Laboratory 5: “Sorting in Arrays”

Arkadiusz Chrobot, PhD

September 4, 2026

1 Introduction

According to Prof. Donald Knuth, sorting and searching are the most frequently performed operations by computers. Many algorithms have been discovered for these operations. In the first section of this laboratory manual we describe the *selection* sort algorithm and show its example implementation. In the second section the *binary search* algorithm is discussed. The manual ends with a set of programming exercises for solving during the laboratory classes.

2 Sorting In Linear Arrays

A Linear Array is one of the data structures for which the sorting operation can be performed. Sorting can be internal, i.e. performed entirely in the RAM, or external, i.e. performed mostly in a mass memory device. If the sorting algorithm preserves the initial order of the same values in the array, then it is called a stable sort algorithm. An algorithm that lacks this property is called an unstable sort algorithm. To show how these algorithms work let's assume that in an array are two 5. Let the first one be marked 5′ and the second one 5". If after sorting the 5′ is still before the 5"[1], then the sorting is stable, but if the 5" is before the 5′ then the sorting is unstable. Some of the sorting algorithms don't demand additional memory as they work, i.e. they use the memory they had been initially given throughout the whole operation. These algorithms perform in-place sorting (lat. *in situ*).

2.1 Selection Sort

The Selection Sort algorithm is one of the simplest to implement, although it is not the most efficient algorithm of this kind. It is in-place, internal and unstable sorting algorithm. The idea behind this algorithm is quite simple. It gets an unsorted array as input[2]. Since the output of that algorithm should be sorted array, then its first element should have the minimal value. Thus, the selection sort algorithm searches for this value in the entire array and if it finds out that it is in a different element than the first one, it swaps the values of these elements. Next, it repeats these actions for the second element, but this time it excludes the first one for searching for the minimal value. The algorithm ends when it sort the values of the last to elements. A function in the listing 1 applies the selection sort algorithm for sorting an array of integers (numbers of the `int` type). The implementation of the `swap()` function, which called by `selection_sort()` was given in the fifth lecture.

```
void selection_sort(int array[])
{
    int i,j;

    for(i=0; i<NUMBER_OF_ELEMENTS-1; i++) {
        int min = i;
        for(j=i+1; j<NUMBER_OF_ELEMENTS; j++)
            if(array[min]>array[j])
                min = j;
        if(min!=i)
            swap(&array[min],&array[i]);
    }
}
```

Listing 1: Function that implements the selection sort algorithm

[1] They don't have to occupy the same element of the array as before, though.

[2] It may happen that the array is already sorted. In that case sorting is not necessary, but the algorithm performs it anyway. It is also possible for the array, to be sorted in reversed order.

3 Binary Search Algorithm

Searching for a given value in a sorted array can be performed efficiently with the use of a binary search algorithm. To find a specific value the algorithm calculates the position of the array's middle element[3] and compares the value of the element with the value it searches for. If they are the same, then the algorithm returns the index of the element — the value has been found. If the value is smaller than the one in the middle of the array, then the algorithm takes all the elements, that are on the left side, and finds the middle of that part and then repeats the steps described earlier. However, if the target value is bigger than the one in the center of the array, then the algorithm takes all the elements on the right side, calculates its middle and also repeats the steps described earlier. The algorithm terminates if it finds the value or it runs out of elements to verify.

If we compare the binary search array and the linear search array discussed in the lecture then we will find the following differences:

1. the binary search algorithm can be applied only to a sorted array, while the linear search array can be used both for sorted and unsorted array,
2. if the target value is repeated several times in the array, then the linear algorithm finds it's first occurrence and the binary sort algorithm just finds it's occurrence — it is not guaranteed to be the first one,
3. in each iteration the linear search algorithm reduces the number of the elements that it still needs to scan by one, and the binary search algorithm by half, hence the latter is more efficient.

A function in the listing 2 is an implementation of the binary search algorithm that locates a given number in an array of integers. In case the target number is not present in the array, the algorithm returns -1.

```
int binary_search(int array[], int value)
{
    int down=0, up=NUMBER_OF_ELEMENTS-1;

    while(down<=up) {
        int middle = down+((up-down)/2);
        if(array[middle]==value)
            return middle;
        if(array[middle]<value) {
            down = middle + 1;
            continue;
        }
        if(array[middle]>value)
            up = middle - 1;
    }

    return -1;
}
```

Listing 2: A function implementing the binary search algorithm

4 Exercises

All Your programs have to have functions with parameters!

[3] If the array has an even number of elements, then the middle element is the one on the left side, closest to the center of the array.

1. Write a program that fills with randomly chosen numbers ranging from 0 to 10, an array of 30 elements, displays the content of the array and how many times each of these numbers occurs in the array. *Hint: create a histogram.*

2. Write a program that sorts an array of 30 elements of the `short int` type, filled with randomly chosen natural numbers ranging from 0 to 8. The program should display the content of the array before and after sorting.

3. There are specialized algorithms that sort an array of integers more efficiently than the general–purpose sorting algorithms. Using the solution of the first problem (1) try to discover such a specialized algorithm and modify the program from the second problem (2) to implement it.

4. How would You solve the third problem (3), if the array contained only unique numbers? You can create such an array using algorithms presented in the slides for the fifth lecture. Write a program to demonstrate Your solution.

5. Modify the program from the fist problem (1) so it counts the occurrences of values in an array of randomly chosen integers ranging from -5 to 5.

6. Write a program that fills with randomly chosen integers, ranging from -4 to 4 an array of 20 elements, sorts it and finds the median[4], and the first and third quartile[5].

[4] If the array has an even number of elements then the median is an arithmetic mean of values stored in two elements that are located in the middle of the array.

[5] The first quartile (Q_1) is the value of the element located in $\frac{1}{4}$ of the distance from the beginning of the array. Likewise, the third quartile (Q_3) is the value of the element located in the $\frac{3}{4}$ distance from the beginning of the array.