On the Conversion Between Number Systems

Houssain Kettani

***Abstract*—This brief revisits the problem of conversion between number systems and asks the following question: given a nonnegative decimal number d, what is the value of the digit at position j in the corresponding base b number? Thus, we do not require the knowledge of other digits except the one we are interested in. Accordingly, we present a conversion function that relates each digit in a base b system to the decimal value that is equal to the base b number in question. We also show some applications of this new algorithm in the areas of parallel computing and cryptography.**

***Index Terms*—Algorithm's efficiency, base b numbers, conversion.**

I. Introduction and Preliminaries

A number system is a language for representing numbers. In this language, an unsigned b-ary, radix b, or base b number is represented as the following string of digits:

$$(d_n d_{n-1} \dots d_0 . d_{-1} d_{-2} \dots d_{-m+1} d_{-m})_b$$

where the integer $b \geq 2$, $d_j \in \{0, 1, \dots, b-1\}$ for $j = n, n-1, \dots, -m+1, -m$, and "." is referred to as the radix point. The d_j's are referred to as the digits of the base b. Accordingly, the term to the left of the radix point is referred to as the integer part, while that to the right of the radix point is referred to as the fraction part.

To illustrate the use of number systems, the everyday number system that we use is the decimal (base 10). Other number systems that are used in the computer work are binary (base 2), octal (base 8), and hexadecimal (base 16).

To understand the meaning of a number in this representation, it would be of interest to develop a mechanism by which we can go back and forth between different bases. To this end, the classical way to convert a number in base b_1 to another number in base b_2, so that both have the same decimal value, is to first convert base b_1 to decimal then convert from the latter to base b_2—see [3] for example.

A more convenient conversion method is adopted when $b_2 = b_1^p$ where p is a positive decimal integer. Consequently, to convert from base b_2 to base b_1, we gather p base b_1 digits (from right to left for the integer part, and from left to right for the fraction part) into one base b_2 digit so that both have the same decimal value. On the other hand, to convert from base b_1 to base b_2, we expand one base b_2 digit into p base b_1 digits so that both have the same decimal value. In other words, the relation between base b and base b^p is given by

$$(\dots d_1 d_0 . d_{-1} \dots)_b = \left(\dots d'_1 d'_0 . d'_{-1} \dots\right)_{b^p}$$

where

$$d'_j = (d_{pj+p-1} \dots d_{pj+1} d_{pj})_b = \sum_{i=0}^{p-1} d_{pj+i} b^i.$$

To illustrate this point, consider conversion between binary and hexadecimal. In this case we have $b_1 = 2$ and $p = 4$. As an illustrative numerical example, we have $(00101000)_2 = (28)_{16}$.

Suppose now we would like to write

$$(d)_{10} = (d_n d_{n-1} \dots d_0 . d_{-1} d_{-2} \dots d_{-m+1} d_{-m})_b$$

where the right hand side is a decimal number and the left-hand side is a base b number. Then, by definition, to convert from base b to decimal, we write

$$d = \sum_{i=-m}^{n} d_i b^i. \tag{1}$$

Alternatively, the classical way to convert from decimal to base b, is to repeatedly divide the integer part of d by b and record the remainders. Then, starting with the first recorded one, we write down these remainders from right to left starting to the left of the radix point. Thus, we obtain the integer part of the corresponding base b number. This algorithm is referred to as repeated division.

On the other hand, the fraction part is obtained as follows. We repeatedly multiply the fraction part of d by b and record the resultant integer. Then, starting with the first recorded one, we write down these integers from left to right starting to the right of the radix point. Thus, we obtain the fraction part of the corresponding base b number. This algorithm is referred to as repeated multiplication. Illustrative examples of both algorithms can be found at [3, pp. 8–11]. A nice history on the other hand, and background about number systems are laid out in [2, pp. 194–213].

