

Sviluppo



Di Michele Costabile

In tempi di servizi cloud e librerie di oggetti, soffermiamoci sull'efficienza degli algoritmi alla base dello sviluppo.

Il peso di Hamming

Questo mese non proveremo a spaccare i bit, ma ci metteremo a contarli seriamente. Ispirandoci a una domanda posta sovente nei colloqui di lavoro per aziende di software internazionali, poniamoci il problema di contare quanti bit sono impostati a uno in una word.

Questo problema è noto come calcolo del peso di Hamming oppure population count (*popcount*).

Contare il numero dei bit attivi in una word ha applicazioni interessanti, soprattutto nei sistemi di controllo in cui i bit attivi possono segnalare qualche condizione degna di nota e il sistema potrebbe essere confezionato in modo di acquisirli a gruppi di trentadue, ma non ci interesseremo delle applicazioni pratiche dell'algoritmo, è molto più interessante esplorare le possibilità di calcolo.

DUE PAROLE IN GENERALE

Qual è il modo giusto di calcolare il numero di bit settati in una word? Come in tutte le applicazioni informatiche, la risposta finisce per essere: dipende. Dipende dall'architettura della Cpu che stiamo usando, dal grado di parallelismo che abbiamo a disposizione, dal numero di bit che ci aspettiamo di trovare. Alcuni loop stretti possono stare interamente in qualche cache ed

essere più veloci del previsto, alcune soluzioni basate su ricerca in tabelle di valori precalcolati possono richiedere meno tempo di calcolo, ma fanno saltare l'accesso a memoria in un'area più ampia, magari non presente nella cache del processore, scatenando costosi interventi della memoria virtuale. Le variabili non finiscono qui: per esempio, la Cpu potrebbe avere un'istruzione specifica per il compito. In questo caso, probabilmente questa è la nostra scelta migliore, anche se non necessariamente.

Naturalmente, il modo migliore di scegliere una soluzione è quella di eseguire scrupolose misure. Ed è quello che faremo noi, un po' di benchmark per valutare le differenze di prestazioni fra diversi algoritmi. Per codificare i diversi algoritmi, useremo un linguaggio che abbiamo lasciato sullo scaffale per qualche tempo, sicuramente il più adatto per operazioni di questo tipo: il C++. Gli esempi di codice, comunque, sono facili da tradurre in qualsiasi altro linguaggio che abbia l'operazione di shift aritmetico. I lettori che desiderassero scaricare il codice di questo articolo lo troveranno in un gist (un comodo spazio per incollare frammenti di codice) su Github, all'indirizzo gist.github.com/michelecos/1cfd8a02cb1f547bdcf

LA SOLUZIONE PIÙ NAIF

Quando si codifica una soluzione, conviene partire dalla possibilità più semplice, con il codice più leggibile e discostarsene solo quando è dimostrata la convenienza, cosa che puntualmente accade in questo caso.

Il modo più semplice di contare i bit, è estrarre il bit meno significativo dalla word in esame, sommarlo a un accumulatore, fare uno shift verso destra e passare al bit successivo. Ripetendo il processo trentadue volte abbiamo risolto il problema. Ecco il codice che realizza quello che ci siamo proposto. La variabile *c* è quella che accumula il conteggio, *v* è il valore della word di cui contare i bit, che continuiamo a shiftare verso destra in modo da portare sempre un nuovo bit nella posizione meno significativa.

Dato che *v* è un intero non segnato, lo shift sarà logico e il bit più significativo verrà riempito con uno 0.

```
unsigned int bitcount_
vanilla2(unsigned int v) {
    unsigned int c = 0;

    for (int i = 0; i < 32; i++) {
        c += v & 1;
        v >>= 1;
    }
}
```

```
return c ;
}
```

