Моделирование и анализ информационных систем. Т. 24, № 4 (2017), с. 481–495
Modeling and Analysis of Information Systems. Vol. 24, No 4 (2017), pp. 481–495

DOI: 10.18255/1818-1015-2017-4-481-495

UDC 510.52:372.851

Automated System for Teaching Computational Complexity of Algorithms Course

Rublev V. S., Yusufov M. T.

Received April 14, 2017

Abstract. This article describes problems of designing automated teaching system for "Computational complexity of algorithms" course. This system should provide students with means to familiarize themselves with complex mathematical apparatus and improve their mathematical thinking in the respective area. The article introduces the technique of algorithms symbol scroll table that allows estimating lower and upper bounds of computational complexity. Further, we introduce a set of theorems that facilitate the analysis in cases when the integer rounding of algorithm parameters is involved and when analyzing the complexity of a sum. At the end, the article introduces a normal system of symbol transformations that allows one both to perform any symbol transformations and simplifies the automated validation of such transformations. The article is published in the authors' wording.

Keywords: automated learning, algorithm complexity analysis, algorithm's symbol scroll table, normal system of symbol transformations

For citation: Rublev V. S., Yusufov M. T., "Automated System for Teaching Computational Complexity of Algorithms Course", *Modeling and Analysis of Information Systems*, **24**:4 (2017), 481–495.

About the authors:
Vadim S. Roublev, orcid.org/0000-0002-0252-9958, PhD,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003, Russia, e-mail: roublev@mail.ru

Murad T. Yusufov, orcid.org/0000-0002-0362-2951, graduate student,
P.G. Demidov Yaroslavl State University,
14 Sovetskaya str., Yaroslavl, 150003, Russia, e-mail: flood4life@gmail.com

Introduction

One of the most important aspects of educating a Computer Science specialist is teaching them how to analyze computational complexity of algorithms. This enables them to choose the algorithm that is most appropriate for a certain set of conditions and predict the time necessary for the program execution. Success in this aspect requires a lot of individual learning. However, a large amount of individual assignments forces the professor to spend a lot of personal time. A possible solution for this problem is a certain set of computer programs that allows to manage individual learning for each student, which is also known as automated teaching system. Remote education is an alternative education method that does not require the student's personal contact with his professor. Ability to study in comfortable hours and places provides current popularity growth of various remote education methods.

1. Approaches to designing an automated teaching system

Existing automated teaching systems are fine for teaching areas of knowledge focused on definitions and shallow connections between them. But algorithm analysis requires development of logical-mathematical thinking; there is, therefore, a need for smart teaching systems that are focused on development of this kind of thinking and acquiring erudition and skills in a complex area of knowledge [1].

There is a large amount of program systems called teaching ones (e.g., Moodle, Claroline, Dokeos, ATutor), but most of them do not support full cycles of teaching (methods), because they are just applications that provide access to texts, allow to take part in some tests and check up whether the user has passed those tests [2]. A more advanced solution is to use various techniques that modify the system behavior towards each user depending on their individual traits. Adaptive teaching systems (ATS), whose primary paradigm is adapting to every user, is a suggested solution to this issue. Adaptive technologies for teaching is a relatively recent development but they have already become popular with teaching systems developers. The following techniques are the base adaptive teaching paradigm:

- building the course teaching sequence;
- smart analysis of user's solutions;
- interactive support during problem solving;
- problem solving support using examples;
- adaptive support for navigation;
- adaptive presentation;
- adaptive support for users collaboration.

Applying these techniques secures the system flexibility in interacting with its users and in presenting the material for studying. An addition to the concept of adaptive teaching systems is the following proposition:

Proposition 1. *Learning can be reduced to an aggregate of following pieces:*

- *the information for studying;*
- *control events that allow to check knowledge of that information;*
- *a method for assessing the quality of knowledge;*
- *the following management, the most important and complex component that makes the system actually teaching.*

Therefore, we have to provide answers for the following questions: how to perform partitioning, how to check up knowledge and how flexible should the system be in interaction with its users.

Adaptive teaching method aligns well with these postulates about the learning process:

- any information that is too hard to understand can be divided to a sequence of such smaller chunks so that each chunk can be understood when all previous chunks are understood too;
- any chunk of information has a certain finite amount of such tests so that if the student has successfully passed all of them then it is fairly certain that they have absorbed the chunk.

Therefore, it is possible to partition both information that has to be taught and control tests in such sections so that each section is a couple of a chunk of information and a set of control tests related to this chunk.

