The Bitwise Operations Related to a Fast Sorting Algorithm

Krasimir Yordzhev

Faculty of Mathematics and Natural Sciences
South-West University
Blagoevgrad, Bulgaria

Abstract—in the work we discuss the benefit of using bitwise operations in programming. Some interesting examples in this respect have been shown. What is described in detail is an algorithm for sorting an integer array with the substantial use of the bitwise operations. Besides its correctness we strictly prove that the described algorithm works in time O(n). In the work during the realization of each of the examined algorithms we use the apparatus of the object-oriented programming with the syntax and the semantics of the programming language C++

Keywords—*bitwise operations; programming languages C/C++ and Java; sorting algorithm*

I. INTRODUCTION

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 to directly operate with every bit of an arbitrary variable situated in the random access memory of the computer. In the current article we are going to describe some methodical aspects for work with the bitwise operations.

As an interesting example of application of the bitwise operations comes the realised by us algorithm for sorting an integer array, for which we strictly prove its correctness and the fact that this algorithm will use operations included in the standard of the programming language C++. A main role in the realisation of the algorithm plays the bitwise operations.

II. BITWISE OPERATIONS

The bitwise operations can be applied for integer data type only. For the definition of the bitwise operations and some of their elementary applications could be seen, for example, in [2], [5] for C/C++ programming languages and in [4], [7] for Java programming language.

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 and y be integer variables or constants and let z be integer variables 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 $i = 0, 1, 2, \ldots, w-1$, the

new contents of the i -th bit in z will be as it is presented in the Table I.

TABLE I.   BITWISE OPERATIONS IN PROGRAMMING LANGUAGES C/C++ AND JAVA

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

In case that k is a nonnegative integer, then the statement z = x<<k; (bitwise shift left) will write in the $(i+k)$ bit of z the value of the k bit of x, where $i = 0, 1, \ldots, w-k-1$, 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 the similar way. But we must be careful if we use the programming language C or C++, as in various programming environments this operation has different interpretations: somewhere k bits of z from the very left place are compulsory 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 type of variables (if the opposite is not necessary) while working with bitwise operations (see also Example 3). In the Java programming language, this problem is solved by introducing the two different operators: z=x>>k and z=x>>>k [4], [7].

Bitwise operations are left associative.

The priority of operations in descending order is as follows: ~ (bitwise NOT); the arithmetic operations * (multiply), / (divide), % (remainder or modulus); the arithmetic operations + (addition) - (subtraction); the bitwise operations << and >>; the relational operations <, >, <=, >=, ==, !=; the bitwise operations &,^ and |; the logical operations && and ||.

III. SOME ELEMENTARY EXAMPLES OF USING THE BITWISE OPERATIONS

*Example 1:***1** To compute the value of the i-th bit (0 or 1) of an integer variable x we can use the function:

```
int BitValue(int x, unsigned int i) {

    int b = ((x & 1<<i) == 0) ? 0 : 1;

    return b;

}
```

Example 2: **2** Directly from the definition of the operation bitwise shift left (<<) follows the efficiency of the following function computing 2^n, where n is a nonnegative integer:

```
unsigned int Power2(unsigned int n) {

    return 1<<n;

}
```

Example 3: **3** The integer function $f(x) = x \% 2^n$ implemented using operation bitwise shift right (>>).

```
int Div2(int x, unsigned int n) {

    int s = x<0 ? -1 : 1;

/* s = the sign of x */

    x = x*s;

/* We reset the sign bit of x */

    return (x>>n)*s;

}
```

When we work with negative numbers we must consider that in the computer the presentation of the negative numbers is through the so called true complement code. The following function gives us how to code the integers in the memory of the computer we work with. For simplicity we are going to work with type short, but it is not a problem for the function to be overloaded for other integer types, too.

Example 4: 4 A function showing the presentation of the numbers of type short in the memory of the computer.

```
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 are given in Table II.

TABLE II. PRESENTATION OF SOME NUMBERS OF TYPE SHORT IN THE MEMORY OF THE COMPUTER

An integer of type short	Presentation in 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

Compare the function presented in Example 4 to the next function presented in Example 5.

Example 5: 5 A function that prints a given integer in binary notation.

```
void DecToBin(int n) {

    if (n<0) cout<<'-';

/* Prints the sign - , if n<0: */

    n = abs(n);

    int b;

    int d = sizeof(int)*8 - 1;

    while ( d>0 && (n & 1<<d ) == 0 ) d--;

/* Skips the insignificant zeroes at the beginning: */

    while (d>=0) {

        b= 1<<d & n ? 1 : 0;

        cout<<b;

        d--;

    }

}
```

Example 6: 6 The following function calculates the number of 1 in a given integer n written in a binary notation. Here again we ignore the sign of the number (if it is negative) and we work with its absolute value.