Adesso, per quanto ci siamo ripromessi di essere naif, c'è un limite a tutto. Per esempio, non possiamo consentirci di eseguire la ricerca in tutti i bit quando si sa che non ci sono più bit da considerare. A ogni shift, il bit più significativo diventa zero e il meno significativo si perde, quindi *v* sarà uguale a zero quando il bit di rango massimo sarà stato shiftato fuori, e lo shift di *v* ci darà uno zero. Quando questo si verifica, interrompiamo il ciclo: non c'è scopo nel cercare bit impostati a uno in una word che vale zero. Ecco la versione ottimizzata della ricerca naif.

```
unsigned int bitcount_vanilla(unsigned int v) {
    unsigned int c;

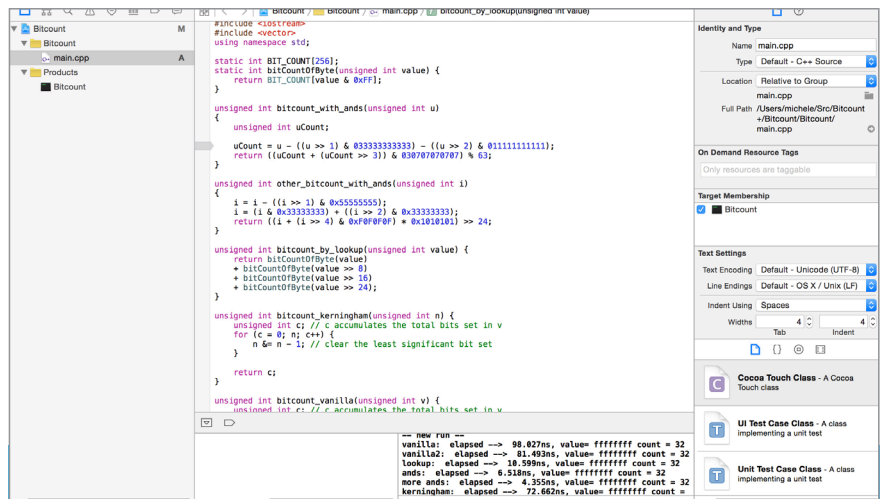
    for (c = 0; v; v >>= 1) {
        c += v & 1;
    }

    return c;
}
```

con un singolo for inizializziamo *c*, verifichiamo che *v* non sia zero e spostiamo i bit di *v* per il test successivo. Come è logico aspettarsi, vedremo che per numeri piccoli questa ottimizzazione ha effetti notevoli sul tempo di esecuzione.

UN PO' DI INFRASTRUTTURA

Abbiamo due versioni della stessa funzione ed è ora di metterle alla prova. Per valutare il tempo di esecuzione di una funzione, la chiamiamo un numero configurabile di volte, abbastanza alto da avere un tempo di esecuzione reale misurabile in secondi. Se, per ipotesi, una data funzione impiega tre secondi per essere eseguita un milione di volte, valuteremo il tempo di esecuzione in tre microsecondi, da cui dovremmo sottrarre il tempo di esecuzione del loop stesso. I tempi di esecuzione dei loop dipenderanno drasticamente anche da come stanno in cache le istruzioni eseguite, ma non arriveremo fino a questo punto. Cominciamo dall'inizio, prendiamo il tempo con la funzione clock (una delle



Il codice in fase di realizzazione, all'interno di XCode, l'ambiente di sviluppo di Apple scaricabile gratuitamente dal sito della società di Cupertino, developer.apple.com

ragioni per cui scegliamo di usare il C++ è per non perderci nelle specificità delle diverse piattaforme).

La funzione clock restituisce un `clock_t`, che sul nostro Mac è un `unsigned long`, così come sulla maggior parte dei sistemi.

Prendiamo il valore del clock all'inizio e alla fine del loop di elaborazione in modo da poter calcolare il tempo di esecuzione, che sarà in colpi di clock, dato che la funzione clock non restituisce un valore temporale specifico, ad esempio microsecondi, ma un valore dipendente dal sistema, cioè la granularità del clock, che in molti sistemi è

```
clock_t start, stop;
```

```
start = clock();
long count = 0;

for (unsigned int i = 0; i <
    ITERATIONS; i++) {
    count =
    bitcount_vanilla(value);
}
```

```
stop = clock();
```

Adesso, convertiamo in millisecondi la differenza espressa in colpi di clock

```
double ns = ((double)stop -
    (double)start)*1000.0/
    CLOCKS_PER_SEC;
```

Infine, presentiamo i risultati, con

```
cout << "elapsed --> " << ns <<
    "ns, value= 0x" << std::hex <<
    value;
cout << " count = " << std::dec <<
    count << endl;
```

Questo ci darà una riga di output simile a questa:

```
elapsed --> 11.557ns, value=
0xdeadcafe count = 22
```

Notiamo solo che `std::hex` richiede la formattazione dei numeri in formato esadecimale, mentre `std::dec` ripristina la visualizzazione decimale.

