Olympiads in Informatics, 2024, Vol. 18, 63–80

DOI: 10.15388/ioi.2024.05

Preparing of Youngest Students for Participation in Programming Contests

Krassimir MANEV
zh.k. Yavorov, bl. 12A, entr. A., Sofia, Bulgaria
e-mail: krmanev@gmail.com

Abstract. The goal of this paper is to present in brief the analysis of large set of tasks from Bulgarian national and regional programming contests for age group E (4th–5th grades), published in (Manev, 2023). As a result of the analysis, the identified in more than 350 tasks topics are arranged in order of smoothly increasing difficulty and could be used as training curriculum for preparation of programming contestants of early age.

Keywords: programming contests, training curriculum, beginners.

1. Introduction

Programming contests in Bulgaria are organized in five age groups (Manev *et al.*, 2007). In the youngest group E students of 4th–5th grades which just started learning programming take part. That is why choosing tasks for their first programming contests must be in accordance with the stage of learning a programming language (C/C++) they reached.

Recently we published a book aimed to help the process of training the youngest students for participation in programming contest, as well as to identify the types of tasks, which are appropriate for the distinct stages of the preparation (Manev, 2023). In the process of creating the book all tasks for group E from the national programming contest since 2007 was analyzed in the order they appeared in contests' calendar:

- Autumn tournament (AT, held usually in November, 51 tasks);
- Winter tournament (WT, that was held in December since 2009 to 2017, 27 tasks);
- First round of the National Olympiad in Informatics (NOI1, held usually in January, 45 tasks);
- Second round of the National Olympiad in informatics (NOI2, held usually in February, 45 tasks);
- Third round of the National Olympiad in informatics (NOI3, held usually in Marc, 87 tasks);

- Spring tournament (SpT, held usually in April, 48 tasks);
- Summer tournament (SuT, held in June from 2018 till now, 15 tasks).

Recently two regional contests are also organized, tasks of which were also analyzed:

- Sofia autumn tournament (SAT, held in October, 18 tasks);
- Sofia spring tournament (SST, held in beginning of April, 15 tasks).

A total of 351 tasks we analyzed, part of them solved as illustrations of the corresponding theoretical material, and for each of the others a guidance for solving was given – shorter or vaster.

In this paper we shortly present the chapters of the book. The chapters are ordered in the way we are teaching them when training young contestants. So, the content (and the order of topics) could be used by the coaches as one-year curriculum for training beginners.

2. Preliminary Knowledge and Skills

We suppose that before beginning of their preparation for programming contests the pupils passed the course Intro to programming in C/C++, which is why we are not including in the book text about the basic programming skills. The first chapter, called **Programming Contests,** is dedicated to such knowledge and skills about the programming competitions that introductory course in programming does not contain.

The name of the first section in the chapter is **The Contest Tasks**. Because the goal of introduction to programming is to familiarize learners with the syntax and the semantics of the programming language, the tasks that are solved in such course are, in some sense, "artificial" – they are strongly oriented to some specific syntactical or semantical rules. Traditionally the contest tasks are different (Verhoeff, 2008). They look like taken from the real life and they are real life problems indeed, in most cases. Beside the inevitable problem formulation, the competitive task must contain strict description of input and output formats, the constraints on input data, sample test data and correct output for the sample test data– sections that are not usual for tasks solved in introductory courses but are so important for the programming contests.

Second and third sections of the chapter – **The contest systems** and **Evaluation of a contest tasks** – are introducing future contestants in the process of grading their solutions. Nowadays using programming contest grading systems is inevitable. In Bulgarian programming contests we recently are using own grading system BOS, which is conform with the systems used in International Olympiads, and that all contestants, including the youngest must know (Petrov & Kelevedjiev, 2022). Good knowledge of grading system and the process of grading leads contestant to a systematic approach they have to apply during the work on a contest task, in order to escape rejections for: Unsuccessful compilation (CE), Memory limit exceeding (MLE), Canceling the execution because an exception (RTE), and especially the Time limit exceeding (TLE) and Generating of a wrong answer (WA).

Fourth, fifth and sixth sections of the chapter are dedicated to some important for competitive programming elements of the programming language and the integrated development environment which are not a subject of the introductory course. In section **Including Standard Functions Libraries** we only stress on the modern form of including libraries with preprocessor instruction `#include <bits/stdc++.h>`, which liberate contestants of necessity to memorize where a standard function is defined. Section **Accelerating Input and Output** is particularly important. Beginners prefer to use stream input/output (`cin` and `cout`) instead the functions for formatted input/output. That is why they must be able to desynchronize stream input/output from the formatted one. Section **Preprocessing** in C/C++ is introducing helpful instructions that the preprocessor offer.

