ALGORITHM OF ADDING THE *M*-BIT NUMBERS

Katarzyna Woronowicz

Faculty of Computer Science, Bialystok University of Technology, Białystok, Poland

Abstract: In the classic algorithm of adding two *m*-bit numbers with carries we add a single bits of the added numbers on each of the *m* positions. If we assume for a single iteration of the algorithm to calculate the value of a single bit of the sum, then for each pair of *m*-bit numbers the algorithm executes *m* iterations. In this paper we propose a recursive algorithm of adding two numbers for which the number of the executed iterations is variable and ranges from 0 to *m*.

Keywords: addition, adding two numbers

1. Introduction

Adding numbers is a basic arithmetic operation performed by a computer. The best known algorithm of adding two binary numbers with carries works in the same way as the algorithm of adding two numbers represented in any positional system - it calculates the sum of the numbers in each position, and if the sum exceeds the base of the system, adds one to the next position. The classical algorithm and its time complexity is described, among others, in positions [2] and [1]. The practicalities of the issue of adding numbers is described for instance in [3]. In this paper we look at the binary numbers as sequences of bits, on which we perform logical operations. As a result, we found a new algorithm for calculating the sum of two numbers. Each iteration of the algorithm determines the successive approximations of the sum of two input numbers. Execution time depends on the structure of the binary representation of these numbers - the maximum number of iterations, required to determine the sum of two numbers, is equal to the length of the binary representation of the input numbers.

In this paper we present an algorithm of adding two positive integers which uses three bitwise operations: xor, and, shift to the left by one position. We prove its correctness and also present an alternative version of the algorithm, based on a

Advances in Computer Science Research, vol. 12, pp. 95-108, 2015.

recursive equation of the second degree. We calculate the average and the pessimistic time complexity of the algorithm. In the last section we summarize the average and pessimistic time complexity of the algorithm for different sizes of input data.

In this paper we use the following notation: additive group modulo 2^m is denoted by $(\mathbb{Z}_{2^m}, \oplus)$ and the inverse of the element $p \in \mathbb{Z}_{2^m}$ is denoted by $\ominus p$. The group of bit sequences X with xor operation is denoted as $(X, \otimes)$. The expressions $p \ll k$, $p \gg k$ mean shifting a sequence of bits p of k positions to the left or to the right respectively. The length of a bit sequence is denoted by m. In the binary representation of the number p bits are numbered from right to the left, i.e. $p = [p^{m-1}, p^{m-2}, \ldots, p^1, p^0]$. The i-th bit of the number p is denoted by p^i. Notation $p^{[j \ldots i]}$, where $j > i$, means a fragment of the binary representation of the number p from the i-th bit to the j-th bit.

2. Algorithm

In this section we introduce the recursive algorithm of adding two m-bit numbers. The basic form of the algorithm is presented in the Theorem 1. The alternative form of the algorithm is described in the Proposition 1.

Lemma 1. *Let $p, q \in \mathbb{Z}_{2^m}$. Then:*

1. $p \otimes q = (p \vee q) \oplus (\ominus(p \wedge q))$,
2. $p \oplus q = (p \vee q) \oplus (p \wedge q)$,
3. $p \oplus q = (p \otimes q) \oplus ((p \wedge q) \ll 1)$.

Proof.

1. Note that:

$$(p \wedge q) \otimes p = (\neg(p \wedge q) \wedge p) \vee (p \wedge q \wedge \neg p) = (\neg(p \wedge q) \wedge p) =$$

$$((\neg p \vee \neg q) \wedge p) = (\neg p \wedge p) \vee (\neg q \wedge p) = p \wedge \neg q,$$

therefore:

$$p \otimes q = (p \vee q) \wedge (\neg(p \wedge q)) = ((p \vee q) \wedge (p \wedge q)) \otimes (p \vee q) = (p \wedge q) \otimes (p \vee q).$$

If the i-th bit of the expression $p \wedge q$ is equal 1, then the i-th bit of the expression $p \vee q$ is also equal 1. Therefore, when we subtract $p \wedge q$ of $p \vee q$ we have no borrows and an operation $(p \vee q) \oplus (\ominus(p \wedge q))$ is equivalent to xor operation $(p \vee q) \otimes (p \wedge q)$.