The whole information in "Computational Complexity of Algorithms" course of study can be divided into two large categories. The first category includes sections on basic theoretical knowledge and definition of algorithm, computational complexity and asymptotic estimation of complexity. The main goal of these sections is development and training student's memory using memorization techniques. Tests are the most common control method for these sections.

However, the course of study also covers teaching of informal usage of mathematical apparatus and this presents additional difficulties. Those are technical difficulties related to formula input and transformation and to insufficient level of logical thinking of the student who has to reach some result by building a transformation sequence. So, the second section is focused on mathematical methods for estimating algorithm computational complexity and logical-mathematical thinking development.

The following traits distinguish the second category from the first one:

1. Tests cannot be the only control method because they cannot test students' ability to apply various mathematical transformations;
2. The system has to teach the student to combine single transformations into a directed process by building a sequence of previously learned transformations;
3. The system has to be able to control students' ability to connect multiple processes when solving the final problem on estimating algorithm computational complexity.

One of the tools used in the control events of sections from the second category is the algorithm for verifying symbol transformations.

The most important component of the system is the one that carries out the information and control events interaction. It determines the volume of information for a single session, set and amount of control events and other parameters of the system. It is this component that adds flexibility to the system and distinguishes adaptive teaching system from the basic ones. The next few paragraphs describe a suggested use case scenario of the system and the user interaction.

The user logs in into the system and sees a list of the course sections that may be available or unavailable depending on the user's progress in the course. The user has

to pass the sections sequentially (a principle of linear learning). After the user enters a section they have chosen they see the information it contains. Then the control event starts. To pass it the user has to complete a certain amount of multiple choice tests defined for each section. The number of answers for each question is determined by the developers of the specific tasks. That is why this number may be different for different questions and their choice depends on the program itself. The presented answers may contain either only correct answers or only incorrect ones or both.

The user has to tick all the correct answers missing incorrect ones. If the user has not chosen all the correct answers or has chosen some of the incorrect ones, the system will display a warning text ("Not all correct answers are selected", "Some selected answers are incorrect" or a combination of both) and a chunk of material related to the question. Now the user has an opportunity to fix the answer. If the answer is incorrect again, the system removes this question and adds two new questions. So, the amount of questions required to complete the section may increase. If this amount grows too much, the current session ends. If this happens too many times in a row, the system temporarily disables the user's account, which means a visit to the professor.

If the user has a lot of penalty questions but has a long streak of correct answers, the system begins to remove them according to some progression. Such an approach makes the user study the section information carefully. Thus, it impacts both the control events and the learning process. The user can move on to the next section only after completing all the questions in the current section.

However, there are some sections whose information relies on that from the previous ones, hence it is impossible for the students to absorb it unless they have mastered the previous material. Such sections require to additionally control the level of knowledge of information from previous sections. In this case it could be enough to answer only one control question.

2. Computational complexity of algorithms and techniques for its estimation

This section deals with the subject area closely related to the material above. It starts with the definition of algorithm computational complexity. Generally, it is the time (amount of steps) necessary for the algorithm to finish; it usually depends on some input parameters. Despite that in some cases also require memory usage analysis, this section considers only time-wise complexity.

The O-notation is a common tool that allows to estimate the amount of steps $T(n)$, where n is an input parameter, growth speed estimation: $T(n) = O(f(n))$. This expression means that the upper bound of the $T(n)$ growth can be expressed through $f(n)$, i.e.:

$$\exists C > 0, n_0 \; \forall n \geq n_0 : \; T(n) \leq C \cdot f(n),$$

while Ω-notation: $T(n) = \Omega(g(n))$ expresses the lower bound of the $T(n)$ growth through $g(n)$, i.e:

$$\exists C > 0, n_0 \; \forall n \geq n_0 : \; T(n) \geq C \cdot g(n).$$

These bounds may be both too high and too low. E.g., for $T(n) = n^2 - n + 3$ estimations $T(n) = \Omega(n), T(n) = O(n^3)$ are correspondingly too low and too high. So, the most accurate bounds estimations are done using the Θ-notation [2]: $T(n) = \Theta(f(n))$ that expresses both the upper and the lower bounds with the same $f(n)$ function, i.e.:

$$\exists C_1 > 0, C_2 > 0, n_0 \; \forall n \geq n_0 : \; C_1 \cdot f(n) \leq T(n) \leq C_2 \cdot f(n).$$

Note: for the example above, $T(n) = \Theta(n^2)$.

