Computing Order Statistics in the Farey Sequence

Corina E. Pătraşcu[1] and Mihai Pătraşcu[2]

[1] Harvard University, Department of Mathematics
patrascu@fas.harvard.edu
[2] MIT, Computer Science and Artificial Intelligence Laboratory
mip@mit.edu

Abstract. We study the problem of computing the k-th term of the Farey sequence of order n, for given n and k. Several methods for generating the entire Farey sequence are known. However, these algorithms require at least quadratic time, since the Farey sequence has $\Theta(n^2)$ elements. For the problem of finding the k-th element, we obtain an algorithm that runs in time $O(n \lg n)$ and uses space $O(\sqrt{n})$. The same bounds hold for the problem of determining the rank in the Farey sequence of a given fraction. A more complicated solution can reduce the space to $O(n^{1/3}(\lg \lg n)^{2/3})$, and, for the problem of determining the rank of a fraction, reduce the time to $O(n)$. We also argue that an algorithm with running time $O(\text{poly}(\lg n))$ is unlikely to exist, since that would give a polynomial-time algorithm for integer factorization.

1 Introduction

For any positive integer n, the Farey sequence of order n is the set of all irreducible fractions $\frac{p}{q}$, with $0 < p < q \leq n$, arranged in increasing order. An alternative definition could include $\frac{0}{1}$ and $\frac{1}{1}$ as special fractions. For example, the Farey sequence for $n = 5$ is: $\frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}$. The number of elements of the Farey sequence is asymptotically $\frac{3}{\pi^2}n^2 + O(n \lg n)$ [5].

The Farey sequence is a well-known concept in number theory, whose exploration has lead to a number of interesting results. However, from an algorithmic point of view, very little is known. In particular, the only problem that appears to be investigated is that of generating the entire sequence for a given n. Several interesting solutions exist for this problem, though none of them presents any algorithmic challenge:

- Sort all unreduced fractions $\frac{p}{q}$, and remove duplicates. The running time is $O(n^2 \lg n)$, which is almost optimal, but the space is $O(n^2)$. (We assume that we are only interested in generating the fractions, not storing them; otherwise, quadratic space is clearly the best possible.)
- The space in the above algorithm can be reduced to $O(n)$, without changing the running time [7]. This uses a priority queue to merge n sequences, where the i-th such sequence is $\frac{1}{i}, \frac{2}{i}, \dots, \frac{i-1}{i}$.

- To obtain the sequence of order $n+1$ from the sequence of order n, consider all consecutive fractions $\frac{a}{b}$ and $\frac{c}{d}$ from the sequence of order n, and insert the mediant fraction $\frac{a+c}{b+d}$ between them, if the denominator is $n+1$ [5]. This surprising construction is based on the initial observation made by Farey in 1816 [4]. The resulting algorithm is the worst so far: the running time is $O(n^3)$, and the space is $O(n^2)$.
- Combining several properties satisfied by Farey sequence, one can get a trivial iterative algorithm, which generates the next Farey fraction, based on the previous two ([5], problem 4-61). If $\frac{p}{q}$ and $\frac{p'}{q'}$ are the last two fractions, the next one is given by:

$$p'' = \left\lfloor \frac{q+n}{q'} \right\rfloor p' - p, \quad q'' = \left\lfloor \frac{q+n}{q'} \right\rfloor q' - q$$

  This is an ideal algorithm: it uses $O(n^2)$ time, and $O(1)$ space.
- The Stern-Brocot tree is obtained by starting with $\frac{0}{1}$ and $\frac{1}{1}$, and repeatedly inserting the mediant between any two fractions that are consecutive in the in-order traversal of the tree [5]. Farey fractions form a subtree of the Stern-Brocot tree, often called the Farey tree. One can generate the Farey fractions in order, by recursively exploring the tree. The algorithm requires quadratic time, and $O(n)$ memory (corresponding to the maximum depth of the Farey fractions of order n).

The model of computation assumed by these algorithms, as well as the remaining algorithms from this paper, is the standard word RAM (Random Access Machine). Such a machine can access words of $O(\lg n)$ bits, and can perform usual arithmetic operations on such words in unit time. Space is also counted in words.

