INTEGER COMPUTATION PUZZLES AS PART OF THE HOBBY TIME TRAINING APPROACH

Assoc. Prof. Milen Loukantchevsky, PhD, IEEE & ACM Member
Department of Computer Systems & Technologies,
University of Ruse "Angel Kanchev"
Phone: 0877 303 850
e-mail: mil@ieee.org

Abstract: *The concept of Developer's point of view (DPV) learning approach is based on the idea of "perception the very solution to the problem as a game" and takes the gamification of learning to the next level.*

The Hobby Time Training (HTT) concept is part of the DPV learning approach. It assumes solving small, apparently simple problems, which encapsulates deeply hidden potential. The problem solving itself takes place during the students' free time and assumes unobtrusive guidance with as little as possible obligatory moments.

In previous papers the HTT is presented by distinctive bitwise operations. They contain the sought-after hidden creative potential, mainly due to the limited support both at the high and low levels. Here the HTT is developed by another kind of apparently "simple" problems of the area of integer computation. Somewhat unexpectedly for the unprepared one it turns out that these problems have a deeply hidden inner content. Like bitwise operations, integer computation algorithms suppose usage of some special techniques such word-level parallelism, unrolling loops, and branch elimination.

The attention is focused here on some elements of Hamming sequence generation: factorization by 2, 3 and 5, divisibility check by 3 and 5, fast integer division and multiplication by 3 and 5.

Keywords: *Constructivism, Factorization, Gamification, Hamming Numbers, Hobby Time, Number Sequences Games, Regular Numbers, x86/x64*

ASJC Codes: *1701, 1708, 1712*

INTRODUCTION

The concept of *Developer's point of view* (*DPV*) learning approach is described as "*perceive the very solution to the problem as a game*". That is, the decision-making process itself becomes a game. Moreover, it assumes the usage of conventional development environments. Thus, we go up to the next level of gamification in comparison to the more primitive perception of gamification as "*make the hard stuff fun*".

The *Hobby Time Training* (*HTT*) concept is a part of the *DPV* learning approach [7, 8]. As such, the *HTT* assumes solving of small, apparently simple problems, which encapsulates deeply hidden potential that could be found out only during the problem solving [1].

In previous papers the HTT is presented by distinctive bitwise operations [7, 8]. They contain the sought-after hidden creative potential, mainly due to the limited support both at the high and low levels. Here the *HTT* is developed by another kind of apparently "simple" problems of the area of integer computation. Somewhat unexpectedly for the unprepared one it turns out that these problems have a deeply hidden inner content. Like bitwise operations, integer computation algorithms suppose usage of some special techniques such *word-level parallelism*, *unrolling loops*, and *branch elimination*. Especially useful when the field of study is modern superscalar computer architectures.

The attention is focused here on some elements of *Hamming Numbers Sequence* (*HNS*) generation: factorization by *2*, *3* and *5*, divisibility check by *3* and *5*, fast integer division and multiplication by *3* and *5*.

The code presented is *Windows 32*-bit platform oriented, produced by the *Embarcadero C++ Builder® 11.3* development environment and its classic *BCC32* compiler [3]. This choice is optional and therefore not a limitation. In our case, it stems from the practice of the author as corporate developer and from the opportunity to use inline Assembly technology.

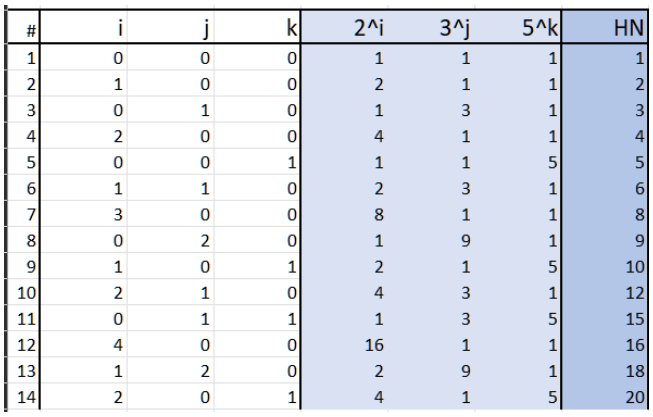
EXPOSITION

1. Problem Definition

So called *regular*, *5-smooth*, or *Hamming number* (*HN*) is product of three primes *2*, *3* and *5* (Eq. 1).