In seventh section – **Memory usage** – we briefly describe the organization of the computer's memory. Main issue for the beginners there is avoiding declarations of arrays inside the functions body, to not overfull the stack of the process. Not easy for understanding concepts standard input and standard output are discussed in the eight sections of the chapter. This discussion is particularly important for avoiding the bad practice of typing input data from keyboard when testing the code. This bad practice is unacceptable for programming contests because it is time consuming and risky. Last section of the chapter is brief introduction to possibilities of the command interpreter. The focus is placed on using command procedures with redirection of the standard input/output to text files. Without using this approach testing of the code, especially with large test cases, is impossible.

3. Early Stage of the Preparation

We call the early stage of the preparation the period when usage of loops and arrays are avoided. In this stage tasks in Bulgarian programming contest for beginners are created on the base of knowledge and skills of the contestants gained in math classes.

3.1. *Quotient and Remainder*

Second chapter of the book is dedicated to topic **Quotient and Remainder** – the results of the integer division `n/m` of two natural numbers n and m, $m \neq 0$. The theory in this topic is elementary and well known from math classes. One typical task in this group, for solving of which only ability to compose correct arithmetic expressions, is the following:

KLETKI (***AT 2015, Group E***)*. *K pigeons landed in a line of N cages, one pigeon in a cage, K ≤ N. The number of the empty cages between two neighbor pigeons is called*

* Full statements of the tasks mentioned in the article could be found (in Bulgarian), trough the link Tasks (in Bulgarian **Задачи**) and then trough the links `E` for different national contests, on the training site of the Bulgarian Olympiads in Informatics `https://arena.olimpiici.com`

distance between them. When landing pigeons are trying to maximize the distances between each two neighbors. Having in mind that it is not always possible to make distances equal, write a program to find the number of minimal distances of the best landing.

For example, if $N = 8$ and $K = 4$ then the best landing will be with two 1-distance couples and one 2-distance couple, so the asked number is 2.

Solving tasks for finding quotient and/or remainder is an appropriate moment to introduce the standard functions `floor` and `ceil`, which are not studied neither in school nor in an introductory course.

3.2. *Ordering of Numbers and Characters*

Next topic that we teach is **Ordering of numbers and characters**. Without using arrays and loops, the tasks of this kind require ordering (we do not even mention the term sorting) of only 3–4 numbers or characters. Nevertheless, the moment is appropriate to attract attention of the students to the big topic **Sorting**. This is the moment to attract the attention of the students on the fact that values of the variables of type `char` are numerical and the order is the natural – *letters* are ordered as in the alphabet – all capitals before all small letters, and *digits* by their numerical values. So, there is no difference in the procedures of ordering numbers and characters.

The natural technique bubble sort, which is easy understandable by the beginners, is used. We start with demonstrating the operation exchanging of two values – first by using an intermediate variable and then we introduce the standard function `swap`. As a side effect of ordering, we get the minimum element of the set, which is the first element in the arrangement, and the maximum, which is the last element.

Another side effect is that we carefully introduce students in the topic ***Time complexity of the algorithms*** (in the worst case) which is of enormous importance for the future contestants and must be systematically taught. Time complexity in this case is eased to introduce as the number of executed by the program operations. It is important for students to understand that the complexity of these tasks is a constant function and to perceive the asymptotic notation $O(1)$. The following task is typical for this topic:

GUESSN (*NOI2, 2023*). *Three of the numbers a_1, $a_2 = a_1 + d$, $a_3 = a_2 + d$ and $a_4 = a_3 + d$ are given in random order. Write a program to find the fourth number.*

For example, if the numbers 4, 8 and 6 are given, after ordering them we could conclude that forth number is 2 or 10. If the given numbers are 10, 1 and 4, then after the ordering the gap between 1 and 4 is 3, the gap between 4 and 10 is 6 so the answer is 7.

Then in the chapter we discuss some more difficult ordering tasks. First, we consider tasks for ordering of objects with two parameters by two criteria – first by one of the criteria, and when we have two objects with equal parameters by this criteria values – then by the other. One such task is the following:

BOOKS (*NOI1, 2015*). *Four books of given height and thickness must be arranged so that of two books of different heights, the one with the greater height is on the left, and if they are of equal height, on the left to be the thicker one. Write a program to find such an arrangement.*

For example, if the heights/thickness of the four books are 2/23, 70/150, 70/100 and 22/37, then the asked order is 70/150, 70/100, 22/37 and 2/23.This task is appropriate also to demonstrate that there is not much difference between ordering in decreasing and increasing order.

