Fundamentals of Programming 1

Sorting on Linear Arrays

Arkadiusz Chrobot

Department of Computer Science

November 30, 2020

Outline

Sorting

Data sorting (ordering) and data searching are among the most frequently performed operations by computers. They are also the main topic of the third volume of "The Art of Computer Programming" by prof. Donald E. Knuth, which first edition was published in 70ties. In the book the Author has written that if all computers were halted in the same moment most of them would be stopped while performing sorting and searching. This is also true today. The data that undergo sorting may be stored in any data structure, but in this lecture only sorting on a linear array will be discussed. As a result of sorting the data are in an ascending (non-descending) or descending (non-ascending) order.

Sorting

Properties of Sorting

There are many sorting algorithms, that have different properties. In internal sorting all sorted data are stored in the main memory[1], while in external sorting they are in the secondary storage. In *stable sorting* the relative order of the same values is maintained. For example if there are two 10s in the unsorted array denoted by $10'$ and $10''$ then after sorting the $10'$ still will be before $10''$ in the array. For numbers this property is of less meaning, but in case of some more advanced data types it can be useful. When an algorithm sorts a data structure using only a constant amount of memory it is called an in-place (lat. *in situ*) algorithm and the operation is called an in-place sorting. In the lecture only sorting of numbers will be discussed, but all other data, like characters, also can be sorted.

[1] The virtual memory is not taken into consideration here.

Sorting

Example Algorithms

Three sorting algorithms that sort the linear array are introduced in the lecture. All of them are in-place, internal sorting algorithms. Two of them are stable sorting algorithms and the third one (selection sorting) can be easily modified to be so. For arrays with a relatively small number of elements their effectiveness is almost the same. In the lecture are presented implementations of the algorithms that sort arrays in an ascending (non-descending) order, but also information of how to modify the algorithms to sort in a reverse order is provided.

Selection Sort

Algorithm Description

The Selection Sort algorithm is quite simple. It is related to the algorithm for finding a minimum value in an unsorted array. Selection sort is based on the observation, that the first element in a sorted array should store the minimum value. Therefore, the element holding such a value has to be located and if it is not the first element, its value should be swapped with the value of the first one. Then the operation is repeated, but this time with the exception of the first element of the array (starting with the second), and so on. After sorting on the last two elements the whole array is sorted.

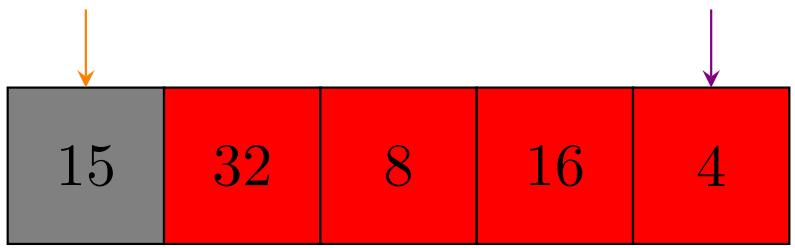
Selection Sort

Simulation

The next slide contains a visualisation of sorting on a linear array of five elements, containing some natural numbers with the use of the selection sort algorithm. The orange arrow indicates the currently sorted element. The violet arrow indicates the element that stores the smallest value and belongs to the still unsorted part of the array. This part also includes the currently sorted element. The latter is also indicated by the dark gray background. Those elements that are already sorted have a green background. The still unsorted elements have a red background. The steps required to find the minimal value are not shown in the visualisation. Those are the same as in the algorithm for finding the minimal value in an unsorted array.