Adesso ci serve solo un piccolo artificio per creare una funzione che possa fare un benchmark di un metodo qualsiasi. Vogliamo passare il nome della funzione e il parametro su cui deve operare nell'invocazione della nostra funzione generica, quindi le diamo una signature come segue:

```
void benchmark_function(unsigned
    int (*method_under_test)(unsigned
    int param), unsigned int value)
```

Dobbiamo solo tenere presente che la segnatura `unsigned int (*method_under_test)(unsigned int param)` è il modo di indicare un puntatore a funzione che accetta un parametro `unsigned int` e restituisce un valore dello stesso tipo. Modifichiamo quindi il punto in cui invochiamo la funzione da così

```
for (unsigned int i = 0; i <
```

```

ITERATIONS; i++) {
    count =
    bitcount_vanilla(value);
}

```

a così

```

for (unsigned int i = 0; i <
ITERATIONS; i++) {
    count =
    method_under_test(value);
}

```

In sostanza, tutta la magia sta nella dichiarazione del primo parametro della funzione, l'invocazione resta praticamente uguale. Ecco il testo completo della funzione di benchmarking

```

#define ITERATIONS 1000000

void benchmark_function(unsigned
int (*method_under_test)(unsigned
int param), unsigned int value)
{
    clock_t start, stop;

    start = clock();
    long count = 0;

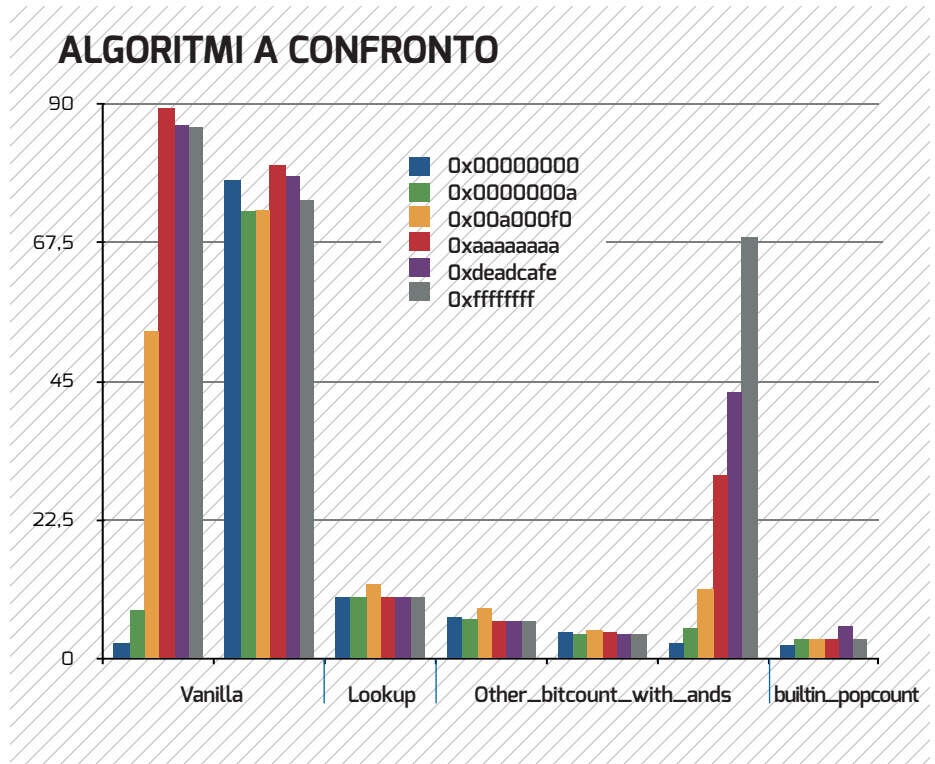
    for (unsigned int i = 0; i <
ITERATIONS; i++) {
        count =
        method_under_test(value);
    }

    stop = clock();

    double ns = ((double)
stop - (double)start)*1000.0/
CLOCKS_PER_SEC;

    cout << "elapsed --> " << ns
<< "ns, value= " << std::hex <<
value;

```



```

cout << " count = " << std::dec
<< count << endl;
}

```

LA SOLUZIONE DI KERNINGHAM

Passiamo ora a considerare un algoritmo per la ricerca dei bit settati che si deve nientedimeno che a Brian Kerningham, una delle star del C e di Unix. L'approccio di Kerningham è basato su una specificità dell'aritmetica binaria, il modo in cui si propagano i riporti nella sottrazione.