2. Consider two numbers $p, q \in \mathbb{Z}_{2^m}$. Note that:

$$p \vee (\neg p \wedge q) = (p \vee \neg p) \wedge (p \vee q) = 1 \wedge (p \vee q) = p \vee q$$

and

$$q \wedge (\neg(\neg p \wedge q)) = q \wedge (p \vee \neg q) = (q \wedge p) \vee (q \wedge \neg q) = (p \wedge q) \vee 0 = p \wedge q.$$

Because $p \wedge (\neg p \wedge q) = 0$, i.e. there is no index i such that the i-th bit of the expression p is equal 1 and the i-th bit of the expression $(\neg p \wedge q)$ is equal 1, so when we add p and $\neg p \wedge q$ we have no carries and:

$$p \vee (\neg p \wedge q) = p \oplus (\neg p \wedge q).$$

Similarly if the i-th bit of $\neg p \wedge q$ is equal 1, then the i-th bit of q also is equal 1, so:

$$q \wedge (\neg(\neg p \wedge q)) = q \oplus (\ominus(\neg p \wedge q)).$$

Thus, finally:

$$p \oplus q = p \oplus (\neg p \wedge q) \oplus q \oplus (\ominus(\neg p \wedge q)) = (p \vee q) \oplus (p \wedge q).$$

3. Because $p \otimes q = (p \vee q) \oplus (\ominus(p \wedge q))$, so $p \vee q = (p \otimes q) \oplus (p \wedge q)$. Thus:

$$p \oplus q = (p \vee q) \oplus (p \wedge q) = (p \otimes q) \oplus (p \wedge q) \oplus (p \wedge q) = (p \otimes q) \oplus ((p \wedge q) \ll 1).$$

□

Theorem 1. *Let $p, q \in \mathbb{Z}_{2^m}$. Then $p \oplus q = p_m$, where p_m is the m-th term of a sequence defined as follows:*

$$\begin{cases} p_0 = p \\ q_0 = q \\ p_{n+1} = p_n \otimes q_n \\ q_{n+1} = (p_n \wedge q_n) \ll 1 \end{cases} \tag{1}$$

Proof. In the proof we use induction due to the m.

1. If $m = 1$ then $p \oplus q = p \otimes q = p_1$.

2. Let $n < m$.
   Suppose that: if $p, q \in \mathbb{Z}_{2^n}$, then $p \oplus q = p_n$.
   We show that: if $p, q \in \mathbb{Z}_{2^{n+1}}$, then $p \oplus q = p_{n+1}$.
   For the proof, consider the first iteration of the algorithm. Note that:

$$(p \oplus q)^0 = (p \otimes q)^0.$$

   Indeed, the expression $(p_n \wedge q_n) \ll 1$ is always an even number, which means that $((p_n \wedge q_n) \ll 1)^0 = 0$. Therefore:

$$(p \oplus q)^0 = (p \otimes q)^0 \otimes ((p_n \wedge q_n) \ll 1)^0 = (p \otimes q)^0 \otimes 0 = (p \otimes q)^0 = p_1^0.$$

   Denote $(p \otimes q)^0$ as r_0. Now we consider the remaining n bits of the numbers p_1 and q_1:

$$p_1^{[n\ldots1]} = (p \otimes q) \gg 1,$$

$$q_1^{[n\ldots1]} = ((p \wedge q) \ll 1) \gg 1 = p \wedge q.$$

   Note that $(n+1)$-th term of the sequence $\{p_k\}_{k\in\mathbb{N}}$ for numbers p, q is equal to n-th term of the sequence $\{p_k\}_{k\in\mathbb{N}}$ for numbers $p \otimes q = p_1$ and $(p \wedge q) \ll 1 = q_1$. Furthermore:

$$p \oplus q = p_1 \oplus q_1 = (p \otimes q) \oplus ((p \wedge q) \ll 1).$$

   Since, by Lemma (1), $p \oplus q = (p \otimes q) \oplus ((p \wedge q) \ll 1)$ and because the length of sequences $p_1^{[n\ldots1]}$, $q_1^{[n\ldots1]}$ equals n, so from the assumption:

$$p_1^{[n\ldots1]} \oplus q_1^{[n\ldots1]} = (p_1^{[n\ldots1]} \otimes q_1^{[n\ldots1]}) \oplus ((p_1^{[n\ldots1]} \wedge q_1^{[n\ldots1]}) \ll 1) = p_{n+1}^{[n\ldots1]}.$$

