Cari studenti,

questo è un articolo che scrivo dopo parecchio lavoro e richiede molta pazienza per affrontarlo. Passate oltre se siete deboli di cuore...

Il problema

Durante la Coding Jam di Marzo, nel quarto problema, bisognava individuare un numero che serviva a decodificare un messaggio segreto.

In particolare il meccanismo era di partire da 4 numeri numeri casuali tra 1024 e 4096: x,y,w,z. Poi si definiscono le seguenti grandezze:

$M = xy - 1$
$e = wM + x$
$d = zM + y$
$n = \frac{ed-1}{M}$

Dato un numero m questo viene criptato facendo il calcolo $c = em \mod n$, ovvero il valore criptato c è dato dal resto della divisione per n del prodotto di e per m.

Il valore ricostruito r (che sarà uguale a m) può essere ottenuto come $r = cd \mod n$, ovvero calcolando il resto della divisione per n del prodotto di c per d.

In particolare, quanto vale d, sapendo che $e = 17459243613$ e $n = 66624478857659$?

Cosa vogliamo fare (in C)

La prima cosa che mi è venuta in mente è stata: prendiamo un numero a caso, lo cripto e poi provo a decodificarlo con tutti i numeri possibili. Quante prove devo fare? I quattro numeri vanno da 1024 a 4096, quindi

$M_{min} = 1024^2 - 1 \rightarrow d_{min} = 1024 \cdot (1024^2 - 1) + 1024 = 1024^3$
$M_{max} = 4096^2 - 1 \rightarrow d_{max} = 4096 \cdot (4096^2 - 1) + 4096 = 4096^3$

In totale sono $67\ 645\ 734\ 912$ di prove. Non drammatico. Ecco il codice:

```c
// Versione 1
#include <stdio.h>
#include <time.h>
#include <inttypes.h>

double diff_timespec(const struct timespec* time1, const struct timespec* time0) {
    return (time1->tv_sec - time0->tv_sec)
        + (time1->tv_nsec - time0->tv_nsec) / 1000000000.0;
}

int main()
{
    struct timespec start;
    (void)timespec_get(&start, TIME_UTC);

    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    // m è un valore a caso che quando sarà moltiplicato per e non sforerà i 64 bit
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    for (uint64_t d = dmin; d <= dmax; ++d) {
        if (d % dmin == 0) {
            printf("Current: %"PRIu64" - ", d);
            struct timespec stop;
            (void)timespec_get(&stop, TIME_UTC);
            double elapsed = diff_timespec(&stop, &start);
            printf("Elapsed: %g\r", elapsed);
        }
        uint64_t r = c * d % n;
        if (r == m) {
            printf("Solution: %"PRIu64" - ", d);
            struct timespec stop;
            (void)timespec_get(&stop, TIME_UTC);
            double elapsed = diff_timespec(&stop, &start);
            printf("Elapsed: %g\n", elapsed);
            break;
        }
    }
}
```

(per la cronaca `PRIu64` è `"llu"`, ma in versione standard per `uint64_t`) Dopo 08:47 min le ha provate tutte. Ma non ha trovato la soluzione... What?!

Qual è il problema? Ero stato attento ad evitare che `c * m` superasse i 64 bit. Purtroppo non ho garantito lo stesso per `c * d`. Infatti `c = 13227610363687` e quando `d` è maggiore di 1394563, il prodotto non è rappresentabile in 64 bit. Che cosa succede matematicamente? Che calcoliamo

$$((c * d) \bmod 2^{64}) \bmod n$$

(per i matematici: qui $\bmod$ *è l'operatore che mi dà il resto della divisione, il % del C).*

Da dove salta fuori quel $\bmod 2^{64}$? Lo standard C99 (§6.2.5/9) dice che

> A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

Insomma, se il prodotto sfora i 64 bit il risultato perde tutti i bit oltre il 64. Allora che cosa posso fare?

Usiamo 128 bit!

È possibile usare 128 bit? Non in C standard. Però GCC supporta questa estensione e quindi possiamo fare (non mostro più il codice per misurare il tempo):

```c
// Versione 2 - GCC
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    for (uint64_t d = dmin; d <= dmax; ++d) {
        uint64_t r = (__int128)c * d % n;
        if (r == m) {
            printf("Solution: %llu\n", d);
            break;
        }
    }
}
```

Fatto! Soluzione trovata in 15,17 minuti. Non una scheggia, ma almeno funziona. (Compilato con `gcc -O3`).