$$HN = 2^i \times 3^j \times 5^k, i \geq 0, j \geq 0, k \geq 0 \tag{1}$$

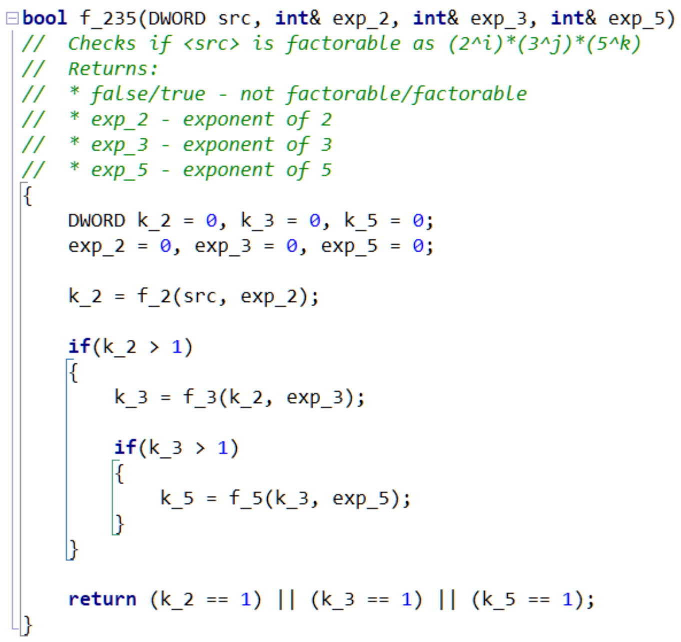
Respectively, the ascending sequence of *HN* is called *Hamming numbers sequence* (*HNS*) or simply *Hamming sequence*. Richard Hamming is the first asked for an efficient algorithm to generate the list, in ascending order, of all numbers of this kind. The problem was popularized by Edsger Dijkstra.



#	i	j	k	2^i	3^j	5^k	HN
1	0	0	0	1	1	1	1
2	1	0	0	2	1	1	2
3	0	1	0	1	3	1	3
4	2	0	0	4	1	1	4
5	0	0	1	1	1	5	5
6	1	1	0	2	3	1	6
7	3	0	0	8	1	1	8
8	0	2	0	1	9	1	9
9	1	0	1	2	1	5	10
10	2	1	0	4	3	1	12
11	0	1	1	1	3	5	15
12	4	0	0	16	1	1	16
13	1	2	0	2	9	1	18
14	2	0	1	4	1	5	20

Fig. 1. *HNS-14*

The problem is trivial only in its classical form: to generate *HNS* in ascending order by successively traversing and checking a given subset of the natural numbers. From Fig. 1 could be seen that powers of *2*, *3*, and *5* do not increase monotonically. Consequently, the problem becomes significantly more complicated if a full traversal is to be avoided.



```
bool f_235(DWORD src, int& exp_2, int& exp_3, int& exp_5)
//  Checks if <src> is factorable as (2^i)*(3^j)*(5^k)
//  Returns:
//  * false/true - not factorable/factorable
//  * exp_2 - exponent of 2
//  * exp_3 - exponent of 3
//  * exp_5 - exponent of 5
{
    DWORD k_2 = 0, k_3 = 0, k_5 = 0;
    exp_2 = 0, exp_3 = 0, exp_5 = 0;

    k_2 = f_2(src, exp_2);

    if(k_2 > 1)
    {
        k_3 = f_3(k_2, exp_3);

        if(k_3 > 1)
        {
            k_5 = f_5(k_3, exp_5);
        }
    }

    return (k_2 == 1) || (k_3 == 1) || (k_5 == 1);
}
```

Fig. 2. Function *f_235()*

Following the principles of the *HTT* we are interested (at least initially) in the individual independent subproblems into which the problem could be broken down. At the top level of the solution (Fig. 2) is placed the function *f_235()*. It attempts factorization of <*src*> as in Eq. 1.

