¿Dónde están los problemas con tal multiplicación?

Tenemos varios problemas con el algoritmo.
El primero seguramente ya es obvio: la operación `Add` cual usamos tanto para aumenter `r` y disminuir `q` no es atómica (aquí de nuevo su código):

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

Se varios hilos actuan con `Add` sobre la misma variable, puede ser que el efecto de algunos participantes *desaparece* ya que el valor que intenten escribir será sobre-escrito por otros.

Sincronización

- A veces se quiere que un hilo actua sin que otros interfieran en su tarea.
- Es decir, se quiere una ejecución con exclusión mutua del código.
- Dicho concepto se llama también atomicidad de las operaciones,
- o también ejecución segura con hilos (*threadsafe*).
- En Java existen diferentes posibilidades para conseguir exclusión mutua.

Atomicidad en Java

- Solo las asignaciones a variables de tipos simples de 32 bits son atómicas.
- `long` y `double` no son simples en este contexto porque son de 64 bits.
- Hay que declarar esas variables como `volatile` para obtener acceso atómico.

synchronized

- En Java es posible forzar la ejecución del código en un bloque de modo sincronizado, es decir, como mucho un hilo, que tenga acceso compartido a `obj`, puede ejecutar el código dentro de dicho bloque.
  ```
  synchronized(obj) { ... }
  ```
- La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un vector.
- Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta por un sólo hilo (y ningún otro método sincronizado del mismo objeto tampoco).
- La máquina virtual instala un cerrojo (mejor dicho, un monitor, ya veremos dicho concepto más adelante) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

Métodos syncronizados

- Declarar un método como
  `synchronized void f(...) { ... }`
  es equivalente a usar un bloque sincronizado en su interior:
  `void f(...) { synchronized(this) { ... } }`
- Los monitores (implementados en la MVJ) permiten que el mismo hilo puede acceder a otros métodos o bloques sincronizados del mismo objeto sin problema.
- Se libera el cerrojo sea el modo que sea que termine el método.
- Los constructores no se pueden declarar `synchronized`.

Enteros con `int` y método sincronizado

```
// Implementation of our integer with a simple int
// and synchronized Add.
class Int {
  public int i;
  Int(int i) { this.i=i; }
  synchronized void Add(Int I) { i=i+I.i; }
  int Get() { return i; }
}
```

¡Implementa con esta clase el algoritmo!

Multiplicación concurrente correcto

Tenemos cuidado que los hilos no hagan nada "demás"... es decir, comparamos con $q > n$:

```
public void run() {
  try {
    final Int minusOne=new Int(-1);
      // Here, we check "greater than n" to avoid negative q !!
    while(q.Get()>n) {
      r.Add(p);
      q.Add(minusOne);
    }
  }
  catch(Exception E) {
    System.out.println("some error..."+id);
  }
```

Multiplicación concurrente correcto

Nos preocupamos que se realiza todo lo que hay que hacer... realizamos los posibles remanentes en el hilo principal después de la sincronización con las trabajadoras.

```
for(int i=0; i<threads.length; ++i) {
  threads[i].join();
}
  // Here we take care of the left-overs.
final Int minusOne=new Int(-1);
while(q.Get()>0) {
  r.Add(p);
  q.Add(minusOne);
}
```