   If we include bits of p^0 and q^0 we obtain:

$$((p_1 \oplus q_1) \ll 1) \oplus r_0 = ((((p \otimes q) \gg 1) \oplus (p \wedge q)) \ll 1) \oplus r_0 =$$
$$((((((p \vee q) \oplus (\ominus(p \wedge q))) \gg 1) \oplus (p \wedge q))) \ll 1) \oplus r_0 =$$
$$((((p \vee q) \gg 1) \oplus ((p \wedge q) \gg 1)) \ll 1) \oplus r_0 = (p \vee q) \oplus (p \wedge q) \oplus r_0 = p \oplus q. \square$$

Lemma 2. *Let $p, q \in \mathbb{Z}_{2^m}$. Then:*

$$p \wedge (p \oplus q) = p \wedge \neg q.$$

Proof. Note that:

$$p \wedge (p \oplus q) = p \wedge ((\neg p \wedge q) \vee (p \wedge \neg q)) =$$
$$= (p \wedge (\neg p \wedge q)) \vee (p \wedge (p \wedge \neg q)) = 0 \vee (p \wedge \neg q) = (p \wedge \neg q).$$

□

Proposition 1. *System of recursive equations (1) can be converted into the recursive equation of the second degree:*

$$p_{n+2} = p_{n+1} \otimes ((p_n \wedge \neg p_{n+1}) \ll 1). \tag{2}$$

Proof. By definition $p_{n+2} = p_{n+1} \otimes q_{n+1}$ and $q_{n+1} = (p_n \wedge q_n) \ll 1$. Because $p_{n+1} = p_n \otimes q_n$, so $q_n = p_n \otimes p_{n+1}$. Thus:

$$p_{n+2} = p_{n+1} \otimes q_{n+1} = p_{n+1} \otimes ((p_n \wedge q_n) \ll 1) =$$
$$= p_{n+1} \otimes ((p_n \wedge (p_n \otimes p_{n+1})) \ll 1).$$

By Lemma (2) we have $p_n \wedge (p_n \otimes p_{n+1}) = p_n \wedge \neg p_{n+1}$, so we obtain:

$$p_{n+2} = p_{n+1} \otimes ((p_n \wedge \neg p_{n+1}) \ll 1).$$

□

2.1 Stopping conditions

Note that if the term $q_n = 0$, then $q_{n+1} = \ldots = q_m = 0$ and $p_n = p_{n+1} = \ldots = p_m$. Indeed, if $q_n = 0$, then:

$$p_{n+1} = p_n \otimes q_n = p_n \otimes 0 = p_n$$

and:

$$q_{n+1} = (p_n \wedge q_n) \ll 1 = (p_n \wedge 0) \ll 1 = 0.$$

It means that if $q_n = 0$, then for all $i = n+1, \ldots, m$ the term $p_i = p_n$, therefore if $q_n = 0$, then $p \oplus q$ is equal to p_n and we can stop the algorithm - successive iterations will not change the result. Consequently, if $q = 0$, the number of iterations is equal to 0. Hence we can formulate the following lemma:

Lemma 3.

1. The algorithm (1) executes more than n iterations if and only if $q_n \neq 0$.
2. The algorithm (1) executes exactly $n+1$ iterations if and only if $q_n \neq 0$ and $q_{n+1} = 0$.

2.2 Example

Let $p = p_0 = 22_{10} = 10110_2$, $q = q_0 = 27_{10} = 11011_2$, $m = 5$. Then:

$$p_1 = p_0 \otimes q_0 = 10110 \otimes 11011 = 01101$$
$$q_1 = (p_0 \wedge q_0) \ll 1 = (10110 \wedge 11011) \ll 1 = 00100$$

$$p_2 = p_1 \otimes q_1 = 01101 \otimes 00100 = 01001$$
$$q_2 = (p_1 \wedge q_1) \ll 1 = (01101 \wedge 00100) \ll 1 = 01000$$

$$p_3 = p_2 \otimes q_2 = 01001 \otimes 01000 = 00001$$
$$q_3 = (p_2 \wedge q_2) \ll 1 = (01001 \wedge 01000) \ll 1 = 10000$$

$$p_4 = p_3 \otimes q_3 = 00001 \otimes 10000 = 10001$$
$$q_4 = (p_3 \wedge q_3) \ll 1 = (00001 \wedge 10000) \ll 1 = 00000$$

