Hamming weight

The **Hamming weight** of a string is the number of symbols that are different from the zero-symbol of the alphabet used. It is thus equivalent to the Hamming distance from the all-zero string of the same length. For the most typical case, a string of bits, this is the number of 1's in the string. In this binary case, it is also called the **population count**, **popcount**, or **sideways sum**.[1] It is the digit sum of the binary representation of a given number and the ℓ_1 norm of a bit vector.

History and usage

Examples

string	Hamming weight
11101	4
11101000	4
00000000	0
789012340567	10

The Hamming weight is named after Richard Hamming although he did not originate the notion.[2] Irving S. Reed introduced a concept, equivalent to Hamming weight in the binary case, in 1954.[3]

Hamming weight is used in several disciplines including information theory, coding theory, and cryptography. Examples of applications of the Hamming weight include:

- In modular exponentiation by squaring, the number of modular multiplications required for an exponent e is $\log_2 e$ + weight(e). This is the reason that the public key value e used in RSA is typically chosen to be a number of low Hamming weight.
- The Hamming weight determines path lengths between nodes in Chord distributed hash tables.[4]
- IrisCode lookups in biometric databases are typically implemented by calculating the Hamming distance to each stored record.
- In computer chess programs using a bitboard representation, the Hamming weight of a bitboard gives the number of pieces of a given type remaining in the game, or the number of squares of the board controlled by one player's pieces, and is therefore an important contributing term to the value of a position.
- Hamming weight can be used to efficiently compute find first set using the identity ffs(x) = pop(x ^ (~(-x))). This is useful on platforms such as SPARC that have hardware Hamming weight instructions but no hardware find first set instruction.[5]
- The Hamming weight operation can be interpreted as a conversion from the unary numeral system to binary numbers.[6]
- In implementation of some succinct data structures like bit vectors and wavelet

trees.

Efficient implementation

The population count of a bitstring is often needed in cryptography and other applications. The Hamming distance of two words *A* and *B* can be calculated as the Hamming weight of *A* xor *B*.

The problem of how to implement it efficiently has been widely studied. Some processors have a single command to calculate it (see below), and some have parallel operations on bit vectors. For processors lacking those features, the best solutions known are based on adding counts in a tree pattern. For example, to count the number of 1 bits in the 16-bit binary number a = 0110 1100 1011 1010, these operations can be done:

Expression	Binary	Decimal	Comment
`a`	01 10 11 00 10 11 10 10		The original number
`b0 = (a >> 0) & 01 01 01 01 01 01 01 01`	01 00 01 00 00 01 00 00	1,0,1,0,0,1,0,0	every other bit from a
`b1 = (a >> 1) & 01 01 01 01 01 01 01 01`	00 01 01 00 01 01 01 01	0,1,1,0,1,1,1,1	the remaining bits from a
`c = b0 + b1`	01 01 10 00 01 10 01 01	1,1,2,0,1,2,1,1	list giving # of 1s in each 2-bit slice of a
`d0 = (c >> 0) & 0011 0011 0011 0011`	0001 0000 0010 0001	1,0,2,1	every other count from c
`d2 = (c >> 2) & 0011 0011 0011 0011`	0001 0010 0001 0001	1,2,1,1	the remaining counts from c
`e = d0 + d2`	0010 0010 0011 0010	2,2,3,2	list giving # of 1s in each 4-bit slice of a
`f0 = (e >> 0) & 00001111 00001111`	00000010 00000010	2,2	every other count from e
`f4 = (e >> 4) & 00001111 00001111`	00000010 00000011	2,3	the remaining counts from e
`g = f0 + f4`	00000100 00000101	4,5	list giving # of 1s in each 8-bit slice of a
`h0 = (g >> 0) & 0000000011111111`	0000000000000101	5	every other count from g
`h8 = (g >> 8) & 0000000011111111`	0000000000000100	4	the remaining counts from g
`i = h0 + h8`	0000000000001001	9	the final answer of the