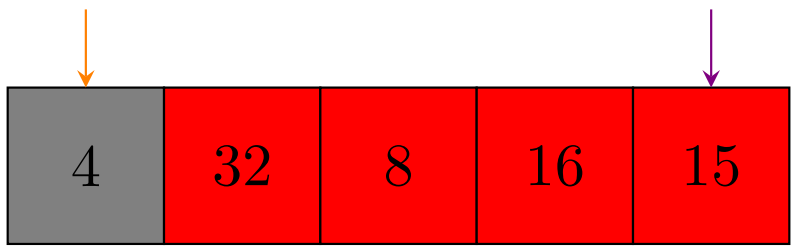
Selection Sort

Simulation



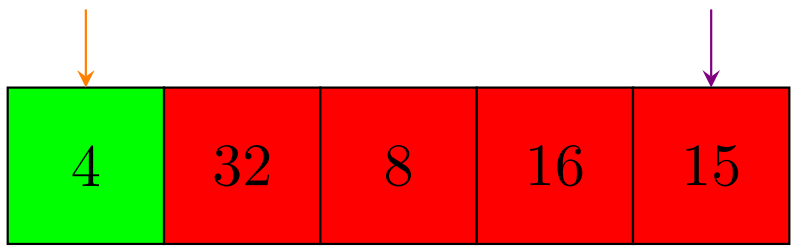
Selection Sort

Simulation



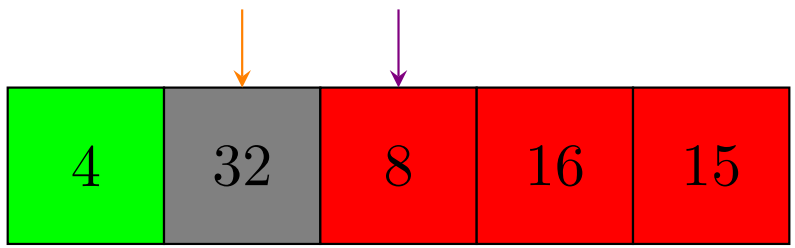
Selection Sort

Simulation



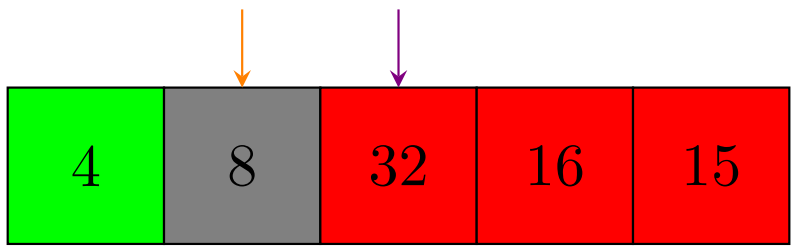
Selection Sort

Simulation



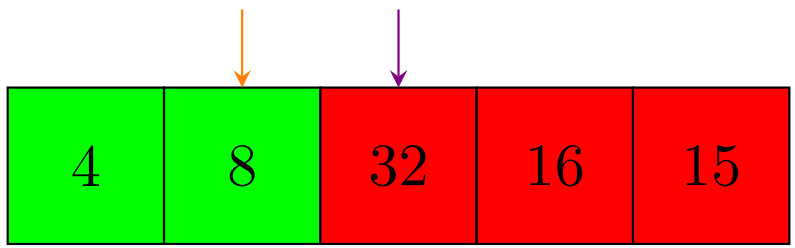
Selection Sort

Simulation



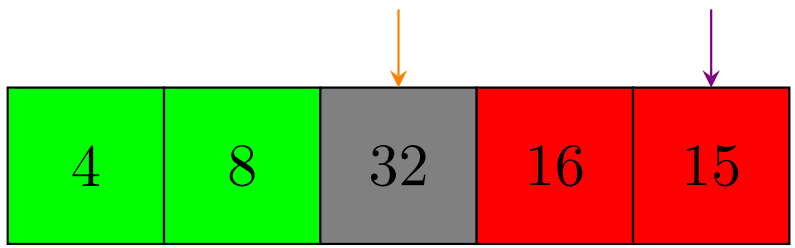
Selection Sort

Simulation



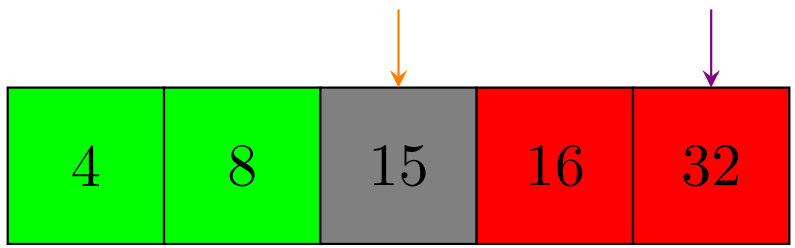
Selection Sort

Simulation



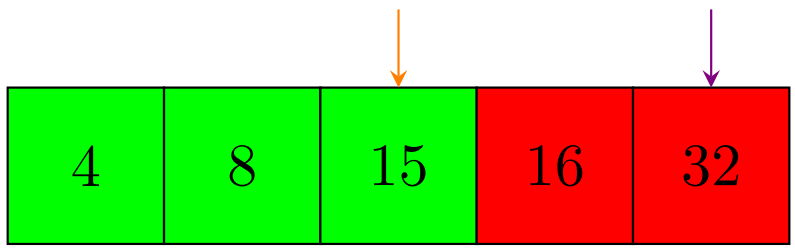
Selection Sort

Simulation



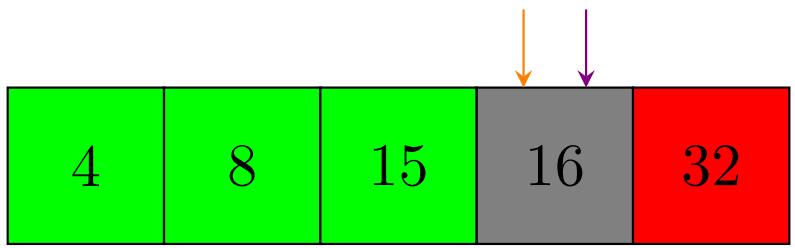
Selection Sort

Simulation



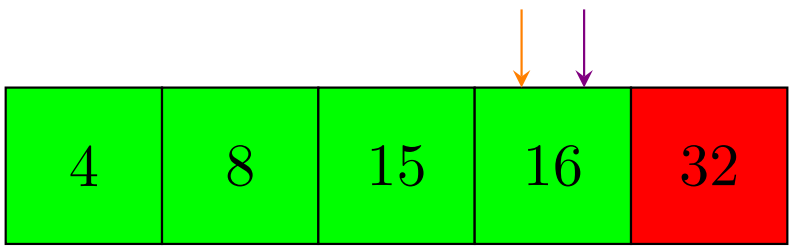
Selection Sort

Simulation



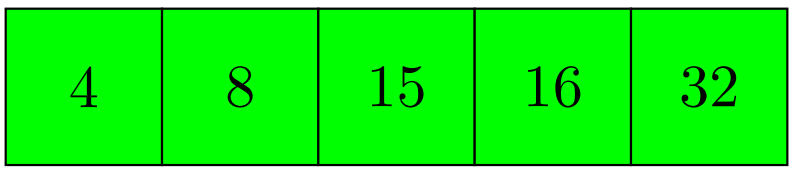
Selection Sort

Simulation



Selection Sort

Simulation



Selection Sort

Implementation

```
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]);
    }
}
```

