Olympiads in Informatics, 2010, Vol. 4, 52–66
© 2010 *Institute of Mathematics and Informatics, Vilnius*

Algorithms without Programming

Marcin KUBICA, Jakub RADOSZEWSKI
Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
e-mail: {kubica,jrad}@mimuw.edu.pl

Abstract. Programming contests are generally intended to popularize computer science, particularly algorithmic thinking and programming skills. However, such contests are addressed to a limited group of pupils – those that can write programs and know at least some programming techniques. But how can we attract those pupils that know nothing about programming or algorithms? We argue that one of the ways to do it is to present tasks that require some algorithmic thinking, but no programming. The paper contains several such example tasks together with their analysis.

Key words: algorithmic tasks, programming contests.

1. Introduction

One of the main goals of various programming contests is popularization of programming, or more generally, computer science, among youngsters. However, many such contests are addressed to pupils that are already familiar with programming (Verhoeff *et al.*, 2006; Diks *et al.*, 2007). Usually the focus is on correctness and efficiency. One could say that they promote knowledge of programming techniques and algorithmics. But how can we bring the attention of young pupils to programming in the first place? The contests that we mentioned may be too much for the first step. Before introducing a programming language, we could bring students' attention to very simple algorithms, without formulating them as programming tasks.

It is not surprising that the best candidates for such contests are pupils interested in math, since computer science is a younger sibling of mathematics. In many computer science problems one can find interesting mathematical sub-problems. Also, many mathematical problems can be solved with the aid of computers. There is a wide spectrum of problems with purely mathematical problems on one end, and purely algorithmic problems on the other end. Many examples of intermediate tasks can be found in such contests as: the Beaver competition (`www.bebras.org`; Dagienė, 2006; 2010), the Ugāle team competition (Opmanis, 2009), and Project Euler (`projecteuler.net`). The problem how to choose good tasks for a contest has brought attention of many researchers, e.g. (Dagienė and Futschek, 2008; Diks *et al.*, 2008; Verhoeff *et al.*, 2006; Forišek, 2006). In our opinion, offering tasks or puzzles requiring various levels of algorithmic thinking is a good way to popularize learning programming among young pupils. This opinion is based on two years of cooperation with a monthly journal *Delta* (`www.mimuw.edu.pl/delta`) addressed to secondary school pupils and dedicated to mathematics, computer science, physics and astronomy.

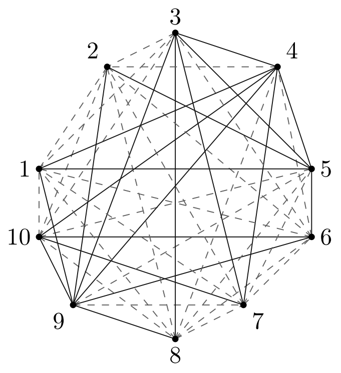
In the following sections we present a couple of simple problems formulated in a purely mathematical manner. Each problem is followed by a hint how to solve it, a solution and some methodological comments. All the problems require elements of algorithmic thinking. However one does not have to write programs, or even formally define an algorithm. These problems have already been presented (Radoszewski, 2009) in an article addressed to secondary-school pupils. The key properties of these tasks that make them suitable for initial education of algorithmic skills are summarized in Conclusions.

The authors would like to thank anonymous referees for many helpful comments.

2. Uni-Color Triangles

2.1. *Problem*

You are given 10 points in the plane, such that no 3 of them are collinear. Each pair of points is connected by a green (dashed) or blue (continuous) line – see the following figure.



How many uni-color triangles, with vertices at the given points, are present in the figure?

Hint: You will find the following table useful: the cell located in the ith row and jth column denotes the color (b – blue, g – green) of the line connecting points i and j.