In this work, we consider the most natural question in addition to that of generating the entire Farey sequence. For given n and k, our problem is to generate just the k-th element of the Farey sequence of order n (often called the k-th order statistic [2]). Our motivation is not based on any practical application of this problem (we are aware of none), but rather on the algorithmic challenges it presents. It seems impossible to obtain good, or even just subquadratic time bounds for these problems by modifying the algorithms listed above (the obvious choice, and the one to which we devoted the most attention was the solution involving the Stern-Brocot tree). Instead, our algorithms will be based on a set of rather different ideas.

Our solution involves algorithms for another natural problem: given a fraction, determine its rank in the Farey sequence. The bounds we obtain are usually identical for both problems. In section 2, we describe a reduction between these problems, and design an initial algorithm that runs in time $O(n \lg^2 n)$ and uses space $O(n)$. In section 3, we improve the time complexity of this algorithm to $O(n \lg n)$. Finally, section 4 improves the space complexity to $O(\sqrt{n})$, while preserving the running time.

We have implemented the final algorithm, as well as the methods described above for generating the entire sequence, and found ours to be very efficient, both in terms of time, and space. Experimental results or source code can be obtained from the authors.

This result leaves two natural questions unanswered. The first one is how little memory suffices for an algorithm with roughly linear, say $O(n \cdot \mathrm{poly}(\lg n))$, running time. In particular, reducing the memory to $\mathrm{poly}(\lg n)$ would be interesting. In section 5, we present a more complex algorithm, based on a result of Deléglise and Rivat [3], which uses space $O(n^{1/3}(\lg\lg n)^{2/3})$. The algorithm can determine the rank of a fraction in $O(n)$ time; for our original problem (finding a fraction of a given rank) the $O(n \lg n)$ time bound is not improved.

The second question is concerned with time, rather than space. Note that the input to the problem consists of just two words, or $O(\lg n)$ bits, namely n and k. Therefore, there is nothing that prohibits the existence of an algorithm with running time sublinear in n. For somewhat related problems, such as computing the number of primes less than a certain value, sublinear time algorithms are known [1]. It seems reasonable to hope that the running time of the algorithm from section 5 can be improved to $O(n^{1-\varepsilon})$, for some constant $\varepsilon > 0$. We describe two subproblems which would be sufficient to obtain such a result, but which we cannot solve. In section 6 we argue instead that a much faster algorithm, with running time $O(\mathrm{poly}(\lg n))$, is unlikely to exist. More precisely, we show that such a polynomial algorithm for our problem would immediately imply a polynomial-time algorithm for integer factorization.

2 An Initial Algorithm

We now describe a first attempt to solve the problem, which will give an algorithm running in $O(n \lg^2 n)$ time and $O(n)$ space. This algorithm forms the basis of our improved algorithms from the following sections. Our solution uses as a subroutine an algorithm for determining the rank of a given fraction (not necessarily in reduced form) in the Farey sequence. The subroutine developed in this section will run in $O(n \lg n)$ time and $O(n)$ space.

We begin by a reduction from the order statistic problem to the fraction rank problem. Assume we want to compute the k-th order statistic. We first determine a number j such that the answer lies in the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$. This can be done by binary search, as follows. Assume we have a guess for the value of j. Determine the rank of $\frac{j}{n}$ in the Farey sequence. If this rank is at most k we know the correct value of j is not smaller than the current guess. Otherwise, we should continue searching below j. This stage of the algorithm uses $O(\lg n)$ calls to the fraction rank subroutine.

To solve the problem, we must now determine the fraction with rank equal to $k - rank(\frac{j}{n})$ among all irreducible fractions from the interval $\left[\frac{j}{n}, \frac{j+1}{n}\right)$. Notice that there is at most one fraction in this interval for any denominator less than n. This follows from the fact that the length of the interval is $\frac{1}{n}$, and consecutive fractions with denominators $q < n$ are separated by $\frac{1}{q} > \frac{1}{n}$. In addition, for a given q, this fraction can be found in constant time, since the numerator must be $\left\lfloor \frac{(j+1)q-1}{n} \right\rfloor$. There might not be any fraction in the range for a certain denominator, so we also need to check that the fraction we obtain in this way lies in the feasible interval.