A version of ordering by two criteria is when values of the second parameter, for example the ordering number in the input of the objects, are not involved in the ordering. In such case the values of the second parameter have just to be swapped always when the main parameters' values are swapped. For example:

CAKES (*NOI1, 2014*). *Three cakes with different diameters, labeled with 1, 2 and 3 by the order their diameters are given in the input, must be arranged one over the other in such way that cake with bigger diameter is not arranged over cake with smaller. Write a program to output the required order of the labels of the cakes starting with the biggest one.*

In this chapter we also introduce the standard functions `min` and `max`, which could be helpful in some tasks.

3.3. *Positional Number Systems*

The topic **Positional number systems** is fundamental for education in computer science. But it is also appropriate for making tasks for the early stage of the preparation of contestants. Main difficulty in the topic could be the bad knowledge of the operation exponentiation of the base of the system, which is crucial for understanding the corresponding algorithms. That is why some knowledge about the operation must be taught in the beginning of the topic. Because arrays and loops are not used during this stage the considered numbers are with no more than 3–5 digits. The numbers are usually in decimal system and very rarely in some other – binary, ternary, etc.

Two are the main subtasks that the tasks of this kind could contain. First subtask is to separate the digits of a given number and second is to restore a number from given digits. A task could contain one of the two subtasks or both. For the first task a contestant could consider the given number as value of type `int` or as a sequence of values of type `char`. In first case the knowledge of the topic Quotient and Remainder is necessary because the last digit of the decimal number is the remainder of the number's division by 10 and integer division by 10 will remove the separated digit. If the number itself is necessary for some steps of the algorithm – this is the best way to separate its digits.

If the number itself is not necessary for some steps of the algorithm it could be input in few variables of type `char` which will be the necessary separating of the digits. For

the purpose students must know only how to transform ASCII value of the characters to corresponding digits. This approach could be used in the case when the number is necessary for some steps of the algorithm, but in such case the number must be restored from its digits.

Because the numbers in these tasks are small their restoring could be done by multiplication of the digits by corresponding exponents of 10 and summing of obtained terms. But we prefer to introduce for the purpose the Horner's rule which is the only alternative for the same kind of tasks but for larger numbers. Something more, Horner's rule eliminate the necessity to precalculated exponents of 10 when the number of digits is arbitrary, decreases in such a way the number of multiplications and ameliorate the time complexity.

The following task is typical for this topic:

DIFFERENCE (*NOI1, 2015*). *Let A be the smallest and B the biggest number which are formed by the digits of given number N, 100 ≤ N ≤ 999. Write a program to find the difference B – A.*

3.4. *Metric Units*

In tasks of the topic **Metric units** some calculation is usually required that involve different metric units of the same kind – for weight, distance, time, money. The standard approach here is to transform all units to smallest one, mentioned in the statement of the task, to make calculation with smallest unit and then to restore the result in the format required by the task's statement.

As a sample see the following tasks:

SONG (*NOI1, 2009*). *A musical composition which is n minutes and m seconds long must be recorded on a disk. The free space on the disk is k MiB, and to record one second of sound 16 KiB is required. Write a program that outputs YES if the song can be recorded or NO if it cannot. In case the free space on the disk is not enough for recording the song, then the program must print how many KiB are not enough.*

A specific case in this topic are the tasks in which a calculation with dates are necessary. For example:

DATE (*NOI1, 2009*). *Write a program that, given a valid date consisting of day d, month m and year y, finds the date of the next day. A year is a leap year if it is divisible by 4 but not divisible by 100 or divisible by 400.*

Because tasks for manipulating dates appear sometimes in the contest task set we recommend to students to create a function `nextday` that solve the task. The three variables are passed to the function as global and the result is obtained in the same variables to escape the not easy problem of passing parameters by pointers. This is the moment we suggest to student for first time to start creating their own auxiliary functions.

3.5. *Ad Hoc Tasks*

In competitive programming we call a task *ad hoc* (from Latin – for specific or immediate needs) if it could not be classified in any topic and usually there is no well-known algorithm/approach for its solving. We consider the ability to solve ad hoc tasks very important because it develops the creativity of the contestants – future professional software developers. Something more, most of the tasks given in international programing contest are ad hoc per se.

For the chapter **Ad hoc tasks,** we made thorough analysis of the ad hoc tasks that appeared in contests for the earliest age in Bulgaria, solving of which do not require loops and arrays. The goal is to propose to the beginners idea how to proceed with this kind of tasks.