According to the Lemma 3, the next iteration is not necessary: the term $p_5 = p_4 \otimes q_4 = 10001 \otimes 00000 = p_4$ and the term $q_5 = (p_4 \wedge q_4) \ll 1 = (10001 \wedge 00000) \ll 1 = 00000 = q_4$. Note that in one case the order of input numbers p and q is important. If $p \neq 0$ and $q = 0$ then the algorithm executes no iteration, but if $p = 0$ and $q \neq 0$ then is executed one iteration.

3. Time complexity

The single iteration of the basic algorithm (1) executes three operations: xor, and, shift to the left by one position. Let us assume that the elementary operation of the algorithm is to execute one of those three bitwise operations. This is a simplification, because each of this operation in fact works on m-bit sequences. However, note that if k-th bit is the first nonzero bit of the number $p_n \wedge q_n$, then $p_n^{[k,k-1,\ldots,0]} = p_{n+1}^{[k,k-1,\ldots,0]} = \ldots = p_m^{[k,k-1,\ldots,0]}$. Consequently, it is not necessary to execute operations on all m bits. In this paper we have focused mainly on the number of iterations executed by an algorithm, so we assume this simplification. Then the number of elementary operations which executes algorithm while adding of two numbers is equal to the number of iterations multiplied by three. Therefore, in order to examine the time complexity of the algorithm, we should count pessimistic and average number of executed iterations. Note that the minimum number of iterations is equal to 0 and the maximum number of iterations is equal to m.

3.1 Average time complexity

Let $p, q \in \mathbb{Z}_m$. To determine the average time complexity of the algorithm we should calculate for how many pairs (p,q) the algorithm executes a given number of iterations. By Lemma (3), the algorithm executes more than 0 iterations if and only if:

$$q_0 \neq 0. \tag{3}$$

It is easy to count that the number of pairs (p,q) for which condition 3 is fulfilled is equal to $2^m \cdot (2^m - 1)$.

By Lemma (3), a necessary and sufficient condition that the number of executed iterations is greater than 1 is:

$$q_1 = (p_0 \wedge q_0) \ll 1 \neq 0. \tag{4}$$

We count for how many pairs (p,q) this condition is fulfilled. Expression $p \wedge q$ is non-zero if and only if there exists index i such that $p^i = q^i = 1$. Because we consider the expression $(p \wedge q) \ll 1$, so for $i = m-1$ we have:

$$p \wedge q = 2^{m-1} \oplus (p \wedge q)^{[m-2,\ldots,0]}$$

and:

$$(p \wedge q) \ll 1 = (2^{m-1} \oplus (p \wedge q)^{[m-2,\ldots,0]}) \ll 1 = (p \wedge q)^{[m-2,\ldots,0]} \ll 1,$$

so such choice of an index i does not guarantee that the condition (4) is fulfilled. Thus $i \in \{0,1,\ldots,m-2\}$.

To compute for how many pairs (p,q) there exists at least one index i such that $p^i = q^i = 1$ we use the inclusion-exclusion principle. First we calculate the number of pairs (p,q) such that an index i is fixed and remaining bits of numbers p, q take all possible values. It means, that for a fixed index i there exists an index $j \neq i$ such that $p^j = q^j = 1$ and some pairs (p,q) are computed more than once - sets of pairs (p,q) such that $p^i = q^i = 1$ and $p^j = q^j = 1$ are not disjoint. Thus in the next step we compute for how many pairs there exists two indices i_1, i_2 such that $p^{i_1} = q^{i_1} = 1$ and $p^{i_2} = q^{i_2} = 1$, then we do the calculations for the $3,4,\ldots,k$ indices and we apply the inclusion-exclusion principle.