	1	2	3	4	5	6	7	8	9	10
1		g	g	b	b	g	g	g	b	g
2	g		g	g	b	g	g	g	b	g
3	g	g		b	b	g	b	b	b	g
4	b	g	b		b	g	b	g	b	b
5	b	b	b	b		b	g	g	g	g
6	g	g	g	g	b		g	g	b	b
7	g	g	b	b	g	g		g	g	b
8	g	g	b	g	g	g	g		b	g
9	b	b	b	b	g	b	g	b		b
10	g	g	g	b	g	b	b	g	b	

2.2. *Solution*

The total number of triangles with vertices it the given 10 points is:

$$\binom{10}{3} = \frac{10 \cdot 9 \cdot 8}{1 \cdot 2 \cdot 3} = 120.$$

Instead of counting uni-color triangles, we can count multi-color triangles (i.e., with exactly two sides of equal color), and subtract the number of such triangles from 120.

Each multi-color triangle has exactly **two** vertices in which two sides of different colors meet. For each of the 10 points, let us count the number of green and blue edges incident to it – the results are shown in the following table:

Point	1	2	3	4	5	6	7	8	9	10
Blue edges	3	2	5	6	5	3	3	2	7	4
Green edges	6	7	4	3	4	6	6	7	2	5

Now, we can multiply the numbers of green and blue edges incident to the given points, and sum the products:

$$\begin{aligned} &3 \cdot 6 + 2 \cdot 7 + 5 \cdot 4 + 6 \cdot 3 + 5 \cdot 4 + 3 \cdot 6 + 3 \cdot 6 + 2 \cdot 7 + 7 \cdot 2 + 4 \cdot 5 \;= \\ &\qquad = 18 + 14 + 20 + 18 + 20 + 18 + 18 + 14 + 14 + 20 \;=\; 174. \end{aligned}$$

This way each multi-color triangle is counted twice. Hence, the number of uni-color triangles in the figure equals:

$$120 - \frac{174}{2} = 33.$$

2.3. *Methodological Comments*

Let us denote the number of vertices by n, here we have $n = 10$. A naive solution requires checking $\Theta(n^3) = \binom{n}{3}$ triangles, that is 120 triangles for $n = 10$. This is quite a large value – it suggests that it could be worthwhile to look for a more efficient solution. The combinatorial observation above reduces the problem to counting $\Theta(n^2) = \binom{n}{2}$

edges, plus processing n vertices. It is the optimal solution, since the size of the input is $\Theta(n^2)$. Although there is nothing about time complexity in the problem statement, the difference between computations that can be easily performed by hand and computations too complex to do manually corresponds to the difference between the optimal and inefficient solutions. This problem was presented at the 4th Polish Olympiad in Informatics, 1996/1997 (Guzicki, 1997).

3. Continuous Subsequences Divisible by 13

3.1. *Problem*

Does the following sequence of numbers:

$$(1,\ 1,\ 9,\ 7,\ 12,\ 4,\ 12,\ 5,\ 8,\ 2,\ 7,\ 2,\ 10,\ 2,\ 3)$$

contain a non-empty, continuous subsequence, whose sum is divisible by 13? If so, what is the number of such subsequences?

Hint: First, count the length of the given sequence. Then, count the sums of elements in consecutive prefixes (leading fragments) of the given sequence.

3.2. *Solution*

Observe that the sum of elements in any continuous subsequence equals the sum of elements in the prefix ending at the end of the subsequence, minus the sum of elements in the prefix ending one position before the beginning of the subsequence, see Fig. 1. Therefore, the first step should be to compute the sums of elements for all the prefixes of the given sequence (including zero for the empty prefix).

If for some continuous subsequence, the sum S of elements in it is divisible by 13, then the sums S_1 and S_2 of elements in the corresponding prefixes give the same remainders when divided by 13. Conversely, if there are two different prefixes, for which the sums of their elements give the same remainders modulo 13, then the sum of elements in the corresponding continuous subsequence is divisible by 13. The given sequence contains 15 elements, consequently, by the Dirichlet's principle (also known as the *pigeonhole principle*), there must exist two such prefixes, that the sums of their elements give the same remainder modulo 13. Hence, the answer to the first question is yes.