Given these properties, it is not hard to find efficient algorithms for this subproblem. A particularly easy one would consist of generating all (up to n) fractions from the feasible interval, sorting them, and eliminating duplicates. Alternatively, one can make a list of just the irreducible fractions, and then use a linear time order statistic algorithm [2] on this list. However, we describe a more interesting solution. This solution has the advantage that it runs in $O(n)$ time, and uses just $O(1)$ memory, which will prove important in the following section. First, generate the fractions in the range by considering all possible denominators. As fractions are generated, keep just the minimum fraction found that is strictly greater than $\frac{j}{n}$. At the end, reduce both $\frac{j}{n}$ and the minimum fraction greater than it. We now have two consecutive fractions from the Farey sequence. As mentioned in the introduction, there is a simple constant-time algorithm that can generate the next fraction in the Farey sequence based on the previous two. This means that we can iterate through the fractions in the range in increasing order, remembering just the previous two fractions. All we have to do is count up to the desired rank, and return the corresponding fraction.

We are now left with giving an algorithm for the fraction rank problem. We will actually solve a slightly more general problem: for any real number x, determine the number of irreducible fractions $\frac{p}{q} \le x$, with $q \le n$. Let A_q be the number of such irreducible fractions with denominator equal to q. Any fraction with denominator q is either irreducible, in which case it should be counted in A_q, or has a unique reduced representation. The denominator of the reduced representation is a divisor of q. This transformation is, in fact, reversible: given any irreducible fraction $\frac{c}{d}$, where d is a divisor of q, we can multiply both the numerator and the denominator by q/d to get a fraction with denominator equal to q. So we have a bijection between the set of all fractions in $(0, x]$ with denominator q and the set of reduced fractions with denominator d, for all divisors d of q. This gives a recursive formula for A_q, which leads to the solution of our problem:

$$rank(x) = \sum_{q=1}^{n} A_q, \qquad A_q = \lfloor x \cdot q \rfloor - \sum_{d<q,\ d|q} A_d \tag{1}$$

The way to translate this formula into an efficient algorithm is obviously not the most direct one, since there is no fast way to iterate over all divisors of a number. Instead, begin by initializing an array $A[1..n]$ by $A[q] = \lfloor x \cdot q \rfloor$. Then consider all numbers q in increasing order from 1 to n. For each q, consider all multiples mq, and subtract $A[q]$ from $A[mq]$. At step number q, we always have $A[q] = A_q$, and this algorithm is a simple reformulation of the recursive formula from above. The rank can be computed by summing the final values of all $A[q]$. The running time is $O\left(n + \sum_{q=1}^{n} \frac{n}{q}\right) = O\left(n \sum_{q=1}^{n} \frac{1}{q}\right) = O(n \lg n)$. So we have an $O(n \lg n)$ algorithm for the rank problem, and thus an $O(n \lg^2 n)$ algorithm for the order statistic problem. Both algorithms use $O(n)$ space.

3 Improving the Running Time

We now improve the time complexity of the algorithm from the previous section. Since the running time of our order statistic algorithm is dominated by the running time of the fraction rank algorithm, we concentrate on the latter. As noted before, the order statistic algorithm makes $O(\lg n)$ calls to the fraction rank subroutine. The key idea suggested by this fact is that one should try to introduce some amount of preprocessing, which gives a one-time cost, in order to improve the running time of every call to the fraction rank subroutine.

To find a way to trade preprocessing for running time of the rank subroutine, let us re-examine relation 1. After recursive expansions of all A_q, the resulting $rank(x)$ will be a linear combination of terms of the form $\lfloor x \cdot q \rfloor$, for all $q \leq n$. Except for these terms, the recursive formula is independent of x. Therefore, we can precalculate the coefficient of every $\lfloor x \cdot q \rfloor$. Now the rank routine becomes trivial: for all q, calculate $\lfloor x \cdot q \rfloor$, and add this value to the rank, weighted by the appropriate coefficient. The numbers appearing at intermediate steps in the computation, and even the coefficients themselves, may become large. However, the end result (the rank) is obviously bounded by n^2, so if all computations are performed modulo a number greater than that, the result will be correct (this is important because we can only manipulate in constant time numbers that have $O(\lg n)$ bits). This issue is transparent to the implementation, since normally all computations are carried out modulo a power of two, such as 2^{32} for the usual 32-bit machines.