It is very important to choose a proper set of $f(n)$ functions used in computational complexity estimation. The following set of functions is commonly accepted in the theory of algorithm complexity:

$$\log n, \; n^{m/k}, \; 2^n,$$

where n is an algorithm parameter, which is usually an integer, and m, k are some arbitrary integer constants. Any possible superposition of these functions is also a valid estimation of algorithm complexity. E.g.,

$$\log \log n, \; \log^{1/2} n, \; n^{3/2}, \; n^{\log n}, \; 2^{2^n}.$$

If the algorithm has several input parameters, then the Θ-notation is defined via the superposition of functions of each parameter, e.g. $T(n, m, k) = \Theta(2^n m \log k)$.

The next several paragraphs describe the method for estimating the bounds of algorithm computational complexity. Its goal is to get Θ-notation expressed bounds if possible, or O-notation and Ω-notation ones. Since this estimation is strongly related to loop complexity estimation, it makes sense to describe the estimation process for a single loop.

The basis for algorithm complexity estimation is a symbol scroll table that helps defining the complexity by the amount of loop executions. Each variable of the algorithm has a corresponding column in the table. The table also contains several special columns:

- for each loop there is a column with the index of this loop and a symbol for the last time when the loop is executed;
- column *loop condition* that shows symbol condition of the loop. It has comment *last* for the loop last execution condition and comment *exit* for the loop exit condition.

These conditions use the symbol value of the loop index for its last execution. These conditions provide two estimations for the loop execution amount defined by a symbol: the upper and the lower bounds. The analysis of these estimations allows to determine algorithm computational complexity if condition expressions are not too complex. Most of the time [2] the upper and the lower bounds differ only by a constant factor; therefore, the estimation for the algorithm runtime growth is a theta-estimation $T(n) = \Theta(f(n))$. Analyze the following example 1 of an algorithm with a single loop:

```
void f1 (unsigned long n) {
    float x = n;
    while (x > 2)
        x = sqrt(x);
}
```

It is possible to build an algorithm symbol scroll table by including a single column for the loop index, a single column for the single variable and the column for the loop condition. Note that p in the i column means the index of the loop last execution.

Table 1. Symbol scroll table for the Example 1

i	x	loop condition	
	n		
1	$n^{1/2}$	$n > 2$	
2	$n^{(1/2)^2}$	$n^{1/2} > 2$	
3	$n^{(1/2)^3}$	$n^{(1/2)^2} > 2$	
...	...	...	
i	$n^{(1/2)^i}$	$n^{(1/2)^{i-1}} > 2$	
...	...	...	
p	$n^{(1/2)^p}$	$n^{(1/2)^{p-1}} > 2$	last
$p+1$		$n^{(1/2)^p} \not> 2$	exit

Analysis of the loop last execution provides inequality $n^{(1/2)^{p-1}} > 2$, which is transformed into $\log_2 n > 2^{p-1}$ and then into $p < \log_2 \log_2 n + 1$. The loop exit condition $n^{(1/2)^p} \leq 2$ is transformed into $p \geq \log_2 \log_2 n$ and, since p defines the algorithm computational complexity, the result is

$$T_{f1}(n) = \Theta(\log \log n).$$

If there are several loops which are not nested and not dependent (variables affected in one loop do not affect the other loops), then it is sufficient to estimate the algorithm complexity as the maximum of its loops complexities.

If there are loops that are nested but not dependent (amount of executions for the inner loop is not dependent on the amount of executions for the outer loop), then the product of both loops complexities equals to the algorithm complexity. E.g., if the outer loop A complexity is $T_A(n_A) = \Theta(f(n_A))$ and the inner loop B complexity is $T_B(n_B) = \Theta(g(n_B))$, then their combined complexity is

$$T_{AB}(n_A, n_B) = \Theta(f(n_A) \cdot g(n_B)).$$

If there are loops that are nested and dependent (inner loop parameter is determined in the outer loop) but the complexity of the inner loop is estimated with the same function $\Theta(g(n))$ for any iteration of the outer loop. The complexity of the outer loop is $\Theta(f(n))$. We will call such case *weak dependency*. Combined complexity of weakly dependent loops is the same as for the case of independent loops.