```
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;

}
```

IV. BITWISE SORTING

In this section we are going to suggest a fast algorithm for sorting an arbitrary integer array. And since during its realisation we are substantially going to use bitwise operations, we will call it "**Bitwise sorting**". We will prove that the bitwise sorting works in time $O(n)$, where n is the size of the array. This is an excellent evaluation regarding the criterion time. For comparison below we give some of the most famous sorting algorithms and their evaluations by criterion time [1], [6], [8].

- **Selection sort** – works in time $O(n^2)$;

- **Bubble sort** – works in time $O(n^2)$;

- **Bidirectional bubble sort** (**Cocktail sort**) – works in time $O(n^2)$;

- **Insertion sort** – works in time $O(n^2)$;

- **Merge sort** – works in time $O(n \log n)$;

- **Tree sort** – works in time $O(n \log n)$;

- **Timsort** – works in time $O(n \log n)$;

- **Counting sort** – works in time $O(n+m)$, where m is a second parameter, giving the number of the unique keys;

- **Bucket sort** – works in time $O(n)$.

Notes:

1) Timsort has been developed for use with the programming language Python[??].

2) For Counting sort $O(n+m)$ additional memory is necessary.

3) For Bucket sort $O(m)$ additional memory is necessary, where m is another parameter, giving the number of the unique keys, and it is also necessary to have knowledge of the nature of the sorted data which goes beyond the functions "swap" and "compare".

V. PROGRAMME CODE OF THE ALGORITHM

The algorithm created by us, described with the help of programming language C++, is shown below (algorithm 1). Due to some obvious reasons, first we create a function which sorts an array whose elements are either only nonnegative, or only negative. The second function divides the given array into

two disjoint subarrays respectively only with negative and only with nonnegative elements. After sorting each one of these subarrays, we merge them so that we obtain one finally sorted array.

Algorithm 1:7 Bitwise sorting.

```
/* The first function sorts integer elements with the same signs: */

template <class T>

void BitwiseSort1(T A[],int n){

int t;

 /* t - size of the type T in bits */

    t = sizeof(T)*8;

    T A0[n], A1[n];

 /* A0 remembers the elements for which the k-th bit is 0 */

 /* A1 remembers the elements for which the k-th bit is 1 */

    int n0; // size of A0

    int n1; // size of A1

    for (int k=0; k<t-1; k++) {

 /* k - number of the bit. The numeration starts from 0.
Does not check the sign bit */

        n0=0;

        n1=0;

        for (int i=0; i<n; i++){

 /* checks the k-th bit of the i-th element of the array: */

        if (A[i] & 1<<k) {

            A1[n1] = A[i];

            n1++;

            }

        else {

            A0[n0] = A[i];

             n0++;

            }

        }

 /* We merge the two arrays. As a result we obtain an array
whose elements if the k-th bit is equal to 0 are at the beginning,
and if it is equal to 1 at the end. */

        for (int i=0; i<n0; i++)

            A[i] = A0[i];

        for (int i=0; i<n1; i++)

            A[n0+i] = A1[i];

    }
```

```
}

/* The second function sorts the whole array */

template <class T>

void BitwiseSort(T A[],int n){

    T Aminus[n], Aplus[n];

/* Aminus[n] - An array with the negative values of A */

/* Aplus[n] - An array with the nonnegative values of A */

    int Nm = 0, Np = 0;

/* Nm -- number of elements written in Aminus */

/* Np -- number of elements written in Aplus */

    for (int i=0; i<n; i++)

        if (A[i] < 0) {

            Aminus[Nm] = A[i];

            Nm++;

        }

        else {

            Aplus[Np] = A[i];

            Np++;

        }

/* Sorts the negative elements: */

    BitwiseSort1(Aminus,Nm);

/* Sorts the nonnegative elements */

    BitwiseSort1(Aplus,Np);

/* We merge the two arrays: */

    for (int i=0; i<Nm; i++)

        A[i] = Aminus[i];

    for (int i=Nm; i<n; i++)

        A[i] = Aplus[i-Nm];

