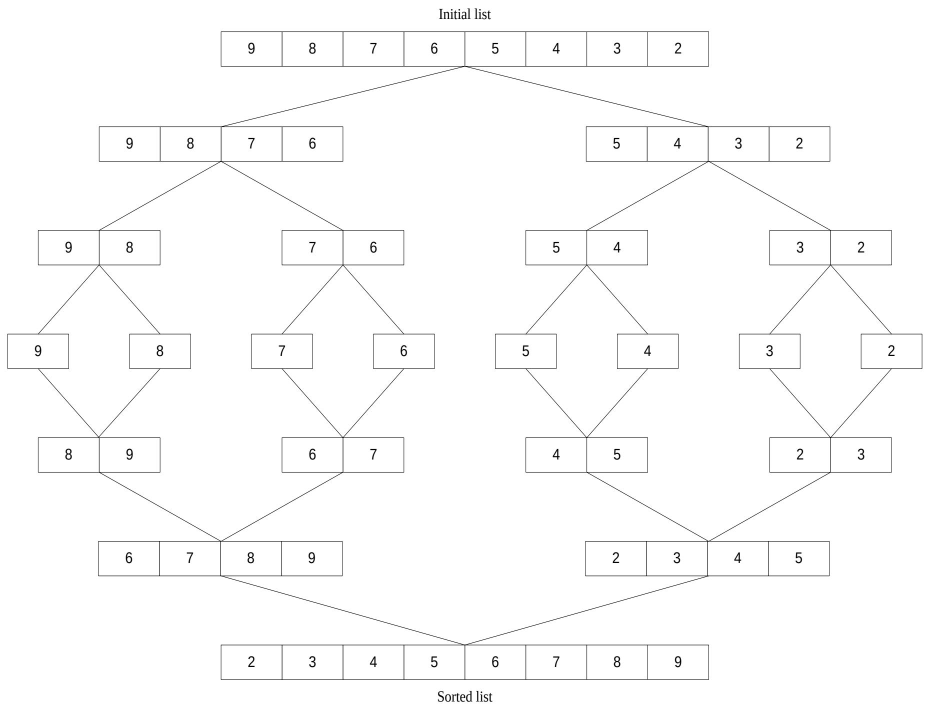
```
void quickSort(int Array[], int left, int right)
{
        o  if(left >= right)
              return;

        o  create the required indices:
           lb = left
           rb = right

        o  Choose a pivot value: array[ (left + right) / 2 ];

        o  loop until lb crosses rb
              ▪  Position lb at the first value greater than or equal to the pivot value
              ▪  Position rb at the first value less than or equal to the pivot value
              ▪  if(lb <= rb)
                    •  swap the values at lb and rb
                    •  move lb one element to the right
                    •  move rb one element to the left


        o  quickSort(Array, left, rb)
        o  quickSort(Array, lb, right)
}
```

Best Case	Average Case	Worst Case
O(n log(n))	**O(n log(n))**	**O(n^2)**

Merge Sort (Recursive)

The merge sort algorithm apply the divide and conquer approach. The problem consists of separating a list into two parts whose sizes are as nearly equal as possible. Next, we divide each part into two parts and continue this process until the part is small or the part cannot be divided any more. Once the part is very small, we merge the parts two by two so that each new part is a sorted combination of the two parts that were merged. The merging continues until we end up with one list again.

The following figure illustrate the array subdivision and merging from the initial list to the sorted list.



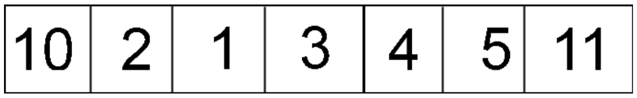
Pseudo-Code

```
void mergeSort(int Array[], int left, int right)
{
        o  The base case of the recursion is when the array to be sorted
           consists of one element. In this case nothing is done and the
           recursion break.
              ▪  if(left == right)
                     return

        o  Compute middle = (left + right)/2
        o  Split the data list into two sub-list
              ▪  Recursive call of the mergeSort algorithm on the first half
                 (from left to middle)
              ▪  Recursive call of the mergeSort algorithm on the second half
                 (from middle+1 to right)

        o  Merging the lists two by two
              ▪  When merging two sub-lists, we are sorting the data as well
}
```

Best Case	Average Case	Worst Case
O(n log(n))	**O(n log(n))**	**O(n log(n))**

<u>Step by step example:</u>

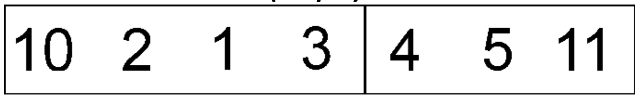
Starting with the following unsorted array:



First split: left = 0, right = 6, middle = (0+6)/2 = 3

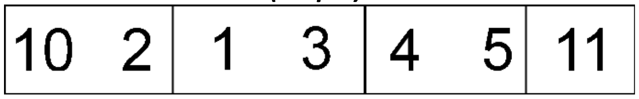
The first sub-array is from index 0 to index 3 (which is the middle). The second sub-array is from index 4 (which is middle + 1) to index 6.