To this end, a compelling question that is addressed in this brief is the following: is there any simple function that we can use to convert from decimal to base b? Moreover, what if we are only interested in the value of d_j for some $j \geq 0$? Then based on the repeated division algorithm, we need to compute all the values d_k for $k = 0, 1, \dots, j$. Also, what if we are only interested in the value of d_j for some $j < 0$? Then based on the repeated multiplication algorithm, we need to compute all the values d_k for $k = -1, -2, \dots, j$.

Section II answers the above questions and presents a very simple and useful result. Section III presents a comparison between the new method and the classical method to find the digit in position j. Section IV points out an application of the introduced method in the areas of parallel computing, cryptography

Manuscript received March 16, 2006. This paper was presented in part at the 2004 International Conference on Algorithmic Mathematics and Computer Science (AMCS'04) Las Vegas, Nevada June 2004. This paper was recommended by Associate Editor L. Lavagno.

The author is with the Department of Computer Science, Jackson State University, Jackson, MS 39217 USA (e-mail: houssain.kettani@jsums.edu).

Digital Object Identifier 10.1109/TCSII.2006.882856

and steganography. We end the brief by concluding remarks in Section V.

II. Functional Conversion

The questions posed in the previous section are answered in the following theorem which presents a very simple and useful result. By this theorem, the d_j's are directly accessible without the requirement of computing any other d_j except the one of interest. In this theorem, we use $\lfloor \cdot \rfloor$ to denote the floor function.

A. Theorem

Let d be a positive real number, and let $b \geq 2$ be an integer. Let d, written in base b, be given by

$$d = (d_n d_{n-1} \dots d_0 . d_{-1} \dots d_{-m+1} d_{-m})_b \tag{2}$$

where n and m are positive integers and, for $-m \leq j \leq n$, the digits d_j of the expansion belong to the set $\{0, 1, \dots, b-1\}$. Then, for each j

$$d_j = \lfloor db^{-j} \rfloor - b \lfloor db^{-j-1} \rfloor. \tag{3}$$

B. Proof of Theorem

From the expansion in (1), we have

$$db^{-j} = \sum_{i=j}^{n} d_i b^{i-j} + \sum_{i=-m}^{j-1} d_i b^{i-j}.$$

Next note that

$$0 \leq \sum_{i=-m}^{j-1} d_i b^{i-j} < 1. \tag{4}$$

To show this, we first note that the left-hand side of the inequality is straightforward since each term in the summation in nonnegative. The right-hand side, on the other hand, is obtained as follows:

$$\begin{aligned} \sum_{i=-m}^{j-1} d_i b^{i-j} &\leq \sum_{i=-m}^{j-1} (b-1) b^{i-j} \\ &= \sum_{i=-m}^{j-1} b^{i-j+1} - \sum_{i=-m}^{j-1} b^{i-j} \\ &= 1 - b^{-m-j} \\ &< 1. \end{aligned}$$

Thus, it follows that

$$\lfloor db^{-j} \rfloor = \sum_{i=j}^{n} d_i b^{i-j}. \tag{5}$$

Following a similar argument, we have

$$\lfloor db^{-j-1} \rfloor = \sum_{i=j+1}^{n} d_i b^{i-j-1}$$

so that, on multiplying by b, we have

$$b \lfloor db^{-j-1} \rfloor = \sum_{i=j+1}^{n} d_i b^{i-j} \tag{6}$$

and subtracting (6) from (5) gives the result of the theorem.

C. Remarks

The minimum value of index n for which (2) holds can easily be expressed from (1) and is given by

$$n_{\min} = \begin{cases} \lfloor \log_b d \rfloor, & \text{if } d \geq 1 \\ 0, & \text{otherwise.} \end{cases}$$

The minimum value of index m, on the other hand, is not as elegantly expressible. In fact, m may diverge to infinity. For example, consider the decimal number 0.1, then we have

$$(0.1)_{10} = (0.0001100110011 \dots)_2.$$

In this case, although the decimal fraction part has finite number of digits, the corresponding binary fraction part does not have a finite number of bits. In some cases, however, we can write a general formula for the d_j's. To illustrate this point, in the previous example, we can write

$$d_j = \begin{cases} 1, & \text{for } j = -4k, -4k-1; k = 1, 2, \dots \\ 0, & \text{otherwise.} \end{cases}$$

For practical purpose, and as done in the decimal case, we tend to truncate the number. In other words, we stop at a fixed m to approximate the representation of the number.