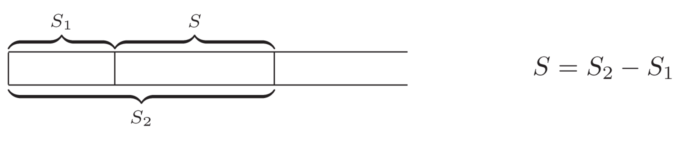


Fig. 1. Sum of elements in the continuous subsequence expressed using sums of elements in the corresponding prefixes.

Now, let us count the number of such continuous subsequences. Let us group the prefixes of the given sequence by the remainders (modulo 13) of the sums of their elements. Assume that for some remainder we have k prefixes in one group. How many continuous subsequences, with the sums of elements divisible by 13, do they generate? Exactly the number of (unordered) pairs of different such prefixes, that is:

$$\binom{k}{2} = \frac{k(k-1)}{2}.$$

The sums of elements of the prefixes are as follows:

Given sequence		1	1	9	7	12	4	12
Prefix length	0	1	2	3	4	5	6	7
Sum of elements	0	1	2	11	18	30	34	46
Remainder modulo 13	0	1	2	11	5	4	8	7

Given sequence	5	8	2	7	2	10	2	3
Prefix length	8	9	10	11	12	13	14	15
Sum of elements	51	59	61	68	70	80	82	85
Remainder modulo 13	12	7	9	3	5	2	4	7

If we group equal remainders, we obtain the following results:

Remainder	0	1	2	3	4	5	6	7	8	9	10	11	12
Number of prefixes	1	1	2	1	2	2	0	3	1	1	0	1	1
Generated cont. subseq.	0	0	1	0	1	1	0	3	0	0	0	0	0

Hence, the total number of continuous subsequences, for which the sum of their elements is divisible by 13, equals 6.

3.3. *Methodological Comments*

This problem illustrates some elements of dynamic programming. First, the original problem is reduced to a (parameterized) problem of counting the number of prefixes for which the sum of their elements, modulo 13, equals a given remainder. This problem, in turn, is reduced to calculating the sums of elements for all the prefixes of the given sequence.

It also informally illustrates the issue of time complexity. Let us denote by n the length of the given sequence, here $n = 15$. A naive solution requires checking $\Theta(n^2)$ continuous subsequences, and processing a subsequence takes $\Theta(n)$ time in average – this gives $\Theta(n^3)$ total time complexity. The first observation reduces the time needed to calculate the sum of elements in a continuous subsequence to $O(1)$, and hence the whole solution requires $\Theta(n^2)$ time. Finally, the presented solution requires $\Theta(n)$ time and is easy to perform by hand.

4. Subsequences Divisible by 5

4.1. *Problem*

What is the number of subsequences of the following sequence:

$$(3,\ 2,\ 3,\ 4,\ 1,\ 1,\ 2,\ 3)$$

such that the sum of their elements is divisible by 5? Subsequences that are equal, but formed from elements at different positions, should be counted separately, e.g. the sequence $2, 1, 2$ should be counted twice, since there are 2 different ways to obtain it.

Hint: For each prefix of the given sequence, count the number of its subsequences, for which the sum of elements modulo 5 equals, respectively, 0, 1, 2, 3 and 4.

4.2. *Solution*

The hint tells us what should be calculated, but it does not say how. First, let us consider a couple of the shortest prefixes of the given sequence. We start with the empty prefix. Of course, it has only one subsequence, the empty one, and the sum of its elements equals zero. Then there is a single element prefix (3). We can either take this element or not. Hence, there is one subsequence with the sum of elements equal 0 (the empty one), and one with the sum of elements equal 3. Now, let us consider the prefix $(3, 2)$. There are four subsequences of such a prefix: the empty one, (3), (2), and $(3, 2)$. The first two have already been considered, as subsequences of (3). The other two contain the last element of the prefix, as their last element.

The above observation can be generalized. All the subsequences of a given nonempty prefix can be divided into two groups: these containing the last element of the prefix, and these not containing it. Let us denote the last element of the prefix by x. Then, the number of subsequences of the prefix, for which the sum of elements modulo 5 equals k, is equal to the sum of:

- the number of subsequences of a prefix shorter by one position, for which the sum of elements modulo 5 equals k, and
- the number of subsequences of a prefix shorter by one position, for which the sum of elements modulo 5 equals $(k - x) \bmod 5$.

