

Quickly Converting Decimal to Binary

Connor Baker

January 2017
Version 0.2b

Abstract

This paper briefly discusses ways to convert decimal numbers to their binary representation using several fast algorithms which exploit powers of two as a means of performing less arithmetic operations, with the caveat of using a look up table.

Contents

1	From Decimal to Binary	1
2	Algorithms	2
2.1	A Simple Approach	2
2.2	Exploiting Larger Powers of Two	2
2.2.1	The 2^2 Algorithm	3
2.2.2	The 2^3 Algorithm	3
2.2.3	The 2^4 Algorithm	4
2.2.4	The 2^n Algorithm	4
3	Examples	6
3.1	The 2^1 Algorithm	6
3.2	The 2^2 Algorithm	6
3.3	The 2^3 Algorithm	7
3.4	The 2^4 Algorithm	7

1 From Decimal to Binary

Given a number ABCDE in any base β , one can rewrite that number in what is called exponential form: that is, the sum of each digit multiplied by the base to a power. For our example $ABCDE_\beta$ (the notation means a number ABCDE in base β), we would write it as follows: $A \times \beta^4 + B \times \beta^3 + C \times \beta^2 + D \times \beta^1 + E \times \beta^0$. As an example, given the number 44675_{10} , we can rewrite it as $4 \times 10^4 + 4 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 5 \times 10^0$.

In terms of changing the base of a number, the goal is to rewrite it as the sum of some numbers multiplied by the desired base to a power. Take for example the number 7_{10} . We know that we can rewrite it as 7×10^0 , but how could we express this number in base two? Our goal is to end up with something that looks like $a_1 \times 2^n + a_2 \times 2^{n-1} + \dots + a_b \times 2^0$. The question then becomes, how can we find those weights $\{a_1, a_2, \dots, a_b\}$ that make the sum seven? The answer is division.

In the table below, we perform the division necessary to calculate the binary representation of seven in base two. Each time we divide the decimal by two and get a remainder of one, we put a one in the rightmost place of our decimal representation. Likewise, if we have a remainder of zero, we append a zero. Once the value of the quotient is zero, we halt computation because we have found the binary representation in full.

Quotient	Remainder	Push to Stack	Binary Representation
7			
3	1	'1'	1
1	1	'1'	11
0	1	'1'	111

In the first row, we have the original decimal, an empty remainder since we have not begun to divide yet, and an empty binary representation. The second row has the whole number portion of the quotient, the remainder, and the beginnings of our binary representation. The third and fourth rows continue filling out binary representation. The fifth row is where we meet the halt condition, which is why the binary representation is unchanged.

So now that we know our binary representation is 111_2 , we can say that $7_{10} = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 111_2$.

In the next section, we discuss several different algorithms that we can use to help speed up this computation, which quickly becomes unwieldy for arbitrarily large numbers.

2 Algorithms

2.1 A Simple Approach

This is the algorithm we used in Section 1 for converting 7_{10} to binary. To convert a decimal number to binary, we divide each resultant quotient by two repeatedly until we meet the halt condition (zero as a quotient), and then take the remainders at each step in reverse.

In terms of an analog to computer science, we can think of keeping track of the remainders as adding each new one to the top of a stack, and at the end, pulling from the top of the stack (which reads all entries in reverse). This means that the remainder of the last division operation is going to be the most significant bit in the binary representation.

Remainder	Push to Stack
0	'0'
1	'1'

Table 1: The 2^1 Algorithm

We should be dissatisfied with the speed of this algorithm. For smaller numbers, this algorithm is fine. But what if we needed to compute the binary representation of a larger number, like 37_{10} or 51_{10} (both examples we have done in class)?

Our goal then is to find a faster algorithm. The easiest way to start is by speculating. One thing that we could examine is what happens when we divide by a number that is larger than two – perhaps a power of two.

2.2 Exploiting Larger Powers of Two

Given any two numbers $n, p \in \mathbb{N}$, $n \bmod p$ has a range of remainders (all in the set of natural numbers) of $[0, p - 1]$. With the algorithm we have above, we see that we account for both possibilities of the remainder. If we divide by a larger power of two, we now have a different number of possible remainders.

One caveat of dividing by larger powers of two is that the remainders are larger, and we still have to convert them to binary. When dividing by two, the only remainders we get are zero and one, which in a way, are already in binary for us. When we divide by larger numbers like four, or eight, we must be comfortable with quickly converting the range of number $[0, 2^{n-1}]$ (where n is the power of two we choose to divide by) to binary by hand.

For a machine, this is no problem: the trade-off between number of arithmetic operations vs. using a look up table is almost always a bargain to be made.

2.2.1 The 2^2 Algorithm

Per our range of possible remainders given a number p , if we decide to divide by four, our range of remainders is now $[0, 3]$. As such, we have to take into account these possibilities.