The first kind of ad hoc tasks that we identified could be called "*does what the statement requires"*. That means the procedure/algorithm that the contestant has to code is described in the statement of the task. It could seem that these kinds of tasks are easy, but it is not always true. Sometime such task requires a perfect programming skill and could take a lot of the contestant's time (so called time killer tasks). For example:

CONDITIONING (*NOI1, 2014*). *Air conditioner executes inside one hour commands of the format:*

<code> <actual temperature> <required temperature>

where the code is one of the following:

- `f` – *downgrades the temperature to the required, but if the actual temperature is lower than required, does nothing.*
- `h` – *increases the temperature to the required, but if the actual temperature is higher than required, does nothing.*
- `a` – *downgrades the temperature if the actual is higher then required or increases the actual to required otherwise.*
- `v` – *the air conditioner only ventilates the air and does not change the temperature.*

Write a program that for given actual temperature, required temperature and a code of a command output the actual temperature after an hour.

Most of the other ad hoc tasks of this stage could be called "*consider the different cases*". For example:

TOURIST (*WT, 2010*). *Group composed of K students are preparing for a hike in the mountain and must choose one, two or three of the available tents so that the weight of the chosen tents is no more than W kilograms and they are able to accommodate all students. The first tent weighs A_1 kilograms and accommodates B_1 students, the second weighs A_2 kilograms and accommodates B_2 person and the third weighs A_3 kilograms and accommodates B_3 person. Write a program that determines in how many ways can be selected the tents.*

Identifying the different cases that the program must consider is not easy sometime. Some initial knowledge for generation of combinatorial configurations – permutations,

combinations, and variations – over sets with 3–5 elements will be necessary, as well as some skill of splitting the possible input data to equivalent cases.

4. Tasks that Require Loops

For the second half of the season, we include in the preparation the tasks that require loops considering ability of students to organize loops one of the most important skills for the future programmers. With solving tasks that require loops we can start to discuss seriously the time complexity of the used algorithms.

4.1. *Properties of a Set or a Sequence of Elements*

The simplest tasks that require loops are the tasks for finding some properties of numerical sets or sequences of elements. For example, finding the minimal or maximal element, the sum of the elements, the average for a numerical set and so on. Such kind of tasks we have solved for small number of elements in the early stage of the preparation and now we just extend the skills of the students to solve them with using loops. In a similar way we also extend the skills from the early stage for finding optimal/extremal element of a set or sequence when the elements have two or more parameters.

In this category we could also classify the tasks which require to solve many times task that was solved earlier. For example, to include in a loop body code of the function `nextday` (see Subsection 3.4) to so solve the task: for given date find which will be the date after *n* days.

4.2. *Finding Subsequences*

Many tasks that need using of loops of national or regional contests are for finding subsequences with some properties. These tasks could be solved without using arrays when two conditions are valid: the required subsequences do not overlap and not the subsequence itself is required but some characteristic of it – for example, longest or shortest one. Sample of such task is the following:

LOVABLE (*WT 2009*). *A sequence of integers is called sympathetic if it contains a subsequence of length at least 2 composed of the one and the same integer. Write a program that checks whether a given sequence is sympathetic and if it is sympathetic to output the integer composing longest subsequence delete it of one and the same integer. If there is more than one such subsequence, then the biggest integer that composes such subsequence has to be output.*

For solving such kind of tasks we suggest to students to keep the status of the search in 5 variables:

- `new` – for the currently considered integer;

- `last` – for the integer considered lately;
- `len` – for the length of currently considered subsequence;
- `maxlen` – for the length of longest found to the moment subsequence;
- `maxnum` – for the integer of the longest found to the moment subsequence.

When `new == last` we just increase `len` by 1, and when is not then we update `maxlen` with `len` when `len > maxlen` or `len == maxlen` but `last > maxnum`.

4.3. *Nested Loops*

Organizing nested loops is the most difficult topic in this kind of tasks. Especially when the boundaries of changing of the control variable of the inner loop is depending on the value of the control variable of the outer loop. We carefully introduce the approach for building nested loops through tasks for drawing figures with characters. For example:

TRIANGLE. *Write a program to output an equilateral triangle of height n as shown on the Fig. 1.*

For solving the task first an outer loop with n steps is organized for drawing on i-th step a row of the triangle. In the body of this loop, we must organize two inner loops – one to output sequence of intervals (with control variable j) and one for output sequence of asterisks (with control variable k). Observing the example from statement of the task for $n = 6$ we suggest creation of the Table 1 to identify the boundaries for j and k.

Now it is easy to write the corresponding code.

```
     *
    ***
   *****
  *******
 *********
```

Fig. 1.

Table 1

	For j		For k	
	From	To	From	To
1	1	5	1	1
2	1	4	1	3
3	1	3	1	5
4	1	2	1	7
5	1	1	1	9
6	1	0	1	11
i	1	$n - i$	1	$2i - 1$

4.4. *More about the Complexity of the Algorithms*

We are using tasks that need loops for deeper consideration of the problem for evaluation the complexity of algorithms. In this case the function of complexity is no longer constant like in the tasks that do not need loops. It is relatively easy to explain that the single loop of n steps that has body of constant complexity $O(1)$ is $O(n)$. It is not difficult to understand also that the complexity of two nested loops, outer making m steps and inner – n steps is $O(mn)$.