Using this rule, one can calculate the number of respective subsequences for all the prefixes. The following table shows the results for the given sequence:

Given sequence		3	2	3	4	1	1	2	3
Prefix length	0	1	2	3	4	5	6	7	8
$\equiv 0 \pmod 5$	1	1	2	3	4	7	13	26	51
$\equiv 1 \pmod 5$	0	0	0	1	2	6	13	26	52
$\equiv 2 \pmod 5$	0	0	1	1	4	6	12	25	50
$\equiv 3 \pmod 5$	0	1	1	3	3	7	13	26	52
$\equiv 4 \pmod 5$	0	0	0	0	3	6	13	25	51

Finally, we have 51 subsequences (including the empty one) of the given sequence, for which the sum of elements is divisible by 5. But what are they?

Enumerating all the 51 subsequences would be too tedious. Nevertheless, let us deal with this problem for a shorter prefix: $(3, 2, 3, 4)$. From the table we know that there are four such sequences. To find all of them, we should trace back, how the number 4 has been obtained. It was obtained as the sum $4 = 3 + 1$, where 3 and 1 are the numbers of subsequences of $(3, 2, 3)$ giving remainders, respectively, 0 and 1. Recursively, we can trace back how the numbers 3 and 1 have been obtained. The latter one corresponds to the sequence $(3, 3, 4)$. The other three are subsequences of $(3, 2, 3)$. Two of them are subsequences of $(3, 2)$ – clearly, they are () and $(3, 2)$. The last one is a subsequence of $(3, 2)$, giving the remainder 2, extended by 3 – that is $(2, 3)$. Hence, all the four subsequences are: (), $(3, 2)$, $(2, 3)$ and $(3, 3, 4)$.

4.3. *Methodological Comments*

This problem is a clear illustration of dynamic programming. First, the hint shows how the original problem can be reduced to a number of simpler sub-problems. Then, knowing the recursive dependencies between these sub-problems, one can calculate the table with their results. Finally, it also illustrates a more general property of dynamic programming: knowing the number of solutions is half way to knowing the actual solution.

Without the dynamic programming, it is not feasible to do this task "by hand". The number of all possible subsequences that have to be considered is exponential. However, dynamic programming reduces the time complexity to $\Theta(n)$, where n is the length of the given sequence.

5. Coins

5.1. *Problem*

You are given 11 coins of the following values:

$$7,\ 300,\ 35,\ 83,\ 1,\ 17,\ 2,\ 1,\ 17,\ 170,\ 5.$$

What is the smallest (positive integer) amount of money, that cannot be paid using the coins?

Hint: Let us order the coins according to their value:

$$1,\ 1,\ 2,\ 5,\ 7,\ 17,\ 17,\ 35,\ 83,\ 170,\ 300.$$

Now, consider the sets of amounts of money that can be paid using consecutive prefixes of the ordered sequence of coins.

5.2. *Solution*

Let us follow the hint. Using just the first two coins, it is possible to pay any amount from 0 to 2. If we include the third coin, 2, the range of amounts that can be paid extends to from 0 to 4. What if we include coin 5? Can we pay any amount of money from 0 to $4 + 5 = 9$? Yes, for any amount from 0 to 4, we do not have to use coin 5, and for any amount x from 5 to 9, we use coin 5 and the amount $x - 5$ can be paid using the remaining coins.

In general, if the range of amounts that can be paid using the first k coins is from 0 to r, and the following coin has value x, then either:

- $x \leqslant r + 1$ and it is possible to pay any amount from 0 to $r + x$ using the first $k + 1$ coins, or
- $x > r + 1$ and it is not possible to pay the amount of $r + 1$, since x and all the following coins have greater values.