Selection Sort

A Comment to the Implementation

The counter of the outer `for` loop indicates the element of the array that should be sorted (just like the orange arrow in the visualisation). The inner `for` loop searches for the smallest value in the part of the array to the right of the currently sorted element. The algorithm is interested in the location of the minimum value not in the value itself. In other words it is interested in the index of the element that stores the value (the violet arrow in the visualisation). If the `min` variable indicates different element than the `i` variable, when the inner loop finishes, then the values of the elements indicated by both variables should be exchanged. It is done by the `swap()` function, which was introduced in the previous lecture. The algorithm can be modified to perform a stable sorting by changing the greater than operator in the conditional statement to the greater or equal. If the operator is reversed then the array will be sorted in a descending (non-ascending) order. The efficiency of the algorithm can be improved by modifying it to sort the array "at both ends" and search for the smallest and biggest value simultaneously.

The swap() Function

A Different Implementation (Digression)

```
void swap(int *first, int *second)
{
    if(first!=second) {
        *first ^= *second;
        *second ^= *first;
        *first ^= *second;
    }
}
```

The function presented above swaps values of two variables passed to it by pointers, but without using an additional variable. It is enabled by applying the ^ (xor) operator, which effect can be reversed. However, if the address of the same variable is passed by both its parameters then the variable will be zeroed out. Hence, if the conditional statement detects such a case, no action will be taken by the function.

The `swap()` Function

A Different Implementation (Digression) — Continuation

The advantage of such an implementation of the `swap()` function is that it uses a little less memory than the more common implementation. However, it is less efficient and it cannot be applied to variables of the `float`, `double` and more advanced data types. The function can also be implemented with the use of some arithmetic operators, but it will still undergo similar limitations.

Insertion Sort

Algorithm Description

The insertion sort algorithm takes a different approach to the problem of sorting on a linear array. It is based on the behaviour of same players playing a card game. If they take a new card they try to find a place in a deck for it by moving those cards that they already keep in a hand. The algorithm begins sorting on the array starting from the second element, by taking its value and comparing it with the value of the first element. If the latter is greater, then the algorithm copies it to the element on the right and inserts the value from the second element to the first one. Then it proceeds similarly with the rest of the elements, i.e. it takes the value of the nth element and compares it with the values of elements to the left of it. If the value of some element is greater than the value of the nth element then it is copied to the element to the right. Otherwise the value of the nth element is inserted into the element located to the right of the element which value was compared the last. The algorithm finishes when it finds the right place for the value of the last element of the array.

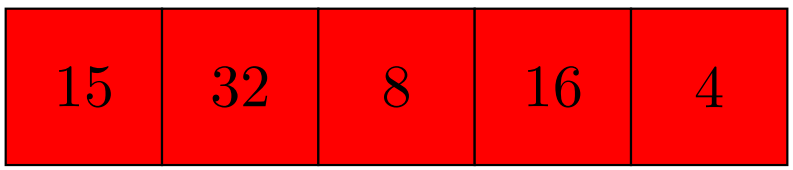
Insertion Sort

Simulation

The next slide contains a visualization of sorting on a linear array with the use of the insertion sort algorithm. The element which value is currently sorted has a dark gray background. The value itself is shown and moved above the elements with which it is compared. Please observe, that instead of exchanging values between elements of array, the algorithm just copies the value of an element to the element to its right. Such an operation is called a “half exchange”. The elements with green background are already sorted, the ones with red background are yet to be sorted.