Difficulties arise by the case when the boundaries of the control variable j (or k) of the inner loop depend on the value of the control variable i of the outer, like in the example above. In this case students must be able to find the complexity $T(i)$ of each inner loop, for each value of i and to sum obtained functions. Let $T_1(i)$ be the time complexity function of first inner loop for the example above and $T_2(i)$ – of the second. From Table 1 we have obviously $T_1(i) = n - i$ and $T_2(i) = 2i - 1$. So, for the complexity $T(n)$ of the algorithm we have:

$$\begin{aligned} T(n) &= T_1(1) + T_1(2) + \ldots + T_1(n-1) + T_1(n) + T_2(1) + T_2(2) + \ldots + T_2(n) = \\ &= (n - 1 + n - 2 + \ldots + 1 + 0) + (1 + 3 + \ldots + 2n - 1) = \\ &= n\,(n-1)\,/\,2 + n^2 = O(n^2). \end{aligned}$$

For explaining how the first sum is calculated we are using the "proof" of the young Gauss, and for the second – just observation of the partial sums: 1, 4, 9, 16, and so on.

4.4. *Ad Hoc Tasks*

Having the loop construction some more difficult and more interesting tasks could be formulated. Fort example:

JUMPS (*SST, 2022*). *A grasshopper is perched on one end of L cm long stick, $L \leq 10^{18}$, and makes successive jumps along the stick towards its other end, until with the last hop falls from her. First jump is m centimeters long, and each subsequent one is longer than the previous by n centimeters. Write a program that determines how many jumps the grasshopper made on the stick before it fells of it.*

Trivial simulation of the process will not pass the tests with very large L and small m and n. For accelerating the simulation let us precompute, using the summation formula that students know, the length P_0 of the first 100 jumps of the grasshopper:

$$P_0 = m + (m + n) + (m + 2n) + \ldots + (m + 99n) = 100m + 4450n.$$

The distance P_1 for next 100 jumps will be:

$$\begin{aligned} P_1 &= (m + 100n) + (m + 101n) + (m + 102n) + \ldots + (m + 199n) \\ &= 100m + 100.100n + 4450n = P_0 + 10000n, \end{aligned}$$

and so on – for each 100 jumps the past distance will be 10000*n* cm longer than the past distance by the previous 100 jumps and the length of the current jump will increase by 100*n*. Using this observation, the length of the simulation, which means the complexity of the algorithm also, will decrease about 100 times.

If we precompute the lengths of the first 1000, second 1000, and so on jumps instead 100, which is not so different, we will accelerate the simulation 1000 times.

5. Using Arrays

Arrays are extremely important instrument for future programmers. Including tasks that require arrays we start teaching young contestants to structure its data – correctly and efficiently. Because only the adequate combination of algorithms and data structures could produce efficient programs (Wirth, 1976). Using arrays, we teach the contestants to implement basic abstract data types such as queues, stacks, maps, frequency chart, etc. in static arrays before the start of using the dynamic implementations of STL. This gives to authors possibilities to create more interesting and more edifying tasks, for solving which more sophisticated algorithmic approach are necessary. In this topic we include also different tasks over strings because the strings are *de facto* arrays of characters.

5.1. *Sorting and Merging*

Ordering 3–5 values at the early stage of the preparation we demonstrated the importance of the sorting of data for efficient solving of some tasks. At this stage it is time to stress this importance, especially for large amounts of data, to demonstrate that classic $O(n^2)$ algorithms are rather not applicable for hundreds of thousands or million elements. We introduce standard sorting function of STL, which use fast $O(n.\log n)$ algorithm, carefully explaining log *n* function which the students do not know from mathematic classes yet.

We use the discussions about the different sorting algorithms and their time complexity to introduce the linear $O(n + m)$ algorithm *Counting sort* of *n* integers when elements are smaller than *m*. The very simple implementation of this algorithm makes it inevitable alternative of the standard STL method when data is appropriate.

This is the moment also to introduce algorithmic approach known as *merging of sorted areas*. It happens that beside the initial purpose of the approach – to be a part of the efficient $O(n.\log n)$ *Merge sort* – the approach is appliable to some different tasks. For example finding the union and intersection of two sets, represented in sorted areas, etc. Example of such task is the following:

GARDEN (*NOI3, 2014*). *In one row (bed) n decorative bushes have been sown and in a second – m decorative bushes. If two bushes of the same height are found in each of the two beds, one of them must be moved to a third bed. Write a program that finds the heights of the bushes in the new bed, arranged from the lowest bush to the tallest one. There will be always at least one bush in the new bed.*

Searched set is the intersection of two sets. First, we are sorting the two arrays and then apply a small modification of the classic merging – only when the values of the two considered elements are equal, we move a bush of this height to the third bed. Final loop of the classic merging of two sorted arrays is not necessary at all.

