Математика *Volume 62,* *Mathematics*
и информатика *Number 3, 2019* *and Informatics*

Educational Technologies
Образователни технологии

ON THE CONCEPT OF BITWISE OPERATIONS IN THE PROGRAMMING COURSES

Krasimir Yordzhev
South-West University "Neofit Rilski" – Blagoevgrad (Bulgaria)

Abstract. This article is intended for anyone who studies programming, as well as for his teachers. The article deals with the some aspects of teaching programming languages C++ and Java. It presents some essential and interesting examples of the advantages of using bitwise operations to create efficient algorithms in programming. Particular attention is paid to the entertaining task of writing a program receiving Latin squares of arbitrary order.

Keywords: bitwise operations; binary notation; set; Latin square

1. Introduction

The present study is thus especially useful for students educated to become programmers as well as for their lecturers. In the article we present some meaningful examples for the advantages of using bitwise operations for creating effective algorithms. To implement the algorithm, we will use essentially bitwise operations.

The use of bitwise operations is a powerful means during programming with the languages C/C++ and Java. Some of the strong sides of these programming languages are the possibilities of low-level programming. Some of the means for this possibility are the introduced standard bitwise operations, with the help of which it is possible directly operating with every bit of an arbitrary variable situated in the computer memory. Unfortunately, in the widespread books, the topic on effective using of bitwise operations is incomplete or missing. The aim of this article is to correct this lapse to a certain extent and present some meaningful examples of a programming task, where the use of bitwise operations is appropriate in order to facilitate the work and to increase the effectiveness of the respective algorithm. For a deeper study of this subject, we recommend the book [Yordzhev, 2019].

For more details see, for example, in [Davis, 2014, Kernighan and Ritchie, 1998, Todorova, 2002a, Todorova, 2002b] for C/C++ programming languages and in [Evans and Flanagan, 2015, Hadzhikolev and Hadzhikoleva, 2016, Schildt, 2014, Schildt, 2017] for Java programming language. In section 2 we will only recall the definitions of the basic concepts.

2. Bitwise operations – basic definitions

The bitwise operations apply only to integer data type, i.e. they cannot be used for float and double types.

We assume, as usual that bits numbering in variables starts from right to left, and that the number of the very right one is 0.

Let x,y and z be integer variables or constants of one type, for which *w* bits are needed. Let x and y be initialized (if they are variables) and let the assignment z = x & y; (*bitwise AND*), or z = x | y; (*bitwise inclusive OR*), or z = x ^ y; (*bitwise exclusive OR*), or z = ~x; (*bitwise NOT*) be made. For each $i = 0,1,2,\dots,w-1$ $i = 0,1,2,\dots,w-1$, the new contents of the i-th bit in z will be as it is presented in the Table 1.

Table 1: Bitwise operations

i-th bit of x	i-th bit of y	i-th bit of z = x&y;	i-th bit of z = x|y;	i-th bit of z = x^y;	i-th bit of z = ~x;
	0	0	0	0	1
	1	0	1	1	1
	0	0	1	1	0
	1	1	1	0	0

In case that k is a nonnegative integer, then the statement z = x<<k (*bitwise shift left*) will write $(i+k)$ in the bit of z the value of the k bit of x, where $i = 0,1,\dots,w-k-1$, and the very right k bits of x will be filled by zeroes. This operation is equivalent to a multiplication of x by 2^k.

The statement z=x>>k (*bitwise shift right*) works in a similar way. However, we must be careful if we use the programming language C or C + +, as in various programming environments (see for example in [Glushakov et al., 2001]), this operation has different interpretations. Somewhere k bits of z from the very left place are by requirement filled by 0 (logical displacement), and elsewhere the very left k bits of z are filled with the value from the very left (sign) bit, i.e. if the number is negative, then the filling will be with 1 (arithmetic displacement). Therefore, it is recommended to use unsigned types of variables (if the opposite is not necessary) while working with bitwise operations. In the Java programming language, this problem is solved by introducing the two different operators: z=x>>k and z=x>>>k [Evans and Flanagan, 2015, Schildt, 2017].

Bitwise operations are left associative.

