Course 5: Basic sorting methods

* The problem of sorting
* Insertion sort
* Selection sort
* Bubble sort
* Exercises

# The problem of sorting

# It is considered a finite set of objects, each having an associated characteristic, called a *key*, which takes values in a set on which an order relation is defined. *Sorting* is the process by which the elements of the set are rearranged so that their keys are in a certain order.

# *Example 1.* We consider the set of integer values: (5,8,3,1,6). In this case, the sorting key coincides with the value of the element. By ascending sorting we get the set (1,3,5,6,8) and by descending sorting we get (8,6,5,3,1).

# *Example 2.* Consider a table consisting of students' names and grades: ((Popescu,9), (Ionescu,10), (Voinescu,8),(Adam,9)). ˆIn this case, the sorting key can be the name or grade. By ascending order by name we get ((Adam,9),(Ionescu,10),(Popescu,9), (Voinescu,8)) and by descending order by grade we get ((Ionescu,10),(Popescu, 9), (Adam, 9), (Voinescu, 8)).

# To simplify the following presentation, we will consider that the processed set consists of scalar values that represent the sorting keys themselves and that the goal is ascending ordering theirs. Thus, ordering the set is equivalent to finding a permutation of order *n*, *(p(1), p(2),...,p(n))* so that .

# We will also consider that the elements of the set are stored on an information medium that allows random access to the data. In this case it is about *internal sorting*. If the information support only allows sequential access to the data, specific methods must be used in the category of *external sorting* methods.

# On the other hand, depending on the maneuvering space for sorting there is:

# Sort that uses a maneuver area of the size of the data set. If the initial set of data is represented by the table *x[1..n]*, the sorted one will be obtained in another table *y[1..n]*.

# Sorting in the same memory area (sorting in place - *in situ*). The elements of the array *x[1..n]* change their positions so that after the end of the process they are sorted. It is possible to use a memory area in this case too, but this is at most the size of one element and not the size of the entire array.

# In the following, we will only analyze internal sorting methods in the same memory area. Sorting methods can be characterized by:

# *Simplicity*. A method is considered simple if it is intuitive and easy to understand.

# *Efficiency*. A method is considered efficient if it doesn’t require a large amount of resources. From the point of view of memory space, an in situ method is more efficient than one based on a maneuvering area size of the array. From the point of view of execution time, it is important to perform as few operations as possible. In general, it is taken into account only operations upon the array (comparisons and permutations).

# *Naturalness.* A method is considered natural if number of operations decreases once with *distance* between initial array and the sorted one. A measure of that distance can be number of inversions of permutations corresponding to initial array.

# *Stability.* A sorting method is considered stable if the relative order of the elements that have the same key value does not change during the sorting process. For example, if a stable method of descending order by grade is applied to the grade table, ((Ionescu,10),(Popescu,9),(Adam,9),(Voinescu,8)) is obtained, while if it is not stable we obtain ((Ionescu,10),(Adam,9),(Popescu,9),(Voinescu,8)).

# Further, we will consider some *basic* methods of sorting that are very simple, they are not the most efficient but they represent a good starting point. For each of them will be presented: principle, correctness check and complexity analysis.

# Insertion sort

# This method princple is:

# *Starting with the second element, each is inserted at the appropriate position in the substring that precedes it.*

# At each stage of sorting, for is searched its appropriate position in the destination substring *x[1.. i − 1]* (which is already ordered) comparing *x[i]* with the elements of *x[1..i − 1]* starting with *x[i − 1]*¸ and creating space by shifting to the right elements greater than *x[i]*. Thus, the general structure of the algorithm is:

# 

# There are other methods of implementing the method. Thus, to avoid performing the comparison *j 1* at each internal iteration, the value of *x[i]* is placed on position 0 in the array *x*, this position playing the role of a *pennant*(*fanion*). Thus, at the latest when *j=0,* *x[j] = x[i]* is reached. This variant can be described as follows:

# 

# Correctness check. The precondition of the problem is *{ n 1}* and the postcondition is *{x[1..n]* is ordered ascending*}*. For the outer cycle (after *i*) we prove that the invariant property is *{x[1.. i-1]* is ordered ascending}. At the beginning of the cycle *i = 2*, so *x[1..1]* is implicitly increasing. At the end of processing *(i = n + 1)*, the invariant obviously implies the postcondition. It remains to show that the property remains true even after performing the processing within the cycle. For this it is enough to show that for the inner cycle (after *j*) the assertion *{x[1..j]* is increasing and *aux x[j + 1] = x[j + 2] ... x [i]}* is invariant. At the end of the WHILE cycle, this property would imply one of the relationships:

# in case when exit cycle condition is *aux x[j];*

# in case when exit cycle condition is j = 0.

# Any of these two relations would lead by assigning *x[j + 1] ← aux to x[1] ≤ . . . ≤ x[j] ≤ x[j + 1] ≤ x[j + 2] ≤ . . . ≤ x[i]* and by moving to the next value of the counter *(i ← i + 1)* to the fact that *x[1..i − 1]* is increasing.

# It remains only to justify the invariant of the WHILE cycle. At the beginning of the cycle, the relationships take place: *j = i − 1, aux = x[i]* so *aux = x[j + 1] = x[i]* and since *x[1..i − 1]* is increasing, it follows that the proposed invariant property for WHILE is satisfied. We show that it is not altered by the processing within the cycle: if *aux < x[j]*, by assigning *x[j + 1] ← x[j]* we obtain: *aux < x[j] = x[j + 1] ≤ . . . ≤ x[i]* and after *j ← j − 1* the statement will be true: *{ x[1..j]* increasing and *aux < x[j + 1] = x[j + 2] ≤ . . . ≤ x[i]*. As *x[j + 1] = x[j + 2]* by *x[j + 1] ← x[j]* no information is lost from the array.