We end this section by noting that there are other equivalent expressions to express the d_j's in (3). In fact, it can easily be shown that the values of the d_j's in (3) are also equal to $\lfloor b(db^{-j-1} - \lfloor db^{-j-1} \rfloor) \rfloor$, and $\lfloor b^{-j} \mathrm{mod}(d, b^{j+1}) \rfloor$. This follows from the properties of the modulus and floor functions. The latter expression follows by using the following expression for the modulus function for real numbers a and $b \neq 0$, (see [1, p. 39])

$$\mathrm{mod}(a, b) = a - b \lfloor a/b \rfloor. \tag{7}$$

III. Efficiency Comparison

Let us now compare the efficiency of the classical algorithm to that of the new algorithm introduced in Theorem II-A. Let us first remind ourselves of what the problem is. Given a nonnegative decimal number d, we would like to find the value of the digit at position j in the corresponding base b number.

For the sake of clarity, a brute-force implementation of the classical algorithm to solve the aforementioned problem is named here $ClassicConvert$ and it stems from [2, p. 319] together with the use of (7). The new algorithm, on the other hand, which can be viewed as a divide-and-conquer algorithm for base conversion, and is based on Theorem II-A, is named here $NewConvert$.

Accordingly, the classical algorithm consists of two subalgorithms. The algorithm first checks whether the position j is nonnegative. If the latter is the case, then the algorithm picks the integer part of the decimal number d and calls the subalgorithm $ClassicConvertInt$ which applies the repeated division algorithm discussed in Section I to this integer part. On the other hand, if j is negative, the algorithm picks the fraction

part of d and calls the subalgorithm $ClassicConvertFrac$ which applies the repeated multiplication algorithm described in Section I to this fraction part. Thus, the algorithm $ClassicConvert$ can be expressed as follows:

Algorithm: $d_j = ClassicConvert(d, b, j)$

// $d \geq 0$ is a decimal number.

// $b \geq 2$ is a decimal integer.

// j is an integer.

if $j \geq 0$

    $d = \lfloor d \rfloor$ // extracts the integer part

    $d_j = ClassicConvertInt(d, b, j)$

else

    $d = d - \lfloor d \rfloor$ // extracts the fraction part

    $d_j = ClassicConvertFrac(d, b, j)$

end

return d_j

where the subalgorithm $ClassicConvertInt$ is expressed as follows:

Algorithm: $d_j = ClassicConvertInt(d, b, j)$

// $d \geq 0$ is a decimal number.

// $b \geq 2$ is a decimal integer.

// $j \geq 0$ is an integer.

$c = d$

$d_0 = c - b\lfloor c/b \rfloor$

while $j \neq 0$ do

    for $i = 1 \ldots j$ do

        $c = \lfloor c/b \rfloor$

        $d_i = c - b\lfloor c/b \rfloor$

    end

end

return d_j

and the subalgorithm $ClassicConvertFrac$ is expressed as follows:

Algorithm: $d_j = ClassicConvertFrac(d, b, j)$

// $d \geq 0$ is a decimal number.

// $b \geq 2$ is a decimal integer.

// $j < 0$ is an integer.

$c = db$

$d_{-1} = \lfloor c \rfloor$

while $j \neq -1$ do

    for $i = -2 \ldots j$ do

        $c = (c - \lfloor c \rfloor)b$

        $d_i = \lfloor c \rfloor$

    end

end

return d_j

The new algorithm, on the other hand, which can be viewed as a divide-and-conquer algorithm for base conversion, and is based on Theorem II-A, is expressed as follows:

Algorithm: $d_j = NewConvert(d, b, j)$

// $d \geq 0$ is a decimal number.