5.2. *Finding Subsequences*

In subsection 4.2 we excluded from considering the tasks for finding subsequences when the possible subsequences overlap or/and the subsequence itself must be found. With usage of an array this kind of tasks are solvable. If the subsequence must be output as a result we just append to the status of the search, defined in 4.2, two more variables to memorize the begin and the end of the best found to the moment subsequence.

If the subsequences could overlap, we introduce the algorithmic approach that we call *sliding of window* (called by some colleagues a two pointers approach). There are two kinds of such tasks – when the required subsequence is of fixed length and when is of variable length. Searching subsequence of fixed length L, we start with opening of the window over first L elements of the sequence and then we slide the window "closing" it by one element from the beginning and extending it by one element after the end. If subsequences do not overlap sliding a window of fixed length is possible without using an array also. Appropriate for sliding window of fixed length is the following task:

SUMDIGITS. *Write a program to find a subsequence of length k of a sequence of n integers having maximal sum of digits of its elements.*

Crucial for the speed of program in these tasks is the efficient finding the properties of the new window. For this purpose, we recommend to students to keep in appropriate structure the necessary data for the fast calculation properties of the window. For the task above, for example, recommendation is to keep in another array the sum of the digits of each integer so for sliding of the window one subtraction from and one addition to the current sum are enough.

If the searched subsequence is of variable length, then longest or shortest subsequence with the given properties must be found usually. In this case, for sliding the window, more than one element from the beginning could be eliminated (until the necessary property is no more valid) and then one or more elements to be appended to the end (until the property get valid again). Example:

ALL LETTERS. *Write a program that for given text (sequence) composed of small letters find the shortest subsequence of consecutive letters such that each letter is included in the subsequence at least once.*

For fast calculation property of the searched window in this task we recommend keeping a map where for each letter to save how many times it is included in the current window and a counter of the different letters in the window.

5.3. *Linear Abstract Data Types*

As mentioned above, for solving tasks of this group implementations of some linear abstract types are necessary sometime. Besides the usual queues and stacks especial attention is paid to implementation of different kind of *maps*, that lead to more efficient solutions. Here we clearly differentiate the dynamic implementations of STL from the static implementations that the students must be able to code themselves, stressing the specific features of the two different implementations.

The advantage of dynamic implementations of maps from STL is that they allow a large enough range of the keys, using memory only for couples that are included in the map. The disadvantage is that, because the maintaining of the map is in some tree structure, usual operations (inserting in the map and check for presence of a key) are of complexity $O(\log n)$. Static implementation in an array (or in vector) will need memory proportional of the range of keys and is not applicable when the range of keys is large enough. But in the static implementations the complexity of usual operation is constant. So, the ability of the contestant to choose which of the two kinds of implementations to choose, depending on the task, must be trained systematically.

5.4. *Two Dimensional Arrays*

In contests for this age group there are not so many tasks that require two dimensional arrays. But it is an important step in teaching contestants to structure their data. And more, including two dimensional arrays in the training give even more possibilities to create interesting tasks with increasing complexity. These tasks almost always require using the nested loops and develop skills of the contestants to organize such loops and to evaluate the time complexity of their algorithms.

Main object in these tasks usually is a rectangular table of cells with given number of rows and columns for which an element (or elements) with some properties is (are) searched. Example is the following task:

MINESWEEPER (*AT, 2016*). *The board of the Minesweeper game is divided into nine equal cells with quadratic form – three rows with three cells in each row. In some of cells, that are labeled with 9, there is a bomb. The others, labeled with 0, are empty. The player's goal is, for each empty cell, to find the number of bombs in its neighbors – these that share a vertex or side with it. Write a program to find the required numbers.*

Principle difficulty in such a task could be the fact that cells on corners of the board, lying on the sides of the board and the inner cells have different number of neighbors – 3, 5 or 8 respectively. A very helpful approach in such cases is to border the table – up, down, left, and right – with *neutral cells*. In our example these cells must contain zeros. Then all cells of the board will have equal number of neighbors.

For exercising the organization of nested loops we use the classic task:

F	i	n	d
r	i	n	
t	!	g	t
s		e	h

Fig. 2.