Interestingly enough, we must increase the number of bits that we push to the stack at once. The numbers zero and one can be represented in binary with a single bit – but the numbers two and three require two bits. In general, if we divide by 2^n , each remainder's binary representation must use n place values (so for 2^2 , we must use two place values to represent every number).

Remainder	Push to Stack
0	'00'
1	'01'
2	'10'
3	'11'

Table 2: The 2^2 Algorithm

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^1 algorithm.

2.2.2 The 2^3 Algorithm

Per our range of possible remainders given a number p , if we decide to divide by eight, our range of remainders is now $[0, 7]$. As such, we have to take into account these possibilities.

Here, since we are dividing by 2^3 , we must use three decimal places for *all* binary representations of the remainders.

Remainder	Push to Stack
0	'000'
1	'001'
2	'010'
3	'011'
4	'100'
5	'101'
6	'110'
7	'111'

Table 3: The 2^3 Algorithm

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^2 algorithm.

2.2.3 The 2^4 Algorithm

Per our range of possible remainders given a number p , if we decide to divide by eight, our range of remainders is now $[0, 15]$. As such, we have to take into account these possibilities.

Here, since we are dividing by 2^4 , we must use four decimal places for *all* binary representations of the remainders.

Remainder	Push to Stack
0	'0000'
1	'0001'
2	'0010'
3	'0011'
4	'0100'
5	'0101'
6	'0110'
7	'0111'
8	'1000'
9	'1001'
10	'1010'
11	'1011'
12	'1100'
13	'1101'
14	'1110'
15	'1111'

Table 4: The 2^4 Algorithm

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^3 algorithm.

2.2.4 The 2^n Algorithm

Per our range of possible remainders given a number p , if we decide to divide by 2^n , our range of remainders is now $[0, n - 1]$. As such, we have to take into account these possibilities.

Here, since we are dividing by 2^n , we must use n decimal places for *all* binary representations of the remainders.

Remainder	Push to Stack
0	'0000...00'
1	'0000...01'
\vdots	\vdots
\vdots	\vdots
\vdots	\vdots
$n-1$	'1111...10'
n	'1111...11'

Table 5: The 2^n Algorithm

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^{n-1} algorithm.

3 Examples

In this section, we will be converting the numbers 37_{10} , 55_{10} , and 128_{10} to binary using each of the four algorithms covered in this paper to show the difference in speed.

3.1 The 2^1 Algorithm

Quotient	Remainder	Push to Stack	Binary Representation
37			
18	1	'1'	1
9	0	'0'	01
4	1	'1'	101
2	0	'0'	0101
1	0	'0'	00101
0	1	'1'	100101

Quotient	Remainder	Push to Stack	Binary Representation
55			
27	1	'1'	1
13	1	'1'	11
6	1	'1'	111
3	0	'0'	0111
1	1	'1'	10111
0	1	'1'	110111

Quotient	Remainder	Push to Stack	Binary Representation
128			
64	0	'0'	0
32	0	'0'	00
16	0	'0'	000
8	0	'0'	0000
4	0	'0'	00000
2	0	'0'	000000
1	0	'0'	0000000
0	1	'1'	10000000

3.2 The 2^2 Algorithm

Quotient	Remainder	Push to Stack	Binary Representation
37			
9	1	'01'	01
2	1	'01'	0101
0	2	'11'	100101

Quotient	Remainder	Push to Stack	Binary Representation
55			
13	3	'11'	11
3	1	'01'	0111
0	3	'11'	110111

Quotient	Remainder	Push to Stack	Binary Representation
128			
32	0	'00'	00
8	0	'00'	0000
2	0	'00'	000000
0	1	'01'	10000000

3.3 The 2^3 Algorithm

Quotient	Remainder	Push to Stack	Binary Representation
37			
4	5	'101'	101
0	4	'100'	100101

Quotient	Remainder	Push to Stack	Binary Representation
55			
6	7	'111'	111
0	6	'110'	110111

Quotient	Remainder	Push to Stack	Binary Representation
128			
16	0	'000'	000
2	0	'000'	000000
0	2	'010'	010000000

We notice here that we have a leading zero. As it holds no value, it's fine to discard from the final representation. This is a result of using a power that is larger than two, since we have more than a single bit.

3.4 The 2^4 Algorithm

Quotient	Remainder	Push to Stack	Binary Representation
37			
2	5	'0101'	0101
0	2	'0010'	00100101

Quotient	Remainder	Push to Stack	Binary Representation
55			
3	7	'0111'	0111
0	3	'0011'	00110111

Quotient	Remainder	Push to Stack	Binary Representation
128			
8	0	'0000'	0000
0	8	'1000'	10000000