We consider the case when there is at least one fixed index i such that $p^i = q^i = 1$. We can choose the index i in $m-1$ ways, the remaining $m-1$ bits of the number p and $m-1$ bits of the number q we fill in all possible ways, therefore there is $(m-1) \cdot (2^{m-1})^2$ pairs (p,q) satisfying the condition.

Now we count for how many pairs there are at least two fixed indices i_1, i_2 such that $p^{i_1} = q^{i_1} = 1$ and $p^{i_2} = q^{i_2} = 1$. We can choose indices in $\binom{m-1}{2}$ ways, the

remaining $m-2$ bits of p and q take all possible values, so there is $\binom{m-1}{2} \cdot (2^{m-2})^2$ possibilities.

Generally for k indices $i_1, \ldots, i_k$, $k \leq m-1$, there is $\binom{m-1}{k} \cdot (2^{m-k})^2$ pairs such that $p^{i_1} = q^{i_1} = 1, \ldots, p^{i_k} = q^{i_k} = 1$. Thus, from inclusion-exclusion principle, a number of pairs for which $(p \wedge q) \ll 1 \neq 0$ is equal to:

$$\sum_{i=1}^{m-1} (-1)^{i+1} \cdot \binom{m-1}{i} \cdot (2^{m-i})^2.$$

Using similar reasoning we count for how many pairs the algorithm executes more than two iterations. By lemma (3), this is true if and only if:

$$q_2 = (p_1 \wedge q_1) \ll 1 = ((p_0 \otimes q_0) \wedge ((p_0 \wedge q_0) \ll 1)) \ll 1 \neq 0. \tag{5}$$

The condition (5) is fulfilled if and only if for a pair (p,q) there exists an index i such that:

$$(p \otimes q)^i = ((p \wedge q) \ll 1)^i = 1 \tag{6}$$

and

$$(p \otimes q)^{i+1} \otimes ((p \wedge q) \ll 1)^{i+1} = 1. \tag{7}$$

We count the number of the pairs (p,q) for which there exists at least one fixed index i such that conditions (6), (7) are fulfilled. Note that after the second iteration of the algorithm we execute two shifts to the left by one position. It means that, as in the previous case, that if $i = m-2$, then condition (5) need not to be fulfilled. It is obvious, that if $i = m-1$, then (7) is not satisfied, because p and q are m-bit numbers and bits p^m, q^m do not exist. Therefore $i \in \{0, 1, \ldots, m-3\}$. Besides, the condition (7) means that if we choose an index i, then we have two possibilities for bits p^{i+1} and q^{i+1}

$$p^{i+1} = 0 \text{ and } q^{i+1} = 1 \tag{8}$$

or:

$$p^{i+1} = 1 \text{ and } q^{i+1} = 0 \tag{9}$$

Thus the number of pairs p, q for which there exists at least one fixed index i satisfying the conditions (6) and (7) is equal to $(m-2) \cdot (2^{m-2})^2 \cdot 2$.

Denote the number of possible choices of k indices $i_1, \ldots, i_k$, $k \leq \lfloor \frac{(m-1)}{2} \rfloor$, as $a^2_{m,k}$. If we choose sets of k indices for the pair $(p^{[m-2,\ldots,0]}, q^{[m-2,\ldots,0]})$, then all of

these sets of k indices can be chosen for the pair (p,q). Additionally we can choose sets containing index $m-1$. Since the choice of index i causes that index $i+1$ cannot be selected, we should consider all possible $k-1$-element sets of indices without index $m-2$ and add index $m-1$ to all of them, which means that we choose all sets of indices for the pair $p^{[m-3,\ldots,0]}$, $q^{[m-3,\ldots,0]}$. Therefore a sequence $\{a^2_{m,k}\}$ is defined by the following recursive equation:

$$\begin{cases} a^2_{m,k} = m-2 & , k=1, m \geq 3, \\ a^2_{m,k} = a^2_{m-1,k} + a^2_{m-2,k-1} & , k>1, m \geq 3. \end{cases} \tag{10}$$