(Step 1)

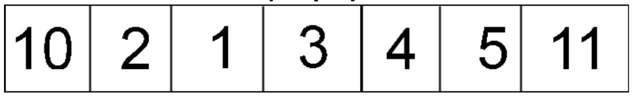


Splitting to sub-arrays will keep happening until every sub-array is made from one element:

(Step 2)

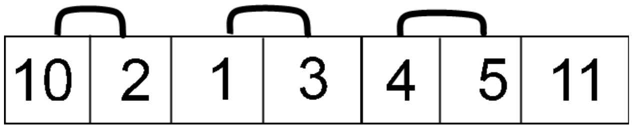


(Step 3)

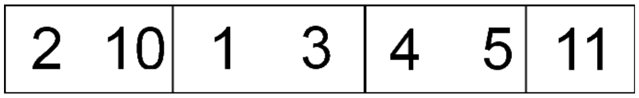


At this point, the array is split into individual elements. We can now start merging these sub-arrays back together.

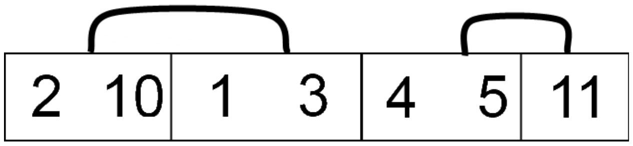
Going up in our recursive calls, we sort the sub-arrays (that were created from "Step 2" to "Step 3") among each other.



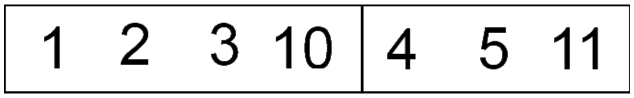
The above 3 sub-arrays sorting calls lead to the following:



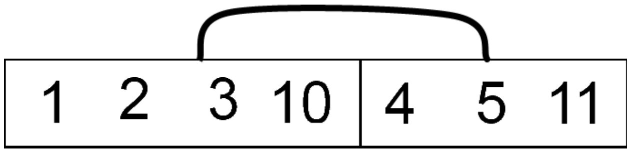
Next we merge the sub-arrays from step 2 together:



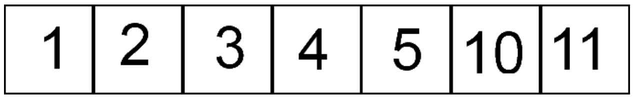
Which leads to:



And finally, we merge the first sub-arrays created **(Step 1)**

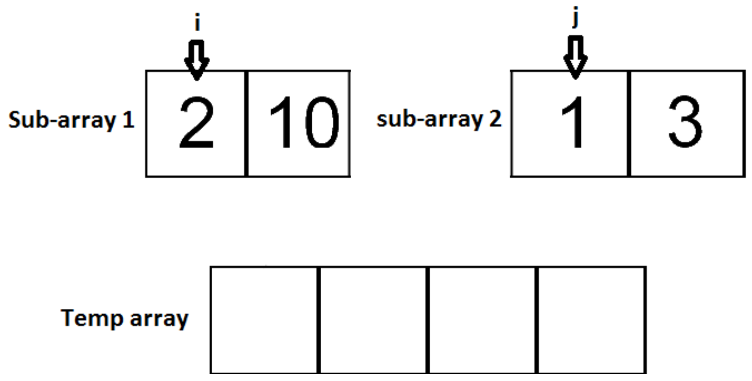


Final sorted array:



We can now see that the array is sorted. However how do we merge and sort two sub-arrays?