If the loops are not nested, then overall complexity equals to the maximum of the loop complexities:

$$T_{A+B} = T_A + T_B = \max\{T_A, T_B\}.$$

If there are non-nested dependent loops, then it is imperative to determine the value of the variable used in the second loop. Example 2:

```
void f2 (unsigned long n) {
    float x = n, z = n;
    while (x > 2) {
        x = sqrt(x);
        z = z * z;
    }
    while (z /= 2 > 1);
}
```

The algorithm symbol scroll table includes a N_l column with loop numbers 1 and 2, columns i_1, i_2 with corresponding loop indexes, columns x, z with values of these variables and the loop condition column. Here, p_1 is the index for the last execution of the loop 1 (outer) and p_2 means the index for the last execution of the loop 2 (inner).

Table 2. Symbol scroll table for Example 2

N_l	i_1	i_2	x	z	loop condition	
			n	n		
1	1		$n^{1/2}$	n^2	$n > 2$	
	2		$n^{(1/2)^2}$	n^{2^2}	$n^{1/2} > 2$	
	...		...	...	...	
	i		$n^{(1/2)^i}$	n^{2^i}	$n^{(1/2)^{i-1}} > 2$	
	...		...	...	...	
	p_1		$n^{(1/2)^{p_1}}$	$n^{2^{p_1}}$	$n^{(1/2)^{p_1-1}} > 2$	last
	$p_1 + 1$				$n^{(1/2)^{p_1}} \not> 2$	exit
2		1		$n^{2^{p_1}}/2$	$n^{2^{p_1}} > 1$	
		2		$n^{2^{p_1}}/2^2$	$n^{2^{p_1}}/2 > 1$	
		...		...	...	
		p_2		$n^{2^{p_1}}/2^{p_2}$	$n^{2^{p_1}}/2^{p_2-1} > 1$	last
		$p_2 + 1$			$n^{2^{p_1}}/2_2^p \not> 1$	exit

The analysis for the loop 1 is the same as in Example 1 and gives the following bounds estimation: $\log_2 \log_2 n \ \leq p_1 < \log_2 \log_2 n + 1$ which is transformed to $p_1 = \log_2 \log_2 n$. Therefore, loop 1 complexity is $T_1(n) = \Theta(\log \log n)$.

The analysis for the loop 2 last execution condition: $n^{2^{p_1}}/2^{p_2} > 1$, using p_1 value gives inequality $2^{p_2} < n^{2^{\log_2 \log_2 n + 1}} = n^{2 \log_2 n}$. Applying logarithm to the both sides gives $p_2 < 2log_2^2 n$.

The analysis for the loop 2 exit condition: $n^{2^{p_1}}/2^{p_2+1} \leq 1$ using value of p_1 is transformed to $2^{p_2+1} \geq n^{2^{\log_2 \log_2 n}} = n^{\log_2 n}$. Applying logarithm to the both sides gives $p_2 \geq log_2^2 n - 1$. Consider that $1 \leq \frac{1}{4} \log_2^2 n$ when $n \geq 4$. Perform the substitution in the previous inequality: $p_2 \geq \frac{3}{4} \log_2^2 n$. Coupled with the loop last execution inequality, it results in $\frac{3}{4} \log_2^2 n \leq p_2 \leq 2 \log_2^2 n$. Therefore, loop 2 complexity is $T_2(n) = \Theta(log_2^2 n)$.

The algorithm complexity is the maximum of loops complexities:

$$T_{f2}(n) = \Theta(log^2 n).$$

If there are nested dependent loops, then algorithm complexity is determined as the total amount of inner loop executions throughout the total amount of outer loop executions. Example 3:

```
void f3 (unsigned long n) {
    float x = n, y, z = n;
    while (x > 2) {
        x = sqrt(x);
        z = z * z;
        y = z;
        while (y /= 2 > 1);
    }
}
```

The algorithm symbol scroll table includes the column N_l with loop numbers 1 and 2, columns i_1, i_2 with corresponding loop indexes, columns x, y, z with values of the corresponding variables and the column with the loop condition. Here, p_1 means the index for the last execution of the loop 1 (outer) and p_2 means the index for the last execution of the loop 2 (inner).

The analysis for the loop 1 is the same as for examples 1 and 2 and gives following estimations: $\log_2 \log_2 n \ \leq p_1 < \log_2 \log_2 n + 1$, which is transformed to $p_1 = \log_2 \log_2 n$.