Ok, bravo GCC. Ma se voglio usare il compilatore della Microsoft? Sono piuttosto certo che in linguaggio macchina la motiplicazioni ritorni il risultato in due registri, uno per i 64 bit meno significativi e uno per i 64 bit più significativi. Posso accedere a quei valori? La risposta è sì, nuovamente utilizzando caratteristiche non standard.

La maggior parte delle funzioni è contenuta in librerie, ma alcune funzioni sono integrate (ovvero intrinseche) al compilatore. Queste sono chiamate funzioni intrinseche o più comunemente intrinsic functions o solo intrinsic. Se una funzione è un intrinsic, il codice per quella funzione viene solitamente inserito in linea, evitando il costo di una chiamata a funzione e generando istruzioni in linguaggio macchina molto più efficienti.

Nella libreria `<intrin.h>` sono disponibili le funzioni `_umul128` e `_udiv128`. La prima moltiplica due numeri a 64 bit e restituisce il risultato in due parti (i 64 bit meno significatvi e quelli più significativi). La seconda accetta il dividendo a 128 bit (di nuovo diviso in due parti) e il divisore a 64 bit e restituisce quoziente e resto a 64 bit.

Possiamo allora scrivere:

```c
// Versione 2 - MSVC
#include <stdio.h>
#include <inttypes.h>
#include <intrin.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    for (uint64_t d = dmin; d <= dmax; ++d) {
        uint64_t cdhigh;
        uint64_t cdlow = _umul128(c, d, &cdhigh);
        uint64_t r;
        _udiv128(cdhigh, cdlow, n, &r);
        if (r == m) {
            printf("Solution: %"PRIu64"\n", d);
            break;
        }
    }
}
```

Soluzione trovata in 12,91 minuti. Anche meglio!

Si può fare usando solo 64 bit?

La domanda è legittima. Facciamo un po' di ripasso di quello che la maestra ci ha insegnato alle elementari. Supponiamo di avere numeri in base 10 e che questi abbiano 3 cifre. Per fare la moltiplicazione procediamo così:

$$\begin{array}{rr} 987 & \times \\ 456 & = \\ \hline 5922 & + \\ 4935 & + \\ 3948 & = \\ \hline 450072 & \end{array}$$

Per calcolare il risultato della moltiplicazione di due numeri a 3 cifre possono servire fino 6 cifre per rappresentare il risultato. Per calcolare i risultati intermedi però ne bastano solo 4. Ogni risultato intermedio viene poi moltiplicato per 1, per 10 e per 100 prima di sommare nuovamente.

Spulciando un po' di aritmetica modulare, si impara che:

$$\begin{aligned} (a + b) \bmod n &= (a \bmod n + b \bmod n) \bmod n \\ (a \times b) \bmod n &= (a \bmod n \times b \bmod n) \bmod n \end{aligned}$$

Supponiamo allora di voler calcolare la moltiplicazione di prima $\bmod 991$:

$$(987 \times 456) \bmod 991 = 158$$

L'osservazione di prima non ci aiuta molto, visto che i due fattori sono entrambi minori di 991. Però abbiamo detto che:

$$987 \times 456 = 5922 + (4935 \times 10) + (3948 \times 100)$$

allora

$$
\begin{aligned}
&(987 \times 456) \bmod 991 = \\
&(5922 \bmod 991 + (4935 \times 10) \bmod 991 + (3948 \times 100) \bmod 991) \bmod 991 = \\
&(967 + (971 \times 10) \bmod 991 + (975 \times 100) \bmod 991) \bmod 991 = \\
&(967 + 791 + (975 \times 10 \times 10) \bmod 991) \bmod 991 = \\
&(967 + 791 + 382) \bmod 991 = \\
&((967 + 791) \bmod 991 + 382) \bmod 991 = \\
&(767 + 382) \bmod 991 = \\
&1149 \bmod 991 = \\
&158
\end{aligned}
$$

In pratica possiamo fare ogni somma parziale e ogni moltiplicazione modulo 991 ottenendo lo stesso risultato che avremmo avuto facendo il modulo alla fine. Sono sufficienti però solo 4 cifre. Notate che moltiplicare per 100 richiederebbe 5 cifre, ma se lo facciamo in due passi (sempre facendo il modulo sul risultato) ne bastano 4.