The functions *f_2()*, *f_3()* and *f_5()* for factorizing to *2*, *3* and *5*, respectively, are placed a level down. While *f_2()* is solved elementary, *f_3()* and *f_5()* implies several alternative solutions:

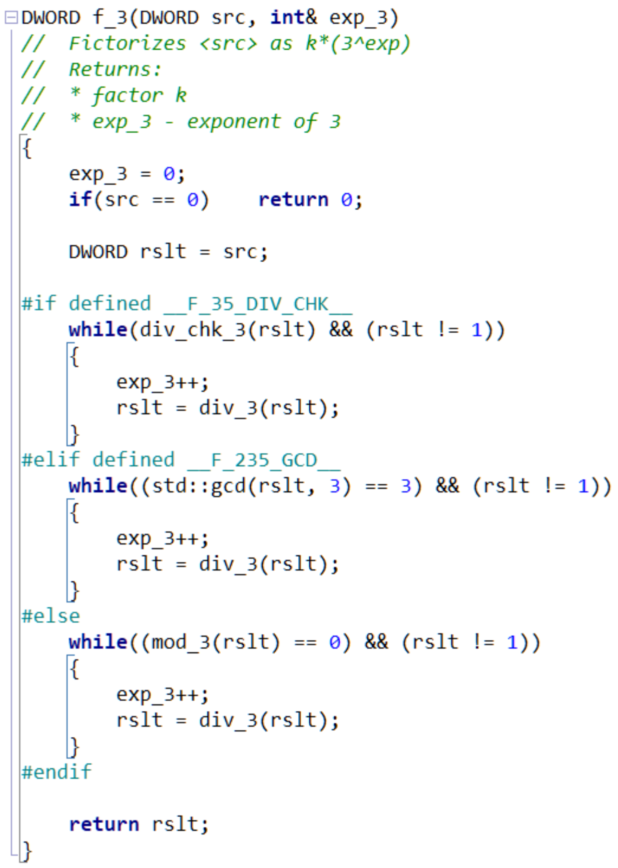
- By the modulo division operation.
- By *GCD*.
- By alternative method of divisibility verification.

The *goal* is to investigate different alternatives of implementation of *f_3()* and *f_5()*.

Let consider here in little more detail some key moments of factorization by *3*, divisibility check by *3* and modulus of *3*.

2. Factorizing by *3*

The function *f_3()* for factorizing to *3* has three main alternatives separated by the relevant conditional preprocessor directive (Fig. 3).



```
DWORD f_3(DWORD src, int& exp_3)
//  Fictorizes <src> as k*(3^exp)
//  Returns:
//  * factor k
//  * exp_3 - exponent of 3
{
    exp_3 = 0;
    if(src == 0)    return 0;

    DWORD rslt = src;

#if defined __F_35_DIV_CHK__
    while(div_chk_3(rslt) && (rslt != 1))
    {
        exp_3++;
        rslt = div_3(rslt);
    }
#elif defined __F_235_GCD__
    while((std::gcd(rslt, 3) == 3) && (rslt != 1))
    {
        exp_3++;
        rslt = div_3(rslt);
    }
#else
    while((mod_3(rslt) == 0) && (rslt != 1))
    {
        exp_3++;
        rslt = div_3(rslt);
    }
#endif

    return rslt;
}
```

Fig. 3. Function *f_3()*

The divisibility check by *3* function *div_chk_3()*, modulus of *3* function *mod_3()* and the standard library function *std::gcd()* are placed a level down.

3. Divisibility Check by *3*

The main idea here is to choose a method that does not require division. And if possible, to apply *branch elimination* technique, in view of the specifics of modern superscalar architectures.