The total amount of the loop 2 executions is defined as $T_2(n) = \sum_1^{p_1} p_2(i)$. In order to solve it, we will define common term $p_2(i)$ as the amount of executions of the loop 2 at i-th execution of the loop 1. At the last execution of the loop 2 the following inequality is truthy: $n^{2^i}/2^{p_2(i)} > 1$, which is transformed into $2^{p_2(i)} < n^{2^i}$. Applying logarithm to the both sides gives $p_2(i) < 2^i \log_2 n$. At the loop exit the following statement is truthy: $n^{2^{p_1}}/2^{p_2(p_1)+1} \leq 1$, which is transformed into $2^{p_2(i)+1} \geq n^{2^i}$. Applying logarithm to the both sides and moving the 1 on the other side gives $p_2(i) \geq 2^i \log_2 n - 1$. Placing these inequalities into the sum gives the upper and the lower bounds for the total amount of the loop 2 executions: $\sum_1^{p_1}(2^i \log_2 n - 1) \leq \sum_1^{p_1} p_2(i) < \sum_1^{p_1} 2^i \log_2 n$. Transforming the sum in the upper bound: $\sum_1^{p_1} 2^i \log_2 n = \log_2 n \sum_1^{p_1} 2^i = 2(2^{p_1} - 1) \log_2 n = 2(2^{\log_2 \log_2 n} - 1) \log_2 n = 2(\log_2 n - 1) \log_2 n < 2 \log_2^2 n$.

Transforming the sum in the lower bound: $\sum_1^{p_1}(2^i \log_2 n - 1) = \sum_1^{p_1} 2^i \log_2 n - p_1 = 2(\log_2 n - 1) \log_2 n - p_1 = 2 \log_2^2 n - \log_2 n - \log_2 \log_2 n > 2 \log_2^2 n - \log_2^2 n - 1/2 \cdot \log_2^2 n = 1/2 \cdot \log_2^2 n$. Placing both bounds gives $1/2 \cdot \log_2^2 n < \sum_1^{p_1} p_2(i) < 2 \log_2^2 n$, which gives algorithm complexity:

$$T_{f3}(n) = \Theta(log^2 n).$$

Note that for all the previous examples loop exit condition is a simple inequality for the single algorithm input parameter. Generally, the loop exit condition may be a complex boolean function. However, any boolean function can be represented using the disjunctive normal form (DNF). It is possible to analyze all the equalities and inequalities for each elementary conjunction in order to estimate the lower bound of loop index that makes all the conditions in the conjunction truthy. The minimum of these estimations across all elementary conjunctions in the DNF is the lower bound estimation for the loop. It is possible to estimate the upper bound in the same way. Example 4:

Table 3. Symbol scroll table for Example 3

N_l	i_1	i_2	x	z	y	loop condition	
			n	n			
1	1		$n^{1/2}$	n^2	n^2	$n > 2$	
2		1			$n^2/2$	$n^2/2 > 1$	
		2			$n^2/2^2$	$n^2/2^2 > 1$	
		...			...	...	
		$p_2(1)$			$n^2/2^{p_2(1)}$	$n^2/2^{p_2(1)} > 1$	last
		$p_2(1)+1$			$n^2/2^{p_2(1)+1}$	$n^2/2^{p_2(1)+1} \not> 1$	exit
1	2		$n^{(1/2)^2}$	n^{2^2}	n^{2^2}	$n^{1/2} > 2$	
2		1			$n^{2^2}/2$	$n^{2^2}/2 > 0$	
		2			$n^{2^2}/2^2$	$n^{2^2}/2^2 > 0$	
		...			...	...	
		$p_2(2)$			$n^{2^2}/2^{p_2(2)}$	$n^{2^2}/2^{p_2(2)} > 1$	last
		$p_2(2)+1$			$n^{2^2}/2^{p_2(2)+1}$	$n^{2^2}/2^{p_2(2)+1} \not> 1$	exit
...	...	...	...	...	...	...	
1	i		$n^{(1/2)^i}$	n^{2^i}	n^{2^i}	$n^{(1/2)^{i-1}} > 2$	
2		1			$n^{2^i}/2$	$n^{2^i}/2 > 1$	
		2			$n^{2^i}/2^2$	$n^{2^i}/2^2 > 1$	
		...			...	...	
		$p_2(i)$			$n^{2^i}/2^{p_2(i)}$	$n^{2^i}/2^{p_2(i)} > 1$	last
		$p_2(i)+1$			$n^{2^i}/2^{p_2(i)+1}$	$n^{2^i}/2^{p_2(i)+1} \not> 1$	exit
...	...	...	...	...	...	...	
1	p_1		$n^{(1/2)^{p_1}}$	$n^{2^{p_1}}$	$n^{2^{p_1}}$	$n^{(1/2)^{p_1-1}} > 1$	last
2		1			$n^{2^{p_1}}/2$	$n^{2^{p_1}}/2 > 1$	
		2			$n^{2^{p_1}}/2^2$	$n^{2^{p_1}}/2^2 > 1$	
		...			...	...	
		$p_2(p_1)$			$n^{2^{p_1}}/2^{p_2(p_1)}$	$n^{2^{p_1}}/2^{p_2(p_1)} > 1$	last
		$p_2(p_1)+1$			$n^{2^{p_1}}/2^{p_2(p_1)+1}$	$n^{2^{p_1}}/2^{p_2(p_1)+1} \not> 1$	exit
1	p_1+1					$n^{(1/2)^{p_1}} \not> 2$	exit