Using this observation, we obtain the following sequence of included coins and ranges of amounts that can be paid:

Coins	1	1	2	5	7	17	17	35	83
Maximum amount	1	2	4	9	16	33	50	85	168

And finally, $170 > 168 + 1 = 169$, so the smallest amount that cannot be paid using the given coins is 169.

Of course, if we do not find such a coin x, that $x > r + 1$, then the smallest amount that cannot be paid is simply the sum of all coins' values plus one.

5.3. *Methodological Comments*

This problem resembles the classical problem: how to pay a given amount of money using the coins from the given set. Of course, as a side effect we obtain information about the amounts that cannot be paid. This would be an application of dynamic programming (again). The running time of such a solution is $\Theta(n \cdot S)$, where n is the number of coins and S is the sum of their values. It is simple to perform, but tedious.

On the other hand, the solution presented here is in a way greedy. Moreover, it is much faster – it runs in $\Theta(n)$ time.

6. Encyclopedia

6.1. *Problem*

Your bookshelf contains 12 volumes of encyclopedia. Their order has become quite random:

$$11, 1, 10, 4, 3, 2, 8, 7, 12, 6, 9, 5.$$

In one move you can take out one volume and insert it anywhere in the row of volumes (shifting some volumes if needed). What is the minimal number of moves necessary to order the volumes from 1 to 12, left to right?

Hint: What is the largest set of volumes that may be left and *not moved at all*?

6.2. *Solution*

Any set of volumes that may be left and not moved at all must obviously form an increasing subsequence of the given sequence of volumes. On the other hand, if we choose any increasing subsequence of the given sequence, and decide not to move these volumes, we can insert all the remaining volumes at their correct positions, so that all the volumes are ordered. Hence, the requested minimal number of moves equals the length of the given sequence, minus the length of its longest increasing subsequence.

It remains to show how to find the length of the longest increasing subsequence of the given sequence of volumes:

$$(11,\ 1,\ 10,\ 4,\ 3,\ 2,\ 8,\ 7,\ 12,\ 6,\ 9,\ 5).$$

The main idea of the solution is to calculate, for each element of the sequence, the length of the longest increasing subsequence ending at that element. Then it suffices to take the maximum of such lengths.

For the first two volumes the requested result is 1. The third volume has number 10, so it can be added at the end of a single-element increasing subsequence that ends at volume number 1, resulting in a subsequence of length 2. The same holds for the following 3 volumes: 4, 3 and 2. After that there is the volume 8, which can be added at the end of any of the last three 2-element subsequences formed, what results in a subsequence of length 3. Generally, for each element x, either:

- x is a single-element increasing sequence, or
- x can be appended at the end of the longest increasing subsequence ending at some y, provided that y precedes x in the given sequence and $y < x$.

Basing on this observation, we can calculate the following table of intermediate results:

Volume	11	1	10	4	3	2	8	7	12	6	9	5
Length of subsequence	1	1	2	2	2	2	3	3	4	3	4	3

Therefore, the longest increasing subsequence of the given sequence has 4 elements. Examples of such sequences are: $(1, 3, 7, 12)$ and $(1, 2, 6, 9)$. Hence, the smallest number of moved volumes equals $12 - 4 = 8$.

6.3. *Methodological Comments*

After reducing the main problem to the problem of finding the longest increasing subsequence, we deal with dynamic programming again – instead of one problem of the longest increasing subsequence, we have to consider a whole table of such problems. However, the dependencies between increasing subsequences ending in different elements allow us to calculate their lengths more easily.

Trivial implementation of the dynamic programming algorithm runs in $\Theta(n^2)$ time, where n is the number of volumes. Without dynamic programming, one would have to consider an exponential number of subsequences of the given sequence. Note that using any of the classical $\Theta(n \log n)$ time algorithms for the longest increasing subsequence problem would be impractical here.

7. Heavy Encyclopedia

7.1. *Problem*

Let us modify the previous problem. The encyclopedia volumes turned out to be so heavy, that you can hardly move them from one position to another, or shift the remaining volumes to make space. You have changed your mind and decided to order the volumes only by swapping adjacent volumes on the shelf. What is the minimal number of such moves needed to order the volumes from 1 to 12, again left to right?

