Course 7: Algorithm development techniques: reduction and division

* Motivation
* Recursive algorithms
* Technique of reduction
* Technique of division
* Exercises

# Motivation

# Consider the problem of solving

# Algorithm for solving would be:

#

# Algorithm has complexity of Θ(*n*)*.* But if the hypothesis is used, it is observed that because . Therefore, to calculate , it is enough to calculate and it amounts to square. In turn, the calculation of can be reduced to the calculation of and to a squaring, etc. The decomposition can continue until is reached, whose calculation is simple.

# Algorithm can be described as:

#

# Using as an invariant for repetitive processing the statement: it can be easily demonstrated that the *putere2* algorithm calculates . On the other hand, it can be seen that the processing is of complexity *Θ(m) = Θ(lg(n))*. The reduction of complexity derives from the fact that at each stage the value of a single factor from the two is calculated, since they are identical. The idea of solving this problem is common to reduction and division techniques which are based on solving a problem of size *n* on solving one or more similar problems but of smaller size. The size reduction continues until a problem of small enough size is reached to be solved directly (for example *n=2* or *m=1*). A description of this solution method that better illustrates the idea of size reduction is:

#

# If the values *x* and *m* are transmitted, the algorithm for the calculation of can be described by:

#

# The last two algorithms have a particularity: within the processing they perform there is also a self-call (the same function is called within the function). Such algorithms are called *recursive*.

# The above example illustrates the fact that using the idea of reducing the solution of a problem to solving a similar but smaller problem can lead to a reduction in complexity. On the other hand, there are problems for which the solution approach in this way is easier, leading to intuitive algorithms.

# It should also be mentioned that the techniques in this category do not always lead to a reduction in complexity if the algorithm is implemented on a sequential machine. An example in this sense is the calculation of the factorial (as will be demonstrated later, applying the reduction technique yields an *Θ(n)* complexity algorithm, just like applying the brute force technique).

# We consider the problem of determining the maximum from a finite sequence of real values stored in the array *x[1..n]*. Applying the idea of division, it turns out that it is enough to determine the maximum in the subarray *x[1.. [n / 2]]* and the maximum in the subarray *x[[n / 2] + 1 ..n]*. The result will be the highest of the obtained values. The algorithm can be described recursively as follows:

#

# It is not difficult to notice that the order of complexity of this algorithm is also *Θ(n)* as in the case of the classical algorithm. This is true if the machine on which the algorithm will be executed is a sequential one. If, however, the machine is parallel (at a given moment, several processes can be performed simultaneously), then through such an approach, we win some time.

#

# Recursive algorithms

# The last algorithms presented in the previous section are part of the category of recursive algorithms. They are used to describe processing that can be specified by themselves. A recursive algorithm is characterized by:

#  *Stop condition.* It specifies the situation in which the result can be obtained by direct calculation without the need to call the same algorithm.

# *Self-call.* It is called at least once for other parameter values. The values of the parameters corresponding to the sequence of calls must ensure the closeness to the satisfaction of the stop condition.

# As a result of the cascade of self-calls, a recursive algorithm actually performs a repetitive processing, even if this is not explicit. A simple example of repetitive processing described recursively is the one corresponding to the determination of the LCD(least common divisor) of two non-zero natural numbers.

#

# and the one based on the subtraction operation by:

#

# The above examples are characterized by simple and direct recursion. A recursive algorithm is considered simple recursive if it contains a single self-call and multiple recursive if it contains two or more self-calls (for example the *maxim* algorithm from the previous section).

# There is also the possibility that the algorithm does not call itself directly but indirectly through another algorithm. For example, algorithm *A1* calls algorithm *A2*, which calls algorithm *A1*. In this case it is indirect recursion.

# The notion of recursion can also be used in the context of defining some concepts. For example, the concept of arithmetic expression can be defined as follows: "an arithmetic expression is made up of operands and operators; an operand can be a constant, a variable or an *expression*". Recursive description allows the specification of infinite structures using a finite set of rules.

# Call trees. To illustrate the way an iterative algorithm works, it can be useful to graphically represent the structure of calls and returns with the return of the obtained result. In the case of an arbitrary recursive algorithm, the call structure is hierarchical leading to a call tree. In the case of simple recursion, the call tree degenerates into a linear structure.