# Algorithm stop is assured by using a counter.

# Complexity analysis. We will take in consideration only operations of comparison and permutation upon elements of the array. Let be the number of comparisons and permutations respectively upon the elements of the array for each Best case corresponds to array being sorted in ascending order and worst case being descending.

# In best case: and (we talk about number of operations that imply using the helping variable *aux*) so In worst case and , obtaining

# Thus , meaning that insertion sort fits in classes Ω(n) and O().

# Other properties. Insertion sort is natural as long as the condition of interior loop is *aux x[j]* and is stable. If it is *aux x[j]* method is not stable anymore.

# Selection sort

# Method principle is:

# *For each position i, starting with the first one, the smallest element is selected from the subset starting with that position and placed in that place(through interchange with current element on position i).*

# General structure of the algorithm:

# 

# Detailed structure:

# 

# Correctness check. We show that an invariant of the outer loop (after *i*) is: *{x[1.. i-1]* is ordered ascending and *x[i-1] x[j] for* }. At the beginning *i=1* so *x[1..0]* is an empty string. The inner FOR loop (after *j*) determines the minimum value in x[i..n]. This is placed by interchange on position i. It is obtained in this way that x[1..i] is ordered ascending and that *x[i]* *x[j]* for . After incrementing *i* (at the end of the loop after *i*), the invariant property is regained. At the end, *i=n* and the invariant leads to ascending *x[1..n-1]* and *x[n-1] x[n]*, i.e. *x[1..n]* is ascending.

# Complexity analysis. No matter the initial arrangement of the elements, number of comparisons is:

# 

# In best case (set being sorted in ascending order) number of interchanges is In worst case, for each *i* a interchange is made, so the cost of all interchanges is Thus, selection sort fits class: Θ().

# Other properties. Algorithm is partially natural(number of comparisons does not depend on order of sorting the set). In shown variant (when minimal value is interchanged with current position) algorithm is not stable. If instead of a single interchange would be made a shift to the right with one position of the elements of subarray *x[i.. k-1]* and *x[k]* would transfer to *x[i]*, algorithm would have been stable.

# Bubble sort

# Method principle is:

# *The array is traversed and neighboring elements are compared, and if they are not in the correct order, they are exchanged. The traversal is resumed until no more interchange is necessary.*

# General structure of the algorithm:

# 

# Consider the interchange of adjacent elements when they are in incorrect order:

# 

# Using the property *{x[i] x[j], }* as an invariant of the repetitive processing, it can be shown that the above processing leads to the satisfaction of the postcondition: *{x[n] x[i], }*. For *i = 1* the invariant property is true because *x[1] x[1]*. We assume that the property is true for *i*. If *x[i] x[i+1]* then no processing is performed and it remains true for *i+1* as well. If on the other hand *x[i] > x[i+1]* then the interchange is performed so that *x[i] < x[i+1]* so the property becomes true for *i+1* as well.

# Based on this property of the sequence of interchanges, it is deduced that it is sufficient to apply this processing successively for *x[1..n], x[1..n-1], ..., x[1..2]*. It follows that the algorithm can be described as:

# 

# Correctness check. Since it was demonstrated that the effect of the inner loop is that it places the maximum value on position *i*, it follows that for the outer loop the property {*x[i + 1..n]* is increasing and *x[i+1] x[j]* can be considered as invariant for }.

# Complexity analysis. Number of performed comparisons does not depend on array initial order of sorting, being in any situation:

# 

# The number of interchanges depends on the properties of the string as follows: in the best case is obtained and in the worst case is obtained (an interchange requires 3 assignments) . Thus, the number of processings satisfies: that is, the algorithm represented above belongs to the class Θ().

# However, the algorithm can be improved by reducing the number of comparisons performed, in the sense that it is not always necessary to go through the table for . For example, if the table is ordered from the beginning, a single traversal would be sufficient to verify that no exchange is necessary. Starting from this idea, we arrive at the algorithm:

# 

# For this algorithm, the number of comparisons performed in the best case is and in the worst case is . Regarding the number of interchanges is the same as for the first version of the algorithm, . Therefore, the total number of repetitions of the analyzed processes satisfies:

# 

# meaning that algorithm fits in Ω(*n*) and Θ().

# Since an ascending subarray is formed at the end of the array, it follows that it is no longer necessary to make comparisons in that portion. This portion is limited below by the highest index for which the exchange was made. Thus the algorithm can be rewritten as:

# 

# As long as condition for interchange is specified through strict inequality ( *x[i]* > *x[i+1]* )*,* any of algorithm’s variants are stable.

# Comparison value of number of processings of each algorithm described above:

# 

# Exercises

# Propose an algorithm that an array *y[1..n]* in ascending order starting from the array *x[1..n].*

# Consider an array whose elements contain 2 types of data: name and grade. Sort in descending order by grade, and for the same grade in ascending order by name.

# Analyze the complexity of algorithm interschimbari3.

# Consider a matrix with *m* rows and *n* columns of real elements. Reorganize the matrix through interchanges of rows and column so that elements of the main diagonal are sorted in ascending order.