```
/* FILE: elle.cpp   last change: 17-Aug-2012   author: Romeo Rizzi
 * This program is a solver for problem 3 (elle) in 03-09-2012 exam in Algorithms
 */

#define NDEBUG // NDEBUG disabilita tutte le assert se definita prima di includere <cassert>
#include <cassert>

#ifndef NDEBUG
#  include <iostream>  // non voglio includerla quando Non compilo in modalita' DEBUG
#endif
#include <cstdlib>
#include <fstream>
#include <climits>

using namespace std;

const int M = 10000000;
const int UNKNOWN = -1;
const int MAX_N_COUNT = 100000;
long long int n;
int mem[MAX_N_COUNT+1][3]; // mem[nA][nA-nC]
void init_mem() { for(int nA = 0; nA <= MAX_N_COUNT; nA++) for(int d = 0; d <=2; d++)  mem[nA][d] = UNKNOWN; }

int ric_num_sol_mod_m(int nA, int nB, int nC) { // Scritta come "first writing" per elucidare l'approccio.
/* we assume the three rows (A, B and C) have been partly occupied, starting from the left and with no gaps,
   so that the current situation, as received in input by <ric_num_sol_mod_m>, looks for example as follows
     XXXXX.......    <-- row A has nA=8 still empty (.) positions and 5=n-nA occupied (X) positions
     XXXX.........    <-- row B has nB=9 still empty (.) positions and 4=n-nB occupied (X) positions
     XXXXXX.......    <-- row C has nC=7 still empty (.) positions and 6=n-nC occupied (X) positions
   and can be compactly described by the 3 numbers <nA>, <nB> and <nC>.
   We also assume that max(nA, nB, nC) <= min(nA, nB, nC) +2.
   Given <nA>, <nB> and <nC>, the recursive procedure <ric_num_sol_mod_m> outputs the number
   of different perfect tilings of L-shapes whithin the empty positions.
*/
  if( (nA < 0) | (nB < 0) | (nC < 0) )  return 0;
  if( (nA == 0) & (nB == 0) & (nC == 0) )  return 1;
  assert( max(nA, max(nB, nC)) <= min(nA, min(nB, nC)) +2 );
  if( nA < nC ) { int tmp = nA; nA = nC; nC = tmp; } // the answer does not change, by simmetry
  assert( nA >= nC );
  if( nA == nC ) {
    if( nB == nA ) /* situation:
                        X...                    XAA.                        XAA..
     XA..           XABBB                       X...       --> either   XA..  and then forced to XAB..
 or  XA.. forced to XAB..                        X...                    XA..                        XABBB
     XAA.           XAA..     */
                       return ( ric_num_sol_mod_m(nA-2, nB-2, nC-4) + ric_num_sol_mod_m(nA-4, nB-2, nC-2) ) % M;
  }
  if( nA == nC +1) {
    if( nB == nA -1) /* situation:
                         X...                   XAA.           XAAA..
XAAACCC
                         XX..     --> either   XXA.       or  XXBA.. forced to
XXBAC..
                         XX..                   XXA.           XXBBB.
XXBBB..              */
                            return ( ric_num_sol_mod_m(nA-2, nB-1, nC-1) + ric_num_sol_mod_m(nA-6, nB-3, nC-3) ) % M;
  }
  if( nA == nC +2) {
    if( nB == nA ) /* situation:
                              X...                    XA..             XABBB.          X
AAA           XAAAB.
                              X...     --> either   XAAA  forced to XAAABC     or  X
A.. forced to XABBBC
                              XXX.                    XXX.             XXXCCC          X
XX.           XXXCCC       */
                            return ( 2*ric_num_sol_mod_m(nA-4, nB-5, nC-3) ) % M;
  }
  assert( 0 ); // la lista dei casi considerati e' completa (include ogni caso generato)
}

int mem_num_sol_mod_m(int nA, int nC) {
// Versione memoizzata della procedura <ric_num_sol_mod_m> vista sopra.
// La memoizzazione conduce ad uno speed-up esponenziale.
// La procedura e' stata riscritta privilegiando la compattezza, anche al fine di facilitare la memoizzazione.
// In particolare abbiamo tolto il parametro nB, sopra rivelatosi ridondante, per pesare meno sullo stack
// (altrimenti foravamo per n grande "Segmentation fault (core dumped)").
// Abbiamo anche aggiunto la seguente precondizione sull'input: ( nA >= nC )
  assert( nA >= nC ); assert( nA <= nC +2 );
  if( (nA < 0) | (nC < 0) )  return 0;
  if( (nA == 0) & (nC == 0) )  return 1;
  if( mem[nA][nA-nC] != UNKNOWN )  return mem[nA][nA-nC];
  if( nA == nC ) {
     /* assert( nB == nA );situation:
                              X...                    XAA.                        XAA..
        XA..             XABBB                       X...       --> either   XA..  and then forced to XAB..
   or  XA.. forced to XAB..                            X...                    XA..                        XABBB
        XAA.             XAA..    */
    return mem[nA][nA-nC] = ( 2*mem_num_sol_mod_m(nA-2, nC-4) ) % M;
  }
  if( nA == nC +1) {
     /* assert( nB == nA -1);situation:
                               X...                    XAA.          XAAA..           X
AAACCC                  XXBBB..
                               XX..     --> either   XXA.    or  XXBA.. forced to X
XBAC..   flipped to   XXBAC..
                               XX..                    XXA.          XXBBB.           X
XBBB..  (by simmetry) XAAACCC */
    return mem[nA][nA-nC] = ( mem_num_sol_mod_m(nA-2, nC-1) + mem_num_sol_mod_m(nC-3, nA-6) ) % M;
  }
  if( nA == nC +2) {
    /* assert( nB == nA );  situation:
                              X...                    XA..             XABBB.
XAAA           XAAAB.
                              X...     --> either   XAAA  forced to XAAABC     or
XA.. forced to XABBBC
                              XXX.                    XXX.             XXXCCC
XXX.           XXXCCC       */
    return mem[nA][nA-nC] = ( 2*mem_num_sol_mod_m(nA-4, nC-3) ) % M;
  }
  assert( 0 ); // la lista dei casi considerati e' completa (include ogni caso generato)
}

// NOTA: puoi verificare la procedura <mem_num...> confrontandola contro <ric_num...> per valori di n moderati.
//       (Io avevo dimenticato di flippare i parametri nella seconda chiamata de
```

```
l caso 2: ( nA == nC +1).)
// Dopo aver curiosato il numero di tilings per vari piccoli valori di n viene n
aturale la congettura:
bool esistenza_conjecture(long long int n) { if(n%8) return false; else return t
rue; }
// che reggendo alla seguente verifica deve essere vera:
bool check_conjecture() {
  for(int n = 0; n < 100; n++)
    if( esistenza_conjecture(n) != (bool) ric_num_sol_mod_m( n, n, n) ) return f
alse;
  return true;
}

int main() {
  assert( check_conjecture() );
  assert( 2*M < INT_MAX ); // scrupolo forse eccessivo (i casting impliciti dovr
ebbero lavorare a favore)
  ifstream fin("input.txt"); assert(fin);
  ofstream fout("output.txt"); assert(fout);
  init_mem(); bool only_decide = false;
  for(int i = 1; i <= 5; i++) {
    fin >> n;
    if( n > MAX_N_COUNT ) only_decide = true;
    if(only_decide)  fout << esistenza_conjecture(n) << " ";
    else fout << mem_num_sol_mod_m (n, n) << " ";
  }
  fin.close(); fout.close();
  return 0;
}
```