BY SPIRAL. *In each cell of a table with m rows and n columns one of the letters of a texts is written in spiral order. The spiral starts in upper left corner of the table and go first right, then down, left, and up, then right again and so on, without repeating a cell (see Fig. 2 for m = n = 4). Write a program to restore the text.*

Solving this task, the approach from Section 4.3 for identifying the borders of four inner loops of the spiral must be applied.

Many other tasks in this subtopic could by classified as ad hoc and need approaches that we discussed above.

6. Divisibility

The chapter **Divisibility** of (Manev, 2003) is dedicated to the divisibility of natural numbers. It is extending in a natural way the topic **Quotient and remainder** that we started with. First, we discuss the main notions of the subject – *prime number*, *divisor* and *multiple, greatest common divisor, smallest common multiple*, as well as the trivial algorithms that follow from the definitions. For example:

- to check if a given natural number *n* is prime by finding its remainders of division to the numbers, less than or equal to the square root of *n* (here we introduce in intuitive level the notion *square root*);
- to factorize a given natural number *n* to its prime divisors by checking for divisibility to each smaller number as many times as necessary;
- to find the number of different divisors of a natural number *n* from the degrees of its prime divisors.

Examples of such tasks are:

PRIME (*ST, 2008*). *Write a program that determines how many prime numbers are in given sequence of n positive integers, where n < 100 and numbers are less than 200000.*

and

PRIMES (*WT, 2010*). *Write a program that determines how many digits in total have the prime numbers in given interval [A; B], A < B.*

Then we introduce *Euclid's' algorithm* for finding greatest common divisor (GCD) and least common multiple (LCM) of two natural numbers, as well as the Eratosthenes'

sieve for finding the prime numbers less than given n. Showing here that complexity of Euclid's algorithm for finding GCD(m, n), when $m < n$, is $O(\log n)$ is a real challenge for the mathematical culture of the contestants.

Examples of such tasks are:

CHESTNUTS (*NOI3, 2013*). *On the main street of the host city of the National Olympics are planted n chestnuts arranged in a straight line. The distances between the different neighboring trees are different. Write a program that finds the minimum number of trees that must be planted between the given so that the distances between every two neighboring trees are equal.*

MIRROR (*NOI3, 2013*). *Write a program that determines the number of these prime integers p in given interval [A; B], A < B, for which is true that p is equal to the number q obtained by reading digits of p in reverse order.*

7. Other Tasks

In the last chapter of the book, we consider tasks that were not classified in the previous chapters. One such kind of talks are the tasks that require usage of prefix values for solving some range query tasks. For example, maximal sum in the range or minimum/maximum in the range. Such an approach is inevitable when many queries for finding some extremal range must be executed. Example for such task is the following:

MAXIMAL SUM. *Write a program that for given sequence of n integers, absolute value of each of which is less or equal to 1000000, execute q queries for finding the maximal sum in q given intervals, n ≤ 1000000, q ≤ 1000000.*

It is obvious that with building prefix sums of the given sequence for $O(n)$ time, we could find each of the required maximal sum with time complexity $O(1)$.

Another, very specific kind are the tasks that excite *scanning of a raster*. In this kind of task, we call a *raster* some (one- or two-dimensional) structure that contains one *pixel* of data for each moment of an interval of time or for each point of some space. The essence of these tasks is to build a raster to keep the given data and then carefully to scan the raster to find the solution.

Example of such task is the following:

AIRPORT (*NOI3, 2012*). *In one day, N planes land and take off at an airport. It has been known that a subsequent aircraft may re-use a sleeve 10 minutes after the previous aircraft has disengaged from the arm. Write a program that determines the minimum number of sleeves that are required for servicing all aircraft taking off and landing. Times in the schedule are set in hours and minutes.*

For solving the task, we must build a raster, having one pixel for each minute of the day and for each landing airplane to increase by 1 the value of pixels that correspond of its staying in the airport and to decrease them by 1 after the depart of the airplane. Then the required value is the maximal value of the raster recorded in some minute of the day.

In the end of the chapter, we also consider some ad hoc tasks that happened to by difficult according to the results of the contestants from the corresponding contest. Two examples of such tasks are:

DIG (*NOI3, 2015*). *Given a positive integer n with no more than 30 digits. Write a program that finds the smallest k-digits positive integer that contains at least once each of the decimal digits that are not digits of n.*

EXPFIELD (*ST, 2021*). *Students of one class had been asked to plant m.n plants labeled from 1 to m.n in the experimental field of their school, in m row lines and n column lines. The students have planted them "by rows" – the plants labeled from 1 to n in the first row, plants labeled labeled from n + 1 to 2n in the second and so on. Then the teacher of them explained that the correct order is "by columns" – plants labeled from 1 to m in the first column, plants from m + 1 to 2m in the second and so on. Write a program to find the number of plants that will not be replanted.*

We will leave to the interested reader to find the best solutions of these tasks.