If we choose k indices $i_1,\ldots,i_k$, then each pair of bits p_{i_j+1}, q_{i_j+1}, $j=1,\ldots,k$ must satisfy condition (7), so each such pair fulfill condition (8) or (9). The remaining $m-2k$ bits of p and q take all possible values, so from inclusion-exclusion principle, the number of pairs for which conditions (6) and (7) are satisfied is equal to:

$$\sum_{i=1}^{\lfloor \frac{(m-1)}{2} \rfloor} (-1)^{i+1} \cdot a^2_{m,i} \cdot 2^{2m-3i}.$$

Now we consider for how many pairs the algorithm executes more than three iterations. This is true if and only if:

$$\begin{gathered} q_3 = (p_2 \wedge q_2) \ll 1 = \\ = \Big(((p_0 \otimes q_0) \otimes ((p_0 \wedge q_0) \ll 1)) \wedge ((p_0 \otimes q_0) \wedge ((p_0 \wedge q_0) \ll 1)) \ll 1\Big) \ll 1 \neq 0. \end{gathered} \tag{11}$$

Similarly as in the previous case, the condition (11) is fulfilled if for pair (p,q) exists index i such that:

$$(p \otimes q)^i = ((p \wedge q) \ll 1)^i = 1 \tag{12}$$

and

$$(p \otimes q)^{i+1} \otimes ((p \wedge q) \ll 1)^{i+1} = 1 \tag{13}$$

and

$$(p \otimes q)^{i+2} \otimes ((p \wedge q) \ll 1)^{i+2} = 1. \tag{14}$$

If there is at least one such fixed index i, then, because we made already three shifts to the left, we can select it from indices $\{0,1,\ldots,m-4\}$, so we can choose i in $m-3$ ways. We can fill pairs of bits (p^{i+1},q^{i+1}) and (p^{i+2},q^{i+2}) in $2\cdot 2$ ways, the remaining $m-3$ bits of p and $m-3$ bits of q we can fill in all possible ways, so there is $(m-3)\cdot(2^{m-3})^2 \cdot 2^2$ pairs p, q satisfying the conditions (12), (13) and

(14) for which there exists at least one such index i. Now we define sequence $a^3_{m,k}$ analogously to the sequence (10).

If we choose sets of k indices for pair $p^{[m-2,\ldots,0]}$, $q^{[m-2,\ldots,0]}$, $k \leq \lfloor \frac{(m-1)}{3} \rfloor$, then all of these sets of indices can be chosen for pair p, q. We should join all sets containing index $m-1$ to the selected family of sets. Since the choice of index i causes that indices $i+1$ and $i+2$ cannot be selected, we should consider all $k-1$-elements sets of indices for pair $p^{[m-4,\ldots,0]}$, $q^{[m-4,\ldots,0]}$. We obtain the sequence defined as follows:

$$\begin{cases} a^3_{m,k} = m-3 & , k = 1, m \geq 4, \\ a^3_{m,k} = a^3_{m-1,k} + a^3_{m-3,k-1} & , k > 1, m \geq 4. \end{cases} \tag{15}$$

The number of pairs for which conditions (12), (13) and (14) are satisfied is equal to:

$$\sum_{i=1}^{\lfloor \frac{(m-1)}{3} \rfloor} (-1)^{i+1} \cdot a^3_{m,i} \cdot 2^{2m-4i}.$$

Now we define the general formula for the number of pairs, for which algorithm (1) executes more than n iterations. The necessary and sufficient condition, by lemma (3) is that $q_n \neq 0$. We count how many pairs satisfy this condition. First, we define a sequence $a^n_{m,k}$. After the n-th iteration algorithm executed n shifts to the left by one position. The choice of index i causes that $p^{i+1} \otimes q^{i+1} = 1, p^{i+2} \otimes q^{i+2} = 1, \ldots, p^{i+(n-1)} \otimes q^{i+(n-1)} = 1$. The remaining bits of p, q we can fill in all possible ways. The recursive formula for the general case is defined as follows:

$$\begin{cases} a^n_{m,k} = m-n & , k = 1, m \geq n+1, \\ a^n_{m,k} = a^n_{m-1,k} + a^n_{m-n,k-1} & , k > 1, m \geq n+1. \end{cases} \tag{16}$$

Theorem 2. *An explicit formula for the term $a^n_{m,k}$ is given as follows:*

$$a^n_{m,k} = \frac{\big(m-((n-1)\cdot k+1)\big)\cdot\big(m-((n-1)\cdot k+2)\big)\cdot\ldots\cdot\big(m-n\cdot k\big)}{k!} = \\ = \binom{m-((n-1)\cdot k+1)}{k} \tag{17}$$