The priority of operations in descending order is as follows:

• – ~ (bitwise NOT);

• – the arithmetic operations * (multiply), / (divide) and % (remainder or modulus);
• – the arithmetic operations + (addition) and – (subtraction);
• – the bitwise operations << and >>;
• – the relational operations <, >, <= and >=;
• – the relational operations == and !=;
• – the bitwise operations &,| and ^;
• – the logical operations && and ||.

Example 1. *Directly form the definition of the operation bitwise shift left follows the effectiveness of the following function computing 2^k, where k is a nonnegative integer:*

```
unsigned int Power2(unsigned int k) {
        return 1<<k;
}
```

Example 2. *To divide the integer x into 2^S by cutting the remaining of the result (the "remainder" operation $f(x) = x \ \% \ (2^n)$), we can take advantage of the C++ function:*

```
int Div2(int x, unsigned int n)
{
        int s = x<0 ? -1 : 1; // s is sign of x
        x = x*s; // or x = abs(x);
        return (x>>n)*s;
}
```

Example 3. *To compute the value of the i-th bit (0 or 1) in computer presentation of a nonnegative integer variable x we can use the function:*

```
int BitValue (int x, unsigned int i) {
        return ( (x & 1<<i) == 0 ) ? 0 : 1;
}
```

3. Bitwise operations in relation to the binary representation of integers

In order to understand all the possibilities of bitwise operations, there is a need for a solid knowledge of the notion of "*number system*". The great applications of different number systems and most of all the binary system for representation of integers in computer science and in programming are well known. The main features of this concept are studied in secondary schools and universities. In these courses, the representation of integers in different number systems is studied in detail. In computer science and programming, it is particularly important to present the integers in *binary notation* (*binary number system*).

Example 4. *The next function prints an integer in binary notation. We do not consider and we do not print the sign of integer. For this reason, we work with |n| (absolute value, or module of integer n).*

```
void DecToBin(int n)
{
        n = abs(n);
        int b;
        int d = sizeof(int)*8 - 1;
        while ( d>0 && (n & 1<<(d-1) ) == 0 ) d--;
        while (d>=0)
                {
                b= 1<<(d-1) & n ? 1 : 0;
                cout<<b; d--;
                }
}
```

Example 5. *The following function calculates the number of 1's in the binary notation of an integer n. Again, we ignore the sign of the number.*

```
int NumbOf_1(int n)
{
        n = abs(n);
        int temp=0;
        int d = sizeof(int)*8 - 1;
        for (int i=0; i<d; i++)
                if (n & 1<<i) temp++;
        return temp;
}
```

When working with negative numbers, we have to take into account that computer presentation of negative numbers became a special way. How to represent the integers of type short can be seen from Example 6, where we essentially use bitwise operations. The function that we will describe is a modification of the function from Example 4.

Integer of type short	Representation in the computer's memory.
0	0000000000000000
1	0000000000000001
-1	1111111111111111
2	0000000000000010
-2	1111111111111110
$16 = 2^4$	0000000000010000
$-16 = -2^4$	1111111111110000
$26 = 2^4 + 2^3 + 2$	0000000000011010
$-26 = -(2^4 + 2^3 + 2)$	1111111111100110
$41 = 2^5 + 2^3 + 1$	0000000000101001
$-41 = -(2^5 + 2^3 + 1)$	1111111111010111
$32767 = 2^{15} - 1$	0111111111111111
$-32767 = -(2^{15} - 1)$	1000000000000001
$32768 = 2^{15}$	1000000000000000
$-32768 = -2^{15}$	1000000000000000

Table 2: Representation of some integers of type short in the computer's memory

Example 6. *Function showing the representation of the integers of type short in the computer's memory.*

```
void BinRepl(short n)
{
        int b;
        int d = sizeof(short)*8 - 1;
        while (d>=0)
                {
                b= 1<<d & n ? 1 : 0;
                cout << b;
                d--;
                }
}
```

Some experiments with the function BinRepl(short) are given in Table 2.