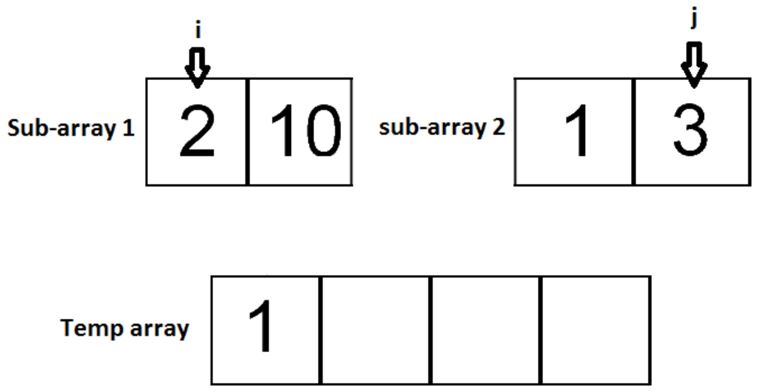
Assuming we have the following 2 sub-arrays:



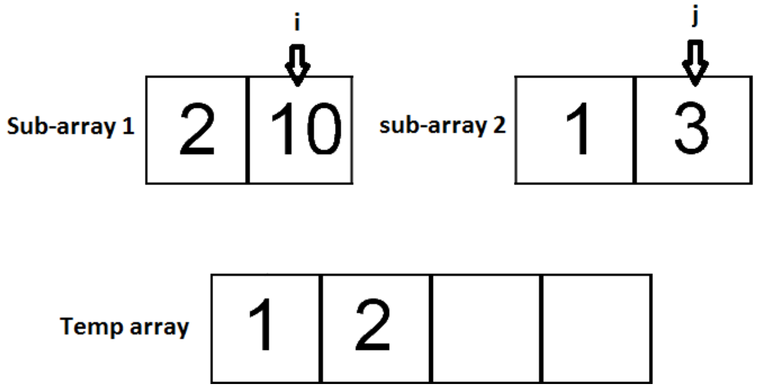
Note: The two sub-arrays look as if they are not connected, but in fact they are part of the original array.

First create a new array equal to the size of the two sub-arrays combined **(Temp array).** Start two indices **"i"** starting in the beginning of **"sub-array 1"** and **"j"** starting at **"sub-array 2".**

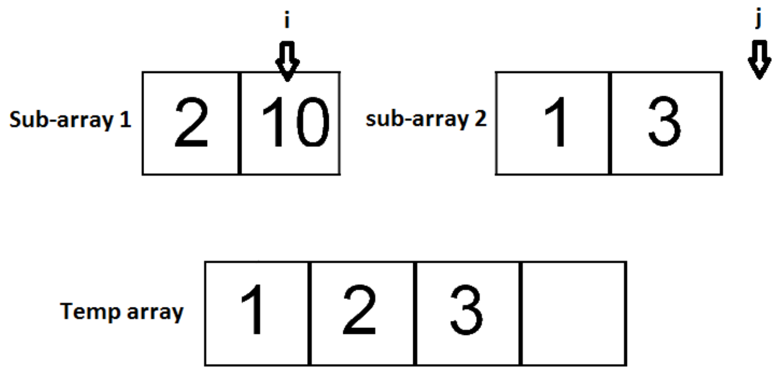
Now, compare the value at index **"i"** with the value at index **"j"**. Copy the lowest to the **"Temp array".** Since the lowest is **"1"**, move the index **"j"** one element to the right.



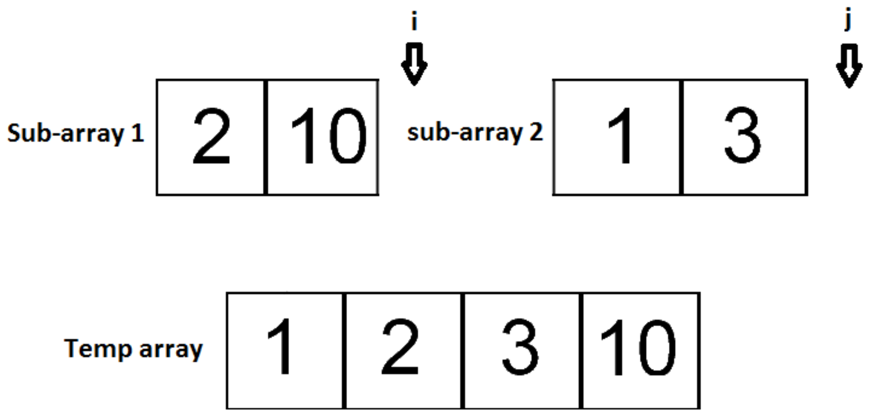
Again, compare the value at index **"i"** with the value at index **"j"** and copy the lowest to the **"Temp array".** Since the lowest is **"2"**, move the index **"i"** one element to the right.



Again, compare the value at index **"i"** with the value at index **"j"** and copy the lowest to the **"Temp array".** Since the lowest is **"3"**, move the index **"j"** one element to the right.