// $b \geq 2$ is a decimal integer.

// j is an integer.

$c = db^{-j}$

$d_j = \lfloor c \rfloor - b\lfloor c/b \rfloor$

return d_j

Thus, when $j \geq 0$, a brute-force implementation of the classical algorithm which is named here $ClassicConvert$, requires $1 + \sum_{i=1}^{j} 1 = 1 + j$ additions, $2 + \sum_{i=1}^{j} 3 = 2 + 3j$ multiplications, and $2 + \sum_{i=1}^{j} 2 = 2 + 2j$ applications of the floor function. Similarly, when $j < 0$, this algorithm requires $-j$ additions, $-j$ multiplications, and $-2j$ applications of the floor function. Accordingly, let $A_c(j)$, $M_c(j)$, and $F_c(j)$ denote the number of additions, multiplications, and floors, respectively, when using the classical algorithm. Then we have the following:

- $A_c(j) = -ju(-j) + (1 + j)u(j)$;
- $M_c(j) = -ju(-j) + (2 + 3j)u(j)$;
- $F_c(j) = -2ju(-j) + (2 + 2j)u(j)$;

where $u(j)$ denotes the step function defined as

$$u(j) = \begin{cases} 0 & \text{if } j < 0 \\ 1 & \text{if } j \geq 0. \end{cases}$$

The new algorithm, on the other hand, which is named here $NewConvert$ requires one addition, $3 + |j|$ multiplications, and two applications of the floor function. Accordingly, let $A_{\text{new}}(j)$, $M_{\text{new}}(j)$, and $F_{\text{new}}(j)$ denote the number of additions, multiplications, and floors, respectively, when using the new algorithm. Then, we have the following:

- $A_{\text{new}}(j) = 1$;
- $M_{\text{new}}(j) = 3 + |j|$;
- $F_{\text{new}}(j) = 2$.

Thus, both algorithms are linear in multiplication, with the proposed one requiring about one third multiplications as that of the classical algorithm when $j \geq 0$. In addition, the classical algorithm remains linear in addition and floor while the new algorithm is constant. This in turn shows that the new algorithm is more efficient and consequently faster than the classical algorithm.

IV. Suggested Applications

We note that the result of this brief finds its applications in many areas of research. For example, consider the problem of converting a very large decimal number to binary or another base. Then, we can use the method suggested in this brief together with parallel computing to optimize with respect to time and space. In other words, we can compute each bit or set of bits independently of the others by sending them to different processors at the same time. Then the final result is constructed from the results of the subproblems. The latter is a nice example of a divide-and-conquer algorithm.

To illustrate this point, suppose we want to convert a nonnegative decimal number d to $(d_{n_1}d_{n_1-1}\dots d_0.d_{-1}\dots d_{-m_1+1}d_{-m_1})_b$. Thus, from the analysis in Section III, the classical algorithm will require the following:

- $A'_c(n_1,-m_1)=A_c(n_1)+A_c(-m_1)$;
- $M'_c(n_1,-m_1)=M_c(n_1)+M_c(-m_1)$;
- $F'_c(n_1,-m_1)=F_c(n_1)+F_c(-m_1)$.

Consequently, the classical algorithm will require the following:

- $A'_c(n_1,-m_1)=1+n_1+m_1$;
- $M'_c(n_1,-m_1)=2+3n_1+m_1$;
- $F'_c(n_1,-m_1)=2+2n_1+2m_1$.