Three main points can be distinguished:

1) Let the number checked is represented in numbering system with radix p. The rule of divisibility of the number $(p+1)$ requires that the difference $|\sigma_e - \sigma_o|$ between the sum σ_e of all even digits and the sum σ_o of all odd digits is divisible by $(p+1)$.
2) The sums are calculated by *POPCNT x86/x64* machine instruction [9], i.e. using appropriate hardware support.
3) After calculating the sum of even bits and the sum odd bits is checked if their difference $|\sigma_e - \sigma_o|$ is zero. If it is zero, the divisibility condition is met. Otherwise, is checked if the difference is divisible by $(p+1)$.
4) The divisibility check of the difference is tabular. The table used is compressed into a *DWORD* constant (Fig. 4). The so compressed table lookup is performed by right logical shift at C bits, where the binary code of C is $C_3C_2C_1C_0$.

C_3 C_2 C_1 C_0	1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1
0 0 0 0	1
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
. . .	. . .
1 1 0 1	0
1 1 1 0	0
1 1 1 1	1

Fig. 4. Compressed table of divisibility by *3*

Note that the method is directly applicable to check for divisibility by *3*, as internal machine representation uses radix $p = 2$ and hence $(p+1) = 3$.

The same method is also applicable to check divisibility by *5*. For that, the binary code of the number checked is formed in pairs of two bits representing a binary coded digit with radix *4*.

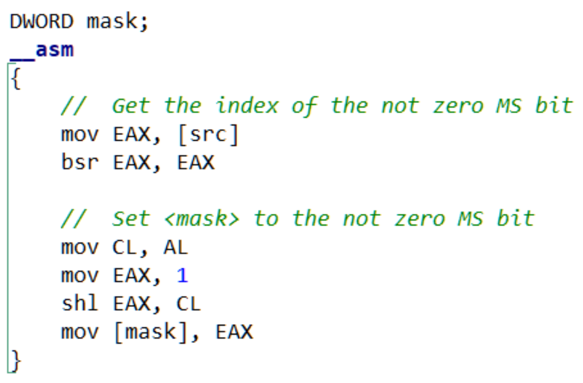
4. Modulus of *3*

The basic solution of the operation, of course, implies the use of the built-in high-level operator, e.g. *%* in *C/C++*. However, it is of interest to study an alternative avoiding division operation. Such is the *Deterministic Finite Automata* (*DFA*) method. The *State Transition Table* of the *DFA* for modulus of *3* is presented in Fig. *5*. S_i is the current state of the *DFA*, and S_{i+1} – its next state. The implementation of the *DFA* of Fig 5 implies scanning from *MSB* to *LSB*.

input	S_i	S_{i+1}
0	0	0
0	1	2
0	2	1
1	0	1
1	1	0
1	2	2

Fig. 5. State Transition Table of *DFA* for modulus of *3*

Initial Phase: Additional speedup is obtained if scanning begins from the MSB but from the most significant set bit using *BSR* machine instruction [9], as shown in Fig. 6.

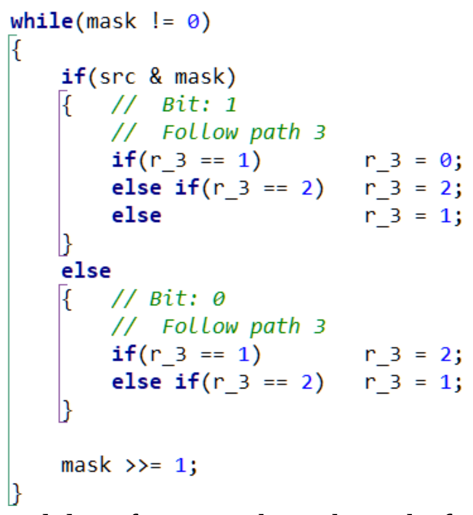


```
DWORD mask;
__asm
{
    //  Get the index of the not zero MS bit
    mov EAX, [src]
    bsr EAX, EAX

    //  Set <mask> to the not zero MS bit
    mov CL, AL
    mov EAX, 1
    shl EAX, CL
    mov [mask], EAX
}
```

Fig. 6. Modulus of *3* first phase

Three variants of the main phase are considered.
<u>Main Phase [Ver. 1]:</u> traditional, through *if/else* operator (Fig. 7).