4. The bitwise operations and sets

Let n be a positive integer. In the present work we will only consider values of n, such that $1 \le n \le "sizeof(long)" * 8 - 1$, where sizeof(long)=4 for C++ programming language and sizeof(long)=8 for Java programming language. Throughout $\mathcal{U}$ denotes the set

$$\mathcal{U} = \{1,2,\dots,n\}.$$

Let $A \subseteq \mathcal{U}A \subseteq \mathcal{U}$. We denote by $\mu_i(A)$ the functions

$$\mu_i(A) = \begin{cases} 1 & if \quad i \in A \\ 0 & if \quad i \notin A \end{cases}, \quad i = 1,2,\dots. \tag{1}$$

Then the set A can be represented uniquely by the integer

$$\nu(A) = \sum_{i=1}^{n} \mu_i(A) 2^{i-1}, \quad 0 \le \nu(A) \le 2^n - 1, \tag{2}$$

where $\mu_i(A)$, $i = 1,2,\dots,n$ is given by formula (1). In other words, each subset of $\mathcal{U}$, we will represent uniquely with the help of an integer from the interval $[0, 2^n - 1]$ (*integer representation of sets*).

It is readily seen that

$$\nu(\mathcal{U}) = 2^n - 1. \tag{3}$$

Evidently if $A = \{a\}$, i.e. $|A| = 1$, then

$$\nu(\{a\}) = 2^{a-1}. \tag{4}$$

The empty set $\emptyset$ is represented by

$$\nu(\emptyset) = 0. \tag{5}$$

The expressions, that represent operations with sets, we will use in the next section 5 for a description of the algorithm to obtain a random exponential Latin square.

1. According to equation (3), the set $\mathcal{U} = \{1,2,\dots n\}$ is represented by the equation

$$U = (1 << n) - 1;$$

2. According to equation (4), a singleton $A = \{a\} \subset \mathcal{U}$ is represented by the equation

$$A = 1 << (a - 1);$$

3. Let A and B be two integers, which represent two subsets of $\mathcal{U}$. Then the integer C that represents their union is represented by the equation

$$C = A|B;$$

4. Let $A \subseteq B$. Then, to remove all elements of A from B, we can do it with the help of the equation

B^A.

In programming languages C/C++ and Java there is no standard type ”set” and standard operations with sets [Todorova, 2011]. In this case we have to look for additional instruments to work with sets - for example the associative containers set and multiset realized in Standard Template Library (STL) [Azalov, 2008, Collins, 2003, Horton, 2015, Lischner, 2009, Wilson, 2007]. It can be used the template class set of the system of computer algebra ”Symbolic C++”, programming code is given in details in [Tan et al., 2000]. Of course we can be built another class set, and specific methods of this class can be described, as a training. This is a good exercise for the students, when the cardinal number of the basic (”universal”) set $\mathcal{U}$ is not very big. Below we give the solution of this task using bitwise operations [Kostadinova and Yordzhev, 2011, Yordzhev, 2018a].