Hint: First perform such a sequence of moves, after which the first volume is located at the leftmost position on the shelf. Reduce the problem eliminating the first volume from further considerations, and repeat this procedure for the volumes $2, \ldots, 12$.

7.2. *Solution*

First, let us check, whether the hint is correct, i.e. generates the minimal number of moves. Let us consider some optimal solution and begin by focusing on volume 1. Let i be its initial position. The hint suggests that we should first move it to the left $i - 1$ times. If in the considered solution volume 1 is moved only to the left, then clearly there are $i - 1$ such moves. Moreover, without changing the total number of moves, we can make them first. This way we obtain an optimal solution that starts as suggested in the hint.

What if volume 1 is not only moved to the left? Let us assume that there is a volume x, which is swapped with volume 1, jumping over it to the left, that is: $\ldots, 1, x, \ldots \rightarrow \ldots, x, 1, \ldots$ In such a case, they must be swapped again at some point. Observe that we can save two moves, by "pushing" volume 1 in front of x and not swapping them at all. Hence, such a situation cannot occur in the optimal solution.

Now, assume that we first move volume 1 to the very left. Clearly, if it is swapped with any other volume later on, such a solution cannot be optimal. Hence, we can reduce the problem to ordering a smaller number of volumes. So, the hint is really sound.

Let us simulate the ordering described in the hint. The following table shows the sequence of volumes, after putting consecutive volumes in place:

Step	Moves	Volumes
1	1	11, 1, 10, 4, 3, 2, 8, 7, 12, 6, 9, 5
2	4	1, 11, 10, 4, 3, 2, 8, 7, 12, 6, 9, 5
3	3	1, 2, 11, 10, 4, 3, 8, 7, 12, 6, 9, 5
4	2	1, 2, 3, 11, 10, 4, 8, 7, 12, 6, 9, 5
5	7	1, 2, 3, 4, 11, 10, 8, 7, 12, 6, 9, 5
6	5	1, 2, 3, 4, 5, 11, 10, 8, 7, 12, 6, 9
7	3	1, 2, 3, 4, 5, 6, 11, 10, 8, 7, 12, 9
8	2	1, 2, 3, 4, 5, 6, 7, 11, 10, 8, 12, 9
9	3	1, 2, 3, 4, 5, 6, 7, 8, 11, 10, 12, 9
10	1	1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10, 12
11	0	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
12	0	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
13		1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

The total number of moves is 31.

This result can also be obtained without simulating intermediate sequences of volumes. Note that the number of moves needed to position volume x equals the number of such volumes y, that $y > x$ and y is to the left of x in the original problem. Simply, these are the volumes that have to be swapped with volume x at some point, and after this, volume x is in its final position. Thus, the numbers of moves can be calculated just by analysing the original sequence of volumes.

Volumes	11	1	10	4	3	2	8	7	12	6	9	5
Moves	0	1	1	2	3	4	2	3	0	5	3	7

7.3. *Methodological Comments*

This problem illustrates two programming techniques: greedy programming and “divide and conquer” rule. The greedy algorithm sorts the volumes by placing consecutive volumes in place. The “divide and conquer” rule, after putting the first volume in place, allows us to reduce the problem to a smaller one.

When designing a greedy algorithm, usually, we have to prove two properties:

- the greedy property – that is, that there is an optimal solution starting as suggested in the hint,
- sub-problem property – that is, that the greedy step and an optimal solution of the smaller problem give the optimal solution of the original problem.

The greedy property is proved in the first two paragraphs of the previous section. The sub-problem property is rather trivial here, since we optimize the total number of moves.

It is worth noting that the number of moves is, in fact, the number of inversions in the given sequence. Clearly, for any two volumes x and y, such that $x < y$ and volume x is to the right of volume y, these volumes have to swapped at some moment in time. Hence, the number of moves is at least the number of inversions. Moreover, the final sequence, in which all the volumes are in place, contains no inversions, and every move of the described solution reduces the number of inversions by one. Hence, the number of moves equals the number of inversions. As a consequence, every algorithm that reduces the number of inversions within each move, produces an optimal solution.