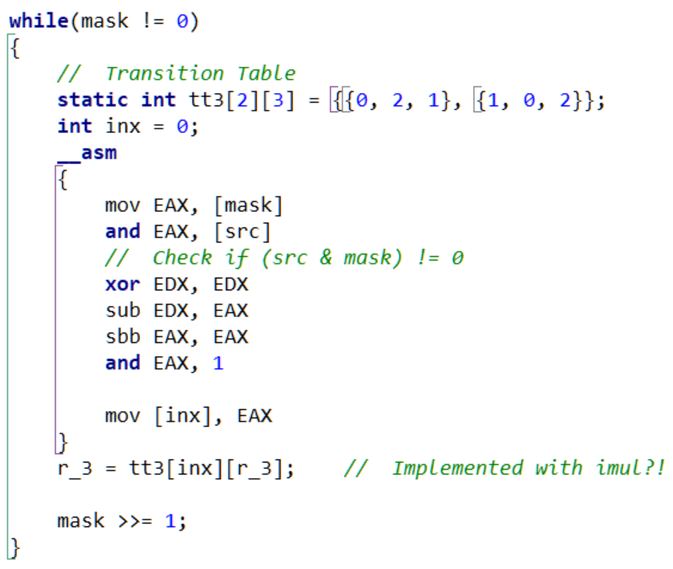


```
while(mask != 0)
{
    if(src & mask)
    {   //  Bit: 1
        //  Follow path 3
        if(r_3 == 1)         r_3 = 0;
        else if(r_3 == 2)    r_3 = 2;
        else                 r_3 = 1;
    }
    else
    {   // Bit: 0
        //  Follow path 3
        if(r_3 == 1)         r_3 = 2;
        else if(r_3 == 2)    r_3 = 1;
    }

    mask >>= 1;
}
```

Fig. 7. Modulus of *3* main phase through *if/else* operator

<u>Main Phase [Ver. 2/Subversion 1]:</u> through *2D* state transition table (Fig. 8). Implements *branch elimination* technique. Hower, it turns out that to access the *2D* table *tt3* the compiler generates machine instruction *IMUL* for integer multiplication [9]! As a result, the efficiency of Subversion 1 is reduced.

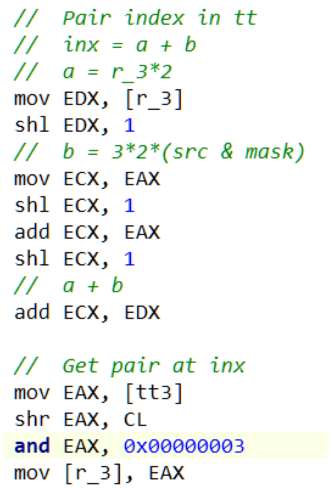


```
while(mask != 0)
{
    //  Transition Table
    static int tt3[2][3] = {{0, 2, 1}, {1, 0, 2}};
    int inx = 0;
    __asm
    {
        mov EAX, [mask]
        and EAX, [src]
        //  Check if (src & mask) != 0
        xor EDX, EDX
        sub EDX, EAX
        sbb EAX, EAX
        and EAX, 1

        mov [inx], EAX
    }
    r_3 = tt3[inx][r_3];    //  Implemented with imul?!

    mask >>= 1;
}
```

Fig. 8. Modulus of *3* main phase through *2D* state transition table



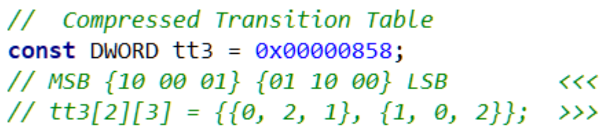
```
//  Pair index in tt
//  inx = a + b
//  a = r_3*2
mov EDX, [r_3]
shl EDX, 1
//  b = 3*2*(src & mask)
mov ECX, EAX
shl ECX, 1
add ECX, EAX
shl ECX, 1
//  a + b
add ECX, EDX

//  Get pair at inx
mov EAX, [tt3]
shr EAX, CL
and EAX, 0x00000003
mov [r_3], EAX
```

Fig. 9. Fast access to the compressed state transition table

Main Phase [Ver. 2/Subversion 2]: through compressed state transition table (Fig. 9).