Cominciamo a considerare il sistema decimale e immaginiamo di sottrarre

uno dal numero 94. Il risultato è ovviamente 93, cioè cala solo la cifra delle unità, mentre se sottraiamo 5, il risultato è 89, perché per eseguire l'operazione dobbiamo tenere conto del riporto. Consideriamo adesso un caso limite della sottrazione, quello in cui sottraiamo uno da mille. La differenza è 999. Lo zero meno significativo diventa la cifra precedente. La diminuzione si propaga verso sinistra fino al primo uno, che viene diminuito spegnendo la cifra più significativa del numero mille. Suona complicato solo perché lo abbiamo espresso in modo più generico. Se in binario eseguiamo $0x70 - 1$ abbiamo (in sedici bit per semplicità)

```

01110000 -
00000001 =

```

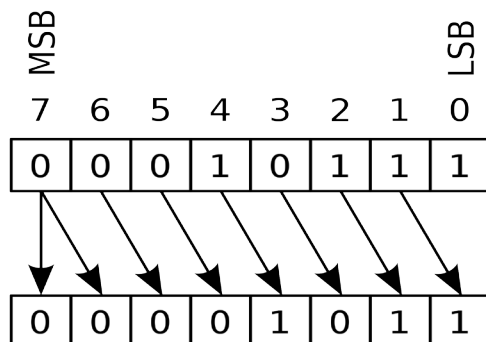
```

01101111

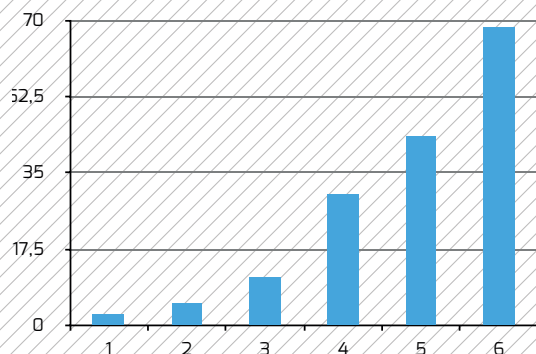
```

vediamo che succede qualcosa di analogo a togliere una unità da un numero decimale che finisce con una sequenza di zeri. Gli zeri meno significativi diventano 1 (non nove, in binario), mentre il primo bit a uno diventa zero. Sfruttando questa particolarità, l'algoritmo di Kerningham usa il risultato della sottrazione di uno da un numero, come maschera per spegnere il primo bit impostato. Il risultato

L'istruzione di shift, comune a tutte le Cpu e presente in diversi linguaggi, come il C, ma anche JavaScript, consiste nello spostamento dei bit di una word a sinistra o a destra. Per le particolarità dell'aritmetica in base due lo shift di una posizione a destra equivale a dividere per due, mentre lo shift a sinistra equivale a moltiplicare per due.

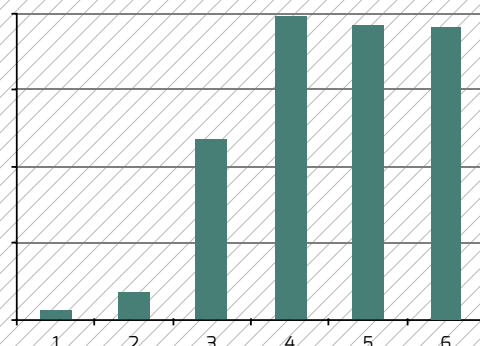


KERNINGHAM



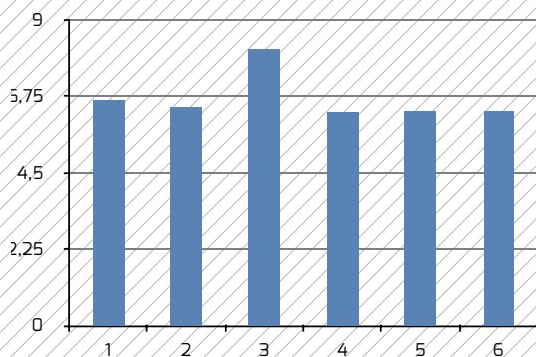
L'algoritmo è efficace con pochi bit da contare, il tempo di esecuzione cresce con word più ricche di cifre a uno.