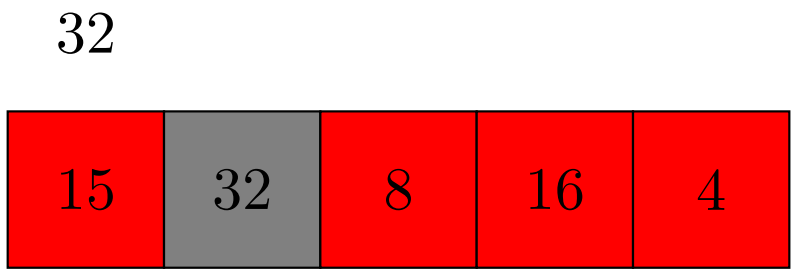
Insertion Sort

Simulation



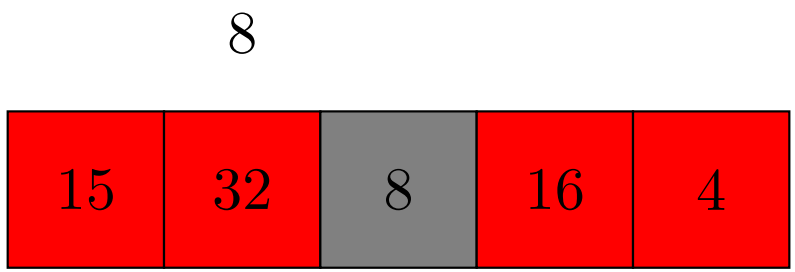
Insertion Sort

Simulation



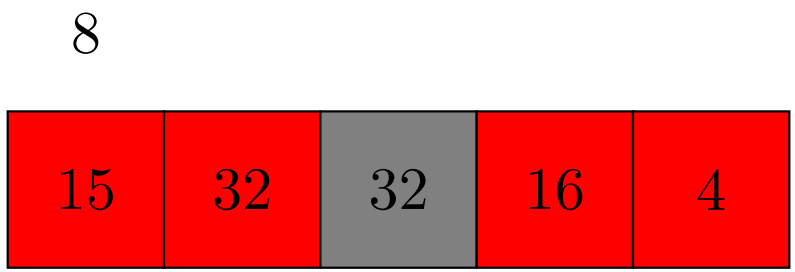
Insertion Sort

Simulation



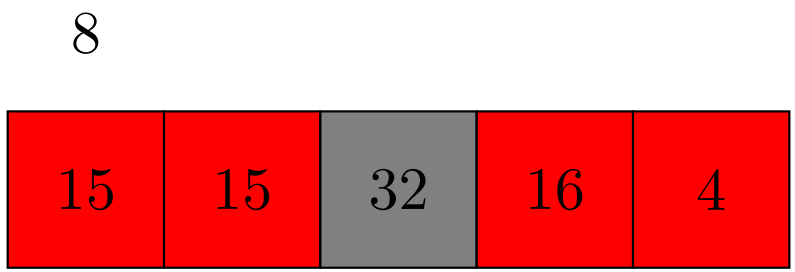
Insertion Sort

Simulation



Insertion Sort

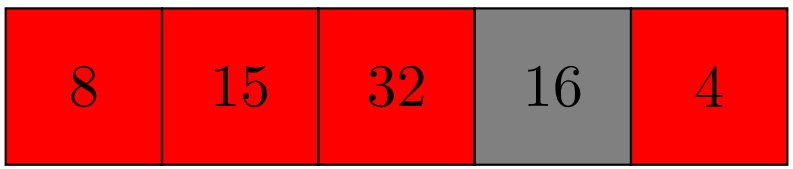
Simulation



Insertion Sort

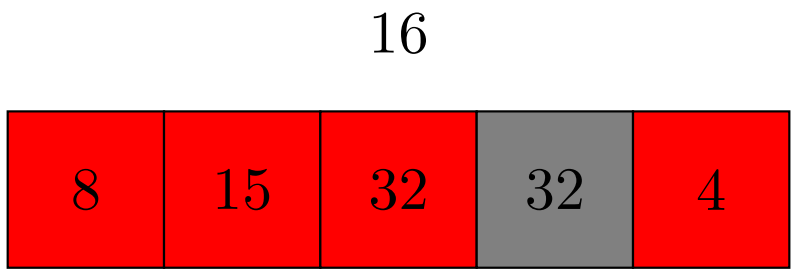
Simulation

16



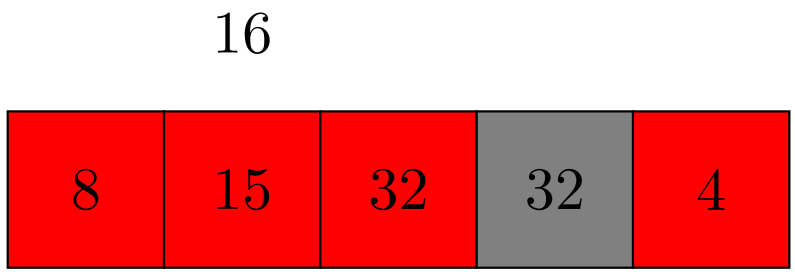
Insertion Sort

Simulation



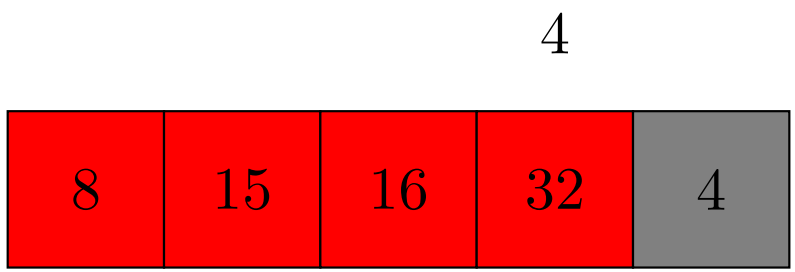
Insertion Sort

Simulation



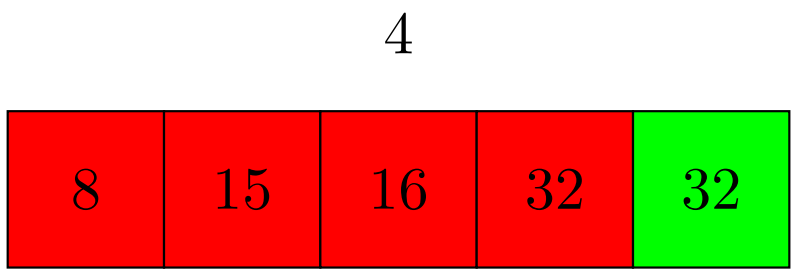
Insertion Sort

Simulation



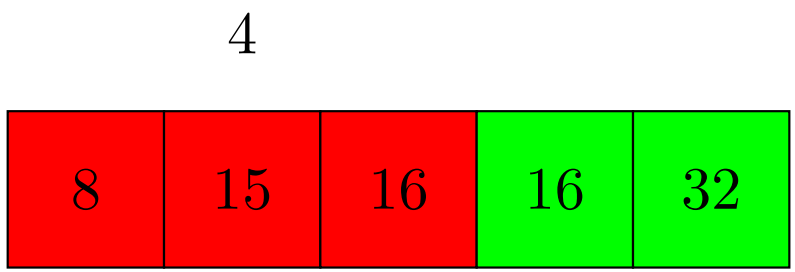
Insertion Sort

Simulation



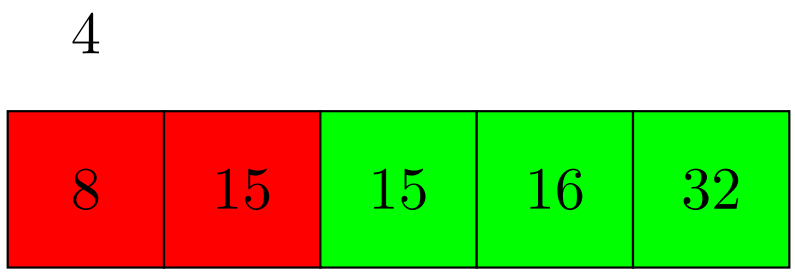
Insertion Sort

Simulation



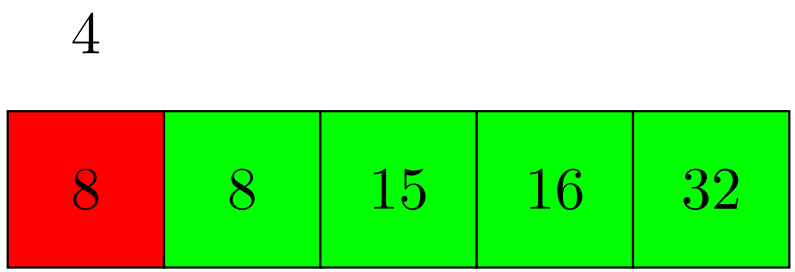
Insertion Sort

Simulation



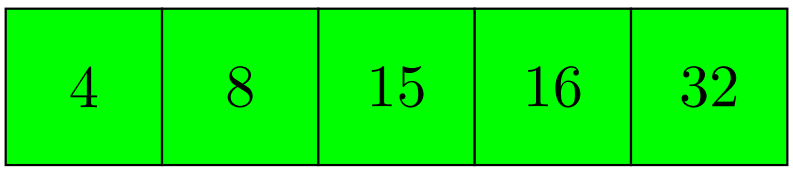
Insertion Sort

Simulation



Insertion Sort

Simulation



Insertion Sort

Implementation

```
void insertion_sort(int array[])
{
    int i;
    for(i=1;i<NUMBER_OF_ELEMENTS;i++) {
        int key = array[i];
        int j = i-1;
        while(j>=0&&array[j]>key) {
            array[j+1]=array[j];
            j--;
        }
        array[j+1]=key;
    }
}
```

Insertion Sort

Comment to the Implementation

The implementation of the insertion sort is more complicated than the implementation of selection sort. The counter of the `for` loop indicates an element which value is currently sorted. In the `while` loop the value is compared with the values of the elements located to the left of it. If those values are greater then they are moved right. Otherwise