Proof. We know that:

$$a^n_{m,k} = a^n_{m-1,k} + a^n_{m-n,k-1}.$$

We can use the formula (17) for terms $a^n_{m-1,k}$ and $a^n_{m-n,k-1}$:

$$a^n_{m-1,k} = \frac{\big(m-1-((n-1)\cdot k+1)\big)\cdot\big(m-1-((n-1)\cdot k+2)\big)\cdot\ldots\cdot\big(m-1-n\cdot k\big)}{k!}$$

and

$$a^n_{m-n,k-1} = \frac{\big(m-n-((n-1)\cdot(k-1)+1)\big)\cdot\big(m-n-((n-1)\cdot(k-1)+2)\big)\cdot\ldots\cdot\big(m-n-n\cdot(k-1)\big)}{(k-1)!}.$$

Note that:

$$m-1-((n-1)\cdot k+1) = m-n-((n-1)\cdot(k-1)+1),$$
$$m-1-((n-1)\cdot k+2) = m-n-((n-1)\cdot(k-1)+2),$$
$$\ldots$$
$$m-1-((n-1)\cdot k+k-1) = m-n-(n\cdot(k-1)).$$

Thus:

$$a^n_{m,k} = a^n_{m-1,k} + a^n_{m-n,k-1} =$$
$$= \frac{\big(m-1-((n-1)\cdot k+1)\big)\cdot\big(m-1-((n-1)\cdot k+2)\big)\cdot\ldots\cdot\big(m-1-n\cdot k\big)}{k!} +$$
$$+ \frac{\big(m-n-((n-1)\cdot(k-1)+1)\big)\cdot\big(m-n-((n-1)\cdot(k-1)+2)\big)\cdot\ldots\cdot\big(m-n-n\cdot(k-1)\big)}{(k-1)!} =$$
$$= \frac{\Big(\big(m-1-((n-1)\cdot k+1)\big)\cdot\big(m-1-((n-1)\cdot k+2)\big)\cdot\ldots\cdot\big(m-1-((n-1)\cdot k+k-1)\big)\Big)\cdot\Big(m-1-n\cdot k+k\Big)}{k!} =$$
$$= \frac{\Big(\big(m-((n-1)\cdot k+2)\big)\cdot\big(m-((n-1)\cdot k+3)\big)\cdot\ldots\cdot\big(m-((n-1)\cdot k+k)\big)\Big)\cdot\Big(m-((n-1)\cdot k+1)\Big)}{k!} =$$
$$= \frac{\big(m-((n-1)\cdot k+1)\big)\cdot\big(m-((n-1)\cdot k+2)\big)\cdot\ldots\cdot\big(m-n\cdot k\big)}{k!} =$$
$$= \binom{m-((n-1)\cdot k+1)}{k}.$$

□

Therefore, the general formula for computing the number of pairs (p,q) for which the algorithm (1) executes more than n iterations is given as follows:

$$\sum_{i=1}^{\lfloor\frac{(m-1)}{n}\rfloor} (-1)^{i+1}\cdot a^n_{m,i}\cdot 2^{2m-(n+1)\cdot i}. \tag{18}$$

Denote the sum from (18) as $A_{n,m}$. Let $A_{0,m} = 2^m\cdot(2^m-1)$ and, for the formalities, let $A_{-1,m} = 2^{2m}$. Then the average number of the algorithm iterations is described by the formula:

$$I_{\text{avg}}(m) = \frac{1}{2^{2m}}\cdot\sum_{i=0}^{m}(A_{i-1,m}-A_{i,m})\cdot i = \frac{1}{2^{2m}}\cdot\sum_{i=0}^{m-1}A_{i,m} - m\cdot A_{m,m}, \tag{19}$$

and, because $A_{m,m} = 0$, we obtain:

$$I_{\text{avg}}(m) = \frac{1}{2^{2m}} \cdot \sum_{i=0}^{m-1} A_{i,m}. \tag{20}$$

The average time complexity of the algorithm is equal to:

$$T_{\text{avg}}(m) = \frac{3}{2^{2m}} \cdot \sum_{i=0}^{m-1} A_{i,m}. \tag{21}$$

3.2 Pessimistic time complexity

The number of pairs (p,q), for which the algorithm (1) executes more than $m-1$ iterations is equal to $A_{m,m-1}$. In the pessimistic case the algorithm executes m iterations, so there are $A_{m,m-1} = 2^m$ such pairs (p,q). Therefore pessimistic data, for which algorithm executes $3 \cdot m$ elementary operations are $\frac{1}{2^m}$ of possible data.

