# Lab: Sorting and Searching Algorithms

This document defines the **in-class exercises** assignments for the ["Algorithms" course @ Software University](https://softuni.bg/trainings/1688/algorithms-july-2017).

# Part I – Sorting

1. **Merge Sort**

Sort an array of elements using the famous merge sort

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 1 2 3 4 5 |

### Hints

Create your Mergesort generic class with a single Sort method



Create an **auxiliary array** that will help with merging subarrays



Now to implement the Merge() method



As the two subarrays are sorted, if the **largest element in the left** is smaller than the **smallest in the right**, the two subarrays are **already merged**



If they are not, however, **transfer all elements to the auxiliary array**



Then **merge them back** in the main array



Now to create the recursive Sort() method



If there is **only one element** in the subarray, it is **already sorted**



If not, however, you need to **split it into two subarrays**, **sort them recursively** and then **merge them on the way up** of the recursion (as a post-action)



You can now call the Sort() method



## Quicksort

Sort an array of elements using the famous quicksort

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5 4 3 2 1 | 1 2 3 4 5 |

### Hints

You can learn about the Quicksort algorithm from [Wikipedia](https://en.wikipedia.org/wiki/Quicksort).

A great tool for visualizing the algorithm (along with many others) is available at [Visualgo.net](http://visualgo.net/sorting.html).

The algorithm in short:

* Quicksort takes unsorted partitions of an array and sorts them
* We choose the **pivot**
  + We pick the first element from the unsorted partition and move it in such a way that all smaller elements are on its left and all greater, to its right
* With pivot moved to its correct place, we now have two unsorted partitions – one to the left of it and one to the right
* **Call the procedure recursively** for each partition
* The bottom of the recursion is when a partition has a size of 1, which is by definition sorted

First, define the **class** and its **sorting method**:



Now to implement the private Sort() method. Don't forget to handle the **bottom of the recursion**



First, find the pivot index and rearange the elements, then sort the left and right partitions recursively:



Now to choose the pivot point.. we need to create a method called Partition()



If there is **only one element**, it is already partitioned and the index of the pivot is the index of its only element



Finding the pivot point involves **rearanging all elements** in the partition so it satisfies the condition **all elements to the reft of the pivot to be smaller** from it, and **all elements to its right to be greater** than it



# Part II – Implement Binary Search

## Implement Binary Search

Implement an algorithm that finds the index of an element in a sorted array of integers in logarithmic time

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 2 3 4 5  1 | 0 | Index of 1 is 0 |
| -1 0 1 2 4  1 | 2 | Index of 1 is 2 |

### Hints

First, if you’re not familiar with the concept, read about binary search in [Wikipedia](https://en.wikipedia.org/wiki/Binary_search_algorithm).

[Here](http://www.dave-reed.com/book/Chapter8/search.html) you can find a tool which shows visually how the search is performed.

In short, if we have a **sorted collection** of comparable elements, instead of doing linear search (which takes linear time), we can eliminate half the elements at each step and finish in logarithmic time.

Binary search is a **divide-and-conquer** algorithm; we start at the middle of the collection, if we haven’t found the element there, there are three possibilities:

* The element we’re looking for is smaller – then look to the left of the current element, we know all elements to the right are larger;
* The element we’re looking for is larger – look to the right of the current element;
* The element is not present, traditionally, return -1 in that case.

Start by defining a class with a method



Inside the method, define two variables defining the bounds to be searched and a while loop



Inside the while loop, we need to find the midpoint



If the key is to the left of the midpoint, move the right bound. If the key is to the right of the midpoint, move the left bound



That’s it! Good job!