the currently sorted value is inserted into an element located to the right of an element that stores a value less or equal to it. If the internal loop finishes "outside" the array, the value stored in the `key` variable is inserted to the first element of the array. To change the direction of sorting the operator in the `array[j]>key` expression should be reversed.

Insertion Sort

Insertion Sort vs. Selection Sort

To some extent the insertion sort algorithm is the opposite of the selection sort:

- selection sort matches the right element to the right value, the insertion sort matches the right value to the right element,
- the selection sort scans the array form the left to the right and the insertion sort from the right to the left.

Bubble Sort

Algorithm Description

There are several versions of bubble sort algorithm. In one of them the algorithm scans the array from the right to the left, an element after an element, and it compares the values of neighbouring elements. If they are not sorted then it exchanges the values of those elements. After each cycle of such scanning at least one of the elements at the left side of the array is sorted. This element is omitted in the next cycle. The scanning is repeated until the last two elements of the array are sorted.

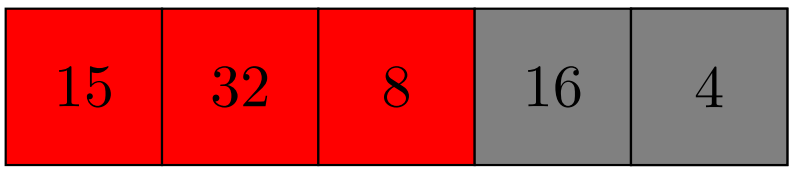
Bubble Sort

Simulation

The next slide shows a visualization of sorting on a linear array with the use of the bubble sort algorithm. The pair of elements that is currently sorted has a gray background. The elements with the green background are already sorted, the ones with the red background need to be sorted.