			16-bit word

Here, the operations are as in C, so `x >> y` means to shift X right by Y bits, X & Y means the bitwise AND of X and Y, and + is ordinary addition. The best algorithms known for this problem are based on the concept illustrated above and are given here:

```
//types and constants used in the functions below

const uint64_t m1  = 0x5555555555555555; //binary: 0101...
const uint64_t m2  = 0x3333333333333333; //binary: 00110011..
const uint64_t m4  = 0x0f0f0f0f0f0f0f0f; //binary:  4 zeros,  4 ones ...
const uint64_t m8  = 0x00ff00ff00ff00ff; //binary:  8 zeros,  8 ones ...
const uint64_t m16 = 0x0000ffff0000ffff; //binary: 16 zeros, 16 ones ...
const uint64_t m32 = 0x00000000ffffffff; //binary: 32 zeros, 32 ones
const uint64_t hff = 0xffffffffffffffff; //binary: all ones
const uint64_t h01 = 0x0101010101010101; //the sum of 256 to the power of 0,1,2,3...

//This is a naive implementation, shown for comparison,
//and to help in understanding the better functions.
//It uses 24 arithmetic operations (shift, add, and).
int popcount_1(uint64_t x) {
    x = (x & m1 ) + ((x >>  1) & m1 ); //put count of each  2 bits into those  2 bits
    x = (x & m2 ) + ((x >>  2) & m2 ); //put count of each  4 bits into those  4 bits
    x = (x & m4 ) + ((x >>  4) & m4 ); //put count of each  8 bits into those  8 bits
    x = (x & m8 ) + ((x >>  8) & m8 ); //put count of each 16 bits into those 16 bits
    x = (x & m16) + ((x >> 16) & m16); //put count of each 32 bits into those 32 bits
    x = (x & m32) + ((x >> 32) & m32); //put count of each 64 bits into those 64 bits
    return x;
}

//This uses fewer arithmetic operations than any other known
//implementation on machines with slow multiplication.
//It uses 17 arithmetic operations.
int popcount_2(uint64_t x) {
    x -= (x >> 1) & m1;             //put count of each 2 bits into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits into those 4 bits
    x = (x + (x >> 4)) & m4;        //put count of each 8 bits into those 8 bits
    x += x >>  8;  //put count of each 16 bits into their lowest 8 bits
    x += x >> 16;  //put count of each 32 bits into their lowest 8 bits
    x += x >> 32;  //put count of each 64 bits into their lowest 8 bits
    return x & 0x7f;
}

//This uses fewer arithmetic operations than any other known
//implementation on machines with fast multiplication.
```

```
//It uses 12 arithmetic operations, one of which is a multiply.
int popcount_3(uint64_t x) {
    x -= (x >> 1) & m1;             //put count of each 2 bits into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits into those 4 bits
    x = (x + (x >> 4)) & m4;        //put count of each 8 bits into those 8 bits
    return (x * h01)>>56;  //returns left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

The above implementations have the best worst-case behavior of any known algorithm. However, when a value is expected to have few nonzero bits, it may instead be more efficient to use algorithms that count these bits one at a time. As Wegner (1960) described,[7] the bitwise and of *x* with *x* – 1 differs from *x* only in zeroing out the least significant nonzero bit: subtracting 1 changes the rightmost string of 0s to 1s, and changes the rightmost 1 to a 0. If *x* originally had *n* bits that were 1, then after only *n* iterations of this operation, *x* will be reduced to zero. The following implementation is based on this principle.

```
//This is better when most bits in x are 0
//It uses 3 arithmetic operations and one comparison/branch per "1" bit in x.
int popcount_4(uint64_t x) {
    int count;
    for (count=0; x; count++)
        x &= x-1;
    return count;
}
```

If we are allowed greater memory usage, we can calculate the Hamming weight faster than the above methods. With unlimited memory, we could simply create a large lookup table of the Hamming weight of every 64 bit integer. If we can store a lookup table of the hamming function of every 16 bit integer, we can do the following to compute the Hamming weight of every 32 bit integer.