The algorithm for precalculating the coefficients of $\lfloor x \cdot q \rfloor$ is symmetric to our old algorithm for calculating the rank. It is based on the following recursion defining the coefficients:

$$C_q = 1 - \sum_{t > q,\ q \mid t} C_t, \quad \text{for all } q \leq n \tag{2}$$

The correctness of the formula follows from "reverse induction", since the calculation of C_q uses only coefficients C_t with $t > q$. Indeed, the term $\lfloor x \cdot q \rfloor$ appears initially in A_q. Then, A_q is subtracted from all its multiples t. At that point A_t contains $\lfloor x \cdot t \rfloor$ with coefficient 1, and $\lfloor x \cdot q \rfloor$ with coefficient -1. All subsequent operations involving A_t contribute to the total coefficient of $\lfloor x \cdot q \rfloor$ exactly by minus the coefficient of $\lfloor x \cdot t \rfloor$: since A_t is the only one that contains $\lfloor x \cdot t \rfloor$ initially, all operations involving A_t are described by the final coefficient of $\lfloor x \cdot t \rfloor$.

The algorithm follows immediately from the formula, and calculates the coefficients from C_n down to C_1. The running time is $O\left(\sum_{q=1}^{n} \frac{n}{q}\right) = O(n \lg n)$. This cost is paid once, and every call to the rank subroutine can be answered in $O(n)$ time, so the total running time of the order statistic algorithm is also $O(n \lg n)$.

4 Improving the Space Complexity

The key observation for improving the space complexity is that coefficients C_q and $C_{q'}$ are identical whenever $\lfloor n/q \rfloor = \lfloor n/q' \rfloor$. This is not hard to see, and also offers a good intuition for the new algorithm. Consider some value q. The term $\lfloor x \cdot q \rfloor$ is added initially to A_q. Then A_q, which includes $\lfloor x \cdot q \rfloor$, is subtracted from $m_1 q$, for all possible m_1. The values $A_{m_1 q}$, which now contain $\lfloor x \cdot q \rfloor$ with a coefficient of -1, are then subtracted from $A_{m_2 m_1 q}$, and so on. The branches of this recursion are only trimmed off when $q \prod m_i > n$, which is equivalent to $\prod m_i \geq \lfloor n/q \rfloor$. So the coefficient C_q only depends on $\lfloor n/q \rfloor$.

Based on this fact, we observe that there are only $\sqrt{n}$ distinct values for the coefficients among all C_q with $q > \sqrt{n}$. To avoid such repetitions, we break the coefficients into two groups. The coefficients $C_1, \ldots, C_{\lfloor \sqrt{n} \rfloor}$ are stored as before. Instead of storing C_q for $q > \sqrt{n}$, we store an array D_r, such that $C_q = D_{\lfloor n/q \rfloor}$ for any $q > \sqrt{n}$. Notice that both arrays have $O(\sqrt{n})$ elements. The fraction rank algorithm remains trivial. For all $q \leq \sqrt{n}$, we calculate $\lfloor x \cdot q \rfloor$ and add it to the rank, weighted by C_q. For the remaining q's, we instead use the weight $D_{\lfloor n/q \rfloor}$.

It remains to show how to precompute the sequences D_r and C_q using just $O(\sqrt{n})$ space. The computation of D_r is based on the following recursive formula:

$$D_r = 1 - \sum_{t=2}^{r} D_{\lfloor r/t \rfloor} \tag{3}$$

The formula can be obtained by careful relabeling of formula 2. Take a q, such that $r = \lfloor n/q \rfloor$. Now consider consider our previous recursive formula for C_q (slightly rewritten here):

$$C_q = 1 - \sum_{t=2}^{\lfloor n/q \rfloor} C_{tq}$$

By definition, we have $D_r = C_q$ and $\lfloor n/q \rfloor = r$. Also, $C_{tq} = D_{\lfloor n/tq \rfloor}$. The index on the right-hand side can be rewritten as $\left\lfloor \frac{n}{tq} \right\rfloor = \left\lfloor \frac{\lfloor n/q \rfloor}{t} \right\rfloor = \lfloor r/t \rfloor$, finalizing the transformation into formula 3.