The time complexity of each of the presented solutions (the one based on simulation and the one analyzing inversions) is proportional to the number of required moves, that is $\Theta(n^2)$. On the other hand, calculating the total number of inversions can be done in $\Theta(n \log n)$ time, using appropriate data structures or algorithms – however, such a complicated solution would be impractical here.

8. Anti-Binary Sets

8.1. *Problem*

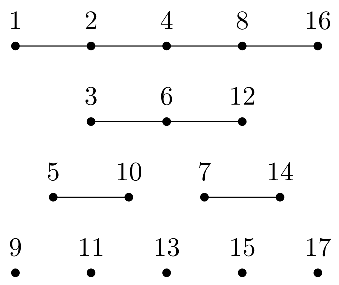
Let $Z = \{1, 2, \ldots, 17\}$. Let us consider such subsets $A \subseteq Z$, that for every $x \in A$ we have $2x \notin A$. Such subsets of Z are called *anti-binary*.

What is the largest anti-binary subset of Z? How many anti-binary subsets of Z are there?

Hint: Make a graph with vertices labeled from 1 to 17, and edges connecting vertices m and $2m$ (for $m = 1, 2, \ldots, 8$).

8.2. *Solution*

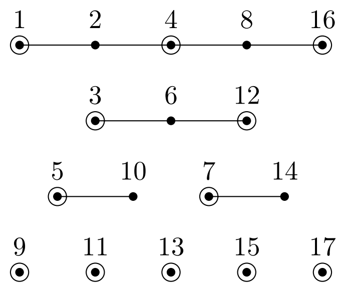
Let us construct the graph described in the hint:



It consists of a number of "paths": one containing 5 vertices, one containing 3 vertices, 2 containing 2 vertices each, and 5 single vertices. The definition of the anti-binary set means that such a set cannot contain elements connected by an edge. We can consider each path separately, since they are not connected with each other. First, we can select all 5 single vertices. From each path of size 2 we can select just one vertex. From the path of size 3 we can select 2 vertices – its ends. Finally, we can select 3 vertices from the path of size 5 – its ends plus the middle vertex. So, the maximum number of vertices that can form an anti-binary set is 12. An example maximum anti-binary set:

$$A = \{1, 3, 4, 5, 7, 9, 11, 12, 13, 15, 16, 17\}$$

is shown in the following figure:



Now, let us consider the number of anti-binary sets. This number also can be deduced from the structure of the constructed graph. Denote by A_n the number of anti-binary subsets of a path of size n. Then, the number of all anti-binary subsets of Z equals:

$$A_1^5 \cdot A_2^2 \cdot A_3 \cdot A_5.$$

One can easily check that $A_1 = 2$ and $A_2 = 3$. For longer paths it gets more and more complicated. Let us split all the anti-binary sets of a path of size n into two groups: containing the last vertex, and not containing it. The number of latter ones is simply A_{n-1}. If an anti-binary set contains the last vertex, then it does not contain its predecessor. Hence, the number of such sets equals A_{n-2}. From this, we obtain:

$$A_n = A_{n-1} + A_{n-2}.$$

Does it look familiar? Sure! It is a definition of the Fibonacci numbers! More precisely, $A_n = \mathit{Fib}_{n+2}$. In particular, $A_3 = 5$, $A_4 = 8$ and $A_5 = 13$. Hence, the total number of anti-binary subsets of Z equals:

$$2^5 \cdot 3^2 \cdot 5 \cdot 13 = 18\,720.$$

8.3. *Methodological Comments*

The first important step of the solution is to view the problem as a graph problem – we are looking for a maximum size independent set in the constructed graph. In general, this problem is NP-hard. However, since we consider graphs consisting of separate paths, it is much easier. Then the problem can be reduced even further, to graphs consisting of single paths. However, still the number of anti-binary sets is exponential, and the observation to use Fibonacci numbers is needed to compute it in linear time.