VANILLA



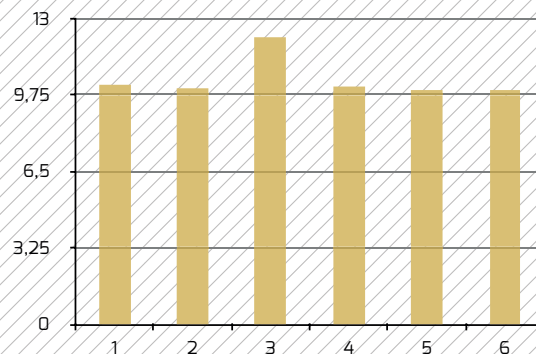
L'algoritmo più semplice è quello che offre i tempi peggiori, come potevamo aspettarci.

LOOKUP



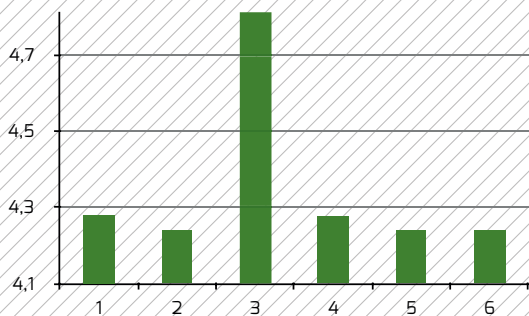
L'uso di valori precalcolati ci fa guadagnare in velocità. Ma l'accesso alla memoria rallenta il processore.

ACHMEM



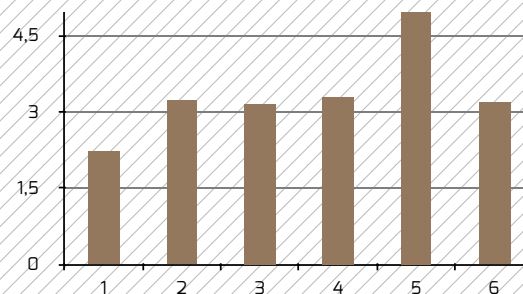
Il processore opera interamente nei registri con operazioni binarie semplici; i tempi sono uniformemente buoni.

OTHER BITCOUNT WITH ANDS



Operare sui bit a gruppi di tre, con questo algoritmo, ci porta ancora più avanti con prestazioni migliori.

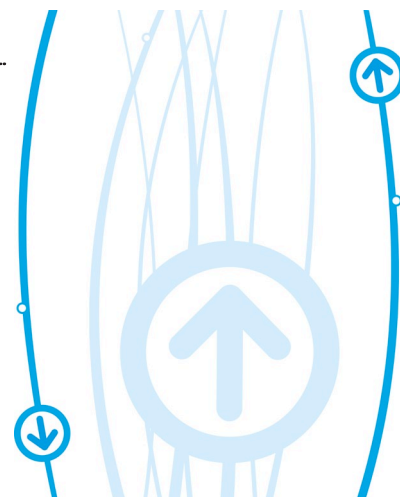
BUILTIN POPCOUNT



L'algoritmo della routine di libreria è quasi venti volte più veloce del metodo più semplice che abbiamo usato.

IL CALCOLO DELLA NORMA DI UN VETTORE, IN UNO SPAZIO L^1 A 32 DIMENSIONI

Per i più educati matematicamente, il problema del conteggio dei bit in una word è identico al calcolo della norma di un vettore in uno spazio L^1 a 32 dimensioni. Per definizione, la norma è la radice unesima della somma delle trentadue componenti elevate alla prima potenza, quindi si riduce alla somma delle componenti. Poiché queste sono bit, e valgono zero o uno, si vede che in pratica, la norma L^1 è uguale alla somma dei bit impostati. Si tratta solo di un altro modo di chiamare le cose, quindi, ma faremmo ottima figura in un incontro di matematici. Il modulo che siamo abituati a conoscere in fisica, come radice della somma dei quadrati delle componenti, corrisponde al caso di uno spazio L^2 , mentre in uno spazio L^3 parleremmo di radice cubica della somma dei cubi.



conterrà solo i bit impostati di ordine superiore.