The symbol scroll table for this case includes one loop condition for each elementary conjunction in the DNF.

Condition 1 analysis gives bounds $\log_2 n - 1 \le p < \log_2 n$ which may give $\Theta(\log n)$ estimation. The analysis for condition 2 gives different bounds: $n/128 - 17 \le p < n/128 - 16$ which may give $\Theta(n)$ estimation. Therefore, an algorithm may have different estimations in different intervals of the n parameter. In order to get the precise bounds of these intervals, it makes sense to write down equalities for both the upper and the lower bounds: $n/128 - 16 = \log_2 n, n/128 - 17 = \log_2 n - 1$; Equalities are the same, so we can focus only on one of them. Its roots are numbers $n_1 = 3558, n_2 \approx 0.00001$.

It is clear that the loop will not be executed if $n \in [0, 2048]$ hence n_2 is irrelevant. If

```
void f4(unsigned long n) {
    float x, y;
    x = y = n;
    while (x > 1 || y < 2048) {
        x = x / 2;
        y = y - 128;
    }
}
```

Table 4. Symbol scroll table for Example 4

i	x	y	condition 1	condition 2	
	n	n			
1	$n/2$	$n-128$	$n/2>1$	$n-128>2048$	
2	$n/2^2$	$n-128\cdot 2$	$n/2^2>1$	$n-128\cdot 2>2028$	
...	...	...	...	...	
i	$n/2^i$	$n-128\cdot i$	$n/2^i>1$	$n-128\cdot i>2048$	
...	...	...	...	...	
p	$n/2^p$	$n-128\cdot p$	$n/2^p>1$	$n-128\cdot p>2048$	last
$p+1$	$n/2^{p+1}$	$n-128\cdot(p+1)$	$n/2^{p+1}>1$	$n-128\cdot(p+1)>2048$	exit

$n \in [2049, 3558]$, then the estimation is linear, and if $n \in [3559, +\infty)$, then the estimation is logarithmic. Therefore, the algorithm complexity is $\Theta(\log n)$.

3. Loop conditions analysis and its simplification for certain cases

The following relation is truthy for all the previously mentioned loop conditions:

$$f(p,n)\text{<relation sign><expression that does not contain } n \text{ or } p\text{>},$$

where n is a natural input parameter, p is the loop execution number parameter. In simple cases of $f(p,n)$ it is relatively easy to obtain the $\Theta(n)$ estimation. However, in more advanced cases it is sufficient to obtain the estimation for a simpler function $g(p,n)$, which allows to conduct the further analysis of the following equality:

$$f(p,n) = \Theta(g(p,n))$$

that means

$$\exists C_1 > 0, C_2 > 0, n_0, \forall p > 0, n \geq n_0 : C_1 \cdot g(p,n) \leq f(p,n) \leq C_2 \cdot g(p,n).$$

If the algorithm is an integer one (parameters can be only integers because of some transformation), then it may be too hard to analyze the necessary expressions. Finding

sums for the sequences that are not arithmetic or geometric may also be too hard. The following theorems allow to get the Θ-notation estimation for multiple operations on integer parts of linear expressions for the n parameter, or for integrals of integer sums.