Dal punto di vista operativo è equivalente svolgere l'operazione nel seguente modo:

$$
\begin{aligned}
&(987 \times 456) \bmod 991 = \\
&(987 \times 6 + 987 \times 10 \times 5 + 987 \times 10 \times 10 \times 4) \bmod 991 = \\
&(967 + 951 \times 5 + 951 \times 10 \times 4) \bmod 991 = \\
&(967 + 791 + 951 \times 10 \times 4) \bmod 991 = \\
&(767 + 951 \times 10 \times 4) \bmod 991 = \\
&(767 + 591 \times 4) \bmod 991 = \\
&(767 + 382) \bmod 991 = \\
&1149 \bmod 991 = \\
&158
\end{aligned}
$$

Ovvero moltiplichiamo il moltiplicando per 10 (e teniamo il modulo), moltiplichiamo per la cifra successiva (e teniamo il modulo), sommiamo al risultato intermedio e continuiamo.

Questo come ci aiuta? Guardiamo i nostri numeri:

```
e    = 17459243613    = 0x 0004 10a6 ea5d
n    = 66624478857659 = 0x 3c98 3865 49bb
m    = 526990399      = 0x 0000 1f69 3c3f
c    = 13227610363687 = 0x 0c07 cb04 5727
dmax = 68719476736    = 0x 0010 0000 0000
```

Stanno tutti in 48 bit. Se trattiamo ogni 16 bit come una "cifra" del nostro numero, dobbiamo fare il prodotto di numeri a 3 "cifre" (blocchi da 16 bit). Per quello che abbiamo detto prima, possiamo calcolare tutto con 4 blocchi, ovvero con 64 bit. Ok, facciamolo:

```c
// Versione 3
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    for (uint64_t d = dmin; d <= dmax; ++d) {
        uint64_t x = c;
        uint64_t y = d;
        uint64_t r = 0;
        while (1) {
            r += x * (y & 0xffff) % n;
            r %= n;
            y >>= 16;
            if (y == 0) {
                break;
            }
            x = (x << 16) % n;
        }
        if (r == m) {
            printf("Solution: %"PRIu64"\n", d);
            break;
        }
    }
}
```

Soluzione trovata in 26,54 minuti (MSVC). Ci sono una moltiplicazione e tre divisioni e quindi è ragionevole come peggioramento. Inciso: GCC ci mette 3,43 minuti! Su questo approfondiremo successivamente.

L'ottimizzatore, che meraviglia. Ma a volte bisogna guardarci dentro.

Guardando il codice macchina generato dalla prima soluzione (quella che non funzionava) mi sono accorto di un fenomeno particolare. Vediamo un esempio semplificato:

```c
// Intermezzo
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t c = 1234567;
    uint64_t m = 12345670;

    for (uint64_t d = 1; d <= 100; ++d) {
        uint64_t r = c * d;
        if (r == m) {
            printf("Solution: %"PRIu64"\n", d);
            break;
        }
    }
}
```

Il codice assembly generato da MSVC (/O2) è il seguente (rivisto):

```
        [...]
        mov     edx, 1                          ; edx = d
        mov     eax, 1234567                    ; eax = c
ciclo:
        cmp     rax, 12345670                   ; if (r == m) goto solution
        je      solution
        inc     rdx                             ; ++d
        add     rax, 1234567                    ; r = r + c
        cmp     rax, 123456700                  ; if (r < c*100) goto ciclo
        jbe     ciclo
        [...]
        ret     0                               ; return from main
solution:
        lea     rcx, format_string              ; "Solution: %llu\n"
        call    printf
        [...]
```

In pratica l'ottimizzatore si accorge che facciamo $r \leftarrow c \times 1$, poi $r \leftarrow c \times 2$, poi $r \leftarrow c \times 3$, ... e quindi propone un più banale $r \leftarrow c$, poi $r \leftarrow r + c$, poi ancora $r \leftarrow r + c$ e così via. Insomma, evita completamente moltiplicazione. Questo però possiamo farlo anche noi, come al solito facendo il modulo dopo ogni operazione.

```c
// Versione 4
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    uint64_t r = 0;
    for (uint64_t d = 1; d <= dmax; ++d) {
        r = (r + c) % n;
        if (r == m) {
            printf("Solution: %"PRIu64"\n", d);
            break;
        }
    }
}
```

Soluzione trovata in 5,10 minuti (MSVC). Inciso: GCC ci mette 1 minuto e 35 secondi! Su questo approfondiremo successivamente (bis).