# Checking the correctness of recursive algorithms. If the recurrence relation that describes the connection between the solutions corresponding to the different instances of the problem is correct, then the algorithm that implements it is also correct. On the other hand, since a recursive algorithm specifies an implicit repetitive processing, to prove its correctness it is enough to show that:

# There is an assertion regarding the state of the algorithm that has the following properties: (i) it is true for the particular case (when the stopping condition is satisfied); (ii) is true when returning from the recursive call and performing local processing; (iii) implies the postcondition.

# The stopping condition is satisfied after a finite sequence of recursive calls.

# For the cmmdcr1 algorithm, an invariant assertion is *{rez = cmmdc(a, b)}*. For the particular case *b = 0* we have *rez = a = cmmdc(a, 0) = cmmdc(a, b)*. If *rez = cmmdc(a, b)* before the recursive call because *cmmdc(a, b) = cmmdc(b, aMODb)* it results that *rez = cmmdc(a, b)* after the recursive call. For the *cmmdcr2* algorithm, the invariant property is also *{ rez = cmmdc(a, b)}*.

# Analysis of the complexity of recursive algorithms. Consider a problem of size *n* and recursive algorithm is of form:

#

# with *h(n)* a decreasing function with the property that there exists *k* with .If the processing of *P1* has a constant cost and the determination of *h(n)* has the cost *c*, then the cost of the algrec algorithm can be described by:

#

# Therefore, the execution time of a recursive algorithm satisfies a recurrence relation. To determine the expression of *T(n)* starting from the recurrence relation, one of the methods can be used:

# *Substitution method (direct)*. Starting from the recurrence relation, the general form of *T(n)* is intuited, after which the validity of the expression of *T(n)* is demonstrated by mathematical induction.

# The method of iteration (inverse substitution). Write the recurrence relation for *n, h(n), h(h(n)), ...* , after which successively substitute *T(h(n)), T(h(h(n)))* etc. From the relationship obtained, based on some algebraic calculations, the expression of *T(n)* results. The method is also known as the inverse substitution method.

# *Example 1.* Consider the case: *h(n) = n – 1.*

# By iteration method we obtain:

#

# By summing up all the relations and reducing the corresponding terms, we obtain: *T(n) = c(n - ) +* for *n > .*

# *Example 2.* Consider the recursive algorithm (*putere3*) for solving . If we note the number of multiplications performed with *T(n)*, we obtain the recurrence relation:

#

# Apllying iteration technique we obtain:

#

# As the number of relations above is m, by summing them all and reducing the corresponding terms, *T(n) = m = lgn* is obtained.

# *Example 3.* We analyze the *maxim* algorithm described in the previous section. Denoting with *T(n)* the number of comparisons performed, the recurrence relation is obtained:

#

# If n is not a power of 2, the iteration method is more difficult to apply. We start from the particular case for which the recurrence relation leads to *T(n) = 2T(n / 2) + 1* for obtaining the sequence:

#

# Multiplying the relations with the factors in the last column, summing the relations and making the reductions, we get: . This result is only valid for . To show that it is true for anything, we apply mathematical induction. Obviously *T(1) = 1 - 1 = 0*. We assume that *T(k) = k - 1* for any *k < n*. It results that

# Observation. It can be shown that if for , *T(n)* is increasing for large arguments and *f(n)* has the property for *c* a constant then for any *n.*

# Technique of reduction

# Principle. The reduction technique is based on the relationship that exists between the solution of a problem and the solution of a instance of reduced size of the same problem. As a rule, the size reduction is based on decreasing of a constant (in most cases 1) from the size of the problem.

# Factorial calculation. The simplest example is that of the factorial calculation. The starting relationship is:

#

# The algorithm can be described in the recursive version as follows:

#

# Denoting with T(n) the number of multiplication operations performed, the recurrence relation is obtained:

#

# From this relation it can be intuited that *T(n) = n*. We prove this by mathematical induction after n. For *n = 1* we obtain *T(1) = T(0) + 1 = 1*. We assume that *T(n - 1) = n - 1*. From the recurrence relation we will obtain *T(n) = T( n - 1) + 1 = n*. So, indeed, *T(n) = n*. The same result is obtained by applying the iteration technique:

#

# By summing up all the relations and performing the reductions, we obtain: *T(n) = n.*

# Generating permutations. Let's consider the problem of generating permutations of order *n*. There are several ways to apply the reduction technique. A variant starts from the idea that a permutation of order *n* can be obtained from a permutation of order *n - 1* by placing the value *n* on all the possible *n* positions. Thus if n = 3 there are two permutations of order n - 1 = 2: (1, 2) and (2, 1). For each of these the value 3 can be inserted in each of the three possible positions: first, second and third leading to (3, 1, 2), (1, 3, 2), (1, 2, 3) respectively at (3, 2, 1), (2, 3, 1), (2, 1, 3). This idea would correspond to a „bottom-up” approach - starting from a permutation of a given order, a permutation of an immediately higher order is built.

# For a „top-down” approach to the problem (a permutation of order *n* is specified by a permutation of order *n - 1*) let's observe that to generate all permutations of order *n* it is enough to successively place all the values in position *n* possible (from *{1,2, ..., n}*) and for each value thus placed, we generate all the permutations corresponding to the values found on the first *n - 1* positions. To generate them, permutations of order n-2 are used until permutations of order 1 are reached (a single permutation consisting of the value in position 1).

# To describe the algorithm, let's assume that the permutations will be obtained in an array *x[1..n]* accessed in common by all (self)calls of the algorithm and initialized so that *x[i] = i*. When *x* contains a permutation it is displayed.

#

# Applying this algorithm, the permutations of order 3 are obtained in the following order: (2, 3, 1), (3, 2, 1), (3, 1, 2), (1, 3, 2), (2, 1 , 3), (1, 2, 3).

# To analyze the complexity of the algorithm, we count the number of interchanges performed and observe that it satisfies:

#

# By applying the iteration technique we obtain:

#

# Multiplying each relation by the factors specified in the last column and adding up all the relations yields: . Therefore, for *k = n* we obtain .

# Towers of Hanoi. We consider three rods placed vertically and identified by *s* (source), *d* (destination) and *i* (intermediary). On rod *s* are placed *n* discs in descending order of their radius (first disc has maximal radius). It is requested to transfer all the discs to the rod *d* so that they are placed in the same order. The rod can be used as an intermediary with the restriction that at any moment there are only discs of smaller radius above a disc. The idea of the solution is: "transfer disks from *s* to *i* using *d* as an intermediary; transfer the remaining disk from *s* directly to *d*; transfer the *n - 1* disks from *i* to *d* using *s* as an intermediary". This idea can be simply described recursively:

#

# In the Hanoi algorithm, the first argument specifies the number of discs to be transferred, the second indicates the source rod, the third the destination rod and the last one the rod used as an intermediary. The processing specifies that the disk will be transferred from the *s* rod to the *d* rod.

# To analyze the complexity, we count the number of disk moves. It is observed that:

#

# By applying the iteration technique we obtain:

#

# Summing up the relations, it turns out that meaning, the algorithm has the order of complexity .

# Multiplication „a la russe”. The size reduction can be achieved not only by decreasing a constant but also by dividing by a constant. The most frequent case is that of dividing the size of the problem by 2. A simple example of this is the multiplication "a la russe". The calculation rule in this case is:

#

# Starting from this relation, the algorithm can be described by:

#

# To analyze the complexity of the algorithm, we consider that the size of the problem is represented by the pair *(a, b)* and the dominant operation is the division of *a* by 2 (the other operations: multiplying *b* by 2 and addition are not performed more than once). Thus, the recurrence relation can be written for the execution time *T(a, b)*:

#

# By applying the iteration technique for particular case we obtain:

#

# Summing up the relationships results in *T(a, b) = k + 1 = lga + 1*. It can be seen that the execution time depends only on the value of a and for we have . Since *T(a)* is increasing for large values of a and it follows that the result is also valid for the general case.

# Technique of division

# Principle. Technique of division (also called “divide et impera” or “divide and conquer”) consists of:

# *Decomposition into subproblems*. A problem of size *n* is decomposed into two or more subproblems of smaller size. The classic case is when the subproblems have the same nature as the initial problem. Ideally, the sizes of the subproblems should be as close as possible (if the problem of size *n* is broken down into *k* subproblems, it is preferable that they have sizes close to ). For this technique to be effective, the subproblems to be solved must be independent (the same subproblem must not be solved more than once).