Example 7. *The class Set_N describes construction and operations with sets of integers by means of overloading of operators and using bitwise operations:*

```
class Set_N
{
 /*
        The set is encoded by a non-negative integer n in binary notation:
 */
        unsigned int n;

        public:

 /*
        Constructor without parameter – it creates the empty set:
 */
        Set_N()
        {
        n = 0;
        }
 /*
Constructor with parameter – it creates a set containing the integer i, if and only
if the i-th bit of the parameter k is 1:
 */
        Set_N(unsigned int k);
        {
                n = k;
        }
 /*
```

```
		Returns the integer n that encodes the set:
	*/
		get_n()
		{
			return n;
		}
	/*
```
The intersection $A \cap B$ of two sets. This operation we will denote with A*B:
```
	*/
		Set_N operator * (Set_N B)
		{
			return (this->n) & B.get_n();
		}
	/*
```
The union $A \cup B$ of two sets. This operation we will denote with A+B:
```
	*/
		Set_N operator + (Set_N B)
		{
			return (this->n) | B.get_n();
		}
	/*
```
The union $A \cup \{k\}$ of the set A with the one-element set $\{k\}$. This operation we will denote with A+k:
```
	*/
		Set_N operator + (int k)
		{
			return (this->n) | (1<<(k-1));
		}
	/*
```
Adding the integer $k \in \mathcal{U}$ to the set A. This operation we will denote with k+A. Here we have to note that from the algorithmic point of view A + k and k + A are realized differently, taking into account the standard of C++ programming language, regardless of commutativity for the operation of union of two sets:
```
	*/
		friend Set_N operator + (int, Set_N);
	/*
```
Removing the integer k from the set A. If $k \notin A$ then A does not change. This operation we will denote with A-k:
```
	*/
		Set_N operator – (int k)
		{
```

```
                int temp = (this->n) ^ (1<<(k-1));
                return (this->n) & temp;
        }
    /*
```
By definition $A\backslash B = \{k \mid k \in A and k \notin B\}$. This operation we will denote with A-B:
```
    */
        Set_N operator – (Set_N B)
        {
                int temp = this->n ^ B.get_n();
                return (this->n) & temp;
        }
    /*
```
Checking whether $A \supseteq B$, that is, whether the set A contains the subset B. This operation we will denote with A>=B. The result is true or false:
```
    */
        bool operator >= (Set_N B)
        {
                return (this->n | B.get_n()) == this->n;
        }
    /*
```
Checking whether $A \subseteq B$, that is, whether the set A is subset of the set B. This operation we will denote with A<=B. The result is true or false:
```
    */
        bool Set_operator <= (Set_N B)
        {
                return (this->n | s.get_n()) == B.get_n();
        }
    /*
```
Verifying that sets A and B are equal to each other we will denote with A==B. The result is true or false:
```
    */
        bool operator == (Set_N B)
        {
                return ((this->n ^ B.get_n()) == 0);
        }
    /*
```
Checking whether the sets A and B are different will be denoted by A!=B. The result is true or false:
```
    */
        bool operator != (Set_N B)
        {
                return !((this->n ^ B.get_n()) == 0);
        }
/*
    Checks whether the integer k belongs to the set:
 */
        bool in(int k)
        {
                return this->n & (1<<(k-1));
        }
}

Set_N operator + (int k, Set_N A)
{
        return (1<<(k-1)) | A.get_n();
}
```

5. An entertainment example – algorithm to obtain a random Latin square using bitwise operations

A *Latin square* of order n is a $n \times n$ matrix where each row and column is a permutation of elements of the set $\{1,2, \ \dots, n\}$. Thus far it is known the number of all Latin squares of order n, where $n \leq 11$ [McKay and Wanless, 2005, Sloane, 2019].

Latin squares and hypercubes have their applications in coding theory, error correcting codes, information security, decision making, statistics, cryptography, conflict-free access to parallel memory systems, experiment planning, tournament design, enumeration and study of H-functions, etc [Kovachev, 2011, Laywine and Mullen, 1998].

A special kind of Latin squares are the Sudoku-matrices [Yordzhev, 2018b].

Definition. [Yordzhev, 2016] *A matrix $M_{n\times n} = \left(\alpha_{ij}\right)_{n\times n}$ is called an exponential Latin square of order n if the following condition hold:*

1. For every $i, j \in \{1,2, \dots, \mathrm{n}\}$ there exists $k \in \{1,2, \dots, \mathrm{n}\}$, such that $\alpha_{i,j} = 2^k$;
2. The matrix $M_{n\times n}{}' = \left(\log_2 \alpha_{ij}\right)_{n\times n}$ is a Latin square of order n.

If n is a positive integer, then it is readily seen that the set of all $n \times n$ Latin squares and the set of all $n \times n$ exponential Latin squares are isomorphic.

In this section we will show that it is easy to create an algorithm for generating random exponential Latin squares of order n using bitwise operations. The presentation of the subsets of the set $\mathcal{U} = \{1,2, \dots n\}$ using the formula (2) and the convenience in this case to work with bitwise operations are the basis of algorithm described by us. Some other algorithms for obtaining random Latin squares and random Sudoku matrices and their valuation are described in detail in [DeSalvo, 2017, Fontana, 2011, Yordzhev, 2012, Yordzhev, 2016].