I mostri di StackOverflow

Stupito dal comportamento dell'ottimizzatore in una variante ulteriore che non sto qui a riportare, ho chiesto su StackOverflow e l'utente chtz fa questo commento:

> Something like this should also be sufficient: `r+=c; r= r>=n?r-n:r` since after adding `c` once `r` can't increase by more than `n`.

Mhhh... Sai che è vero? In effetti $r \in \mathbb{N} \cap [0, n)$, viene incrementato di $c \in \mathbb{N} \cap [0, n)$, quindi $r < 2n$. Quindi quando $n \leq r < 2n$, $r - n = r \bmod n$.

```c
// Versione 5
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    uint64_t r = 0;
    for (uint64_t d = 1; d <= dmax; ++d) {
        r = r + c;
        r = r >= n ? r - n : r;
        if (r == m) {
            printf("Solution: %"PRIu64"\n", d);
            break;
        }
    }
}
```

Soluzione trovata in 46,193 secondi (MSVC). Inciso: GCC ci mette 16,127 secondi!

Le magie di GCC

Vediamo che cosa fa MSVC per implementare la "Versione 5" del nostro codice:

```
        xor     r8d, r8d                    ; r = 0 (r8 è r)
        mov     rbx, 66624478857659         ; rbx è n
        mov     r9, 13227610363687          ; r9 è c
        mov     r11, -53396868493972        ; r11 è c - n
        mov     r10, 68719476736            ; r10 è dmax
        lea     edx, QWORD PTR [r8+1]       ; d = 1 (rdx è d)
        npad    11
ciclo:
        lea     rax, QWORD PTR [r8+r9]      ; tmp = r + c
        mov     rcx, r11                    ; tmp2 = c - n
        cmp     rax, rbx                    ; if (tmp < n)
        cmovb   rcx, r9                     ;     tmp2 = c
        add     r8, rcx                     ; r = r + tmp2
        cmp     r8, 526990399               ; if (r == m)
        je      print                       ;     goto print
        inc     rdx                         ; ++d
        cmp     rdx, r10                    ; if (d <= dmax)
        jbe     ciclo                       ;     goto ciclo
        [...]
        ret     0                           ; return from main
print:
        lea     rcx, format_string          ; "Solution: %llu\n"
        call    printf
        [...]
        ret     0                           ; return from main
```

Quindi che cosa fa MSVC? Invece che incrementare `r` e poi sottrarre `n` se il risultato è maggiore, decide di sommare a `r` il valore di `c -n` o di `c` a seconda di quello che farebbe sommargli `r`. Non ho una spiegazione sul perché! Però fila logicamente. In ogni caso non modifica la logica che ho scritto nel codice. Ad ogni ciclo si fanno due `cmp`.

Vediamo invece GCC:

```
        mov     esi, 1                  ; d = 1 (esi è d)
        xor     eax, eax                ; r = 0 (rax è r)
        movabs  rdi, 13227610363687     ; rdi è c
        movabs  rcx, 66624478857658     ; rcx è n
        movabs  r9, -53396868493972     ; r9 è c - n
        movabs  r8, 68719476737         ; r8 è dmax + 1
        jmp     ciclo                   ; goto ciclo
sottrai_e_confronta:
        add     rax, r9                 ; r = r + c - n
        cmp     rax, 526990399          ; if (r == m)
        je      print                   ;     goto print
        add     rsi, 1                  ; ++d
        cmp     rsi, r8                 ; if (d == dmax + 1)
        je      fine                    ;     goto fine
ciclo:
        lea     rdx, [rax+rdi]          ; tmp = r + c
        cmp     rcx, rdx                ; if (n < tmp)
        jb      sottrai_e_confronta     ;     goto sottrai_e_confronta
        add     rsi, 1                  ; ++d
        mov     rax, rdx                ; r = tmp
        cmp     rsi, r8                 ; if (d != dmax + 1)
        jne     ciclo                   ;     goto ciclo
fine:
        [...]
        ret
print:
        [...]
        call    printf
        [...]
        ret
```

Magia. Intanto somma una sola volta invece che due volte per ciclo, ma soprattutto controlla se `r` è uguale a `m` solo nel caso in cui si sottrae `n`, ovvero nei casi di sforamento. Perché?