# *Solving subproblems*. Each of the independent subproblems obtained by division is solved. If they are similar to the initial problem, then the division technique is applied again. The division process continues until subproblems of a sufficiently small size (critical size, ) are reached to be solved directly.

# *Combining the results*. In order to obtain the answer to the initial problem, sometimes the results obtained for the subproblems must be combined.

# The general structure of an algorithm developed using the division technique is:

#

# Examples. In practice, *k = 2* and are most often used. The algorithm for determining the maximum presented in the first section is based on the division technique for *k = 2* and the critical size . The subproblems are independent and the composition of the results consists of processing "".

# Let's consider the problem of searching for a value *v* in an array *a[1..n]* sorted in ascending order.

# we choose an element *a[j]* from *a[1..n]* that is compared with *v.*

# if *v = a[j]* then searched element was found.

# If *v < a[j]* then the search continues in subarray else it continues in subarray

# The most natural choice for *j* is to be as close as possible to the middle of the array. On the other hand, it can be seen that for this problem only one of the two subproblems must be solved. The critical dimension is in this case .

# The algorithm developed in this way, called the *binary search* algorithm, can be described as follows:

#

# with *s* and *d* delimiting the area in the array where the search is concentrated at a given moment. At the beginning, the entire table is searched so that at the first call we have and . The algorithm can also be described in the iterative version:

#

# In order to avoid comparing the searched value twice with array elements *(a[m] = v* and *v < a[m])*, the algorithm can be rewritten in the following form:

#

# To analyze the binary search algorithm, let's consider that *T(n)* represents the maximum number of comparisons made on the array elements (it is reached in the worst case, when *v* is not in the array). The corresponding recurrence relation is:

#

# To establish the value of *T(n)*, let's first analyze the particular case . Applying the iteration method we obtain:

#

# Summing the *m + 1* relations, we obtain *T(n) = m + 1 = lgn + 1*, for . For random *n*, the relationship is demonstrated by mathematical induction. We assume that *T(k) = [lgk] + 1* for any *k < n* and show that *T(n) = [lgn] + 1*. We treat separately the cases: (i) *n = 2k*; (ii) *n = 2k + 1*. In the first case, we obtain:

#

# In second case we obtain:

#

# using relation for .

# Thus, binary search is of complexity *O(lgn)*.

# Observation. The binary search algorithm can rather be seen as an example of the application of the reduction technique (by dividing the problem into two subproblems of which only one can be solved).

# Master method. It represents a technique for analyzing the algorithms developed through the division technique. We assume that a problem of size *n* is decomposed into *m* subproblems of size of which it is necessary to solve . We consider that the division and composition of the results together have the cost and the cost of solving in the particular case is . With these assumptions, the cost corresponding to the algorithm verifies the recurrence relation:

#

# The determination of *T(n)* starting from this recurrence is facilitated by the following theorem. The master theorem. If then:

#

# *Example.* For the binary search algorithm we have . Apply the second case of the master theorem and obtain . For the *maxim* algorithm we have , so the third case of the theorem is applied, obtaining *T(n) = n*.

# Exercises

# Propose a more efficient algorithm than *putere1* for solving for an natural random number *n.* Establish the complexity of proposed algorithm.

# Propose an algorithm for solving , with *A* being square matrix of order *n.* Algorithm must have a complexity less than

# Modify algorithm *maxim* ­­so that it allows determining maximal value and also minimal. Establish the number of performed comparisons. Is the obtained algorithm more efficient than the one that consists in the separate determination of the minimum and the maximum.

# Write an algorithm that implements the following idea for the simultaneous determination of the minimum and maximum value in an array: The elements of the array are grouped into *[n/2]* pairs. For each pair, the minimum value is transferred into a set with minimums and the maximum value into -a set with maxima. The same processing is applied to the two sets, eliminating the large values from the set of minimums and the small values from that of the maxima. The process continues until these sets contain one element each. Analyze the proposed algorithm.

# Write a permutation generation algorithm based on the "bottom-up" approach and analyze its complexity.*Indication.* The algorithm will be called with *perm(1)* and can be described as:

# Modify the algorithm of binary search so that it return the position of searched value in case when that is present in the array or return -1 if is not in the array.