Fundamentals of Programming 1

Algorithms for Linear Arrays

Arkadiusz Chrobot

Department of Information Systems

December 5, 2024

Outline

Introduction

In this lecture, several basic algorithms for linear arrays are introduced. Particularly, the sorting and searching algorithms are discussed. They are among the most frequently performed operations by computer systems, and so important, that D. E. Knuth has devoted the entire third volume of "The Art of Computer Programming" to them. Both operations may be applied to many data structures, but in this lecture they are discussed in the context of linear arrays.

Finding the Minimum Value

The Algorithm

Solving some problems may require finding the smallest value in an unsorted array. The algorithm for that task is quite simple:

1. Store the value of the first element of the array in a separate variable. Assume, for now, that it is the minimum value.
2. Visit all other elements of the array, starting from the second one. If the currently visited element has a value smaller than the one stored in the variable, then store the value of the element in the variable.
3. If all elements of the array has been visited, the minimal value is in the variable.

Finding the Minimum Value

The Implementation

The function below implements the algorithm described in the previous slide. It searches for the minimum value in an array of the number of elements given by the constant `NUMBER_OF_ELEMENTS`.

```
int find_min(int *array)
{
    int min;
    unsigned int i;
    min=array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++)
        if(min>array[i])
            min=array[i];
    return min;
}
```

Finding the Maximum Value

The Implementation

The algorithm for finding the maximum value is basically the same as for finding the minimum value. The only difference is the operator used in the conditional statement — less than instead of greater than. The function that implements the algorithm is named accordingly and so is the variable inside the function, that stores the result.

```
int find_max(int *array)
{
    int max;
    unsigned int i;
    max=array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++)
        if(max<array[i])
            max=array[i];
    return max;
}
```

Finding the Extremes

It is easy to spot, that in the case where both values (the minimum and the maximum) are needed it would be better to search for both of them simultaneously than look them up separately. The function below realizes this idea. It passes the minimum and maximum values by the `min` and `max` parameters.

```
void find_exterme_values(int array[], int *min, int *max)
{
    unsigned int i;
    *max = *min = array[0];
    for(i=1; i<NUMBER_OF_ELEMENTS; i++) {
        if(*min>array[i])
            *min=array[i];
        if(*max<array[i])
            *max=array[i];
    }
}
```

Linear Search

The Algorithm

The problem of locating a specific value in an array is very common. There are many variants of this issue. Here, we are interested in finding the first occurrence (starting from the first element) of the value in the array. The algorithm is quite simple. It checks all elements of an array one by one. If it finds the value in one of them it stops and returns the index of that element. Otherwise, if it does not find such a value in any of the elements, it returns a number, for example `-1`, indicating that the array does not contain such a value. Such an algorithm is known as the *linear search algorithm*.

Linear Search
The Implementation

Below is a function that implements the linear search algorithm.

```
int find_value_index(int array[], int value)
{
    unsigned int i;
    for(i=0; i<NUMBER_OF_ELEMENTS; i++) {
        if(array[i]==value)
            return i;
    }
    return -1;
}
```

Sorting
Properties of Sorting

Many sorting algorithms have been invented, that got different properties. The *internal sorting* is applied to data that are stored in the main memory, while in *external sorting* they are in the secondary storage. *Stable sorting* preserves the relative order of the same values. For example, let's suppose that there are two 8s in the unsorted array marked $8'$ and $8''$. If the sorting is stable, then after it is performed the $8'$ still will be before $8''$. In case of numbers, this property is of small value, but it can be useful for more advanced data types. 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 is discussed, but other values, like characters or strings of characters can also be sorted.

Sorting

Example Algorithm

The sorting is discussed with the use of only one sorting algorithm that operates on a linear array. It performs internal, in-place, but unstable sorting. However, it can be modified in such a way, that it will sort in a stable way. The efficiency of this algorithm is rather low — its *time complexity* is expressed as a square of the number of elements of the sorted array. The undeniable advantage of the algorithm is its simplicity. In this lecture, a version of this method is discussed that sorts an array in the non-descending order, but information on how to modify the algorithm to reverse this order is also given.

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 the last two elements the whole array is sorted.

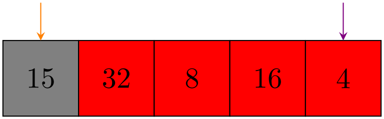
Selection Sort

Simulation

The next slide contains a visualisation of sorting in a linear array of five elements, containing some natural numbers with the use of the selection sort algorithm. The orange arrow specifies the currently sorted element. The violet arrow specifies 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 marked 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.