Let $\mu_1(a \cdot n + b) = a \cdot n + b$ is a linear form, n is a natural number and $a > 0$. Let $\mu_p(a \cdot n + b) = a \cdot (a \cdot \ldots (a \cdot n + b) + \cdots + b) + b$ is a μ_1 form applied p times, and let $\mu_p([a \cdot n + b]) = [a \cdot [a \cdot \ldots [a \cdot n + b] + \cdots + b] + b]$, where square brackets are the integer part of the expression. It is the μ_1 form applied p times to the integer part of the expression. If the amount of loop executions is expressed via such a function, then we need to be able to estimate it. Theorem 1 allows to remove the integer operation for the sake of getting Θ-notation estimation.

Theorem 1. *Let $a, b \in \mathbb{R}$ are the coefficients of the linear form $a \cdot n + b$, where $a > 0$, $a \neq 1, n \in \mathbb{N}$. Then the following equality is truthy:*

$$\mu_p([a \cdot n + b]) = \Theta(\mu_p(a \cdot n + b)) = \Theta(a^p n).$$

An example for using theorem 1 is the analysis of the following algorithm A:

```
void A (unsigned long N) {
    for (unsigned long k = N; k > 1; k = k/2);
}
```

The loop is executed p times and at the last execution the variable k is 1. So, the condition $[1/2 \cdot [1/2 \cdot \ldots [1/2 \cdot n] \ldots]] = 1$ is transformed to $1 \leq (1/2)^p \cdot N < 2$. Applying logarithm to the both sides gives $\log_2 N - 1 < p \leq \log_2 N$, hence, algorithm complexity is estimated as $\Theta(\log_2 N)$.

Theorem 2 allows to estimate a finite sum by estimating an integral using Θ-notation.

Theorem 2. *Let $f(x), x \geq 0$ is a non-negative monotonic growing function. Let $f(x) \leq C \cdot f(x-1), C \geq 1$. Then the following equality is truthy:*

$$\sum_{x=m}^{n} f(x) = \Theta\left(\int_{m-1}^{n} f(x)dx\right) (\forall m > 0).$$

An example for using theorem 2 is the analysis of the following algorithm B:

```
void B (unsigned long n) {
    unsigned long m = 0;
    for (unsigned long i = 1, j = 2; i < n; i++, j <<= 1)
        m += i * j;
    while (m--);
}
```

The first loop complexity is $\Theta(n)$, and the second loop complexity is $\Theta\left(\sum_{i=1}^{n} i \cdot 2^i\right) = \Theta\left(\int_{x=0}^{n} x \cdot 2^x dx\right) = \Theta(n \cdot 2^n)$. So, B complexity is $\Theta(n \cdot 2^n)$.

Theorem condition is substantial because without it the integral may be a non-elementary function. However, it is still possible to estimate algorithm complexity as Θ-notation for a function with an argument value that is equal to the sum upper bound.

Theorem 3. *Let* $f(x), x \geq 0$ *is a positive monotonic growing function. Let* $g(x) \equiv \frac{f(x)}{f(x-1)}, x \geq 1$ *is a non-negative monotonic growing function with no upper bound. Then the following equality is truthy:*

$$\sum_{x=m}^{n} f(x) = \Theta\left(f(n)\right), (\forall n > m > 0).$$

An example for using theorem 3 is the analysis of the following algorithm C:

```
void C (unsigned long n) {
    unsigned long k = 0, m = 1;
    for (unsigned long i = 1; i <= n; i++) {
        m *= i;
        k += m;
        while (k--);
    }
}
```

The first loop complexity is $\Theta(n)$, and the second loop complexity is $k = \sum_{i=1}^{n} i!$. Since $g(i) = \frac{i!}{(i-1)!} = i$ and $g(i) > 2$ when $i > n_0 = 2$, using theorem 3 $\Theta\left(\sum_{i=1}^{n} i!\right) = \Theta(n!)$. Using Stirling's approximation we can find C complexity as $\Theta(n^n)$.

4. Normal system of symbol transformations

The previously mentioned theorems allow to speed up the algorithm complexity analysis for the algorithms of certain classes. But since it is not possible in every case, it is necessary to transform the symbol expressions and the equalities and inequalities containing the symbol expressions. Hence the issue to control correctness of such transformations done by the students. Using complex mathematical packages like *Mathcad* is not correct since they do not help in teaching students to perform such transformations. We propose the following solution: 1) select the parts of the analysis process that require such transformations, 2) select a limited set of allowed transformations which can be used to express any transformation. Let this set be called *normal transformations*.

The following use cases require symbol transformations:

- algorithm symbol scroll with transforming expressions in the table;
- writing inequalities for loop parameter;
- symbol transformation of the equalities and inequalities for both the variables and loops executions amount;
- algorithm complexity estimation through loop complexity.