An alternative approach using the new method is to send a group of n_2 digits to different processors p at the same time, where $-p_-\le p\le p_+$ and $p_++1=\lceil n_1/n_2\rceil$ is the number of processors allocated to convert the integer part and $p_-=\lceil m_1/n_2\rceil$ is the number of processors allocated to convert the fraction part. To illustrate this point, the integer part will be partitioned as follows:

$$\underbrace{d_{(p_++1)n_2-1}\dots d_{p_+n_2}}_{p=p_+}\cdots\underbrace{d_{2n_2-1}\dots d_{n_2}}_{p=1}\underbrace{d_{n_2-1}\dots d_0}_{p=0}$$

while the fraction part is partitioned as follows:

$$0.\underbrace{d_{-1}\dots d_{-n_2}}_{p=-1}\underbrace{d_{-n_2-1}\dots d_{-2n_2}}_{p=-2}\cdots\underbrace{d_{(-p_-+1)n_2-1}\dots d_{-p_-n_2}}_{p=-p_-}.$$

In this case, the total number of additions and floors required to calculate this group of n_2 digits is $A'_{\text{new}}(n_2)=n_2$, and $F'_{\text{new}}(n_2)=2n_2$, respectively. The number of multiplications on the other hand is

$$M'_{\text{new}}(n_2)=\max_p\sum_{j=pn_2}^{(p+1)n_2-1}M_{\text{new}}(j)=3n_2+\max_p\sum_{j=pn_2}^{(p+1)n_2-1}|j|.$$

The maximum $M'_{\text{new}}(n_2)$ occurs at either $p=p_+$ or $p=p_-$. For simplicity, we use the following approximation: $p_+\approx n_1/n_2$ for $n_2\ll n_1$ and $p_-\approx m_1/n_2$ for $n_2\ll m_1$. Consequently, we obtain

$$M'_{\text{new}}(n_2)\approx n_2\max\{n_1+2.5,m_1+3.5\}-n_2^2/2.$$

Hence, while the efficiency of the new divide-and-conquer conversion algorithm is better than the classical one in addition and floor, the classical algorithm is better in multiplication. However, the suggested application comes handy in case if memory is an issue.

We end this section by suggesting another use of the result introduced in this brief in the areas of cryptography and steganography. The former is the fact of hiding a message while the latter is the fact of hiding the existence of a message. The application of the introduced result to these areas can be as follows. The sender and receiver can agree before hand on the base and positions of digits that carry the real information. Thus, for example, the sender can flood a channel with ones and zeros. While other users think of this as noise, the receiver will know what to do with the hidden information.

V. Concluding Remarks

We have discussed conversion between any two number systems. Unlike the classical conversion from decimal to base b system, we have presented a functional conversion that expresses each digit in base b in terms of the decimal number in question. This in turn, provides us with a direct access to each base b digit instead of the need of knowing any previous base b digits. Thus, we do not require the knowledge of other digits except the one we are interested in. We showed how this new algorithm is used in parallel computing to convert large numbers. We also presented a use of the new algorithm in the areas of cryptography and steganography.

Throughout this brief, we worked with unsigned numbers. However, the use of this result for signed numbers should be straightforward. This is the case since we can go back and forth between signed and unsigned numbers depending on how the sign is represented: signed-magnitude, signed-1's complement, signed-2's complement, etc—See [3, pp. 132–143] for details.

A general version of number systems suggests that b can be any nonzero number (real or imaginary) and that the d_j's can be chosen from any specific set of numbers so that any number is uniquely expressible in this base. In such case, very interesting results and properties can arise. For more information and historical prospective on such general case see [2, pp. 195–213]. Thus, it would be of interest to generalize the result of this brief to the case when $b\in\mathbb{C}^*$ and d_j's are chosen from any appropriately defined set.

In such general case, inequality (4) in the proof of Theorem II-A may no longer hold true and therefore the theorem does not generally hold true in the aforementioned general case. To provide a counter example, consider converting the number 10 from decimal to negabinary (base -2 with digits 0 and 1). Accordingly, we have $10=(11110)_{-2}$. Thus, we have for example, $d_2=1$. However, applying Theorem II-A naively, results in the wrong value of $d_2=-2$.

Acknowledgment

The author thanks Prof. A. Chan, Fayetteville State University, NC, for various comments on the efficiency of the new method. He also thanks the reviewers for their constructive comments and valuable suggestions during the revision process.

References

[1] E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.
[2] ——, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1998.
[3] M. M. Mano and C. R. Kime, *Logic and Computer Design Fundamentals*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2001.