Example 8. *A program code for obtaining $N \times N$ random Latin squares:*

```
#define N 12 // N is the order of the Latin square.

int U = (1<<N) – 1;
        /* U represents the universal set {1,2, ... ,N}
                according to equation (3)
        */
int L[N][N];
        /*
        L – exponential Latin square
*/

int choice(int k)
{
        /* In this case, “choice” means random choice of a 1
        from the binary notation of positive integer k, 0<k<=U */
if (k<=0 || k>U)
{
        cout<<”The choice is not possible \n”;
        return 0;
}
int id =0;
        /* id – the number of 1 in the binary notation of k */
for (int i=0; i<N; i++)
{
        if (k & 1<<i) id++;
}
srand(time(0));
int r = rand();
        /* r is a random integer, 1<=r<=id. The function will
        chooses the r-th 1 from the binary notation of k.*/
int s=0, t=1;
while (1)
{
        if (k&t) s++;
```

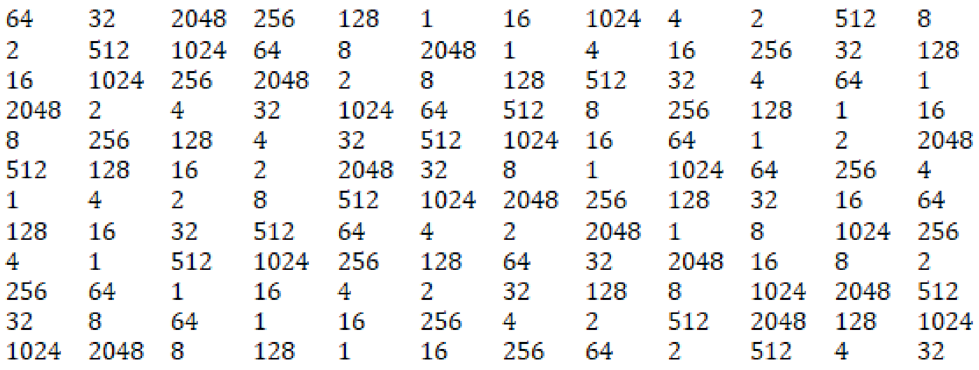
```
            /* s-th bit of the integer k is equal to 1. */
        if (s==r) return t;
            /* The function returns the integer t=2^{r-1},
            which represents the singleton {r}. */
        t=t<<1;
            /* t=t*2 and check the next bit */
        }
}

void use()
{
            /* In this case "use" means "printing" */
        for (int i=0; i<N; i++)
        {
            for (int j=0; j<N; j++)
            {
                cout<<L[i][j]<<" ";
            }
                cout<<endl;
            }
}

int main()
{
        int row,col;
        int A;
            /* A represents a subset of {1,2,...,n}
            according to equation (2) */
        for (row=0; row<N; row++)
        {
            col =0;
            while (col<N) {
                A=0; // empty set
                for (int i=0; i<row; i++) A = A | L[i][col];
                for (int j=0; j<col; j++) A = A | L[row][j];
                A = U ^ A;
                /* The algorithm will select an element of this set
                and insert it into the next position. */
                if (A!=0)
            {
                L[row][col] = choice(A);
```

```
                        col++;
                        }
                        else col = 0;
                }
        }
        use();
return 0;
}
```

With the help of algorithm described in Example 8, we received a lot of random exponential Latin squares, for example the next one of order 12:



```
64    32    2048  256   128   1     16    1024  4     2     512   8
2     512   1024  64    8     2048  1     4     16    256   32    128
16    1024  256   2048  2     8     128   512   32    4     64    1
2048  2     4     32    1024  64    512   8     256   128   1     16
8     256   128   4     32    512   1024  16    64    1     2     2048
512   128   16    2     2048  32    8     1     1024  64    256   4
1     4     2     8     512   1024  2048  256   128   32    16    64
128   16    32    512   64    4     2     2048  1     8     1024  256
4     1     512   1024  256   128   64    32    2048  16    8     2
256   64    1     16    4     2     32    128   8     1024  2048  512
32    8     64    1     16    256   4     2     512   2048  128   1024
1024  2048  8     128   1     16    256   64    2     512   4     32
```

REFERENCES

Azalov, P. (2008). *Object-oriented programming. Data structures and STL*. Ciela, Sofia. (in Bulgarian)

Collins, W. (2003). *Data structures and the standard template library*. McGraw-Hill, New York.