Once we have the values D_r, computing the array C_q can be done as before, using relation 2. The only difference is that whenever the algorithm needs C_t for $t > \sqrt{n}$, is should instead use $D_{\lfloor n/t \rfloor}$, since we are only computing the values C_t for $t \leq \sqrt{n}$. The time required by the computation of the sequence D_r is quadratic in the size of the table, which is $O(n)$. Computing C_q takes $O(n \lg n)$ time, as before. Finally, rank queries still require linear time, so the overall running time is unchanged, and the space is reduced to $O(\sqrt{n})$.

5 A Better Way to Calculate Coefficients

This section describes a better way to calculate the coefficients D_r from the previous section, resulting in improved, but more complicated, algorithms. It can be seen that D_r is precisely equal to $M(r)$, where M is the summatory function of the Möbius function, $M(r) = \sum_{t=1}^{r} \mu(t)$. The Möbius function is defined by:

- $\mu(1) = 1$;
- $\mu(t) = 0$ if t has a squared prime factor;
- $\mu(t) = (-1)^k$ if $t = p_1 \cdot \ldots \cdot p_k$, where all p_i's are distinct and prime.

The identity between our coefficients and $M(r)$ is immediate, since $M(r)$ satisfies the recursive formula 3 defining our coefficients ([5], relation 4.61) – this is one of the fundamental properties of the Möbius function. Given this identity, we can use an algorithm by Deléglise and Rivat [3], which calculates $M(r)$ using $O(r^{2/3}(\lg\lg r)^{1/3})$ time and $O(r^{1/3}(\lg\lg r)^{2/3})$ space.

Remember that our algorithm needs the coefficients $C_q, q \in \{1, \ldots, n\}$ and that $C_q = D_{\lfloor n/q \rfloor}$. For all $q < n^{1/6}$, we will use the algorithm of [3] to calculate $M(\lfloor n/q \rfloor)$. This first stage will require $n^{1/6} \cdot n^{2/3+o(1)} = n^{5/6+o(1)}$ time. Calculating C_q for all $q \geq n^{1/6}$ can be done by calculating D_r for all $r \leq n^{5/6}$. Since $D_r = \sum_{t=1}^{r} \mu(t)$, calculating the D_r's in the increasing order of r reduces to calculating $\mu(t)$ for all $t \leq n^{5/6}$. In turn, calculating $\mu(t)$ is trivial if we have the prime factorization of t. There exist several factorization algorithms which can factor t in $t^{o(1)}$ time [6], so we obtain a running time of $n^{5/6+o(1)}$ for calculating all coefficients. The dominant factor in space usage is the space used by a call to the subroutine for calculating $M(r)$, which is $O(n^{1/3}(\lg\lg n)^{2/3})$.

The dominant factor in the running time is no longer calculating the coefficients. As each C_q becomes available, we need to multiply it by $\lfloor x \cdot q \rfloor$ and add the result to an accumulator. Thus, the running time for computing the rank of a fraction is $O(n)$. The time needed to compute the fraction of a given rank is still $O(n \lg n)$, because we run a binary search on top of the rank computation, as explained in section 2.

Since the algorithm of this section can compute all coefficients in sublinear time, there is hope of obtaining a sublinear running time overall. We now describe a possible approach; however, since we cannot solve the two necessary subproblems, this remains speculation. First, to compute ranks in sublinear time, we could observe that many consecutive terms of $\lfloor x \cdot q \rfloor$ have the same coefficient C_q when $q > \sqrt{n}$. Thus, we should look for an algorithm to evaluate $\sum_{q=a}^{b} \lfloor x \cdot q \rfloor$ in time $O((b-a)^{1-\varepsilon})$, for some constant $\varepsilon > 0$. Even this, however, would not imply a sublinear time algorithm for finding the fraction of a given rank. This is because the binary search from section 2 can only narrow down the range of possible fractions, and does not actually produce an output fraction in the form p/q. As described, the range is reduced to $\left[\frac{x}{n}, \frac{x+1}{n}\right)$; we do not know how to find a fraction of a given rank from this range without enumerating all $O(n)$ fractions. Alternatively, we could reduce the range to an interval of size $O(1/n^2)$, and then we would need to find the unique fraction left in this range.

6 Relation to Factorization

We now show that a polynomial-time algorithm (i.e. one running in $O(\text{poly}(\lg n))$ time) for our problem is unlikely to exist, since that would immediately give a polynomial algorithm for integer factorization. It is a well-known conjecture that such an algorithm does not exist.