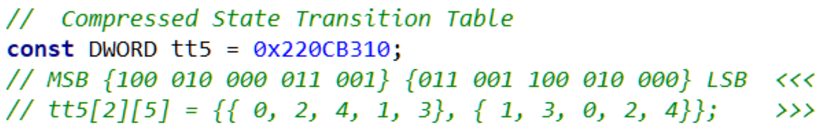
In this case, the access to the *2D* state transition table is reduced to one fast multiplication by *2* (for the first index *a*) and one fast multiplication by *3* (for the second index *b*).



```
//  Compressed Transition Table
const DWORD tt3 = 0x00000858;
// MSB {10 00 01} {01 10 00} LSB        <<<
// tt3[2][3] = {{0, 2, 1}, {1, 0, 2}};  >>>
```

Fig. 10. Compressed state transition table

The *2D* state transition table for Fig. 5 is compressed here to *DWORD* constant via *linearization* shown in Fig. 10.



```
//  Compressed State Transition Table
const DWORD tt5 = 0x220CB310;
// MSB {100 010 000 011 001} {011 001 100 010 000} LSB  <<<
// tt5[2][5] = {{ 0, 2, 4, 1, 3}, { 1, 3, 0, 2, 4}};    >>>
```

Fig. 11. Compressed state transition table for modulus of *5*

For comparison the appropriate compressed state transition table is presented in Fig. 11. In both cases the *Little Endian* multibyte order of x86/x64 family is taken into account.

CONCLUSION

Following the *HTT* and the principle "the simplest first" are given the problems of factorization by *2*, *3* and *5*, divisibility check by *3* and *5*, fast integer division and multiplication by *3* and *5*. They are products of division to subtasks of the more general problem of *Hamming Numbers Sequence* (*HNS*) generation.

Key moments of the factorization by *3*, divisibility check by *3* and modulus of *3* are considered in more detail. These operations in respect to 5 are based on the same ideas but are not "copy-paste" solvable. Their implementation requires a definite creativity from the students.

The full code is accessible in the *F_235* subfolder of the author's *GitHub* repository [6]. An alternative method for fast division by *3* and *5* is also shown there. It consists in replacing the operation of division with multiplication [10]. Would be very useful and instructive for the future specialists to use the code provided for evaluation and comparison of the considered alternatives.

For more advanced students who are successful, one can continue with different variants of the accelerated *HNS* generation problem: without a full traversal of a subset of the natural numbers. But this is already another topic and implies separate consideration.

REFERENCES

[1] Anderson, S. Bit Twiddling Hacks. Retrieved September 29, 2023 from http://graphics.stanford.edu/~seander/bithacks.html#ParityParallel

[2] Dijkstra, E. (1976). A Discipline of Programming. Prentice-Hall, Inc., NJ. ISBN 0-13-215871-X

[3] Embarcadero. RAD Studio Docwiki: C++ Compilers. Retrieved September 29, 2023 from https://docwiki.embarcadero.com/RADStudio/Alexandria/en/C%2B%2B_Compilers

[4] Intel. (2023). Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-048

[5] Intel. (2023). Intel® 64 and IA-32 Architectures Software Developer's Manual. Order Number: 325462-081US

[6] Loukantchevsky, M. GitHub Repository: Hamming Numbers Generation. Retrieved October 12, 2023 from https://github.com/milphaser/HN

[7] Loukantchevsky, M. (2022). The Hobby Time Training Approach. In: Proceedings of the University of Ruse - 2022, vol 61, book 3.2, ISSN 2603-4123

[8] Loukantchevsky, M. (2023). Yet Another Parallelism Within the "Hobby Time Training". In: Yang, XS., Sherratt, R.S., Dey, N., Joshi, A. (eds) Proceedings of Eighth International Congress on Information and Communication Technology. ICICT 2023. Lecture Notes in Networks and Systems, vol 694. Springer, Singapore. https://doi.org/10.1007/978-981-99-3091-3_19

[9] Microsoft technical documentation. x86 Instructions. Retrieved September 29, 2023 from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-instructions

[10] Warren, H. (2012). Hacker's Delight. 2nd Ed. Addison-Wesley Professional. ISBN 978-0321842688