Davis, S. R. (2014). *C++ for Dummies*. John Wiley & Sons, N.J., 7 edition.

DeSalvo, S. (2017). Exact sampling algorithms for latin squares and sudoku matrices via probabilistic divide-and-conquer. *Algorithmica*, 79(3): 742 – 762.

Evans, B. J. and Flanagan, D. (2015). *Java in a Nutshell*. O'Reilly, 6 edition.

Fontana, R. (2011). Fractions of permutations. an application to sudoku. *Journal of Statistical Planning and Inference*, 141(12): 3697 – 3704.

Glushakov, S. V., Koval, A. V. and Smirnov, S. V. (2001). *The C++ Programming Language*. Folio, Kharkov. (in Russian)

Hadzhikolev, E. and Hadzhikoleva, S. (2016). *Fundamentals of programming with Java*. University Publishing House ”Paisiy Hilendarski”, Plovdiv, (in Bulgarian).

Horton, I. (2015). *Beginning STL: Standard Template Library*. Apress.

Kernighan, B. W. and Ritchie, D. M. (1998). *The C programming Language*. AT&T Bell Laboratories, N.J., 2 edition.

Kostadinova, H. and Yordzhev, K. (2011). An entertaining example for the usage of bitwise operations in programming. In *Proceedings of the Fourth International Scientific Conference – FMNS2011*, volume 1, pages 159–168, Blagoevgrad, Bulgaria. SWU "N. Pilsky".

Kovachev, D. S. (2011). On some classes of functions and hypercubes. *Asian-European Journal of Mathematics*, 4(3):451 – 458.

Laywine, C. F. and Mullen, G. L. (1998). *Discrete Mathematics Using Latin Squares*. Wiley Series in Discrete Mathematics and Optimization (Book 49). John Wiley & Sons, New York.

Lischner, R. (2009). *STL Pocket Reference*. O'Reilly Media.

McKay, B. D. and Wanless, I. M. (2005). On the number of latin squares. *Annals of Combinatorics*, 9(3):335 – 344.

Schildt, H. (2014). *Java: The Complete Reference*. McGraw-Hill, 9 edition.

Schildt, H. (2017). *Java: A Beginner's Guide*. McGraw-Hill, 7 edition.

Sloane, N. J. A. (2019). A002860 – number of latin squares of order n; or labeled quasigroups. *The On-Line Encyclopedia of Integer Sequences (OEIS)*. http://oeis.org/A268523, Last accessed on 9 April 2016.

Tan, K. S., Steeb, W.-H., and Hardy, Y. (2000). *Symbolic C++: An Introduction to Computer Algebra using Object-Oriented Programming*. Springer-Verlag, London.

Todorova, M. (2002a). *Programming in C ++*, volume 1. Ciela, Sofia (in Bulgarian).

Todorova, M. (2002b). *Programming in C ++*, volume 2. Ciela, Sofia (in Bulgarian).

Todorova, M. (2011). *Data structures and programming in C ++*. Ciela, Sofia (in Bulgarian).

Wilson, M. D. (2007). *Extended STL: Collections and iterators*. Extended STL. Addison-Wesley.

Yordzhev, K. (2012). Random permutations, random sudoku matrices and randomized algorithms. *International Journal of Mathematical Sciences and Engineering Applications*, 6(VI): 291 – 302.

Yordzhev, K. (2016). Bitwise operations in relation to obtaining latin squares. *British Journal of Mathematics & Computer Science*, 17(5): 1 – 7.

Yordzhev, K. (2018a). The bitwise operations in relation to the concept of set. *Asian Journal of Research in Computer Science*, 1(4): 3072 – 3079.

Yordzhev, K. (2018b). How does the computer solve sudoku - a mathematical model of the algorithm. *Mathematics and informatics*, 61(3): 259 – 264 (in Bulgarian, abstract in English).

Yordzhev, K. (2019). *Bitwise Operations and Combinatorial Applications*. LAP Lambert Academic Publishing.

✉ **Assoc. Prof. Krasimir Yordzhev, DSc.**
Faculty of Natural Sciences
South-West University "Neofit Rilski"
Blagoevgrad, Bulgaria
E-mail: yordzhev@swu.bg