Per chiarire, il numero originale è 01110000, il risultato della sottrazione è 01101111 e

```
01110000 && 01101111 => 01100000
```

Il numero di partenza, con il primo bit a uno resettato. Se ripetiamo il procedimento fino ad avere zero, il numero di volte che abbiamo eseguito il procedimento è uguale al numero di bit a uno. Ecco la realizzazione dell'algoritmo

```
unsigned int bitcount_
kerningham(unsigned int n) {
    unsigned int c;
    for (c = 0; n; c++) {
        n &= n - 1; // clear the
least significant bit set
    }
    return c;
}
```

Uno dei pregi di questo algoritmo è la compattezza e la semplicità del codice, facile da capire e da ricordare, nel caso venga chiesto in una intervista. L'altro vantaggio è che il numero di loop dipende dal numero di bit impostati a uno, quindi si tratta di un algoritmo efficace se ci si aspetta che la word sia per lo più popolata di zeri. Purtroppo, come nel caso dell'algoritmo precedente, se i bit a uno sono parecchi, il tempo di esecuzione peggiora notevolmente.

USARE UN ARRAY DI RISULTATI PRECALCOLATI

Una delle tecniche di base per velocizzare un procedimento di calcolo è non calcolare affatto, o meglio precalcolare una tabella di risultati e limitarsi a

ricercare il valore richiesto nella tabella. Questo procedimento può essere vantaggioso nel caso di operazioni particolarmente pesanti, come trasformazioni geometriche e potrebbe avere applicazione anche in questo caso.

Nel caso del numero di bit impostati in una word di trentadue, però, il numero di risultati precalcolati che ci serve è parecchio elevato, perché si tratta di 2^{32} word, cioè quattro Gbyte di memoria. Una soluzione, è usare una tabella di 256 valori per ogni byte e spezzare la nostra word in quattro.

Quindi, all'inizializzazione del programma calcoliamo il numero di bit per ogni combinazione di otto byte. Ecco come precalcolare in un array statico il numero di bit per i numeri da uno a 256

STATIC INT BIT_COUNT[256];

```
void initialize_count() {
    for (unsigned int i = 0; i <
sizeof(BIT_COUNT)/sizeof(unsigned
int); i++) {
        BIT_COUNT[i] =
bitcount_vanilla(i);
    }
}
```

La variabile BIT_COUNT è in maiuscolo per sottolineare che si tratta di una costante. Per calcolare il numero di bit nel byte meno significativo di una word, possiamo usare il codice che segue

```
static int bitCountOfByte(unsigned
int value) {
    return BIT_COUNT[value & 0xFF];
}
```

Ora non ci resta che sommare il risultato della valutazione di bitCountOfByte su una word eseguendo uno shift di otto

bit per tre volte consecutive, in modo da esaminare un byte alla volta.

```
unsigned int bitcount_by_
lookup(unsigned int value) {
    return bitCountOfByte(value)
+ bitCountOfByte(value >> 8)
+ bitCountOfByte(value >> 16)
+ bitCountOfByte(value >> 24);
}
```

L'idea sembra molto furba, in realtà questo è uno degli algoritmi peggiori, perché comunque le operazioni sono parecchie e l'accesso alla memoria è sparso.

UN SALTO MORTALE

Un altro algoritmo di calcolo sfrutta in modo molto astuto le particolarità dell'aritmetica binaria, per esempio il fatto che la divisione per due o per quattro si limita a uno shift a destra di una o due posizioni, una operazione che costa poco su un computer. Per capire il procedimento consideriamo una stringa di tre bit, poi estenderemo il procedimento a una intera word, vedendola come una serie di cifre ottali, cioè di sottostringhe di tre bit. Sappiamo che in binario 110 rappresenta il numero sei, l'uno più significativo sta per 4, il successivo per 2. In effetti, il numero 110 sta per

$$1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$$

cioè

$$1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

se le cifre, invece di 1, 1 e zero fossero genericamente a, b e c, avremmo

$$\text{numero} = 4a + 2b + c$$

Se dividiamo il numero per 2 e non consideriamo i decimali, abbiamo

```
numero/2 = 2a + 1b + 0
```

Sottraendo questo dall'originale, abbiamo

```
numero - numero/2 = 2a + b + c
```

Se dividiamo il numero originale per quattro, abbiamo

```
numero/4 = a + 0 + 0
```

sottraendo questo valore al risultato precedente abbiamo

```
numero - numero/2 - numero/4 = (4a + 2b + c) - (2a + b) - (a)
```

e il risultato è proprio

```
a + b + c
```

cioè la somma dei bit.

In codice questo si traduce facilmente, anche se il codice che ne risulta è piuttosto difficile da decifrare, senza sapere a priori cosa fa e perché.

