# Programación Concurrente

## Trabajo Práctico

Se desea implementar en Java una clase que encapsule la lógica para ordenar una lista de números de manera concurrente mediante el método RadixSort. Se debe proveer una clase `ConcurRadixSort` que implemente la operación `radixSort`, que dada una lista la ordena aplicando el algoritmo RadixSort concurrentemente. Para esto durate la creación de un `ConcurRadixSort` se debe tomar un parámetro entero `threads`, que indica la cantidad máxima de threads a utilizar para realizar el ordenamiento.

Consideraremos la siguiente variante del algoritmo RadixSort para números enteros representados con 32 bits (i.e., `int`).

```
radixSort(list) {
  for (i = 0; i < 32; ++i) {
    aux = split(list, i);
    list = aux[0] + aux[1];
  }
  return list;
}

split(list, i) {
  zeros = []
  ones = []
  mask = 1 << i;
  for (i : list) {
    if (i & mask)
      ones.add(i);
    else
      zeros.add(i);
  }
  return {ones, zeros};
}
```

Se recomienda mantener en la representación privada de la clase `ConcurRadixSort` las variables auxiliares necesarias para realizar el proceso (e.g., una lista auxiliar del mismo tamaño que la lista de entrada).

El mecanismo por el cual los threads toman y procesan nuevo trabajo debe ser llevado a cabo por un "thread pool" con la siguiente estructura:

1. Una clase `Buffer` (implementada como un monitor utilizando métodos synchronized) que actúa como una cola FIFO concurrente de capacidad acotada. Es decir, bloquea a un consumidor intentando sacar un elemento cuando está vacía y bloquea a un productor intentando agregar un elemento cuando está llena. La capacidad del Buffer debe ser un parámetro configurable.

2. Una clase `Worker` que extiende de `Thread` y realiza la operación deseada. Un `Worker` debe tomar una cantidad de elementos para trabajar de un `Buffer`

conocido al momento de su creación, y una clase una clase `ThreadPool`, que se encarga de instanciar e iniciar la cantidad de `Workers` correspondiente a valor del parámetro `threads`.

3. Cualquier otra clase auxiliar que considere necesaria.

Adicionalmente se requiere un test que ordene con múltiples threads una lista de diez mil números generados de manera aleatoria. La cantidad de threads debe poder ser configurable, de manera tal que sea simple probar distintas variantes. Se pide además determinar empíricamente la cantidad óptima de threads a utilizar al ordenar la lista (considerando el hardware en el que hace el desarrollo).

**Pautas de Entrega**

- El TP se hace en grupos de a lo sumo dos personas (salvo un grupo que puede ser 3 personas). Es posible realizar el trabajo de manera individual, aunque no es recomendable.
- La entrega del TP se realiza por ítems, es decir, en primera instancia se pedirá el ítem 1, en la segunda el 2, y para la tercera el resto del TP. Habrán dos fechas adicionales para hacer reentrega si es necesario. La tabla de fechas de entrega es:

| | |
|---|---|
| Entrega Ítem 1 (Buffer) | 28/06 (Martes) |
| Entrega Ítem 2 (Worker y Threadpool) | 1/06 (Sábado) |
| Entrega TP | 4/06 (Martes) |
| Reentrega 1 | 8/06 (Sábado) |
| Reentrega 2 | 11/06 (Martes) |

- Se debe hacer entrega del código Java que resuelva el enunciado (con todas las clases y paquetes utilizados). Si por alguna razón el armado del entorno requiere hacer algo más que solo la importación, en la entrega deben estar listadas las instrucciones necesarias para configurarlo. Para la prueba empírica, sólo es requerido entregar una documentación (En formato `txt`, `odt`, o `pdf`), explicando las pruebas realizadas y la conclusión alcanzada.
- La entrega se debe hacer por mail a las direcciones de los docentes con el asunto `TP-PCONC-2019S1-Apellido1-Apellido2` (donde `Apellido1` y `Apellido2` son los apellidos de los integrantes del grupo), y se debe adjuntar el código en un archivo `.zip` con nombre:

  `tp-pconc-2019s1-apellido1-apellido2.zip`