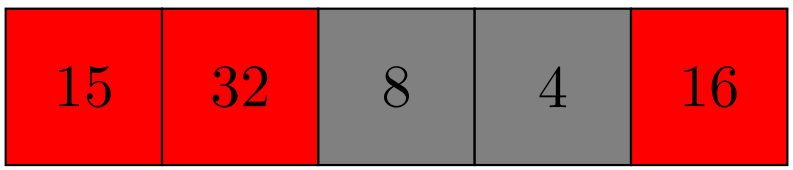
Bubble Sort

Simulation



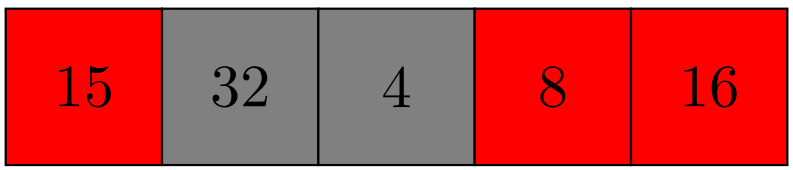
Bubble Sort

Simulation



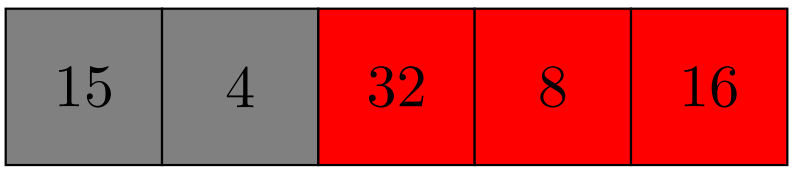
Bubble Sort

Simulation



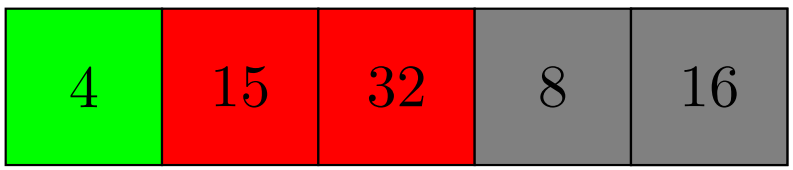
Bubble Sort

Simulation



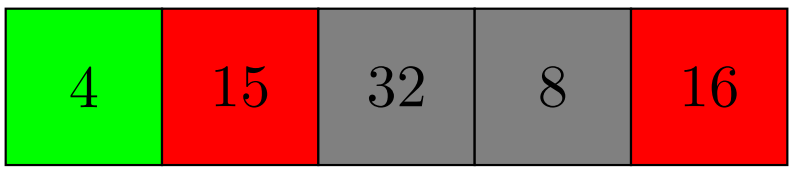
Bubble Sort

Simulation



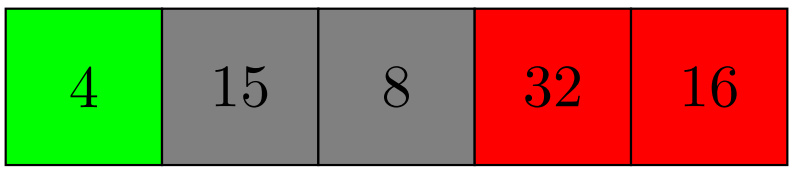
Bubble Sort

Simulation



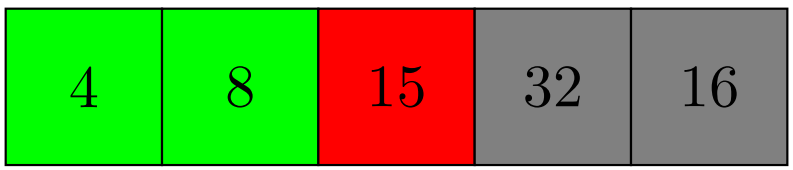
Bubble Sort

Simulation



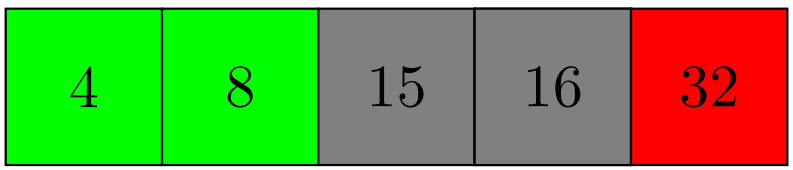
Bubble Sort

Simulation



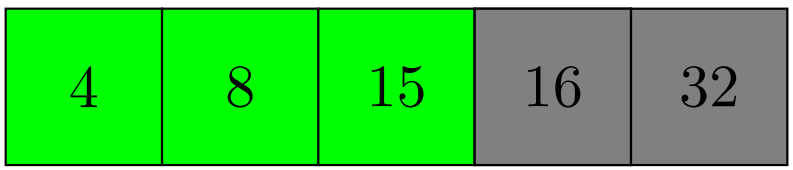
Bubble Sort

Simulation



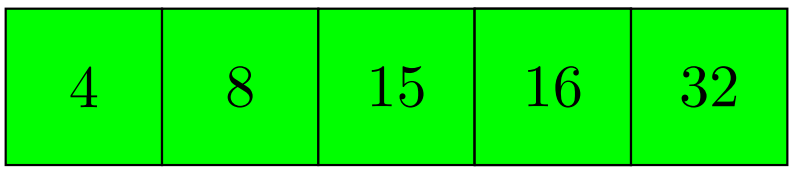
Bubble Sort

Simulation



Bubble Sort

Simulation



Bubble Sort

Implementation

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

Bubble Sort

Comment to the Implementation

The internal `for` loop scans the array from the right to the left and compares pairs of neighbouring elements. The range of the scan is limited by the loop counter of the external `for` loop. To reverse the order of sorting the operator in the conditional statement should be changed to "less than".