}
```

VI.   EVALUATION OF THE ALGORITHM

As a main disadvantage of the algorithm described by us comes the fact that it is applicable only to arrays of integers or symbols (type char). This is because for it we substantially use bitwise operations, which are applicable only over integer types of data. But this disadvantage is compensated by its high speed. As we will see below, algorithm 1 works in time $O(n)$, where n is the number of the elements which are subjected to sorting.

Except through the multiple experiments which we have made, with the help of the following theorem we will prove the correctness of the algorithm created by us.

Theorem 1:8 During every execution of algorithm 1:7 with an arbitrary input array of integers, as a result a sorted array is obtained.

Proof. It is enough to prove that function BitwiseSort1 works so as to fulfill the conditions of the theorem.

Let $A = \{a_0, a_1, \ldots, a_{n-1}\}$ be an arbitrary integer array with length n and let $A^{(k)} = \{a_o^{(k)}, a_1^{(k)}, a_{n-1}^{(k)}\}$ be the array which is obtained after iteration with number k, where $k = 0, 1, \ldots, t-2$, t =sizeof(T)*8, i.e. t is equal to the number of the bits which every element of A occupies in the memory of the computer.

Let x be an integer. For every natural number $k = 0, 1, 2, \ldots$ we define the functions:

$$\mu_k(x) = x \% 2^k,$$

where just like in programming languages C/C++ and Java the operator % denotes the remainder during integer division. Apparently $\mu_{s-1}(x) = x$ if the absolute value of the integer x can be written with no more than s digits 0 or 1 in a binary notation. Therefore in order to prove that as a result of the work of the algorithm the array $A^{(t-2)}$ is sorted, it is enough to prove that the array

$$\overline{A^{(t-2)}} = \{\mu_{t-2}(a_0^{(t-2)}), \mu_{t-2}(a_1^{(t-2)}), \ldots, \mu_{t-2}(a_{n-1}^{(t-2)})\}$$

is sorted. Applying inductive reasoning, we will prove that for every s, such that $0 \le s < t-1$, the array

$$\overline{A^{(s)}} = \{\mu_s(a_0^{(s)}), \mu_s(a_1^{(s)}), \ldots \mu_s(a_{n-1}^{(s)})\}$$

is sorted.

When $s = 0$ the assertion follows from the fact that during iteration with number 0 ($k = 0$), $A^{(0)}$ is ordered so that first come all elements of the array which in their binary notation end in 0, followed by all elements of the array which in their binary notation end in 1.

We assume that for a certain natural number s, $0 \le s < t-2$ the array

$$\overline{A^{(s)}} = \{\mu_s(a_0^{(s)}), \mu_s(a_1^{(s)}), \ldots \mu_s(a_{n-1}^{(s)})\}$$

is sorted. But then analysing the work of the algorithm in $(s+1)$-th iteration, it is easy to see that the array A0, which is obtained from $\overline{A^{(s)}}$ taking in the same row only these elements of $A^{(s)}$ having 0 in bit with number $s+1$, is a sorted array. Analogously we prove that the array A1 is sorted and in

bit with number $s+1$ on each of its elements stands 1. Then the array

$$\overline{A^{(s+1)}} = \{\mu_{s+1}(a_0^{(s+1)}), \mu_{s+1}(a_1^{(s+1)}), \ldots \mu_{s+1}(a_{n-1}^{(s+1)})\},$$

which is obtained from the merger of the arrays A0 and A1 where the elements of A0 precede the elements of A1, is sorted. And with this we have proven the theorem.

Theorem 2:9 Algorithm 1 described with the help of programming language C++ works in time $O(n)$.

Proof. The assertion of the theorem follows from the fact that in function BitwiseSort1 we have only two nested loops. In the inner loop exactly n iterations are performed, and in every iteration once the operation & (bitwise conjunction), once the operation << (bitwise shift left), once the if statement, once the assignment statement and once the increment statement are performed. Each of the aforesaid operations is performed in constant time. The outer loop does $t-2$ iterations, where t is a constant, and in every iteration besides the inner loops there are also two assignment operations.

In function BitwiseSort the division of the array into two disjoing subarrays is performed apparently in time $O(n)$. The newly obtained two arrays are sorted in total time $O(n)$. The following merger of the two sorted arrays with total length n is apparently also performed in time $O(n)$.

REFERENCES

[1] J. Darlington, A Synthesis Of Several Sorting Algorithms. Acta Informatica, Volume 11, Number 1, December 18, 1978, 1-30.

[2] S. R. Davis, C++ For Dummies, IDG Books Worldwide, 2000.

[3] M. L. Hetlan, Python Algorithms: Mastering Basic Algorithms In The Python Language, Apress, 2010.

[4] D. Flanagan, JAVA In A Nutshell, O'Reilly, 2002.

[5] B. W. Kernigan, D. M. Ritchie, The C Programming Language, AT T Bell Laboratories, 1998.

[6] D. E. Knuth, The Art Of Computer Programming. Volume 3: Sorting And Searching, Second Edition, Addison-Wesley, 1998.

[7] H. Schildt, Java 2 A Beginner's Guide, Mcgraw-Hill, 2001.

[8] R. Sedgewick, Algorithms In C++, Parts 1-4 Fundamentals, Data Structures, Sorting, Searching, Third Edition, Addison Wesley Longman, 1999