multiplicamos

```
837649587637 * 984758392081 = ?
```

multiplicamos otra vez

```
837649587637 * 984758392081 = ?
824882461048724816302597
```

algoritmo secuencial

- Asumimos que tengamos solamente las operaciones aritméticas *sumar* y *subtraer* disponibles en un procesador ficticio y queremos multiplicar dos números positivos.
- Un posible algoritmo sequencial que multiplica el número *p* con el número *q* produciendo el resultado *r* es:

```
Initially:  set p and q to positive numbers

a: set r to 0
b: loop
c:   if q equal 0 exitloop
d:   set r to r+p
e:   set q to q-1
f: endloop
g: ...
```

¿Cómo se comprueba si el algoritmo es correcto?

- Primero tenemos que decir que significa correcto.
- El algoritmo (secuencial) es correcto si
  - una vez se llega a la instrucción `g:` el valor de la variable `r` contiene el producto de los valores de las variables `p` y `q` (se refiere a sus valores que han llegado a la instrucción `a:`)
  - se llega a la instrucción `g:` en algún momento
  - y la entrada habia sido la correcta.

¿Cómo se comprueba si el algoritmo es correcto?

- Tenemos que saber que las instrucciones atómicas son correctas,
- es decir, sabemos exactamente su significado, incluyendo todos los efectos segundarios posibles.
- Luego usamos el concepto de inducción para comprobar el bucle.

Invariante e inducción

- Sean p_i, q_i, y r_i los contenidos de los registros `p`, `q`, y `r`, después de la iteración i del bucle.
- Una invariante cuya corrección hay que comprobar con el concepto de inducción es entonces:

$$r_i + p_i \cdot q_i = p \cdot q$$

- ¿Cómo encontrar una invariante adecuada?
- usar *ingenio*...
  (RAE: Facultad del ser humano para discurrir o inventar con prontitud y facilidad.)
- Además, si comprobramos que q_i al final (saliendo del bucle) es cero, entonces, obviamente, el registro `r` contendrá el producto.

algoritmo concurrente

Re-escribimos el algoritmo secuencial para que "*funcione*" con dos procesos:

```
Initially:  set p and q to positive numbers

a: set r to 0
 P0                              P1
b: loop                            loop
c:   if q equal 0 exit               if q equal 0 exit
d:   set r to r+p                    set r to r+p
e:   set q to q-1                    set q to q-1
f: endloop                         endloop
g: ...Harvey-van der Hoeven
```

Implementamos la multiplicación con Java

- Realizamos una clase para valores enteros.
- Implementamos el bucle que realiza la multiplicación.
- Colocamos todo en un programa completo.

Enteros como objectos

Realizamos una clase para tener los enteros como clase propia (`Integer` no nos vale):

```
class Int {
  int i;
  Int(int i) { this.i=i; }
  void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

Preparar trabajador

Preparamos los hilos trabajadores con acceso a las variables comunes:

```
class Mul implements Runnable {
  private int id; // thread identity
  private Int p;  // reference to shared first factor
  private Int q;  // reference to shared second factor
  private Int r;  // reference to shared result

  Mul(int id, Int p, Int q, Int r) {
    this.id=id;
    this.p=p;
    this.q=q;
    this.r=r;
  }
```

El trabajo del trabajador

Implementamos el método `run()` para realizar el bucle de multiplicación:

```
  public void run() {
    try {
      System.out.println("starting worker... "+id);
      Int minusOne=new Int(-1);
      while(q.Get()!=0) {
        r.Add(p);
        q.Add(minusOne);
      }
    }
    catch(Exception E) { System.out.println("??? "+id);
    finally { System.out.println("exiting... "+id); }
  }
}
```

El programa principal I

Tratar la entrada:

```
class Multi {
  static Int p,q,r; // our shared variables for r=p*q

  public static void main(String[] args) {
    try {
      if(args.length!=3) {
        System.out.println("please 3 arg's: p q n");
        System.exit(1);
      }

      p=new Int(Integer.parseInt(args[0]));
      q=new Int(Integer.parseInt(args[1]));
      r=new Int(0);
```

El programa principal II

Crear, lanzar, y sincronizar los trabajadores:

```
final int n=Integer.parseInt(args[2]);
final Thread[] threads=new Thread[n];

for(int i=0; i<threads.length; ++i) {
  threads[i]=new Thread(new Mul(i,p,q,r));
}

for(int i=0; i<threads.length; ++i) {
  threads[i].start();
}

for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
```

El programa principal III

Visualización del resultado:

```
        System.out.println(
          args[0]+"*"+args[1]+"="+r.Get()+" ??"
        );
      }
      catch(Exception E) {
        System.out.println("catched an exception...");
        System.exit(1);
      }
      finally {
        System.out.println("exiting...");
      }
    }
}
```

¿Qué es que observamos?

- El algoritmo es no-determinista,
- en el sentido que no se sabe de antemano en qué orden (en un procesador o en un conjunto de procesadores) se van a ejecutar las instrucciones,
- o más preciso, cómo se van a intercalar las instrucciones atómicas de ambos procesos.
- El no-determinismo puede provocar situaciones con errores, es decir, el fallo ocurre solamente si las instrucciones se ejecutan en un orden específico.
- El resultado del programa no es predecible, y lo peor es, a veces es correcto.
- Desde el punto de vista de concurrencia tiene varios *problemas*.