The first allowed transformation is the order change of two immediate additive terms or factors. The rest of allowed transformations do not change the order of the affected

terms or factors. It is easier to control correctness of such transformations, but they still allow to express any necessary transformation.

The list of the allowed symbol transformations:

- changing order of terms or factors;
- moving a term from one side of equality to the other with changing its sign;
- reducing a fraction by a single factor;
- factoring a single factor out;
- factorial expansion by a single factor;
- using fundamental equalities (a set of pre-defined functions);
- symbol parentheses removing;
- symbol grouping by factoring a single factor out;
- symbol factorization;
- symbol factorization for powers;

There is a similar system of the normal symbol transformations for inequalities, which requires a few additional transformations:

- moving lead additive term (maximum growth speed) to the first place on one of sides;
- strengthening the upper bound estimation by discarding negative additive terms;
- strengthening the lower bound estimation by discarding positive additive terms;
- estimating a non-lead positive additive term in the upper bound via lead term;
- estimating a non-lead negative additive term in the lower bound via lead term;

This system of normal symbol transformations requires additional sections in the adaptive teaching system to teach it to the students. So, we propose the following order of the sections in the teaching system:

1. Algorithm complexity characteristics
2. Determining algorithm computational complexity
3. System of normal symbol transformations for equalities
4. Algorithm symbol scroll table
5. System of normal symbol transformations for inequalities
6. Estimating complexity of an algorithm with a single loop

7. Estimating complexity of an algorithm with a nested independent loops
8. Estimating complexity of an algorithm with a non-nested dependent loops
9. Estimating complexity of an algorithm with a nested dependent loops
10. Estimating complexity of an algorithm with an integer transformations
11. Estimating complexity of an algorithm with a sequence sum
12. Final exam

It is possible to further divide the sections that may prove to be too hard, e.g., the sections about the system of the normal symbol transformations.

5. Conclusion

This method of teaching Computational Complexity of Algorithms course of study allows to

- develop the methodical programs that extract information and control events for each section;
- develop the software that allows to automate all the stages of the learning process.

We hope that we will manage to implement the described method and that it will show its effectiveness in teaching this course to the students and developing their logical-mathematical thinking.

References

[1] Ermilova A. V., Rublev V. S., "Problemy razvitiya matematicheskogo myshleniya uchashchikhsya na primere obuchayushchey sistemy po kursu "Algoritmy i analiz slozhnosti"", *Sovremennye informatsionnye tekhnologii i IT-obrazovanie*, Sbornik izbrannykh trudov IX Mezhdunarodnoy nauchno-prakticheskoy konferentsii, INTUIT.RU, Moskva, 2014, 297–304, (in Russian).

[2] Cormen T. H., *Introduction to algorithms*, 3rd ed., MIT press, 2009.

Рублев В. С., Юсуфов М. Т., "Автоматизированная Обучающая Система для обучения курсу анализа сложности алгоритмов", *Моделирование и анализ информационных систем*, **24**:4 (2017), 481–495.

DOI: 10.18255/1818-1015-2017-4-481-495

Аннотация. В данной работе исследуются вопросы построения автоматизированной обучающей системы “Анализ сложности алгоритмов”, которая позволит учащемуся освоить сложный математический аппарат и развить логико-математическое мышление в этом направлении. Вводится технология символьной прокрутки алгоритма, позволяющая получать верхние и нижние оценки вычислительной сложности. Приводятся утверждения, облегчающие анализ в случае целочисленного округления параметров алгоритма, а также при оценке сложности сумм. Вводится

нормальная система символьных преобразований, позволяющая, с одной стороны, делать учащемуся любые символьные преобразования, а с другой стороны – упростить автоматический контроль корректности таких преобразований. Статья публикуется в авторской редакции.

Ключевые слова: автоматизированное обучение, анализ сложности алгоритмов, таблица символьной прокрутки алгоритма, нормальная система символьных преобразований

Об авторах:
Рублев Вадим Сергеевич, orcid.org/0000-0002-0252-9958, канд. физ.-мат. наук, профессор,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003, Россия, e-mail: roublev@mail.ru

Юсуфов Мурад Теймурович, orcid.org/0000-0002-0362-2951, аспирант,
Ярославский государственный университет им. П.Г. Демидова,
ул. Советская, 14, г. Ярославль, 150003, Россия, e-mail: flood4life@gmail.com