Sorting — Comparison of the Algorithms

All of the introduced sorting algorithms have the same properties except for stability. However, they are some differences in their efficiency. From the complexity theory point of view, it is the same for all of them. It means, that the time needed to sort an array is proportional to the square of the number of elements in the array[2]. However in practise, for reasonably large arrays, the bubble sort algorithm is the worst choice. It is not the worst possible sorting algorithm, because this title holds the *BogoSort*, which generates a permutation of the values in the array and then checks if they are sorted. Nevertheless, the bubble sort is considered to be an anti example of efficient sorting, because it makes a lot of time-consuming swap operations.

[2] More accurate analysis of the time complexity of those algorithms, which also depends on the initial configuration of the values in the array, is one of the topics of the Algorithms and Data Structures course.

Sorting — Comparison of the Algorithms

Continuation

The bubble sort algorithm should not be used in practise. Some computer scientists believe that it should not be also taught. The difference in the efficiency of the two other algorithms is small, but generally the performance of insertion sort is better than selection sort. However, it depends on the initial configuration of the values in the array. If they are "almost" sorted then the selection sort will be more efficient, otherwise the insertion sort will be a better choice.

Binary Search

Algorithm Description

It shows up, that searching for a given value in a sorted array can be more efficient than performing the same operation on an unsorted array. This, however requires a special algorithm that is called a binary search. It consists of several steps. In the first one the middle element of the array is located. If the array has an even number of elements then the left to the middle element is assumed to be the one. Its value is compared with the one that is desired. If they are the same the algorithm returns the index of the element and finishes. However, if the value of the middle element is greater than the desired value then it means that the latter can only be in the part of the array that is located to the left of the middle element. If the result of comparing those two values is the opposite, then the desired value can only be in the part of the array located to the right of the middle element. The algorithm repeats the steps for the chosen part of the array.

Binary Search

Algorithm Description — Continuation

The algorithm repeats the steps until it locates the desired value or the part of the array that needs to be checked becomes so small that it contains no elements. In the latter case the desired valued does not occur in the array. The algorithm always stops, because after every iteration it halves the number of the elements of the array that need to be checked in the next iteration. By contrast, the linear search algorithm reduces the number of elements to search by one element in each iteration. Therefore the binary search algorithm is much more efficient. The description of the algorithm is simple, but implementing it can be challenging. According to Jon Bentley the algorithm was already known in 1946, but it had not been correctly implemented until 1962.

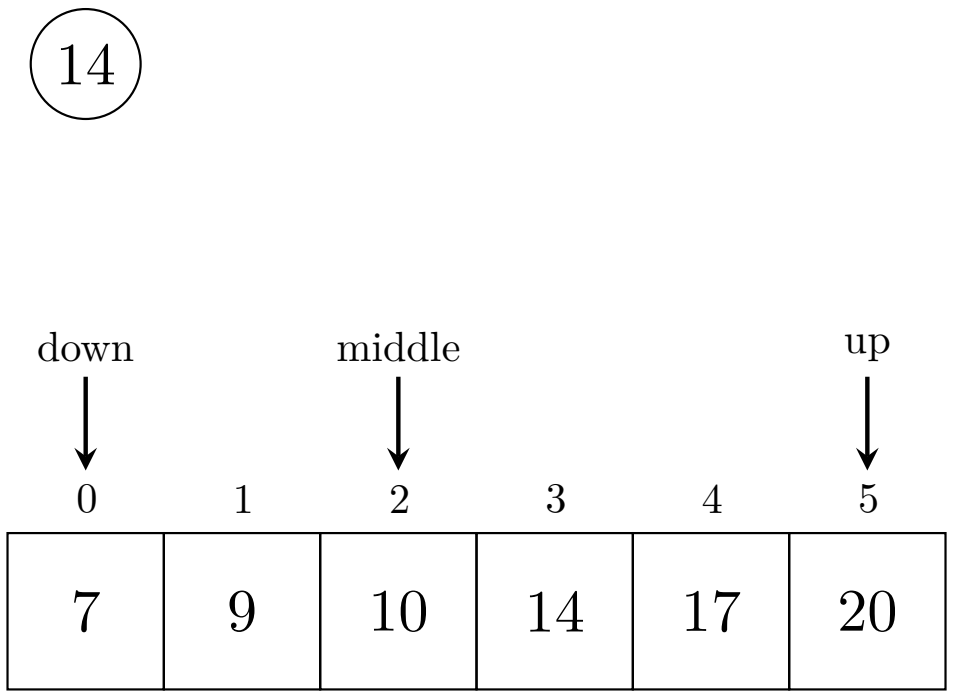
Binary Search

Simulation

The next slide shows a visualization of the binary search algorithm for a sorted array of six elements that contain natural numbers. The `down` and `up` arrows indicate the first and the last element of the part of the array that needs to be checked in the next iteration. Initially, this area contains the whole array, but in the subsequent iterations it becomes smaller. The `middle` arrow indicates the middle element in the checked part of the array. The desired value is in the circle on the left.