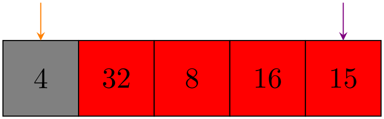
Selection Sort

Simulation



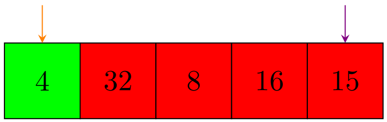
Selection Sort

Simulation



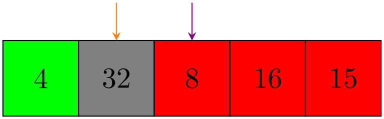
Selection Sort

Simulation



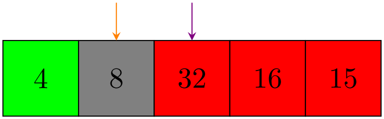
Selection Sort

Simulation



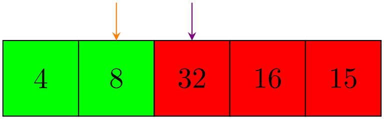
Selection Sort

Simulation



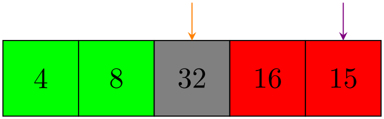
Selection Sort

Simulation



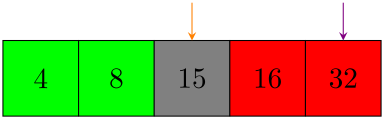
Selection Sort

Simulation



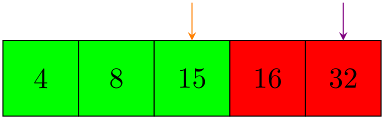
Selection Sort

Simulation



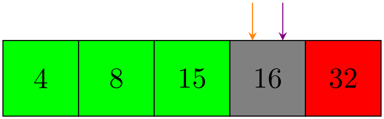
Selection Sort

Simulation



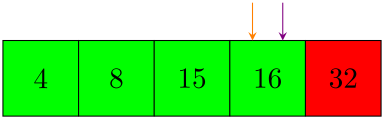
Selection Sort

Simulation



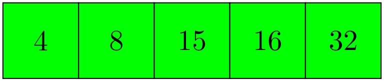
Selection Sort

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 specifies 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 specifies different element than the `i` variable, when the inner loop finishes, then the values of the elements specified by both variables should be exchanged. It is done by the `swap()` function, which was introduced in the previous lecture. If the operator in the conditional statement 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 made possible 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.

Binary Search

Algorithm Description

It turns out, 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

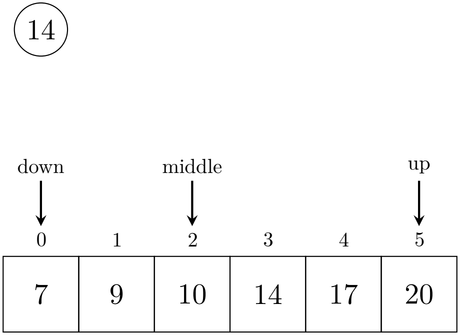
The algorithm repeats the steps until it locates the desired value or the part of the array that needs to be checked becomes so small that it contains no elements. In the latter case the desired valued does not occur in the array. The algorithm always stops, because after every iteration it halves the 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 verify 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 specify 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 specifies the middle element in the checked part of the array. The desired value is in the circle on the left.

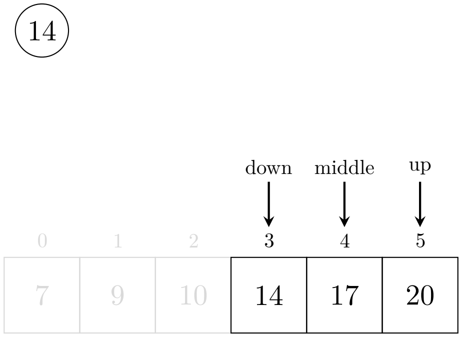
Binary Search

Simulation



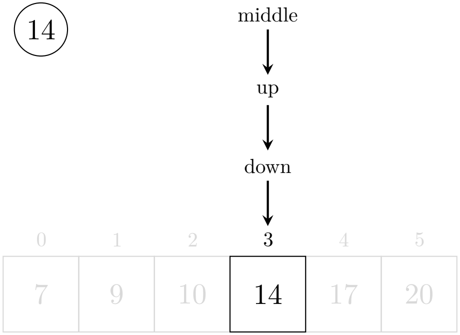
Binary Search

Simulation



Binary Search

Simulation



Binary Search

Implementation

```c
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;
        }
        up = middle - 1;
    }
    return -1;
}
```

Binary Search

Comment to the Implementation

The variables that specify 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 `up = middle - 1;` statement to be executed, because it needs to be performed only when the two conditions that proceed it are false. If the condition in the second conditional statement is true then there is no need execute this statement. 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!