Effettivamente `c` è maggiore di `m`. L'unico caso in cui è possibile che sommando `c` si ottenga un valore minore di `c` è quando si sottrae `n`! Per me è veramente incredibile che se ne accorga. Bisogna stringere la mano a chi ha avuto questa intuizione. Vorrei vedere com'è un algoritmo che controlla questa roba in ogni `if`...

Alla luce di questo possiamo anche riscrivere in modo più furbo il codice C:

```c
// Versione 6
#include <stdio.h>
#include <inttypes.h>

int main()
{
    uint64_t e = 17459243613;
    uint64_t n = 66624478857659;
    uint64_t m = 0x7fafffbffcffdfff / e;
    uint64_t c = e * m % n;

    uint64_t dmin = 1024ULL * 1024 * 1024;
    uint64_t dmax = 4096ULL * 4096 * 4096;
    uint64_t drng = dmax - dmin;

    uint64_t r = 0;
    for (uint64_t d = 1; d <= dmax; ++d) {
        r = r + c;
        if (r >= n) {
            r -= n;
            if (r == m) {
                printf("Solution: %"PRIu64"\n", d);
                break;
            }
        }
    }
}
```

Soluzione trovata in 17,6217 secondi (MSVC). Il codice assembly di GCC non cambia (quindi immagino ci metterà lo stesso tempo di prima).

L'ultimo sforzo: come fa GCC a fare così in fretta quando usiamo il modulo?

Nella "Versione 4" abbiamo praticamente solo `r = (r + c) % n`. Al solito dobbiamo dare un'occhiata all'assembly:

```
lea     rcx, [rdx+r10]  ; tmp = r + c
mov     rax, rcx        ;
mul     r9              ; tmp2 = tmp * 0x8731'8BF1'68D0'7B7F
shr     rdx, 45         ; tmp3 = tmp2 >> 109
mov     rax, rdx
imul    rax, r8         ; tmp4 = tmp3 * n
sub     rcx, rax        ;
mov     rdx, rcx        ; r = tmp - tmp4
```

WTF?! Ma cos'è quel numero? Perché questo dovrebbe essere equivalente a `r = (r + c) % n`?

Ho dovuto lavorarci un po', ma alla fine il trucco è trasformare la divisione in una moltiplicazione, perché la divisione è più lenta da fare. Partiamo dal modulo. Che cos'è?

$$x \bmod n = x - (x \operatorname{div} n) \times n$$

dove div è la divisione intera. Questo è esattamente quello che si vede nelle ultime 4 istruzioni del codice precedente. In pratica si fa `r = tmp - tmp3*n`. Quindi `tmp` è il dividendo e `n` è il divisore. A questo punto

`tmp3` deve per forza essere `(r+c) / n`. Ma sembra che per qualche ragione venga calcolato così:

$$\left\lfloor \frac{x}{66624478857659} \right\rfloor = \left\lfloor \frac{x \times 9741721337940966271}{2^{109}} \right\rfloor$$

Aspetta, ma

$$\frac{2^{109}}{66624478857659} = 9741721337940966270$$

Quindi

$$\left\lfloor \frac{x \times 9741721337940966271}{2^{109}} \right\rfloor = \left\lfloor \frac{x}{66624478857659} + \frac{x}{2^{109}} \right\rfloor$$

Sarò sincero, non ho capito la necessità di quel $+1$, ma immagino che sia legato ai casi limite dell'intero inferiore. Però il risultato complessivo mi sembra chiaro. Invece che dividere, moltiplico per una costante e faccio uno shift a destra. Il tempo di queste due operazioni è evidentemente inferiore a quello della divisione.

Ok. Ho finito. Se sei arrivato fino a qui leggendo tutto, sei un vero nerd. Eccoti un premio: 🏆