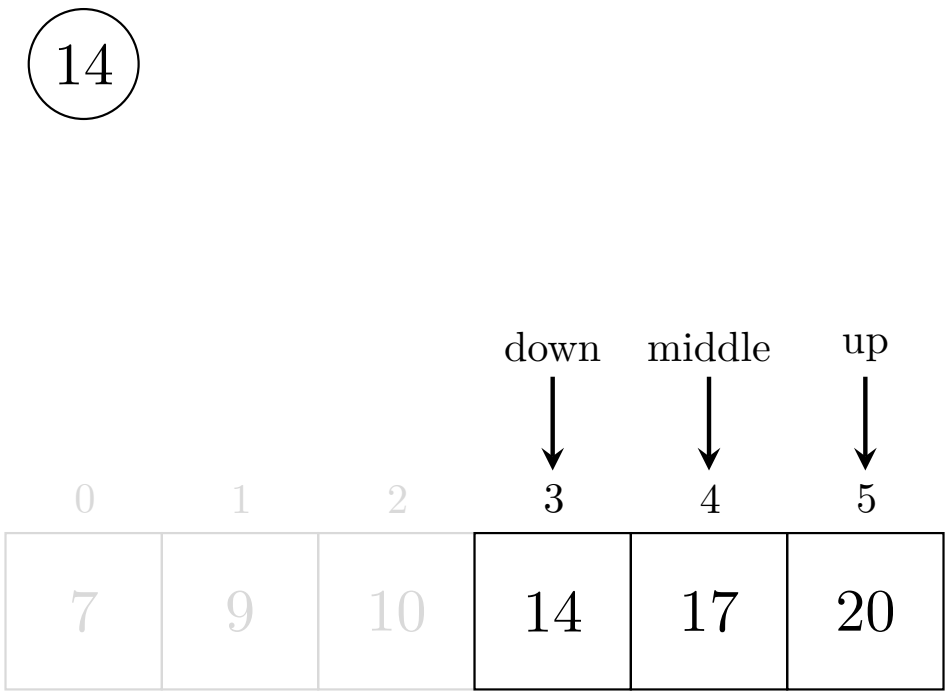
Binary Search

Simulation



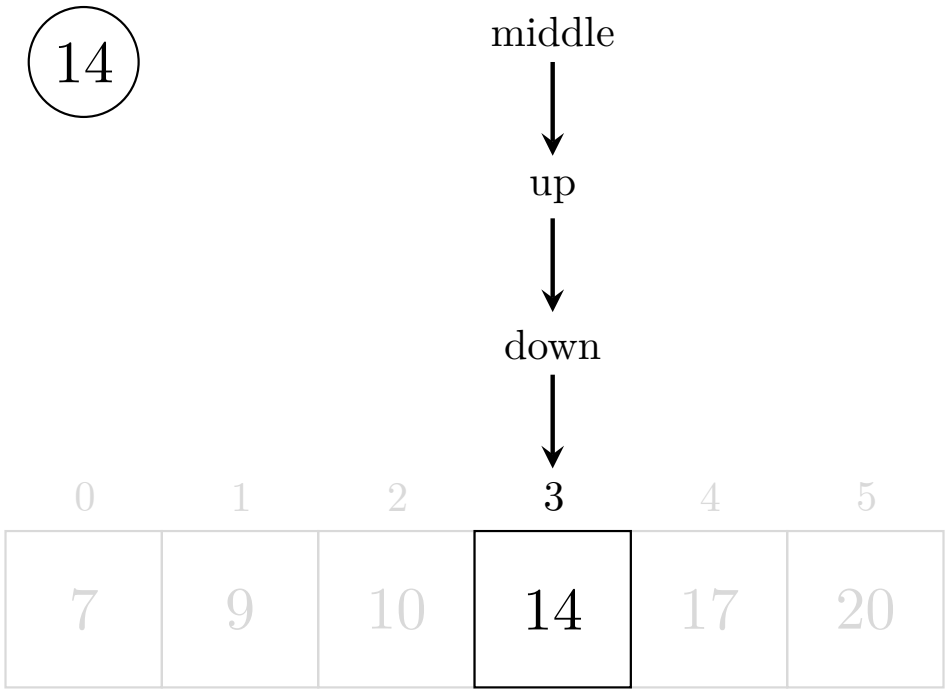
Binary Search

Simulation



Binary Search

Simulation



Binary Search

Implementation

```
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;
}
```

Binary Search

Comment to the Implementation

The variables that indicate the beginning, the end and the middle of the area of the array that needs to be checked have the same names as the arrows in the simulation. The first line of the function that catches the attention is the expression that locates the middle element. Why the function does not calculate the average of the `down` and `up` instead? Unfortunately, calculating the average of those two variables may lead to the integer overflow in case of very large arrays (above one billion of elements). The expression used in the function does not have such a drawback. The second important part of the function is the detection of the case when the desired value is not in the array. It is accomplished in the condition of the loop, when the value of the `down` variable becomes greater than the value of the `up` variable. It means that the area of the array that needs to be checked contains no elements.

Binary Search

Comment to the Implementation — Continuation

The last interesting element of the implementation is an exclusion of the middle element from the area that needs to be checked in the next iteration. The element certainly does not contain the desired value and leaving it in the search area can lead to errors. The `continue` keyword in the second `if` statement prevents the condition in the third `if` statement to be tested. If the condition in the second conditional statement is true the condition in the next `if` statement is surely false and there is no need to check it. If the desired value is not present in the array, the function returns `-1`. If the desired value occurs many times in the array, then the function always finds one of its occurrences, but not necessarily the first one.

The Minimum and Maximum in a Sorted Array

Finding the minimum and maximum becomes trivial in a sorted array. If the values in the array are sorted in an ascending (non-descending) order, the minimum is in its first element and the maximum in the last one. If the values in the array are sorted in a descending (non-ascending) order, the minimum is in the last element of the array and the maximum is in the first one.

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, MSc for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!