La prima parte del discorso, la divisione per due si esegue in parallelo su tutti i gruppi di tre bit della word sotto test. L'and con una maschera con due bit impostati ogni tre serve a spegnere eventuali bit di riporto

```
((u >> 1) & 033333333333)
```

Similmente, si calcola quello che abbiamo chiamato numero/4 nella spiegazione precedente

```
((u >> 2) & 011111111111)
```

Una riga di codice tutt'altro che intuitiva è l'istruzione return, che somma gruppi di tre bit a due a due adiacenti con

```
((uCount + (uCount >> 3))
```

quindi, azzeri i gruppi di tre bit dispari con

```
030707070707
```

e infine ottiene la somma in modo non troppo intuitivo estraendo la radice digitale base-64 calcolando il resto della divisione per 63. Si può provare a cercare la pagina di Wikipedia relativa a digital-root per una discussione molto approfondita.

```
unsigned int bitcount_with_ands(unsigned int u)
{
    unsigned int uCount;
    uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);
    return ((uCount + (uCount >> 3)) & 030707070707) % 63;
}
```

Una spiegazione estesa di questo algoritmo si può trovare su StackOverflow (stackoverflow.com/questions/19729466/how-to-find-number-of-1s-in-a-binary-number-in-o1-time). Abbiamo trovato sul web anche un altro algoritmo che usa un approccio simile, ecco il sorgente, che pubblichiamo senza commenti.

```
unsigned int other_bitcount_with_ands(unsigned int i)
{
    i = i - ((i >> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
    return ((i + (i >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
}
```

AND THE WINNER IS

Abbiamo sottoposto a test i diversi algoritmi utilizzando diversi valori di test partendo da numeri, piccoli come 0xa fino a numeri con un maggior numero di bit validi, come 0xdeadcafe. Abbiamo anche incluso i casi ai bordi, come 0 e 0xffffffff, in modo da vedere come si comportano i diversi algoritmi che abbiamo illustrato. I numeri che pubblichiamo sono quelli di una

singola esecuzione. Ripetendo i test si hanno numeri diversi, ma si tratta di variazioni sperimentali accettabili, di ordine inferiore ai valori misurati, cioè di normali fluttuazioni.

Il vincitore indiscusso è la funzione di libreria standard `__builtin_popcount`, che il compilatore traduce in una singola istruzione assembly sui sistemi con Cpu Intel e il compilatore Gcc. Non solo la serie di risultati è consistentemente la migliore, ma lo spettro dei risultati è composto di valori molto vicini, un fattore molto desiderabile. Una funzione di libreria ideale, infatti, non solo deve avere alte prestazioni, ma anche essere prevedibile nel tempo di elaborazione, senza punti dolenti in cui la performance crolla. L'algoritmo di Kerningham ha ottime prestazioni quando il numero di bit da contare non è alto, ma il tempo impiegato cresce linearmente fino non fare meglio dell'algoritmo più ingenuo nei casi peggiori. L'algoritmo Hakmem e quello che abbiamo chiamato `more_ands` si comportano in maniera egregia e hanno tempi di risposta costanti sui diversi valori di test, quindi sono molto adatti per un uso generico. Anche il lookup con tabella precalcolata si comporta degnamente, dando tempi di risposta prevedibili in tutti i casi, ma la velocità è inferiore. La ragione principale sta nel fatto che l'accesso sparso a memoria costa più di un procedimento algoritmico abbastanza corto da stare interamente nella cache di primo livello del processore.

Si tratta comunque di un algoritmo facile da capire con risposta costante, ma non dimentichiamoci del costo dell'inizializzazione della tabella di lookup. Le realizzazioni più ingenue sono facili da capire e da spiegare, ma hanno prestazioni terribili e, nel caso dell'algoritmo naïf con piccola ottimizzazione, tempi che variano in modo radicale dai casi più facili a quelli più complicati. Il verdetto finale, comunque, è che non vale la pena di brigare per la ricerca di un algoritmo particolarmente intelligente, quando c'è comunque a disposizione una funzione di libreria ottimizzata fino al ferro e di facile uso.

PER SAPERNE DI PIÙ

La spiegazione in dettaglio dei due metodi basati su shift e and si può trovare a questo indirizzo, insieme a parecchi link utili per approfondire l'argomento <http://stackoverflow.com/questions/19729466/how-to-find-number-of-1s-in-a-binary-number-in-o1-time>