4. Experiments

Up to now we have focused on the average number of iterations of the algorithm, necessary for calculating the sum of the two numbers. We not yet managed to find a compact form of the function, which estimates the average time complexity of the algorithm. The average number of iterations and average time complexity, presented in the Table (1), are calculated by formulas (20) and (21) respectively.

For a few selected cases we compared the calculated results with the average complexity obtained by inserting a counter into the code of implemented algorithm and counting complexity by executing the algorithm for all possible pairs of numbers. The results are consistent. In the tests for the m-bit sequences we received average numbers of iterations presented in the Table (1). The results obtained show that, for the tested cases, the average number of iterations of the algorithm is close to the square root of m, and for m bigger than 20 does not exceed the square root of m.

In future studies we plan to determine the compact function expressing the average number of algorithm iterations depending on the size of input data. Another issue that we intend to explore is the exact time complexity of the algorithm, understood as the average number of logical operations executed on the individual bits of the input, instead, as in this paper, on the sequences of bits. In the calculations we will take into account the fact that, depending on the structure of the input numbers, in the second and successive iterations logical operations can be executed on sequences of the changeable length. Making such analysis will allow us to compare the time complexity of the proposed algorithm with the complexity of the classical algorithm.

Table 1. Average time complexity.

m	Average number of iterations	Average time complexity
3	1,4375	4,3125
4	1,82812	5,48438
5	2,16797	6,50391
6	2,46582	7,39746
7	2,72632	8,17896
8	2,95636	8,86908
9	3,16039	9,48116
10	3,34288	10,02865
11	3,50719	10,52158
12	3,65619	10,96857
13	3,79216	11,37647
14	3,91700	11,75099
15	4,03225	12,09674
16	4,13919	12,41756
17	4,23887	12,71662
18	4,33219	12,99657
19	4,41987	13,25961
20	4,50255	13,50764
25	4,85694	14,57081
30	5,14105	15,42316
35	5,37821	16,13464
40	5,58179	16,74536
45	5,76012	17,28036
50	5,91879	17,75637
55	6,06171	18,18512
60	6,19172	18,57515
65	6,31096	18,93289
70	6,42110	19,26330
75	6,52342	19,57025
80	6,61895	19,85686
85	6,70856	20,12567
90	6,79292	20,37876
95	6,87262	20,61786
100	6,94815	20,84444
200	7,96248	23,88744
300	8,55218	25,65653
400	8,96957	26,90870
500	9,29291	27,87873

5. Acknowledgment

The author is grateful to Professor Czesław Bagiński and Professor Wiktor Dańko for their help during the preparation of the paper, insightful observations and useful tips.

References

[1] D. E. Knuth. *The art of computer programming, Vol. 2 Seminumerical Algorithms*. Reading, Massachusetts: Addison-Wesley.

[2] A. A. Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics*, 1995.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest Introduction to Algorithms. *MIT Press and McGraw-Hill*, 1990.

ALGORYTM DODAJĄCY DWIE M-BITOWE LICZBY

Streszczenie W klasycznym algorytmie dodawania dwóch m-bitowych liczb z przeniesieniami dodajemy po kolei bity na poszczególnych pozycjach binarnych reprezentacji danych wejściowych. Jeśli przyjmiemy za iterację algorytmu wyznaczenie wartości pojedynczego bitu sumy, to dla każdej pary m-bitowych liczb algorytm wykonuje m iteracji. W niniejszej pracy proponujemy rekurencyjny algorytm dodawania dwóch liczb, który w pojednynczej iteracji wykonuje trzy operacje logiczne, a liczba iteracji wynosi od 0 do m.

Słowa kluczowe: dodawanie, dodanie dwóch liczb