The solution contains elements of greedy programming, when constructing the maximum size anti-binary subset. On the other hand, in counting anti-binary subsets and computing Fibonacci numbers, there are elements of dynamic programming.

9. Conclusions

In this paper we have discussed several problems of algorithmic nature that we claim can be used to popularize algorithmic thinking among pupils not familiar with programming. Let us note that each of these problems is a small instance of a regular programming task, accompanied with a particular solution of the task that we would like to force. One could ask what distinguishes the tasks we have chosen in this paper that makes them good no-programming problems. Let us state a few conditions that we consider important.

The desired solution should be the one of the solutions to the problem that minimizes the total time of inventing it and "execution" by hand. In particular, both the time complexity and the constant factor are important. Moreover, the solution should technically be simple, contain just a few different patterns of steps to be performed. For all this to be possible, one should carefully examine possible simple solutions of the problem with worse time complexities and heuristics that could work well for the chosen instance of the problem.

Additionally, it is better to exclude problems that require the knowledge of classical algorithms and advanced techniques. This is for the problems to be solvable by pupils without large algorithmic background and for the problems to be interesting to pupils that have some knowledge of algorithms. Finally, the best situation is when there exists a very simple but slow solution for the problem that should be obvious for anyone solving the problem – this makes the task more understandable.

References

Beaver Competition. `www.bebras.org`.

Dagienė, V. (2006). Information technology contests – Introduction to computer science in an attractive way. *Informatics in Education*, 5(1), 37–46.

Dagienė, V. (2010). Sustaining informatics education by contests. In: *4th International Conference on Informatics in Secondary Schools – Evolution and Perspectives, ISSEP*, LNCS 5941, 1–12.

Dagienė, V., Futschek, G. (2008). Bebras international contest on informatics and computer literacy: Criteria for good tasks. In: *3rd International Conference on Informatics in Secondary Schools – Evolution and Perspectives, ISSEP*, LNCS 5090, 19–30.

Delta. `www.mimuw.edu.pl/delta`.

Diks, K., Kubica, M., Radoszewski, J., Stencel, K. (2008). A proposal for a task preparation process. *Olympiads in Informatics*, 2, 64–74.

Diks, K., Kubica, M., Stencel, K. (2007). Polish olympiads in informatics – 14 years of experience. *Olympiads in Informatics*, 1, 50–56.

Forišek, M. (2006). On the suitability of programming tasks for automated evaluation. *Informatics in Education*, 5(1), 63–76.

Guzicki, W. (1997). Uni-color triangles (Trójkąty jednobarwne). In: *IV Polish Olympiad in Informatics 1996/1997*, 119–120 (in Polish).

Opmanis, M. (2009). Team competition in mathematics and informatics "Ugāle" – finding new task types. *Olympiads in Informatics*, 3, 80–100.

Project Euler. `http://projecteuler.net`.

Radoszewski, J. (2009). Non-informatics tasks (Zadanka (nie)informatyczne). *Delta*, 8, 1, 12–14 (in Polish).

Verhoeff, T., Horváth, G., Diks, K., Cormack, G. (2006). A proposal for an IOI Syllabus. *Teaching Mathematics and Computer Science*, 4(1), 193–216.

M. Kubica (1971), PhD in computer science, assistant professor at Institute of Informatics, Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, scientific secretary of Polish Olympiad in Informatics, IOI-ISC member and former chairman of Scientific Committees of IOI’2005 in Nowy Sącz, CEOI’2004 in Rzeszów, BOI’2001 in Sopot, and BOI’2008 in Gdynia, Poland. His research interests focus on combinatorial and text algorithms.

J. Radoszewski (1984), PhD student at Faculty of Mathematics, Informatics and Mechanics of University of Warsaw, chairman of the jury of Polish Olympiad in Informatics, former member of Host Scientific Committees of IOI’2005 in Nowy Sącz, CEOI’2004 in Rzeszów, and BOI’2008 in Gdynia, Poland. His research interests focus on text algorithms and combinatorics.