```
static uint8_t wordbits[65536] = { /* bitcounts of integers 0 through 65535, inclusive */ }
static int popcount(uint32_t i)
{
    return (wordbits[i&0xFFFF] + wordbits[i>>16]);
}
```

Language support

Some C compilers provide intrinsics that provide bit counting facilities. For example, GCC (since version 3.4 in April 2004) includes a builtin function `__builtin_popcount` that will use a processor instruction if available or an efficient library implementation otherwise.[8] LLVM-GCC has included this function since version 1.5 in June, 2005.[9]

In C++ STL, the bit-array data structure `bitset` has a `count()` method that counts the number of bits that are set.

In Java, the growable bit-array data structure `BitSet` has a `BitSet.cardinality()` method that counts the number of bits that are set. In addition, there are `Integer.bitCount(int)` and `Long.bitCount(long)` functions to count bits in primitive 32-bit and 64-bit integers, respectively. Also, the `BigInteger` arbitrary-precision integer class also has a `BigInteger.bitCount()` method that counts bits.

In Common Lisp, the function logcount, given a non-negative integer, returns the number of 1 bits. (For negative integers it returns the number of 0 bits in 2's complement notation.) In either case the integer can be a BIGNUM.

Starting in GHC 7.4, the Haskell base package has a `popCount` function available on all types that are instances of the `Bits` class (available from the `Data.Bits` module).[10]

MySQL version of SQL language provides `BIT_COUNT()` as a standard function.[11]

Fortran 2008 has the standard, intrinsic, elemental function `popcnt` returning the number of nonzero bits within an integer (or integer array), see page 380 in *Metcalf, Michael; John Reid; Malcolm Cohen (2011). Modern Fortran Explained. Oxford University Press. ISBN 0-19-960142-9*.

Processor support

- Cray supercomputers early on featured a population count machine instruction, rumoured to have been specifically requested by the U.S. government National Security Agency for cryptanalysis applications.
- AMD's Barcelona architecture introduced the abm (advanced bit manipulation) ISA introducing the POPCNT instruction as part of the SSE4a extensions.

- Intel Core processors introduced a POPCNT instruction with the SSE4.2 instruction set extension, first available in a Nehalem-based Core i7 processor, released in November 2008.
- Compaq's Alpha 21264A, released in 1999, was the first Alpha series CPU design that had the count extension (CIX).
- Donald Knuth's model computer MMIX that is going to replace MIX in his book The Art of Computer Programming has an `SADD` instruction. `SADD a,b,c` counts all bits that are 1 in b and 0 in c and writes the result to a.
- The ARM architecture introduced the VCNT instruction as part of the Advanced SIMD (NEON) extensions.
- Analog Devices' Blackfin processors feature the ONES instruction to perform a 32-bit population count.

See also

- Minimum weight
- Two's complement
- Most frequent k characters

References

1. ^ *D. E. Knuth (2009). The Art of Computer Programming Volume 4, Fascicle 1: Bitwise tricks & techniques; Binary Decision Diagrams. Addison–Wesley Professional. ISBN 0-321-58050-8*. Draft of Fascicle 1b available for download.
2. ^ *Thompson, Thomas M. (1983), From Error-Correcting Codes through Sphere Packings to Simple Groups, The Carus Mathematical Monographs #21, The Mathematical Association of America, p. 33*
3. ^ *Reed, I.S. (1954), "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme", I.R.E. (I.E.E.E.), PGIT-4: 38*
4. ^ Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. 11, 1 (Feb. 2003), 17-32. Section 6.3: "In general, the number of fingers we need to follow will be the number of ones in the binary representation of the distance from node to query."
5. ^ *SPARC International, Inc. (1992). The SPARC architecture manual : version 8 (PDF) (Version 8. ed.). Englewood Cliffs, N.J.: Prentice Hall. p. 231. ISBN 0-13-825001-4*.

A.41: Population Count. Programming Note.
6. ^ *Blaxell, David (1978), "Record linkage by bit pattern matching", in Hogben, David; Fife, Dennis W., Computer Science and Statistics--Tenth Annual Symposium on the Interface, NBS Special Publication **503**, U.S. Department of Commerce / National Bureau of Standards, pp. 146–156.*
7. ^ *Wegner, Peter (1960), "A technique for counting ones in a binary computer", Communications of the ACM **3** (5): 322, doi:10.1145/367236.367286*
8. ^ "GCC 3.4 Release Notes" GNU Project
9. ^ "LLVM 1.5 Release Notes" LLVM Project.
10. ^ *"GHC 7.4.1 release notes"*. GHC documentation.
11. ^ *"12.11. Bit Functions — MySQL 5.0 Reference Manual"*.

External links

- Aggregate Magic Algorithms. Optimized population count and other algorithms explained with sample code.
- HACKMEM item 169. Population count assembly code for the PDP/6-10.
- Bit Twiddling Hacks Several algorithms with code for counting bits set.
- Necessary and Sufficient - by Damien Wintour - Has code in C# for various Hamming Weight implementations.
- Best algorithm to count the number of set bits in a 32-bit integer? - Stackoverflow