Given a polynomial time algorithm for finding an order statistic, we can construct a polynomial time algorithm for finding the rank of a given fraction (this is the reverse of the reduction used for our actual algorithms). Observe that we can test whether a guessed rank is too high or too low, using only one oracle query to the order statistic algorithm: simply determine the fraction with that rank, and compare it with the input fraction. Therefore, we can use galloping binary search to solve the problem. We begin by trying powers of two until we either obtain a fraction greater than the input fraction, or we exceed the number of fractions in the Farey sequence (as reported by the order statistic algorithm). Then, we do a binary search for the correct rank in the interval between the last powers of two tried. This algorithm makes $O(\lg n)$ calls to the order statistic algorithm, and no other expensive computations, so it runs in polynomial time.

Our algorithm for factorization is based on yet another problem: given a number n, and $k \leq n$ that is relatively prime to n, report the number of integers in $[2, k]$ that are relatively prime to n. Given a polynomial time algorithm for this problem, we can use binary search to find a factor of n. Assume k is relatively prime to n (otherwise, we immediately get a factor), and we know the number of integers in $[2, k]$ that are relatively prime to n. If this number is $k - 1$, we know that the smallest factor of n is greater than k; otherwise, there is at least one factor below k.

It remains to describe the relation between the Farey sequence and this problem. Observe that all numbers $i \in [2, k]$ that are relatively prime to n give fractions $\frac{i}{n}$ in the Farey sequence of order n. To count these numbers, we begin by finding the rank of $\frac{k}{n}$ in the Farey sequence of order n. Then, we determine the largest fraction, strictly smaller than $\frac{k}{n}$. This can be done by one call to the order statistic algorithm, since we already know the rank of $\frac{k}{n}$. Since $\frac{k}{n}$ is irreducible (by assumption), and it is the mediant of neighboring fractions from the Farey sequence, it follows that the preceding fraction has a denominator strictly smaller than n. We now find the rank of this fraction, in the Farey sequence of order $n - 1$. Observe that the difference between the rank of $\frac{k}{n}$ and this rank is equal to the number of irreducible fractions $\frac{i}{n} \leq \frac{k}{n}$, which is what we wanted to count.

The reduction from above uses the order statistic oracle in one place, namely to find the largest fraction smaller than $\frac{k}{n}$. One may wonder whether the reduction holds if we assume just a polynomial time algorithm for the problem of ranking a fraction, proving also the hardness of this problem. The following smarter reduction answers this question in the affirmative. Consider the fractions $\frac{k+1}{n}$ and $\frac{k-1}{n}$. Since their difference is $\frac{2}{n}$, there exists exactly one fraction in this range with a denominator of $n - 1$. Find this fraction (with $O(1)$ arithmetic operations), and reduce it, which takes $O(\lg n)$ time. Now find the ranks of this fraction in the Farey sequences of order n and $n-1$. The difference in ranks is exactly the number of irreducible fractions $\frac{i}{n} < \frac{k}{n}$, possibly plus one due to $\frac{k}{n}$. This nonuniformity is easily fixed by testing whether our fraction with a denominator of $n-1$ is smaller or larger than $\frac{k}{n}$, so we can again count the number of integers from $[2, k]$ that are relatively prime to n.

7 Acknowledgements

We thank an anonymous referee for valuable comments leading to the results from section 5. This problem, with time and space limitations corresponding to our initial algorithm from section 2, was proposed by the authors as a competition problem for the Balkan Olympiad in Informatics 2003. We thank all people who expressed positive comments on that occasion. In addition, we thank Erik Demaine and Noam Elkies for discussions and encouragement.

References

1. E. Bach and J. Shallit, *Algorithmic Number Theory, Volume I: Efficient Algorithms*, MIT Press, 1996.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 2nd edition, MIT Press and McGraw-Hill, 2001.
3. M. Deléglise and J. Rivat, *Computing the Summation of the Möbius Function*, Experimental Mathematics 5, 291-295, 1996.
4. J. Farey, *On a Curious Property of Vulgar Fractions*, London, Edinburgh and Dublin Phil. Mag. 47, 385, 1816.
5. R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd edition, Addison-Wesley, 1994.
6. D. E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, 1981.
7. V. Mitrana, *Provocarea algoritmilor* (in Romanian), Ed. Agni, Bucharest, Romania, 1995.