Since **"j"** reached an index outside of **"sub-array 2"** we should stop comparing values among the two sub-arrays. At this point, we should just copy the remaining values in **"sub-array 1"** to the temp array.



Note: It is possible that "i" reaches outside of "sub-array 1" first then you would loop through the remaining values in "sub-array 2" and copy them to the "Temp array".

Now that we have a sorted **"Temp array"**, we should just copy the values to our original array.

Searching

Sequential Search

Sequential search is the simplest form of searching algorithms. It is called sequential search because the algorithm is actually going through the list of data in sequence trying to find the required data.

Example: Assume we have an array if integers and we would like to find the index of the first occurrence of a certain value inside the array. The code will look something like this:

Algorithm

```
public static int findFirstOccurenceOf(int[] array, int value)
{
	for(int i = 0; i < array.length; ++i)
	{
		if(array[i] == value)
		{
			/* When value is found, return its index */
			return i;
		}
	}

	/* Value not found */
	return -1;
}
```

Explanation

As you can see, the loop is going through the array one element at a time and checking if the element's value is equal to the user specified value. If the value is found, the index is returned else -1 is returned.

The algorithm's best case is met if the first element inside the array is the value we are looking for. The worst case is when the value is found at the end or not found at all since the loop has to go through the whole array.

Best Case	Average	Worst Case
O(1)	**O(n)**	**O(n)**

Binary Search

Binary search is a faster searching algorithm than sequential search but has a limitation; The list of data has to be sorted.

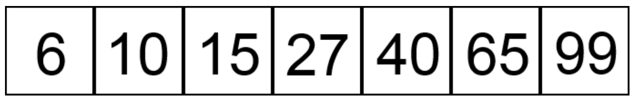
Assume we want to find the index of a value inside a sorted array of integers. The algorithm will start by comparing the user specified value with the value in the middle of the sorted array:

- If they are equal, we found our index.
- If the value in the middle of the array is greater than the user specified value then we continue our search in the lower part of the array (indices smaller than the middle index).
- If the value in the middle of the array is smaller than the user specified value then we continue our search in the upper part of the array (indices greater than the middle index).

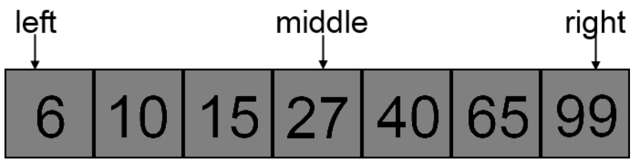
Note: The above description assumes that the array is sorted in ascending order.

Step by step example:

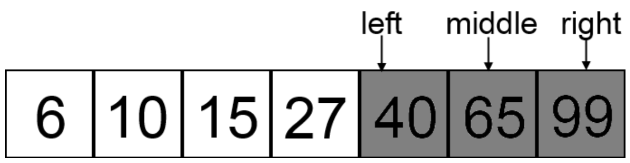
Below is our sorted array and we are searching for the value 40.



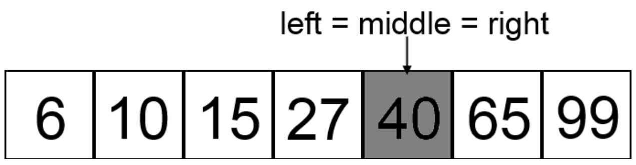
Initially, the range of indices that we are looking in is **left = 0** to **right = array length - 1**. Next we need to determine the middle point **(left + right) / 2**.



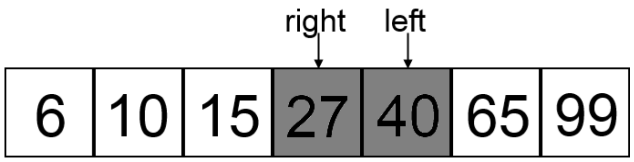
The first step is to compare the value at the middle index to the value we are searching for. Since 27 is smaller than 40, we determine the new range for our search (**middle + 1** to **right**).



Going through the same steps with our new range, we determine that the middle value 65 is greater than 40 and by that, our new range becomes the values between the indices **left** and **middle - 1.**



Here we can see that **left = middle = right**, which usually means that the value is found. And in fact, the picture shows that we found our value, so we can return the index 4.
Now the question is, what if the value we need is not found in the array, say we are looking for 41. We would still end up at the above diagram however the next pass will make the **left** index cross the **right** index, which is an indication that the value doesn't exist in the array.



Best Case	Average	Worst Case
O(1)	**O(log n)**	**O(log n)**