8. Discussion

Bulgarian experience in preparation of young contestants is big enough. From the beginning of 90's separate contests for youngest contestants (5th–7th grade) was organized and some years later the contests for students of 4th–5th was separated too. Since then, the intensity of the contests calendar for the youngest is the same as for all other age groups (with shorter contest's time). We consider that efforts for early preparation are crucial reason small-populated Bulgaria (less than 7 million in the moment) to be still in the top of the countries by the results in IOI, for example – 4th–5th by the total number of medals, and 9th by their quality (see `https://stats.ioinformatics.org/countries/`, links Total and Medals).

Training of very young programmers, especially for participation in programming contests, is a specific activity. Pupils in this age have not clear vision for their future profession yet and just try to find their road in different directions. And the road of the professional programmer is not easy at all. That is why augmentation of difficulties in the training process must be smooth and careful. The main goal of our textbook was to arrange the material in such way that passing from one topic to the next to be as easy as possible.

Some important conclusions for the early preparation of young contestants could be extracted from the analysis of the tasks from Bulgarian national and regional programming contests. On the first place this is the fact that, in the beginning, contestants have not good command of the programming language, and are not still ready to understand and use sophisticated algorithm and data structures. That is why the contest tasks are based mainly on school mathematics. Our analysis of the mathematic skills necessary for successful start in the competitive programming was published in (Manev, 2024).

Our statistic is definitive – most of the school students that demonstrate an early will to participate in programming contests in Bulgaria are coming from the specialized mathematical schools that accept students of 5th grade. Most of the necessary mathematical knowledge and skills mentioned in our analysis the students in some mathematical schools get in math classes or in out of school classes educational forms. These contestants can master the material from the book for one-year being in 5th grade. Students that are not in these mathematical school must start preparation in 4th grade to master necessary material and to have some success in the contests.

We sincerely hope that our experience will be helpful for colleagues that are engaged with the preparation of contestants at an early age and we are ready to discuss more details through personal contact.

References

Kelevedjiev, E., Branzov, T., Petrov, P., Shalamanov, M. (2020). Bulgarian Platform for Competitions in Informatics. *Mthematics and Education in Mathematics,* pp. 123–130. [In Bulgarian: Келеведжиев, Е, Брънзов, Т., Петров, П., Шаламанов, М., Българска платформа за състезателна информатика. *Математика и математическо образование*, стр. 123–130.]
http://www.math.bas.bg/smb/2020_PK/tom_2020/pdf/123-130.pdf

Manev, K. (2023). *Introduction to Competitive Programming*. KLMN, Sofia [In Bulgarian, Кр. Манев, *Увод в състезателното програмиране*. КЛМН, София.] ISBN 978-954-8212-12-0

Manev, K., Karadjova, R. (2024). Mathematics for Beginning Programers. *Mthematics and Education in Mathematics,* pp. 25–35. [In Bulgarian: Манев, К., Караджова, Р., Математика за начинаещи програмисти. *Математика и математическо образование,* стр. 25–35.]
http://www.math.bas.bg/smb/2024_PK/tom_2024/pdf/025-035.pdf

Manev, K., Kelevedjiev, E., Kapralov, S. (2007). Programming Contests for School Students in Bulgaria. *Olympiads in Informatics*, 1, 112–123.

Verhoeff, T. (2008). Programming Task Packages: Peach Exchange Format. *Olympiads in Informatics*, 2, 192–207.

Wirtt, N. (1976). *Algorithms + Data structures = Programs*. Prentice-Hall Inc. Englewood Cliffs, New Jersey. ISBN 0-13-022418-9

K. Manev is a retired professor, PhD in Computer Science. He was teaching Discrete mathematics, Programming and Algorithms in many Bulgarian universities – mainly Sofia University, American University in Bulgaria, and New Bulgarian University. He has published over 75 scientific papers and more than 30 textbooks in the fields of Informatics and Information Technologies. He was member of Bulgarian National Committee for Olympiads in Informatics since 1982 and President of the Committee from 1998 to 2002; also leader or deputy leader of Bulgarian team for IOI (International Olympiads in Informatics) in 1989, 1998, 1999, 2000, 2005 and 2014; member of the organizing team of IOI'1989 and IOI'1990; chairman of IOI'2009. From 2001 to 2003 and from 2011 to 2013 he was elected member of International Committee of IOI, from 2005 to 2010 – member of IC, representing the Host country of IOI'2009, and from 2015 to 2017 – a President of IOI. In 2017 he was one of the creators of European Junior Olympiad in Informatics and its President from 2017 to 2020.